Кент Рейсдорф и Кен Хендерсон

BORLAND C++BUILDER. ОСВОЙ САМОСТОЯТЕЛЬНО

Данная книга представляет собой практический курс программирования в C++Builder — новой среде фирмы Borland, предназначенной для визуальной разработки приложений на языке Си++. Книга адресована широкому кругу пользователей — от только начинающих изучать программирование до опытных программистов, работавших в средах Borland C++ и Borland Delphi.

Содержание

Введение	19
1-я неделя. Краткий обзор	21
День 1. Первое знакомство	**23**
Что такое C++Builder?	23
Обзор IDE C++Builder	24
Hello World	26
Hello World, часть II — консольное приложение Win32	28
Обзор языка Си++	33
Истоки Си++	34
Переменные	34
Типы данных Си++	35
Операторы Си++	37
Функции Си++	39
Функция main()	43
Массивы	45
Символьные массивы	47
Заключение	51
Вопросы и задания	52
Вопросы и ответы	52
Вопросы	52
Упражнения	53
День 2. Операторы управления и структуры	**55**
Оператор if	55
Циклы	59
Цикл for	60
Цикл while	62
Цикл do-while	63
Оператор goto	63
Операторы continue и break	64
Оператор switch	65
Области видимости	66
Переменные extern	69
Структуры	69
Массивы структур	71
Заголовки и исходные файлы	71

Пример использования структур 73
Заключение 76
Вопросы и задания 77
Вопросы и ответы 77
Вопросы 77
Упражнения 78
День 3. Указатели, ссылки и функции в Си++ 79
Указатели 79
Локальное и динамическое распределение памяти 80
Динамическое размещение объектов и указатели 81
Назад к нашей программе 82
Разыменование указателей 83
Собираем все вместе 85
Ссылки 87
Передача аргументов функции с использованием ссылок и указателей 89
Ключевое слово const 91
Операторы new и delete 92
Оператор new 92
Оператор delete 93
Решение еще одной загадки 95
new[] и delete[] 95
Функции в Си++ 96
Перегрузка функций 96
Аргументы, передаваемые по умолчанию 98
Функции-члены класса 99
Подставляемые функции 100
Заключение 101
Вопросы и задания 101
Вопросы и ответы 101
Вопросы 102
Упражнения 102
День 4. Классы и объектно-ориентированное программирование 103
Что такое класс? 103
Анатомия класса 104
Уровни доступа к членам класса 104
Конструкторы 106
Списки инициализаторов 108
Деструкторы 109
Члены-данные 110
Функции-члены 112
Что такое this? 113
Один пример 115
Наследование 122

Множественное наследование 124
Базовый файловый ввод-вывод 126
Ввод из файла 127
Вывод в файл 129
Режимы открытия файлов 130
Заключение 131
Вопросы и задания 132
Вопросы и ответы 132
Вопросы 133
Упражнения 133
День 5. Библиотеки классов Си++ и модель визуальных компонентов 135
Краткая история 135
Зачем нужны библиотеки классов? 136
Какой ценой все это дается? 138
Библиотеки классов как пример объектно-ориентированного программирования 138
Сравнение библиотек классов Си++ 139
Библиотека объектов Windows фирмы Borland 139
Библиотека базовых классов фирмы Microsoft 140
Кто же победитель? 141
Библиотека визуальных компонентов 141
Компоненты 142
Свойства, методы и события 142
C++Builder и VCL 154
VCL для программиста на Си++ 155
Структура VCL 157
Классы форм и приложений 158
Классы компонентов 158
Но это еще не все... 162
Заключение 163
Вопросы и задания 163
Вопросы и ответы 164
Вопросы 165
Упражнения 165
День 6. Изучение IDE C++Builder: проекты и формы 167
Проекты в C++Builder 168
Файлы, используемые в проектах C++Builder 168
Модули исходного кода 172
Главное меню и панель инструментов C++Builder 173
Использование палитры компонентов 175
Размещение нескольких копий компонента 175
Размещение компонента в центре формы 176
Контекстное меню палитры компонентов 176

Перемещение по палитре компонентов 176
Приложение, содержащее несколько форм 177
Компиляция, построение и компоновка 179
Компиляция и построение других программ на Си++ 181
Дополнительные сведения о формах C++Builder 182
Формы для главного окна 182
Формы для диалоговых окон 183
Сравнение вторичных и диалоговых окон 188
Модель многодокументного интерфейса 188
Основные свойства форм 188
Методы форм 191
События форм 192
Инспектор объектов 193
Селектор компонентов 194
Вкладка Properties 195
Вкладка Events 196
Пример MDI-программы 197
Шаг 1: Создание формы для главного окна 197
Шаг 2: Написание кода для пунктов меню File | Open и File | Save As 199
Шаг 3: Написание кода для меню Window 201
Шаг 4: Создание подчиненной формы MDI 201
Шаг 5: Создание окна About 202
Заключение 204
Вопросы и задания 205
Вопросы и ответы 205
Вопросы 206
Упражнения 206
День 7. Работа с редактором форм и редактором меню 209
Работа с редактором форм 209
Контекстное меню редактора форм 209
Размещение компонентов 210
Сетка редактора форм 211
Выделение компонентов 212
Перемещение компонентов 216
Защита компонентов от перемещения и изменения размера 217
Упорядочивание компонентов и работа с буфером обмена 218
Изменение размера компонентов 219
Выравнивание компонентов 221
Меню, пожалуйста! 230
Создание главного меню 231
Написание кода 238
Контекстные меню 244
Создание и сохранение шаблонов меню 245

Заключение 246
Вопросы и задания 246
Вопросы и ответы 246
Вопросы 247
Упражнения 247
1-я неделя. Подведение итогов 249
2-я неделя. Краткий обзор 251
День 8. Компоненты VCL 253
Обзор 253
Визуальные и невизуальные компоненты 254
Свойство Name 254
Важные общие свойства 256
Свойство Align 256
Свойство Color 256
Курсоры 258
Свойство Enabled 258
Свойство Font 259
Подсказки 260
Свойства ParentColor, ParentCtl3D, ParentFont и ParentShowHint 261
Свойство Tag 261
Другие общие свойства 262
Основные методы компонентов 262
Основные события 263
Класс TStrings 265
Стандартные элементы управления Windows 266
Элементы редактирования 267
Компоненты ListBox и ComboBox 270
Кнопки 273
Основы работы с кнопками 274
Компонент Label 281
Компонент ScrollBar 282
Компонент Panel 282
И это еще не все... 283
Стандартные диалоговые окна 284
Метод Execute 284
Диалоговые окна File Open и File Save 285
Диалоговое окно Color 287
Диалоговое окно Font 288
Диалоговые окна Find и Replace 288
Заключение 289
Вопросы и задания 289
Вопросы и ответы 289
Вопросы 291

Упражнения 291
День 9. Создание приложений в C++Builder 293
Работа с репозиторием объектов 293
Страницы и опции репозитория объектов 294
Создание новых объектов из репозитория 297
Добавление объектов в репозиторий 298
Добавление проектов в репозиторий 300
Конфигурирование репозитория объектов 300
Создание форм и приложений с помощью мастеров 302
Мастер диалогов 303
Создание приложений с помощью мастера приложений 304
Добавление функций и членов-данных в исходный код 308
Как C++Builder управляет объявлениями классов 309
Включение функций в программу 311
Добавление членов-данных класса 313
Удаление кода, созданного C++Builder 313
Использование файлов ресурсов 314
Заключение 320
Вопросы и задания 321
Вопросы и ответы 321
Вопросы 322
Упражнения 322
День 10. Подробнее о проектах 325
Проекты нужны всем 325
Менеджер проектов 325
Окно менеджера проектов 325
Оперативная панель менеджера проектов 327
Исследование проектов 330
Команды проводника проекта 331
Опции проекта 332
Вкладка Forms 333
Вкладка Application 335
Вкладка C++ 336
Вкладка Pascal 340
Вкладка Linker 340
Вкладка Directories/Conditionals 342
Редактор кода C++Builder 343
Основные операции редактора 344
Специальные возможности редактора 348
Контекстное меню редактора кода 350
Изменение опций редактора 351
Заключение 355
Вопросы и задания 355

Вопросы и ответы 355
Вопросы 356
Упражнения 357
День 11. Работа с отладчиком 359
Для чего нужен отладчик? 360
Отладочные пункты меню 360
Контрольные точки 361
Установка и удаление контрольных точек 362
Окно списка контрольных точек 363
Простые контрольные точки 365
Условные контрольные точки 365
Команда Run to Cursor 366
Наблюдение за переменными 367
Контекстное меню списка объектов наблюдения 367
Диалоговое окно Watch Properties 368
Разрешение и запрещение наблюдения за объектами 369
Добавление переменных в список объектов наблюдения 369
Использование списка объектов наблюдения 370
Инспектор отладки 372
Другие инструменты отладки 375
Evaluate/Modify 375
Просмотр стека вызовов 376
CPU View 377
Команда Go to Address 377
Пошаговое выполнение программы 378
Методика отладки 380
Диагностические макросы: TRACE и WARN 380
Отслеживание GPF 382
Короткие советы по отладке 383
Заключение 384
Вопросы и задания 385
Вопросы и ответы 385
Вопросы 386
Упражнения 387
День 12. Вспомогательные инструменты и опции C++Builder 389
Редактор изображений 389
Цвета переднего плана и фона 390
Инструменты рисования редактора изображений 392
Масштабирование изображения 393
Палитра толщины линии 394
Работа с файлами растровых изображений 395
Редактирование значков 396
Редактирование курсоров 398

Контекстные меню редактора изображений 399
Создание проекта ресурсов 399
WinSight: наблюдение за Windows 401
Система сообщений Windows 402
Дерево окон 403
Окно трассировки сообщений 404
Наблюдение за окном 404
Опции трассировки сообщений 405
Другие возможности WinSight 406
Инструменты командной строки 408
grep.exe 409
implib.exe 410
touch.exe 411
tdump.exe 412
Конфигурирование меню Tools 413
Диалоговое окно Configure Tools 413
Установка опций среды 415
Вкладка Preferences 415
Вкладка Library 417
Вкладка Palette 417
Заключение 418
Вопросы и задания 419
Вопросы и ответы 419
Вопросы 420
Упражнения 421
День 13. После изучения основ 423
Декоративные элементы окна 423
Оперативные панели 423
Строки состояния 429
Панели инструментов 436
Управление доступом к командам 436
Проблемы управления доступом к командам 436
К спасительному Onldle 438
От теории к практике 438
В заключение... 440
Печать в приложениях C++Builder 440
Стандартные диалоговые окна печати 441
Простой способ печати 443
Печать через QuickReport 445
Сложный способ печати 445
Использование курсоров 451
Курсоры 451
Загрузка и использование базовых курсоров 452

Загрузка и использование курсоров пользователя 453
Заключение 454
Вопросы и задания 454
Вопросы и ответы 455
Вопросы 456
Упражнения 456
День 14. Углубленное программирование 459
Реализация контекстно-зависимой справки 459
Создание справочного файла 460
Идентификаторы контекста и свойство HelpContext 461
Реализация контекстно-зависимой справки 461
Использование заголовочных файлов справки 464
Практика 465
Обработка исключений 466
Ключевые слова обработки исключений: try, catch и throw 467
Захват исключений с try и catch 467
Формирование исключений 468
Захват 469
Захват необрабатываемых исключений на уровне приложения 471
Отладка с обработкой исключений 472
Работа с реестром 473
Ключи реестра 474
Типы данных реестра 475
Класс TRegistry 475
Использование TRegistry 477
Специальная обработка сообщений 483
Более подробно о сообщениях Windows 484
Отправление и пересылка 486
Обработка событий 487
Обработка других сообщений Windows 488
Сообщения, определяемые пользователем 493
Заключение 494
Вопросы и задания 494
Вопросы и ответы 494
Вопросы 495
Упражнения 496
2-я неделя. Подведение итогов 497
3-я неделя. Краткий обзор 511
День 15. OLE, ActiveX и OCX 513
Autosrv и Autocon 514
Создание собственного автоматизированного сервера 518
Olemdi и Olesdi 524
Olemdi изнутри 525

Элементы управления OCX 526
Заключение 526
Вопросы и задания 527
Вопросы и ответы 527
Вопросы 527
Упражнения 527
День 16. Архитектура доступа к базам данных в C++Builder 529
Некоторые термины, которые необходимо знать 529
Обзор архитектуры 532
Обзор компонентов C++Builder, обеспечивающих работу с базами данных 534
TSession 535
TDatabase 537
TTable 541
TQuery 549
TStoredProc 554
TBatchMove 558
TDataSource 560
TUpdateSQL 561
TField 563
Заключение 566
Вопросы и задания 566
Вопросы и ответы 566
Вопросы 567
Упражнения 567
День 17. Создание форм для работы с базами данных 569
Простая форма 569
Мастер форм баз данных 569
Изучение созданной формы 573
Создание простой формы сложным способом 575
Запуск новой формы 579
Заключение 580
Вопросы и задания 580
Вопросы и ответы 580
Вопросы 581
Упражнения 581
День 18. Построение приложений для работы с базами данных 583
Компонент TDatabase 583
Псевдонимы BDE 584
Модули данных 586
Создание формы главный/подчиненный 589
Отчеты 593
Заключение 595
Вопросы и задания 595

Вопросы и ответы 595
Вопросы 596
Упражнения 596
День 19. Создание и использование DLL 597
Обзор DLL 597
Что представляет собой динамически компонуемая библиотека? 597
Почему стоит использовать DLL? 598
Вызов функций и использование классов в DLL 601
Загрузка DLL 602
Вызов функций, расположенных в DLL 602
Создание DLL 603
Импорт и экспорт функций и классов 604
Секрет заголовков DLL 607
Создание DLL с помощью репозитория объектов 608
Создание библиотеки импорта 612
Создание вызывающего приложения 613
Добавление #include для заголовка DLL 614
Добавление файла библиотеки импорта 614
Использование форм в DLL 615
Вызов формы DLL из приложения C++Builder 615
Использование ресурсов в DLL 618
Создание DLL ресурсов 618
Использование ресурсов DLL 620
Заключение 621
Вопросы и задания 621
Вопросы и ответы 621
Вопросы 623
Упражнения 623
День 20. Создание компонентов 625
Предварительные сведения 625
Изменение существующих компонентов 627
Создание компонентов заново 627
Регистрация компонента 627
Мастер компонентов 628
Создание компонента 629
Написание свойств 631
Свойства имеют базовые члены-данные. 631
Свойства имеют методы записи 632
Свойства имеют методы чтения 633
Свойства могут использовать прямой доступ 634
Свойства могут быть доступны только для записи или только для чтения 634
Свойства могут иметь значения по умолчанию 635
Свойства могут быть отображаемыми или неотображаемыми 636

Написание методов для компонентов 636
Уровни доступа для компонентов 636
TFlashingLabel: пример компонента 637
Предварительное планирование 642
Тестирование компонента 643
Добавление компонента в библиотеку компонентов 645
Диалоговое окно Install Components 645
Инсталляция компонента FlashingLabel 646
Добавление значка для кнопки компонента 647
Написание событий для компонентов 648
Обзор событий 648
План работы 649
Замещение событий базового класса 653
Собираем все вместе 654
Заключение 659
Вопросы и задания 660
Вопросы и ответы 660
Вопросы 661
Упражнения 661
День 21. C++Builder и Delphi 663
Основные различия между C++Builder и Delphi 663
Другие различия между C++Builder и Delphi 664
Файлы проектов 664
Библиотека компонентов 664
Поддержка нескольких языков программирования 665
Заголовочные файлы 665
Эмуляция элементов Delphi в C++Builder 665
Изменения в компиляторе 665
Новые классы 665
Поддержка компонентов 665
Открытые массивы 666
Среда разработки 666
Доступ к базам данных 667
Заключение 667
Вопросы и задания 667
Вопросы и ответы 667
Вопросы 667
Упражнения 668
3-я неделя. Подведение итогов 669
День 22. Создание приложений для Internet 671
Общие сведения 671
Обзор вкладки Internet палитры компонентов 672
Создание собственного Web-браузера 673

«Оживление» новой формы 675
Заключение 681
Вопросы и задания 681
Вопросы и ответы 681
Вопросы 682
Упражнения 682
День 23. Расширения Си++ в C++Builder 683
Немного истории 683
Расширения языка: великие дебаты 684
Новые ключевые слова Си++ в C++Builder 686
_automated 687
_classid 687
_closure 687
_declspec 687
_int8, _intl6, _int32, _int64 688
_property 688
_published 690
Заключение 691
Вопросы и задания 691
Вопросы и ответы 691
Вопросы 692
Упражнения 692
Приложение А. Ответы на вопросы 693
День 1 693
День 2 693
День 3 694
День 4 694
День 5 695
День 6 695
День 7 696
День 8 696
День 9 697
День 10 697
День 11 697
День 12 698
День 13 698
День 14 699
День 15 699
День 16 699
День 17 700
День 18 700
День 19 700
День 20 701

День 21 701
День 22 702
День 23 702

Краткое содержание

Введение **19**

1-я неделя. Краткий обзор **21**

День 1. Первое знакомство 23

День 2. Операторы управления и структуры 55

День 3. Указатели, ссылки и функции в Си++ 79

День 4. Классы и объектно-ориентированное программирование 103

День 5. Библиотеки классов Си++ и модель визуальных компонентов 135

День 6. Изучение IDE C++Builder: проекты и формы 167

День 7. Работа с редактором форм и редактором меню 209

1-я неделя. Подведение итогов **249**

2-я неделя. Краткий обзор **251**

День 8. Компоненты VCL 253

День 9. Создание приложений в C++Builder 293

День 10. Подробнее о проектах 325

День 11. Работа с отладчиком 359

День 12. Вспомогательные инструменты и опции C++Builder 389

День 13. После изучения основ 423

День 14. Углубленное программирование 459

2-я неделя. Подведение итогов **497**

3-я неделя. Краткий обзор **511**

День 15. OLE, ActiveX и OCX 513

День 16. Архитектура доступа к базам данных в C++Builder 529

День 17. Создание форм для работы с базами данных 569

День 18. Построение приложений для работы с базами данных 583

День 19. Создание и использование DLL 597

День 20. Создание компонентов 625

День 21. C++Builder и Delphi 663

3-я неделя. Подведение итогов **669**

День 22. Создание приложений для Internet 671

День 23. Расширения Си++ в C++Builder 683

Приложение А. Ответы на вопросы **693**

Инсталляция программ с CD-ROM **703**

Благодарности

Я хотел бы выразить благодарность тем людям, которые сделали возможным издание этой книги. Прежде всего, мне хочется сказать спасибо ребятам из Delphi TeamB. Они терпеливо отвечали на огромное количество моих вопросов. Я хочу отдельно поблагодарить Боба Арнсона (Bob Arnson) за помощь. Я обращался к нему всякий раз, когда нужно было быстро получить ответ на какой-либо вопрос.

Разумеется, сотрудники Borland заслуживают отдельных слов благодарности, потому что без C++Builder не было бы и книги. Я хочу поблагодарить Брайана Маерса (Brian Myers), который стимулировал мою работу, когда существовала лишь общая идея этой книги. Спасибо команде разработчиков C++Builder, ответивших для меня на множество вопросов. Спасибо также тем людям из службы технической поддержки Borland, которые были вовлечены в редактирование.

Кроме того, спасибо издателям из Sams Publishing, которые поверили, что я смогу написать эту книгу.

Наконец, я хочу выразить особую признательность моей жене Дженифер. Ей пришлось вытерпеть много недель, когда я, находясь рядом, мысленно полностью отсутствовал. Без ее поддержки и понимания эта книга была бы невозможна.

Кент Рейсдорф

Отдельное спасибо моему товарищу и наставнику Нейлу Кою (Neil Coy), который учил меня искусству программирования; всем замечательным людям из Sams Publishing, особенно Крису Денни (Chris Denny), за вклад в успешное завершение этого проекта; Дугу Тинглеру (Doug Tingler), моему другу, за прекрасное время, проведенное вместе; моим сотрудникам, которым пришлось иметь дело с огромным количеством вопросов, когда этот проект занимал мое время; и, наконец, моим родителям, которые создали мне все условия для написания этой книги и поддерживали меня во время работы.

Кен Хендерсон

Об авторах

Кент Рейсдорф — старший программист в TurboPower Software, а также независимый автор. Он является членом TeamB, добровольной группы оперативной поддержки фирмы Borland. Будучи членом TeamB, Кент каждую неделю по много часов отвечает на вопросы пользователей Borland C++, касающиеся программирования в Windows. Обычно вы можете найти его в различных форумах сети CompuServe. Кент живет в Колорадо Спрингс со своей женой Дженифер и детьми.

Кен Хендерсон — разработчик и администратор баз данных с более чем 10-летним стажем. Он является автором нескольких коммерческих программных пакетов, в том числе вспомогательных средств для повышения эффективности программирования и библиотек программ. Он также частый докладчик на промышленных выставках. В настоящее время Кен работает консультантом по администрированию баз данных и архитектуре клиент/сервер. С ним можно связаться по адресу 74763.2305@compuserve.com.

Введение

Что заставило вас открыть эту книгу? Возможно, вы много лет программировали на Си++ и вас привлекла обещанная в C++Builder быстрая разработка приложений (RAD — rapid application development). Может быть, вы использовали Borland Delphi и хотите усовершенствовать свои знания в среде, основанной на Си++. Не исключено, что ваш босс приказал вам разобраться в этой книге. Или, возможно, вы совсем новичок, который хочет начать изучение увлекательного мира программирования в Windows.

Независимо от того, почему вы здесь, добро пожаловать! Я могу заверить вас, что путешествие будет интересным. Не сомневаюсь, что оно также доставит вам удовольствие. Вам придется поработать, но иногда можно будет и поразвлечься. Нет ничего занимательнее, чем ухватить мелькнувшую мысль и воплотить ее в работающую Windows-программу.

Я призываю вас к самостоятельной работе по мере чтения книги. Отложив книгу в сторону и немного поэкспериментировав, вы можете получить больше, чем даст вам самый лучший учитель. Я также хочу, чтобы вы изучали различные сетевые источники. Форумы Borland в сети CompuServe содержат массу информации. Там вы сможете найти членов Borland TeamB (добровольной группы поддержки) и других пользователей Borland, которые отвечают на вопросы как опытных, так и начинающих пользователей. Не забудьте посетить Web-сайт фирмы Borland (http://www.borland.com). Там вы найдете дополнительную информацию по C++Builder, а также дискуссионную группу, где сможете задать вопросы, касающиеся программирования в C++Builder. (Web-сервер фирмы Borland открыт и в России (http://www.borland.ru). — *Прим. ред.*)

Как работать с этой книгой

Книга серии «Освой самостоятельно», «Borland C++ Builder» содержит 21 главу, как следует из самого названия. Предполагается, что вы будете изучать по одной главе в день. Однако вы можете работать в своем собственном темпе. Если вы рассчитываете проходить в день по две главы и более, желаем успехов! С другой стороны, если вам потребуется более одного дня на какую-то главу, изучайте ее столько времени, сколько нужно.

В конце каждой главы содержится раздел «Вопросы и ответы», составленный по материалу этого дня. Вам также предлагаются контрольные вопросы для самопроверки. Чтобы применить новые знания на практике, в каждую главу включены несколько упражнений. Мы настоятельно рекомендуем не пропускать эти разделы, чтобы лучше закрепить пройденный материал.

Для кого предназначена книга

Несмотря на то, что вам не требуется предварительного знания программирования для чтения этой книги, вы можете обнаружить, что над некоторыми темами приходится проводить больше времени. Это естественно и не должно повергать вас в уныние. Вам придется запастись терпением, чтобы вникнуть во все детали. Конечно, если у вас есть опыт программирования, фрагменты кода будут восприниматься намного легче.

В разработке программ с использованием современных инструментов, основанных на графическом интерфейсе пользователя (GUI), можно выделить два этапа. Первый этап заключается в создании пользовательского интерфейса из визуальных компонентов с помощью техники drag-and-drop. Это довольно просто и напоминает рисование картинок в программе Paintbrush, входящей в Windows. Второй этап более сложен, и здесь пригодится книга типа *Borland C++Builder*. Этот этап включает в себя написание кода на Си++, который объединяет компоненты формы и определяет работу вашей программы. Пока вам не нужно знать об этом коде. Вы научитесь всему необходимому по мере работы с книгой. Спустя очень короткое время вы будете твердо стоять на пути к тому, чтобы стать программистом в Borland C++Builder.

Соглашения, принятые в книге

В книге используется специальное выделение важной информации.



Замечание, которое содержит дополнительные сведения или предлагает другой взгляд на изучаемый предмет, чтобы помочь вам лучше понять основные идеи, скрытые под конкретной реализацией.



Совет, касающийся практического применения только что изученного, или указание на то, как немного повысить эффективность работы.



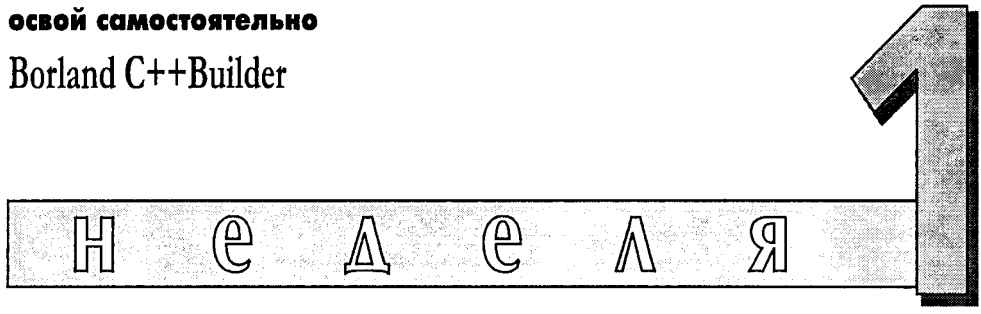
Здесь автор делится с вами своим опытом, чтобы помочь избежать некоторых ошибок.



После этого знака дается определение или вводится новое понятие.



Подробно рассматривается работа программы, представленной в листинге.



Краткий обзор

В первую неделю вы приступите к изучению программирования под Windows на языке Си++. Изучать Си++ нелегко, однако этот язык принят в качестве стандартного во многих корпорациях и правительственных учреждениях по всему миру.

Изучение Си++ — не самая легкая задача, за которую можно было бы взяться, но ваши усилия будут вознаграждены — как морально, так, возможно, и материально.

Первые четыре дня этой недели вы проведете за изучением основ языка Си++. Работая над материалом первых четырех глав, вы будете писать простые программы-примеры, чтобы закрепить понимание отдельных возможностей Си++. Я должен предупредить, что эти программы, вероятно, не будут похожи на те, для создания которых вы приобрели C++Builder.















Программы-примеры в первые четыре дня будут консольными приложениями. Они будут работать так же, как DOS-программы. Они не будут эффектно выглядеть и, вероятно, не произведут на вас сильного впечатления. Однако эти программы помогут вам изучить основы Си++.

Начиная с пятого дня, вы приступите к изучению некоторых вещей, которые делают визуальное программирование с помощью C++Builder мощнейшим инструментом. Мы поговорим о библиотеках классов и о той роли, которую они играют при программировании в Windows. На пятый день вы напишете простую программу, используя инструментарий визуального программирования. После этого мы потратим пару дней на изучение интегрированной среды разработки C++Builder, чтобы вы поняли, как она работает и как ее использование облегчает решение стоящих перед программистом задач.

Начиная с этого момента, изучение Си++ станет более интересным. В конце первой недели вы уже сможете написать несколько работающих Windows-программ.

Итак, двинемся вперед.

Первое знакомство



Кент Рейсдорф

Поздравляю — вы выбрали один из новейших и многообещающих программных инструментов! Однако для того, чтобы начать в полной мере использовать возможности C++Builder, необходимо сперва в общих чертах познакомиться с языком Си++. В этой главе вы найдете:

- Краткий обзор C++Builder
- Сведения о том, как писать консольные приложения Win32
- Введение в язык Си++
- Сведения о переменных и типах данных Си++
- Сведения о функциях Си++ (включая функцию main())
- Сведения о массивах

Что такое C++Builder?

C++Builder — это новый продукт фирмы Borland, предназначенный для быстрой разработки приложений (RAD — rapid application development) на языке Си++. С помощью C++Builder вы можете создавать Windows-программы на Си++ быстрее и проще, чем когда-либо ранее. Вы можете создавать как консольные приложения Win32, так и использовать графический интерфейс пользователя (GUI — graphical user interface). При создании GUI-приложений Win32 с помощью C++Builder вам доступна вся мощь языка Си++, заключенная в среду RAD. Это означает, что вы можете создавать интерфейс пользователя (меню, диалоговые окна, главное окно и т.д.), используя технику drag-and-drop. Вы можете также помещать в формы элементы управления OCX для создания в считанные минуты специализированных приложений, таких как Web-браузеры. При этом вы не жертвуете скоростью выполнения программ, потому что вся мощь языка Си++ по-прежнему остается в вашем распоряжении.

Обращаю ваше внимание еще раз на то, что язык Си++ не является простым даже для профессионалов. Не думайте, что можно купить пакет типа C++Builder и стать профессиональным программистом в Windows за

один день. Чтобы научиться хорошо программировать в Windows, необходимо потратить много труда. В C++Builder многое сделано для того, чтобы скрыть некоторые низкоуровневые детали, которые составляют «внутренности» Windows-программ, но он не может писать программы за вас. В конце концов, вы должны быть программистом, а для этого нужно изучать программирование. Вам придется пройти длинный тернистый путь. C++Builder может сделать ваш путь относительно безболезненным и даже приятным. Да, вы можете работать и получать от этого удовольствие!

Обзор IDE C++Builder

Этот раздел содержит краткий обзор интегрированной среды разработки (IDE — Integrated Development Environment), входящей в состав пакета C++Builder. Мы подробно изучим эту среду на шестой день, «Изучение IDE C++Builder: проекты и формы». Поскольку вы беретесь за программирование в Windows, я полагаю, что вы сможете сами разобраться, как запустить C++Builder. При первом запуске программы на экране будет присутствовать пустая форма и IDE, как показано на рис. 1.1.

Интегрированная среда C++Builder разделена на три части. Верхнее окно является основным. Оно содержит оперативную панель (speedbar) слева и палитру компонентов (Component Palette) справа. Оперативная панель позволяет одним щелчком на соответствующей кнопке выполнять такие действия, как открытие, сохранение и компиляция проектов. Палитра компонентов содержит большой набор компонентов, которые вы можете вставлять в свои формы (к компонентам относятся текстовые метки, окна списка, кнопки и т.п.). Для удобства все компоненты разделены на группы. Вы заметили за-

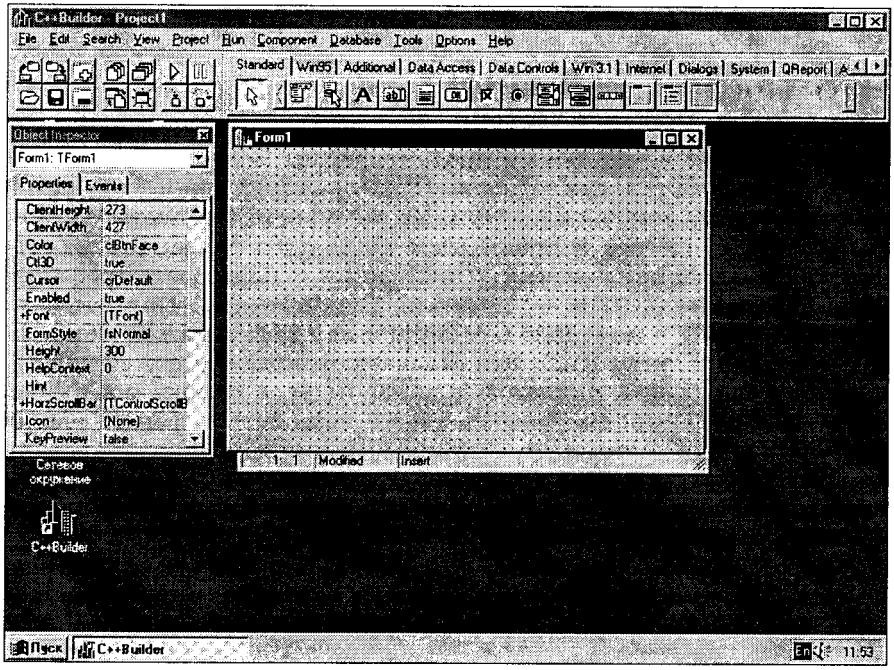


Рис. 1.1 Окно IDE и пустая форма

головки на верхнем краю палитры? Пощелкайте на заголовках вкладок, чтобы изучить доступные компоненты. Для помещения компонента в форму просто щелкните на соответствующей кнопке, а затем щелкните в том месте формы, где он должен появиться. Не беспокойтесь по поводу того, что вы еще ничего не знаете об использовании компонентов. Мы вернемся к этому в свое время. Когда вы закончите исследование, щелкните на вкладке Standard, потому что именно эти компоненты потребуются вам дальше.



Компонент (component) — это функционально законченный участок двоичного кода, выполняющий некоторую специальную предопределенную задачу, например, отображение текста, элемента редактирования или окна списка.

Под оперативной панелью и палитрой компонентов, вплотную к левому краю экрана, расположен инспектор объектов (Object Inspector). С помощью инспектора объектов вы можете изменять свойства компонентов и события, на которые они реагируют. Вы будете использовать инспектор объектов все время, пока работаете с C++Builder. Окно инспектора объектов имеет одну или две вкладки, в зависимости от того, какой компонент выбран. Вкладка Properties (Свойства) присутствует всегда. Свойства объекта определяют его поведение. Например, изменение свойства Color (Цвет) приведет к изменению цвета фона данного объекта. Список доступных свойств изменяется от компонента к компоненту, однако обычно компоненты имеют несколько стандартных свойств (например, Width (Ширина) и Height (Высота).



Свойство (property) определяет поведение компонента.

Обычно инспектор объектов имеет вкладку Events (События) в дополнение к вкладке Properties. События происходят тогда, когда пользователь взаимодействует с компонентом, например щелкает на нем мышью. Вы можете написать код, который будет соответствовать подобным событиям и выполнять определенные действия при их возникновении. Так же как и свойства, события, на которые возможна реакция, различны от компонента к компоненту.



Событие (event) — это метод, вызываемый в компоненте в результате взаимодействия этого компонента с пользователем.

Справа от инспектора объектов расположена рабочая область (workspace). Здесь изначально располагается редактор форм (Form Editor). Как вы уже, вероятно, догадались, редактор форм позволяет создавать формы. Форма может быть главным окном программы, диалоговым окном или окном любого другого типа. В процессе создания формы с помощью редактора форм в ней можно размещать компоненты, изменять их положение и размеры. За редактором форм скрыт редактор кода (Code Editor). В окне редактора кода вы набираете текст программы. Инспектор объектов, редактор форм, редактор кода и палитра компонентов работают взаимосвязанно во время построения приложения.

Теперь, когда вы увидели, из чего состоит IDE C++Builder, давайте что-нибудь сделаем на практике.

Hello World

Это традиция. Почти все учебники по программированию предлагают вам начать с создания программы, выводящей на экран Hello World. Я намеревался было сделать что-нибудь другое, но традиция — это сила, с которой невозможно не считаться, так что Hello World — перед вами. Над материалом следующих нескольких глав придется поработать, поэтому я полагаю, что вам следует почувствовать возможности C++Builder до того, как погрузиться в изучение не таких эффектных на первый взгляд основ языка. C++Builder (как и его родственник, Delphi) предоставляет, пожалуй, самый быстрый способ вывода на экран Hello World по сравнению со всеми существующими на сегодняшний день системами программирования для Windows.

Сейчас у вас должен быть запущен C++Builder и на экране должна находиться пустая форма. По умолчанию ей присваивается имя Form1. (В C++Builder имя формы существенно, но мы обратимся к этому немного позже). Расположенный слева инспектор объектов показывает свойства формы. Щелкните на заголовке инспектора объектов. Свойство Caption (Заголовок) окажется выделено; там же будет располагаться курсор, ожидая ваших действий. (Если свойство Caption отсутствует в поле зрения, прокрутите список в окне инспектора объектов; свойства расположены в алфавитном порядке). Введите Hello World!, чтобы изменить заголовок формы.



Когда вы изменяете свойства объектов, C++Builder сразу отображает результат изменения, если это возможно. Вводя новый заголовок, обратите внимание, что заголовок окна формы изменяется в соответствии с набираемым текстом.

Теперь щелкните на кнопке Run (Запуск), которая расположена на оперативной панели (кнопка с зеленой стрелкой). Вы можете также нажать F9 или выбрать пункт Run | Run в главном меню. C++Builder начнет построение программы. На экране появится окно состояния компилятора, показанное на рис. 1.2, и вы сможете наблюдать, как C++Builder обрабатывает необходимые файлы. Спустя некоторое время окно состояния компилятора исчезнет и на экране отобразится форма с заголовком Hello World!. В данном случае запущенная программа выглядит почти так же, как и пустая форма. Вы можете даже не заметить, что программа заработала, потому что ее окно будет расположено в точности на месте формы в редакторе форм. (Различие состоит в том, что редактор форм отображает сетку для выравнивания компонентов, а работающая программа — нет.) Поздравляю — вы только что написали свою первую программу на Си++ под Windows с помощью C++Builder. Это было совсем не трудно!

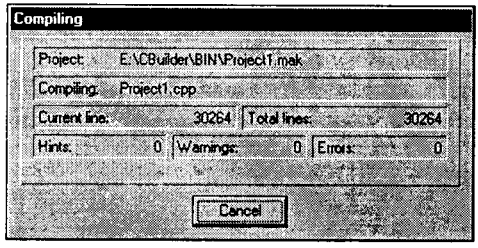


Рис. 1.2
Диалоговое окно состояния компилятора

«Но что это такое?» — спросите вы. Это настоящая Windows-программа, ее окно можно переместить по экрану, зафиксировав курсор мыши на заголовке, сжать, растянуть, свернуть, развернуть и закрыть, щелкнув на кнопке Close.

Ну хорошо, возможно, что написать Hello World! только в заголовке было небольшим мошенничеством. Давайте слегка исправим дело. Если программа Hello World все еще работает, закройте ее, щелкнув на кнопке Close в верхнем правом углу окна. На экране снова появится редактор форм и вы сможете внести изменения в форму (и, следовательно, в программу).

Чтобы сделать программу более реалистичной, мы собираемся добавить текст в центр самого окна. Чтобы сделать это, к форме нужно добавить текстовую метку (text label). Сначала щелкните на вкладке Standard палитры компонентов. На третьей кнопке изображен символ A. Если вы установите курсор мыши на эту кнопку, появится всплывающая подсказка Label. Щелкните на кнопке, а затем в любом месте формы. Компонент Label будет помещен в форму. Теперь обратите внимание на инспектор форм. Он показывает свойства объекта Label1 (вспомните, что ранее он показывал свойства формы Form1). Свойство Caption снова выделено. Щелкните на заголовке инспектора объектов или на свойстве Caption и наберите Hello World!. Теперь метка, размещенная в форме, приняла вид Hello World!. Пока мы находимся в окне инспектора объектов, давайте изменим заодно размер текста. Дважды щелкните на свойстве Font. Под ним появится список дополнительных атрибутов. Выберите атрибут Size и установите размер шрифта равным 24 (текущий размер должен быть 8). После нажатия клавиши Enter или щелчка мышью на форме текст изменит размер.

Поскольку метка скорее всего расположена не по центру формы, вы можете переместить ее. Чтобы изменить положение компонента, просто щелкните на нем и перетащите в нужную позицию. Когда текст размещен желаемым образом, можно заново откомпилировать и запустить программу. Вновь щелкните на кнопке Run. C++Builder начнет компиляцию и спустя некоторое время (в этот раз оно будет меньше) программа запустится. Теперь Hello World! присутствует не только в заголовке окна, но и в его центре. На рис. 1.3 показано окно программы после запуска.

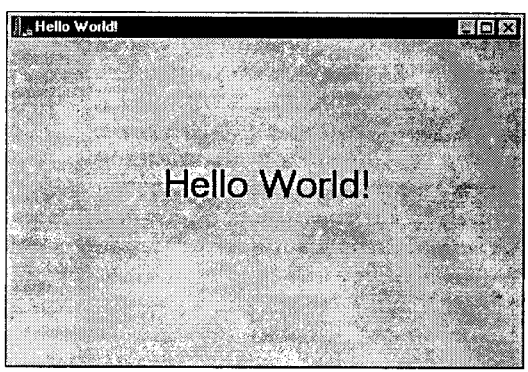


Рис. 1.3
Работающая программа Hello World

На этом простом примере вы могли убедиться, что писать программы с помощью C++Builder намного интереснее, чем в старые добрые времена. Чтобы подготовиться к следующему шагу, вам нужно закрыть текущий проект в IDE. Выберите в главном меню пункт File | Close All. Щелкните на

кнопке No после того, как появится приглашение сохранить изменения в проекте Project1, или сохраните проект под именем HelloWorld, если вы в восторге от вашего нового творения.

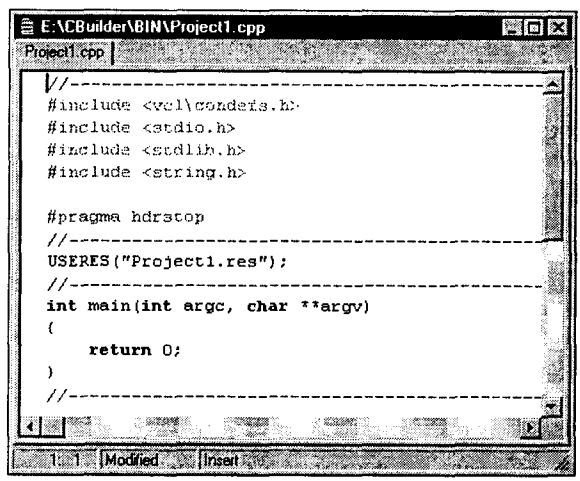
Hello World, часть II — консольное приложение Win32

На протяжении следующих нескольких глав вы будете изучать основы языка Си++. По пути вам предстоит написать несколько простых программ-примеров. Эти программы будут работать лучше, будучи консольными приложениями. При запуске они во всех отношениях выглядят как DOS-программы. Хотя между консольными приложениями Win32 и DOS-программами есть существенные различия, сейчас вам не нужно об этом беспокоиться. Теперь давайте с помощью C++Builder создадим Hello World как консольную программу Win32.

Консольное приложение (console application) Win32 представляет собой 32-разрядную программу, которая запускается в окне DOS под Windows 95 или Windows NT.

Выберите в главном меню пункт File | New. C++Builder откроет репозиторий объектов (object repository). Достаточно неожиданно, что в заголовке репозитория объектов написано New Items (Новые объекты), однако не теряйтесь. Репозиторий объектов содержит предопределенные проекты, формы, диалоговые окна и другие объекты, которые вы можете добавить к существующему приложению или использовать для создания нового проекта. Репозиторий объектов будет детально описан на девятый день, «Создание приложений в C++Builder». А сейчас щелкните на вкладке New (Новый) в репозитории и дважды щелкните на Console App для создания нового консольного приложения. C++Builder создаст проект и вызовет окно редактора кода, после чего вы можете начинать ввод текста программы. На рис. 1.4 показано окно редактора кода перед началом работы над новым консольным приложением.

Рис. 1.4
Окно редактора кода C++Builder



Вы заметите несколько различий между внешним видом IDE C++Builder сейчас и раньше, когда мы создавали GUI-приложение. Прежде всего, отсутствует редактор форм, поскольку консольные приложения не могут отображать формы (вообще говоря, это не совсем верно, но для нашего случая достаточно точно). Также обратите внимание, что окно инспектора объектов пусто. Компоненты можно помещать только в форму, поэтому при работе с консольными приложениями инспектор объектов не используется.



При написании консольных приложений вы можете закрыть инспектор объектов, чтобы освободить больше места для окна редактора кода. Для этого щелкните на кнопке Close, расположенной в строке заголовка окна инспектора объектов. Чтобы снова открыть инспектор объектов, нажмите клавишу F11 или выберите в главном меню пункт View | Object Inspector.

Обратившись к редактору кода, вы увидите, что в его окне находится следующий текст:

```
//---------------------------------
#include <condefs.h>
#include <stdio.h>
#include <stdlib.h>

#pragma hdrstop
//---------------------------------
int main(int argc, char** argv)
{
  return 0;
}
//---------------------------------
```



Это программа на языке Си++, которая ничего не делает, но тем не менее написана правильно. Ее текст можно моментально изменить так, чтобы выполнялись какие-либо действия, но сначала я хочу обратить ваше внимание на строки, начинающиеся с //. Это строки комментариев, которые в данном случае введены только для визуального разделения текста программы. (Обычно строки комментариев используются для документирования текста.) C++Builder автоматически добавляет эти строки при создании нового консольного приложения. (В последующих листингах я буду исключать комментарии для экономии места.) Обратите внимание, что единственный оператор в этой программе оканчивается точкой с запятой. (Кстати, это еще и единственный действительно выполняемый оператор.) Точка с запятой ставится в конце каждого оператора программы, написанной на Си++.

В самом начале изучения Си и Си++ будущий программист должен понять разницу между выражением и оператором. Согласно «официальному» определению, оператор есть «выражение, за которым следует точка с запятой». Точка с запятой закрывает выражение и делает его своего рода блоком кода, состоящим из одной строки. Я скоро расскажу о блоках, но пока вы должны запомнить, что *выражение* — это фрагмент кода, в котором вычисляется некоторое значение. *Оператор* — это выражение, которое закрыто. Например, рассмотрим следующий оператор:

```
c = a + b;
```

В этом примере часть оператора справа от знака равенства, a + b, является выражением. Строка в целом представляет собой оператор. Я знаю, это может поначалу привести в замешательство, но затем, по мере продвижения вперед, все должно проясниться. Я постараюсь быть очень аккуратным при использовании этих двух терминов. Сейчас нужно только запомнить, что оператор заканчивается точкой с запятой и представляет собой закрытое выражение.

Обратите внимание также на открывающую и закрывающую скобки в программе. В Си++ блок кода начинается с открывающей фигурной скобки ({) и заканчивается закрывающей фигурной скобкой (}). Скобки используются для обозначения начала и конца блоков кода, связанных с циклами, функциями, операторами if, а также в других случаях. Данная программа очень проста, и в ней использован только один набор скобок.

Чтобы вывести на экран Hello World!, нам потребуется класс Си++, называемый iostream, поэтому нужно коротко рассказать об этом классе. (Вы еще не знаете о классах, но сейчас об этом не беспокойтесь.) Класс iostream использует *потоки (streams)* для выполнения базового ввода и вывода, например, печати текста на экране или приема информации, вводимой пользователем. Поток cout используется для пересылки данных в стандартный выходной поток. В консольных приложениях стандартный выходной поток означает консоль, т.е. экран. Поток cin используется для приема с консоли данных, вводимых пользователем. В iostream включены два специальных оператора для помещения информации в поток или извлечения ее из потока. *Оператор вставки* (<<) используется для помещения данных в выходной поток, а *оператор извлечения* (>>) — для извлечения данных из входного потока. Для вывода информации на консоль вам следует написать

```
cout << "Do something!";
```

Это является указанием поместить текст Do something! в стандартный выходной поток. При выполнении данной строки программы текст будет выведен на экран.



cout предназначен для использования только в приложениях, работающих в консольном режиме. GUI-приложение Windows не имеет стандартного выходного потока (в GUI-приложениях все основано на графике), поэтому в таком приложении вывод из cout отправляется «в никуда». В стандартных Windows-программах для отображения текста на экране используется функция DrawText() или TextOut(). Программы, написанные с помощью C++Builder, также используют DrawText() или TextOut(), либо через Windows API, либо через класс TCanvas.

Перед тем, как использовать cout, необходимо сообщить компилятору, где находится описание (называемое *объявлением*) класса iostream. Объявление для iostream расположено в файле с именем IOSTREAM.H. Этот файл называется *заголовочным файлом (header file)*.



Заголовочный файл (или, для краткости, *заголовок*) содержит объявления одного или нескольких классов.

Чтобы компилятор искал описание класса iostream в файле IOSTREAM.H, используйте директиву #include:

```
#include <iostream.h>
```



Объявление (declaration) является уведомлением о намерениях. Оно предшествует *определению (definition)* некоторого события. Например, избиратель *объявляет* о своей принадлежности к демократам или республиканцам. Затем своим голосованием на выборах он *определяет*, к какой партии на самом деле принадлежит. В Си и Си++ очень важно различать эти два понятия.

Теперь компилятор сможет найти класс iostream и поймет, что нужно делать, когда ему встретится оператор cout.



Если вы забудете включить заголовочный файл для класса или функции, на которые ссылается ваша программа, компилятор выдаст сообщение об ошибке. Увидев сообщение типа Undefined symbol 'cout' (Не определен символ 'cout'), следует немедленно проверить, все ли необходимые заголовочные файлы включены в программу. Чтобы определить, в каком файле находится объявление класса или функции, щелкните на имени этого класса или функции и нажмите клавишу F1. Это приведет к запуску справочной системы Windows и на экране появится справка по объекту, на котором находится курсор. Где-то в верхней части справочного окна вы увидите ссылку на заголовочный файл, в котором находится объявление функции или класса.

До того, как мы напишем консольную версию Hello World, я упомяну еще об одной вещи. Класс iostream содержит специальные *манипуляторы (manipulators)*, с помощью которых можно управлять процессом вывода. Сейчас мы будем иметь дело только с одним из них, а именно endl (end line — перевод строки), который вставляет символ новой строки в выходной поток. Мы используем endl для перехода на следующую строку после вывода текста на экран.

Теперь, когда вы имеете некоторое представление о классе iostream, мы можем написать программу Hello World как консольное приложение. Отредактируйте текст, чтобы он выглядел как в листинге 1.1. Я пронумеровал строки листинга для последующих ссылок; вы должны пропустить эти номера и пробелы после них при наборе текста.

Листинг 1.1. HELLO.CPP

```
 1: #include <condefs.h>
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4: #include <iostream.h>                  // добавьте эту строку
 5: #pragma hdrstop
 6:
 7: int main(int argc, char** argv)
 8: {
 9:   cout << "Hello World!" << endl; // добавьте эту строку
10:   return 0;
11: }
```



В Си++ пробелы игнорируются. В большинстве случаев не имеет значения, где вы вставите пробел или новую строку. Очевидно только, что нельзя вставлять пробелы в середине ключевых слов или имен переменных; все остальное допустимо. Например, следующие строки кода эквивалентны:

```
int main(int argc, char** argv)
{
cout << "Hello World!";
return 0;
}
```

есть то же самое, что и

```
int main(int argc,char** argv){cout<<"Hello World!";return 0;}
```

Очевидно, что первая запись читается легче и поэтому более предпочтительна. Стили написания программ различаются, но если вы будете следовать соглашениям, принятым в этой книге, то при переходе к программированию реальных задач все будет в порядке.



Щелкните на кнопке Run оперативной панели. Программа будет откомпилирована и запущена. В момент запуска вы увидите окно DOS и слова Hello World!... которые через долю секунды исчезнут. Что же случилось? По достижении конца функции main() программа завершает свою работу и консольное окно сразу же закрывается. Чтобы этого не произошло, мы должны добавить к программе еще несколько строк. Стандартная библиотека Си содержит функцию getch(), предназначенную для ввода символа с клавиатуры. Ее мы и используем, чтобы предотвратить закрытие консольного окна. Вновь отредактируйте текст программы так, чтобы он выглядел в соответствии с листингом 1.2. Можете не добавлять строки комментариев. Не забудьте исключить номера строк.

Листинг 1.2 HELLO.CPP (измененный)

```
 1: #include <condefs.h>
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4: #include <iostream.h>
 5: #include <conio.h>                    // добавьте эту строку
 6: #pragma hdrstop
 7:
 8: int main(int argc, char** argv)
 9: {
10:   cout << " Hello World!" << endl;
11:   // добавьте следующие две строки
12:   cout << endl << "Press any key to continue...";
13:   getch();
14:   return 0;
15: }
```



На этот раз, когда после запуска программы на экране появится Hello World!, консольное окно останется открытым. Для завершения программы и закрытия окна нажмите любую клавишу на клавиатуре.



Тексты программ, приведенные в этой книге, вы можете найти на прилагаемом к ней CD-ROM. Перед компиляцией программы необходимо установить на жесткий диск. Короткие программы поначалу лучше всего вводить вручную, тогда как длинные примеры имеет смысл загружать с жесткого диска, чтобы избежать ошибок при наборе и связанных с ними сообщений компилятора.

Пока это все. Hello World, часть II не слишком воодушевляет, но вы найдете хорошее применение для консольных приложений по мере изучения Си++ на протяжении следующих страниц. Вот почему вам необходимо понимать, как создаются и запускаются консольные приложения. А сейчас давайте перейдем к основам языка Си++.

Обзор языка Си++

Си++ является мощным языком. Он позволяет вам делать то, что невозможно в других языках. Однако это могущество не дается даром. Надо сказать, что Си++ предоставляет вам достаточно возможностей, чтобы «повесить» компьютер — и поначалу это будет случаться довольно часто. Крах программы обычно происходит при выходе за границу отведенной памяти или нарушении прав доступа к системным ресурсам.

Я буду прилагать все усилия, чтобы описать Си++ в пределах небольшого отведенного места. По языку Си++ были написаны многие книги (и большего объема, чем эта!), поэтому не ожидайте, что я смогу свести их все в несколько глав. Я с большой вероятностью полагаю, что прочитав эту книгу и поэкспериментировав некоторое время с C++Builder, вы купите книгу с более детальным описанием Си++.

Си++ позволяет в полном объеме использовать возможности объектно-ориентированного программирования (ООП). ООП — это не просто модное слово. Оно имеет реальные преимущества, позволяя вам создавать объекты, которые можно использовать не только в текущей, но и в будущих программах.



Объект (object), подобно компонентам, описанным ранее, представляет собой участок двоичного кода, выполняющий определенную задачу. (Компоненты являются объектами, но не все объекты — компоненты. Я объясню это позже.)

Объект раскрывает пользователю (программисту, использующему этот объект) только ту часть себя, которая необходима для работы с ним. Все внутренние механизмы, о которых пользователю не нужно знать, скрыты от взгляда. Все это входит в концепцию объектно-ориентированного программирования. ООП позволяет осуществлять модульный подход к программированию, избавляя вас от необходимости постоянно изобретать велосипед. Программы C++Builder являются объектно-ориентированными по причине интенсивного использования объектов. После того, как объект (ваш собственный или встроенный в C++Builder) создан, он может использоваться в любой программе C++Builder. Объект также может быть расширен путем наследования для создания нового объекта с дополнительными свойствами. Самым ценным является то, что объекты скрывают свои внутренние детали и позволяют программисту сконцентрироваться на достижении максимальной эффективности их использования. Объекты и классы Си++ подробно обсуждаются на четвертый день, «Классы и объектно-ориентированное программирование».

Истоки Си++

Си++ построен на основе языка программирования Си и сначала назывался «Си с классами». Эта основа все еще хорошо заметна во многих программах на Си++, которые пишутся сегодня. Си++ был создан не для замены Си, а скорее для его усиления. Остаток этой главы и большая часть следующей будут посвящены той части языка Си++, которая уходит своими корнями в Си. Фактически, мы будем иметь дело с языком Си и перейдем к Си++ позже, на второй день, «Операторы управления и структуры в Си++». Для вас не должно иметь значения, какая часть представленного материала относится к Си, а какая — к Си++, потому что все это включено в один язык, который называется Си++.

Лучше всего было бы представлять язык Си++ последовательно. Однако все возможности языка, которые мы будем обсуждать, пересекаются, поэтому последовательное изложение практически невозможно. Я буду каждый раз брать отдельные аспекты и затем начну соединять их вместе. К концу третьего дня, «Указатели, ссылки и функции в Си++», вы будете иметь достаточно полное представление о Си++. Не беспокойтесь, если вы не сможете понять все до конца. Кое-что из того, что требуется для полного понимания Си++, приходит только с практикой.

Переменные

Поскольку мы должны с чего-то начать, давайте рассмотрим переменные. По существу, *переменная* — это имя, присвоенное некоторому участку памяти. После объявления переменной вы можете использовать ее для операций с данными в памяти. Вероятно, вы не очень почувствовали то, что я сказал, поэтому позвольте привести несколько примеров. В следующем фрагменте кода использованы две переменные. В конце строк приведены комментарии, объясняющие, что происходит при выполнении данной строки:

```
int x;          // объявлена переменная 'x' целого типа
x = 100;        // теперь 'x' содержит значение 100
x += 50;        // теперь 'x' содержит значение 150
int y = 150;    // 'y' объявлена и инициализирована значением 150
x += y;         // теперь 'x' содержит значение 300
x++;            // теперь 'x' содержит значение 301
```



Переменная (variable) — это область памяти компьютера, отведенная для хранения некоторой величины.

Обратите внимание, что значение x изменяется при операциях с переменной. Немного позже я расскажу об операторах, используемых для работы с переменными.



Переменные, которые объявлены, но не инициализированы, содержат случайные значения. Поскольку область памяти, на которую указывает переменная, не была инициализирована, нельзя точно сказать, что там находится. Рассмотрим следующий фрагмент:

```
int x;
int y;
x = y + 10;    // ошибка!
```

В этом примере переменная x может содержать любое значение, потому что переменная y не была инициализирована перед использованием.

Исключением из этого правила являются глобальные переменные, а также переменные, объявленные с модификатором static, которые по умолчанию инициализируются нулем. Все остальные переменные до инициализации или присваивания содержат случайные величины.

Имена переменных могут состоять из букв верхнего и нижнего регистров, цифр и знака подчеркивания (_), но не могут содержать пробелы или другие специальные символы. Имя переменной должно начинаться с буквы или знака подчеркивания. Вообще говоря, не очень хорошо начинать имя переменной со знака подчеркивания, потому что этот знак часто используется компилятором в начале имен специальных переменных и функций. Максимально допустимая длина имени переменной различается от компилятора к компилятору, но если вы будете использовать имена, состоящие не более чем из 31 символа, все будет в порядке. Практически не имеет смысла делать имена длиннее 20 символов.

Типы данных Си++



В Си++ *тип данных (data type)* определяет способ хранения информации в памяти.

В некоторых языках программирования вы можете присвоить переменной любое значение. Например, рассмотрим фрагмент программы на Бейсике:

```
x = -1;
x = 1000;
x = 3.14;
x = 457000;
```

Транслятор Бейсика отводит под переменную достаточное количество памяти, чтобы разместить там число любой величины и типа. Однако В Си++ вы должны объявить тип переменной перед ее использованием:

```
int x1 = -1;
int x = 1000;
float y = 3.14;
long z = 457000;
```

Это позволяет компилятору выполнить проверку соответствия типов и гарантировать отсутствие соответствующих ошибок при выполнении программы. При неправильном использовании типов данных компилятор выдаст сообщение об ошибке или предупреждение, с помощью которого можно проанализировать и исправить ситуацию до запуска программы. Некоторые типы данных могут иметь знаковую и беззнаковую разновидность. Переменные знакового типа могут содержать как положительные, так и отрицательные числа, тогда как беззнаковые — только положительные. В таблице 1.1 приведены базовые типы данных Си++, их размеры и диапазоны допустимых значений.

Таблица 1.1. Типы данных, используемые в Си++ (32-разрядные программы)

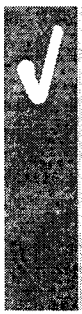
Тип данных	Размер в байтах	Диапазон допустимых значений
char	1	от -128 до 127
unsigned char	1	от 0 до 255
short	2	от -32 768 до 32 767
unsigned short	2	от 0 до 65 535
long	4	от -2 147 483 648 до 2 147 483 647
unsigned long	4	от 0 до 4 294 967 295
int	4	совпадает с long
unsigned int	4	совпадает с unsigned long
float	4	от 1.2E-38 до 3.4E38
double	8	от 2.2E-308 до 1.8E308
bool	1	true или false



В C++Builder (как и в Borland C++ 5.0), bool является самостоятельным типом данных. Некоторые компиляторы Си++ распознают ключевое слово BOOL, но не рассматривают его как тип данных. BOOL здесь определяется с помощью typedef как эквивалент типа int. Оператор typedef устанавливает соответствие между двумя символами, после чего компилятор рассматривает их как эквивалентные. Пример использования typedef:

```
typedef int BOOL;
```

Это является указанием компилятору считать BOOL другим словом для обозначения int.



Только типы double и float предназначены для представления чисел с плавающей точкой (чисел с десятичной дробной частью). Остальные типы являются целочисленными. Хотя присваивание десятичного значения переменной целого типа вполне законно, дробная часть при этом будет отброшена. Например,

```
int x = 3.75;
```

приведет к тому, что x будет содержать значение 3. Обратите внимание, что при этом не производится округления до ближайшего целого числа; вместо этого дробная часть отбрасывается целиком. Кстати, в большинстве Windows-программ числа с плавающей точкой требуются на удивление редко.

Си++ автоматически выполняет преобразования между различными типами данных, когда это возможно. Рассмотрим следующий фрагмент кода в качестве примера:

```
short result;
long num1 = 200;
long num2 = 200;
result = num1 * num2;
```

В этом примере я попытался присвоить результат умножения двух целых чисел типа long целой переменной типа short. Несмотря на то, что в этой формуле смешаны два типа данных, Си++ способен выполнить преобразование. Хотите узнать, что получилось? Наверное, вы будете удивлены, обнаружив в

результате -25 536. Это следствие *заворачивания (wrapping)*. Обратившись к таблице 1.1, Вы увидите, что short может иметь максимальное значение 32 767. Что произойдет, если вы возьмете переменную типа short со значением 32 767 и прибавите к ней 1? Результатом будет -32 768. Здесь происходит то же самое, что и в одометре автомобиля, когда он сбрасывается с 99 999 в 00 000 на последней миле. Для иллюстрации вышесказанного, введите и запустите программу, приведенную в листинге 1.3.

Листинг 1.3. WRAPME.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4:
 5: int main(int argc, char** argv)
 6: {
 7:   short x = 32767;
 8:   cout << "x = " << x << endl;
 9:   x++;
10:   cout << "x = " << x << endl;
11:   getch();
12:   return 0;
13: }
```

Результатом ее выполнения будет:

```
x = 32767;
x = -32768;
```

Вы не слишком ошибетесь, если выберете в качестве основного типа int. Вряд ли при этом возникнут проблемы с заворачиванием, потому что int позволяет работать в диапазоне примерно от -2 миллиардов до +2 миллиардов. Однако ваши программы немного увеличатся в размере из-за использования большего количества памяти, чем обычно требуется.

Так, о чем я говорил? Ах, да, об автоматическом преобразовании типов. В отдельных случаях Си++ не может выполнить преобразование. Тогда вы получите одно из нескольких возможных сообщений об ошибке, которое будет выглядеть как Cannot convert from X to Y (Невозможно преобразовать X в Y). Вы можете также получить предупреждение Conversion may lose significant digits (При преобразовании возможна потеря значащих цифр).



Приучайте себя рассматривать предупреждения компилятора как ошибки, поскольку компилятор пытается сообщить вам о том, что не все в порядке. В идеале вам следует добиваться полного отсутствия предупреждений. В некоторых случаях предупреждение нельзя игнорировать, поэтому убедитесь, что вы тщательно разобрались со всеми предупреждениями. Приложите все усилия, чтобы понять причину, по которой возникло предупреждение, и устраните ее, если это возможно.

Операторы Си++

Операторы (operators) используются для обработки данных. Операторы выполняют вычисления, сравнения, присваивания, а также много других специфических задач, в которые большая часть программистов никогда не вникает. В Си++ существует большое количество операторов. Я приведу здесь только те, которые наиболее часто используются. Они сведены в таблицу 1.2.

Таблица 1.2. Обычно используемые операторы Си++

Оператор	Описание	Пример
	Арифметические операторы	
+	Сложение	x = x + z;
–	Вычитание	x = y - z;
*	Умножение	x = y * x;
/	Деление	x = y / z;
	Операторы присваивания	
=	Присваивание	x = 10;
+=	Сложение с присваиванием	x += 10; (то же, что и x = x + 10;)
-=	Вычитание с присваиванием	x -= 10;
*=	Умножение с присваиванием	x *= 10;
/=	Деление с присваиванием	x /= 10;
&=	Поразрядное И с присваиванием	x &= 10;
|=	Поразрядное ИЛИ с присваиванием	x |= 10;
	Логические операторы	
&&	Логическое И	if (x && 0xFF) {...}
||	Логическое ИЛИ	if (x || 0xFF) {...}
	Операторы отношения	
==	Равно	if (x == 10) {...}
!=	Не равно	if (x != 10) {...}
<	Меньше	if (x < 10) {...}
>	Больше	if (x > 10) {...}
<=	Меньше или равно	if (x <= 10) {...}
>=	Больше или равно	if (x >= 10) {...}
	Унарные операторы	
*	Косвенная адресация (размывание)	int x = *y;
&	Взятие адреса	int* x = &y;
~	Поразрядное НЕ	x &= ~0x02;
!	Логическое НЕ	if (!valid) {...}
+	Инкремент	x++; (то же, что и x = x + 1;)
--	Декремент	x--;
	Операторы классов и структур	
::	Разрешение области видимости	MyClass::SomeFunction();
->	Косвенный доступ	MyClass->SomeFunction();
.	Прямой доступ	MyClass.SomeFunction();

Как видите, список операторов слегка ошеломляет, но не пытайтесь сейчас запомнить каждый из них. Работая с Си++, вы постепенно научитесь использовать все эти операторы.

Следует заметить, что в некоторых случаях операторы могут быть как преинкрементными (++x), так и постинкрементными (x++). *Преинкрементный (pre-increment)* оператор указывает компилятору: «Увеличить значение переменной и затем использовать эту переменную». *Постинкрементный (post-increment)* оператор указывает компилятору: «Сначала использовать переменную, а затем увеличить ее значение». Например, результатом выполнения фрагмента

```
int x = 10;
cout << "x = " << x++ << endl;
cout << "x = " << x << endl;
cout << "x = " << ++x << endl;
cout << "x = " << x << endl;
```

будет

```
x = 10;
x = 11;
x = 12;
x = 12;
```

Многое из этого вы не почувствуете, пока не поработаете с Си++ некоторое время. Однако будьте терпеливы, и понимание в конце концов придет к вам.



В Си++ операторы могут быть перегружены (overloaded). Это означает, что программист может взять один из стандартных операторов и сделать так, чтобы он выполнял определенную операцию над определенными классами. Например, вы можете перегрузить оператор ++ для одного из ваших классов так, чтобы он увеличивал значение переменной на 10, а не на 1. Перегрузка операторов относится к технике профессионального программирования на Си++ и не будет подробно рассматриваться в рамках этой книги.

Вы заметите, что для некоторых операторов используются одинаковые символы. Значение символа зависит от контекста. Например, звездочка (*) может использоваться для умножения, объявления указателя или его разыменования. Поначалу нетрудно запутаться, но, по правде говоря, путаница время от времени возникает независимо от того, как долго вы программируете на Си++.

Вы будете встречать много примеров использования перечисленных операторов по мере прочтения книги. Вместо того, чтобы напрямую запоминать функцию каждого из них, попытайтесь сделать это посредством тщательного изучения примеров программ и фрагментов кода.

Функции Си++

Функции представляют собой блоки кода, не входящие непосредственно в основную программу. Они вызываются (выполняются) по мере необходимости для выполнения специальных действий в программе. Например, вы можете написать функцию, которая берет два числа, выполняет сложную математическую операцию над этими числами и возвращает результат. Вам может также потребоваться функция, которая берет строку, анализирует ее и возвращает часть строки.



Функция (function) — это блок кода, не входящий в основную программу, который выполняет одну определенную задачу.

Функции являются существенной частью любого языка программирования, и Си++ — не исключение. Функция простейшего типа не принимает никаких аргументов и возвращает значение типа void (это означает, что на самом деле она ничего не возвращает). Другие функции могут принимать один или более аргументов и возвращать некоторое значение. Имена функций образуются по тем же правилам, что и имена переменных. Структура функции показана на рис. 1.5.



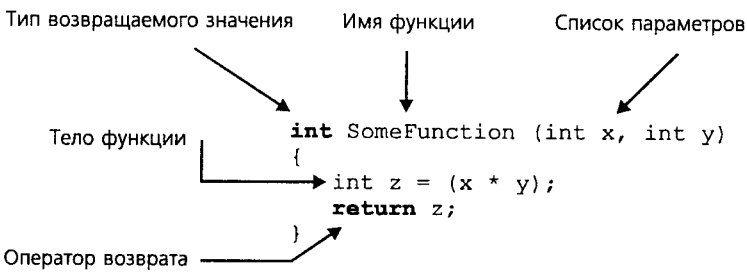
Аргумент (parameter) — это передаваемая функции величина, которая используется для изменения работы функции или указания объекта, над которым выполняются действия.

До использования функции ее необходимо объявить. *Объявление функции (function declaration)*, или *прототип (prototype)*, сообщает компилятору, сколько аргументов принимает функция, тип каждого аргумента и тип возвращаемого значения. В листинге 1.4 приведен пример использования функций.



Прототип (prototype) — это объявление функции, или ее предварительное определение.

Рис.1.5
Структура функции



Листинг 1.4. MULTIPLY.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4:
 5: int multiply(int, int)
 6: void showResult(int)
 7:
 8: int main(int argc, char** argv)
 9: {
10:   int x, y, result;
11:   cout << endl << "Enter the first value: ";
12:   cin >> x;
13:   cout << "Enter the second value: ";
14:   cin >> y;
15:   result = multiply(x, y);
16:   showResult(result);
17:   cout << endl << endl << "Press any key to continue...";
18:   getch();
19:   return 0;
```

```
20: }
21:
22: int multiply(int x, int y)
23: {
24:   return x * y;
25: }
26:
27: void showResult(int res)
28: {
29:     cout << "The result is: " << result << endl;
30: }
```

Эта программа запрашивает у пользователя два числа (используя стандартный входной поток, cin) в строках 11 — 14, вызывает функцию multiply() для перемножения этих чисел (строка 15) и затем вызывает функцию showResult() для вывода результата (строка 16). Обратите внимание на прототипы функций multiply() и showResult() в строках 5 и 6 перед началом основной программы. В прототипах указаны только тип возвращаемого значения, имя функции и типы аргументов. Это минимальные требования к объявлениям функций.

При желании в прототипе можно указать имена переменных, чтобы пояснить те действия, которые выполняет функция. Например, объявление для функции multiply() может быть написано следующим образом:

```
int multiply(int firstNumber, int secondNumber);
```

В этом случае и так достаточно очевидно, что делает функция multiply(), но никогда не помешает задокументировать текст программы как через комментарии, так и через сам код.

Взгляните снова на листинг 1.4. Обратите внимание, что определение функции multiply() (строки 22 — 25) находится вне блока кода, определяющего функцию main() (строки 8 — 20). Определение функции содержит тело функции. В нашем примере тело функции имеет минимальный размер, поскольку функция просто перемножает два своих аргумента и возвращает результат.

Функция multiply() из листинга 1.4 может быть вызвана несколькими способами. Вы можете передать переменные, константы или даже результаты вызова другой функции:

```
result = multiply(2, 5);        // передача констант
result = multiply(x, y);        // передача переменных
showResult(multiply(x, y));     // возвращаемое значение используется
                                // в качестве аргумента другой функции
multiply(x, y);                 // возвращаемое значение игнорируется
```

Обратите внимание на пример, в котором значение, возвращаемое функцией, не используется. В данном случае не имеет большого смысла вызывать функцию multiply() и затем игнорировать результат, но такие вызовы часто применяются при программировании на Си++. Существует много функций, которые выполняют определенные действия и затем возвращают код завершения операции. Иногда возвращаемое значение не имеет отношения к вашей программе, и вы вполне можете его игнорировать. Если вы никак не используете возвращаемое значение, оно просто отбрасывается без всяких последствий. Например, мы игнорируем результат выполнения функции getch() (которая возвращает ASCII-код нажатой клавиши) в наших программах-примерах.

Функции могут (часто так и делается) вызывать другие функции. Функция может даже вызывать сама себя. Это называется рекурсией и является одним из потенциальных источников проблем при программировании на Си++. Лучше всего не использовать рекурсию, пока вы не приобретете некоторый опыт работы с Си++.



Рекурсия (recursion) — это процесс, при котором функция вызывает сама себя.

Сведения, приведенные в этом разделе, относятся к *автономным (standalone)* функциям в программах на Си и Си++ (т.е. функциям, не являющимися членами классов). Автономные функции могут использоваться в Си++ точно так же, как и в Си. Однако Си++ предоставляет больше возможностей. Я не буду сейчас касаться этой темы и вернусь к ней после более глубокого знакомства с Си++.

Правила работы с функциями

- Функция может принимать любое количество аргументов или не иметь аргументов вообще.
- Функция может возвращать значение, но это не является обязательным.
- Если для возвращаемого значения указан тип void, функция не возвращает никакого значения. Если вы попытаетесь возвратить значение в такой функции, компилятор выдаст сообщение об ошибке. Функция, возвращающая void, может не содержать оператора return, но при желании его можно оставить. Допустим любой вариант. Если оператор return отсутствует, возврат осуществляется автоматически по достижении конца функции (закрывающей фигурной скобки).
- Если в объявлении функции указано, что она возвращает значение, в теле функции должен содержаться оператор return, возвращающий это значение. В противном случае компилятор выдаст предупреждение.
- Функции могут иметь любое количество аргументов, но возвращаемое значение всегда одно.
- Аргументы могут передаваться функции по значению, через указатели или по ссылке. (Мы обсудим это немного позже.)

Синтаксис

Формат объявления (прототипа) функции:

```
ret_type function_name(argtype_1 arg_1, argtype_2 arg_2, ..., argtype_n arg_n);
```

Объявление идентифицирует функцию, которая будет использована в программе. В нем указывается тип значения, возвращаемого функцией (ret_type), имя функции (function_name), определяются порядок (arg_1, arg_2, ..., arg_n) и типы (argtype_1, argtype_2, ..., argtype_n) аргументов, ожидаемых функцией.

Формат определения функции:

```
ret_type function_name(argtype_1 arg_1, argtype_2 arg_2, ..., argtype_n arg_n)
{
statements;
}
```

Определение функции содержит блок кода (statements), который составляет функцию. В определении также указываются: тип возвращаемого значения (ret_type), имя функции (function_name), идентифицирующее функцию, параметры функции (arg_1, arg_2, ..., arg_n) и их типы (argtype_1, argtype_2, ..., argtype_n).

Функция main()

Программа на Си++ должна иметь функцию main(), которая служит точкой входа в программу. Вы видели это во всех предыдущих примерах. Однако не во всех Си++ программах используется традиционная функция main(). Для Windows-программ, написанных на Си и Си++, точкой входа чаще является функция с названием WinMain().

GUI-приложения C++Builder содержат функцию WinMain(), однако она скрыта от вас. C++Builder освобождает вас от необходимости заботиться о низкоуровневых деталях Windows-программ и позволяет сконцентрироваться на создании интерфейса пользователя и остальной части программы.

main() является такой же функцией, как и любая другая. Это означает, что она имеет ту же базовую структуру. Вы уже видели, что для 32-разрядных консольных приложений C++Builder по умолчанию создает функцию main() со следующим прототипом:

```
int main(int argc, char** argv);
```

В такой форме main() принимает два аргумента и возвращает целое число. Как вы узнали ранее, аргументы передаются функции при ее вызове. Но функцию main() вы никогда не вызываете непосредственно — это происходит автоматически при запуске программы. Так как же функция main() получает свои аргументы? Вот ответ: из командной строки. Позвольте мне проиллюстрировать сказанное.

Предположим, что у вас есть консольное приложение Win32, запускаемое из приглашения DOS следующей командной строкой:

```
grep WM_KILLFOCUS -d -i
```

В этом случае вы запускаете программу под названием grep с аргументами WM_KILLFOCUS, -d и -i. Сейчас я покажу, как они переводятся в argc и argv внутри функции main(). Прежде всего, целая переменная argc будет содержать число параметров, переданных в командной строке. Это число всегда не меньше 1, т.к. имя программы также считается параметром. Переменная argv является массивом указателей на строки. Этот массив будет содержать каждый из параметров, переданных в командной строке. В данном примере переменные получат следующие значения:

```
argc      содержит 4
argv[0]   содержит c:\bc5\bin\grep.com
argv[1]   содержит WM_KILLFOCUS
argv[2]   содержит -d
argv[3]   содержит -i
```

Давайте проверим, как все это работает, с помощью небольшой программы. Создайте новое консольное приложение C++Builder и введите программу, приведенную в листинге 1.5.

Листинг 1.5. ARGSTEST.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4:
 5: int main(int argc, char** argv)
 6: {
 7:   cout << "argc = " << argc << endl;
 8:   for (int i = 0; i < argc; i++)
 9:     cout << "Parameter " << i << ": " << argv[i] << endl;
10:   getch();
11:   return 0;
12: }
```

Сохраните проект как ARGSTEST.CPP. Вместо того, чтобы щелкнуть на кнопке Run, выберите в главном меню пункт Project | Build All. Программа будет построена, но не запущена. Когда построение программы закончится, выберите в главном меню пункт Run | Parameters. Наберите в открывшемся диалоговом окне:

```
one two three "four five" six
```

Теперь щелкните на кнопке Run, после чего программа запустится, используя заданные параметры командной строки. Можно также запустить программу из приглашения MS-DOS следующей командной строкой:

```
argtest one two three "four five" six
```

Когда программа запустится, она выведет на экран количество переданных аргументов и список этих аргументов. Результат работы программы должен соответствовать рис. 1.6. Запустите программу несколько раз, задавая в командной строке различные аргументы, и понаблюдайте за результатами.



Рис. 1.6
Результат работы ARGTEST.EXE

В большинстве программ значение, возвращаемое функцией main(), несущественно, т.к. оно обычно не используется. Фактически, вашей функции main() вообще не нужно возвращать никакого значения. Существует несколько форм записи для main(). Все приведенные ниже объявления правильны:

```
main();
int main();
int main(int argc, char** argv);
void main();  // то же, что и первое сверху
void main(int argc, char** argv);
```

Не знаю, поверите вы или нет, но существует даже больше возможностей, чем перечислено здесь. Если вы не собираетесь использовать параметры командной строки и возвращать какое-либо значение, можно ограничиться первой формой записи. В этом случае возвращается void и не используются никакие параметры (что подтверждается пустыми скобками). Другими словами, в самом основном варианте функция main() не принимает аргументов и не возвращает никакого значения.

Массивы

Вы можете организовать данные любого встроенного типа Си++ в массивы. *Массив (array)* — это просто набор данных. Например, вы хотите иметь целочисленный массив для хранения пяти чисел типа int. Для этого вам нужно написать следующее объявление:

```
int myArray[5];
```

Компилятор выделит память для массива, как показано на рис. 1.7. Поскольку каждое число типа int требует 4 байта памяти, весь массив займет 20 байт.

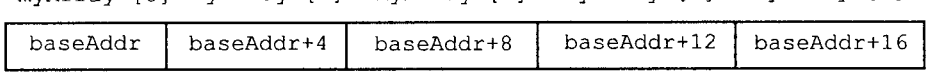


Рис. 1.7
Распределение памяти для массива из пяти чисел типа int

После объявления массива вы можете заполнить его числами, используя *индексный оператор* ([]):

```
myArray[0] = -200;
myArray[1] = -100;
myArray[2] = 0;
myArray[3] = 100;
myArray[4] = 200;
```

Доступ к отдельным элементам массива осуществляется также с помощью индексного оператора:

```
int result = myArray[3] + myArray[4];  // result примет значение 300
```

Массив можно объявить и заполнить одновременно. Это выглядит следующим образом:

```
int myArray[5] = { -200, -100, 0, 100, 200 };
```

Более того, если вы точно знаете, сколько элементов должно быть в массиве, и целиком заполните массив при его объявлении, можно даже опустить размер массива:

```
int myArray[] = { -200, -100, 0, 100, 200 };
```

В этом случае компилятор определит количество элементов в массиве по списку значений и выделит для размещения массива соответствующее количество памяти.

Массивы могут иметь более одного измерения. Для создания двумерного массива целых чисел нужно использовать следующую запись:

```
int mdArray[3][5];
```

При этом выделяется память для хранения 15 целых чисел (всего 60 байт). Вы можете получить доступ к элементам этого массива так же, как и для простого массива, с той лишь очевидной разницей, что нужно использовать два индексных оператора:

```
int x = mdArray[1][1] + mdArray[2][1];
```

На рис. 1.8 показано, как двумерный массив размещается в памяти.

	mdArray[][0]	mdArray[][1]	mdArray[][2]	mdArray[][3]	mdArray[][4]
mdArray[0][]	baseAddr	baseAddr+4	baseAddr+8	baseAddr+12	baseAddr+16
mdArray[1][]	baseAddr+20	baseAddr+24	baseAddr+28	baseAddr+32	baseAddr+36
mdArray[2][]	baseAddr+40	baseAddr+44	baseAddr+48	baseAddr+52	baseAddr+56

Рис. 1.8
Размещение двумерного массива в памяти



Вы должны соблюдать осторожность, чтобы не выйти за верхнюю границу массива. Одним из мощных средств Си++ является прямой доступ к памяти. В силу этого Си++ не запрещает запись в любой участок памяти, даже если эту память не предполагалось использовать в процессе выполнения программы. Следующий код вполне допустим, но он приведет к краху вашей программы (или Windows):

```
int array[5];
array[5] = 10;
```

Это распространенная ошибка: вы полагаете, что последний элемент массива 5, тогда как на самом деле это 4. При выходе за границу массива вы не сможете узнать, какой участок памяти был изменен, и в лучшем случае результат будет непредсказуемым. В худшем же случае произойдет крах программы, а возможно даже и Windows. Ошибку такого типа бывает трудно обнаружить, поскольку часто к измененному участку памяти долгое время не происходит обращения, и крах наступает гораздо позже (оставляя вас недоумевать, что же случилось). Будьте осторожны при записи в массив.

Правила работы с массивами

- Базовый элемент массива имеет индекс 0. Индекс второго элемента — 1, третьего — 2, и т.д.
- Размеры массивов должны быть константами, определенными на этапе компиляции. Во время построения программы компилятор должен знать, сколько памяти выделить для размещения массива. Вы не можете использовать переменные для хранения размера массива, поэтому следующий фрагмент приведет к выдаче сообщения об ошибке:

```
int x = 10;
int myArray[x]    // здесь произойдет ошибка компиляции
```

- Соблюдайте осторожность, чтобы не выйти за верхнюю границу массива.
- Размещайте большие массивы в «куче» (heap), а не в стеке. (Вы узнаете об этом позже.)

Символьные массивы

Это может показаться странным, но в Си++ отсутствует поддержка строковых переменных (переменных, содержащих текст). Для представления строк в программах на Си++ используются массивы переменных типа char. Например, вы можете выполнить следующее присваивание:

```
char text[] = "This is a string.";
```

Это приведет к выделению 18 байт памяти для хранения строки. Однако, как вы могли заметить, в данной строке всего 17 символов. Причина выделения 18 байт состоит в том, что каждая строка завершается нуль-символом, и Си++ учитывает его при размещении массива в памяти.

Нуль-символ (terminating null) — это специальный символ, представляемый в программах как \0, эквивалентный числу 0.

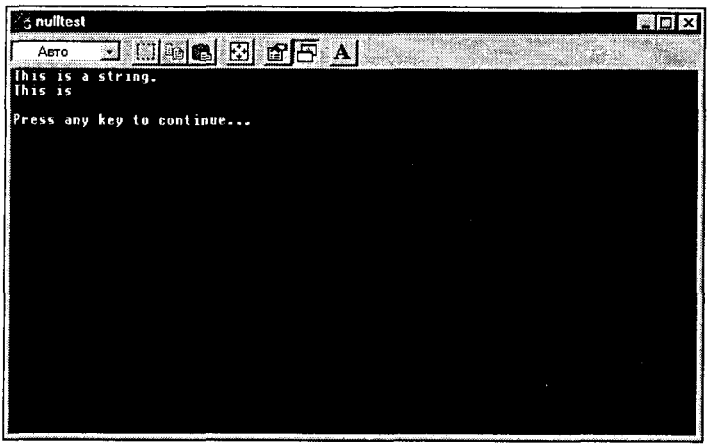
Когда программа встречает в символьном массиве 0, она интерпретирует его как конец строки. Чтобы увидеть, как это происходит, введите и запустите программу из листинга 1.6 как консольное приложение.

Листинг 1.6. NULLTEST.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4:
 5: int main(int argc, char** argv)
 6: {
 7:   char str[] = "This is a string.";
 8:   cout << str << endl;
 9:   str[7] = '\0';
10:   cout << str << endl;
11:   getch();
12:   return 0;
13: }
```

На рис. 1.9 показан результат работы программы из листинга 1.6.

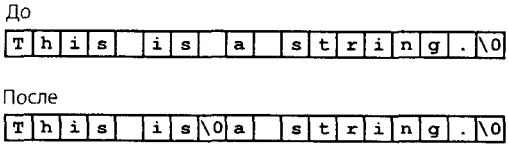
Рис. 1.9
Результат работы программы NULLTEST.CPP





Изначально символьный массив содержит строку This is a string, оканчивающуюся нуль-символом. Эта строка посылается на экран через cout. В следующей строке седьмому элементу массива присваивается '\0', т.е. нуль-символ. Строка снова посылается на экран, однако на этот раз выводится только This is. Причина в том, что компьютер считает концом строки седьмой элемент массива. Остальные символы по-прежнему хранятся в памяти, но не выводятся. На рис. 1.10 показано, как выглядит символьный массив до и после замены седьмого элемента на нуль-символ.

Рис. 1.10
Содержимое символьного массива



Я мог бы просто присвоить 0 вместо '\0' в листинге 1.6. Допустим любой вариант, т.к. число 0 и '\0' типа char эквивалентны.



В программах на Си++ существует различие между одиночными и двойными кавычками. Присваивая нуль-символ (или любое другое символьное значение) элементу массива, вы должны использовать одиночные кавычки. Символ в кавычках заменяется целым числом (ASCII-кодом этого символа), которое затем и хранится в памяти. Когда символьному массиву присваивается строка, необходимо использовать двойные кавычки. Если вы используете кавычки неправильно, компилятор выдаст сообщение об ошибке.

Функции для работы со строками

Если вы до этого изучали язык программирования, в котором есть тип данных string, работа со строками в Си++ может показаться довольно утомительной. На самом деле требуется не так много времени, чтобы привыкнуть к новым правилам. Для облегчения работы со строками в стандартной библиотеке Си есть несколько специальных функций. В таблице 1.3 перечислены

наиболее часто используемые функции и приведены их краткие описания. Полные описания каждой из этих функций и примеры их использования содержатся в справочной системе C++Builder.

Таблица 1.3. Функции для работы со строками

Функция	Описание
strcat()	Добавляет строку в конец строки назначения
strcmp()	Сравнивает две строки
strcmpi()	Сравнивает две строки без учета регистра
strcpy()	Копирует содержимое одной строки в другую
strstr()	Осуществляет поиск первого вхождения подстроки в строку
strlen()	Возвращает длину строки
strupr()	Преобразует все символы строки к верхнему регистру
sprintf()	Формирует строку на основе переменного числа параметров



Операции со строками, обсуждаемые здесь, соответствуют представлению строк в Си. Большинство компиляторов Си++ поставляется с классом cstring, использование которого упрощает работу со строками. (Библиотека визуальных компонентов, входящая в C++Builder, содержит класс Strings, поддерживающий строковые операции. Дополнительная информация о классе Strings содержится во встроенной справочной системе C++Builder.) Хотя поддержка строк в Си немного экзотична, она ни в коем случае не устарела. Программисты на Си++ ежедневно работают со строками как в стиле Си, так и используя строковые классы типа cstring.

Я не буду приводить примеры использования всех функций, перечисленных в таблице, а ограничусь только наиболее широко используемыми. Функция strcpy() копирует одну строку в другую. Исходная строка может быть переменной или строковой константой. Рассмотрим для примера следующий фрагмент:

```
// выделение памяти для строки из 29 символов
char buff[30];
// копирование строковой константы в буфер
strcpy(buff, "This is a test.");
// вывод строки на экран
cout << buff << endl;
// инициализация второго буфера
char buff2[] = "A second string.");
// копирование содержимого этой строки в первый буфер
strcpy(buff, buff2);
cout << buff << endl;
```

Случайный выход за верхнюю границу символьного массива еще более вероятен, чем для числовых массивов, обсуждавшихся ранее. Например, представьте, что вы сделали следующее:

```
char buff[10] = "A string";
// ...
strcpy(buff, "This is a test.");    // ошибка!
```

Здесь мы объявили символьный массив для хранения 10 символов и инициализировали его строкой, требующей 9 байт (не забывайте о завершающем нуль-символе). Позднее, возможно, забыв о величине массива, мы скопировали в буфер строку, требующую 16 байт, и тем самым вышли за границу массива на 6 байт. В результате этого небольшого ложного шага где-то в памяти были испорчены шесть байт. Будьте осторожны, копируя данные в символьные массивы.

Другая часто используемая при работе со строками функция — sprintf(). Эта функция позволяет получить форматированную строку, состоящую из текста и чисел. В следующем примере складываются два числа, после чего используется функция sprintf() для построения строки, сообщающей о результате:

```
char buff[20];
int x = 10 * 20;
sprintf(buff, "The result is: %d", x);
cout << buff;
```

После выполнения этого фрагмента программы на экран будет выведено:

```
The result is: 200
```

В данном примере %d указывает функции sprintf(), где необходимо вставить целое число. После форматной строки в списке аргументов стоит переменная x, значение которой будет вставлено в строку на указанное место. Функция sprintf() уникальна тем, что она может иметь переменное число аргументов. Вы должны обязательно указать буфер и форматную строку, но количество аргументов после форматной строки может быть любым. Вот пример использования sprintf() с тремя дополнительными аргументами:

```
int x = 20;
int y = 5;
sprintf(buff, "%d + %d = %d", x, y, x + y);
cout << buff;
```

После выполнения этого фрагмента кода на экране будет следующий результат:

```
20 + 5 = 25
```



Одиночная наклонная черта (slash) используется в строках для представления специальных символов. Например, '\n' обозначает перевод строки, '\t' – символ табуляции. Чтобы вставить в строку саму наклонную черту, необходимо использовать двойную наклонную черту:

```
strcpy(fileName, "c:\\windows\\system\\win.ini");
```

Забывая про этот факт, программисты проводили много бессонных ночей в попытках отладить программу. Это очень распространенная ошибка. Не говорите потом, что я не рассказывал вам о ней!

Функция sprintf() имеет двоюродную сестру по имени wsprintf(), которая является Windows-версией sprintf(). Вы можете встретить в Windows-программах обе эти функции. wsprintf() функционально идентична sprintf(), с одной лишь существенной разницей: она не позволяет помещать в форматную строку числа с плавающей точкой. Вы можете использовать любую из этих

функций в своих программах, но sprintf() предпочтительней, т.к. она полностью поддерживает плавающую точку (к тому же набирать нужно на один символ меньше!). Чтобы получить полное представление о возможностях sprintf(), используйте встроенную справочную систему C++Builder.

Массивы строк

Вы можете работать не только с символьными массивами, но и массивами символьных массивов (иначе говоря, с массивами строк). Возможно, это звучит несколько запутанно, но на самом деле вы уже имели дело с подобным массивом в программе ARGTEST, рассмотренной раньше. Вы можете объявить массив строк следующим образом:

```
char strings[][20] = {
"This is a string 1",
"This is a string 2",
"This is a string 3",
"This is a string 4",
};
```

Хотя можно использовать массивы и такого типа, в C++Builder существует более простой способ работы со строковыми массивами. (Я откладываю его рассмотрение до тех пор, пока вы не узнаете больше о C++ Builder.)



Если вы собираетесь интенсивно использовать строковые массивы, следует заглянуть в библиотеку стандартных шаблонов (STL — Standard Template Library). STL содержит классы Си++, которые позволяют работать с массивами строк гораздо проще, чем при использовании массивов в стиле Си. STL также включает класс string.

Заключение

Сегодня вы изучили большое количеством основополагающего материала. Прежде всего, вы в общих чертах познакомились с IDE C++Builder, создав GUI-программу Hello World. После этого вы узнали о консольных приложениях и написали программу Hello World, часть II. Затем после небольшого вступления вы приступили к изучению основ Си как фундамента для освоения Си++. Вы узнали о следующих элементах Си и Си++:

- ☐ Переменных
- ☐ Операторах
- ☐ Типах данных
- ☐ Функциях
- ☐ Функции main()
- ☐ Массивах
- ☐ Представлении строк в Си и Си++

В этой главе собрано большое количество материала. Не огорчайтесь, если вам не удалось во всем разобраться. Вернитесь назад и заново перечитайте то, что осталось неясно.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Чем отличаются GUI-приложения Win32 и консольные приложения Win32?

GUI-приложение — это обычная Windows-программа. Ее окно, как правило, содержит заголовок, меню и рабочую область. Консольное приложение — это 32-разрядная программа, которая запускается в окне MS-DOS. Консольное приложение внешне выглядит как DOS-программа.

Могут ли мои функции принимать аргументы и возвращать значения?

Функции, которые вы пишете, могут иметь аргументы и возвращать значение, но ни то, ни другое не обязательно. Если написанная функция возвращает значение, вы должны использовать для этого оператор return, иначе компилятор выдаст предупреждение.

Могу ли я присвоить переменной целого типа значение, содержащее дробную часть?

Да, но дробная часть при этом будет отброшена (без округления) и переменной будет присвоена только целая часть числа.

Может ли Си++ гарантировать, что я не испорчу какую-либо область памяти, если случайно выйду за верхнюю границу массива?

Нет. Одним из преимуществ Си++ является то, что он дает вам возможность прямого доступа к памяти. Но это влечет за собой и ответственность. Программист должен сам заботиться о том, чтобы программа использовала только отведенную ей память. Если вы случайно выйдете за границу отведенной памяти, Windows выдаст сообщение об общем нарушении защиты (GPF — General Protection Fault) или нарушении прав доступа. GPF может возникнуть сразу или через некоторое время, когда измененный участок памяти будет использован другой частью программы, другой программой или самой Windows.

Вопросы

1. Что неправильно в этой программе?

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

void displayText();
displayText()
{
cout << "Hello Bubba!" << endl;
}
```

2. Сколько значений может возвращать функция?
3. Что делает функция strcpy()?
4. Какое значение имеет переменная после объявления?
5. Сколько функций может содержать программа?
6. Может ли функция вызывать другую функцию?
7. Что неправильно в этой программе?

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

int main(int argc, char** argv)
{
  doSomething();
  return 0;
}

void doSomething()
{
  cout << "I'm doing something now" << endl;
}
```

8. Сколько функций с именем main() может содержать программа?
9. Посмотрите на строку программы:

```
char buff[20];
```

Строку какой длины можно разместить в этом массиве?

10. Какой индекс имеет первый элемент массива, 0 или 1?

Упражнения

1. Напишите Windows GUI-программу, которая после запуска отображает в своем окне слова «Welcome to C++Builder!»
2. Измените программу из упражнения 1 так, чтобы она отображала текст Hello There! (**Указание:** вы должны изменить свойство Caption компонента Label.)
3. Напишите консольное приложение Windows, которое выводит на экран текст This is a test.
4. Напишите консольное приложение Windows. Объявите в программе две переменные и присвойте им значения. Перемножьте эти два числа и выведите результат на экран.
5. Напишите консольное приложение, которое вызывает функцию для вывода на экран текста Function entered, sir!!
6. Напишите консольное приложение, которое принимает в качестве аргумента целое число, умножает его само на себя и возвращает результат.
7. Введите и откомпилируйте следующую программу:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
#include <stdio.h>
#pragma hdrstop
```

```
void getSqrRoot(char* buff, int x);
int main(int argc, char** argv);
{
  int x;
  char buff[30];
  cout << "Enter a number: ";
  cin >> x;
  getSqrRoot(buff, x);
  cout << buff;
  getch();
  return 0;
}

void getSqrRoot(char* buff, int x)
{
  sprintf(buff, "The square root is: %f", sqrt(x));
}
```

Что эта программа делает?

Операторы управления и структуры



Кент Рейсдорф

Вы взяли хороший старт в изучении Си++. В этой главе будет продолжен рассказ об основах Си++, берущих начало в Си. Сегодня вы узнаете о следующих элементах языка:

- Ключевых словах if и else
- Циклах: for, do и do-while
- Операторе switch
- Областях видимости
- Структурах

Оператор if

Существует несколько программных конструкций, общих для всех языков программирования. Одна из них представлена в Си++ оператором if. Оператор if используется для проверки некоторого условия и последующего выполнения одного из блоков кода в зависимости от того, имеет ли результат проверки значение true (истина) или false (ложь). Рассмотрим пример:

```
int x;
cout << "Enter a number: ";
cin >> x;
if (x > 10)
  cout << "You entered a number greater than 10." << endl;
```

Этот фрагмент программы запрашивает у пользователя число. Если введенное число больше 10, выражение x > 10 принимает значение true и на экран выводится сообщение; в противном случае ничего не выводится. Обратите внимание, что если условное выражение имеет значение true, то выполняется оператор, следующий непосредственно за оператором if.



Оператор if используется для проверки условия и последующего выполнения блока кода в зависимости от того, истинно это условие или ложно.



Не завершайте условное выражение точкой с запятой. Точка с запятой сама по себе обозначает пустой оператор. Если вы случайно поставите в конце оператора if точку с запятой, компилятор интерпретирует пустой оператор как оператор, который нужно выполнить при истинности условия. Вот пример:

```
if (x == 10);          // Осторожно! Лишняя точка с запятой!
  DoSomething(x);
```

В этом случае функция DoSomething(x) будет вызываться в любом случае, т.к. компилятор не считает ее первым оператором после if. Поскольку этот код абсолютно законен (хотя и не выполняет предполагаемой задачи), компилятор не выдаст никаких предупреждений.

Если в случае истинности условного выражения должны выполняться несколько строк кода, необходимо использовать фигурные скобки для объединения этих строк в блок:

```
if (x > 10) {
  cout << "The number is greater than 10" << endl;
  DoSomethingWithNumber(x);
}
```

Если условное выражение принимает значение false, блок кода, связанный с оператором if, игнорируется, и выполнение программы продолжается с первого оператора, следующего за этим блоком.



В Си++ есть много возможностей упростить запись программ. Одна из них заключается в том, что для проверки на истинность достаточно только имени переменной. Посмотрите на этот фрагмент:

```
if (fileGood) ReadData();
```

и сравните его с более длинной формой:

```
if (fileGood == true) ReadData();
```

В данном примере использована переменная типа bool, однако то же самое верно для любого другого типа данных. Выражение истинно, пока переменная содержит ненулевое значение. Вы можете выполнить проверку на ложность, применив к переменной оператор логического отрицания (!):

```
bool fileGood = OpenSomeFile();
if (!fileGood) ReportError();
```

Знание подобных возможностей Си++ поможет вам писать более элегантные программы, а также разбираться в текстах примеров и листингах.

В некоторых случаях при истинности условия нужно выполнить одни действия, а при ложности — другие. Для этого применяется оператор else:

```
if (x == 20) {
  DoSomething(x);
}
else {
  DoADifferentThing();
}
```



Оператор else используется в сочетании с оператором if и определяет блок кода, который выполняется в том случае, когда условное выражение принимает значение false.

В приведенном примере будет вызвана только одна из двух функций.



Обратите внимание, что для оператора равенства используются два знака «равно» (==), а для оператора присваивания — один (=). Распространенная ошибка заключается в том, что вместо оператора равенства используется оператор присваивания. Например, если предыдущий пример по невнимательности написан как

```
if (x = 20) {
  DoSomething(x);
}
```

то x получит значение 20. Поскольку эта операция пройдет успешно, условное выражение примет значение true. Подобные ошибки, несмотря на их тривиальность, очень трудно устранять, поэтому будьте внимательны при проверке на равенство.

Вы можете при необходимости использовать вложенные операторы if. В этом случае за обычным оператором if следует один или несколько таких же операторов. Рассмотрим пример:

```
if (x > 10)
  if (x < 20)
    cout << "X is between 10 and 20" << endl;
```

Имейте в виду, что это упрощенный пример. В реальной ситуации вы можете запутаться в лабиринте скобок, которые отделяют один блок от другого. Взгляните, например, на этот фрагмент:

```
if (x > 100) {
  y = 20;
  if (x > 200) {
    y = 40;
    if (x > 400) {
      y = 60;
      DoSomething(y);
    }
  }
}
else if (x < -100) {
  y = -20;
  if (x < -200) {
    y = -40;
    if (x < -400) {
      y = -60;
      DoSomething(y);
    }
  }
}
```

Даже это достаточно простой пример, но, надеюсь, вы поняли идею.



Редактор кода C++Builder имеет полезную функцию для поиска соответствующих друг другу скобок. Установите курсор на одну из скобок и нажмите комбинацию клавиш Alt+[или Alt+], после чего курсор установится на парную ей скобку. Не имеет значения, с какой скобки вы начали — с открывающей или закрывающей. *В любом случае соответствующая скобка будет найдена.*



Если блок кода содержит более двух или трех последовательных операторов if, проверяющих одну и ту же переменную на разные значения, то это подходящий случай для использования оператора switch. Оператор switch будет рассмотрен далее в соответствующем разделе этой главы.

Ранее я упомянул о сокращениях в Си++. Для комбинации if-else также существует сокращенная запись. Взгляните на следующий фрагмент:

```
if (direction == EAST) lost = true;
else (lost == false);
```

Эти две строки можно объединить в одну:

```
direction == EAST ? lost = true : lost = false;
```

Хотя это сокращение поначалу выглядит немного странно, вы быстро научитесь распознавать его в тексте. Оператор if очень широко используется в Си++. Он на редкость прямолинеен и у вас не должно возникнуть с ним никаких трудностей. Главное — правильно расставлять скобки.

Синтаксис

Оператор if, первая форма:

```
if (cond_expr) {
    true_statements;
}
else {
    false_statements;
}
```

Если условное выражение cond_expr истинно (не равно нулю), то выполняется блок true_statements. Если это выражение ложно, то при наличии необязательного оператора else выполняется блок false_statements.

Оператор if, вторая форма:

```
if (cond_expr_1) {
    true_statements_1;
}
else if (cond_expr_2) {
    true_statements_2;
}
else {
    false_statements;
}
```

Если условное выражение cond_expr_1 истинно (не равно нулю), выполняется блок true_statements_1. Если это выражение ложно и истинно выражение cond_expr_2, выполняется блок true_statements_2. Если оба условия cond_expr_1 и cond_expr_2 ложны, выполняется блок false_statements.

Циклы

Цикл является общей конструкцией для всех языков программирования. Он может быть использован для обработки массивов, для выполнения какого-либо действия определенное количество раз, для чтения файла с диска... Возможностей бесконечно много. Я рассмотрю здесь циклы нескольких типов, и в основном они работают похожим образом. Любой цикл состоит из следующих элементов:

- Точки входа
- Тела цикла, обычно заключенного в фигурные скобки, которое содержит операторы, выполняемые за один проход
- Точки выхода
- Условного выражения, определяющего момент окончания цикла
- Дополнительно используемых операторов break и continue



Цикл (loop) — это конструкция языка программирования, которая используется для выполнения каких-либо действий определенное количество раз.

Точкой входа в цикл является один из операторов цикла Си++ (for, while или do), за которым следует открывающая фигурная скобка. Тело цикла содержит операторы, которые будут выполняться при каждом проходе цикла. Тело цикла может содержать любой допустимый в Си++ код. Точкой выхода является закрывающая скобка.

Большинство циклов работает примерно следующим образом. После входа в цикл вычисляется условное выражение. Если оно истинно, выполняются операторы, составляющие тело цикла. Когда программа достигает нижней точки цикла (обычно это закрывающая скобка), происходит переход на начало и снова вычисляется условное выражение. Если оно истинно, весь процесс повторяется. Если условие не выполнено, управление передается на оператор, следующий за закрывающей скобкой. Исключением из этих правил является цикл do-while, у которого проверка условия выполняется не в начале, а в конце цикла.

Условное выражение определяет момент окончания цикла. Например, условие может гласить «выполнять, пока x равно 10» или «выполнять чтение, пока не будет достигнут конец файла». Цикл продолжается до тех пор, пока условие истинно.



Довольно легко по ошибке написать цикл, в котором условие будет всегда выполнено. Это приведет к зависанию (зацикливанию) программы. Вам останется только нажать Ctrl+Alt+Del и прекратить ее выполнение. Появится окно завершения программы Windows, в котором после названия программы будет стоять Not Responding (Не отвечает). Вам нужно будет выбрать программу из списка и щелкнуть на кнопке End Task (Снять задачу).



В C++Builder программа обычно запускается с помощью кнопки Run на оперативной панели или нажатием F9. Если нужно прекратить выполнение программы, запущенной из IDE, вы можете выбрать в главном меню пункт Run | Reset Process или нажать комбинацию клавиш Ctrl+F2.

После этого общего обзора давайте рассмотрим каждый тип цикла в отдельности.

Цикл for

Цикл for, пожалуй, используется чаще других. Он имеет три параметра: инициализирующее выражение, условное выражение и инкрементное выражение.

Синтаксис

Оператор цикла for:

```
for (initial; cond_expr; adjust) {
    statements;
}
```

В цикле for блок кода, обозначенный как statements, выполняется до тех пор, пока условное выражение cond_expr истинно (не равно нулю). Для инициализации цикла используется оператор initial. После выполнения блока statements переменная цикла модифицируется оператором adjust.

Чтобы лучше понять все вышесказанное, рассмотрим несколько конкретных примеров. Для начала возьмем типичный for-цикл:

```
for (int i = 0; i < 10; i++) {
  cout << "This is iteration " << i << endl;
}
```

В этом примере оператор внутри фигурных скобок будет выполнен 10 раз. Первый параметр, int i = 0, определяет начальное значение переменной цикла. (Я объявил и инициализировал переменную внутри оператора for. Это совершенно законно и часто применяется в for-циклах.) Второй параметр, i < 10, является условием, согласно которому цикл будет повторяться, пока значение переменной i меньше 10. Поскольку я начал с 0, нужно будет остановиться до того, как i примет значение 10. Последний параметр, i++, увеличивает переменную i на единицу после каждого прохода.



Имя i для переменной (по-видимому, от iterator) традиционно используется в циклах for. Разумеется, можно использовать и любое другое имя, но вы часто будете встречать i в этих циклах.

Давайте рассмотрим другой вариант того же цикла, который достигает в точности обратного эффекта по сравнению с первым:

```
for (int i = 10; i > 0; i--) {
  cout << "This is iteration " << i << endl;
}
```

Теперь я начинаю с 10, останавливаюсь, когда i равно 0 и уменьшаю i на единицу в каждом проходе. Это пример цикла с обратным счетом.



Давайте напишем небольшую программу, иллюстрирующую использование цикла for. Введите, откомпилируйте и запустите программу из листинга 2.1. Она называется FORLOOP.CPP и содержится также на CD-ROM, прилагаемом к книге. Результат работы программы показан на рис. 2.1.



В предыдущих примерах открывающая и закрывающая фигурные скобки не обязательны. При отсутствии скобок телом цикла считается оператор, следующий непосредственно за оператором for. Для ясности и лучшей читаемости программ неплохо расставлять скобки даже тогда, когда в этом нет особой необходимости.

Листинг 2.1. FORLOOP.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4:
 5: int main(int argc, char** argv)
 6: {
 7:   cout << endl << "Starting program..." << endl << endl;
 8:   int i;
 9:   for (i = 0; i < 10; i++) {
10:     cout << "Iteration " << i << endl;
11:   }
12:   cout << endl;
13:   for (i = 10; i > 0; i--) {
14:     cout << "Iteration " << i << endl;
15:   }
16:   getch();
17:   return 0;
18: }
```

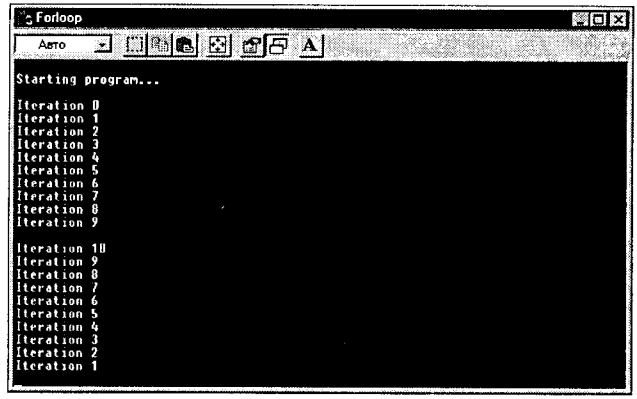


Рис. 2.1
Результат работы программы FORLOOP.CPP

Теперь вы знаете, что начальное значение переменной цикла может быть любым (предполагается, конечно, что оно лежит в диапазоне значений выбранного типа данных). В качестве условного выражения допустимо любое выражение Си++, результатом которого может быть true. Величина, с которой сравнивается переменная цикла, может быть константой, как в вышеприведенных примерах, переменной или возвращаемым значением функции. Ниже приведены примеры правильной записи условных выражений:

```
for (int i = 0; i < 100; i++) {...}
for (int i = 1; i == numberOfElements; i++) {...}
for (int i = 0; i <= GetNumberOfElements(); i += 2) {...}
```

Обратите внимание на последний пример. Здесь я увеличиваю счетчик на 2 при каждом проходе цикла. Вы можете использовать инкрементное

выражение с любым параметром инкремента. Например, в этом цикле происходит увеличение на 10:

```
for (int i = 0; i < 100; i += 10) {...}
```

После того, как вы увидели цикл for в действии, не составит большого труда применить те же принципы к циклам while и do-while. Сейчас мы перейдем к их рассмотрению.

Цикл while

Цикл while отличается от цикла for тем, что он содержит только условие, которое проверяется в начале каждой итерации. Пока условие истинно, цикл выполняется. Рассмотрим пример:

```
int x;
while (x < 1000) {
  x = DoSomeCalculation();
}
```

В этом примере я вызываю функцию, которая когда-либо должна возвратить значение, большее 1000. Пока возвращаемое значение меньше 1000, цикл while продолжает выполняться. Как только переменная x примет значение, большее или равное 1000, условие станет ложным и выполнение программы продолжится с первой строки за закрывающей фигурной скобкой цикла. Обычно условное выражение в циклах while содержит переменную типа bool, значение которой изменяется внутри тела цикла:

```
bool done = false;
while (!done) {
  // операторы тела цикла
  done = SomeFunctionReturningABool();
  // операторы тела цикла
}
```

Ожидается, что в какой-то момент переменная done примет значение false и цикл закончится. Программа из листинга 2.2 иллюстрирует использование цикла while.

Листинг 2.2. WHILETST.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4: int main(int argc, char** argv)
 5: {
 6:   cout << endl << "Starting program..." << endl << endl;
 7:   int i = 6;
 8:   while (i-- > 0) {
 9:     cout << endl << "Today I have " << i;
10:     cout << " problems to worry about.";
11:   }
12:   cout << "\b!\nYipee!";
13:   cout << endl << endl << "Press any key to continue...";
14:   getch();
15:   return 0;
16: }
```

Синтаксис

Оператор цикла while:

```
while (cond_expr) {
    statements;
}
```

Пока условное выражение cond_expr имеет значение true (не равно нулю), выполняется блок statements. Цикл должен быть инициализирован до оператора while, а изменение переменной цикла должно происходить внутри тела цикла. Цикл заканчивается, как только выражение cond_expr примет значение false.

Цикл do-while

Цикл do-while похож на цикл while. Однако между ними есть существенное различие. Как вы могли видеть из листинга 2.2, в цикле while условие проверяется до начала выполнения тела цикла. В случае цикла do-while условное выражение вычисляется в нижней части цикла:

```
bool done = false;
do {
  // операторы тела цикла
  done = SomeFunctionReturningABool();
  // операторы тела цикла
} while (!done);
```

Какой цикл использовать, while или do-while, зависит от того, что этот цикл делает.

Синтаксис

Оператор цикла do-while:

```
do {
    statements;
} while (cond_expr);
```

Пока условное выражение cond_expr имеет значение true (не равно нулю), выполняется блок statements. Цикл должен быть инициализирован до оператора do, а изменение переменной цикла должно происходить внутри тела цикла. Цикл заканчивается, как только выражение cond_expr примет значение false.

Оператор goto

Я упомяну goto только для того, чтобы вы знали о его существовании. Оператор goto позволяет перейти на предварительно объявленную метку. После имени метки ставится двоеточие. Это иллюстрируется следующим фрагментом кода:

```
bool done = false;
StartPoint:
// какие-то операторы
if (!done) goto (StartPoint);
// цикл закончен
```

Здесь нет необходимости использовать фигурные скобки, поскольку все строки между оператором goto и меткой будут выполнены.



Использование оператора goto в программах на Си++ считается плохим стилем. Все, что вы хотели бы сделать с помощью goto, можно сделать с помощью циклов while или do-while. Очень немногие из уважающих себя программистов на Си++ используют goto в своих программах. Если вы переходите к программированию на Си++ от другого языка, использующего операторы goto, то обнаружите, что основные конструкции Си++ делают goto ненужным.



Оператор goto:

```
    goto label
    .
    .
    .
label:
```

Оператор goto осуществляет безусловный переход на строку, обозначенную меткой label.

Операторы continue и break

До того, как мы закончим рассмотрение циклов, вам нужно узнать еще о двух ключевых словах, которые помогают управлять выполнением цикла. Оператор continue используется для перехода на начало цикла. Например, вам нужно, чтобы часть цикла не выполнялась, если результатом некоторой проверки будет true. В этом случае следует использовать continue, чтобы перейти обратно к началу цикла и избежать выполнения кода, расположенного после оператора continue:

```
bool done = false;
while (!done) {
  // операторы тела цикла
  bool error = SomeFunction();
  if (error) continue; // переход к началу цикла
  // операторы, которые будут выполняться при отсутствии ошибки
}
```

Оператор break используется для досрочного выхода из цикла. Например, вы просматриваете массив типа int в поисках определенного числа. Прекратив выполнение цикла поиска, когда это число будет найдено, вы можете получить индекс нужного элемента массива:

```
int index = 0;
int searchNumber = 50;
for (int i = 0; i < numElements; i++) {
  if (myArray[i] == searchNumber) {
    index = i;
    break;
  }
}
if (index)
  cout << "Number found at index " << index << endl;
else
  cout << "Number not found in array." << endl;
```

Операторы continue и break полезны во многих ситуациях. Вы откроете для себя все возможные применения этих операторов, так же как и большей части того, о чем я говорил, когда приобретете некоторый опыт программирования на Си++.

Оператор switch

Оператор switch можно рассматривать как усложненный if. Он позволяет вам выполнять один из нескольких блоков кода в зависимости от значения выражения. Выражение может быть переменной, возвращаемым значением функции или любым арифметическим выражением Си++. Вот пример использования оператора switch:

```
switch (amountOverSpeedLimit) {
  case 0 : {
    fine = 0;
    break;
  }
  case 10 : {
    fine = 20;
    break;
  }
  case 15 : {
    fine = 50;
    break;
  }
  case 20 :
  case 25 :
  case 30 : {
    fine = amountOverSpeedLimit * 10;
    break;
  }
  default : {
    fine = GoToCourt();
    jailTime = GetSentense();
  }
}
```

Оператор switch состоит из нескольких частей. Прежде всего, вы можете заметить выражение, которое в этом примере представляет собой переменную amountOverSpeedLimit (вспомните, что я говорил насчет длинных имен переменных!). Затем операторы case проверяют выражение на равенство определенной величине. Если amountOverSpeedLimit равно 0 (case 0), переменной fine присваивается значение 0. Если amountOverSpeedLimit равно 10, fine получает значение 20, и т.д. В каждом из первых трех case присутствует оператор break. Он используется для выхода из блока switch — это означает, что значение выражения совпало с одной из величин и оставшаяся часть оператора switch может быть игнорирована. В самом конце вы видите оператор default. Блок, следующий за этим оператором, выполняется при отсутствии совпадений.

Обратите внимание, что для case 20 и case 25 соответствующие операторы отсутствуют. Если выражение amountOverSpeedLimit равно 20 или 25, эти две секции проходятся насквозь и выполняется следующий за ними блок кода. В результате для значений 20, 25 и 30 будет выполнен один и тот же код.



Не забывайте об операторах break! Без них работа switch будет продолжаться даже после обнаружения совпадения и может привести к выполнению блока, который по вашему замыслу не должен был выполняться. Впрочем, иногда это соответствует логике работы вашего switch, но чаще всего нет.

Включение оператора default не обязательно. Вы можете написать switch и без него:

```
switch (x) {
  case 10 : DoSomething(); break;
  case 20 : DoAnotherThing(); break;
  case 30 : TakeABreak();
}
```

Обратите внимание, что последний оператор case не имеет своего оператора break. Поскольку это последняя строка switch, нет смысла включать в нее break.

Как я уже говорил, вы можете использовать switch вместо нескольких расположенных подряд if. Это несколько облегчит для других понимание вашей программы.

Синтаксис

Оператор switch

```
switch (expr) {
    case value_1:
        statements_1;
        break;
    case value_2:
        statements_2;
        break;
    .
    .
    .
    case value_n:
        statements_n;
        break;
    default:
        dflt_statements;
        break;
}
```

Оператор switch позволяет выполнять различные блоки кода в зависимости от значения выражения expr. Блок statements_1 выполняется, когда expr равно value_1, блок statements_2 — когда expr равно value_2, и т.д. до блока statements_n, когда expr равно value_n. Если expr не равно ни одному из значений value_1 ... value_n, выполняется блок dflt_statements. Операторы break не являются обязательными.

Области видимости

Термин *область видимости (scope)* обозначает ту часть программы, в пределах которой возможен доступ к данной переменной. Большинство переменных являются *локальными*. Это означает, что доступ к ним возможен только в пределах того блока, в котором они объявлены. Взгляните на программу, приведенную в листинге 2.3.



Под *областью видимости (scope)* переменной понимается та часть программы, в пределах которой возможен доступ к этой переменной.

Листинг 2.3. SCOPE.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #pragma hdrstop
 4: int x = 20;
 5: void CountLoops(int);
 6: int main(int, char**)
 7: {
 8:   int x = 40;
 9:   int i = 0;
10:   cout << "In main program x = " << x << endl;
11:   bool done = false;
12:   while (!done) {
13:     int x;
14:     cout << endl << "Enter a number (-1 to exit): ";
15:     cin >> x;
16:     if (x != -1) {
17:       cout << endl << "In while loop x = " << x;
18:       CountLoops(++i);
19:     }
20:     else
21:       done = true;
22:   }
23:   cout << endl << "Global x = " << ::x << endl;
24:   cout << endl << "Press any key to continue...";
25:   getch();
26:   return 0;
27: }
28: void CountLoops(int x)
29: {
30:   cout << ", While loop has executed "
31:     << x << " times" << endl;
32: }
```

Первое, что вы могли заметить — это четырехкратное объявление переменной x. Она объявлена в строке 4 вне функции main(), в строке 8 внутри main(), в строке 13 внутри цикла while и в функции CountLoops() (строка 28). Если вы случайно объявляете переменную более одного раза, компилятор выдает сообщение об ошибке Multiple declaration for 'x' (Многократное объявление 'x') и компиляция прекращается. Тем не менее эта программа компилируется и запускается вполне нормально. Почему? Дело в том, что переменные x из листинга 2.3 имеют различные области видимости.

Посмотрите внимательнее на листинг 2.3. Переменная x в строке 13 объявлена внутри тела цикла while и является локальной по отношению к этому блоку кода. В результате она не существует вне цикла. Аналогично этому переменная x, объявленная в строке 28, локальна по отношению к функции CountLoops() и не существует вне этой функции. В данном случае объявление x менее очевидно, т.к. оно содержится в списке параметров функции, но тем не менее это именно объявление переменной.

Теперь обратите внимание на переменные x и i, объявленные внутри функции main(). Эти переменные локальны по отношению к тому блоку, в

пределах которого объявлены, а также доступны из любого внутреннего блока. Другими словами, областью видимости переменных x и i является как функция main(), так и цикл while. Это достаточно легко понять в случае переменной i, поскольку она объявлена один раз. Но как быть с x? Ведь внутри цикла while существуют две переменные с именем x (одна из них объявлена в main(), а другая — в самом цикле), и обе находятся в области видимости. Которая из них будет использована? Та, которая объявлена в цикле while, потому что она имеет меньшую область видимости.



Недавние изменения в стандарте Си++ коснулись областей видимости переменных, объявленных внутри операторов типа for. Например, следующий код приведет к ошибке компиляции:

```
for (int i = 0; i < 10; i++) {
  if (array[i] == 40) break;
}
index = i;
```

Ошибка возникнет вследствие того, что переменная i видима только внутри блока, соответствующего циклу for. Чтобы исправить положение, нужно объявить i вне оператора for:

```
int i;
for (i = 0; i < 10; i++) {
  if (array[i] == 40) break;
}
index = i;
```

Хотя это изменение не является существенным для вас, как для начинающего, в свое время оно повергло в шок многих ветеранов программирования на Си++. В конце концов, не имеет большого значения, какая форма принята в качестве стандарта, пока мы, программисты, знаем правила.

Наконец, мы переходим к переменной x, объявленной вне функции main() (строка 4). Поскольку эта переменная объявлена вне всех функций, она называется *глобальной переменной (global variable)* и имеет *глобальную область видимости (global scope)*. Это означает, что переменная x доступна всюду в программе: внутри функции main(), внутри цикла while и внутри функции CountLoops().

Как было упомянуто раньше, локальная переменная перекрывает глобальную. Но что делать, если вы хотите обратиться к глобальной переменной x из функции main()? Для этого нужно использовать *оператор разрешения видимости (scope-resolution operator)*, ::. Этот оператор содержится в строке 23 листинга 2.3:

```
cout << "Global x = " << ::x << endl;
```

Оператор разрешения видимости сообщает компилятору, что нужно взять значение глобальной переменной x, а не локальной. (Этот оператор используется также при работе с классами, но я расскажу об этом позже, когда мы перейдем к изучению классов.)

Переменные extern

Реальные приложения обычно состоят из нескольких файлов исходного текста. (Термины *модуль (module, unit)* и *исходный файл (source file)* взаимозаменяемы. Я расскажу о многомодульных программах очень кратко.) Глобальные переменные, объявленные в одном из исходных файлов, являются глобальными для этого файла, но не видимы в других модулях. Однако бывают ситуации, когда переменная должна быть доступна из любого модуля программы. Для этого нужно выполнить два действия. Во-первых, объявить переменную в одном из исходных файлов как обычную глобальную переменную. Затем во всех остальных файлах, где она должна быть видима, нужно снова объявить эту переменную с ключевым словом extern:

```
extern int countChickens;
```

Ключевое слово extern сообщает компилятору, что определение переменной, объявленной таким способом, находится в другом файле. Во время построения программы компилятор соберет все файлы вместе и обеспечит доступ к нужным переменным.

Хотя глобальные переменные удобны, их использование не приветствуются в концепции ООП. Обычно можно найти лучшие решения (о которых вы узнаете при изучении классов). К тому же глобальные переменные постоянно занимают память во время выполнения программы. Память для локальных переменных выделяется только на время их нахождения в области видимости. Используйте локальные переменные всюду, где это возможно, и старайтесь свести количество глобальных переменных к минимуму.

Структуры

Структура — это набор взаимосвязанных данных, объединенных в единое целое. Допустим, вам нужно хранить список почтовых адресов. Для этого удобно иметь одну переменную, содержащую все поля типичного адреса. Такую возможность как раз и предоставляет структура. Сначала вам нужно объявить структуру, а затем создать экземпляр этой структуры. Структура описывается с помощью ключевого слова struct:

```
struct mailingListRecord {
  char firstName[20];
  char lastName[20];
  char address[50];
  char city[20];
  char state[4];
  int zip;
  bool aFriend;
  bool aFoe;
};
```

Элементы структуры называются *членами-данными (data member)*. Каждый член структуры должен быть объявлен так же, как и обычная переменная. В этом примере структура содержит пять массивов типа char, один член типа int, и два члена типа bool. Наконец, обратите внимание на точку с запятой после закрывающей фигурной скобки. Она требуется в описаниях классов и структур.



Структура (structure) — это набор взаимосвязанных данных, объединенных под общим именем. После описания структуры возможно создание и использование конкретного *экземпляра (instance)* данной структуры. Элементы структуры называются *членами-данными (data member)*.



Вы можете создавать экземпляры структуры непосредственно при ее описании. Для этого вставьте имя переменной (или имена нескольких переменных) между закрывающей фигурной скобкой и точкой с запятой. Например:

```
struct point {
  int x;
  int y;
} upperLeft, lowerRight;
```

Здесь описывается структура и одновременно создаются два ее экземпляра с именами upperLeft и lowerRight.

Теперь, когда структура описана, я могу ее использовать. Сначала нужно создать экземпляр структуры. Это выглядит следующим образом:

```
mailingListRecord record;
```

В результате для структуры выделяется память (120 байт) и эта область памяти связывается с переменной, имеющей имя record. Затем я могу присваивать значения членам-данным:

```
strcpy(record.firstName, "Bruce");
strcpy(record.lastName, "Reisdorph");
strcpy(record.address, "123 Inspiration Pt.");
strcpy(record.city, "Merced");
strcpy(record.state, "CA");
record.zip = 95031;
record.aFfriend = true;
record.aFoe = false;
```

Здесь вы встретились с новым оператором. Для доступа к членам-данным используется *оператор доступа к членам структуры (structure member operator)*, который представляет собой точку между именем переменной и именем члена структуры. (Если вы забудете добавить этот оператор, то скорее всего компилятор сообщит что-то насчет неопределенного символа.) Оператор доступа позволяет вам обращаться к конкретному члену структуры — как для чтения, так и для изменения его значения.

При желании вы можете инициализировать сразу все члены вновь созданного экземпляра структуры:

```
mailingListRecord rec = {
  "Bruce",
  "Reisdorph",
  "123 Inspiration Pt.",
  "Merced",
  "CA",
  95031,
  true,
  false
};
```

Этот способ короче предыдущего, но он не всегда применим в реальных ситуациях. Обычно структура заполняется в результате ввода данных пользователем или чтения их из файла. В этих случаях присваивание значений подобным образом невозможно.

Синтаксис

Оператор struct

```
struct name {
    data_member_1;
    data_member_2;
    .
    .
    .
    data_member_n;
} instance;
```

Оператор struct описывает группу членов-данных (data_member_1, data_member_2, ..., data_member_n) и присваивает этой группе имя (name). Дополнительно можно создать конкретный экземпляр (instance) описанной структуры.

Массивы структур

Вы можете организовать массив структур точно так же, как и массив данных типа int, char или long. Объявление и использование такого массива выглядит следующим образом:

```
mailingListRecord listArray[5];
strcpy(listArray[0].firstName, "Chuck");
listArray[4].aFoe = true;
// и т.д.
```

Это не намного сложнее, чем использование массива данных любого встроенного типа. Как видите, индексный оператор и оператор доступа к членам структуры используются совместно.

Заголовки и исходные файлы

Исходный файл (source file) — это текстовый файл в формате ASCII, содержащий текст программы. Компилятор обрабатывает этот файл и формирует соответствующий машинный код, выполняемый процессором.

Один из недостатков учебников по программированию заключается в том, что в них используются слишком простые примеры для иллюстрации основных концепций и идей. Вы будете постоянно обнаруживать, что на самом деле все не так просто. Так, мы постоянно имеем дело с очень короткими программами, содержащимися в одном файле. Программы для решения реальных задач обычно разбиваются на несколько исходных файлов. Это делается по нескольким причинам. Одной из основных является лучшая организация программы. Держа взаимосвязанные части кода вместе, можно легче находить нужные фрагменты.

Каким же образом все исходные файлы стыкуются вместе? Прежде всего, для каждого исходного файла (.cpp) компилятор формирует соответствующий объектный файл (.obj). После компиляции всех модулей, компоновщик собирает объектные модули в единый исполняемый файл (.exe). На этом за-

ключительном этапе компоновщик может также присоединить другие необходимые файлы, например, файлы ресурсов (.res) или библиотечные файлы (.lib).



Объявления классов и структур часто находятся в отдельном файле, называемом *заголовочным файлом (header file)*. Эти файлы имеют расширение .h или .hpp. (Я кратко упоминал о заголовочных файлах, когда рассказывал о классе iostream в первый день, «Первое знакомство». Заголовочный файл должен содержать только объявления классов, структур и функций. Никогда не размещайте в нем исполняемые операторы.



От правила не размещать в заголовочных файлах код можно отступать только в одном случае. Вы можете включать в эти файлы подставляемые функции (inline functions). Подставляемые функции — это функции, для которых компилятор генерирует машинный код специальным образом. Вы узнаете о них больше на четвертый день, «Классы и объектно-ориентированное программирование», когда я буду рассказывать о классах.

После создания заголовочного файла для класса или структуры, вы можете включить этот файл в любой модуль с исходным текстом, в котором требуется объявление данного класса или структуры. Для этого служит директива #include:

```
#include "structur.h"
```

Использование директивы #include эквивалентно вставке текста включаемого файла в данное место исходного файла. Листинг 2.4 из следующего раздела содержит программу, которая использует директиву #include. Заголовочный файл для листинга 2.4 содержится в листинге 2.5.



В заголовочные файлы обычно включают специальные константы (sentry), чтобы гарантировать однократное включение данного файла при построении программы. Такая константа сообщает компилятору, что файл уже был обработан один раз и повторное включение не требуется. Использование этих констант выглядит следующим образом:

```
#ifndef _MYCLASS_H
#define _MYCLASS_H
class MyClass {
  // описание класса
};
#endif
```

C++Builder автоматически добавляет специальные константы в модули, которые генерируются в результате создания новых форм или компонентов. Если класс, для которого вы написали заголовочный файл, используется вне библиотеки C++Builder VCL, вы должны сами позаботиться о включении подобных констант.

Заголовочный файл может содержать объявления нескольких классов или структур. Использование отдельных файлов для каждого класса или структуры позволит вам лучше организовать свой проект и облегчить использование классов и структур в других программах. Иногда, напротив, стоит объединить несколько взаимосвязанных классов в одном заголовочном файле. Например, у вас может быть класс, который включает в себя вспомогательный класс для

выполнения определенных функций. В этом случае оба класса, основной и вспомогательный, желательно объявить в одном файле. Разумеется, организация заголовочных файлов в конечном счете зависит от вас.

Не беспокойтесь, если все сказанное кажется вам сейчас немного туманным. Скорее всего, для полного понимания потребуется некоторый опыт написания реальных программ.

Пример использования структур

Листинг 2.4 содержит программу, которая запрашивает у пользователя три имени и адреса, после чего сохраняет их в массиве структур. По окончании ввода имена выводятся на экран и пользователю предоставляется возможность изменить любую запись. После выбора одной из записей, она выводится на экран. Листинг 2.5 содержит заголовочный файл для структуры mailingListRecord, которая используется в программе MAILLIST из листинга 2.4.

Листинг 2.4. MAILLIST.CPP

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#pragma hdrstop
#include "structur.h"
void displayRecord(int, mailingListRecord mlRec);
int main(int, char**)
{
  //
  // Создание массива структур mailingListRecord
  //
  mailingListRecord listArray[3];
  cout << endl;
  int index = 0;
  // Ввод трех записей
  //
  do {
    cout << "First Name: ";
    cin.getline(listArray[index].firstName,
    sizeof(listArray[index].firstName) - 1);
    cout << "Last Name: ";
    cin.getline(listArray[index].lastName,
    sizeof(listArray[index].lastName) - 1);
    cout << "Address: ";
    cin.getline(listArray[index].address,
    sizeof(listArray[index].address) - 1);
    cout << "City: ";
    cin.getline(listArray[index].city,
    sizeof(listArray[index].city) - 1);
    cout << "State: ";
    cin.getline(listArray[index].state,
    sizeof(listArray[index].state) - 1);
    char buff[10];
    cout << "Zip: ";
    cin.getline(buff, sizeof(buff) - 1);
    listArray[index].zip = atoi(buff);
```

см. продолжение

Листинг 2.4. (продолжение)

```
37:     index++;
38:     cout << endl;
39:   }
40:   while (index < 3);
41:   //
42:   // Очистка экрана
43:   //
44:   clrscr();
45:   //
46:   // Вывод трех записей
47:   //
48:   for (int i = 0; i < 3; i++) {
49:     displayRecord(i, listArray[i]);
50:   }
51:   //
52:   // Выбор записи для изменения
53:   //
54:   cout << "Choose a record: ";
55:   char rec;
56:   //
57:   // Можно выбрать только 1, 2 или 3
58:   //
59:   do {
60:     rec = getch();
61:     rec -= '1';
62:   } while (rec < 0 || rec > 2);
63:   //
64:   // Копирование выбранной записи во временную переменную
65:   //
66:   mailingListRecord temp = listArray[rec];
67:   clrscr();
68:   cout << endl;
69:   //
70:   // Вывод выбранной записи
71:   //
72:   displayRecord(rec, temp);
73:   getch();
74:   return 0;
75: }
76: void displayRecord(int num, mailingListRecord mlRec)
77: {
78:   cout << "Record " << num + 1 << ":" << endl;
79:   cout << "Name:     " << mlRec.firstName << " ";
80:   cout << mlRec.lastName;
81:   cout << endl;
82:   cout << "Address:  " << mlRec.address;
83:   cout << endl << "          ";
84:   cout << mlRec.city << ", ";
85:   cout << mlRec.state << "  ";
86:   cout << mlRec.zip;
87:   cout << endl << endl;
88: }
```

Листинг 2.5. STRUCTUR.H

```
 1: #ifndef _STRUCTUR_H
 2: #define _STRUCTUR_H
 3: struct mailingListRecord {
 4:    char firstName[20];
 5:    char lastName[20];
 6:    char address[50];
 7:    char city[20];
 8:    char state[5];
 9:    int zip;
10: };
11: #endif
```

В этой программе есть несколько новых для вас вещей, а также дополнений к тому, что уже было изучено.

Во-первых, здесь использована функция getline() из класса cin для ввода данных пользователем (см., например, строку 19). Я сделал это из-за того, что оператор >> реагирует на пробел не совсем так, как нам нужно. Второй параметр функции getline() определяет максимальное количество символов, помещаемых в буфер (в данном случае буфером является член структуры mailingListRecord). Это число выбирается таким образом, чтобы предотвратить выход за границу массивов структуры. Оператор sizeof() используется для определения размера массива, а следовательно, и буфера.

Функция atoi() в строке 36 также встречается вам впервые. Она принимает строку символов и преобразует ее в целое число. Это необходимо для преобразования текста в поле ZIP-кода (который вводится пользователем как строка цифр) в целое число, которое можно сохранить в члене zip структуры mailingListRecord.

Функция displayRecord(), которая начинается со строки 76, имеет два параметра. Первый из них, num, целого типа и содержит номер выводимой записи. Эта переменная используется только для вывода на экран номера записи. В строке 78 я увеличиваю num на 1, чтобы номера на экране начинались с 1, а не с 0. Второй параметр функции displayRecord() является экземпляром структуры mailingListRecord. Внутри функции я использую локальную копию передаваемой структуры для вывода ее содержимого на экран.



В данном случае я передаю структуру mailingListRecord по значению (by value). Это означает, что при каждом вызове функции displayRecord() создается копия этой структуры, которая и передается функции. Это не очень эффективно, т.к. требует дополнительных затрат памяти и времени для копирования структуры при каждом вызове функции. Гораздо лучше передавать структуру по ссылке (by reference), но я еще не рассказывал об этом, поэтому в данной программе используется передача по значению. Вы узнаете о передаче по ссылке завтра, когда мы будем обсуждать функции в Си++.

Обратите внимание, что функция displayRecord() вызывается в цикле for при выводе всех записей (строка 49), а затем снова из основного тела программы для вывода выбранной записи (строка 72). Именно по этой причине для вывода записи была написана отдельная функция. Используя функцию, я избежал ненужного дублирования кода.



Всякий раз, когда вы замечаете, что определенный фрагмент кода повторяется в программе несколько раз, подумайте, не лучше ли оформить его в виде функции. Сделав это, вы сможете затем в нужном месте просто вызывать функцию.

В этой программе есть еще один фрагмент, который заслуживает внимания. Посмотрите на цикл do-while, который начинается со строки 59:

```
do {
  rec = getch();
  rec -= 49;
} while (rec < 0 || rec > 2);
```

Сначала с помощью функции getch() с клавиатуры вводится символ. Как вы помните, я использую getch() в конце программ, чтобы предотвратить их преждевременное завершение, и при этом игнорирую возвращаемое значение. Вообще, функция getch() возвращает ASCII-код нажатой клавиши. Поскольку ASCII-код клавиши 1 равен 49, для получения значения индекса нужно вычесть из него 49. Если пользователь нажимает 1, функция возвращает 49, 49 - 49 = 0, что соответствует индексу первого элемента массива. Если пользователь нажмет 2, результатом вычисления будет 1 (50 - 49), и т.д. В цикле do-while предполагается, что будет нажата клавиша 1, 2 или 3. При нажатии любой другой клавиши цикл будет повторяться, ожидая правильного ввода.

Наконец, я хочу обратить ваше внимание на строку 66 листинга 2.4:

```
mailingListRecord temp = listArray[rec];
```

Эта строка не является необходимой в данной программе, но я использовал ее в качестве иллюстрации. В ней создается экземпляр структуры mailingListRecord, которому присваивается значение одной из структур массива. Простое присваивание в данном случае оказалось возможно потому, что компилятор знает, как копировать одну структуру в другую. Он просто копирует все члены исходной структуры в соответствующие члены вновь созданного экземпляра.



До этого момента описание структур велось в соответствии с правилами, принятыми в языке Си. В Си++ структуры используются таким же образом, как и в Си, но Си++ допускает включать в них наряду с членами-данными также и функции. Фактически, структуры в Си++ являются классами, все члены которых имеют открытый доступ. Вы поймете это позже, когда я буду рассказывать о классах.

Теперь вы узнали о структурах. Скорее всего, вы не будете часто использовать структуры в своих программах. Тем не менее, важность этого раздела состоит в том, что он послужит своего рода примером при изучении классов на третий день, «Указатели, ссылки и функции в Си++».

Заключение

Эта глава содержит существенные сведения о некоторых основных операторах Си++. Все, что представлено здесь, потребуется вам при программировании в C++Builder. Сначала вы изучили различные типы циклов в Си++; затем узнали об операторе switch и его использовании. Я немного

рассказал об областях видимости и их значении для ваших переменных. В конце главы вы узнали, что такое структуры и увидели, как их можно использовать в программах. Завтра нам предстоит взяться за довольно трудный материал.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Сколько уровней вложенности может иметь оператор if?

Формально ограничений нет. Существует, однако, одно практическое ограничение. При большом числе вложенных if становится очень трудно следить за фигурными скобками!

Будет ли цикл автоматически завершен, если что-то окажется неправильно?

Нет. Если вы случайно написали бесконечный цикл, он будет выполняться до тех пор, пока вы каким-либо способом его не прервете. Вы можете остановить программу, вызвав менеджер задач Windows (или окно завершения программы) и сняв ошибочную задачу. Если программа запущена из IDE C++Builder, можно выбрать для этого пункт Run | Reset Program в главном меню.

Должен ли оператор switch обязательно иметь секцию default?

Нет. Секция default является дополнительной.

Могу ли я иметь несколько переменных с одним и тем же именем?

Да, обеспечив им разные области видимости. Вы не можете объявить две переменные x в одном блоке, однако ничто не мешает вам иметь глобальную переменную x и локальную переменную с тем же именем.

Можно ли использовать структуру непосредственно, не создавая объектов?

Нет. До использования структуры вы должны создать экземпляр этой структуры и обращаться к структуре через соответствующую переменную.

Вопросы

1. Какие операторы выполняются в том случае, если условное выражение в операторе if принимает значение true?
2. Каково назначение трех параметров в операторе цикла for?
3. Чем отличаются циклы while и do-while, кроме синтаксиса?
4. Что делают операторы break и continue?
5. Что такое глобальная переменная?

6. Может ли структура содержать данные разных типов (char, int, long и т.д.)?
7. Как осуществляется доступ к членам структуры?
8. Возможно ли иметь массив структур?

Упражнения

1. Напишите программу, которая считает от 200 до 300 с шагом 5 и выводит результаты на экран.
2. Напишите программу, которая запрашивает у пользователя номер дня недели и затем выводит название этого дня, используя оператор switch.
3. Можете ли вы объяснить, что делают \b и \n в этой строке из листинга 2.2:

   ```
   cout << "\b! \nYipee!";
   ```

   (Указание: найдите в справочном файле «C++ Programmer's Guide» раздел, посвященный esc-последовательностям (esc-sequences).)
4. Создайте структуру, содержащую информацию о служащих фирмы. Включите туда имя, фамилию, адрес, дату принятия на работу, а также член, информирующий о наличии страховки.

Указатели, ссылки и функции в Си++



Кент Рейсдорф

«Не бойся, я держу тебя.» Помните, как вам говорили эти слова, когда вы учились ездить на велосипеде? Язык Си++ часто не прощает ошибок. Благодаря этой главе вы освоитесь с такими понятиями Си++, мимо которых проходит большинство людей. Хотя я не смогу поднять вас на ноги, если вы упадете, я по крайней мере обозначу некоторые из препятствий, о которые вы можете споткнуться по дороге. Сегодня вы изучите:

- Указатели
- Ссылки
- Операторы new и delete
- Функции в Си++

Указатели

Указатели — одна из самых трудных для понимания областей языка Си++. Но при этом они входят в число мощнейших инструментов Си++. Моя задача в этом разделе заключается не столько в том, чтобы дать вам строгое описание указателей, а скорее в том, чтобы научить вас пользоваться указателями при написании программ с помощью C++Builder. Итак, что такое указатель? Это переменная, которая содержит адрес другой переменной. Видите, все довольно просто! Поскольку указатель содержит адрес другой переменной, говорят, что он «указывает» на эту переменную. Это называется косвенной адресацией, т.к. указатель не связан напрямую с данными, хранящимися в памяти.



Указатель (pointer) — это переменная, которая содержит адрес другой переменной.



С помощью указателя осуществляется так называемая *косвенная адресация (indirection)*, т.к. он не связан напрямую с данными в памяти.

Давайте рассмотрим пример. Ранее мы говорили о массивах. Допустим, у вас есть массив чисел типа int. Вы можете обращаться к отдельным элементам массива, используя индексный оператор:

```
int array[] = { 5, 10, 15, 20, 25 };
int someVariable = array[3];   // число 20
```

Вы можете с той же целью использовать указатель:

```
int array[] = { 5, 10, 15, 20, 25 };
int* ptr = array;
int someVariable = ptr[3];
```

В этом примере указателю ptr присваивается адрес начала массива в памяти. Обратите внимание, что указатель имеет определенный тип (указатель на int) и при его объявлении использован *оператор разыменования* (символ *). Вы можете объявить указатель на любой из встроенных типов данных (int, char, long, short и т.д.), а также указатель на объект (структуру или класс). После присваивания указатель содержит адрес начала массива и таким образом указывает на массив.



Имя массива при использовании без индексного оператора возвращает адрес первого элемента массива в памяти. Иными словами, переменная с именем массива представляет собой указатель на начало массива. Это позволяет присваивать массив указателю, как в предыдущем примере.

После присваивания значения указателю ptr его можно использоваться наравне с именем массива. Вы можете удивленно спросить: «Но зачем это нужно?». Действительно, в данном примере использование указателя не дает никаких преимуществ. Реальные достоинства указателей проявляются при создании динамических объектов. В этом случае указатель необходим для доступа к объекту. Я вернусь к этой теме позже, когда буду рассказывать о двух способах создания переменных и объектов.

Локальное и динамическое распределение памяти

До сих пор во всех программах-примерах использовалось локальное размещение объектов — это значит, что память для переменных или объектов выделялась в программном стеке.



Локальное распределение (local allocation) означает, что память для переменной или объекта выделяется в программном стеке.



Стек (stack) — это область оперативной памяти, резервируемая программой при запуске.

Вся память, необходимая программе для создания локальных переменных, вызовов функций и прочих действий, берется из стека. Эта память распределяется по мере необходимости и освобождается, когда потребность в ней исчезает. Обычно это происходит при входе в функцию или локальный блок. Память для локальных переменных, используемых функцией, распределяется при вызове этой функции. Когда функция завершает работу, вы-

деленная ей память освобождается. Все эти действия происходят автоматически; вы не должны задумываться над тем, как и когда выделяется или освобождается память.

Локальное распределение имеет свои плюсы и минусы. С одной стороны, память может быть выделена в стеке очень быстро. С другой стороны, стек имеет фиксированный размер, который не может быть изменен во время работы программы. Если стек переполняется, начинают происходить странные явления. Программа может зависнуть, начать работать нестабильно, или при нормальной на первый взгляд работе компьютер зависает после окончания программы. В 32-разрядной среде переполнение стека доставляет меньше неприятностей по сравнению с 16-разрядной, но все же эту возможность необходимо принимать во внимание.

Для переменных встроенных типов и небольших массивов вполне достаточно использовать локальное распределение. Но если вы собираетесь работать с большими массивами, структурами или классами, имеет смысл прибегнуть к динамическому распределению памяти из «кучи». Эта память складывается из свободной оперативной памяти и свободного пространства на жестком диске. Иными словами, в типичной Windows-системе вы легко можете иметь до 100 Мб динамически распределяемой памяти. Достоинство такого подхода — практически неограниченный размер виртуальной памяти, доступной вашим программам. Недостаток заключается в том, что использование динамически распределяемой памяти требует некоторых накладных расходов и обращение к ней происходит немного медленнее, чем к памяти из стека. В большинстве программ эти накладные расходы можно не принимать во внимание. Работа с динамической памятью предъявляет также повышенные требования к программисту — не настолько большие, как вы можете подумать, но все же.



Динамическое распределение (dynamic allocation) означает, что память для объектов выделяется в «куче».



Под *«кучей» (heap)* в Windows-программах понимается вся виртуальная память компьютера.

Динамическое размещение объектов и указатели



В программах на Си++ динамическое распределение памяти осуществляется с помощью оператора new.

Я намеревался рассказать об операторе new немного позже, но вам нужен небольшой пример, чтобы можно было продолжить обсуждение указателей. В предыдущей главе я рассказывал о структурах и использовал в качестве примера структуру mailingListRecord. Размещение структуры в стеке выглядит следующим образом:

```
mailingListRecord listArray;
strcpy(listArray.firstName, "Ian");
strcpy(listArray.lastName, "Spencer");
// и т.д.
```

Так я поступал раньше, рассказывая о структурах. Теперь вместо локального распределения используем динамическое:

```
mailingListRecord* listArray;
listArray = new mailingListRecord;
strcpy(listArray->firstName, "Ian");
strcpy(listArray->lastName, "Spencer");
// и т.д.
```

В первой строке объявляется указатель на структуру mailingListRecord. Следующая строка инициализирует этот указатель, динамически размещая новый экземпляр структуры mailingListRecord. Таким образом осуществляется динамическое создание и использование объектов в Си++.

Назад к нашей программе

Теперь вы видите, какую роль играют указатели. При динамическом создании объекта оператор new возвращает указатель на этот объект, размещенный в памяти. Указатель необходим вам для использования объекта. На рис. 3.1 показано, как указатель ссылается на объект в памяти. Обратите внимание, что несмотря на размещение объекта в динамической памяти, сам указатель является локальной переменной и находится в стеке.

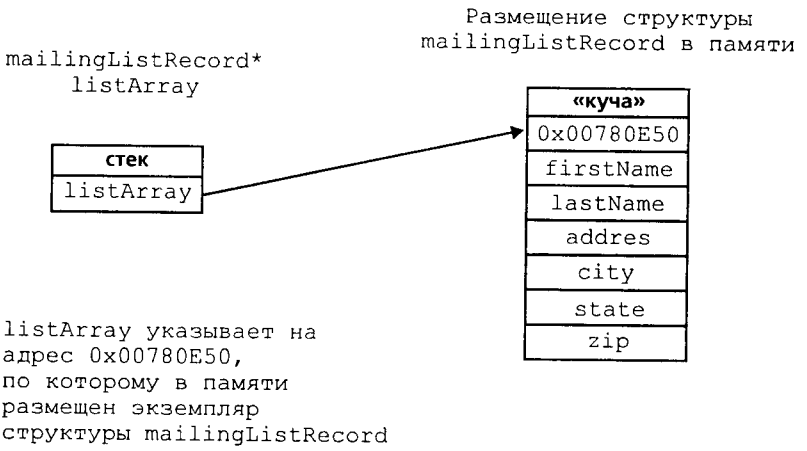


Рис. 3.1. Указатель на объект в памяти

Давайте вернемся к рассмотренному ранее фрагменту кода:

```
mailingListRecord* listArray;
listArray = new mailingListRecord;
strcpy(listArray->firstName, "Ian");
strcpy(listArray->lastName, "Spencer");
// и т.д.
```

В третьей строке для доступа к члену firstName вместо оператора доступа (.) используется оператор *косвенного доступа* (->). (Оператор доступа к члену структуры рассматривался вчера в разделе «Структуры». Далее я буду называть его оператором прямого доступа.) При динамическом создании объекта вы должны использовать для доступа к его членам оператор косвенного доступа.

Создание динамического массива структур выполняется немного сложнее. Вот пример размещения массива в стеке:

```
mailingListRecord listArray[3];
listArray[0].zip = 57441;
```

и динамического размещения:

```
mailingListRecord* listArray[3];
for (int i = 0; i < 3; i++)
  listArray[i] = new mailingListRecord;
listArray[0]->zip = 57441;
```

Обратите внимание, что я должен создавать новый экземпляр структуры отдельно для каждого элемента массива. Для доступа к членам структур используется оператор косвенного доступа в сочетании с индексным оператором.



Неинициализированные указатели содержат случайные значения, так же, как и любые другие неинициализированные переменные. Попытка использовать неинициализированный указатель может привести к хаосу в программе. Во многих случаях указатели инициализируются одновременно с их объявлением:

```
MyArray* array = new MyArray;
```

Иногда, однако, вам нужно будет объявить указатель и инициализировать его где-то далее по ходу программы. Если вы попытаетесь использовать этот указатель до инициализации, он будет содержать адрес некоторого случайного места в памяти, изменение которого может привести к непредсказуемым проблемам. Зачастую изменение памяти не проявляется немедленно, порождая как бы случайные сбои. Для безопасности вам следует при объявлении инициализировать указатель нулевым значением:

```
MyArray* array = 0;
```

При попытке использовать NULL-указатель (это может быть любой указатель, установленный в 0, или NULL), Windows немедленно выдаст сообщение о нарушении прав доступа или общем нарушении защиты. Это не самое приятное, что можно ожидать, но несомненно меньшее из двух зол. Гораздо лучше сразу получить сообщение об ошибке, чем столкнуться затем с проблемами непонятного происхождения.

Разыменование указателей

Необходимость разыменования, то есть получения содержимого памяти, на которое ссылается указатель, возникает довольно часто. Рассмотрим следующий пример:

```
int x = 20;
int* ptrx = &x;
//...
int z = *ptrx;
```

Могу представить, в каком вы сейчас недоумении. Однако все не так уж сложно. В первой строке этого примера объявлена переменная x типа int и ей присвоено значение 20. В следующей строке объявлен указатель на

int, которому присвоен адрес переменной x. Это сделано с помощью *оператора взятия адреса* (&). В данном примере этот оператор сообщает компилятору, что нужно взять адрес переменной x, а не ее значение. После присваивания ptrx содержит адрес, по которому расположена в памяти переменная x. Далее по ходу программы нам нужно получить значение объекта, на который указывает ptrx. Вы могли попробовать сделать так:

```
int z = ptrx; // ошибка!
```

Однако это не сработает, потому что вы пытаетесь присвоить переменной адресное значение. Встретив такую строку, компилятор выдаст сообщение об ошибке Cannot convert int* to int (Невозможно преобразовать int* в int). Переменные z и ptrx имеют различные типы, поэтому необходимо использовать оператор разыменования:

```
int z = *ptrx;
```

Эту операцию можно рассматривать как обратную по отношению к операции взятия адреса. Здесь вам не нужно значение самой переменной ptrx, т.к. оно представляет собой адрес. Вместо этого вам нужно значение объекта, расположенного по данному адресу. Оператор разыменования сообщает компилятору, что нужно взять значение объекта, на который указывает ptrx, а не само значение ptrx.



Разыменование (dereferencing) означает получение содержимого памяти, на которое ссылается указатель.



Как видите, оператор разыменования используется как для объявления указателя (int* x;), так и для взятия значения объекта, на который ссылается указатель (int z = *x;). По контексту можно определить, как должен использоваться оператор в каждом конкретном случае. Вам не нужно беспокоиться о том, что компилятор не поймет ваших намерений.



Синтаксис Си++ допускает различные стили написания программ. Я предпочитаю ставить оператор разыменования сразу за типом данных при объявлении указателя и непосредственно перед указателем при взятии значения объекта:

```
int* x;
SomeClass* aClass = new SomeClass;
char* s = new char[256];
int z = *x;
SomeClass temp = *aClass;
```

Другие предпочитают ставить оператор перед именем переменной:

```
int *x;
// и даже более того...
int * x;
```

Я считаю, что моя запись более наглядна, но другие могут, вероятно, привести аргументы в пользу своего способа. Выберите один из вариантов, который кажется вам лучше, и придерживайтесь его в дальнейшем.

Собираем все вместе

Давайте попробуем применить на практике все то, что вы узнали из предыдущих разделов. Я возьму программу MAILLIST из второго дня, «Операторы управления и структуры в Си++», и использую в ней динамическое распределение памяти. Для этого потребуется внести несколько изменений. Прежде всего, просмотрите модифицированную программу (листинг 3.1), а затем я объясню сделанные изменения.

Листинг 3.1. POINTER.CPP

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#pragma hdrstop
#include "structur.h"
void displayRecord(int, mailingListRecord mlRec);
int main(int, char**)
{
  //
  // Создание массива указателей
  // на структуры mailingListRecord
  //
  mailingListRecord* listArray[3];
  //
  // Создание объекта для каждого элемента массива
  //
  for (int i = 0; i < 3; i++)
    listArray[i] = new mailingListRecord;
  cout << endl;
  int index = 0;
  //
  // Ввод трех записей
  //
  do {
    cout << "First Name: ";
    cin.getline(listArray[index]->firstName,
      sizeof(listArray[index]->firstName) - 1);
    cout << "Last Name: ";
    cin.getline(listArray[index]->lastName,
      sizeof(listArray[index]->lastName) - 1);
    cout << "Address: ";
    cin.getline(listArray[index]->address,
      sizeof(listArray[index]->address) - 1);
    cout << "City: ";
    cin.getline(listArray[index]->city,
      sizeof(listArray[index]->city) - 1);
    cout << "State: ";
    cin.getline(listArray[index]->state,
      sizeof(listArray[index]->state) - 1);
    char buff[10];
    cout << "Zip: ";
    cin.getline(buff, sizeof(buff) - 1);
    listArray[index]->zip = atoi(buff);
    index++;
    cout << endl;
```

см. продолжение

Листинг 3.1. (продолжение)

```
46:    }
47:    while (index < 3);
48:    //
49:    // Вывод трех записей
50:    //
51:    clrscr();
52:    //
53:    // Для передачи объекта функции displayRecord
54:    // необходимо разыменовать указатель
55:    //
56:    for (int i = 0; i < 3; i++) {
57:      displayRecord(i, *listArray[i]);
58:    }
59:    //
60:    // Выбор записи для изменения
61:    //
62:    cout << "Choose a record: ";
63:    char rec;
64:    do {
65:      rec = getch();
66:      rec -= 49;
67:    } while (rec < 0 || rec > 2);
68:    //
69:    // Копирование выбранной структуры во временную переменную.
70:    // Здесь также необходимо разыменование
71:    //
72:    mailingListRecord temp = *listArray[rec];
73:    clrscr();
74:    cout << endl;
75:    //
76:    // Вывод выбранной записи
77:    //
78:    displayRecord(rec, temp);
79:    getch();
80:    return 0;
81: }
82: void displayRecord(int num, mailingListRecord mlRec)
83: {
84:    cout << "Record " << num + 1 << ":" << endl;
85:    cout << "Name:      " << mlRec.firstName << " ";
86:    cout << mlRec.lastName;
87:    cout << endl;
88:    cout << "Address:   " << mlRec.address;
89:    cout << endl << "           ";
90:    cout << mlRec.city << ", ";
91:    cout << mlRec.state << "  ";
92:    cout << mlRec.zip;
93:    cout << endl << endl;
94: }
```



Прежде всего, в строке 13 я объявляю массив listArray как массив указателей. Затем я создаю объекты для каждого элемента массива, используя цикл for в строках 17 и 18. После этого я заменяю оператор прямого доступа (.) на оператор косвенного доступа(->). Я должен также

разыменовать указатели в строках 57 и 72, т.к. нельзя использовать вместо самого объекта его адрес. Обратите внимание, что функция displayRecord (начинающаяся со строки 82) не изменилась. Структура mailingListRecord по-прежнему передается по значению, поэтому код функции не требует изменения.

Если вы раньше работали с Си++, то могли заметить ошибку в программе. До конца этой главы вы узнаете о ней.

Ссылки



Ссылка (reference) — это специальный тип указателя, который позволяет работать с указателем как с обычным объектом.

Ссылки, как и указатели, могут вызвать путаницу. Ссылка объявляется с помощью специального оператора ссылки. Для этого оператора используется тот же символ (&), что и для оператора взятия адреса (не беспокойтесь, компилятор знает, как их различить). Я уже говорил, что ссылка позволяет трактовать указатель как объект. Рассмотрим пример:

```
MyStruct* pStruct = new MyStruct;
MyStruct& ref = *pStruct;
ref.X = 100;
```

Обратите внимание, что со ссылкой используется оператор прямого доступа. Теперь можно избавиться от этих надоедливых операторов -> ! Хотя вы не будете использовать ссылки в большом количестве, они могут оказаться очень удобны, когда в них возникнет необходимость. Между тем, рассмотренный фрагмент стоит немного сжать. В реальной программе я написал бы его следующим образом:

```
MyStruct& ref = *new MyStruct;
ref.X = 100;
```

Это не очень наглядно, но работает точно так же, как и в первом примере. Подобные комбинированные операторы часто используются и позволяют избежать ненужной избыточности.

Давайте еще раз обратимся к примеру MAILLIST. Теперь я изменю его, введя ссылку в цикл do-while. (В действительности я буду изменять пример POINTER из листинга 3.1.) Новая программа приведена в листинге 3.2.

Листинг 3.2. REFERENC.CPP

```
 1: #include <iostream.h>
 2: #include <conio.h>
 3: #include <stdlib.h>
 4: #pragma hdrstop
 5: #include "structur.h"
 6: void displayRecord(int, mailingListRecord mlRec);
 7: int main(int, char**)
 8: {
 9:    cout << endl;
10:    //
11:    // Создание массива указателей на структуры mailingListRecord
12:    //
```

см. продолжение

Листинг 3.2. (продолжение)

```
13:   mailingListRecord* listArray[3];
14:   //
15:   // Создание объектов для каждого элемента массива
16:   //
17:   for (int i = 0; i < 3; i++)
18:     listArray[i] = new mailingListRecord;
19:   int index = 0;
20:   //
21:   // Ввод трех записей
22:   //
23:   do {
24:     // Создание ссылки на текущую запись
25:     mailingListRecord& rec = *listArray[index];
26:     cout << "First Name: ";
27:     cin.getline(rec.firstName, sizeof(rec.firstName) - 1);
28:     cout << "Last Name: ";
29:     cin.getline(rec.lastName, sizeof(rec.lastName) - 1);
30:     cout << "Address: ";
31:     cin.getline(rec.address, sizeof(rec.address) - 1);
32:     cout << "City: ";
33:     cin.getline(rec.city, sizeof(rec.city) - 1);
34:     cout << "State: ";
35:     cin.getline(rec.state, sizeof(rec.state) - 1);
36:     char buff[10];
37:     cout << "Zip: ";
38:     cin.getline(buff, sizeof(buff) - 1);
39:     rec.zip = atoi(buff);
40:     index++;
41:     cout << endl;
42:   }
43:   while (index < 3);
44:   //
45:   // Вывод трех записей
46:   //
47:   clrscr();
48:   //
49:   // Для передачи объекта функции displayRecord
50:   // необходимо разыменовать указатель
51:   //
52:   for (int i = 0; i < 3; i++) {
53:     displayRecord(i, *listArray[i]);
54:   }
55:   //
56:   // Запрос номера записи
57:   //
58:   cout << "Choose a record: ";
59:   char rec;
60:   do {
61:     rec = getch();
62:     rec -= 49;
63:   } while (rec < 0 || rec > 2);
64:   //
65:   // Копирование выбранной записи во временную переменную.
66:   // Здесь также необходимо разыменование указателя
67:   //
68:   mailingListRecord temp = *listArray[rec];
69:   clrscr();
```

```
70:    cout << endl;
71:    //
72:    // Вывод выбранной записи
73:    //
74:    displayRecord(rec, temp);
75:    getch();
76:    return 0;
77: }
78: void displayRecord(int num, mailingListRecord mlRec)
79: {
80:    cout << "Record " << num + 1 << ":" << endl;
81:    cout << "Name:      " << mlRec.firstName << " ";
82:    cout << mlRec.lastName;
83:    cout << endl;
84:    cout << "Address:   " << mlRec.address;
85:    cout << endl << "           ";
86:    cout << mlRec.city << ", ";
87:    cout << mlRec.state << "   ";
88:    cout << mlRec.zip;
89:    cout << endl << endl;
90: }
```



Единственное изменение внесено в цикл do-while — объявлена ссылка на структуру mailingListRecord. При каждом проходе цикла ссылка указывает на следующий элемент массива. Обратите внимание, что я избавился от оператора косвенного доступа и заменил его оператором прямого доступа. Как я уже говорил, ссылка позволяет вам работать с указателем как с объектом. В данном примере это позволило слегка сократить код и улучшить его читаемость. Новая программа по-прежнему содержит одну ошибку. Я исправлю ее в конце главы.

Может показаться, что ссылки предпочтительнее указателей, но это не так. Из-за некоторых специфических свойств ссылки нельзя использовать во многих случаях. Например, ссылку нельзя сначала объявить, а затем присвоить ей значение. Инициализация должна быть выполнена одновременно с объявлением. Следующий фрагмент приведет к выдаче сообщения об ошибке:

```
MyStruct* pStruct = new MyStruct;
MyStruct& ref;
ref = *pStruct;
ref.X = 100;
```

Ссылки нельзя установить в 0 или NULL, как указатели. Это означает, что вы должны следить за тем, чтобы не применить к ссылке оператор delete дважды. Ссылки и указатели часто выполняют одни и те же функции, но не обладают универсальностью. В одних ситуациях удобнее использовать ссылку, в других — указатель.

Передача аргументов функции с использованием ссылок и указателей

Ранее я говорил о передаче функции аргументов по значению. Я также говорил, что для структур и классов передача по ссылке обычно предпочти-

тельнее. По ссылке можно передать любой объект. Это относится не только к структурам и классам, но и к простым типам данных, таким, как int или long. При передаче аргументов по значению, функции передается не сам объект, а его копия. При передаче по ссылке функция получает *указатель на объект*. Это имеет два основных следствия. Во-первых, функция может изменить объект, переданный по ссылке. Во-вторых, исключаются накладные расходы, связанные с созданием копии объекта.

Возможность изменения объекта функцией является наиболее важным следствием передачи по ссылке. Возьмем в качестве примера следующий код:

```
void IncrementPosition(int& xPos, int& yPos)
{
  xPos++;
  yPos++;
}
int x = 20;
int y = 40;
IncrementPosition(x, y);
// теперь x равно 21, а y равно 41
```

Обратите внимание, что после завершения работы функции оба аргумента увеличились на единицу. Это произошло потому, что функция изменяет сами объекты через указатели (вспомните, что ссылка — это разновидность указателя).



Как вы помните, функция может возвращать только одно значение. Передача же аргументов по ссылке эквивалентна возврату нескольких значений. В этом случае функция по-прежнему возвращает одно значение, но объекты, переданные по ссылке, можно изменить. Таким образом, фактически возвращаемых значений может быть несколько.

Как я говорил, еще одна причина для передачи значений по ссылке состоит в исключении накладных расходов в виде копирования объектов при каждом вызове функции. Копирование простых типов данных не приводит к реальным издержкам, но для структур и классов их нужно принимать во внимание. Структуры в любом случае следует передавать по ссылке, как в следующем примере:

```
// передача структуры по ссылке
void someFunction(MuStructure& s)
{
  // какие-то действия с s
  return;
}
MyStructure myStruct;
// какие-то действия, затем...
someFunction(myStruct);
```

Обратите внимание, что вызов функции при передаче аргумента по значению и по ссылке выглядит одинаково.

Вы заметили одну потенциальную проблему, связанную с передачей по ссылке? При использовании ссылок исключаются затраты на копирование, но теперь объект может быть изменен функцией. Иногда такое изменение недопустимо. Так как же быть, если вы хотите передать объект по ссылке

и одновременно предохранить его от изменения? Читайте дальше, и вы узнаете об этом.

Ключевое слово const

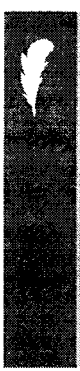


Ключевое слово const позволяет вам объявлять переменную как константу.

После объявления с модификатором const переменная не может быть изменена. Таким образом, решение заключается в том, чтобы передать объект по ссылке и сделать его константой:

```
void SomeFunction(const MyStruct& s)
{
  // какие-то действия с s
  return;
}
MyStructure myStruct;
//...
someFunction(myStruct);
```

Теперь вы можете спокойно передавать по ссылке объект, не беспокоясь, что он будет изменен функцией. Обратите внимание, что при этом вызов функции остается таким же. Изменяется с помощью ключевого слова const только ее определение (и объявление).



При попытке изменить объект, объявленный как const, компилятор выдаст сообщение об ошибке Cannot modify a const object (Невозможно изменить const объект). Именно такое сообщение будет выдано при компиляции следующего фрагмента:

```
void someFunction(const MyStruct& s)
{
  s.dataMember = 100; // константу изменить невозможно
  return;
}
```

Объявив объект как const, вы можете быть уверены, что компилятор не позволит вам его изменить.

Обратите внимание, что объект является константой только внутри функции. Его можно изменить как до, так и после вызова функции (конечно, если он не был изначально объявлен как const).

Передача значений с помощью указателя принципиально не отличается от передачи по ссылке. С точки зрения синтаксиса — это лишняя головная боль, поэтому передача по ссылке более предпочтительна. Давайте возьмем функцию IncrementPosition() из первого примера в этом разделе и изменим ее так, чтобы передача аргументов осуществлялась через указатели:

```
void IncrementPosition(int* xPos, int* yPos)
{
  *xPos++; // разыменование, затем увеличение
  *yPos++;
}
```

Обратите внимание, что указатель необходимо разыменовать перед увеличением значения переменной. В большинстве ситуаций передача по ссылке будет лучшим решением стоящей перед вами задачи, но в отдельных случаях без указателей обойтись трудно. При передаче массива типа char вместо ссылки обычно используется указатель, т.к. он взаимозаменяем с именем массива. В данном случае указатель оказывается удобнее, чем ссылка.

Операторы new и delete

До этого момента я рассказывал главным образом о тех элементах Си++, которые пришли из Си. Теперь мы будем рассматривать возможности, присущие только Си++. Два оператора, new и delete, играют в Си++ очень важную роль.

Как упоминалось в предыдущем разделе, динамическое распределение памяти в программах на Си++ осуществляется оператором new. Для освобождения памяти используется оператор delete. Даже если вы программировали раньше на Си, вы не могли ощутить простоты применения new и delete. В программах на Си для динамического распределения памяти нужно использовать функции malloc(), calloc(), realloc() и free(). При программировании в Windows ситуация еще больше усложняется — существует целый набор функций для локального и глобального распределения памяти. Хотя их использование не так уж сложно, всегда существует вероятность запутаться. Си++ устраняет все проблемы с помощью операторов new и delete.

Оператор new

Вы уже видели new в действии, поэтому придется кое-что повторить. Как говорилось ранее, память можно распределять локально (в стеке) или динамически (в «куче»). Следующий фрагмент содержит примеры размещения двух символьных массивов. В одном случае память выделяется в стеке (локальное распределение), а в другом — из «кучи» (динамическое распределение):

```
char buff[80];
char* bigBuff = new char[4096];
```

Для первого массива размер буфера мал, поэтому не имеет большого значения, где его размещать. Второй массив состоит уже из относительно большого числа элементов, поэтому для экономии стековой памяти его лучше разместить в «куче». Обращение к локальным и динамическим массивам (как вы помните, строка — это всего лишь массив типа char) происходит одинаковым образом. Иными словами, они используют тот же самый синтаксис:

```
strcpy(buff, "Ricky Rat");
strcpy(bigBuff, "A very long string that goes on and on...");
//...
strcpy(bigBuff, buff);
```

Вспомните, что имя локального массива само по себе является указателем на начало памяти, которую занимает массив. При динамическом распределении указателю также присваивается адрес первого элемента массива, поэтому обе формы записи взаимозаменяемы.

Если оператор new не может выделить запрашиваемую память, он возвращает NULL. Теоретически вы должны проверить указатель после вызова new и убедиться, что он имеет ненулевое значение:

```
char* buff = new char[1024];
if (buff) strcpy(buff, "Buteo Regalis");
else ReportError(); // что-то не в порядке
```

В действительности же, если оператор new не срабатывает в 32-разрядной Windows-программе, это говорит о неполадках в самой системе, и в этом случае ни ваша программа, ни какая-либо другая долго не проработают.

Если вы пытаетесь выделить очень большие фрагменты памяти (размером в несколько мегабайт) или распределяете память в критических участках программы, следует проверить значение указателя перед его дальнейшим использованием. При решении обычных задач можно, скорее всего, обойтись и без такой проверки.

Оператор delete

Вся выделенная память должна быть освобождена после использования. Для локальных объектов это происходит автоматически, и вам не нужно принимать специальные меры. Менеджер памяти по мере необходимости выделяет память для ваших объектов в стеке и освобождает ее, когда объекты выходят из области видимости (обычно при возврате из функции или завершении блока, в котором был объявлен объект). При динамическом распределении программист должен сам отвечать за освобождение памяти, выделенной с помощью оператора new.

Освобождение памяти, выделенной с помощью оператора new, осуществляется оператором delete.

Каждому new должен соответствовать свой delete. Если вы не освободите всю память, распределенную с помощью оператора new, ваша программа вызовет «утечку» памяти в системе. Будьте аккуратны в использовании пары new/delete.

Использование оператора delete крайне просто:

```
SomeObject* myObject = new SomeObject;
// использование myObject
delete myObject;  // до свидания!
```

Вот и все! Вам следует только знать некоторые особенности использования delete с указателями. Во-первых, нельзя применять оператор delete к указателю дважды, в противном случае может возникнуть нарушение защиты и прочие неприятности. Во-вторых, вполне допустимо применять delete к указателю, установленному в 0. Но чему это соответствует на практике? Сейчас объясню.

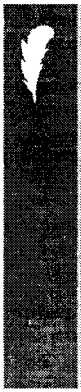
Иногда вы объявляете указатель, предполагая его дальнейшее использование, но не зная точно, будет ли он использован в данной программе. Например, вы создаете некоторый объект только в том случае, если пользо-

ватель выберет один из пунктов меню. Если этот пункт ни разу не был выбран, объект не создается. Пока все идет хорошо. Проблема состоит в том, что нужно освободить память только в случае создания объекта. Применение delete к инициализированному указателю опасно, т.к. вы не знаете, на какое место в памяти он указывает. Существуют два возможных решения.

Я говорил раньше, что стоит инициализировать указатели нулем, если вы не используете их немедленно. Для этого есть две причины. Об одной я уже говорил (неинициализированный указатель содержит случайное значение, что нежелательно). Вторая заключается в том, что к указателю со значением NULL можно применить delete независимо от того, был ли этот указатель ранее использован:

```
Monster* swampThing = 0;
// затем, при выходе из программы...
delete swampThing;
```

В этом случае вам не нужно думать о том, была ли когда-нибудь выделена память под объект. Использование delete безопасно, если указатель содержит адрес объекта или NULL.



Вы можете попасть в ситуацию, когда delete вызывается неоднократно для одного и того же объекта. Например, вы создаете объект в одной части программы и уничтожаете его в другой. Если по логике работы программы блок кода, в котором уничтожается объект, может быть ни разу не выполнен, вы уничтожаете объект также при завершении программы. Чтобы избежать повторного применения delete к указателю, возьмите за правило обнулять указатель после уничтожения объекта:

```
Monster* borg = new Monster;
// затем...
delete borg;
borg = 0;
```

Теперь повторный вызов delete безопасен — указатель содержит NULL.

Другой способ решения проблемы повторного уничтожения объекта — сравнение указателя с нулем до вызова delete:

```
if (swampThing) delete swampThing;
```

При этом предполагается, что вы установили указатель в 0 после уничтожения объекта в другой части программы. Можно использовать любой из способов; главное, чтобы вы не забывали это делать всякий раз, когда возможно повторное применение оператора delete к указателю.



Если вы используете ссылку при динамическом создании объекта, синтаксис оператора delete немного меняется. Вот пример:

```
MyStruct& ref = *new MyStruct;
ref.X = 100;
// затем...
delete &ref;
```

Вместе со ссылкой необходимо использовать оператор взятия адреса. Помните, что ссылку нельзя установить в 0, поэтому будьте внимательны, чтобы не применить оператор delete дважды.

Решение еще одной загадки

Вы поняли теперь, какая ошибка была в программах POINTER и REFERENC? Эти программы расходовали системную память. Я создавал массив структур, размещая их в «куче», и ни разу не освобождал выделяемую память. Чтобы исправить положение, нужно добавить несколько строк в самый конец программы:

```
getch(); // существующая строка
for (int i = 0; i < 3; i++)
  delete listArray[i];
```

Теперь программа работает абсолютно правильно. Я перебрал массив указателей и уничтожил каждую структуру.

new[] и delete[]

Создавая массив с помощью оператора new, вы на самом деле используете версию new[] этого оператора. Вам не обязательно знать подробности его работы, но вы должны уметь правильно уничтожать динамические массивы. Ранее я приводил пример динамического размещения символьного массива. Вот тот же самый фрагмент кода, в который добавлен оператор delete[]:

```
char buff[80];
char* bigBuff = new char[4096];
strcpy(buff, "Ricky Rat");
strcpy(bigBuff, "Some very long string.");
// затем...
delete[] bigBuff;
```

Обратите внимание, что оператор называется delete[], а не просто delete. Я не буду вдаваться в технические подробности того, что здесь происходит, но эта операция гарантирует правильное уничтожение всех элементов массива. Следите за тем, чтобы при использовании динамического массива память освобождалась с помощью оператора delete[].

Правила работы с указателями и динамическим распределением памяти

- Инициализируйте указатели нулем, если они не используются немедленно.
- Следите за тем, чтобы не применить оператор delete к указателю дважды.
- Можно без опасений применять оператор delete к указателю, имеющему значение 0 или NULL.
- Устанавливайте указатели в NULL или 0 после уничтожения объектов.
- Разыменовывайте указатель для получения объекта, на который он указывает.

Функции в Си++

Функции Си++ предоставляют программисту дополнительные возможности по сравнению с функциями Си. В этом разделе рассматривается следующее:

- Перегрузка функций
- Задание аргументов по умолчанию
- Функции-члены класса
- Подставляемые функции

Перегрузка функций

Си++ позволяет вам иметь несколько функций с одинаковым именем, но различными наборами параметров.



Перегрузка функции (function overloading) — это определение двух или более функций с одинаковым именем, но разными наборами параметров.



Функции, имеющие общее имя, называются *перегруженными (overloaded)*.

В первый день я приводил в качестве примера программу, которая содержала функцию multiply(). Эта функция принимала в качестве аргументов два целых числа, перемножала их и возвращала результат. Но как быть, если вы захотите перемножить два числа с плавающей точкой? В Си для этого понадобилось бы две функции:

```
// объявления функций для программы на Си
int multiplyInt(int num1, int num2);
float multiplyFloat(float num1, float num2);
short multiplyShort(short num1, short num2);
```

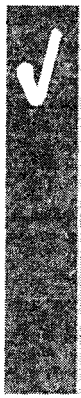
Не проще было бы иметь одну функцию multiply(), достаточно «умную», чтобы определить, когда вы хотите перемножить short, int или long? В Си++ это возможно благодаря перегрузке функций. Вот как выглядят объявления для перегруженных функций:

```
// объявления в Си++
int multiply(int num1, int num2);
float multiply(float num1, float num2);
short multiply(short num1, short num2);
```

Вы по-прежнему должны написать отдельные функции для каждого из этих объявлений, но теперь все функции используют одно и то же имя. Компилятор выберет нужную функцию, основываясь на аргументах, которые вы ей передаете. Например:

```
float x = 1.5;
float y = 10.5;
float result = multiply(x, y);
```

Компилятор видит, что функции передаются два числа типа float и вызывает соответствующую версию multiply(). Аналогичным образом, если передаются два целых числа, будет вызвана целочисленная версия multiply().



Работа перегруженных функций основана на различии списков параметров. Вы можете варьировать тип и/или количество аргументов, принимаемых функцией, но нельзя создавать перегруженные функции, различающиеся только типом возвращаемого значения. Например, следующий фрагмент не рассматривается как перегрузка функций:

```
int DoSomething();
void DoSomething();
```

Если вы попытаетесь откомпилировать программу, содержащую подобные строки, то получите сообщение об ошибке Type mismatch in redeclaration of 'DoSomething()' (Несоответствие типа в повторном объявлении 'DoSomething()'). Эти две функции должны различаться не только возвращаемым значением, чтобы быть перегруженными.



Работа компилятора с перегруженными функциями основана на процессе, называемом искажением имен (name mangling). Это означает, что для внутреннего представления функция получает имя, в котором отражен список ее параметров. После этого ссылка на функцию осуществляется по искаженному имени вместо простого имени, используемого в исходном тексте программы. Например, для функции multiply(), принимающей аргументы с плавающей точкой, искаженное имя будет выглядеть как multiply$qff.

Давайте немного отклонимся от основной темы и поговорим о некоторых вещах, которые необходимо учитывать при использовании перегруженных функций.

Приведение типов

Перегруженные функции работают превосходно, пока вы используете при их вызове правильные типы данных. Но что делать, если типы не совпадают? В этом случае необходимо использовать приведение типов.

Приведение (cast) — это операция, в результате которой компилятор временно изменяет тип используемых данных.

Приведение типов выглядит следующим образом:

```
float x = (float)10 * 5.5;
```

Встретив такую запись, компилятор будет рассматривать число 10 как имеющее плавающую точку (второе число автоматически интерпретируется как float, т.к. оно содержит дробную часть). Посмотрите на следующий фрагмент:

```
int anInt = 5;
float aFloat = 10.5;
float result = multiply(anInt, aFloat);
```

В этом случае вы получите сообщение об ошибке, поскольку типы передаваемых параметров не соответствуют указанным в объявлении функции. Фактически, компилятор сообщит, что он не может определить, какую из версий multiply() нужно вызвать. Ту же ошибку вызовет использование кода наподобие следующего:

```
int result = multiply(10, 10);
// 10 - это float, int или short?
```

Здесь компилятор не может разобраться, какой тип приписать числовым константам — float, int или short. Из подобных ситуаций существует два возможных выхода. Прежде всего, можно отказаться от использования констант при вызове функций. Если вы хотите перемножить целые числа, объявите две переменные типа int и передайте их функции:

```
int x = 10;
int y = 10;
int result = multiply(x, y);
```

Теперь неоднозначности нет, поскольку x и y имеют тип int. В простых ситуациях это, возможно, будет избыточным решением. Тогда вы можете использовать приведение типов, чтобы сообщить компилятору необходимые сведения:

```
int result = multiply((int)10, (int)10);
```

Теперь компилятор знает, что нужно интерпретировать числовые константы как имеющие тип int. Приведение используется также для временного изменения типа некоторой переменной. Давайте вернемся назад к первому примеру в этом разделе и изменим тип одного из аргументов, чтобы избежать несоответствия:

```
int x = 5;
float y = 10.5;
float result = multiply((float)x, y);
```

Здесь x имеет тип int, но вы приводите его к типу float, после чего компилятор успешно вызывает float-версию функции multiply().

В идеале вы должны писать перегруженные функции, для которых не существует неоднозначности в трактовке аргументов. В отдельных случаях это невозможно, и тогда приведение типов будет единственным выходом.

Аргументы, передаваемые по умолчанию



Функции в Си++ могут иметь *аргументы по умолчанию (default parameters)*, которые, как следует из названия, используются при отсутствии явного указания аргумента при вызове функции.

Функция, имеющая аргументы по умолчанию, должна выглядеть следующим образом:

```
// объявление, параметр 'eraseFirst' по умолчанию имеет значение false
void Redraw(bool eraseFirst = false);
// определение
void Redraw(bool eraseFirst)
{
  if (eraseFirst) {
    // стирание
  }
  // рисование
}
```

При вызове этой функции можно опустить аргумент. Если аргумент присутствует, то функция работает как обычно. Если аргумент не указан, будет

автоматически использован аргумент по умолчанию. В данном случае следующие две строки идентичны:

```
Redraw();
Redraw(false);
```

Обратите внимание, что если аргумент имеет значение по умолчанию, он может быть полностью опущен при вызове функции. Вы можете использовать обычные аргументы и аргументы по умолчанию совместно:

```
int PlaySound(char* name, bool loop = false, int loops = 10);
// вызов функции
int res;
res = PlaySound("chime.wav");          // одиночный звук
res = PlaySound("ding.wav", true);     // повтор звука 10 раз
res = PlaySound("bell.wav", true, 5); // повтор звука 5 раз
```

Аргументы по умолчанию полезны по многим причинам. Прежде всего, они облегчают вам жизнь. Функция может в 9 случаях из 10 вызываться с одним и тем же набором аргументов. Передавая их по умолчанию, вы сокращаете объем текста, который нужно набирать при каждом вызове функции.



Аргументы по умолчанию могут использоваться только в конце списка аргументов, передаваемых функции. Следующее объявление неправильно:

```
int MyFunction(int x, int y = 10, int t = 5, int z);
```

Чтобы компиляция прошла успешно, необходимо переместить аргументы по умолчанию в конец списка:

```
int MyFunction(int x, int z, int y = 10, int t = 5);
```

Если вы не сделаете этого, компилятор выдаст сообщение об ошибке.

Функции-члены класса



Как вы узнаете из этого раздела, классы могут содержать свои собственные функции. Такие функции называются *функциями-членами (member functions)*, т.к. они являются элементами класса.

Функции-члены класса подчиняются тем же правилам, что и обычные функции: они могут быть перегружены, иметь аргументы по умолчанию, принимать любое число аргументов, и т.д.

Функции-члены класса можно вызывать только через объекты того класса, которому они принадлежат. Для вызова функции-члена используется оператор прямого доступа (в случае локальных объектов) или оператор косвенного доступа (для динамически создаваемых объектов), точно так же, как это делалось при обращении к элементам структур. Например, пусть у вас есть класс Airplane, который используется для слежения за самолетом в программе-имитаторе полета. Этот класс, вероятно, должен иметь возможность получать текущую скорость самолета через функцию GetSpeed(). Следующий пример служит иллюстрацией того, как можно обращаться к функции GetSpeed() объекта Airplane:

```
Airplane plane; // создание экземпляра класса
int speed = plane.GetSpeed();
cout << "The airplane's current speed is " << speed << endl;
```

Здесь для вызова функции GetSpeed() использован оператор прямого доступа. Функции-члены класса определяются подобно обычным функциям; разница состоит в том, что перед именем функции указывается имя класса с оператором разрешения видимости. Например, определение GetSpeed() в исходном файле может выглядеть так:

```
int Airplane::GetSpeed()
{
    return speed; // speed является переменной-членом класса
}
```

В данном случае оператор разрешения видимости сообщает компилятору, что функция GetSpeed() является членом класса Airplane. Я расскажу подробнее о функциях-членах класса завтра, когда мы будем изучать классы.



По традиции имена функций-членов класса начинаются с букв верхнего регистра. Хотя жестких требований здесь нет, большинство программ на Си++ следуют именно этой традиции. Замечу также, что я не сторонник использования символа подчеркивания в именах функций. Например, я предпочитаю писать GetVideoRect() вместо get_video_rect(). Независимо от того, какое соглашение вы примете для имен своих функций, будьте последовательны и следуйте ему во всех программах.

Подставляемые функции

Обычно функция включается в исполняемый файл один раз. В дальнейшем она используется посредством *вызовов*. При этом в программе каждый раз происходит переход от точки вызова функции на начало этой функции. После выполнения операторов функции и возврата, программа продолжается с оператора, следующего за вызовом функции.

Подставляемая функция (inline function) включается в исполняемый файл каждый раз, когда к ней происходит обращение.

Подставляемые функции объявляются подобно обычным функциям, но с использованием ключевого слова inline. Когда компилятор встречает в исходном тексте программы обращение к подставляемой функции, он помещает в исполняемый файл отдельную копию этой функции. Подставляемые функции выполняются быстрее, поскольку фактически вызова функции не происходит.



Подставляемыми следует делать функции очень небольшого размера, или же функции, которые должны выполняться очень быстро. Большие функции, а также функции, вызываемые много раз по ходу программы, не стоит делать подставляемыми. Это приведет к заметному увеличению размера исполняемого файла.

Подставляемые функции обычно являются функциями-членами класса. Часто их определения (т.е. сами функции) размещаются в заголовочном файле вслед за объявлением класса. (Это единственный случай, когда можно размещать код в заголовочном файле.) Поскольку функция GetSpeed(), упомя-

нутая выше, очень мала, она легко может быть сделана подставляемой. Вот как это выглядит:

```
inline int Airplane::GetSpeed() {
  return speed; // speed является переменной-членом класса
}
```

Подставляемая функция может быть также определена в объявлении класса. Я еще не рассказывал о классах, поэтому отложим обсуждение этой возможности до завтра.

Заключение

Итак, вы преодолели довольно тяжелый, но важный этап. Сегодня мы рассмотрели указатели и ссылки. Если вы смогли разобраться в указателях, то вы твердо стоите на пути к пониманию Си++. Попутно при обсуждении указателей мы рассмотрели локальное и динамическое распределение памяти, что привело к необходимости изучения операторов new и delete. Сегодняшний день завершился рассмотрением тех дополнительных возможностей, которые предоставляет Си++ при использовании функций по сравнению с языком Си.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Я путаюсь в указателях и ссылках. Это свойственно только мне?

Конечно, нет! Указатели и ссылки достаточно сложны для понимания, и чтобы в них разобраться, требуется некоторое время. Вероятно, вам просто нужно поработать какое-то время с Си++.

Должен ли я каждый раз уничтожать динамический объект, созданный с помощью оператора new?

И да, и нет. Все объекты, созданные с помощью new, должны иметь соответствующий delete, иначе программа вызовет «утечку» памяти. Однако в некоторых случаях за уничтожение объектов отвечают объекты-предки. Таким образом, вопрос состоит не в том, нужно ли уничтожать объект, а в том, кто должен это делать. Для ваших собственных классов вы всегда будете использовать оператор delete. Позже, когда вы узнаете о VCL (на пятый день, «Библиотеки классов Си++ и модель визуальных компонентов»), вы увидите, что объекты-предки VCL отвечают за уничтожение объектов-потомков.

Где мне следует создавать свои объекты — в стеке или «куче»?

Это зависит от объекта. Большие объекты лучше размещать в «куче», чтобы сохранить стековую память. Для небольших объектов и простых типов данных следует использовать стек, чтобы упростить их использование и увеличить скорость работы программы.

Каковы причины использования перегруженных функций?

Перегрузка функций позволяет вам иметь несколько функций с одним именем, выполняющих одну базовую операцию, но принимающих различные аргументы. Например, вы можете создать перегруженную функцию DrawObject(). Одна версия этой функции будет принимать в качестве аргумента класс Circle, другая — класс Square, а третья — класс Polygon. В результате вам не нужно будет использовать три разных имени при вызове этих функций.

Можно ли использовать большое число подставляемых функций?

Это зависит от функции. Вообще говоря, нет. Подставляемыми следует делать функции небольшого размера или редко используемые, а также функции, для которых важна скорость выполнения.

Вопросы

1. Что такое указатель?
2. Что значит разыменовать указатель?
3. Какое значение возвращает оператор new?
4. Как следует передавать функциям экземпляры структур или классов — по ссылке или по значению?
5. Для чего используется ключевое слово const?
6. Можно ли рассматривать следующие функции как перегруженные? Обоснуйте ответ.

```
void MyFunction(int x);
long MyFunction(int x);
```

7. Что лучше использовать, ссылку или указатель?
8. Что такое функция-член класса?
9. В чем состоит различие между подставляемой и обычной функцией для компилятора?
10. Содержит ли данный фрагмент ошибку, и если да, то какую?

```
char* buff = new char[200];
// ...
delete buff;
```

Упражнения

1. Напишите программу, которая объявляет структуру, создает динамический экземпляр этой структуры и заполняет структуру данными. (**Указание**: не забудьте уничтожить структуру после использования.)
2. Измените программу из упражнения 1 так, чтобы вместо указателя использовать ссылку.
3. Перепишите программу REFERENC из листинга 3.2 так, чтобы структура mailingListRecord передавалась функции displayRecord() по ссылке, а не по значению.
4. Что неправильно в объявлении этой функции?

```
void SomeFunction(int param1, int param2 = 0, int param3);
```

5. Объясните пятилетнему ребенку разницу между указателем и ссылкой.

Классы и объектно-ориентированное программирование



Кент Рейсдорф

Сегодня вам предстоит познакомиться с классами. Классы — это сердце Си++ и главная составляющая объектно-ориентированного программирования. Классы являются также основой библиотеки визуальных компонентов (VCL — Visual Component Library), которую вы будете использовать при написании GUI-приложений Windows. (VCL детально рассматривается на пятый день, «Библиотеки классов Си++ и модель визуальных компонентов».) Сначала вы узнаете, что такое класс и как его можно использовать. Попутно вы познакомитесь с такими понятиями Си++, как объект, наследование и абстрактное представление данных. В конце главы вас ожидает введение в файловый ввод-вывод в Си++.

Что такое класс?

Класс, как и структура, представляет собой набор данных и функций, предназначенных для совместного выполнения определенной задачи. Как говорят, класс *инкапсулирует* задачу. Классы имеют следующие характерные элементы:

- Средства контроля доступа
- Конструкторы
- Деструкторы
- Члены-данные
- Функции-члены
- Специальный скрытый указатель с именем this.

Перед тем, как погрузиться в изучение этих элементов, позвольте мне показать на небольшом примере, как работают классы. Давайте рассмотрим типичный управляющий элемент интерфейса Windows — например, флажок (check box). Класс, представляющий флажок, мог бы иметь члены-данные

для хранения подписи к флажку и его состояния (установлен или не установлен). В этот класс должны также входить функции, с помощью которых вы будете устанавливать и запрашивать подпись к флажку и его состояние. Эти функции могут иметь имена GetCheck(), SetCheck(), GetCaption() и SetCaption(). После того, как класс описан, вы можете создать экземпляр этого класса для управления флажком в Windows. Если у вас три флажка, вам потребуются три экземпляра класса CheckBox. С их помощью вы сможете управлять каждым флажком в отдельности. Вот пример текста программы:

```
MyCheckBox check1(ID_CHECK1);
MyCheckBox check2(ID_CHECK2);
MyCheckBox check3(ID_CHECK3);
check1.SetCaption("Thingamabob Option");
check1.SetCheck(true);
check2.SetCaption("Doohickey Options");
check2.SetCheck(false);
check3.SetCaption("Whodyacallum Options");
check3.SetCheck(true);
if (check1.GetCheck()) DoThingamabobTask();
if (check2.GetCheck()) DoDoohickeyTask();
// и т.д.
```

В этом примере каждый экземпляр класса является отдельным объектом, имеющим свои собственные члены-данные и работающим независимо от других объектов. Все объекты имеют одинаковый тип, но занимают разные участки памяти. Теперь, после этого краткого вступления, давайте приступим к подробному изучение классов.

Анатомия класса

Класс, как и структура, должен быть предварительно объявлен. Объявление класса обычно содержится в заголовочном файле. Для простых классов объявление и определение могут быть размещены в единственном исходном файле программы, однако при написании реальных программ так обычно не поступают. Как правило, для класса создают отдельный исходный файл с именем, близким к имени класса, и расширением .cpp. Поскольку Windows 95 и Windows NT поддерживают длинные имена файлов, вы можете при желании использовать имя, в точности совпадающее с именем класса. Заголовочный файл для класса обычно имеет то же имя, что и исходный файл, но расширение .h. Например, если у вас есть класс с именем MyClass, исходный файл следует назвать MYCLASS.CPP, а заголовочный — MYCLASS.H.

Уровни доступа к членам класса

Члены классов могут иметь три уровня доступа: *закрытый (private), открытый (public) или защищенный (protected)*. Все они рассмотрены в данном разделе.

Уровни доступа к членам класса определяют способ работы пользователей с классом. Программируя самостоятельно, вы можете быть как создателем классов, так и их пользователем. При работе в команде один программист может быть создателем класса, а остальные — его пользователями.



Позвольте пояснить пару комментариев, которые я сделал на второй день, «Операторы управления и структуры в Си++». Я говорил тогда, что структура является классом, все члены которого открыты. Фактически, это единственное отличие структуры от класса в Си++. Структура может содержать функции наряду с данными. Структура не может использовать модификаторы доступа (private, protected и public), потому что она имеет только открытый доступ. Я говорил также, что вы, вероятно, не будете часто использовать структуры в программах на Си++. Из-за близкого сходства структур и классов вы, скорее всего, предпочтете использовать класс вместо структуры.

Чтобы понять роль уровней доступа, прежде всего нужно разобраться в том, как используются классы. Любой класс содержит *открытую* часть, к которой возможен доступ извне класса, и закрытую часть. *Закрытая* часть класса определяет его внутреннюю реализацию. В хорошо спроектированном классе от пользователя скрыто все, что ему не требуется знать.



Абстрактное представление данных (data abstraction) — это закрытие внутренней реализации свойств класса от взгляда извне.

Абстрактное представление данных дает пользователю возможность знать о классе ровно столько, сколько необходимо, и предохраняет от вмешательства туда, куда вмешиваться не следует. Например, когда вы садитесь в автомобиль и поворачиваете ключ зажигания, хотите ли вы знать все подробности работы машины? Конечно, нет. Вы хотите знать ровно столько, сколько необходимо для безопасной езды. В этой аналогии рулевое колесо, педали, рычаг переключения передач, спидометр и пр. представляют собой интерфейс между водителем и автомобилем. Водитель знает, какими из этих элементов нужно пользоваться, чтобы машина ехала туда, куда он хочет.

Продолжая аналогию, можно сказать, что двигатель, трансмиссия и электрооборудование автомобиля скрыты от взгляда извне. Двигатель будет работать даже в том случае, если вы никогда не захотите открыть капот. (Это работа для станции техобслуживания!) Его устройство — это детали, которые вам не нужно знать. Они скрыты от вас. Представьте, о каком количестве вещей придется беспокоиться, если следить за всем, что происходит в машине: правильно ли карбюратор приготовляет смесь? Достаточно ли смазки в дифференциале? Хватает ли напряжения генератора для нормальной работы зажигания и радиоприемника? В норме ли зазоры впускных клапанов? Бр-р-р-р!!! Кому это нужно? Таким же образом в классах закрыт доступ к внутренним деталям, чтобы пользователь мог не беспокоиться о том, что происходит «под капотом». Внутренняя работа класса закрыта, тогда как интерфейс пользователя является открытым.

Защищенный уровень доступа объяснить немного труднее. К защищенным членам класса, как и к закрытым, пользователь обращаться не может. Однако эти члены могут быть доступны для классов, которые являются производными данного класса. Я буду говорить об этом подробнее немного позже, в разделе «Функции-члены».

В языке Си++ есть три ключевых слова, устанавливающих уровни доступа. Это (как и следовало ожидать) слова public, private и protected. Уровни доступа к членам класса устанавливаются при объявлении класса. Для объявления класса служит ключевое слово class. Объявление класса выглядит подобно объявлению структуры, но с добавлением модификаторов доступа:

```
class Vehicle {
  public:
    bool haveKey;
    bool Start();
    void SetGear(int Gear);
    void Accelerate(int acceleration);
    void Break(int factor);
    void Turn(int direction);
    void ShutDown();
  protected:
    void StartupProcedure();
  private:
    void StartElectricalSystem();
    void StartEngine();
    int currentGear;
    bool started;
    int speed;
};
```

Обратите внимание, как организовано разбиение членов класса по трем уровням доступа. Вы не обязаны использовать все три уровня доступа в своих классах. При желании можно вообще не устанавливать уровни доступа, но обычно у вас будут присутствовать по крайней мере два из них — public и private.



По умолчанию члены класса имеют уровень доступа private. Если вы не используете при объявлении класса модификаторы доступа, все данные и функции класса будут закрытыми. Такие классы в большинстве случаев не очень полезны.

Конструкторы

Классы в Си++ имеют специальную функцию, называемую *конструктором.*



Конструктор (constructor) — это функция, которая автоматически вызывается при создании экземпляра класса.

Конструктор используется для инициализации переменных-членов класса, выделения необходимой памяти и выполнения других действий, необходимых перед началом использования класса. Класс Vehicle, приведенный выше в качестве примера, не содержит явно определенного конструктора. В таких случаях компилятор C++Builder создает *конструктор по умолчанию (default constructor).* Для простых классов это вполне допустимо, но обычно конструкторами снабжают классы любой значимости. Имя конструктора должно совпадать с именем класса. Это служит отличительным признаком конструктора. Теперь, имея все это в виду, давайте добавим объявление конструктора в класс Vehicle:

```
class Vehicle {
  public:
    Vehicle();      // конструктор
    bool haveKey;
    bool Start();
    void SetGear(int Gear);
    void Accelerate(int acceleration);
```

```
    void Break(int factor);
    void Turn(int direction);
    void ShutDown();
  protected:
    void StartupProcedure();
  private:
    void StartElectricalSystem();
    void StartEngine();
    int currentGear;
    bool started;
    int speed;
};
```

Обратите внимание, что для конструктора не указан тип возвращаемого значения. Дело в том, что конструктор не может возвращать никакого значения. Если вы попытаетесь указать тип возвращаемого значения в объявлении конструктора, компилятор выдаст сообщение об ошибке.

Класс может иметь более одного конструктора. Это возможно благодаря перегрузке функций, что обсуждалось на третий день, «Указатели, ссылки и функции в Си++». Например, можно определить конструктор без аргументов (конструктор по умолчанию) и конструктор, принимающий один или несколько аргументов для инициализации членов-данных. Предположим, что у вас есть класс Rect, служащий для представления прямоугольника (прямоугольники часто используются в Windows-программах). Этот класс мог бы иметь два конструктора: конструктор по умолчанию будет устанавливать все члены-данные в 0, а другой конструктор позволит вам инициализировать члены-данные путем передачи конструктору аргументов. Давайте посмотрим, как могло бы выглядеть объявление класса:

```
class Rect {
  public:
    Rect();
    Rect(int _left, int _top, int _bottom, int _right);
    int GetWidth();
    int GetHeight();
    void SetRect(int _left, int _top, int _bottom, int _right);
  private:
    int left;
    int top;
    int bottom;
    int right;
};
```

Конструкторы можно определить следующим образом:

```
Rect::Rect()
{
  left = 0;
  top = 0;
  bottom = 0;
  right = 0;
}
Rect::Rect(int _left, int _top, int _bottom, int _right);
{
  left = _left;
  top = _top;
  bottom = _bottom;
  right = _right;
}
```

Первый конструктор является конструктором по умолчанию, поскольку он не имеет аргументов. Его задача — просто инициализировать нулем все члены-данные. Второй конструктор принимает переданные ему значения и присваивает их соответствующим членам-данным. Имена переменных в списке параметров снабжены знаком подчеркивания, чтобы отличать их от имен членов-данных.



Помните, что неинициализированные переменные содержат случайные значения. Это верно и для членов-данных класса. Для безопасности следует присваивать переменным-членам класса какие-либо начальные значения.



Реализация (instantation) — это создание объекта, экземпляра или класса.

Важно понимать, что вы не можете явно вызвать конструктор. Каким же образом используются разные конструкторы? Это происходит при создании, или *реализации*, объекта или класса. Следующий фрагмент кода создает два экземпляра класса Rect. В первом используется конструктор по умолчанию, а во втором — конструктор с аргументами:

```
Rect rect1;                  // вызывается конструктор по умолчанию
Rect rect2(0, 0, 100, 100) // вызывается второй вариант конструктора
```

Вы можете иметь любое количество конструкторов, но при этом нельзя допускать неоднозначности в списке параметров (как и для перегруженных функций).

Списки инициализаторов



Си++ дает возможность инициализировать члены-данные класса при помощи *списка инициализаторов (initializer list)*.

В следующем примере показан способ инициализации членов-данных класса. Я хочу сразу привести пример, а не пытаться объяснять использование списка инициализаторов. Давайте возьмем два конструктора для класса Rect и инициализируем члены-данные при помощи списка, а не в теле функции, как раньше. Вот как это выглядит:

```
Rect::Rect() :
  left(0),
  top(0),
  bottom(0),
  right(0)
{
}
Rect::Rect(int _left, int _top, int _bottom, int _right) :
  left(_left),
  top( _top),
  bottom(_bottom),
  right(_right)
{
}
```

Обратите внимание на два момента. Во-первых, список инициализаторов начинается с двоеточия. (Двоеточие стоит после заголовка функции, поэтому

вы могли его не заметить.) Во-вторых, переменные в списке разделяются запятыми, но за последней переменной запятая не ставится. Несоблюдение этих правил приведет к выдаче сообщения об ошибке при компиляции.



На третий день я говорил о ссылках. Вы можете включать ссылки в число членов-данных класса, но инициализация ссылок возможна только через список, и никак иначе. Вот пример:

```
class MyClass {
  public:
    MyClass();
    // другие открытые члены
  private:
    OtherClass& other;
    // другие закрытые члены
};
MyClass::MyClass() :
    other(*new OtherClass)  // это можно сделать только здесь!
{
}
```

Попытка инициализировать ссылку в каком-либо другом месте приведет к ошибке при компиляции.

В большинстве случаев не имеет значения, где вы инициализируете члены-данные — в теле конструктора или списке. Лично я предпочитаю использовать список инициализаторов.

Деструкторы



Деструктор (destructor) — это специальная функция, которая автоматически вызывается перед уничтожением объекта.

Деструктор можно рассматривать как противоположность конструктору. Обычно он используется для освобождения памяти, выделенной классу, или выполняет другие задачи по наведению порядка после работы класса. Класс может иметь не более одного деструктора. Наличие деструктора не является обязательным. Деструктор не возвращает никакого значения и не принимает аргументов. Его имя образуется из имени класса с добавлением впереди тильды (~).

Как было сказано, деструктор вызывается непосредственно перед разрушением класса. Класс может быть разрушен либо при выходе из области видимости (в случае размещения в стеке), либо в результате применения оператора delete (в случае динамического размещения). В любом случае, вызов деструктора будет последним действием перед окончательным исчезновением класса.

Далее приведен измененный код для класса Rect:

```
class Rect {
  public:
    Rect();
    Rect(int _left, int _top, int _bottom, int _right);
    ~Rect();          // добавлен деструктор
    int GetWidth();
    int GetHeight();
    void SetRect(int _left, int _top, int _bottom, int _right);
```

```
  private:
    int left;
    int top;
    int bottom;
    int right;
    char* text;      // добавлен новый член класса
};
Rect::Rect() :
  left(0),
  top(0),
  bottom(0),
  right(0)
{
  text = new char[256];
  strcpy(text, "Any Colour You Like");
}
// промежуточный текст опущен
Rect::~Rect()
{
  delete[] text;
}
```

Конструктор в модифицированной версии класса Rect выделяет место для символьного массива с именем text, а деструктор освобождает эту память. (Я не вижу особой необходимости включать в этот класс текст, но кто знает!) Повторю еще раз, что деструктор используется для приведения среды в исходное состояние перед разрушением экземпляра класса.

Члены-данные

Члены-данные класса — это просто переменные, объявленные в описании класса. Их областью видимости по умолчанию является класс. По сравнению с членами-данными структур, вы можете управлять доступом к членам-данным класса, объявляя их как private, public или protected. Независимо от установленного уровня доступа, члены-данные класса могут использоваться всеми функциями этого класса. Уровень доступа определяет видимость членов-данных вне класса. Например, к закрытым или защищенным членам-данным доступ извне невозможен. Обращение к открытым членам-данным возможно только через класс. Рассмотрим, например, класс Rect, объявленный ранее. В нем отсутствуют открытые члены-данные. При попытке обращения к переменным этого класса вы получите сообщение об ошибке:

```
Rect rect(10, 10, 200, 200);
int x = rect.left; // ошибка компиляции!
```

Компилятор выдаст сообщение Rect::left is not accessible (Rect::left недоступен). Это означает, что left является закрытым членом класса и вы не можете обращаться к нему из программы. Если переменную left объявить в разделе public, то компиляция пройдет успешно.

Для работы с закрытыми членами-данными вы можете написать специальные функции. Одна функция будет возвращать значение определенного члена класса, а другая — изменять его значение. Это открытые функции-члены, работающие с закрытыми членами-данными.

Проиллюстрируем сказанное на примере. Пусть у вас есть две функции для обращения к члену left класса Rect:

```
int Rect::GetLeft();
{
  return left;
}
void Rect::SetLeft(int newLeft)
{
  left = newLeft;
}
```

Теперь, чтобы получить значение члена left, вы должны поступить следующим образом:

```
TRect rect;
int x = rect.GetLeft();
```

В некоторых случаях такое решение является избыточным. Тем не менее, использование специальных функций для изменения членов-данных класса дает одно важное преимущество — возможность контроля входных данных. Это позволяет вам следить за значениями, которые имеют члены-данные.



Некоторые экстремисты от ООП говорят, что члены-данные никогда не должны быть открытыми. По их мнению, доступ к членам-данным должен осуществляться только через функции. С другой стороны, есть люди, которые призывают делать все члены-данные открытыми. Истина, как всегда, лежит где-то посередине. Некоторые члены-данные не являются критичными и их можно сделать открытыми, если это более удобно. От других членов существенным образом зависит функционирование класса, и их следует закрыть для прямого доступа. Если вы сомневаетесь, как поступить, лучше сделать соответствующие члены-данные закрытыми.

Для каждого экземпляра класса в памяти создается отдельная копия членов-данных. Исключение делается только для членов-данных, объявленных с модификатором static. В этом случае все экземпляры класса используют одну копию соответствующего члена. Изменение статического члена одним из экземпляров класса приводит к его изменению во всех остальных экземплярах. Статические члены-данные используются не очень часто, поэтому не беспокойтесь, если вы сейчас не совсем поняли, о чем шла речь.

Правила работы с членами-данными класса

- Используйте столько членов-данных, сколько необходимо для нормального функционирования класса, но старайтесь по возможности обходиться локальными переменными.
- Не делайте все члены-данные открытыми.
- Используйте функции для обращения к закрытым членам-данным, которые должны быть доступны.
- Контролируйте значения, присваиваемые закрытым членам-данным с помощью функций, чтобы избежать неправильного ввода.
- Инициализируйте все члены-данные либо в списке инициализаторов, либо в теле конструктора.
- Не забывайте уничтожать члены-данные, размещаемые в памяти динамически.

Функции-члены

Функции-члены класса — это функции, принадлежащие вашему классу. Они являются локальными по отношению к классу, в котором определены, и не существуют вне класса. Функции-члены могут быть вызваны только изнутри класса или через экземпляр класса. Они имеют доступ ко всем открытым, защищенным и закрытым членам-данным своего класса. Функции-члены могут быть объявлены в разделах private, protected или public. Вы должны подумать о том, в какой раздел поместить свои функции.

Открытые функции-члены представляют собой интерфейс между пользователем и классом. Именно через открытые функции пользователи класса получают доступ к тем возможностям, которые предоставляет класс. Например, у вас есть класс для записи и воспроизведения звука. В этом случае открытыми могут быть функции типа Open(), Play(), Record(), Save(), Rewind() и т.д.

Закрытые функции-члены предназначены только для использования самим классом. Их вызов пользователем класса не предполагается; они скрыты от внешнего мира. Часто при создании класса необходимо выполнить ряд инициализирующих действий. (Вы уже познакомились с конструктором, который автоматически вызывается при создании класса.) Для некоторых классов начальные процедуры требуют выполнения довольно большого объема кода. Чтобы не загромождать конструктор, лучше всего написать функцию Init(), которая будет вызываться из конструктора и выполнять все необходимые действия. Эта функция никогда не должна напрямую вызываться пользователем класса. При обращении к ней в неподходящий момент может произойти много неприятностей, поэтому для защиты класса (и пользователя) она должна быть закрытой.

Защищенные функции-члены — это функции, доступные для только для тех классов, которые являются производными данного класса. Доступ извне к этим функциям невозможен. Я еще не говорил о производных классах; для лучшего восприятия они будут обсуждаться позже, в разделе «Наследование».

Синтаксис

Функция типа inline, первая форма:

```
ClassName {
  public:
    ReturnType FunctionName();
};

inline ReturnType ClassName::FunctionName() {
  statements;
}
```

Функция FunctionName объявлена в теле класса ClassName. Определение функции (т.е. сама функция) находится вне объявления класса. При этом используется ключевое слово inline. Имени функции должно предшествовать имя класса с оператором разрешения видимости.

Функция типа inline, вторая форма:

```
ClassName {
  public:
    ReturnType FunctionName()
    {
      statements;
    }
};
```

Функция FunctionName объявлена и определена внутри объявления класса. В этом случае она автоматически считается подставляемой. Ключевое слово inline не требуется.

Также как и члены-данные, функции-члены могут быть объявлены с модификатором static. *Статические функции-члены* более похожи на обычные функции. Они не имеют доступа к членам-данным класса. (Немного позже я скажу, зачем нужно это ограничение.) Статические функции-члены используются довольно редко, но иногда в них возникает необходимость. Например, некоторые функции интерфейса Windows API используют *обратные вызовы (callbacks)* для выполнения повторяющихся действий. Если вы будете использовать подобные функции в своих классах, они должны быть объявлены статическими.

Правила работы с функциями-членами класса

- Делайте открытыми только те функции, которые необходимы пользователю для работы с классом.
- Делайте закрытыми все функции, о которых пользователю не нужно знать.
- Делайте защищенными те функции, которые могут потребоваться производным классам, но о которых пользователю также не нужно знать.
- Используйте статические функции-члены только в особых случаях.
- Объявляйте подставляемыми все функции, которые должны выполняться быстро. Не забывайте, что подставляемые функции должны быть короткими.
- Оформляйте код, который повторяется более двух раз, в виде отдельной функции.

Что такое this?



Все классы имеют скрытый член с именем this. Это указатель на экземпляр данного класса в памяти.

Очевидно, что необходимо сделать некоторое пояснение. Прежде всего, давайте посмотрим, как выглядел бы класс Rect, если бы this не был скрытым членом:

```
class Rect {
  public:
    Rect();
    Rect(int _left, int _top, int _bottom, int _right);
    ~Rect();
    int GetWidth();
    int GetHeight();
    void SetRect(int _left, int _top, int _bottom, int _right);
  private:
    Rect* this;       // если бы this не был невидимым
    int left;
    int top;
    int bottom;
    int right;
    char* text;
};
```

Так выглядит класс Rect для компилятора. При создании экземпляра класса указателю this автоматически присваивается адрес начала объекта в памяти:

```
TRect* rect = new TRect(20, 20, 100, 100);
// теперь 'rect' и 'rect->this' имеют одинаковые значения,
// т.к. оба указывают на один объект
```

«Но для чего нужен this?» — спросите вы. Вспомните, что каждый экземпляр класса имеет свою копию членов-данных. При этом все экземпляры используют один набор функций-членов класса (нет никаких причин создавать дубликаты функций для каждого экземпляра). Как же компилятор определяет, какой из экземпляров вызвал функцию? Для этого каждая функция-член класса имеет скрытый аргумент this. Для иллюстрации предположим, что у вас есть функция GetWidth(), являющаяся членом класса Rect. Она может выглядеть следующим образом:

```
int Rect::GetWidth()
{
  return right - left;
}
```

Так эта функция выглядит для нас с вами. Компилятор, однако, видит ее примерно следующим образом:

```
int Rect::GetWidth(Rect* this)
{
  return this->right - this->left;
}
```

Это не совсем верно с точки зрения внутренней реализации, но достаточно для нашего обсуждения. Из этого примера вы можете увидеть, что this упрощает работу с классами, оставаясь при этом в тени. Вам не нужно беспокоиться о том, как это происходит, вы должны только знать, что это имеет место.



Никогда не изменяйте указатель this. Вы можете передавать его другой функции в качестве аргумента, или использовать при создании новых классов, но не изменять. Приучите себя рассматривать this как переменную только для чтения.

Несмотря на то, что this работает «за кулисами», он остается переменной, доступной внутри класса. Давайте ненадолго обратимся к VCL. В основном вы будете создавать компоненты VCL, помещая их в форму на этапе разработки. При этом C++Builder создает указатель на компонент и выполняет много рутинных действий, избавляя вас от необходимости вникать в технические подробности. Иногда, однако, вам понадобится создать компонент во время выполнения программы. VCL всегда должна знать, с каким объектом-предком связан объект-потомок. Например, вы хотите создать новую кнопку при щелчке на существующей кнопке. Для этого нужно сообщить VCL, какой объект является предком новой кнопки. Соответствующий код будет выглядеть следующим образом:

```
void __fastcall TMyForm::Button1Click(TObject* Sender)
{
  TButton* button = new TButton(this);
  button->Parent = this;
  button->Caption = "New Button";
  button->Left = 100;
  button->Top = 100;
  button->Show();
  // ...
}
```

В этом примере this используется в конструкторе (для установки свойства Owner кнопки, об этом я буду рассказывать на восьмой день, «Компоненты VCL»), а также присваивается свойству Parent вновь создаваемой кнопки. При создании приложений C++Builder вы будете использовать указатель this в основном таким образом.



Ранее я говорил, что статические функции-члены не могут обращаться к членам-данным класса. Дело в том, что статические функции не имеют скрытого аргумента this, без которого доступ к членам класса невозможен.

Начав работать с VCL, вы быстро поймете, когда в приложениях C++Builder требуется использовать this.

Один пример

Сейчас самое время привести пример использования классов. Последующие листинги содержат программу, в которую включены классы. Эта программа позволяет вам побыть в роли авиадиспетчера, выдавая команды трем самолетам. Листинг 4.1 — это заголовочный файл для класса Airplane, листинг 4.2 содержит исходный код для класса Airplane, а листинг 4.3 — основную программу.

Листинг 4.1. AIRPLANE.H

```
 1: //---------------------------------------------------------------------
 2: #ifndef airplaneH
 3: #define airplaneH
 4: #define AIRLINER      0
 5: #define  COMMUTER     1
 6: #define  PRIVATE      2
 7: #define  TAKINGOFF    0
 8: #define  CRUISING     1
 9: #define LANDING       2
10: #define ONRAMP         3
11: #define  MSG_CHANGE   0
12: #define  MSG_TAKEOFF  1
13: #define  MSG_LAND     2
14: #define  MSG_REPORT   3
15: class Airplane {
16:   public:
17:     Airplane(const char* _name, int _type = AIRLINER);
18:     ~Airplane();
```

см. продолжение

Листинг 4.1. (продолжение)

```
19:     virtual int GetStatus(char* statusString);
20:     int GetStatus()
21:     {
22:       return status;
23:     }
24:     int Speed()
25:     {
26:       return speed;
27:     }
28:     int Heading()
29:     {
30:       return heading;
31:     }
32:     int Altitude()
33:     {
34:       return altitude;
35:     }
36:     void ReportStatus();
37:     bool SendMessage(int msg, char* response,
38:       int spd = -1, int dir = -1, int alt = -1);
39:     char* name;
40:   protected:
41:     virtual void TakeOff(int dir);
42:     virtual void Land();
43:   private:
44:     int speed;
45:     int altitude;
46:     int heading;
47:     int status;
48:     int type;
49:     int ceiling;
50: };
51: #endif
```

Листинг 4.2. AIRPLANE.CPP

```
 1: #include <iostream.h>
 2: #include <stdio.h>
 3: #include "airplane.h"
 4: //
 5: // Конструктор выполняет инициализацию
 6: //
 7: Airplane::Airplane(const char* _name, int _type) :
 8:   type(_type),
 9:   status(ONRAMP),
10:   speed(0),
11:   altitude(0),
12:   heading(0)
13: {
14:   switch (type) {
15:     case AIRLINER : ceiling = 35000; break;
16:     case COMMUTER : ceiling = 20000; break;
17:     case PRIVATE  : ceiling = 8000;
18:   }
19:   name = new char[50];
```

```
20:   strcpy(name, _name);
21: }
22: //
23: // Деструктор выполняет очистку памяти
24: //
25: Airplane::~Airplane()
26: {
27:   delete[] name;
28: }
29: //
30: // Прием сообщения от пользователя
31: //
32: bool
33: Airplane::SendMessage(int msg, char* response,
34:   int spd, int dir, int alt)
35: {
36:   //
37:   // Проверка правильности ввода
38:   //
39:   if (spd > 500) {
40:     strcpy(response, "Speed cannot be more than 500.");
41:     return false;
42:   }
43:   if (dir > 360) {
44:     strcpy(response, "Heading cannot be over 360 degrees.");
45:     return false;
46:   }
47:   if (alt < 100 && alt != -1) {
48:     strcpy(response, "I'd crash, bonehead!");
49:     return false;
50:   }
51:   if (alt > ceiling) {
52:     strcpy(response, "I can't go that high.");
53:     return false;
54:   }
55:   //
56:   // Выполнение действий в зависимости от полученной команды
57:   //
58:   switch (msg) {
59:     case MSG_TAKEOFF : {
60:       // Взлет невозможен, самолет уже в воздухе!
61:       if (status != ONRAMP) {
62:         strcpy(response, "I'm already in the air!");
63:         return false;
64:       }
65:       TakeOff(dir);
66:       break;
67:     }
68:     case MSG_CHANGE : {
69:       // На земле изменение невозможно
70:       if (status == ONRAMP) {
71:         strcpy(response, "I'm on the ground");
72:         return false;
73:       }
74:       // Изменение только при передаче неотрицательного значения
75:       if (spd != -1) speed = spd;
76:       if (dir != -1) heading = dir;
77:       if (alt != -1) altitude = alt;
```

см. продолжение

Листинг 4.2. (продолжение)

```
 78:         status == CRUISING;
 79:         break;
 80:       }
 81:       case MSG_LAND : {
 82:         if (status == ONRAMP) {
 83:           strcpy(response, "I'm already on the ground.");
 84:           return false;
 85:         }
 86:         Land();
 87:         break;
 88:       }
 89:       case MSG_REPORT : ReportStatus();
 90:     }
 91:     //
 92:     // Стандартный отклик, если все в порядке
 93:     //
 94:     strcpy(response, "Roger.");
 95:     return true;
 96: }
 97: //
 98: // Выполнение взлета
 99: //
100: void
101: Airplane::TakeOff(int dir)
102: {
103:   heading = dir;
104:   status = TAKINGOFF;
105: }
106: //
107: // Выполнение посадки
108: //
109: void
110: Airplane::Land()
111: {
112:   speed = heading = altitude = 0;
113:   status == ONRAMP;
114: }
115: //
116: // Создание строки состояния самолета
117: //
118: int
119: Airplane::GetStatus(char* statusString)
120: {
121:   sprintf(statusString, "%s, Altitude: %d, Heading: %d, "
122:     "Speed: %d\n", name, altitude, heading, speed);
123:   return status;
124: }
125: //
126: // Получение строки состояния и вывод ее на экран
127: //
128: void
129: Airplane::ReportStatus()
130: {
131:   char buff[100];
132:   GetStatus(buff);
133:   cout << endl << buff << endl;
134: }
```

Листинг 4.3. AIRPORT.CPP

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\condefs.h>
 3: #include <iostream.h>
 4: #include <conio.h>
 5: #pragma hdrstop
 6: USERES("Airport.res");
 7: USEUNIT("airplane.cpp");
 8: #include "airplane.h"
 9: int getInput(int max);
10: void getItems(int& speed, int& dir, int& alt);
11: int main(int argc, char **argv)
12: {
13:   char returnMsg[100];
14:   //
15:   // Создание массива указателей на объекты
16:   // и трех объектов Airplane
17:   //
18:   Airplane* planes[3];
19:   planes[0] = new Airplane("TWA 1040");
20:   planes[1] = new Airplane("United Express 749", COMMUTER);
21:   planes[2] = new Airplane("Cessna 3238T", PRIVATE);
22:   //
23:   // Начало цикла
24:   //
25:   do {
26:     int plane, message, speed, altitude, direction;
27:     speed = altitude = direction = -1;
28:     //
29:     // Выбор самолета, которому адресовано сообщение.
30:     // Вывод списка самолетов и ожидание выбора пользователя
31:     //
32:     cout << endl << "Who do you want to send a message to?";
33:     cout << endl << endl << "0. Quit" <<  endl;
34:     for (int i = 0; i < 3; i++)
35:       cout << i + 1 << ". " << planes[i]->name << endl;
36:     //
37:     // Вызов функции getInput() для получения номера самолета
38:     //
39:     plane = getInput(4);
40:     //
41:     // Если пользователь выбирает пункт 0, выход из цикла
42:     //
43:     if (plane == -1) break;
44:     //
45:     // Подтверждение
46:     //
47:     cout << endl << planes[plane]->name << ", roger.";
48:     cout << endl << endl;
49:     //
50:     // Выбор команды
51:     //
52:     cout << "What message do you want to send?" << endl;
53:     cout << endl << "0. Quit" << endl;
54:     cout << "1. State Change" << endl;
55:     cout << "2. Take Off" << endl;
56:     cout << "3. Land" << endl;
```

см. продолжение

Листинг 4.3. (продолжение)

```
57:     cout << "4. Report Status" << endl;
58:     message = getInput(5);
59:     //
60:     // Выход из цикла, если пользователь выбрал пункт 0
61:     //
62:     if (message == -1) break;
63:     //
64:     // Если пользователь выбрал пункт 1, необходимо
65:     // ввести новую скорость, направление и высоту.
66:     // Вызов функции getItems()
67:     //
68:     if (message == 0)
69:       getItems(speed, direction, altitude);
70:     //
71:     // Передача команды самолету
72:     //
73:     bool goodMsg = planes[plane]->SendMessage(message,
74:       returnMsg, speed, direction, altitude);
75:     //
76:     // Команда не распознана
77:     //
78:     if (!goodMsg) cout << endl << "Unable to comply.";
79:     //
80:     // Вывод отклика самолета
81:     //
82:     cout << endl << returnMsg << endl;
83:   } while (1);
84:   //
85:   // Уничтожение объектов Airplane
86:   //
87:   for (int i = 0; i < 3; i++) delete planes[i];
88: }
89: int getInput(int max)
90: {
91:   int choice;
92:   do {
93:     choice = getch();
94:     choice -= 49;
95:   } while (choice < -1 || choice > max);
96:   return choice;
97: }
98: void getItems(int& speed, int& dir, int& alt)
99: {
100:  cout << endl << "Enter new speed: ";
101:  getch();
102:  cin >> speed;
103:  cout << "Enter new heading: ";
104:  cin >> dir;
105:  cout << "Enter new altitude: ";
106:  cin >> alt;
107:  cout << endl;
108: }
```



Давайте начнем с заголовочного файла из листинга 4.1. Прежде всего, обратите внимание на директивы #define. Они устанавливают соответствие между двумя текстовыми строками. Во время компиляции происходит замена первой строки на вторую. #define используется здесь потому, что гораздо легче запомнить строку текста, чем номер. Что из приведенного ниже вы предпочтете?

```
if (type == AIRLINER) ...
// или
if (type == 0) ...
```

По традиции для записи имен констант, определенных с помощью #define, используются буквы верхнего регистра, но вы можете смешивать произвольным образом прописные и строчные буквы. Я предпочитаю использовать только верхний регистр, поскольку это позволяет с первого взгляда отличать константу от переменной.



Другой способ определить константу — это объявить переменную с модификатором const. Вот пример:

```
const int airliner = 0;
```

Возможно, это более современный способ определения констант.

Следующее, на что нужно обратить внимание в заголовочном файле — это использование подставляемых функций. Эти функции настолько малы, что имеет смысл сделать их подставляемыми. Вы могли также заметить, что класс Airplane имеет одну перегруженную функцию с именем GetStatus(). При вызове с аргументом в виде символьного массива она возвращает строку состояния, а при вызове без аргументов — значение переменной status. Заметьте, что из всех членов-данных лишь один является открытым. Доступ к остальным членам возможен только через открытые функции. Например, вы можете изменить скорость, высоту и курс самолета, только выдав ему соответствующую команду. Аналогично этому, авиадиспетчер не может физически изменить направление полета самолета. Максимум, что он может сделать — это дать указание пилоту изменить курс.

Теперь перейдем к листингу 4.2. Он содержит определение класса Airplane. Конструктор выполняет инициализацию, включающую динамическое размещение символьного массива, содержащего названия самолетов. Выделенная память освобождается деструктором. Основную работу выполняет функция SendMessage(). Оператор switch определяет, какая команда была послана и выполняет соответствующее действие. Заметьте, что функции TakeOff() и Land() не могут быть вызваны непосредственно (они относятся к защищенным), а вызываются через функцию SendMessage(). Таким образом, вы не можете поднять или посадить самолет; можно только выдать команду на совершение одного из этих действий. Функция ReportStatus() вызывает GetStatus() для получения строки состояния, которую та формирует.

Основная программа приведена в листинге 4.3. Первым делом программа инициализирует массив указателей на Airplane и создает три экземпляра этого класса. Затем начинается цикл. Вы можете послать команду любому из трех самолетов, вызывая функцию SendMessage(). При этом вы получаете отклик. Обратите внимание, что условное выражение в цикле do-while представляет собой просто 1. Такой цикл будет выполняться бесконечно. В данном

случае это не страшно, потому что для выхода из цикла использован оператор break. Запустите программу и поиграйте с ней, чтобы почувствовать, как она работает.

Наследование

Одним из наиболее мощных свойств классов в Си++ является возможность их расширения путем наследования.



Наследование (inheritance) — это создание нового класса путем добавления новых членов к уже существующему классу.



Класс, к которому добавляются новые члены, называется *базовым классом (base class)*, а вновь созданный класс — *производным классом (derived class)*.

Давайте рассмотрим в качестве примера класс Airplane. Как вы знаете, между гражданской и военной авиацией существуют большие различия. Для представления военного самолета я создам новый класс на основе класса Airplane, добавив к нему новые члены:

```
class MilitaryPlane : public Airplane {
  public:
    MilitaryPlane(char* name, int _type);
    virtual int GetStatus(char* statusString);
  protected:
    virtual void TakeOff();
    virtual void Land();
    virtual void Attack();
    virtual void SetMission();
  private:
    Mission theMission;
};
```

MilitaryPlane включает в себя все, что содержит Airplane, а также несколько дополнительных элементов. Обратите внимание на первую строку определения класса. Двоеточие после имени класса указывает компилятору, что данный класс является производным от другого класса. За двоеточием следует имя базового класса. Ключевое слово public в данном контексте означает, что я хочу получить доступ ко всем открытым функциям и данным базового класса.



Новый класс, созданный на основе другого класса, обладает всеми функциональными возможностями базового класса. Вы можете добавить новые функции и данные, но нельзя удалить что-либо, принадлежащее базовому классу.

Вы должны были заметить, что в разделе private присутствует строка с объявлением переменной класса Mission. Класс Mission инкапсулирует все, что связано с боевым заданием: цель, навигационные опорные точки, высоты и курсы входа и выхода, и т.д. Таким образом, члены-данные могут быть экземплярами другого класса. Все это вы встретите в большом количестве при программировании в C++Builder.

Есть еще кое-что, о чем я пока не рассказывал. Обратите внимание на ключевое слово virtual. Оно указывает, что данная функция является виртуальной.



Виртуальная функция (virtual function) — это функция, которая будет автоматически вызываться, если в производном классе существует функция с таким же именем.

Например, функция TakeOff() в классе Airplane объявлена виртуальной. Вернитесь к листингу 4.2. TakeOff() вызывается функцией SendMessage() в ответ на команду MSG_TAKEOFF. Если класс MilitaryPlane не содержит собственной функции TakeOff(), будет вызвана функция из базового класса. Но поскольку эта функция на самом деле присутствует в классе MilitaryPlane, она будет вызываться вместо соответствующей функции базового класса.



Замена функции базового класса функцией производного класса называется *замещением (overriding)* функции.

Чтобы произошло замещение, функция производного класса должна в точности совпадать с функцией базового класса. Иными словами, типы возвращаемых значений, имена функций и списки параметров должны быть одинаковыми.

Вы можете использовать замещение для изменения функции базового класса или ее расширения. Рассмотрим, например, функцию TakeOff(). Если вы хотите полностью изменить работу функции TakeOff() класса Airplane, определите в классе MilitaryPlane следующую функцию:

```
void MilitaryPlane::TakeOff(int dir)
{
  // новый код
}
```

Но если вы хотите, чтобы ваша функция дополнительно сохранила все возможности функции базового класса, вызовите сначала прежнюю функцию:

```
void MilitaryPlane::TakeOff(int dir)
{
  Airplane::TakeOff(dir);
  // новый код
}
```

Для расширения возможностей функции базового класса, вы можете добавить соответствующий код до или после ее вызова. Оператор разрешения видимости используется для указания компилятору, что вызывается функция TakeOff() класса Airplane. Заметьте, что функция TakeOff() объявлена в разделе protected класса Airplane. Если бы она находилась в разделе private, этот вызов был бы невозможен, т.к. даже производные классы не имеют доступа к закрытым членам класса-предка. Делая функцию TakeOff() защищенной, вы скрываете ее от внешнего мира, но оставляете доступной для производных классов.



Оператор разрешения видимости требуется только в том случае, когда у вас есть функции базового и производного классов с одинаковыми именами, возвращаемыми значениями и параметрами. При отсутствии замещения вы можете вызывать открытую или защищенную функцию базового класса без этого оператора. Например, если вы хотите получить строку состояния самолета до взлета, можно использовать примерно следующий код:

```
void MilitaryPlane::TakeOff(int dir)
{
  if (GetStatus() != ONRAMP) Land(); // сначала посадка!
  Airplane::TakeOff(dir);
  // новый код
}
```

Функция GetStatus() существует только в базовом классе, поэтому нет необходимости использовать оператор разрешения видимости. Но при обращении к Land() будет вызвана функция из класса MilitaryPlane, т.к. она имеет меньшую область видимости.

Когда вы создаете класс на основе другого класса, необходимо правильно инициализировать базовый класс путем вызова соответствующего конструктора. Конструктор базового класса вызывается в списке инициализаторов. Вот как это будет выглядеть для класса MilitaryPlane:

```
MilitaryPlane::MilitaryPlane(char* _name)
  : Airplane(_name, MILITARY) // вызов конструктора базового класса
{
  // тело конструктора
}
```

Не забывайте вызывать конструктор базового класса всякий раз, когда вы создаете производный класс. На рис. 4.1 показан пример создания иерархии классов путем наследования.

Из рисунка 4.1 видно, что класс с именем F16 является производным от класса MilitaryFighter. В конечном счете, F16 происходит от Airplane, поскольку класс Airplane является базовым для всех остальных классов.

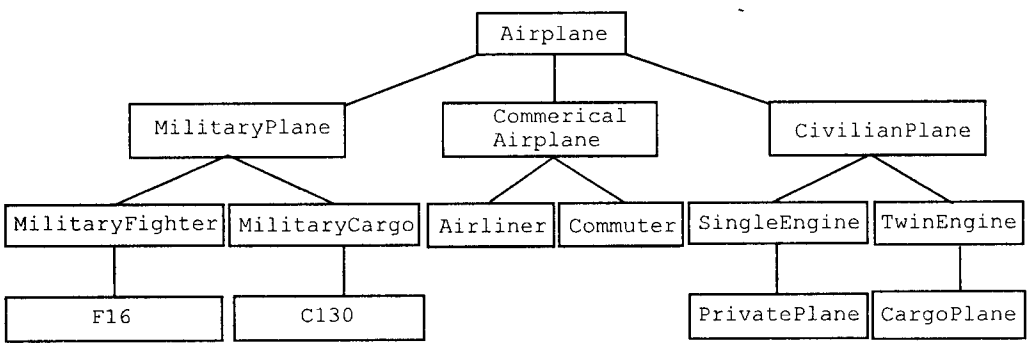


Рис. 4.1
Пример наследования

Множественное наследование



Создание класса на основе двух или более базовых классов называется *множественным наследованием (multiple inheritance)*.

Множественное наследование применяется не часто, но оно может оказаться очень полезным. Пусть у вас есть класс Armaments, который отражает состав вооружения отдельного самолета. Он может выглядеть следующим образом:

```
class Armaments {
  public:
    Armaments();
    LoadArms();
  private:
    bool isArmed;
    int numSideWinders;
    int numSparrows;
    // и т.д.
};
```

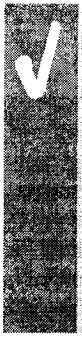
Пусть теперь вам нужно создать класс для представления военного истребителя. Вы можете взять за основу классы MilitaryPlane и Armaments:

```
class Fighter : public MilitaryPlane, public Armaments {
  public:
    Fighter(char* name);
  private:
    // другие объявления
};
```

В результате вы получили класс, содержащий все открытые члены классов MilitaryPlane и Armaments. Теперь можно сделать следующее:

```
Fighter fighter("F16");
fighter.LoadArms();
fighter.SendMessage(...);
// и т.д.
```

Два базовых класса объединены для образования одного производного класса.



Для правильной инициализации вы должны вызывать конструкторы всех базовых классов. Вот пример:

```
F16::F16(char* _name)
  : MilitaryPlane(_name, F16), Armaments()
{
  // тело конструктора
}
```

Для класса, имеющего конструктор по умолчанию, нет особой необходимости вызывать конструктор базового класса. В большинстве ситуаций, однако, вы будете вызывать конструктор базового класса для всех классов-потомков.

Позвольте привести еще один пример. В Соединенных Штатах перевозку армейских подразделений осуществляет специальный авиаотряд MAC (Military Air Command). MAC является чем-то вроде собственной авиакомпании вооруженных сил. Поскольку военные — это в конечном счете груз, для их перевозки требуется военно-транспортный самолет. Но люди — это особый груз, и вы не можете просто бросить их в грузовой отсек (по крайней мере, не всегда). Поэтому вам необходим военно-транспортный самолет с чертами пассажирского авиалайнера. Вернемся обратно к рис. 4.1. Из него видно, что если произвести новый класс от классов MilitaryCargo и Airliner, мы получим то, что требуется. Это иллюстрирует рис. 4.2.

Множественное наследование — очень полезный инструмент, который удобно иметь под рукой, хотя вам не обязательно использовать его часто.

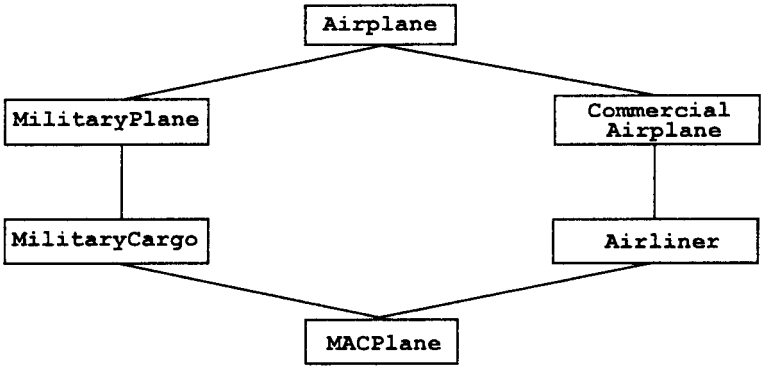


Рис. 4.2 Пример множественного наследовании



Классы VCL не поддерживают множественное наследование. Однако вы можете использовать его в любых других классах, созданных для использования в приложениях C++Builder.

Базовый файловый ввод-вывод

Довольно скоро наступит время, когда вашим приложениям потребуется читать и записывать файлы. Я специально не касался этого раньше, потому что для понимания файлового ввода-вывода в Си++ необходимо знание классов.

Если вы собираетесь интенсивно работать с базами данных при помощи C++Builder, то вам не придется напрямую обращаться к их файлам. Для этого предназначены специальные компоненты VCL. VCL также обеспечивает поддержку чтения и записи для элементов редактирования, окон списков и других базовых элементов интерфейса Windows. API Windows предоставляет функции для чтения файлов конфигурации (.INI-файлов). Многие из элементов управления OCX и ActiveX «знают», как сохранять и загружать специальные файлы, необходимые им для выполнения своих функций.

Как видите, многие объекты, которые вы будете использовать в C++Builder, выполняют файловые операции за вас. Однако время от времени вам придется использовать свои собственные файлы, а для этого необходимо знание основ файлового ввода-вывода.

Базовый файловый ввод-вывод осуществляется в Си++ через три класса:

- Класс ofstream, обеспечивающий вывод в файл
- Класс ifstream, обеспечивающий ввод из файла
- Класс fstream, обеспечивающий как ввод, так и вывод

Все эти классы являются производными от класса iostream. Вы уже имели опыт работы с iostream, хотя могли и не подозревать об этом. Классы cout и cin, использовавшиеся для консольного ввода-вывода, также являются производными от iostream. Сам iostream, в свою очередь, основан на классе ios, который является базовым для всего поточного ввода и вывода в Си++.

Действительно, в лабиринте классов потокового ввода-вывода можно иногда заблудиться. Но сейчас я собираюсь рассказать о файловом вводе-выводе

только в общих чертах, а это не очень сложно. Если же вам потребуется выполнять какие-то нестандартные файловые операции, то можно обратиться к справочным файлам библиотеки потокового ввода-вывода или приобрести хорошую книгу по Си++ (например, *Teach Yourself Borland C++ 4.5 in 21 Days, Second Edition* by Sams Publishing)*, в которой детально рассматривается работа с файлами.

Ввод из файла

Чтение текстового файла в Си++ является довольно простой задачей. Листинг 4.4 содержит программу, которая читает свой собственный исходный файл и выводит на экран строки по мере их считывания. Прежде всего, введите эту программу, не забыв убрать номера строк. Затем сохраните проект под именем READFILE. Если вы не сохраните программу под этим именем, она не будет правильно работать. Откомпилируйте и запустите программу. Поскольку программа читает свой собственный файл, на экран будет выведено содержимое листинга 4.4.

Листинг 4.4. READFILE.CPP

```
 1: #include <condefs.h>
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4: #include <iostream.h>
 5: #include <fstream.h>
 6: #include <conio.h>
 7: #pragma hdrstop
 8:
 9: int main(int argc, char **argv)
10: {
11:   char buff[81];
12:   ifstream infile;
13:   infile.open("readfile.cpp");
14:   if (!infile) return 0;
15:   while (!infile.eof()) {
16:     infile.getline(buff, sizeof(buff));
17:     cout << buff << endl;
18:   }
19:   infile.close();
20:   cout << endl << "Press any key to continue...";
21:   getch();
22:   return 0;
23: }
```



В строке 12 создается экземпляр класса ifstream с именем infile. Строка 13 открывает файл READFILE.CPP для чтения. В строке 14 проверяется, был ли файл успешно открыт, и если нет, программа завершает работу. Строка 15 является началом цикла. Заметьте, что условное выражение представляет собой вызов функции eof() класса istream. Эта функция возвращает true, если достигнут конец файла. В строке 16 из файла считывается одна строка текста с помощью функции getline(). Эта строка помещается в символьный массив buff. Затем содержимое массива выводится на экран. Наконец, строка 19 закрывает файл. Обратите внимание, что в строке

* Из книг, переведенных на русский язык, можно порекомендовать, например, «Освой самостоятельно Borland C++ 5.0», Бином, 1997.

4 в программу включается заголовочный файл IOSTREAM.H, содержащий объявление класса iostream. Как видите, благодаря Си++ и классу ifstream чтение текстового файла не требует больших усилий.

Один из конструкторов ifstream принимает char* в качестве аргумента, поэтому можно указать имя файла непосредственно при создании экземпляра этого класса. В этом случае строки 12 и 13 сливаются в одну:

```
ifstream infile("readfile.cpp");
```

Если вы создаете объект таким образом, вызов функции open() не требуется. Файл будет автоматически открыт конструктором.



Не забывайте о двойной наклонной черте в строковых константах! Например, открыть файл WIN.INI в каталоге Windows можно следующим образом:

```
ifstream infile("c:\\windows\\win.ini");
```

Я уже говорил об этом раньше, но почти уверен, что когда-нибудь вы совершите подобную ошибку. Поэтому, пользуясь случаем, хочу напомнить еще раз.

Вызов функции close() в строке 19 листинга 4.4 не является обязательным. Деструктор класса ifstream проверяет, не остался ли файл открытым, и при необходимости сам вызывает close(). Таким образом, файл в любом случае будет закрыт до разрушения экземпляра класса. В своих программах я почти всегда использую явный вызов close(). Это способствует документированию программы, указывая на завершение работы с файлом.



Программа из листинга 4.4 содержит небольшую ошибку. Из-за выбранного способа определения конца файла, до окончания цикла будет выведена лишняя пустая строка. Чтобы избежать этого, цикл нужно переписать следующим образом:

```
while (!infile.getline(buff, sizeof(buff)).eof()) {
    cout << buff << endl;
}
```

Последовательный вызов функций совершенно законен, но он путает неопытных программистов на Си++. Я специально не стал исправлять ошибку, чтобы облегчить чтение программы.

В силу того, что классы для работы с файлами происходят от iostream, вы можете использовать операторы вставки и извлечения из потока таким же образом, как и при консольном вводе-выводе. В листинге 4.4 использована функция getline(), потому что оператор извлечения (>>) останавливается на первом встреченном пробельном символе. (К *пробельным символам (white-space)* относятся собственно пробел, символ табуляции, и т.д.) Функция же getline() читает из файла символы до тех пор, пока не встретит символ конца строки (EOL — end-of-line). Оператор извлечения из потока очень полезен при чтении отдельных чисел, которые не содержат пробельных символов. В следующем фрагменте из файла считываются числа, которые затем выводятся на экран:

```
ifstream infile("somefile.dat");
while (!infile.eof()) {
  int x;
  infile >> x; // чтение числа из файла и присваивание его
               // переменной x
  cout << x << endl;
}
```

Обратите внимание, что в данном примере файл по-прежнему является текстовым, а не двоичным. Оператор извлечения автоматически преобразует текст в целое число.

Вывод в файл

Вывод в файл в некотором смысле проще, чем ввод, благодаря оператору вставки (<<). В целом процедура близка к той, что используется при чтении. Вместо создания экземпляра класса ifstream, вы создаете экземпляр класса ofstream и начинаете сбрасывать в него данные. Листинг 4.5 содержит программу, которая создает новый файл, записывает в него 10 строк текста и закрывает. Затем файл снова открывается для чтения, его содержимое считывается и выводится на экран.

Листинг 4.5. WRITEFIL.CPP

```
 1: #include <condefs.h>
 2: #include <stdio.h>
 3: #include <stdlib.h>
 4: #include <iostream.h>
 5: #include <fstream.h>
 6: #include <conio.h>
 7:
 8: #pragma hdrstop
 9:
10: int main(int argc, char **argv)
11: {
12:   char buff[81];
13:   cout << "Creating File..." << endl;
14:   ofstream outfile("test.dat");
15:   if (!outfile) return 0;
16:   cout << "Writing File..." << endl;
17:   for (int i = 0; i < 10; i++) {
18:     outfile << "This is line #" << i + 1 << endl;
19:   }
20:   outfile.close();
21:   cout << "Opening File for Input..." << endl;
22:   ifstream infile("test.dat");
23:   if (!infile) return 0;
24:   cout << "Reading File..." << endl << endl;
25:   while (!infile.eof()) {
26:     infile.getline(buff, sizeof(buff));
27:     cout << buff << endl;
28:   }
29:   infile.close();
30:   cout << endl << "Press any key to continue...";
31:   getch();
32:   return 0;
33: }
```


В строке 14 создается экземпляр класса ofstream и файл TEST.DAT. Затем в созданный файл записывается 10 текстовых строк. Строка 18 служит примером использования оператора вставки для записи в файл. Обратите внимание, что после текстовой строки в файл записывается целое число (i + 1). Оно автоматически преобразуется в строку и помещается

в выходной поток. Формирование строки файла завершает манипулятор endl. Этот цикл повторяется 10 раз. Строка 20 закрывает файл после завершения цикла. В данном случае закрытие файла необходимо, поскольку мы собираемся повторно открыть его для чтения. Для незакрытого файла это будет невозможно. В строках 25–28 для вывода на экран используется процедура, похожая на ту, что применялась в примере READFILE.CPP. После запуска программы из листинга 4.5 на экран будет выведено следующее:

```
Creating File...
Writing File...
Opening File for Input...
Reading File...
This is line #1
This is line #2
This is line #3
This is line #4
This is line #5
This is line #6
This is line #7
This is line #8
This is line #9
This is line #10
Press any key to continue...
```

Режимы открытия файлов

Файлы могут быть открыты в нескольких различных режимах. Например, при открытии файла с помощью класса ofstream по умолчанию создается новый файл. Это означает, что при использовании режима по умолчанию вы перезапишете существующий файл. Часто это совсем не то, что нужно. Например, иногда требуется добавить данные в конец существующего файла. В этом случае файл должен быть открыт в режиме добавления. Чтобы указать режим открытия файла, вы должны при создании объекта использовать один из спецификаторов open_mode класса ios в конструкторе ofstream:

```
ofstream outfile("test.dat", ios::app); // открыть в режиме добавления
```

В этом случае новые данные будут записаны в конец файла. Существует несколько спецификаторов, которые можно использовать при открытии файлов. В таблице 4.1 представлен список значений, которые может принимать перечисление open_mode, а также описания соответствующих режимов.

Таблица 4.1. Спецификаторы open_mode класса ios

Спецификатор	Описание
app	Файл открывается и новые данные добавляются в конец файла.
ate	Переход на конец файла при его открытии.
in	Файл открывается для ввода (чтения). Это режим по умолчанию для класса ifstream.
out	Файл открывается для вывода (записи). Это режим по умолчанию для класса ofstream.

Спецификатор	Описание
binary	Файл открывается в двоичном (бинарном) режиме. По умолчанию все файлы открываются в текстовом режиме. При чтении в текстовом режиме комбинации символов возврат-каретки/перевод-строки (CR/LF) преобразуются в один символ перевода строки (LF). При записи символ перевода строки преобразуется обратно в пару CR/LF. В двоичном режиме эти преобразования не производятся.
trunc	Файл открывается и его содержимое уничтожается. Если не указаны спецификаторы app или ate, этот режим устанавливается по умолчанию.
nocreate	Файл не будет создан, если он не существует. Функция open() не сработает, если файл не существует. Данный спецификатор можно использовать для проверки существования файла.
noreplace	Аналогичен nocreate, но функция open() не сработает, пока не будет указан спецификатор app или ate.

Вы можете использовать более одного спецификатора из таблицы 4.1, объединяя их с помощью операции or. Например, пусть вам нужно открыть файл в двоичном режиме и добавить данные в конец. В этом случае конструктор должен выглядеть следующим образом:

```
ofstream("test.dat", ios::app | ios::binary);
```

Результатом его работы будет открытие файла в двоичном режиме и перемещение указателя текущей позиции на конец файла. Новые данные будут записываться в конец файла.



Двоичные файлы обрабатываются не так, как к текстовые (см. спецификатор binary в табл. 4.1). Для обращения к двоичным файлам вы должны понимать, как работают функции read(), write(), put() и get(). У нас нет сейчас времени для изучения каждой из этих функций. Более подробную информацию о работе с двоичными файлами можно найти в какой-либо книге по Си++, где детально рассмотрен файловый ввод-вывод.

Элементарные действия с файлами выполняются очень легко. Но если вам потребуется более подробно узнать о работе с файлами, советую обратиться к справочным файлам C++Builder или приобрести хорошую книгу, в которой рассмотрен файловый ввод-вывод в Си++ (как и раньше, подойдет книга *Teach Yourself Borland C++ 4.5 in 21 Days, Second Edition* by Sams Publishing).

Заключение

Сегодня вы изучали классы в Си++. Хорошо разработанный класс прост в использовании и позволяет сохранить много часов работы программиста. Я бы даже сказал, что работать с хорошим классом просто приятно, особенно, если это ваше собственное творение. Ранее вы узнали о некоторых свойствах функций, реализованных только в Си++ – перегрузке, виртуальных функциях, подставляемых функциях и аргументах по умолчанию. Все это постоянно используется при разработке классов в Си++. В конце главы были рассмотрены основы файлового ввода-вывода в Си++.

Уроки первых четырех дней необходимы для понимания всего дальнейшего материала этой книги. Если вы в чем-то не полностью разобрались, не

отчаивайтесь. По мере продвижения вперед многие вещи будут повторяться и использоваться в программах, имеющих более реальное применение, чем консольные приложения, с которыми мы работали до сих пор.



Изучение Си++ может (скорее всего, так и будет) вызвать головную боль от перенапряжения. Это естественно, и вам не следует беспокоиться. Вы можете даже подумать: «Я никогда не одолею этого». Поверьте мне, вы справитесь.

Иногда просто необходимо отложить книгу на пару дней и дать всему устояться. Я полагаю, что вместо пятого дня можно было бы сделать пустую главу и назвать ее «День отдыха». Подождите немного, и в один прекрасный день вы будете, как Архимед, бегать вокруг своего офиса или дома с криком «Эврика!», озаренные наступившим прозрением. Но не забудьте одеться, ведь соседи могут подглядывать.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Как можно сделать функции-члены класса закрытыми для доступа извне и в то же время разрешить производным классам вызывать их?

Сделайте их защищенными. Защищенная функция недоступна для пользователя вашего класса, но может использоваться в производных классах.

Что такое абстрактное представление данных?

Абстрактное представление данных — это закрытие деталей класса, которые пользователь не должен видеть. Класс может содержать массу данных и функций, однако пользователю могут быть доступны лишь некоторые из них. Делайте видимыми (открытыми) только те функции, которые необходимы для использования класса.

Что такое объект?

Вообще говоря, объект — это любой блок кода, который можно рассматривать как отдельную единицу вашей программы. В Си++ под объектом в основном понимается класс. В C++Builder сюда добавляются компоненты VCL. Элементы управления OCX и ActiveX также можно рассматривать как объекты.

Может ли мой класс иметь более одного конструктора?

Да. Вы можете определять столько конструкторов, сколько необходимо, следуя при этом правилам написания перегруженных функций.

Должен ли я знать все детали файлового ввода-вывода в C++Builder?

Нет. C++Builder обеспечивает работу с файлами через компоненты. Однако в любом случае знание основ файлового ввода-вывода не помешает. Вообще говоря, все зависит от того, что должна делать ваша программа.

Можно ли открыть файл так, чтобы при необходимости производить как чтение, так и запись?

Да. В дополнение к классам, о которых я рассказывал в этой главе, существует также класс fstream. Он позволяет вам читать из файла, записывать в файл и перемещать при необходимости указатель текущей позиции в файле.

Вопросы

1. Чем отличаются классы от структур в Си++?
2. В чем причина использования закрытых членов-данных и функций?
3. Каким образом пользователь может получать и изменять значения закрытых членов-данных?
4. Как и когда вызывается деструктор класса?
5. Что означает замещение функции базового класса?
6. Как можно заместить функцию базового класса и в то же время сохранить все возможности, которые она предоставляет?
7. Что делает список инициализаторов?
8. Может ли класс наряду с членами-данными содержать экземпляры других классов?
9. Как можно объединить свойства двух различных классов в одном классе?

Упражнения

1. Напишите класс, который берет рост человека в дюймах и переводит его в футы.
2. Создайте на основе класса из упражнения 1 новый класс, который дополнительно переводит рост в метры, сантиметры и миллиметры. (**Указание**: один дюйм = 25,4 мм).
3. Напишите программу, которая принимает данные, вводимые пользователем, и записывает их в файл.
4. Измените программу из упражнения 3 так, чтобы она считывала записанный файл и выводила его содержимое на экран.
5. Отдохните один день. Вы заслужили это!

Библиотеки классов Си++ и модель визуальных компонентов



Кент Рейсдорф

Сегодня мы поговорим о библиотеках классов. Я расскажу вам, что они собой представляют и какие возможности появляются у вас для написания Windows-программ в сегодняшней быстро развивающейся индустрии программного обеспечения. Я буду рассматривать следующие библиотеки:

- Библиотеку OWL (Object Windows Library) фирмы Borland
- Библиотеку MFC (Microsoft Foundation Class Library) фирмы Microsoft
- Библиотеку VCL (Visual Component Library) фирмы Borland

Краткая история

В самом начале для написания подавляющего большинства программ под Windows использовался язык Си. Фактически, интерфейс прикладного программирования API (Application Programming Interface) представляет собой огромную коллекцию из сотен Си-функций. До сих пор тысячи программистов продолжают писать Windows-программы на Си.

Однажды несколько сотрудников фирмы Borland решили: «Это должно делаться проще!» (На самом деле революция в подходе к написанию программ для Windows началась одновременно с нескольких направлений, но Borland стала несомненным лидером в этой области.) Было очевидно, что программирование в Windows очень хорошо укладывается в концепции языка Си++, и наоборот. Создание классов, инкапсулирующих рутинные задачи, существенно облегчало труд программиста. Класс, созданный один раз для представления, скажем, окна, мог использоваться снова и снова. Революция началась.

Но я так еще и не сказал, что представляет собой библиотека классов.



Библиотека классов (class library) — это набор классов, инкапсулирующих часто используемые при программировании операции. Для библиотек классов также используется термин *frameworks*.

Стандартные библиотеки содержат классы для работы с окнами, управления редактированием, графических операций, отображения окон списков, растровых изображений, диалоговых окон и т.д.

Зачем нужны библиотеки классов?

Это хороший вопрос. Прежде всего, библиотеки делают программирование в Windows намного проще, чем при использовании обычного Си. Позвольте привести вам один пример. Листинг 5.1 содержит часть Windows-программы, написанной на Си. Данный фрагмент служит для загрузки растрового изображения с диска и вывода его в центре экрана. Сам текст будет для вас сейчас непонятен, но это не имеет значения.

Листинг 5.1. Си-код для загрузки и вывода на экран растрового изображения

```
 1: HPALETTE hPAL;
 2: BITMAPFILEHEADER bfh;
 3: BITMAPINFOHEADER bih;
 4: LPBITMAPINFO lpbi = 0;
 5: HFILE hFile;
 6: DWORD nClrUsed, nSize;
 7: HDC hDC;
 8: HBITMAP hBitmap;
 9: void _huge *bits;
10: do
11: {
12:   if ((hFile = _lopen(data.FileName, OF_READ)) == HFILE_ERROR) break;
13:   if (_hread(hFile, &bfh, sizeof(bfh)) != sizeof(bfh)) break;
14:   if (bfh.bfType != 'BM') break;
15:   if (_hread(hFile, &bih, sizeof(bih)) != sizeof(bih)) break;
16:   nClrUsed =
17:     (bih.biClrUsed) ? bih.biClrUsed : 1 << bih.biBitCount;
18:   nSize =
19:     sizeof(BITMAPINFOHEADER) + nClrUsed * sizeof(RGBQUAD);
20:   lpbi = (LPBITMAPINFO) GlobalAllocPtr(GHND, nSize);
21:   if (!lpbi) break;
22:   hmemcpy(lpbi, &bih, sizeof(bih));
23:   nSize = nClrUsed * sizeof(RGBQUAD);
24:   if (_hread(hFile, &lpbi->bmiColors, nSize) != nSize) break;
25:   if (_llseek(hFile, bfh.bfOffBits, 0) == HFILE_ERROR) break;
26:   nSize = bfh.bfSize - bfh.bfOffBits;
27:   if ((bits = GlobalAllocPtr(GHND, nSize)) == NULL) break;
28:   if (_hread(hFile, bits, nSize) != nSize) break;
29:   hDC = GetDC(hWnd);
30:   hBitmap = CreateDIBitmap(hDC, &(lpbi->bmiHeader), CBM_INIT,
31:                            bits, lpbi, DIB_RGB_COLORS);
32:   if (hBitmap) {
33:     LPLOGPALETTE llpal;
34:     DWORD nsize = sizeof(LOGPALETTE)
35:       + (nClrUsed - 1) * sizeof(PALETTEENTRY);
36:     lppal = (LPLOGPALETTE) GlobalAllocPtr(GHND, nSize);
37:     if (lppal) {
38:       lppal->palVersion = 0x0300;
39:       lppal->palNumEntries = (WORD) nClrUsed;
40:       hmemcpy(lppal->palPalEntry, lpbi->bmiColors,
41:       nClrUsed * sizeof(PALETTEENTRY));
```

```
42:        hPal = CreatePalette(lppal);
43:        (void) GlobalFreePtr(lppal);
44:      }
45:    }
46: } while(FALSE);
47: if (hFile != HFILE_ERROR) _lclose(hFile);
48: HPALETTE oldPal = SelectPalette(hDC, hPal, FALSE);
49: RealizePalette(hDC);
50: HDC hMemDC = CreateCompatibleDC(hDC);
51: HBITMAP oldBitmap = (HBITMAP)SelectObject(hMemDC, hBitmap);
52: BitBlt(hDC, 0, 0, (WORD)bih.biWidth, (WORD)bih.biHeight,
53:   hMemDC, 0, 0, SRCCOPY);
54: SelectObject(hMemDC, oldBitmap);
55: DeleteDC(hMemDC);
56: SelectPalette(hDC, oldPal, FALSE);
57: ReleaseDC(hWnd, hDC);
58: if (bits) (void) GlobalFreePtr(bits);
59: if (lpbi) (void) GlobalFreePtr(lpbi);
```

Не правда ли, это выглядит немного устрашающе? Мне даже пришлось обратиться за помощью к друзьям по телеконференции BCPPLIB сети CompuServe (спасибо, Пол!). Теперь взгляните на тот же пример, написанный с использованием библиотеки OWL фирмы Borland (листинг 5.2).

Листинг 5.2. OWL-код для загрузки и вывода на экран растрового изображения

```
1: TDib dib("test.bmp");
2: TPalette pal(dib);
3: TBitmap bitmap(dib, &pal);
4: TClientDC dc(*this);
5: dc.SelectObject(pal);
6: dc.RealizePalette();
7: TMemoryDC memdc(dc);
8: memdc.SelectObject(bitmap);
9: dc.BitBlt(0, 0, bitmapWidth(), bitmapHeight(), memdc, 0, 0);
```

Какой из вариантов вы предпочтете использовать? Для принятия решения вам даже не нужно знать, что именно делают эти фрагменты кода. Нетрудно заметить, что OWL-версия намного короче и легче для восприятия. (VCL облегчает работу еще больше, предоставляя компонент для работы с растровыми изображениями, который вы можете поместить в форму. Я не хочу сейчас забегать вперед и отложу на некоторое время рассмотрение этих возможностей.)

Рассмотренные примеры дают понять, что представляют собой в целом библиотеки классов. Библиотеки скрывают от вас те детали, знание которых не обязательно. Все, что содержится в листинге 5.1, выполняется OWL-кодом из листинга 5.2. Вам не нужно знать все детали того, как работают классы OWL; возможно, у вас и не возникнет такого желания. Все, что требуется — это взять объекты, которые входят в библиотеку, и использовать их в своей программе.

Хорошие библиотеки реализуют все достоинства ООП. Одни делают это лучше, другие — хуже. Библиотеки OWL и VCL фирмы Borland служат великолепным примером объектно-ориентированного подхода к программи-

рованию. Они обеспечивают тот уровень абстрактного представления данных, который требуется для профессиональной разработки приложений.

Какой ценой все это дается?

Нетрудно сообразить, что за облегчение работы приходится чем-то платить. Это действительно так. Программа, написанная с использованием библиотечных классов, в общем случае должна иметь больший размер и работать медленнее, чем ее аналог на Си. Однако это верно лишь отчасти. Приложения, использующие библиотеки классов, не обязательно будут медленнее программ, написанных на Си. Конечно, языку Си++ присуща некоторая избыточность, но для типичных Windows-программ это практически незаметно.

При использовании Си++ программа обычно становится больше по сравнению с ее аналогом, написанным на Си. Например, пусть у вас есть простая Windows-программа с исходным текстом на Си, имеющая размер 75 Кбайт. Та же программа, написанная с помощью библиотеки классов, может занять до 200 Кбайт. Это существенная разница, но здесь взят худший случай. Различие между приложениями, написанными на Си и Си++ с использованием библиотек классов, наиболее заметно для очень маленьких программ. При увеличении объема исходного текста и возрастании сложности программы, разница в размерах исполняемых файлов становится намного меньше.

Одной из причин такого поведения программ является просто различие между Си и Си++. Избыточность Си++ обусловлена реализацией дополнительных возможностей, к которым относятся обработка особых ситуаций, динамическая идентификация типов (RTTI — Run-Time Type Information) и другие. По моему мнению, увеличение размера кода является вполне приемлемой платой за те возможности, которые предоставляет Си++. Не думайте, что я являюсь сторонником раздувания кода. Напротив, я полагаю, что мы должны всегда писать настолько компактный код, насколько это позволяет используемый инструментарий. Но я также реалист, и понимаю, какие требования к скорости написания программ выдвигает сегодняшний рынок программного обеспечения. Я готов пойти на некоторое увеличение объема кода в обмен на ту мощь, которую дают мне Си++ и библиотеки классов.

Библиотеки классов как пример объектно-ориентированного программирования

Если вы всерьез увлечетесь программированием в Windows, то в конце концов заглянете в исходный текст вашей любимой библиотеки классов. Рано или поздно вам захочется узнать, как работают профессионалы. Исходные тексты OWL или VCL являются великолепным источником подобной информации.



Исходные тексты библиотеки MFC, возможно, не самый хороший пример объектно-ориентированного подхода к программированию. MFC не обладает в полной мере той элегантностью, достаточной степенью абстракции и общим дизайном, которые характерны для библиотеки высшей пробы. К тому же, в ней время от времени нарушаются правила ООП. MFC может быть самой популярной библиотекой, но это не означает, что она самая лучшая с точки зрения ООП.

Поначалу исходные тексты могут выглядеть устрашающе, но спустя некоторое время вы начнете понимать замысел разработчиков. Не напрягайтесь слишком сильно. Попытайтесь определить, где ваших знаний о Си++ недостаточно. Оставьте сложный материал на потом. Обратите внимание на то, как разработчики библиотеки используют в классах закрытый, защищенный и открытый уровни доступа; как и когда включаются подставляемые функции; как скрываются от пользователя детали, которые не должны быть видны. Изучение хорошей библиотеки классов может сильно помочь в освоении Си++ и объектно-ориентированного подхода к программированию.

Сравнение библиотек классов Си++

Все библиотеки классов нужно разделить на две категории: библиотеки Си++ и VCL. Сначала я расскажу о библиотеках Си++, а затем перейду к VCL. Фактически существуют только две жизнеспособные библиотеки Си++: OWL фирмы Borland и MFC фирмы Microsoft.

Библиотека объектов Windows фирмы Borland

Borland заняла лидирующее положение в соревновании библиотек классов, выпустив несколько лет назад OWL 1.0. Она продавалась отдельно и предназначалась для использования с компилятором Borland C++ 3.0. (В действительности, самая первая OWL была написана на Turbo Pascal и только затем переведена на Си++.) OWL 1.0 была хорошей библиотекой, но из-за проблем с правами собственности на синтаксис и других спорных вопросов, она не стала основой для последующих OWL. Тем не менее, OWL 1.0 сослужила хорошую службу всему сообществу программистов в Windows, положив начало массовому использованию библиотек классов. Нельзя сказать, что до OWL 1.0 не было других библиотек, но именно она первой вышла на массовый рынок.

После OWL 1 настало время OWL 2.0. Выпуск OWL 2 был значительным достижением. В ней были реализованы многие новые возможности Си++. Но лучше всего было то, что OWL 2 включалась в состав компилятора Borland C++ 4.0. Начиная с этого момента, Borland включает OWL во все свои пакеты Borland C++. Компиляторы Borland всегда первыми реализовывали новые возможности Си++, а OWL 2 способствовала их широкому применению. Из OWL 2 также был убран весь запатентованный синтаксис, присутствовавший в OWL 1. OWL 2 соответствовала стандартам Си++ и могла использоваться с любым компилятором — по крайней мере, теоретически. Но поскольку в очень немногих компиляторах были реализованы последние и наиболее мощные возможности Си++, OWL 2 фактически использовалась только с компиляторами фирмы Borland.

Затем Borland выпустила обновленную версию OWL 2.0, которая получила название OWL 2.5. По большей части изменения были незначительными. К самой OWL 2 было добавлено очень немного; исправлено несколько ошибок и включено несколько новых классов. Основное отличие OWL 2.5 заключалось в поддержке технологии OLE (Object Linking and Embedding — связывание и внедрение объектов) посредством библиотеки классов, получившей название Object Component Framework (OCF). С технической точки

зрения OCF не является частью OWL. Она превосходно работает с OWL, но может использоваться и независимо.

Последняя и самая мощная версия OWL — 5.0. OWL 5 представляет собой существенное расширение OWL 2.5. Основные изменения состояли во включении классов, инкапсулирующих новые элементы управления Win32. OCF в OWL 5.0 была также обновлена.

OWL обладает значительной мощью. Прежде всего, заслуживает упоминания ее архитектура. Очевидно, что OWL была хорошо продумана с самого начала. Я восхищаюсь ее разработчиками. OWL очень дружественна по отношению к ООП и следует всем правилам объектно-ориентированного программирования. Уровень абстрактного представления данных представляет собой хороший компромисс между легкостью использования и мощностью. OWL имеет одно большое преимущество перед своими конкурентами — она может использоваться как в 16-разрядных, так и в 32-разрядных программах. Borland даже пошла на эмуляцию некоторых элементов управления Win32 для использования в 16-разрядных приложениях. Хотя эта эмуляция не во всех случаях достаточно совершенна, она дает вам ощущение 32-разрядной среды в 16-разрядных программах.

OWL имеет и свои слабые стороны. В ней проделана большая работа по инкапсуляции окружения Windows, и это без всякого сомнения является крупным достоинством. Однако при таком уровне инкапсуляции OWL достаточно сложна, и иногда в ней трудно разобраться тем, кто только приступает к ее изучению. Сложность OWL рассматривается некоторыми как недостаток. Безусловно, чтобы стать мастером, потребуется время. Но разобравшись в OWL, вы сможете писать Windows-программы с очень большой эффективностью.

Библиотека базовых классов фирмы Microsoft

Библиотека MFC фирмы Microsoft была выпущена где-то между OWL 1 и OWL 2. MFC входит в комплект поставки компилятора Microsoft Visual C++. В действительности, версии MFC включаются в состав компиляторов Symantec, Watcom и, хотите верьте, хотите нет, Borland (а также, возможно, и других). Обычно Microsoft не дает лицензии на использование новейшей версии MFC другим производителям компиляторов (Symantec и Watcom), но в состав Borland C++ 5.01 была включена MFC версии 4.1, последней на тот момент (вскоре после этого вышла следующая версия, 4.2).

Нужно сказать, что MFC является другим типом библиотеки классов по сравнению с OWL. Она менее абстрактна и лежит ближе к API Windows. Сила MFC проявляется в трех основных областях. Прежде всего, ее относительно легко изучать. (Учтите, что ни одна из библиотек Си++, предназначенных для программирования в Windows, не является *легкой* для изучения, но в MFC немного проще разбираться по сравнению с другими.) В основном это связано с ее меньшей абстрактностью в некоторых областях. Если вы только начинаете программировать в Windows, то скорее всего не найдете между MFC и OWL большой разницы. Если вы уже имели опыт программирования на Си с использованием API, вам почти наверняка будет проще осваивать MFC.

Другое достоинство MFC, по мнению некоторых, это относительно небольшая надстройка над API Windows. Опять же, для тех программистов, которые переходят от Си к Си++ с использованием MFC, это достоинство. Они могут начать использовать MFC и чувствовать себя почти как дома.

Наконец, очевидным достоинством MFC является ее принадлежность к Microsoft. По мере расширения возможностей и появления новых технологий в Windows, MFC будет первой реализовывать их. Microsoft может выпустить новую технологию, а MFC будет поддерживать ее уже на этапе анонсирования.

MFC также имеет свои слабые стороны. Первая и наиболее существенная из них — это относительно небольшая надстройка над API Windows. «Но подождите!» — скажете вы. — «Я понял из ваших слов, что это как раз является достоинством!» Да, это так. Некоторые склонны считать близость MFC к API ее сильной стороной. Однако основная идея, лежащая в основе библиотеки классов — это изолирование пользователя от той информации, которую ему знать не обязательно. MFC во многих случаях не отвечает этому требованию. Те, кто программировал в Windows на Си, рассматривают это как достоинство. Вы можете сформировать свое собственное мнение. Кроме того, MFC не вполне соответствует концепциям ООП. Порой она выглядит наскоро подобранной коллекцией классов, плохо состыкованных друг с другом, а не единой тщательно спланированной и разработанной системой.

Другая проблема при использовании MFC заключается в том, что ее последние версии полностью 32-разрядные, как и компилятор Visual C++ 4.0. Хотя вы по-прежнему можете писать 16-разрядные приложения с помощью Microsoft Visual C++ 1.5 (который поставляется вместе с Visual C++ 4.0), его среда разработки вряд ли вас удовлетворит.

Кто же победитель?

Без сомнения, MFC распространена более широко, чем OWL. Частично это объясняется тем, что MFC и компилятор Visual C++ носят имя Microsoft. Не секрет, что Microsoft является королем в индустрии программного обеспечения для PC. Не является секретом и то, что Microsoft имеет маркетинговые возможности, о которых другие компании могут только мечтать.

Я убежден, что OWL является лучшей библиотекой классов. Те немногие, кто интенсивно использует как OWL, так и MFC, могли бы аргументировать эту точку зрения. Но MFC является библиотекой, которую сегодня выбирают чаще всего. Для этого есть много причин, и некоторые я уже упомянул. К другим причинам относится заметная переориентировка с Borland на Microsoft в последние годы. Некоторые менеджеры предпочитают покупать продукты «большой M» независимо от степени их технического совершенства. К счастью, это не может привести нас к полному отсутствию конкуренции. Индустрия программного обеспечения нуждается в компаниях, подобных Borland, которые способствуют ее развитию.

Какое будущее ожидает библиотеки классов Си++? Сейчас это нельзя сказать со всей определенностью. Возможно, что как MFC, так и OWL проиграют компонентам. Давайте теперь коротко рассмотрим библиотеку визуальных компонентов.

Библиотека визуальных компонентов

В 1995 году Borland выпустила новый продукт с названием Delphi, в котором была реализована концепция быстрой разработки приложений (RAD) с помощью так называемых *компонентов*. Компоненты — это объекты, ко-

торые можно поместить в форму и управлять ими через свойства, методы и события. Это можно назвать визуальным программированием.

Концепция визуального программирования была впервые реализована в системе Visual Basic фирмы Microsoft. Однако в отличие от Visual Basic, языком программирования Delphi была выбрана разновидность Паскаля. Этот новый язык, названный *Object Pascal (Объектный Паскаль)*, являлся объектно-ориентированной версией Паскаля. По отношению к обычному Паскалю он занимает то же положение, что и Си++ по отношению к Си. Delphi и Object Pascal представляли собой союз объектно-ориентированного и визуального программирования. В дополнение ко всему, Delphi мог создавать автономные исполняемые файлы — *настоящие* программы, которые не требовали для работы загрузки DLL; программы, которые компилировались, а не интерпретировались; программы, которые работали в десятки раз быстрее программ на Visual Basic. Это произвело сильное впечатление на всех, кто занимался программированием.

Delphi не просто предоставлял в ваше распоряжение Object Pascal. Вместе с ним поставлялась библиотека визуальных компонентов (VCL), которая была предназначена для программирования в Windows с использованием Object Pascal. Но VCL нельзя просто сравнивать с OWL и MFC. Да, это библиотека классов, но ее ядро существенно иное. Различие состоит прежде всего в том, что VCL основана на концепции свойств, методов и событий.

Вы можете удивиться, почему я говорю о Delphi. Причина проста — библиотеки VCL, которые составляют центральную часть Delphi и C++Builder, очень похожи. Это может оказаться для вас неожиданным. Если вы изучали раньше Си++, то задумаетесь, как такое может работать. Если вы изучали Паскаль, то, вероятно, улыбнетесь. Если же вы приступили к изучению C++Builder, имея другую подготовку, то сказанное могло не произвести на вас никакого впечатления. В конце концов, не имеет значения, почему это работает. Давайте немного глубже заглянем в VCL.

Компоненты

Как я говорил в первый день, «Первое знакомство», компоненты VCL представляют собой объекты, выполняющие специфические задачи. Компоненты VCL оформлены как классы Object Pascal. С этого момента мы будем встречаться с компонентами ежедневно. Я не буду сейчас детально рассматривать компоненты, поскольку вы будете встречать примеры их использования на протяжении всей книги. Завтра я расскажу о компонентах более подробно.

Свойства, методы и события

В первый день я дал вам краткое введение в модель свойств, методов и событий. Эта триада составляет внешний интерфейс компонентов в VCL (т.е. ту часть компонентов, с которой сталкивается пользователь). Давайте по очереди рассмотрим каждую составляющую.

Свойства

Свойства компонентов определяют их работу. Некоторые свойства одинаковы для разных компонентов. Все визуальные компоненты, например, имеют свойства Top и Left. Эти два свойства определяют положение компонента относительно формы. Для такого компонента, как сама форма, свойства

Top и Left определяют положение формы на экране. Все компоненты имеют свойство Owner, которое используется VCL для отслеживания компонентов, порожденных отдельной формой или самим компонентом.

Одна картинка обычно стоит тысячи слов, поэтому давайте снова запустим C++Builder, и я покажу вам свойства в действии. После запуска C++Builder на экране должны находиться пустая форма и инспектор объектов.

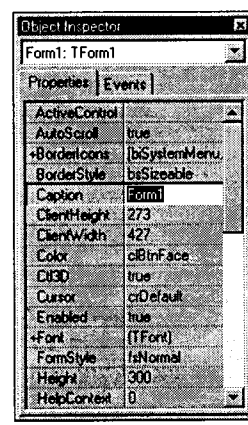


Если в опциях C++Builder установлено сохранение рабочей конфигурации при выходе, то после запуска вы можете увидеть последний проект, над которым работали. В этом случае выберите в главном меню пункт File | New Application, чтобы получить пустую форму.

Сейчас инспектор объектов должен выглядеть примерно как на рис. 5.1. Если необходимо, щелкните на вкладке Properties окна инспектора объектов, чтобы увидеть список свойств. Свойства компонента расположены в алфавитном порядке. Если компонент имеет больше свойств, чем помещается в окне, можно воспользоваться линейкой прокрутки. Окно инспектора объектов можно перемещать и растягивать. Я предпочитаю делать его максимально высоким, чтобы видеть как можно больше свойств одновременно. Найдите в списке свойство Left и щелкните на нем. Измените числовое значение этого свойства (допустимо любое число от 0 до 600) и нажмите Enter на клавиатуре. Вы увидите, как при этом передвинется форма.

Этот пример иллюстрирует важный аспект свойств — они представляют собой больше, чем простые члены-данные класса. Каждому свойству соответствует определенный член класса, но сами свойства не является членами-данными. Изменение свойства часто приводит к неявному выполнению некоторого блока кода.

Рис. 5.1
Окно инспектора объектов





Свойства часто связаны с *методами доступа (access methods)*, которые выполняются при изменении свойств.



В этом месте начинается небольшая путаница. Как я говорил, VCL написана на Паскале. В Паскале используется термин *метод (method)* вместо термина *функция (function)* в Си++. Чтобы запутать вас еще больше, функцией в Паскале называется метод, который возвращает значение, а метод, который не возвращает значения, называется *процедурой (procedure)*. Я, будучи старым Си++ хакером, с радостью называл бы их всех функциями, но при обсуждении VCL придется использовать терминологию Паскаля. В большинстве случаев я буду использовать более общий термин *метод*.

Значения свойств могут изменяться во время разработки (когда вы проектируете свою форму) и во время выполнения программы. В любом случае, если свойство имеет метод доступа, он будет вызван и выполнен при изменении свойства. Вы уже видели пример изменения свойства на этапе разработки, когда установили новое значение для свойства Left и форма передвинулась по экрану. Это является одной из сильных сторон VCL. В C++Builder вы можете постоянно видеть на экране результат своих действий. Не все свойства способны производить видимые изменения в форме на этапе разработки, но когда это возможно, результат присваивания свойству нового значения немедленно отображается.

Для изменения значения свойства во время выполнения программы, вам просто нужно выполнить соответствующее присваивание. При этом VCL автоматически вызывает метод доступа для данного свойства. Для изменения значения свойства Left во время работы программы можно использовать примерно следующий код:

```
MainForm->Left = 200;
```

В случае свойства Left (а также Top) VCL сдвигает и перерисовывает форму. (Те, кто работал с API Windows, могут сообразить, что в конечном итоге это сводится к вызову функций SetWindowPos() и InvalidateRect().)



Обратите внимание, что в предыдущей строке кода для установки значения свойства использован оператор косвенного доступа (->). Все компоненты VCL размещаются в «куче», поэтому для доступа к их свойствам и методам всегда используется этот оператор. Классы, которые вы пишете сами, могут размещаться как в «куче», так и в стеке, но все классы компонентов VCL и все их производные классы должны размещаться только в «куче».



Свойства имеют два *спецификатора доступа (access specifiers)*, которые используются при чтении и модификации их значений. Это *спецификатор чтения (read specifier)* и *спецификатор записи (write specifier)*.

Спецификаторы доступа связывают со свойством методы (функции) чтения и записи. Когда значение свойства считывается или изменяется, автоматически вызывается функция, связанная с этим свойством. Во время присваивания в предыдущем примере вы обращались к спецификатору записи. При этом VCL проверяла наличие метода доступа для спецификатора записи. Если метод существует, он вызывается, в противном случае VCL просто присваивает значение соответствующему члену класса.

Когда вы ссылаетесь на свойство (используете свойство справа от оператора присваивания), вы обращаетесь к спецификатору чтения:

```
int x = MainForm->Left;
```

В этом случае VCL вызывает спецификатор чтения для получения значения свойства Left. Во многих случаях спецификатор чтения не делает почти ничего, кроме возврата текущего значения свойства.

Свойства свойств (извините, я не могу написать иначе) определяются разработчиком компонента. Свойство может быть доступно только для чтения (read-only), и в этом случае вы можете получить значение свойства, но не можете его изменить. В редких случаях свойство может быть доступно только для записи (write-only).

Наконец, некоторые свойства могут быть доступны только во время выполнения (runtime-only). Доступ к такому свойству невозможен на этапе разработки, и оно не отображается в окне инспектора объектов. Свойство типа runtime-only может быть объявлено также и как read-only, что означает возможность доступа только во время выполнения и только для чтения.

Некоторые свойства используют массивы в качестве соответствующих членов-данных. Для иллюстрации поместим компонент Memo в нашу пустую форму. Выберите в палитре компонентов вкладку Standard и щелкните на кнопке Memo. (Используйте всплывающие подсказки, чтобы определить, на какой кнопке находится курсор мыши.) Теперь переместите курсор на форму и щелкните в том месте, где вы хотите расположить верхний левый угол компонента. После этого инспектор объектов переключится на отображение свойств только что размещенного компонента, в данном случае TMemo. Найдите свойство Lines и щелкните на нем. Заметьте, что значение свойства представляет собой текст (TStrings), а справа от значения расположена маленькая кнопка с многоточием (...).



Наличие кнопки с многоточием говорит о том, что данное свойство можно редактировать в отдельном диалоговом окне. Для массива строк, например, в диалоговом окне можно набрать строку. Для свойства Font щелчок на кнопке с многоточием вызовет диалоговое окно выбора шрифта. Тип вызываемого редактора зависит от свойства, но некоторые свойства используют один и тот же редактор. Вы можете вызвать редактор свойств, щелкнув на кнопке с многоточием, или дважды щелкнув на значении свойства.

Свойство Lines компонента Memo представляет собой массив строк. Когда вы дважды щелкнете на значении этого свойства, на экране появится редактор строк и вы сможете ввести текст, который должен отображаться в компоненте Memo после запуска программы. Если вы не хотите выводить какой-либо текст, удалите все строки из окна редактора.

Свойства могут быть экземплярами других классов VCL. Очевидный пример — свойство Font. Оно включает в себя начертание, цвет, размер шрифта и т.д. Найдите свойство Font в окне инспектора объектов. (Не имеет значения, какой компонент при этом выбран — Memo или сама форма.) Обратите внимание, что перед словом Font стоит знак плюс. Это означает, что для данного свойства существует набор дополнительных свойств. Если вы дважды щелкнете на имени свойства, ниже появится список его отдельных элементов, которые теперь можно изменять. Для редактирования можно также вызвать редактор свойств (Property Editor). В обоих случаях конечный результат будет одинаковым.



Некоторые свойства представляют собой *набор (set)* других свойств.

Свойство Style объекта Font — хороший пример набора. Заметьте, что перед Style стоит знак плюс. Если вы дважды щелкнете на свойстве Style, откроется дополнительный список. В этом случае набор состоит из различных стилей, доступных для данного шрифта: полужирный, курсивный, подчеркнутый и перечеркнутый. Двойным щелчком на стиле можно включить или выключить этот стиль (установить значение true или false). Некоторые свойства могут представлять собой *перечисления (enumerations)*.



Перечисление (enumeration) — это список возможных значений свойства.

Когда вы щелкаете на свойстве перечислимого типа, справа от его значения появляется кнопка со стрелкой, направленной вниз. Чтобы увидеть список возможных значений, щелкните на этой кнопке. Можно также дважды щелкнуть на самом значении свойства. При этом инспектор объектов будет циклически перебирать (перечислять) все возможные значения. Свойство Cursor является хорошим примером перечислимого свойства. Выберите это свойство и щелкните на кнопке со стрелкой, чтобы увидеть список курсоров, которые вы можете выбрать.

Пока у вас запущен C++Builder и на экране находится пустая форма, вы можете некоторое время поисследовать различные компоненты и их свойства. Итак, вперед; я подожду.

Правила работы со свойствами

- Свойства подобны членам-данным класса и могут быть доступны аналогичным образом.
- Свойства *не являются* членами-данными. Они относятся к специальной категории членов класса.
- Когда свойству присваивается значение, часто, но не всегда, вызывается метод записи. Это зависит от реализации конкретного свойства.
- Свойства не могут быть использованы в качестве аргументов при вызове функции.
- Свойства обычно имеют значения по умолчанию, которые отображаются в окне инспектора объектов при первом использовании компонента. Эти значения используются при отсутствии присваивания других значений.
- Свойства могут иметь тип read-only, write-only или runtime-only.
- Свойства типа runtime-only не отображаются в окне инспектора объектов и могут быть изменены только после запуска программы.
- Свойства могут включать в себя:
  - Простые типы данных
  - Массивы
  - Наборы
  - Перечисления
  - Объекты классов VCL

Методы

Методы в компонентах VCL — это функции (точнее, *процедуры* и функции), которые могут быть вызваны для выполнения компонентом определенных действий. Например, все визуальные компоненты имеют метод с именем Show() для отображения компонента, и метод с именем Hide() для скрытия компонента. Эти методы вызываются точно так же, как и функции-члены классов (см. День 3, «Указатели, ссылки и функции в Си++»):

```
MyWindow->Show();
// какой-то код, затем...
MyWindow->Hide();
```

По терминологии Си++, методы являются функциями-членами класса данного компонента. Методы в VCL могут быть объявлены как открытые, защищенные или закрытые, точно так же, как и функции в Си++. Ключевые слова public, protected и private имеют в классах Object Pascal то же значение, что и в классах Си++. Открытые методы могут быть доступны пользователю компонента. В данном примере оба метода, Show() и Hide(), являются открытыми. Защищенные методы недоступны для пользователя, но доступны для классов (компонентов), являющихся производными данного компонента. Закрытые же методы, разумеется, могут быть доступны только в пределах самого класса.

Аналогично функциям Си++, одни методы принимают аргументы и возвращают значения, а другие нет. Это определяется разработчиком компонента. Например, метод GetTextBuf() получает текст компонента TEdit. Этот метод может быть использован для получения введенного текста следующим образом:

```
char buff[256];
int numChars = EditControl->GetTextBuf(buff, sizeof(buff));
```

Как видите, данный метод принимает два аргумента и возвращает целое число. В результате его вызова содержимое окна редактирования помещается в buff и возвращается количество введенных символов.

Пока это все, что вам нужно знать для использования методов. Когда я буду рассказывать о написании компонентов, мы рассмотрим методы более детально.

Правила работы с методами

- ☐ Методы могут быть закрытыми, защищенными или открытыми.
- ☐ Методы вызываются с помощью оператора косвенного доступа.
- ☐ Методы могут принимать аргументы и возвращать значения.
- ☐ Некоторые методы не принимают аргументов и не возвращают никакого значения.
- ☐ Процедура — это метод, который не возвращает значения.
- ☐ Функция — это метод, который возвращает значение.
- ☐ Только открытые методы могут быть вызваны пользователями компонента.

События



Windows является средой, *управляемой событиями (event-driven)*. Это означает, что программа управляется событиями, происходящими в среде Windows. К событиям относятся перемещение мыши, щелчки кнопками мыши и нажатия клавиш на клавиатуре.

Программисты, работавшие раньше в среде DOS или на больших ЭВМ, могут испытывать затруднения при переходе к концепции управляющих событий. Прикладная программа непрерывно опрашивает Windows на предмет возникновения событий. К событиям в Windows относятся активизация меню, нажатие кнопки, перемещение окна, требование перерисовки окна, активизация окна и т.д. Windows уведомляет программу о событии, посылая соответствующее *сообщение (message)*. Количество сообщений, которые могут быть посланы приложению, составляет около 175. Это огромное количество! К счастью, вам не нужно знать все из них, чтобы программировать в C++Builder; достаточно лишь пары десятков, которые используются наиболее часто.

События в VCL — это все происходящее в компоненте, о чем пользователь может захотеть узнать. Каждый компонент спроектирован так, чтобы реагировать на определенные события. Обычно имеются в виду события Windows, но это не обязательно. Например, компонент Button (кнопка) будет реагировать на щелчок мышью, как и следовало ожидать. Но невизуальные элементы управления (например, компоненты доступа к базам данных) могут реагировать на события, не относящиеся к Windows, например, достижение пользователем конца таблицы.



Когда вы реагируете на событие, относящееся к данному компоненту, это называется *обработкой (handle)* события. События обрабатываются функциями, которые называются *обработчиками событий (event handlers)*.

Обработка событий в VCL крайне проста. Список событий, на которые реагирует компонент, приведен во вкладке Events окна инспектора объектов. Имя события служит одновременно его описанием. Например, щелчок мышью называется OnClick.



Вы не обязаны обрабатывать все события, на которые реагирует компонент. На практике это требуется редко. Если вы не обрабатываете какое-либо событие, соответствующее сообщение либо отбрасывается, либо обрабатывается предопределенным образом. Вы можете обрабатывать любые события, которые вас интересуют, и игнорировать все остальные.

Вы лучше почувствуете все сказанное, если примените его на практике. Для начала давайте создадим новое приложение. Выберите в главном меню пункт File | New Application. Если появится запрос на сохранение текущего проекта, ответьте No. Теперь на экране снова пустая форма. Давайте сначала установим свойства главного окна:

1. Измените значение свойства Name на PMEForm (PME — сокращение для «properties, methods, and events»).
2. Измените значение свойства Caption на PME Test Program.

Затем нам нужно добавить к форме компонент Memo.

1. Выберите вкладку Standard в палитре компонентов и щелкните на кнопке Memo.
2. Щелкните на форме, чтобы поместить в нее компонент Memo.
3. Измените значение свойства Name на Memo. Убедитесь, что в данный момент выделен именно компонент Memo, чтобы случайно не изменить имя формы.
4. Дважды щелкните на свойстве Lines в столбце значений. На экране появится редактор списка строк.
5. Сотрите слово Memo и наберите A test program using properties, methods and events. Щелкните на кнопке OK, чтобы закрыть редактор списка строк.
6. Растяните компонент Memo так, чтобы он занимал большую часть формы. Внизу оставьте место для кнопки.

Ваша форма должна выглядеть так, как на рис. 5.2.

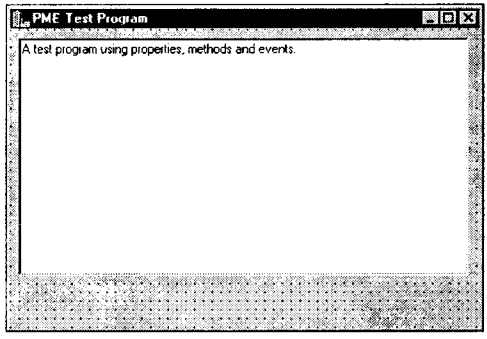


Рис. 5.2
Форма с добавленным компонентом Memo

Теперь давайте поместим в форму кнопку.

1. Выберите вкладку Standard в палитре компонентов и щелкните на кнопке компонента Button.
2. Щелкните на форме под компонентом Memo, чтобы поместить туда кнопку.
3. Измените значение свойства Name для кнопки на Button.
4. Измените значение свойства Caption на Show/Hide.
5. Расположите кнопку по центру формы.



Вы можете отцентрировать компонент визуально, но для большей точности можно использовать палитру выравнивания (Alignment Palette). Выберите в главном меню пункт View | Alignment Palette и затем щелкните на кнопке Center horizontally (отцентрировать по горизонтали).

Мы будем использовать эту кнопку для отображения и скрытия компонента Memo. Теперь нужно написать некоторый код, чтобы кнопка заработала. Убедитесь, что компонент Button выделен, и щелкните на вкладке

Events окна инспектора объектов. Вы увидите список событий, на которые реагирует компонент. Первым в списке должно стоять событие OnClick. Дважды щелкните на столбце значений события OnClick. То, что произойдет дальше — одно из величайших достижений визуального программирования. Окно редактора кода переместится на передний план и отобразит функцию OnClick, готовую для ввода кода. На рис. 5.3 показан редактор кода с обработчиком OnClick.

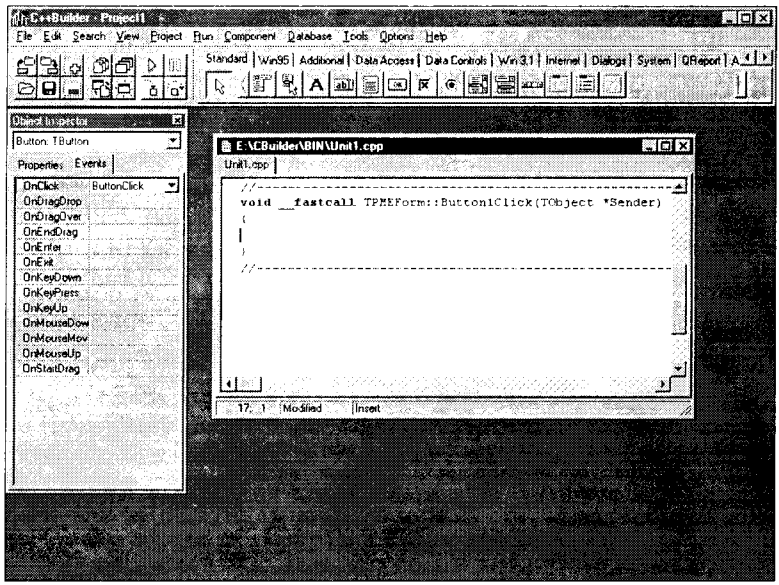


Рис. 5.3 Редактор кода C++Builder с отображенным обработчиком OnClick



Эта функция выглядит подобно функции-члену класса, которые обсуждались на третий день, и фактически ею и является. Единственное различие состоит в том, что функции C++Builder используют ключевое слово __fastcall (обратите внимание, что перед fastcall стоят два знака подчеркивания). Не так важно понимать, что означает __fastcall, главное — знать, что все функции C++Builder используют это соглашение о вызовах.

Возможно, вы захотите сделать окно редактора кода немного шире, чтобы видеть весь отображаемый текст. Перед тем, как двинуться дальше, сохраните проект. Выберите в главном меню пункт File | Save. Прежде всего, у вас будет запрошено имя модуля (исходного файла). Наберите PMEMain и щелкните на кнопке OK. Затем последует запрос на ввод имени проекта. Наберите PMETest и снова щелкните на кнопке OK или нажмите Enter.

Обратите внимание, что функция уже приготовлена для вас, и все, что осталось сделать — это ввести необходимый код. Если вы внимательнее посмотрите на эту функцию, то заметите, что она называется ButtonClick, является членом класса TPMEForm, возвращает void и принимает в качестве аргумента указатель с именем Sender на объект класса TObject. (Я расскажу об аргументе Sender чуть позже.) Все, что осталось сделать — это ввести код, который будет отображать или скрывать компонент Memo при последовательных нажатиях кнопки. Мы заимствуем часть кода из предшеству-

ющего обсуждения методов. Отредактируйте функцию ButtonClick в соответствии с образцом:

```
void __fastcall TPMEForm::ButtonClick(Tobject *Sender)
{
  static bool isVisible;
  isVisible = !isVisible;
  if (isVisible) Memo->Hide();
  else Memo->Show();
}
```

В этом коде объявляется статическая переменная с именем isVisible.



Статическая переменная (static variable) сохраняет свое значение между вызовами функции.

Статические переменные являются исключением из правила, касающегося неинициализированных переменных — они по умолчанию инициализируются нулем. В данном случае isVisible является булевой переменной и инициализируется значением false.

Вторая строка кода функции меняет значение булевой переменной на противоположное, применяя операцию логического отрицания к текущему значению. Это происходит следующим образом. Изначально статическая переменная установлена в false. При первом выполнении функции переменной присваивается значение NOT false, что, разумеется, есть true. При следующем вызове функции переменной присваивается NOT true, и так далее. В результате каждый раз при выполнении функции переменная isVisible содержит значение, обратное по отношению к значению в предыдущем вызове. Затем с помощью пары if/else вызывается либо Show(), либо Hide(), в зависимости от значения isVisible.



Помните, как вы изменили значение свойства Name компонента Memo на Memo? Когда вы делали это, C++Builder работал за кулисами. Сначала он создал новый класс на основе класса TMemo. Затем создал динамический экземпляр нового класса и присвоил ему имя Memo. В нашем коде Memo является указателем на объект. Вот почему доступ к функциям Show() и Hide() осуществляется с помощью оператора косвенного доступа.

Вот и все! Но работает ли это? Давайте проверим. Щелкните на кнопке Run оперативной панели. После компиляции программа будет запущена и на экране появится ее окно. Вот и настал момент истины. Щелкните на кнопке, и компонент Memo исчезнет. Щелкните на кнопке еще раз, и он снова появится. Ура! Заработало! Наигравшись с программой, закройте ее (с помощью кнопки Close Program в верхнем левом углу строки заголовка). Вы окажетесь снова в редакторе кода.

Однако все эти манипуляции со статической булевой переменной немного сложноваты. Вернемся назад к обсуждению свойств. Не лучше ли было компоненту Memo иметь специальное свойство, которое позволяло бы узнать, является ли он в данный момент видимым? Конечно, такое свойство есть! Как можно было ожидать, оно называется Visible. Давайте его используем. Отредактируйте функцию еще раз, пока она не будет выглядеть так, как здесь:

```
void __fastcall TPMEForm::ButtonClick(TObject *Sender)
{
  if (Memo->Visible) Memo->Hide();
  else Memo->Show();
}
```

Снова запустите программу. Кнопка будет работать так, как ей положено. Итак, мы ухитрились использовать свойства, методы и события в одном примере.

Как вы заметили, функция ButtonClick() принимает указатель с именем Sender на объект TObject. Каждая функция-обработчик события будет иметь по крайней мере один аргумент Sender. В зависимости от события, функция может также принимать один или несколько дополнительных аргументов. Например, обработчик события OnMouseDown выглядит следующим образом:

```
void __fastcall TPMEForm::ButtonMouseDown(TObject *Sender,
  TMouseButton Button, TShiftState Shift, Integer X, Integer Y)
{
}
```

В данном случае вы получите информацию о том, какая кнопка мыши была нажата, какая при этом была нажата клавиша на клавиатуре и координаты (x,y) курсора во время нажатия кнопки мыши. Функция-обработчик содержит всю информацию, которая необходима для реакции на событие.

Что же представляет собой Sender? Это указатель на компонент, который посылает сообщение обработчику. В нашем примере аргумент Sender является излишним, поскольку мы знаем, что сообщение исходит от кнопки Show/Hide. Вообще, Sender позволяет иметь один обработчик событий для нескольких компонентов. Для иллюстрации, давайте создадим новую кнопку и сделаем одну из кнопок кнопкой Show, а другую — кнопкой Hide.

1. Если редактор кода находится на переднем плане, нажмите F12, чтобы переключиться обратно в редактор форм.
2. Выделите кнопку Show/Hide, щелкнув на ней мышью. Измените значения свойств Name и Caption на Show.
3. Добавьте к форме новую кнопку справа от кнопки Show. Если хотите, выровняйте кнопки, чтобы форма выглядела лучше.
4. Измените значение свойства Name новой кнопки на Hide. Значение свойства Caption также изменится на Hide после того, как вы нажмете Enter.
5. Щелкните на кнопке Show, а затем на вкладке Events инспектора объектов. Обратите внимание, что для события OnClick теперь указано имя обработчика ShowClick. Замените его снова на ButtonClick. (Начальное имя обработчика устанавливается по умолчанию. Вы можете изменить его на любое другое.)
6. Щелкните на кнопке Hide и найдите событие OnClick в окне инспектора объектов. Справа будет находиться кнопка со стрелкой. Щелкните на ней и выберите ButtonClick из выпадающего списка (в нем сейчас должно быть только одно имя функции).
7. Дважды щелкните на значении ButtonClick. Окно редактора кода окажется на переднем плане и курсор будет располагаться в функции ButtonClick(). Измените код так, чтобы он выглядел следующим образом:

```
void __fastcall TPMEForm::ButtonClick(Tobject *Sender)
{
  if (Sender == Hide) Memo->Hide();
  else Memo->Show();
}
```

8. Выпекайте при 220 градусах 1 час или до золотисто-коричневого цвета. (Я просто хочу убедиться, что вы еще не заснули.)

Сейчас ваша форма должны выглядеть как на рис. 5.4. Откомпилируйте и запустите программу. Щелкните на каждой кнопке и убедитесь, что они работают, как предполагалось.

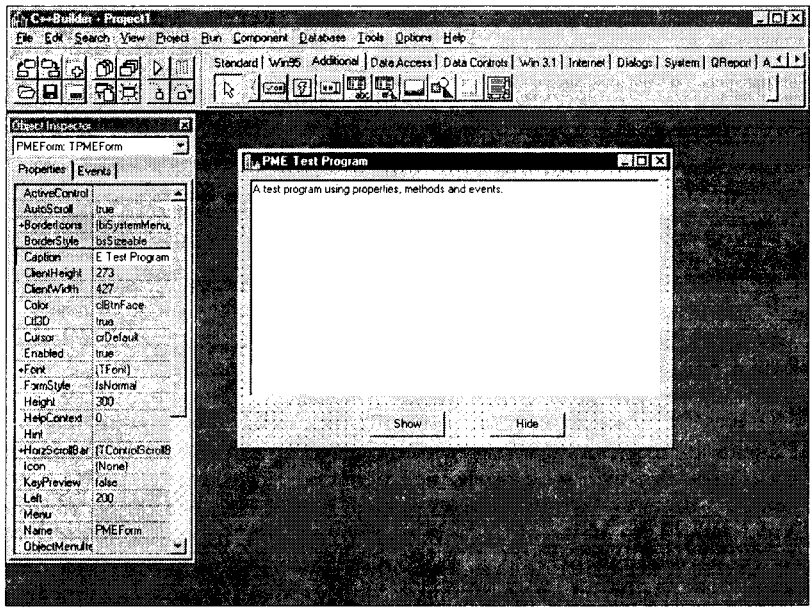


Рис. 5.4
Форма со всеми добавленными компонентами

Итак, мы создали одну функцию, которая обрабатывает события OnClick для обеих кнопок. Мы использовали параметр Sender, чтобы определить, какая кнопка послала сообщение о событии OnClick, и затем скрыть или показать компонент Memo. Мы могли создать отдельные обработчики для каждой кнопки, но тогда код стал бы менее компактным. Кроме того, это хороший пример использования параметра Sender.

Вы могли заметить, что после создания обработчика события OnClick для одного из компонентов, можно связать этот обработчик с событием OnClick любого компонента формы. В дальнейшем я расскажу о событиях подробнее.

Правила работы с событиями

- ☐ Вы можете при необходимости отреагировать на любое событие, поддерживаемое компонентом.
- ☐ Вы не обязаны реагировать на все события, определенные для компонента.
- ☐ События обрабатываются специальными функциями, которые называются *обработчиками событий*.
- ☐ Несколько компонентов могут использовать общий обработчик событий.
- ☐ Имена обработчиков событий, которые C++Builder присваивает им по умолчанию, могут быть изменены программистом.
- ☐ Изменяйте имена обработчиков событий только через инспектор объектов.
- ☐ Аргумент Sender, передаваемый обработчику события, может использоваться для определения компонента, пославшего сообщение о событии.
- ☐ Двойной щелчок на имени обработчика события в окне инспектора объектов вызывает код этого обработчика.
- ☐ Каждый обработчик события содержит параметры, необходимые для правильной реакции на событие.

C++Builder и VCL

Как я уже говорил, библиотека VCL была написана на языке Object Pascal и предназначалась для Delphi. При создании C++Builder не было никаких причин начинать все с нуля, поэтому разработчики фирмы Borland взяли существующую библиотеку и адаптировали ее к новой среде. Дополнительный выигрыш состоял в том, что пользователи Delphi могли легко перейти к C++Builder, и наоборот. Из-за использования одной и той же VCL вам не нужно изучать новую библиотеку, работая внутри семейства Delphi/C++Builder.

C++Builder является компилятором Си++, а VCL — это библиотека Object Pascal. Как же они стыкуются? По правде говоря, вам следует беспокоиться не о том, как все это работает на уровне компилятора, а скорее о том, какое влияние оказывает данный факт на программирование в C++Builder. Вывод таков: VCL написана на Объектном Паскале и в то же время практически невидима. Возьмите, например, следующий фрагмент кода:

```
int screenW = GetSystemMetrics(SM_CXSCREEN);
int screenH = GetSystemMetrics(SM_CYSCREEN);
int h = MainForm->Height;
int w = MainForm->Width;
MainForm->Top = (screenH / 2) - (h / 2);
Mainform->Left = (screenW / 2) - (w / 2);
TPoint cPt;
GetCursorPos(cPt);
h -= 150;
w -= 150;
MainForm->Height = h;
MainForm->Width = w;
```

```
for (int i = 0; i < 150; i += 6) {
  MainForm->Height = h + 1;
  MainForm->Width = w + 1;
  SetCursorPos(
    MainForm->Left + MainForm->Width,
    MainForm->Top + MainForm->Height);
}
SetCursorPos(cPt.x, cPt.y);
```

Где здесь Объектный Паскаль и где Си++? Для вас это все Си++. VCL и Си++ работают как одно целое, чтобы предоставить вам возможность быстрой разработки приложений на Си++. VCL обеспечивает RAD за счет компонентов, а остальная часть кода может быть написана на Си++.

VCL для программиста на Си++

Этот раздел предназначен для опытных программистов на Си++, переходящих к C++Builder, но может быть полезен и для начинающих. Хотя приведенные здесь сведения не очень сложны, они все же рассчитаны на профессионалов, поэтому в крайнем случае вы можете просто продолжить чтение со следующего раздела.

При переходе к C++Builder некоторые вещи могут показаться странными для программистов на Си++. Прежде всего, не забывайте, что VCL написана на Объектном Паскале, а не на Си++. Не пытайтесь переводить ее на Си++. Система работает превосходно, пока вы понимаете это и не выходите за определенные рамки. Весь ваш код будет написан на Си++, но помните, что сама VCL — это не Си++. Держа это в уме, позвольте мне рассказать о некоторых вещах, которые нужно учитывать при использовании VCL.

Все объекты VCL должны размещаться динамически

Когда вы помещаете компонент в форму, C++Builder автоматически создает код для его динамического размещения, поэтому вам не нужно об этом заботиться. Однако может возникнуть необходимость создавать и использовать классы VCL во время выполнения программы. Например, пусть вам нужно отобразить диалоговое окно открытия файла, а в форме отсутствует компонент TFileOpen. Никаких проблем — объект можно создать «на лету». Соответствующий код может выглядеть следующим образом:

```
TOpenDialog* dlg = new TOpenDialog(this);
dlg->Title = "Open a New File";
dlg->Execute();
```

Обратите внимание, что объект должен создаваться с помощью оператора new. Если вы попытаетесь использовать локальное размещение, компилятор выдаст сообщение об ошибке VCL classes must be constructed using operator new (классы VCL должны создаваться с помощью оператора new).



Большинство компонентов VCL может быть создано во время выполнения программы с тем же успехом, что и на этапе разработки. Создавать компоненты во время разработке легче, потому что свойства гораздо проще изменять с помощью инспектора объектов, чем через код. Но тем не менее, остаются ситуации, когда необходимо создать объект во время выполнения программы, и C++Builder позволяет вам делать это.

Объектный Паскаль не поддерживает перегруженные функции; следовательно, классы VCL не имеют перегруженных конструкторов

Фактически, конструкторы в VCL выполняют очень ограниченное число действий. Возьмем, например, VCL-версию класса TRect. Чтобы создать и инициализировать объект типа TRect в VCL, вам потребуется код наподобие следующего:

```
TRect rect;
rect->left = 20;
rect->top = 20;
rect->right = 220;
rect->bottom = 220;
```

OWL также содержит класс TRect. Но в OWL вы можете инициализировать члены top, left, bottom и right через конструктор:

```
TRect rect(20, 20, 220, 220);
```

Очевидно, что задание параметров прямоугольника в конструкторе сокращает объем набираемого текста. В дополнение к этому, OWL-версия TRect содержит несколько конструкторов для создания экземпляров класса различными способами. Конструкторы класса CRect в MFC в основном аналогичны конструкторам TRect в OWL. Все сказанное означает, что в некоторых случаях при создании классов VCL в вашем распоряжении не будет той гибкости, которую дает Си++. Как правило, это не очень большая жертва.

Функции VCL не имеют аргументов по умолчанию

Для иллюстрации давайте рассмотрим функцию MessageBox() из API Windows. Эта функция принимает четыре аргумента: дескриптор окна, содержащего окно сообщения, текст сообщения, заголовок окна сообщения и параметр Flag, несущий информацию о том, какие кнопки и пиктограммы отображаются в окне сообщения. В MFC или OWL вы можете вывести сообщение, указав только его текст:

```
MessageBox("This is a message");
```

Это возможно благодаря наличию аргументов по умолчанию в OWL- и MFC-версиях функции MessageBox(). Вы можете указать дополнительные аргументы, но если этого не делать, будут использованы аргументы по умолчанию. Это удобно для вывода сообщений с минимальными затратами. А вот VCL-эквивалент предыдущей строки:

```
Application->MessageBox("This is a message", "Message", MB_OK);
```

Поскольку в VCL отсутствуют аргументы по умолчанию, вы должны указать все аргументы при вызове функции. Это менее удобно, но не смертельно, конечно. Заметьте, что во всех трех случаях (MFC, OWL и VCL) библиотека берет на себя передачу дескриптора окна.

Классы VCL не поддерживают множественное наследование

Это означает, что вы не можете создать новый компонент на основе двух уже существующих компонентов. Я не вижу здесь серьезного ограничения, поскольку множественное наследование используется не часто. Обычные классы Си++, которые вы пишете для приложений C++Builder, могут использовать множественное наследование.

Структура VCL

Библиотека визуальных компонентов тщательно продумана. Как и большинство хороших библиотек, VCL максимально использует возможности наследования. Основу VCL составляют классы, представляющие компоненты. Остальные классы VCL не имеют отношения к компонентам. Они выполняют различные вспомогательные и служебные задачи.

Иерархия классов VCL, относящихся к компонентам, достаточно сложна. К счастью, вам не нужно знать каждую деталь VCL, чтобы начать программировать в C++Builder. На вершине цепочки классов VCL находится TObject. На рис. 5.5 показаны некоторые из базовых классов и их производные классы.

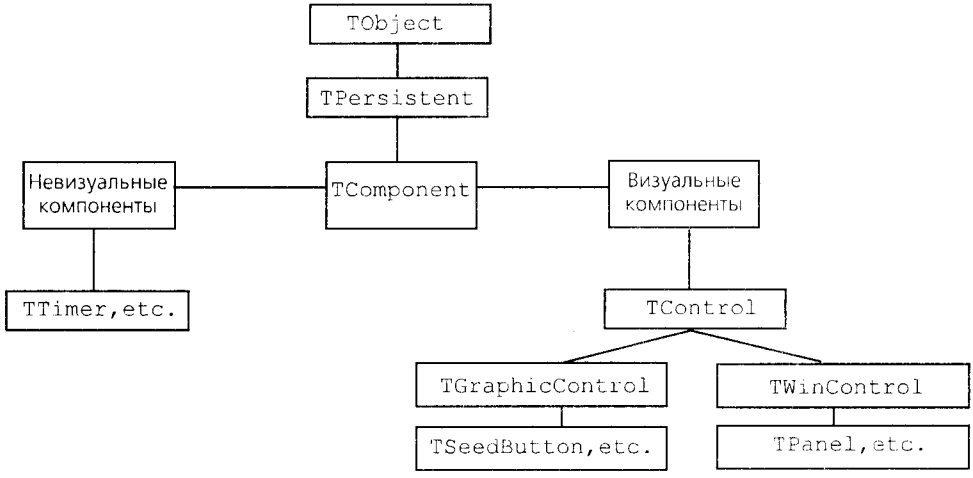


Рис. 5.5
Иерархия классов VCL

TObject является прародителем всех классов компонентов в VCL. Под ним вы видите класс TPersistent. Этот класс определяет способность компонентов сохранять себя в файлах и памяти, а также берет на себя заботу о других низкоуровневых деталях, которые нам знать не обязательно. Я просто счастлив (и вы, конечно, тоже), что для написания большинства приложений в C++Builder нам не нужно слишком много знать о TPersistent.

Класс TComponent является более прямым базовым классом для компонентов. Он обеспечивает функционирование всех основных компонентов. Невизуальные компоненты созданы непосредственно на основе TComponent. Основой же для визуальных компонентов служит TControl, который, как видно из рис. 5.5, произведен от TComponent. TControl обеспечивает дополнительные функции, которые требуются для визуальных компонентов. Отдельные компоненты произведены от TGraphicControl или TWinControl.

Когда вы помещаете компонент в форму, C++Builder генерирует код для создания нового класса на основе класса VCL, соответствующего данному компоненту. При этом создается указатель на объект, чтобы вы могли иметь к нему доступ. Для образования имен класса и указателя C++Builder использует значение свойства Name. Когда мы ранее создавали приложение-пример, в форму был помещен компонент Memo. В этот момент C++Builder

произвел новый класс от TMemo и создал экземпляр этого класса. Аналогично этому, при помещении в форму кнопки C++Builder произвел класс от TButton и создал его экземпляр. Для представления новой формы C++Builder ранее уже создал новый класс на основе класса TForm и, разумеется, экземпляр этого класса.

Очевидно, что для начала работы с VCL требуется немного узнать об ее классах. Я не могу привести здесь обзор всех классов VCL, поэтому давайте рассмотрим некоторые из них, которые вы будете использовать чаще всего.

Классы форм и приложений

Классы форм и приложений служат для представления в VCL форм и объектов Application. Все эти классы являются производными TComponent и, конечно же, сами являются компонентами. Они рассмотрены отдельно, чтобы отличать их от элементов управления, которые вы помещаете в форму.

TApplication

Класс TApplication инкапсулирует основные операции, выполняемые Windows-программой. Он заботится о таких вещах, как управление пиктограммой приложения, обеспечение контекстно-зависимой справки и осуществляет базовую обработку сообщений. Каждое приложение C++Builder имеет указатель на объект TApplication с именем Application. Вы будете использовать класс TApplication в основном для отображения окон сообщений, управления контекстно-зависимой справкой и вывода подписей к кнопкам и строкам состояния. Класс TApplication имеет ту особенность, что некоторые из его свойств (Icon, HelpFile и Title) могут быть установлены через вкладку Application диалогового окна Project Options.

TForm

Класс TForm служит в VCL для инкапсуляции форм. Формы используются для главных, диалоговых, подчиненных окон и для любых других типов окон, которые вы только сможете вообразить. TForm — это рабочая лошадка в VCL. Он имеет около 60 свойств, 45 методов и 20 событий, на которые возможна реакция. Я собираюсь рассказывать о формах завтра, поэтому не буду сейчас вдаваться в детали.

Классы компонентов

В эту группу входит большое количество классов, которые можно распределить по нескольким отдельным подгруппам.

Классы стандартных компонентов

К стандартным компонентам относятся те, которые инкапсулируют наиболее распространенные элементы управления Windows. Классами стандартных компонентов являются TButton, TEdit, TListBox, TMemo, TMainMenu, TPopupMenu, TCheckBox, TRadioButton, TRadioGroup, TGroupBox и TPanel.

Большинство этих классов соответствует элементам управления Windows, поэтому я не буду сейчас обсуждать каждый из них, а остановлюсь лишь на некоторых. Класс TMainMenu инкапсулирует главное меню приложения. Во время разработки формы двойной щелчок на пиктограмме компонента

MainMenu вызывает редактор меню (Menu Editor). Свойства TMainMenu управляют выделением или выключением данного пункта меню, определяют его текст, идентификатор контекстной справки и т.д. Каждый пункт меню реагирует на единственное событие — OnClick, и вы можете связать с ним функцию, вызываемую при выборе данного пункта. Я расскажу о меню и его редакторе более подробно на 7-й день, «Работа с редактором форм и редактором меню».

Другой стандартный компонент, представляющий интерес — TPanel.



Панель (panel) — это прямоугольный участок формы, который обычно содержит другие компоненты и рассматривается как единый компонент.

Например, если вы хотите создать оперативную панель, нужно начать с панели, а затем поместить на нее кнопки. Если вы переместите панель, кнопки переместятся вместе с ней. Панели могут быть использованы в самых разных ситуациях. В частности, можно использовать панель для строки состояния. Свойства панелей определяют тип края панели; будет ли панель приподнятой, утопленной или плоской; ширину рамки. Эти свойства можно использовать в различных сочетаниях для создания трехмерных панелей.

В C++Builder есть еще одна группа компонентов, которые я отнес к стандартным элементам управления. Они собраны во вкладке Additional палитры компонентов. К классам, представляющим их, относятся TBitBtn, TSpeedButton, TMaskEdit, TStringGrid, TDrawGrid, TImage, TShape, TBevel и TScrollBox. Класс TBitBtn представляет кнопку с изображением. TSpeedButton является не настоящей кнопкой, а ее графическим изображением. Это дает возможность иметь большое количество кнопок быстрого доступа, не расходуя ресурсы Windows для каждой из них. Компонент TImage позволяет вам поместить в форму изображение, которое находится в дисковом файле. Компонент TBevel можно использовать для окон и линий, которые приподняты или утоплены относительно поверхности формы. Эти рельефные элементы можно использовать для визуального разделения формы на отдельные области и улучшения ее внешнего вида.

Классы специальных элементов управления Windows 95

VCL содержит классы, инкапсулирующие многие из специальных элементов управления Win32. Это классы TListView, TTreeView, TProgressBar, TTabControl, TPageControl, TRichEdit, TImageList, TStatusBar и некоторые другие. Некоторые из этих элементов управления сами по себе сложны, и соответствующие классы также имеют сложную структуру. Поверьте мне, что VCL делает великое дело, облегчая работу с такими элементами. Для полного понимания этих классов вам придется поработать с ними некоторое время.

Классы компонентов доступа к базам данных

VCL содержит много компонентов, относящихся к базам данных, среди которых есть как визуальные, так и невизуальные. Невизуальные компоненты представлены классами TDataSource, TDatabase, TTable и TQuery. Эти классы инкапсулируют фоновые операции с базами данных.

Визуальные компоненты доступа к базам данных предназначены для организации взаимодействия с пользователем. Например, компонент TDBGrid используется для обеспечения доступа к таблице базы данных, которая может

быть представлена компонентом TTable. Таким образом, TDBGrid выступает в роли интерфейса между пользователем и TTable. Через TDBGrid пользователь может просматривать и редактировать таблицу базы данных на диске.

Компонент TDBNavigator содержит кнопки, позволяющие пользователю перемещаться по таблице базы данных. Это кнопки перехода на следующую, предыдущую, первую или последнюю записи, отмены редактирования, подтверждения и отмены изменений.

Остальные классы интерфейсных компонентов используют стандартные элементы управления Windows для работы с полями базы данных. В частности, это классы TDBText, TDBEdit, TDBListBox и TDBImage.

Классы стандартных диалогов

Как вы, несомненно, знаете, в Windows существуют стандартные диалоговые окна для таких действий, как открытие и сохранение файлов, выбор шрифтов и цвета. VCL содержит специальные классы для представления этих диалоговых окон. К ним относятся TOpenDialog, TSaveDialog, TFontDialog, TColorDialog, TPrintDialog и TPrintSetupDialog. Дополнительно к этой группе относятся классы TFindDialog и TReplaceDialog. Перечисленные компоненты не являются визуальными в том смысле, что они не отображаются на этапе разработки. Во время выполнения программы диалоговые окна, разумеется, видимы.

Классы системных компонентов

Вкладка System палитры компонентов содержит смесь визуальных и невизуальных компонентов. Класс TTimer используется для представления системного таймера Windows. Единственное событие, на которое он реагирует — это OnTimer, которое соответствует каждому отсчету таймера. Интервал работы таймера устанавливается через свойство Interval. TTimer является невизуальным компонентом.

VCL содержит также несколько классов, позволяющих вам создавать собственные диалоговые окна File Open или File Save. Это классы TFileListBox, TDirectoryListBox, TDriveComboBox и TFilterComboBox.

В данную группу классов входит также класс TMediaPlayer, который позволяет воспроизводить аудиозаписи в стандартах WAV и MIDI, а также видео в стандарте AVI. При этом возможны собственно воспроизведение, остановка, пауза, переход в произвольное место записи и многие другие операции. Этот класс имеет много свойств и событий, которые сильно упрощают общение со сложным миром Windows MCI (Media Control Interface).

В группу System входят также классы для поддержки OLE и динамического обмена данными (DDE — Dynamic Data Exchange).

Классы GDI

Классы GDI (Graphics Device Interface) обычно выполняют большую работу в GUI-приложениях Windows. Эти классы инкапсулируют использование растровых изображений, шрифтов, контекстов устройств (DC), кистей и перьев. Именно через объекты GDI происходит отображение графики и текста в окнах. Классы GDI не связаны с определенным компонентом, но многие компоненты содержат экземпляры этих классов в качестве свойств. Например, элемент управления редактированием имеет свойство Font, являющееся экземпляром класса TFont.

Термин *контекст устройства (device context — DC)* хорошо известен всем, кто программирует в Windows как на Си, так и с помощью библиотек Си++. Однако в VCL этот термин широко не используется. DC здесь называется *холстом (canvas)*, а вся сложная структура DC инкапсулирована в классе TCanvas. Холст предоставляет вам поверхность, на которой можно рисовать с помощью методов типа MoveTo(), LineTo() и TextOut(). Растровые изображения выводятся на холст методами Draw() и StretchDraw(). Концепция холста, который вы разрисовываете, более наглядна, чем архаичный термин «контекст устройства», не так ли?

Класс TCanvas содержит экземпляры других классов GDI. Например, когда вы используете последовательность MoveTo()/LineTo(), линия рисуется пером текущего цвета. Текущий цвет пера определяется свойством Pen, которое представляет собой экземпляр класса TPen. Свойства TPen определяют тип линии: ее ширину, стиль (сплошная, пунктирная и т.д.) и режим рисования.

Класс TBrush представляет кисть, которая используется как образец заливки при операциях с холстом типа FillRect(), Polygon() и Ellipse(). К свойствам TBrush относятся Color, Style и Bitmap. Свойство Bitmap позволяет вам указать растровое изображение, которое будет использоваться в качестве образца заливки.

TBitmap инкапсулирует операции с растровыми изображениями в VCL. В число его свойств входят Palette, Height, Width и TransparentColor. Методы включают в себя LoadFromFile(), LoadFromResourceID() и SaveToFile(). TBitmap используется не только TCanvas, но и другими классами, такими, как TImage, TBitBtn и TSpeedButton.

Класс TFont поддерживает операции со шрифтами. К его свойствам относятся Color, Height и Style (полужирный, курсивный, нормальный и т.д.). Класс TFont используется всеми классами компонентов, отображающими текст.

Кроме перечисленных здесь классов GDI, существуют и другие, которые выполняют вспомогательные функции или расширяют базовый класс для обеспечения дополнительных возможностей. Работая с C++Builder, вы постепенно узнаете об этих классах и их использовании. На рис. 5.6 показана иерархия классов VCL, которые инкапсулируют GDI-операции.

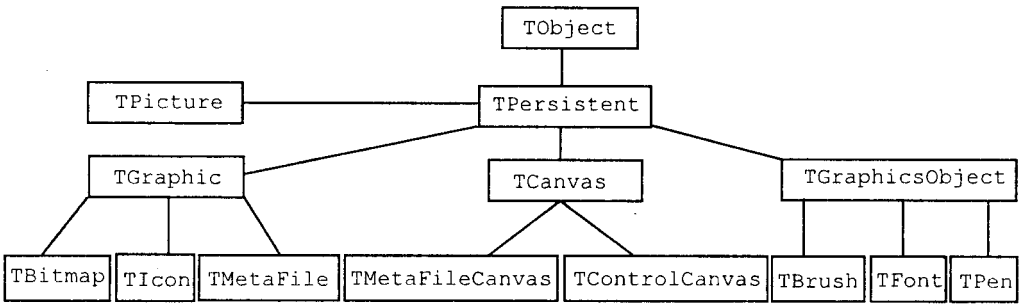


Рис. 5.6
Иерархия GDI-классов VCL

Служебные классы

Только что я рассказал о классах компонентов. VCL содержит также служебные классы, которые вы можете использовать в своих приложениях. Служебные классы упрощают некоторые задачи, возникающие при программировании в Windows. Например, класс TIniFile помогает осуществлять чтение и запись файлов конфигурации Windows (.INI-файлов). Общепризнано, что следует избегать использования файлов .INI, а вместо этого работать с реестром (Registry). Для операций с реестром VCL содержит классы TRegistry и TRegkeyInfo.

Классы TRect и TPoint (которые в действительности являются всего лишь структурами) предназначены для работы с различными функциями VCL и Windows, требующими в качестве аргументов точку или прямоугольник.

Класс TStrings используется для работы со строками, а класс TStringList поддерживает массивы строк. Эти классы используются многими компонентами для хранения строк. Например, класс TMemo использует объект TStringList в качестве свойства Lines. TStringList способен сохранять строки в файле или загружать их из файла, используя методы LoadFromFile() и SaveToFile().

Но это еще не все...

Конечно, я не мог рассказать здесь о всех классах VCL. Но я затронул те из них, которые вы наверняка будете использовать в своих приложениях.

Вернитесь на несколько страниц назад и взгляните еще раз на листинг 5.1 и его OWL-эквивалент — листинг 5.2. Если вы помните, я говорил тогда, что размещение растрового изображения в окне выполняется в C++Builder еще легче. Позвольте мне показать, что я имел в виду. Для начала создайте новое приложение C++Builder. Перед вами должна быть пустая форма. Затем выполните следующую последовательность действий:

1. Измените значение свойства Caption формы на Bitmap Test Program.
2. Щелкните на вкладке Addtional палитры компонентов, выберите компонент Image и поместите его в форму.
3. Найдите свойство Align и измените его значение на alClient. Изображение заполнит всю рабочую область формы.
4. Найдите свойство Stretch и измените его значение на true.
5. Найдите свойство Picture и дважды щелкните на столбце значений. Откроется диалоговое окно редактора изображений (Picture Editor).
6. Щелкните на кнопке Load. Откроется диалоговое окно File Open.
7. Перейдите в каталог \CBuilder\Images\Splash\256color и выберите картинку, которая вам понравится (лично мне нравится HANDSHAKE.BMP). Щелкните на кнопке OK.
8. Вы оказались снова в диалоговом окне редактора изображений, а выбранное изображение выведено в окно предварительного просмотра. Щелкните на кнопке OK. (Если вы хотите выбрать другую картинку, щелкните снова на кнопке Load.) Теперь изображение занимает всю рабочую область формы.
9. Щелкните на кнопке Run. После запуска приложения вы можете изменять размер окна, и при этом изображение все время будет целиком заполнять его рабочую область.

Видите, как легко? Все было бы еще проще, если бы мы не заботились о заполнении всей рабочей области формы. На рис. 5.7 показано окно нашей тестовой программы после запуска.



Рис. 5.7
Тестовая программа запущена

Заключение

Сегодняшний день был посвящен библиотекам классов. Сначала вы узнали об OWL и MFC, а также о той роли, которую они сыграли в формировании современного подхода к программированию в Windows. Затем была рассмотрена VCL и ее отличия от библиотек Си++. Мы обсудили свойства, методы и события, и по ходу дела вы получили некоторый практический опыт. Мы закончили сегодня обзором классов VCL, с которыми вы наверняка встретитесь при программировании в C++Builder. Я не охватил их все и кратко остановился лишь на наиболее часто используемых.

Так куда же развивается индустрия программного обеспечения? По-видимому, будущее принадлежит компонентам, хотя потребность в библиотеках типа MFC и OWL еще сохранится достаточно долго. Некоторые из вас, кто использует сейчас MFC или OWL, предпочтут им среды типа C++Builder и Delphi. Другие будут использовать как старый, так и новый инструментарий. Кто-то остановится на том, что он лучше знает. В любом случае важно понимать, что каждая из библиотек представляет собой инструмент. Мой совет прост: используйте тот из них, который наиболее подходит для текущей работы.

Если вы никогда не использовали OWL или MFC, не беспокойтесь о том, что вы что-то упустили. C++Builder и VCL позволяют вам создавать полноценные приложения гораздо быстрее, чем любая из библиотек Си++, особенно если принять во внимание то время, которое нужно затратить на изучение VCL по сравнению с MFC или OWL. Программирование в C++Builder освоить намного легче, и при этом вы можете писать программы быстрее.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Что такое библиотека классов?

Библиотека классов представляет собой набор классов, упрощающих программирование в Windows. Хорошая библиотека реализует объектно-ориентированный подход к написанию Windows-приложений.

Является ли VCL библиотекой Си++?

Нет. VCL — это библиотека, которая работает с Си++ в C++Builder, но сама она написана на Объектном Паскале, а не на Си++. Причина в том, что изначально VCL предназначалась для Borland Delphi.

Должен ли я знать и Паскаль, и Си++ для написания Windows-программ с помощью C++Builder?

Нет. Тот факт, что VCL написана на Паскале, не имеет к вам прямого отношения. Пока вы не заходите слишком далеко, программирование полностью аналогично Си++. Более опытные пользователи Си++ могут заметить, что в некоторых ситуациях VCL ограничивает их возможности, но для большинства это не имеет значения.

Возникает ощущение, что подход, реализованный в компонентах, является наилучшим. Правда ли это?

Это действительно так для многих приложений, но, разумеется, не для всех. В некоторых случаях библиотеки классов типа OWL или MFC лучше подходят для решаемой задачи. VCL, вероятно, будет очень хорошим выбором для приложений, в которых используется большое число окон, или для работы с базами данных. Кроме того, C++Builder намного легче изучать и использовать, чем библиотеки классов Си++.

Являются ли свойства просто членами-данными класса?

Нет. Свойства — это особые объекты. Некоторые свойства представляют собой просто набор членов-данных класса. Другие свойства при изменении вызывают метод, выполняющий специальные операции с данным свойством. В этих случаях свойство уже не является просто набором членов-данных.

Должен ли я реагировать на все события, определенные для компонента?

Нет. Вы можете обрабатывать только те события, которые нужны вашему приложению, или вообще не реагировать на события.

Существует огромное количество классов VCL. А я думал, что программирование в C++Builder будет простым.

Программирование с помощью C++Builder намного проще, чем программирование в Windows на Си, и проще, чем программирование с использованием библиотек Си++, подобных MFC или OWL. Программирование в Windows, независимо от используемого инструмента, требует большого опыта и знаний. Спустя некоторое время вы овладеете им, если не бросите дело на полдороги.

Можно ли использовать формы C++Builder в OWL- и MFC-программах?

Да. Позже я покажу, как это делается.

Вопросы

1. Все ли компоненты видимы во время разработки и выполнения программы?
2. Что лучше — OWL, MFC или VCL?
3. Возможно ли для объектов VCL локальное размещение (в стеке) наряду с динамическим?
4. Эквивалентны ли методы компонентов VCL функциям Си++?
5. Верно ли, что все классы VCL произведены в конечном итоге от TObject?
6. Назовите один невизуальный компонент VCL.
7. Существуют ли свойства, общие для всех компонентов?
8. Назовите два свойства, общих для всех визуальных компонентов.
9. Возможно ли использование одного обработчика событий двумя или более компонентами?
10. Какой термин используется в VCL для контекста устройства Windows? Как называется класс VCL, инкапсулирующий контекст устройства?

Упражнения

1. Выпишите различия между свойствами и членами-данными класса.
2. Создайте приложение C++Builder, отображающее растровую картинку в главной форме при нажатии на кнопку.
3. Создайте приложение C++Builder, отображающее окно сообщения с текстом Hello, Bubba! при щелчке на главной форме.
4. Создайте приложение C++Builder, отображающее красными буквами текст I've been resized! при изменении размера окна этого приложения.
5. **Дополнительное задание:** измените программу из упражнения 4 так, чтобы текст исчезал спустя 5 секунд.

Изучение IDE C++Builder: проекты и формы



Кент Рейсдорф

Одной из наиболее трудных задач при освоении новой программы является изучение ее оболочки: базовой структуры меню, набора опций и общих принципов работы. Если вы новичок в Си++ или вообще в программировании, дело осложняется тем, что одновременно с изучением самой программы (IDE C++Builder) вам нужно учить новый язык. Временами это может показаться непреодолимым. Я приложу все усилия для того, чтобы сделать изучение IDE C++Builder по возможности безболезненным. По большей части мы будем рассматривать конкретные примеры, что гораздо интереснее (не говоря уже о большей эффективности). Итак, без лишних слов перейдем к делу, начав с рис. 6.1.

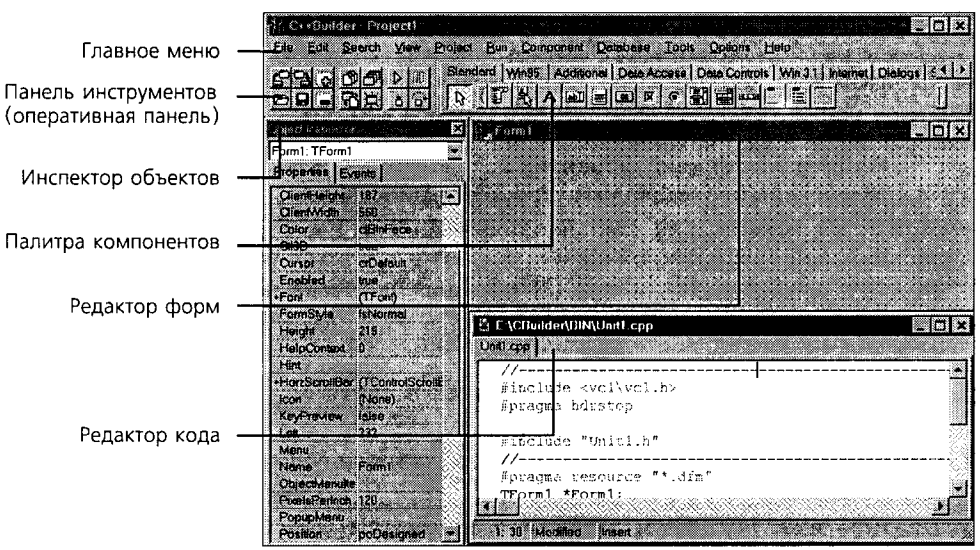


Рис. 6.1
Окна IDE C++Builder

Интегрированная среда разработки C++Builder состоит из следующих основных частей:

- Главного меню и панели инструментов (main menu and toolbar)
- Палитры компонентов (Component Palette)

- Редактора форм (Form Designer)
- Редактора кода (Code Editor)
- Инспектора объектов (Object Inspector)
- Менеджера проектов (Project Manager)

Я не смогу рассмотреть каждую из этих составляющих в одной главе, поэтому в следующих нескольких главах я продолжу обзор IDE C++Builder и подробно расскажу о всех ее возможностях. Сегодня я начну с обсуждения проектов и их использования при написании приложений C++Builder. После этого мы рассмотрим панель инструментов и палитру компонентов. Затем я перейду к более подробному, чем раньше, рассмотрению форм. По ходу дела мы создадим несколько простых программ, иллюстрирующих различные возможности C++Builder. В завершение этого дня мы обратимся к инспектору объектов. Это станет заделом на завтра, когда вы будете изучать редактор форм.

Для начала давайте посмотрим, как C++Builder рассматривает приложения и как это упрощает процесс создания программ.

Проекты в C++Builder

Как вы теперь знаете, при написании приложений в C++Builder многое происходит «за кулисами». На самом деле там происходит еще больше, чем я говорил до сих пор. Вам совсем не обязательно знать все это в деталях, но неплохо иметь хотя бы общее представление.



Проект (project) — это набор файлов, которые используются при создании автономного исполняемого файла или динамически компонуемой библиотеки (DLL).

Файлы, используемые в проектах C++Builder

C++Builder управляет проектами с помощью нескольких вспомогательных файлов. Давайте создадим простое приложение и на его примере посмотрим, что происходит при построении исполняемого файла для вашей программы. Сделайте следующее:

1. Прежде всего, создайте пустой каталог на жестком диске.
2. Теперь выберите в главном меню пункт File | New Application. На экране будет присутствовать пустая форма.
3. Перед тем, как продолжить, выберите в главном меню пункт File | Save Project.
4. Сначала у вас будет запрошено имя исходного файла. Перейдите в только что созданный пустой каталог.
5. Введите имя MyUnit и щелкните на кнопке OK.
6. Теперь у вас будет запрошено имя проекта. Наберите TEST в поле File name и щелкните на кнопке OK.
7. Выберите в главном меню пункт Project | Build All. C++Builder выведет окно состояния компилятора начнет компиляцию вашей программы.
8. Спустя некоторое время в окне состояния появится сообщение о завершении компиляции и кнопка OK станет функционировать. Щелкните на ней, чтобы закрыть окно.

9. Теперь выберите в главном меню пункт Project | Close All. (Да, в этом упражнении нет больше никакой другой цели.)
10. Теперь запустите Windows Explorer (проводник Windows) и перейдите в каталог, где был сохранен проект. Вы найдете там несколько файлов.

Уф! И все это для того, чтобы создать крошечную программу, которая ничего не делает? Да, это так. Сначала позвольте мне рассказать, что происходит, когда C++Builder строит приложение; затем я объясню, для чего нужен каждый из этих файлов.



Файлы с расширением, начинающимся с тильды (~), являются резервными копиями. C++Builder может создать несколько резервных файлов, в зависимости от количества исходных файлов в проекте и установленных опций проекта. Опции проекта рассматриваются на десятый день, «Дополнительные сведения о проектах».

Для нового проекта C++Builder создает как минимум шесть файлов (имеется в виду типичное GUI-приложение C++Builder):

- ☐ Исходный файл проекта
- ☐ Исходный файл главной формы
- ☐ Заголовочный файл главной формы
- ☐ Файл ресурсов главной формы
- ☐ Файл ресурсов приложения
- ☐ Информационный файл проекта

Исходный файл проекта (project source file) содержит функцию WinMain() и прочий стартовый код C++Builder. Вы можете просмотреть этот файл, выбрав в главном меню пункт View | Project Source. *Исходный файл главной формы (main form source file)* и *заголовочный файл главной формы (main form header file)* содержат объявления и определения классов главной формы. C++Builder будет создавать дополнительные исходные и заголовочные файлы для каждой новой формы данного проекта. *Файл ресурсов главной формы (main form resource file)* и *файл ресурсов приложения (application resource file)* представляют собой двоичные файлы, описывающие главную форму и значок приложения. Я объясню это подробнее немного позже, в разделе «Диалоговые окна в традиционной Windows-программе».

C++Builder создает также *информационный файл проекта (project makefile)*. Он представляет собой текстовый файл, содержащий установленные вами опции компилятора, имена исходных файлов и форм, входящих в проект, а также имена необходимых библиотечных файлов.



Существуют два типа библиотечных файлов. *Статическая библиотека (static library)* содержит код общего назначения, необходимый для запуска приложения. *Библиотека импорта (import library)* требуется, если приложение обращается к функциям DLL, например, функциям API Windows. Количество и точные имена необходимых библиотечных файлов определяются приложением. К счастью, вам не нужно заботиться об управлении библиотечными файлами, поскольку C++Builder берет это на себя. Библиотечные файлы имеют расширение .LIB и спрятаны в подкаталоге \lib каталога C++Builder.

Когда вы даете C++Builder указание начать построение приложения, информационный файл передается компилятору. (На самом деле этот файл считывается утилитой Make, но зачем углубляться в детали?) Компилятор читает информационный файл и начинает компиляцию всех исходных файлов, составляющих проект.

Этот процесс разбивается на несколько этапов. Сначала компилятор Си++ преобразует исходные файлы в двоичные объектные файлы. Затем компилятор ресурсов собирает все ресурсы, такие как значки и файлы форм, в двоичный файл ресурсов. После этого в работу вступает компоновщик. Он берет двоичные файлы, созданные компиляторами, добавляет необходимые библиотечные файлы и связывает их вместе для образования исполняемого файла. По ходу дела компоновщик создает еще несколько специальных файлов (я расскажу о них чуть позже). Когда все это завершается, вы получаете автономную программу, которая может быть запущена обычным образом.

Хорошо, но для чего предназначены все файлы, о которых я упоминал? В таблице 6.1 перечислены расширения имен файлов, используемых C++Builder, и даны краткие описания файлов соответствующих типов.

Таблица 6.1. Типы файлов, используемых в C++Builder

Расширение	Описание
.CPP	Исходные файлы Си++. Обычно один файл для каждого модуля и один главный файл проекта. К проекту можно добавить также любые другие исходные файлы.
.DFM	Файл формы (form file). В действительности это другой вид двоичного файла ресурсов (.RES). Он содержит описание формы и всех ее компонентов. Каждой форме соответствует свой .DFM-файл.
.DSK	Файл рабочей конфигурации (desktop file). Этот файл отслеживает расположение компонентов IDE на момент последнего сохранения (или закрытия) проекта. В нем сохраняются размер и положение всех открытых окон, поэтому при открытии проекта IDE выглядит точно так же, как и при выходе из него.
.EXE	Исполняемый файл программы.
.H	Заголовочный файл Си++, содержащий объявления классов. Это может быть стандартный файл C++Builder или ваш собственный файл.
.IL?	Четыре файла с расширениями, начинающимися на IL, создаются пошаговым компоновщиком. Пошаговый компоновщик (incremental linker) экономит ваше время, обрабатывая только ту часть программы, которая была изменена с момента ее последнего построения.
.OBJ	Двоичные объектные файлы компилятора. Они создаются в результате компиляции исходных файлов Си++.
.MAK	Информационный файл проекта. Это текстовый файл, содержащий список файлов, необходимых C++Builder для компиляции и компоновки.
.RES	Двоичный файл ресурсов, создаваемый компилятором ресурсов.
.TDW	Символьная таблица отладчика. Этот файл используется отладчиком во время сеансов отладки.



Приложения C++Builder, предназначенные для работы с базами данных, используют также другие типы файлов. Эти приложения будут подробно рассмотрены в последующих главах.

Файлы, создаваемые C++Builder, можно разделить на две категории: файлы, на основе которых строится проект, и файлы, создаваемые при компиляции и компоновке проекта. Если вы, например, решите перенести все ваши исходные файлы на другой компьютер, вам нужно будет переписать только те из них, которые C++Builder использует для построения приложения. Обычно исходные файлы имеют наименьший размер по сравнению с другими файлами проекта и не требуют много места при резервном копировании.

Файлы с расширениями .CPP, .H, .DFM и .MAK составляют минимальный набор. Все остальные файлы C++Builder создаст заново при построении программы. Вы можете сохранить также файл рабочей конфигурации (.DSK), который отражает состояние IDE на тот момент, когда вы последний раз работали над проектом.



Дополнительно к тем исходным файлам, о которых я упомянул, некоторые приложения используют файл *сценария ресурсов (resource script file)*. Сценарии ресурсов имеют расширение .RC и представляют собой текстовые файлы, используемые для определения ресурсов типа растровых изображений, значков или курсоров. Если вы используете сценарий ресурсов, не забывайте о нем при перемещении проекта в другое место.

На рис. 6.2 схематически показано, как C++Builder формирует исполняемый файл из исходных файлов.

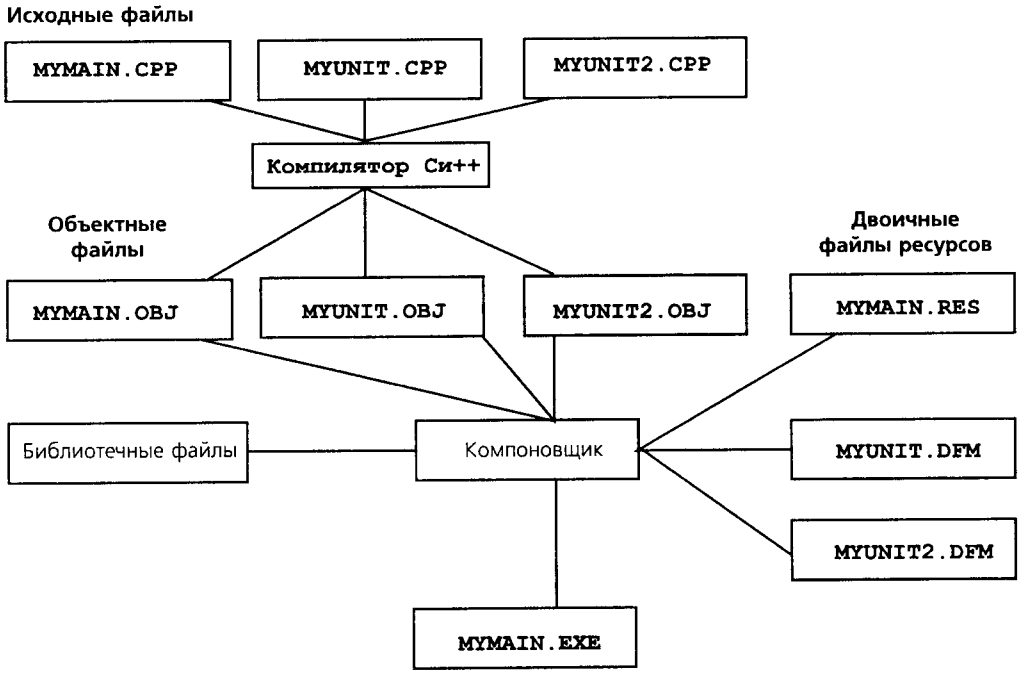


Рис. 6.2
Процесс компиляции/компоновки



Если вы столкнулись с нехваткой места на жестком диске, можно удалить некоторые файлы тех проектов, над которыми вы не работаете. Можно без опасений удалять файлы с расширениями .OBJ, .RES и .TDW, а также файлы с расширениями, начинающимися с .IL. Некоторые из этих файлов вырастают до больших размеров, и нет никакой необходимости сохранять их для проектов, не использующихся в данный момент.



Не удаляйте файлы в каталогах C++Builder, за исключением каталога Examples. Если сомневаетесь, не трогайте!

Модули исходного кода

Ранее я говорил, что большинство приложений, независимо от их размера, состоит из нескольких исходных файлов, которые Borland называет *units*. Термин *unit* перешел в C++Builder из Паскаля и Delphi. C++Builder уходит корнями в IDE Delphi, поэтому *unit* повсеместно используется в VCL и самой интегрированной среде C++Builder. Программисты на Си++ обычно называют файл, содержащий исходный текст, *module*. Хотя использование термина *module* более соответствует Си++ (и менее — Паскалю), замена слова *unit* на *module* потребовала бы существенных изменений в инфрастуктуре C++Builder, поэтому *unit* был оставлен. Если вы уже имели опыт программирования на Си++, употребление *unit* вместо *module* может показаться необычным, но вы привыкнете к этому достаточно быстро. (На русский язык оба термина переводятся как *модуль*. — *Прим. ред.*)



В C++Builder для обозначения исходного файла используется термин *unit*.

Каждый раз при создании новой формы C++Builder делает следующее:

- Создает файл формы (.DFM)
- Создает новый класс на основе TForm или другого класса формы
- Создает заголовочный файл (.H), содержащий объявление класса
- Создает модуль (.CPP), содержащий определение класса
- Добавляет информацию о новой форме в информационный файл проекта

Изначально C++Builder присваивает новой форме имя Form1, соответствующему модулю — имя Unit1.cpp, а заголовочному файлу — имя Unit1.h. Следующая форма получит по умолчанию имя Form2, и т.д.



Сразу после создания нового проекта сохраните его под информативным именем. То же самое следует делать после создания каждой новой формы. Это позволит легче находить формы и модули, когда потребуется внести в них изменения.



Автору технической книги часто приходится попадать в сложные ситуации. Я хочу использовать содержательные примеры, чтобы улучшить подачу материала. Чтобы эти примеры выглядели завершенными, мне приходится использовать приемы и методы, о которых я еще не рассказывал. Но я не могу рассказать о них, пока не приведу хороший, содержательный пример. Но я не могу... Короче, вы понимаете мое положение. Итак, я собираюсь немного отклониться от темы и рассказать о главном меню, панели инструментов и палитре компонентов. При чтении этого раздела не забывайте, что мы сошли с прямой дороги.

Главное меню и панель инструментов C++Builder

Главное меню содержит все пункты, необходимые для работы с C++Builder. Поскольку программирование в C++Builder является в значительной степени визуальным, главное меню можно использовать с меньшей интенсивностью по сравнению с другими средами программирования. Но если вы предпочитаете работать с меню, оно содержит почти все, что может когда-либо потребоваться. Я не собираюсь рассматривать здесь все пункты главного меню, поскольку на протяжении нескольких следующих глав вы постепенно встретитесь с каждым из них.

Панель инструментов C++Builder предоставляет удобный способ выполнения часто повторяющихся действий. Кнопку легче отыскать, чем пункт главного меню, не говоря уже о том, что для этого нужно гораздо меньше работать мышью. Конфигурация панели инструментов C++Builder, устанавливаемая по умолчанию, показана на рис. 6.3.

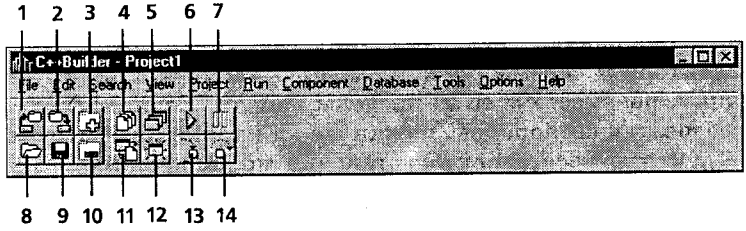


Рис. 6.3
Панель инструментов C++Builder:

1 — открыть проект; **2** — сохранить все; **3** — добавить к проекту; **4** — выбрать модуль; **5** — выбрать форму; **6** — запустить; **7** — приостановить; **8** — открыть файл; **9** — сохранить файл; **10** — удалить из проекта; **11** — отобразить модуль/форму; **12** — новая форма; **13** — трассировка; **14** — пошаговое выполнение

Если вы похожи на меня, то часто будете забывать использовать панель инструментов. Но я настойчиво повторяю: научитесь с ней работать. Как говорят старики: «Делай то, что я говорю, а не то, что я делаю». Это сбережет ваше время и повысит эффективность работы. Одной из причин, заставивших вас приобрести C++Builder, была возможность быстрой разработки приложений для Windows, но реализация этой возможности зависит от вас.

Панель инструментов C++Builder может быть полностью перенастроена. Как видно из рис. 6.1, между панелью инструментов и палитрой компонентов находится вертикальная разделительная линия, которая служит для изменения размеров панели. Когда курсор мыши оказывается на этой линии, он принимает форму двухсторонней черной стрелки. После этого вы можете сдвинуть разделительную линию вправо или влево, чтобы отвести панели инструментов больше или меньше места в главном окне C++Builder.

Настройка панели инструментов крайне проста. C++Builder позволяет вам добавлять, удалять и переставлять кнопки произвольным образом. Настройка панели осуществляется через контекстное меню, для вызова которого нужно поместить курсор мыши на панель и нажать правую кнопку. Пункты меню перечислены в таблице 6.2.

Таблица 6.2. Пункты контекстного меню панели инструментов

Пункт меню	Описание
Show Hints (Отображать подсказки)	Управляет отображением всплывающих подсказок для кнопок панели инструментов.
Hide (Скрыть)	Убирает панель инструментов.
Help (Справка)	Открывает справку по C++Builder на странице, посвященной панели инструментов.
Properties (Свойства)	Вызывает диалоговое окно редактора панели инструментов (Toolbar Editor) для настройки панели.



Если вы убрали панель инструментов, для ее отображения нужно выбрать в главном меню пункт View | Toolbar.

Для настройки панели инструментов выберите пункт Properties в контекстном меню. После этого на экране появится диалоговое окно редактора панели инструментов. Диалоговое окно содержит все кнопки, которые могут быть размещены на панели. Для добавления кнопки к панели просто выберите ее в окне редактора и переместите с помощью мыши в нужное место. Чтобы удалить кнопку, переместите ее за пределы панели. Если в результате ваших действий образовался беспорядок, щелкните на кнопке Reset в окне редактора (см. рис. 6.4), чтобы восстановить установки по умолчанию.

Если вы хотите выделить для кнопок больше места, передвиньте разделительную линию вправо, чтобы сделать панель инструментов шире. После этого останется только переместить на освободившееся место любые кнопки из окна редактора панели. Панель инструментов имеет невидимую сетку, служащую для выравнивания новых кнопок. Достаточно поместить кнопку рядом с тем местом, которое она должна занять, чтобы произошла автоматическая привязка к сетке. Мне нравится иметь на панели инструментов кнопки Cut, Copy и Paste, поэтому я настраиваю свою панель так, чтобы она содержала эти кнопки. На рис. 6.4 показан процесс перемещения кнопки Paste на панель инструментов (кнопки Cut, Copy и Undo уже находятся на своих местах).

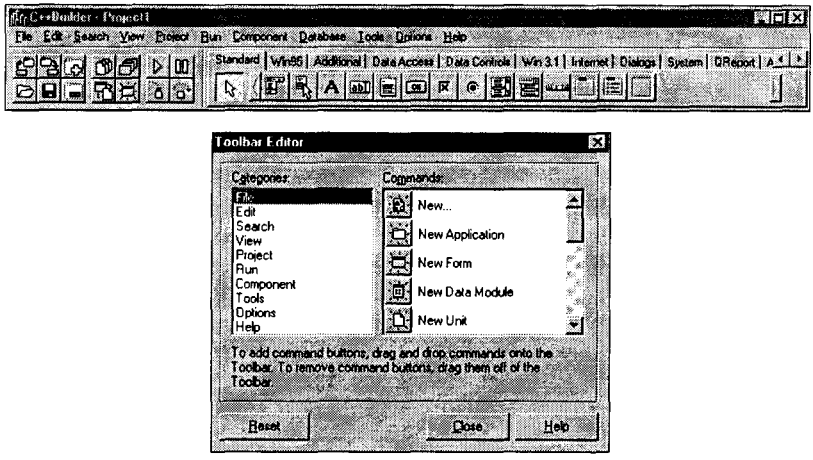


Рис. 6.4 Настройка панели инструментов

Настраивайте IDE C++Builder так, как вам нравится. Это ваша среда разработки, поэтому сделайте ее максимально удобной для себя.

Использование палитры компонентов

Палитра компонентов C++Builder предназначена для выбора компонентов или других элементов управления (таких, как ActiveX), размещаемых в форме. Палитра компонентов представляет собой многостраничное окно. Для отображения компонентов, расположенных на какой-либо вкладке, нужно щелкнуть на ее заголовке.

Размещение компонента в форме представляет собой двухступенчатый процесс. Сначала выберите в палитре компонентов кнопку, соответствующую нужному компоненту. Затем щелкните на форме. Компонент появится в форме, причем его верхний левый угол будет там, где находился курсор мыши в момент щелчка.

В основном вы уже знаете, как работать с палитрой компонентов, но она предоставляет еще некоторые возможности, которые демонстрируются в следующих разделах.

Размещение нескольких копий компонента

До сих пор вы помещали в форму только по одному компоненту за раз. Однако можно легко сделать несколько копий компонента, не выбирая его каждый раз в палитре. Для этого при выборе компонента нажмите и удерживайте клавишу Shift. Затем отпустите клавишу. Кнопка компонента останется нажатой и будет при этом выделена синей рамкой. Щелкните на форме для размещения первого компонента. Заметьте, что кнопка в палитре компонентов по-прежнему нажата. При каждом следующем щелчке в форму будет помещаться новый компонент. Для остановки этого процесса щелкните на кнопке указателя в палитре компонентов (кнопка со стрелкой). Кнопка компонента вернется в прежнее положение, показывая тем самым, что вы вышли из режима копирования.

Увидеть — значит поверить, поэтому сделайте следующее:

1. Создайте новый проект.
2. Нажмите и удерживайте клавишу Shift на клавиатуре, затем щелкните на кнопке компонента Label в палитре компонентов.
3. Щелкните три раза на форме, перемещая каждый раз курсор мыши на новое место.
4. Щелкните на кнопке со стрелкой в палитре компонентов, чтобы остановить копирование и вернуться в режим разработки формы.



Использование описанной техники позволяет быстро поместить все компоненты одного типа в форму. Затем их можно расположить нужным образом, изменив при необходимости размеры.



Размещая несколько копий компонента, легко забыть в конце щелкнуть на кнопке со стрелкой. Если вы случайно поместили в форму больше компонентов, чем намеревались, просто удалите лишние.

Размещение компонента в центре формы

C++Builder обеспечивает упрощенный способ размещения компонента в форме. Просто дважды щелкните на нужной кнопке в палитре компонентов, и компонент появится в форме. При этом он будет отцентрирован по горизонтали и вертикали. Компоненты, размещенные таким способом, можно перемещать в любое место формы, как и компоненты, размещенные обычным путем.

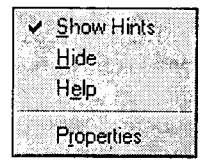


При каждом двойном щелчке на кнопке в палитре компонентов, компонент появляется в центре формы и принимает размер по умолчанию. Если вы повторите двойной щелчок несколько раз, в форму будут помещены несколько копий компонента, наложенных друг на друга. Они будут выглядеть, как один компонент, и вы можете не понять, что на самом деле одно и то же место занимают несколько компонентов. Если вы случайно сделали несколько копий, просто щелкните на лишних компонентах и удалите их из формы.

Контекстное меню палитры компонентов

Если вы установите курсор мыши на палитру компонентов и щелкните правой кнопкой, то увидите контекстное меню, соответствующее палитре компонентов (см. рис. 6.5).

Рис. 6.5
Контекстное меню палитры компонентов



Пункт Show Hints включает и выключает режим отображения всплывающих подсказок для кнопок компонентов. Пока вам не надоели подсказки, этот режим можно оставить включенным. Пункт Hide убирает палитру компонентов с экрана. Для ее отображения нужно будет выбрать в главном меню пункт View | Component Palette. Пункт Help открывает справку по C++Builder на странице, посвященной палитре компонентов. Пункт Properties открывает диалоговое окно Environment Options на странице Palette и вы можете настроить палитру компонентов желаемым образом, в частности, добавить и удалить вкладки. Вы также можете добавить, удалить или переставить компоненты на отдельных вкладках. Я расскажу об этом подробнее на десятый день, когда мы будем рассматривать опции среды.

Перемещение по палитре компонентов

Как я уже говорил, вы можете передвинуть разделительную линию, расположенную между панелью инструментов и палитрой компонентов, чтобы отвести палитре больше или меньше места в главном окне C++Builder. Если палитра компонентов не вмещает все вкладки, в ее правом верхнем углу появляются кнопки прокрутки. Аналогично этому, если отдельная вкладка палитры компонентов содержит больше кнопок, чем позволяет ширина окна,

доступ к остальным кнопкам возможен с помощью кнопок прокрутки. На рис. 6.6 показана палитра компонентов с обоими типами кнопок прокрутки.

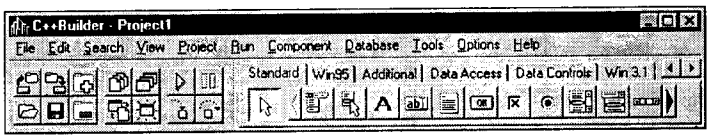


Рис. 6.6
Кнопки прокрутки палитры компонентов

Палитра компонентов не особенно сложна, но умение работать с ней необходимо для программирования в C++Builder. Теперь, закончив это небольшое отступление, мы можем вернуться обратно к основной теме.

Приложение, содержащее несколько форм

Для иллюстрации того, как C++Builder использует модули, давайте создадим приложение с несколькими формами. Это будет простая программа, отображающая вторую форму при нажатии на кнопку.

1. Создайте новый проект, выбрав в главном меню пункт File | New Application.
2. Измените значение свойства Name на MainForm, а свойства Caption — на Multiple Forms Test Program.
3. Сохраните проект. Дайте модулю имя Main, а проекту — имя Multiple.
4. Теперь поместите в форму кнопку. Присвойте свойству Name значение ShowForm2, а свойству Caption — значение Show Form 2.
5. Выберите в главном меню пункт File | New Form (или щелкните на кнопке New Form панели инструментов), чтобы создать новую форму.

После создания новая форма будет иметь имя Form1 и разместится точно поверх главной формы. Нам нужно, чтобы новая форма имела меньший размер и была более-менее отцентрирована относительно главной формы.

6. Измените размер и положение новой формы так, чтобы она была примерно в два раза меньше главной формы и располагалась в ее центре. Для перемещения формы используйте строку заголовка. Размер можно изменить перетаскиванием нижнего правого угла.
7. Измените значение свойства Name новой формы на SecondForm, а свойства Caption — на A Second Form.
8. Выберите в главном меню пункт File | Save (или щелкните на кнопке Save File панели инструментов) и сохраните файл под именем Second.
9. Выберите компонент Label и поместите его в новую форму. Измените текст свойства Caption на This is the second form. При желании измените размер и цвет текста. Отцентрируйте сообщение относительно формы. Теперь ваша форма должна выглядеть примерно как на рис. 6.7.
10. Щелкните на главной форме. Обратите внимание, что вторая форма теперь закрыта главной формой. Дважды щелкните на кнопке Show Form 2. На экране появится окно редактора кода и курсор будет размещен как раз там, где вам нужно начинать ввод текста.

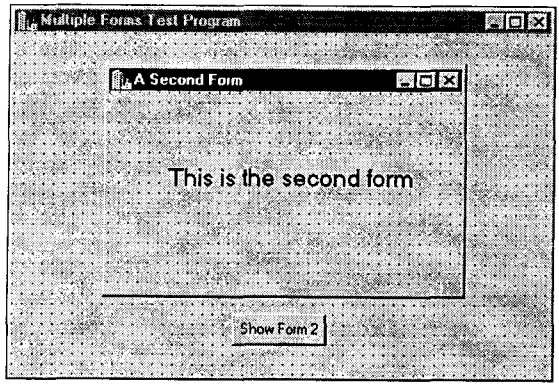


Рис. 6.7
Вид формы к этому моменту

11. Введите текст, приведенный ниже (вам нужно набрать только одну строку):

```
void __fastcall TMainForm::ShowForm2Click(TObject *Sender)
{
  SecondForm->ShowModal();
}
```

12. Запустите программу.

Вы получили сообщение об ошибке Undefined symbol 'SecondForm' (Не определено обозначение 'SecondForm'). Странно... SecondForm должно быть правильным именем, ведь именно это имя мы дали второй форме... Дайте подумать... Ага! Вспомните, что у нас есть два исходных файла с соответствующими заголовками. Но в модуле MainForm нет объявления переменной SecondForm (которая является указателем на экземпляр класса TSecondForm). Мы должны указать, где расположено объявление класса. (Для этого нужно включить заголовочный файл для SecondForm в исходный файл MainForm с помощью директивы #include, см. День 2, «Операторы управления и структуры в Си++».) Переключитесь в окно редактора кода и щелкните на вкладке Main.cpp для отображения модуля главной формы. Перейдите к началу файла. Первые несколько строк должны выглядеть следующим образом:

```
//---------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Main.h"
//---------------------------------------
```

Вы видите, что сюда включен заголовочный файл Main.h, но нет файла Second.h, потому что мы не давали указания C++Builder включить этот файл. Давайте сделаем это сейчас.

1. Выберите в главном меню пункт File | Include Unit Hdr. На экране появится диалоговое окно Include Unit, показанное на рис. 6.8.
2. Перед вами находится список доступных модулей. В данном случае единственным модулем в списке является Second. Щелкните на имени Second, а затем на кнопке OK, чтобы закрыть диалоговое окно.

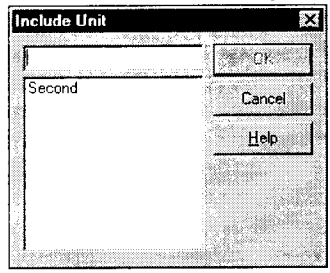


Рис. 6.8
Диалоговое окно Include Unit



Диалоговое окно Include Unit отображает только те модули проекта, которые еще не включены в данный модуль. Включенные модули не содержатся в списке.

После щелчка на кнопке OK C++Builder мгновенно добавит в файл директиву #include для Second.h:

```
//---------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Main.h"
#include "Second.h"
//---------------------------------------
```

Теперь в модуль Main включено объявление класса из модуля Second. Щелкните на кнопке Run, чтобы запустить программу. На этот раз компиляция пройдет без задержек и программа заработает. Когда вы щелкнете на кнопке Show Form 2, на экране появится вторая форма. Вы можете закрыть ее, нажав кнопку закрытия окна в строке заголовка.

Как видите, C++Builder помогает вам управлять модулями. Вы должны использовать опцию Include Unit Hdr, чтобы обеспечить модулям доступ к объявлениям классов, использующихся в других модулях, но в большинстве случаев C++Builder освобождает вас от забот, связанных с исходными файлами. В будущем, когда ваши программы станут более сложными, вам потребуется уделять больше внимания организации исходных файлов, но на данном этапе C++Builder берет на себя большую часть работы.

Теперь давайте рассмотрим различные опции компилятора, доступные при написании программ в C++Builder.

Компиляция, построение и компоновка

Каждый раз, когда вы нажимаете кнопку Run, C++Builder компилирует и компонует вашу программу. Однако при этом нет необходимости компилировать каждый модуль проекта. C++Builder обрабатывает только те модули, которые были изменены с момента последней компиляции. Отслеживание изменений в файлах производится автоматически, поэтому вам не нужно предпринимать какие-либо специальные действия.

Обычно вам нужно сразу видеть результаты изменений в программе. В таких случаях вы нажимаете кнопку Run, после чего программа компилируется, компонуется и выполняется. Но иногда запуск программы не требуется. Например, вам нужно откомпилировать программу только для того, чтобы убедиться в отсутствии ошибок. C++Builder имеет три дополнительных

пункта меню, позволяющих управлять процессом компиляции/компоновки. Если вы выберете в главном меню пункт Project, то увидите три подпункта: Compile Unit (Откомпилировать модуль), Make (Создать) и Build All (Построить все). Давайте рассмотрим их, начиная с самого простого и заканчивая наиболее сложным (с точки зрения компилятора).

Опция Compile Unit очень полезна на практике. Она дает возможность откомпилировать модуль в окне редактора кода с выдачей всех предупреждений и сообщений об ошибках. Это самый быстрый способ проверить текст на отсутствие синтаксических ошибок. Компоновка при выборе опции Compile Unit не производится.

Опция Make предназначена для компиляции всех модулей, которые были изменены с момента последнего создания соответствующих объектных файлов, а также компоновки проекта. Фактически это занимает ненамного больше времени, чем просто компиляция с помощью опции Compile Unit. Используйте опцию Make, если вам нужно создать исполняемый файл, но не запускать после этого программу.



Сочетание клавиш для Make – Ctrl+F9.

Опция Build All выполняется дольше всего. При ее использовании компилируется каждый модуль, независимо от того, был ли он изменен. После компиляции всех модулей C++Builder компонует проект. До сих пор мы позволяли C++Builder автоматически добавлять модули к нашим проектам. В дальнейшем вам может потребоваться вручную редактировать исходные файлы, чтобы включать заголовки и необходимые директивы. Вы можете изменять даже информационный файл проекта. Время от времени программа может начать вести себя совершенно непонятным образом. Используя опцию Build All, можно лучше разобраться в возникших проблемах. Иногда построение программы заново полностью устраняет ошибки компилятора и компоновщика.



Получив неожиданное сообщение об ошибке от компилятора или компоновщика, первым делом попробуйте заново построить проект. Если ошибка была вызвана простым несоответствием версий файлов, она будет устранена. В противном случае придется искать причину.

Если в программе были обнаружены ошибки, то независимо от способа компиляции проекта, в окне состояния компилятора появится сообщение There are errors. вместе с информацией о количестве ошибок и предупреждений. На рис. 6.9 показано окно состояния компилятора при наличии ошибок.

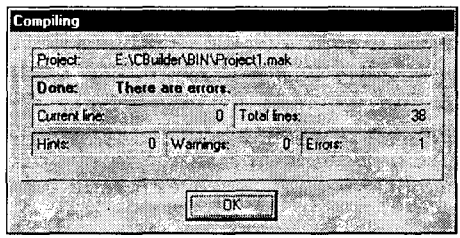


Рис. 6.9
Окно состояния компилятора с информацией об ошибках и предупреждениях

После нажатия на кнопку OK окно состояния компилятора исчезнет и на передний план переместится окно редактора кода, в котором будет выделена первая из ошибочных строк. В нижней части редактора кода при этом откроется дополнительное окно со списком ошибок и предупреждений. В случае успешной компиляции модуля, создания или построения проекта, вы можете немедленно запустить программу кнопкой Run.

Компиляция и построение других программ на Си++

Сила C++Builder заключена в его среде визуального программирования. Эта среда непосредственно связана с VCL и не может использоваться отдельно от нее. Создание приложений на основе VCL позволяет в полной мере реализовать возможности C++Builder. Несмотря на это, иногда может возникнуть необходимость в написании приложений других типов. C++Builder является стандартным компилятором Си++, поэтому с его помощью вы можете также компилировать любые 32-разрядные программы на Си++.

Возможно, наиболее очевидным примером «другой» программы может служить *динамически компонуемая библиотека (dynamic link library — DLL)*. DLL могут казаться чем-то вроде черной магии, но на самом деле они не очень сложны. DLL представляют собой просто откомпилированный код, который можно вызывать из приложений. После создания динамической библиотеки и включения в исходный файл вашей основной программы необходимого заголовочного файла, вызов функции DLL не отличается от вызова обычной функции, содержащейся в основной программе.

Другим типом приложений, которые вы можете создавать с помощью C++Builder, являются консольные приложения. Мы уже написали несколько консольных приложений Win32 при изучении основ языка Си++. Эти приложения удобно использовать для учебных целей и в качестве простых тестовых программ. В виде консольных приложений также могут быть оформлены небольшие утилиты, Web-серверы и почтовые серверы, а также целый ряд других программ. Вообще говоря, любое приложение, не требующее графического интерфейса, может быть успешно сделано консольным. Ранее в этой главе я говорил о входящих в состав C++Builder компиляторе Си++, компиляторе ресурсов и компоновщике. Все эти программы являются консольными приложениями, запускаемыми из-под C++Builder.



Вы можете также компилировать программы, написанные с использованием MFC и OWL. Это позволяет использовать один компилятор для всех ваших разработок, независимо от используемой библиотеки классов. Хотя вы, вероятно, не будете разрабатывать с помощью C++Builder крупные приложения на основе MFC или OWL, нет причин, которые могли бы в принципе помешать этому. Разумеется, для построения OWL- и MFC-приложений необходимо иметь соответствующие библиотечные и заголовочные файлы. Листинг 6.1 содержит OWL-версию программы Hello World. Чтобы ее откомпилировать, прежде всего сконфигурируйте C++Builder для построения OWL-приложений (необходимая информация содержится в документации к C++Builder). Затем введите программу с клавиатуры или загрузите с CD-ROM, прилагаемого к этой книге. После компиляции вы получите OWL-программу, построенную с помощью C++Builder.

Листинг 6.1. OWLHELLO.CPP

```
 1: #include <owl\applicat.h>
 2: #include <owl\framewin.h>
 3: #pragma hdrstop
 4: class TestWindow : public TWindow {
 5:   public:
 6:     TestWindow() : TWindow(0, 0, 0) {}
 7:   protected:
 8:     void Paint(TDC&, bool, TRect&);
 9: };
10: void
11: TestWindow::Paint(TDC& dc, bool, TRect&)
12: {
13:   TRect rect = GetClientRect();
14:   dc.DrawText("Hello World!", -1,
15:     rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
16: }
17: class TestApp : public TApplication {
18:   public:
19:     TestApp() : TApplication("") {}
20:     void InitMainWindow();
21:     {
22:       TFrameWindow* frame = new TFrameWindow(
23:         0, "OWL Hello World", new TestWindow);
24:       SetMainWindow(frame);
25:     }
26: };
27: int OwlMain(int, char* [])
28: {
29:   return TestApp().Run();
30: }
```

Дополнительные сведения о формах C++Builder

Прежде чем продолжить рассказ об интегрированной среде разработки C++Builder, необходимо посвятить еще некоторое время формам. Работая с книгой, вы уже видели несколько примеров использования форм, а завтра мы перейдем к изучению редактора форм. Но до этого вам нужно получить еще некоторые базовые сведения о формах, к изложению которых я и приступаю.

Формы для главного окна

Формы являются основными составляющими, из которых строится приложение C++Builder. Каждое GUI-приложение имеет по крайней мере одну форму, которая служит главным окном. Форма главного окна может быть просто пустым окном, может содержать элементы управления или растровое изображение. В типичной Windows-программе главное окно обычно имеет меню. Окно может также содержать дополнительные элементы, например, панель инструментов или строку состояния. При создании главного окна приложения существует большое разнообразие вариантов. Каждое приложение уникально и предъявляет свои требования.

Формы для диалоговых окон

Там, где традиционные Windows-программы используют диалоговые окна, в C++Builder также применяются формы. Фактически, для пользователя нет разницы между настоящим диалоговым окном и заменяющей его формой. Диалоговые окна обычно имеют следующие характерные черты, отличающие их от обычных окон:

- Фиксированный размер. Диалоговые окна предназначены, как правило, для выполнения определенной задачи, и изменение размера либо не имеет смысла, либо нежелательно.
- Наличие кнопки OK. Некоторые диалоговые окна имеют кнопку Close, выполняющую ту же функцию. Для простых диалоговых окон, наподобие окна About, эта кнопка чаще всего единственная.
- Наличие кнопок Cancel и Help.
- Наличие вкладок. Диалоговое окно может содержать несколько вкладок, предоставляющих пользователю возможность выбора. При щелчке на заголовке вкладки открывается другая страница диалогового окна.
- Использование клавиши Tab для перемещения между элементами управления диалогового окна.

Разумеется, для каждого правила есть свои исключения. Большинство диалоговых окон имеют стандартные свойства, но некоторые могут выполнять специфические задачи и соответственно отклоняться от стандарта в ту или другую сторону.

Диалоговые окна в C++Builder реализованы немного иначе, чем в других средах программирования. Поэтому сначала давайте разберемся, как поддерживаются диалоговые окна там, а затем посмотрим, как это сделано в C++Builder.

Диалоговые окна в традиционных Windows-программах

В традиционной Windows-программе (написанной на Си или с использованием одной из библиотек классов) диалоговые окна создаются с помощью специального редактора. В большинстве случаев, редактор диалоговых окон представляет собой визуальный инструмент, работающий в чем-то аналогично редактору форм C++Builder. Когда пользователь завершает разработку диалогового окна, на основе его визуального представления формируется *сценарий ресурсов* (текстовый файл, который позже преобразуется в двоичный файл ресурсов с помощью компилятора ресурсов). Для иллюстрации сказанного, давайте рассмотрим диалоговое окно на рис. 6.10.

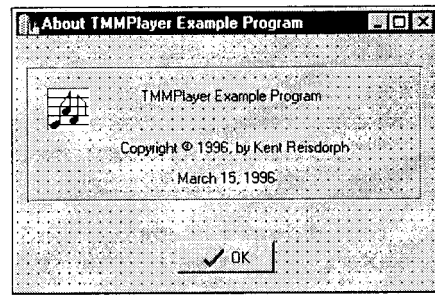


Рис. 6.10
Типичное диалоговое окно About

На этом рисунке представлено типичное диалоговое окно About. Оно содержит имя программы, информацию об авторских правах и значок приложения. Определение сценария ресурсов для этого диалогового окна приведено в листинге 6.2.

Листинг 6.2. Определение ресурсов диалогового окна

```
 1: IDD_ABOUT DIALOG 58, 53, 194, 119
 2: STYLE DS_MODALFRAME | WS_POPUP |
 3:   WS_VISIBLE | WS_CAPTION | WS_SYSMENU
 4: CAPTION "About TMMPlayer Example Program"
 5: FONT 8, "MS Sans Serif"
 6: {
 7: DEFPUSHBUTTON "OK", IDOK, 72, 96, 50, 14
 8: CTEXT "TMMPlayer Example Program", -1, 48, 22, 128, 8
 9: CTEXT "Copyright c 1996 by Kent Reisdorh", -1, 32, 47, 136, 8
10: CTEXT "March 15, 1996", -1, 24, 59, 146, 8
11: CONTROL "", 99, "button", BS_GROUPBOX |
12:   WS_CHILD | WS_VISIBLE | WS_GROUP, 12, 4, 176, 70
13: CONTROL 1, 1, "static", SS_ICON |
14:   SS_SUNKEN | WS_CHILD | WS_VISIBLE, 24, 17, 20, 20
15: }
```

Сценарий ресурсов содержит информацию, на основе которой Windows строит диалоговое окно во время выполнения программы. Сюда относятся количество и тип элементов управления в окне, их размеры, положение, текст, опции и т.д. Разумеется, сценарий ресурсов содержит те же данные о самом диалоговом окне.

Некоторые программисты вообще не используют редактор диалоговых окон, а предпочитают сразу набирать описание окна в текстовом редакторе. Хотя я не могу осуждать такой подход к разработке окон, нужно заметить, что для большинства программистов он будет не самым эффективным. Визуальное проектирование занимает гораздо меньше времени.

Обычно определения для всех диалоговых окон приложения содержатся в едином файле сценария ресурсов, который имеет расширение .RC. В процессе создания программы сценарий ресурсов компилируется в .RES-файл (двоичный файл ресурсов), который затем компонуется с исполняемым файлом. Во время выполнения программы диалоговое окно отображается модально или немодально, в зависимости от его назначения. При вызове диалогового окна Windows загружает соответствующий ресурс из исполняемого файла, строит окно и отображает его на экране.



Модальное (modal) диалоговое окно должно быть закрыто перед продолжением работы с приложением. При вызове такого диалогового окна главное окно приложения блокируется. Большинство диалоговых окон являются модальными. Примером такого окна служит окно состояния компилятора в C++Builder.

Немодальное (modeless) диалоговое окно позволяет пользователю продолжать работу с приложением. В качестве примера можно привести диалоговое окно Find в некоторых текстовых процессорах.

Теперь, зная основные принципы поддержки диалоговых окон в традиционных Windows-программах, давайте рассмотрим диалоговые окна C++Builder.

Диалоговые окна в C++Builder

В C++Builder диалоговое окно является просто разновидностью формы. Вы создаете это окно так же, как и форму для главного окна или любую другую форму. Чтобы зафиксировать размер диалогового окна, вы можете изменить значение свойства BorderStyle на bsDialog или bsSingle. При использовании bsDialog ваше окно будет содержать в строке заголовка только кнопку закрытия, что традиционно принято для диалоговых окон. Кроме этого, вам не нужно предпринимать никаких специальных действий для того, чтобы форма вела себя как диалоговое окно. Все формы C++Builder имеют встроенную поддержку управления клавишей Tab. Вы можете установить порядок переключения с помощью свойств TabOrder отдельных элементов управления, расположенных в диалоговом окне.

Диалоговое окно C++Builder (так же, как и любая форма) может быть модальным или немодальным. Для отображения модального окна нужно вызвать метод ShowModal() класса TForm. Чтобы создать немодальное диалоговое окно, используется метод Show().

Давайте добавим окно About к ранее созданному проекту со многими формами. Если этот проект сейчас не открыт, выберите в главном меню пункт File | Open или щелкните на кнопке Open Project панели инструментов и найдите нужный файл (вы должны были сохранить его под именем Multiple).



C++Builder создает список файлов и проектов, которые вы использовали последними. Выберите пункт File | Reopen для просмотра списка MRU (Most Recently Used). Этот список разделен на две части. Верхняя часть содержит последние использованные проекты, а нижняя — отдельные файлы. Чтобы открыть файл или проект, просто щелкните на его имени.

Прежде всего мы добавим к форме кнопку, которая будет отображать диалоговое окно About.

1. Выберите компонент button в палитре компонентов и поместите кнопку в форму.
2. Расположите две кнопки так, чтобы форма имела сбалансированный вид.
3. Измените значение свойства Name новой кнопки на AboutButton, а значение свойства Caption — на About...
4. Дважды щелкните на только что созданной кнопке AboutButton. На экране появится окно редактора кода с курсором, установленным в функции-обработчике события. Добавьте следующую строку кода, начиная с положения курсора:

```
AboutBox->ShowModal();
```

Окно About еще не создано, но оно получит имя AboutBox, поэтому строку кода можно ввести уже сейчас.

Теперь мы создадим само диалоговое окно.

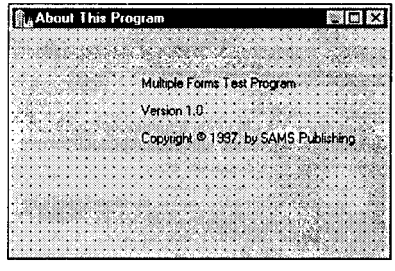
1. Создайте новую форму (для этого нажмите кнопку New Form панели инструментов). Уменьшите ее размер до размера типичного окна About (можно ориентироваться на ранее созданную форму SecondForm).
2. Измените значение свойства Name на AboutBox, а значение свойства Caption — на About This Program.

3. Найдите свойство BorderStyle (оно расположено как раз над Caption) и измените его значение на bsDialog.
4. Теперь добавьте в окно три текстовых метки. Отредактируйте текст так, чтобы окно соответствовало рис. 6.11. (Вы можете, конечно, ввести любой другой текст.) Значения свойств Name для этих компонентов можно не изменять, поскольку мы не будем использовать эти свойства в дальнейшем.



Символ копирайта (c) имеет код ASCII 169. Для ввода этого символа нажмите и удерживайте клавишу Alt, после чего наберите на цифровой клавиатуре 0169 (убедитесь, что режим NumLock включен). После отпускания клавиши Alt символ появится на экране. Таким образом можно вставить любой символ с кодом ASCII. При этом, однако, необходимо ввести все четыре цифры. Например, буква A имеет ASCII-код 65. Для ее ввода нужно нажать Alt и набрать на цифровой клавиатуре 0065.

Рис. 6.11
Диалоговое окно About с текстовыми метками



Следующим шагом будет добавление значка в окно About.

1. Щелкните на вкладке Additional палитры компонентов и выберите компонент Image. Поместите его слева от текста.
2. Найдите свойство AutoSize компонента Image и измените его значение на true.
3. Найдите свойство Picture и дважды щелкните в столбце значений. Откроется диалоговое окно редактора изображений.
4. Нажмите кнопку Load. В диалоговом окне File Open перейдите в каталог \CBuilder\Images\Icons и выберите значок из списка. Нажмите кнопку OK. Выбранный значок появится в окне редактора. Нажмите еще раз кнопку OK, чтобы закрыть окно. Значок будет помещен в форму. Обратите внимание, что размер компонента Image автоматически подстроился под размер значка.
5. Расположите значок желаемым образом.

Сейчас нам нужно поместить в форму кнопку OK. При этом я покажу вам новый компонент.

1. Щелкните на вкладке Additional палитры компонентов, если она еще не открыта. Выберите компонент BitBtn и поместите его в нижнюю часть формы, отцентрировав по горизонтали.
2. Найдите свойство Kind и измените его значение на bkOK. Заметьте, что на кнопке появилась зеленая отметка, а свойство Caption изменило зна-

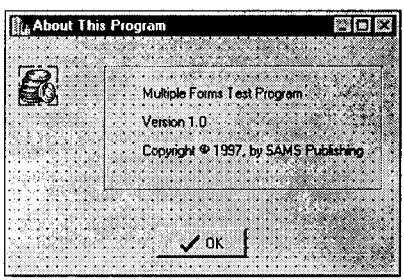
чение на OK. Это все, что мы должны сделать с кнопкой. Компонент BitBtn уже содержит код для закрытия формы при нажатии кнопки OK.

И наконец, последний штрих в оформлении окна About.

1. Найдите кнопку Bevel (на вкладке Additional палитры компонентов) и щелкните на ней.
2. Переместите компонент в форму, но перед тем, как отпустить кнопку мыши, обведите рамкой строки текста. Компонент появится после отпускания кнопки. Если это не удалось сделать точно, измените размеры компонента или переместите его.
3. Найдите свойство Shape и измените его значение на bsFrame. Рамка вокруг текста станет объемной.

Сейчас ваша форма должна выглядеть примерно так же, как на рис. 6.12. Сохраните модуль (File | Save), дав ему имя About.

Рис. 6.12
Законченное окно About



Все ли готово для компиляции и запуска программы? Пока нет. Мы должны включить заголовочный файл для модуля About в модуль главной формы.

1. Переключитесь в окно редактора кода (клавиша F12) и выберите вкладку Main.cpp.
2. Выберите в главном меню пункт File | Include Unit Hdr.
3. Выберите модуль About в открывшемся диалоговом окне и нажмите кнопку OK.

Теперь можно запускать программу. Нажмите кнопку Run. Когда программа заработает, нажмите кнопку About, и вы увидите диалоговое окно. Обратите внимание, что это окно является модальным (вы не сможете вернуться к главному окну, пока открыто диалоговое) и не может изменять размер. Форма About ведет себя во всех отношениях так же, как и обычное диалоговое окно Windows.



Стандартные диалоговые окна, представленные классами TOpenDialog, TSaveDialog, TFontDialog и т.д., не являются формами C++Builder. Эти окна, содержащиеся в системном файле COMDLG32.DLL, предоставляются самой операционной системой и доступны всем приложениям в Windows. Классы диалоговых окон VCL инкапсулируют стандартные диалоговые окна Windows для упрощения их использования.



В C++Builder включены несколько готовых форм, использование которых позволяет максимально ускорить разработку диалоговых окон. Я расскажу об этом на девятый день, «Создание приложений в C++Builder».

Сравнение вторичных и диалоговых окон

Вторичное окно (secondary window) — это форма, которая отображается из главного окна. Когда же форма является вторичным окном, а когда диалоговым? В конечном счете, разницы между вторичными и диалоговыми окнами в C++Builder нет. Вы можете создавать окна, похожие как на диалоговые, так и на обычные окна Windows. По сути дела, все они являются формами, поэтому нет большого смысла говорить о разнице между диалоговым окном и вторичной формой. В традиционных системах программирования процедуры создания диалоговых и вторичных окон различаются. C++Builder освобождает вас от этих неудобств и позволяет рассматривать оба типа окон как одинаковые.

Модель многодокументного интерфейса

До сих пор мы создавали только приложения с *однодокументным интерфейсом (SDI — single-document interface)*. SDI-приложения имеют одно главное окно и могут при необходимости отображать диалоговые окна, но не имеют подчиненных окон.

Некоторые программы используют модель *многодокументного интерфейса (MDI — multiple-document interface)*. MDI-приложения состоят из главного окна (MDI-предка) и подчиненных окон (MDI-потомков). Примером программ, использующих модель MDI, служат редактор системной конфигурации Windows (SYSEDIT) и менеджер программ в Windows 3.1. Одним из наиболее очевидных свойств модели MDI является то, что подчиненные окна ограничены главным окном и могут перемещаться только в пределах этого окна. MDI-приложения практически всегда имеют в главном меню пункт Window. Этот пункт обычно содержит подпункты Cascade и Tile, позволяющие расположить подчиненные окна мозаикой или каскадом. При сворачивании подчиненного окна его пиктограмма остается в пределах главного окна. Если же сворачивается обычное (не MDI) порожденное окно, пиктограмма размещается на рабочем столе Windows.

Для создания MDI-приложения в C++Builder вы должны установить для свойства FormStyle главной формы значение fsMDIForm. При этом свойство FormStyle каждого из подчиненных окон MDI должно иметь значение fsMDIChild. Чтобы MDI-приложение заработало, требуется совсем немного — создать форму для главного окна и необходимое количество форм для подчиненных окон.

Основные свойства форм

Класс TForm имеет большое количество свойств. Некоторые из них весьма туманны и используются редко; другие, напротив, находят широкое применение. Я остановлюсь здесь на наиболее часто используемых свойствах. При этом я не собираюсь рассказывать об очевидных свойствах типа Color, Left,

Top, Width и Height, за исключением их отдельных возможностей, о которых вам следует знать.

Свойства, доступные во время разработки и выполнения программы

Свойства, собранные в этом разделе, могут быть установлены как во время разработки программы, так и во время ее выполнения. Почти все они могут быть также считаны во время выполнения программы.

`ActiveControl`

Свойство ActiveControl используется для выбора элемента управления, который будет активным при обращении к форме. Например, вам может потребоваться, чтобы при открытии диалогового окна был выбран один из элементов управления. Во время разработки столбец значений свойства ActiveControl содержит список всех компонентов формы. Выбранный из списка компонент будет активным при первом отображении формы.

`AutoScroll, HorzScrollBar и VertScrollBar`

Взятые вместе, свойства AutoScroll, HorzScrollBar и VertScrollBar управляют линейками прокрутки для формы. Если AutoScroll имеет значение true, линейки прокрутки автоматически появляются, когда форма становится слишком мала для отображения всех ее компонентов. Свойства HorzScrollBar и VertScrollBar сами имеют несколько свойств, управляющих работой линеек прокрутки.

`BorderStyle`

Свойство BorderStyle определяет тип рамки вокруг формы. Значением по умолчанию является bsSizeable, что соответствует окну переменного размера. Для фиксации размера окна используются значения bsDialog и bsNone.

`ClientWidth и ClientHeight`

С помощью свойств ClientWidth и ClientHeight вы можете указать ширину и высоту рабочей области вместо ширины и высоты всей формы. (*Рабочей областью (client area)* формы называется область ниже строки заголовка и меню, ограниченная рамкой окна.) Используйте это свойство, когда рабочая область должна иметь определенные размеры, а остальная часть окна — подстраиваться под них. Установка ClientWidth и ClientHeight автоматически изменяет значения свойств Width и Height.

`Font`

Свойство Font определяет шрифт, используемый в пределах формы. Важно понимать, что установленный таким образом шрифт будет использоваться всеми компонентами формы. Отсюда следует, что установкой нового шрифта для формы вы можете изменить шрифты сразу во всех ее компонентах. Если шрифт для какого-либо компонента был установлен отдельно, он останется прежним.

`FormStyle`

Это свойство обычно имеет значение fsNormal. Если вы хотите, чтобы форма всегда располагалась поверх остальных окон, используйте стиль fsStayOnTop. Формы MDI должны использовать стиль fsMDIForm, а подчиненные окна MDI — стиль fsMDIChild. Формы и подчиненные окна MDI рассматривались выше в разделе «Модель многодокументного интерфейса».

`HelpContext`

Свойство HelpContext используется для установки идентификатора контекстной справки (context help ID) формы. Если для формы используется контекстная справка, справочная система Windows будет активизирована при нажатии клавиши F1. Данный идентификатор определяет страницу справочного файла, которая должна быть отображена справочной системой.

`Icon`

Свойство Icon определяет значок, который будет использоваться в строке заголовка формы при выполнении программы, а также для свернутой формы. В отдельных случаях значение этого свойства игнорируется, например, когда FormStyle имеет значение fsDialog.

`Position`

Свойство Position определяет размер и положение формы при первом отображении. Возможны три варианта: poDesigned, poDefault и poScreenCenter. Если установлено значение poDesigned, форма отображается на том же месте, что и при разработке. poDefault разрешает Windows установить размер и положение формы в соответствии с обычным алгоритмом Z-упорядочивания. (Z-упорядочивание используется Windows для определения позиции нового окна на экране. Если информация о его размещении отсутствует, то окно будет отображено непосредственно под последним открытым окном со сдвигом вправо.) Если установлена опция poScreenCenter, то форма каждый раз будет отображаться в центре экрана.

`Visible`

Свойство Visible определяет, будет ли форма изначально видима. Это свойство не очень полезно при разработке, но во время выполнения программы оно позволяет определить, видима ли форма в данный момент. Его можно также использовать для скрытия или отображения формы.

`WindowState`

Свойство WindowState может быть считано для определения текущего состояния формы (развернута, свернута или имеет нормальный размер). Его можно также использовать для начальной установки размера формы. Возможные значения: wsMinimized, wsMaximized и wsNormal.

Свойства, доступные только во время выполнения программы

Доступ к некоторым свойствам возможен только через код во время выполнения программы. Далее приведены наиболее часто используемые свойства этого типа.

`ActiveMDIChild`

При чтении свойства ActiveMDIChild возвращается указатель на активное в данный момент подчиненное окно MDI. Свойство доступно только для чтения.

`Canvas`

Холст (canvas) представляет поверхность формы, доступную для рисования. Свойство Canvas обеспечивает вам доступ к холсту формы. Используя это свойство, вы можете отображать в форме растровые изображения, линии, кривые или текст во время работы программы. В большинстве случаев вы будете использовать для отображения текста компонент Label, для графики —

компонент Image, а для кривых — компонент Shape. Однако, иногда может потребоваться изобразить что-либо на холсте в ходе выполнения программы, и тогда свойство Canvas позволит вам сделать это. Это свойство можно также использовать для сохранения образа формы в дисковом файле.

`ClientRect`

Свойство ClientRect содержит координаты рабочей области формы. Оно полезно во многих ситуациях. Например, вам может потребоваться узнать ширину и высоту рабочей области формы для размещения в ее центре растрового изображения.

`Handle`

Свойство Handle возвращает дескриптор окна (HWND) для формы. Используйте это свойство, когда вам нужно передать дескриптор окна функции API Windows.

`ModalResult`

Свойство ModalResult используется для закрытия модального окна. Если у вас есть диалоговое окно с кнопками OK и Cancel, вы можете присвоить ModalResult значение mrOK при нажатии кнопки OK или значение mrCancel при нажатии кнопки Cancel. Вызвавшая окно форма затем может считать ModalResult для определения того, какая кнопка была нажата. К другим возможным значениям относятся mrYes, mrNo и mrAbort.

`Owner`

Свойство Owner представляет собой указатель на владельца формы. Владелец формы — это объект, отвечающий за ее уничтожение после использования. С другой стороны, объектом, порождающим компонент, является окно (форма или другой компонент), которое содержит данный компонент. В случае главной формы приложение является одновременно и владельцем, и порождающим объектом формы. Для остальных компонентов владельцем может быть форма, но порождающим объектом должен быть другой компонент, например, панель.

`Parent`

Свойство Parent представляет собой указатель на порождающий объект формы. Различие между Owner и Parent объяснено выше, в разделе «Owner».

Методы форм

Формы по своей сути также являются компонентами. А это значит, что они имеют много общих с компонентами методов. К этим методам, в частности, относятся Show(), ShowModal() и Invalidate(). Однако существуют методы, используемые только в формах. Как и раньше, я остановлюсь лишь на наиболее часто используемых методах.

`BringToFront()`

Метод BringToFront() используется для расположения данной формы поверх всех остальных форм приложения.

`Close() и CloseQuery()`

Метод Close() закрывает форму после предварительного вызова CloseQuery(), проверяющего, возможно ли нормальное закрытие. Функция Close-

Query(), в свою очередь, вызывает обработчик события OnCloseQuery. Если переменная типа bool, передаваемая обработчику OnCloseQuery, имеет значение false, то форма не закрывается. При значении true форма закрывается нормальным образом. Вы можете использовать обработчик события OnCloseQuery для вывода запроса на сохранение файла и для проверки возможности закрытия формы.

`Print()`

Метод Print() выводит на печать содержимое формы. Печатается только рабочая область формы, без подписи, строки заголовка или рамок. Print() удобен для быстрой распечатки формы.

`ScrollView()`

Метод ScrollView() прокручивает форму, пока указанный компонент не окажется в поле зрения.

`SetFocus()`

Метод SetFocus() активизирует форму и перемещает ее поверх других окон. Если форма содержит компоненты, будет активизирован тот из них, который определяется свойством ActiveControl (см. свойство ActiveControl в разделе «Свойства, доступные во время разработки и выполнения программы»).

`Show()` и `ShowModal()`

Методы Show() и ShowModal() отображают форму. Метод Show() отображает форму как немодальную, при этом другие формы могут быть активизированы, пока данная форма видима. Метод ShowModal() отображает форму модально. Модальная форма должна быть закрыта для продолжения работы с приложением.

Методы MDI

Несколько методов форм выполняют специфические MDI-операции. Метод ArrangeIcons() упорядочивает значки всех свернутых подчиненных окон в главном окне. Метод Cascade() располагает каскадом все несвернутые подчиненные окна. Метод Tile() располагает открытые подчиненные окна мозаикой. Метод Next() активизирует (располагает сверху) следующее окно в списке подчиненных окон, а метод Previous() — предыдущее. Методы MDI применимы только к главным окнам MDI.

События форм

Формы могут реагировать на самые различные события. Некоторые из них, наиболее часто используемые, перечислены в следующих разделах.

`OnActivate`

Событие OnActivate соответствует активизации формы. Форма может быть активизирована в результате ее первоначального создания, при переключении от одной формы к другой или при переключении между приложениями.

OnClose и OnCloseQuery

При закрытии приложения возникает событие OnClose. OnClose вызывает событие OnCloseQuery для проверки возможности нормального закрытия формы. Если OnCloseQuery возвращает false, форма не закрывается.

OnCreate

Событие OnCreate происходит при первоначальном создании формы. Для каждого экземпляра отдельной формы будет вызвано только одно событие OnCreate. Обработчик OnCreate можно использовать для выполнения инициализирующих действий.

OnDestroy

Событие OnDestroy противоположно OnCreate. Используйте это событие для освобождения динамически выделенной памяти или других завершающих действий.

OnDragDrop

Событие OnDragDrop происходит при помещении объекта в форму. Его можно использовать, если ваша форма поддерживает технику drag-and-drop.

OnMouseDown, OnMouseMove и OnMouseUp

События OnMouseDown, OnMouseMove и OnMouseUp используются для реакции на щелчки мышью на форме.

OnPaint

Событие OnPaint происходит тогда, когда форма по какой-либо причине требует перерисовки. Используйте это событие для вывода изображений, которые должны постоянно присутствовать на экране. В большинстве случаев отдельные компоненты самостоятельно обновляют изображения, но иногда вам может потребоваться рисовать на самой форме.

OnResize

Событие OnResize возникает при каждом изменении размера формы. Его можно использовать для выравнивания объектов в форме или для перерисовки формы.

OnShow

Событие OnShow происходит перед тем, как форма станет видимой. Вы можете использовать это событие для выполнения операций, которые требуются непосредственно перед отображением формы.

Инспектор объектов

Инспектор объектов является составной частью IDE C++Builder. Он работает совместно с редактором форм, помогая создавать компоненты. Я собираюсь перейти к редактору форм завтра, но перед этим хочу немного рассказать об инспекторе объектов.

Инспектор объектов предназначен для установки доступных на этапе разработки свойств, которые определяют поведение компонента во время выполнения программы. Окно инспектора объектов состоит из трех основных частей:

- Селектора компонентов
- Вкладки Properties
- Вкладки Events

Вы уже использовали инспектор объектов, поэтому я кратко повторю то, что вы уже знаете, и покажу несколько новых вещей.

Селектор компонентов

Селектор компонентов представляет собой выпадающий комбинированный список, расположенный в верхней части инспектора объектов. Селектор компонентов позволяет вам выбирать объект для просмотра и редактирования.



Обычно самый быстрый способ выделить компонент — это щелкнуть на нем в форме. Выбор с помощью селектора удобен тогда, когда данный компонент закрыт другим компонентом или находится за пределами видимой области формы.

Селектор компонентов отображает имя компонента и класс, от которого этот компонент произведен. Например, компонент с именем Memo будет представлен в окне селектора следующим образом:

```
Memo: TMemo
```

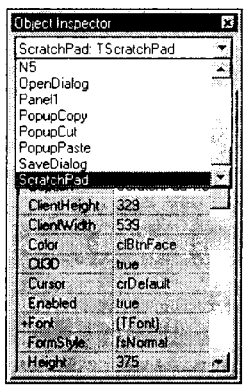
Имя класса не выводится в выпадающем списке компонентов, а присутствует только в верхней части окна селектора. Для выбора компонента сначала щелкните на кнопке вызова списка, а затем на нужном компоненте.



Селектор компонентов показывает только те компоненты, которые присутствуют в текущей форме, а также имя самой формы. Другие формы вместе со своими компонентами не будут отображаться, пока они не сделаны активными в редакторе форм.

Компонент, выбранный в селекторе, будет выделен и в форме. Вкладки Properties и Events будут отображать свойства и события для выделенного компонента. (Помните, что форма также является компонентом.) На рис. 6.13 показано окно инспектора объектов с открытым списком селектора компонентов.

Рис. 6.13
Список селектора компонентов



Вкладка Properties

Вкладка Properties инспектора объектов отображает все свойства выбранного компонента, доступные на стадии разработки. Вкладка разделена на два столбца. Столбец Property слева показывает имя свойства. Столбец Value справа предназначен для выбора или ввода значения свойства.

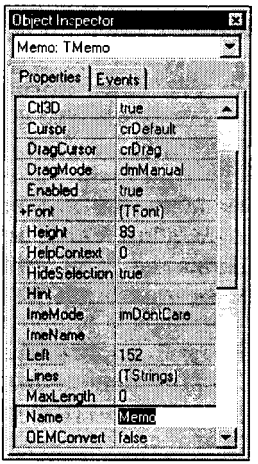
Если выбранный компонент имеет больше свойств, чем вмещает окно инспектора объектов, можно использовать линейку прокрутки.



Если вы выбрали в форме несколько компонентов, инспектор объектов покажет их общие свойства. Это дает возможность изменять свойства нескольких компонентов одновременно. Например, для изменения ширины нескольких кнопок вы можете выделить эти кнопки и затем изменить свойство Width. После нажатия Enter или перехода к другому свойству ширина всех выделенных компонентов примет новое значение.

На рис. 6.14 показан инспектор объектов с выбранным компонентом Memo.

Рис. 6.14
Инспектор объектов, показывающий свойства компонента Memo



В пятый день, «Библиотеки классов Си++ и модель визуальных компонентов», я говорил о свойствах. Мы выяснили, что свойства могут быть целыми числами, перечислениями, наборами, строками, объектами, а также иметь другие типы. При работе со свойствами инспектор объектов учитывает их тип. Например, свойство Top представлено величиной целого типа. Поскольку int является базовым типом данных, никакой специальной поддержки для него не требуется и редактор данного свойства достаточно прост. Редактор целочисленных свойств позволяет набрать число прямо в столбце Value.



В большинстве случаев редактор свойств проверяет правильность ввода целых чисел. Свойство Width, например, не может иметь отрицательное значение. Если вы попытаетесь ввести отрицательное число, C++Builder сделает ширину равной минимально допустимой для данного компонента (обычно это 0). Если вы введете набор символов вместо числа, C++Builder выдаст сообщение об ошибке.

Во многих случаях редактор будет содержать список возможных значений. Значения таких свойств относятся к перечислимому или логическому (булеву) типу. Если вы щелкнете на столбце Value, то увидите справа кнопку вызова списка. Нажатие на эту кнопку приведет к отображению выпадающего списка возможных значений.



Если вы будете дважды щелкать на столбце Value для свойства такого типа, редактор будет циклически перебирать все возможные значения. Например, для быстрого изменения значения типа bool достаточно дважды щелкнуть на нем. Поскольку вариантов всего два – true и false, двойной щелчок просто переключает эти значения.

Если вы внимательнее посмотрите на инспектор объектов, то заметите, что перед именами некоторых свойств стоит знак плюс. Он указывает на то, что данное свойство представляет из себя набор или другой объект и может быть раскрыто для отображения набора или свойств объекта. Для раскрытия свойства дважды щелкните на его имени в столбце Property или выберите пункт Expand в контекстном меню инспектора объектов. Для закрытия списка дважды щелкните на имени свойства еще раз или выберите в контекстном меню пункт Collapse.

Чтобы увидеть пример набора, выберите форму и затем дважды щелкните на свойстве BorderIcons. Соответствующий пункт списка раскроется и вы увидите четыре булевых члена данного набора. При необходимости можно изменить значение любого из них на противоположное.

Для свойств, представляющих собой объекты (экземпляры классов VCL), существуют две возможности редактирования. Во-первых, можно щелкнуть в столбце Value и затем на появившейся справа кнопке с многоточием (...). Нажатие этой кнопки вызывает соответствующий редактор свойств. Например, щелкните на свойстве Font, а затем на кнопке с многоточием. Откроется диалоговое окно Choose Font, в котором вы сможете выбрать один из шрифтов. Второй способ заключается в раскрытии узлового пункта списка свойств. Появится дополнительный список, в котором вы сможете обычным образом редактировать свойства данного свойства (да, это звучит именно так). Найдите снова свойство Font и дважды щелкните на нем. Появится список свойств TFont и вы сможете изменить Height, Color или Name.

Для редактирования некоторых свойств можно использовать только кнопку с многоточием. Ранее вы использовали компонент Image для отображения значка в окне About программы Multiple. Тогда же вы обнаружили, что свойство Picture этого компонента может быть изменено только через редактор свойств. В данном случае редактором свойств служил редактор изображений C++Builder.

Будьте уверены, каждое свойство «знает», какой редактор ему требуется. По мере знакомства с новыми компонентами и их свойствами вы встретитесь с различными типами редакторов свойств.

Вкладка Events

Вкладка Events содержит список событий, которые может обрабатывать данный компонент. Использование этой вкладки крайне просто. Для создания обработчика события вам нужно дважды щелкнуть в столбце Value рядом с

названием этого события. При этом C++Builder создаст функцию со всеми параметрами, необходимыми для обработки события. На экране появится окно редактора кода с курсором, расположенным внутри обработчика. Вам останется только начать ввод кода. Имя сгенерированной функции составляется из значения свойства Name компонента и имени обрабатываемого события. Например, если у вас есть кнопка OK и обрабатывается событие OnClick, функция получит имя OKClick().

Вы можете разрешить автоматическую генерацию имен обработчиков событий или задавать имена функций вручную. Чтобы указать C++Builder имя функции-обработчика, наберите его в столбце Value рядом с названием события и нажмите Enter. В появившемся окне редактора кода будет содержаться функция с указанным именем.

После создания функции-обработчика события для одного компонента, вы можете использовать ее для любых компонентов, обрабатывающих то же событие. Иногда, например, удобно иметь несколько кнопок, использующих одно событие OnClick. Забегая вперед, приведу также в качестве примера пункт главного меню, пункт контекстного меню и кнопку быстрого доступа, для которых можно использовать общий обработчик события OnClick. Вы оцените эту возможность повторного использования кода, когда приобретете некоторый опыт программирования в C++Builder. Несмотря на то, что вы имеете дело с тремя различными компонентами, они могут совместно использовать общий обработчик OnClick. Столбец Value вкладки Events содержит кнопку для вызова списка обработчиков, совместимых с данным событием. Все, что вы должны сделать — это выбрать нужное имя из списка.

Пример MDI-программы

Чтобы лучше усвоить все то, что говорилось сегодня о проектах и формах, давайте создадим MDI-приложение. Это приложение позволит вам открывать и сохранять графические файлы, содержащие растровые изображения, значки и метафайлы. Перед началом работы нужно наметить план действий. Требуется сделать следующее:

1. Создать форму для главного окна MDI, включающую меню.
2. Написать код для пунктов File | Open и File | Save.
3. Написать код для пунктов Cascade, Tile и Arrange All меню Window.
4. Создать подчиненные формы MDI.
5. Создать диалоговое окно About.
6. Полюбоваться на свою работу.

Бездельничать нам некогда (время — деньги!), поэтому приступим к работе немедленно.

Шаг 1. Создание формы для главного окна

Прежде всего мы создадим форму для главного окна. Свойство FormStyle для главного окна MDI-приложения должно иметь значение fsMDIForm. Мы также должны добавить к приложению меню вместе с диалоговыми окнами File Open и File Save.

1. Запустите C++Builder и выберите в главном меню пункт File | New Application.
2. Измените значение свойства Name главной формы на MainForm.
3. Измените значение свойства Caption на Picture Viewer.
4. Установите свойство Height равным 450, а Width — равным 575.
5. Измените значение FormStyle на fsMDIForm.

Итак, основная часть формы готова. Теперь нужно добавить к ней меню. Поскольку я еще не рассказывал о редакторе меню, мы выберем для создания меню кратчайший путь. Мы используем такую возможность C++Builder, как импорт предопределенных меню.

1. Щелкните на вкладке Standard палитры компонентов, а затем на кнопке MainMenu.
2. Поместите компонент MainMenu в форму. Место расположения не имеет значения, поскольку значок, представляющий меню, просто указывает на его наличие и не отображается во время выполнения программы.
3. Измените значение свойства Name на MainMenu.
4. Дважды щелкните на компоненте MainMenu. На экране появится редактор меню. (Мы рассмотрим его более подробно завтра.)
5. Установите курсор мыши на редактор меню и щелкните правой кнопкой. Выберите в контекстном меню пункт Insert from template... (Вставить из шаблона...). Появится диалоговое окно Insert Template (Вставить шаблон), показанное на рис. 6.15.
6. Выберите пункт MDI Frame Menu и нажмите кнопку OK. Меню появится в редакторе.
7. Закройте редактор меню, щелкнув на кнопке закрытия окна.



Рис. 6.15
Редактор меню с открытым диалоговым окном Insert Template

Теперь мы снова вернемся к главной форме. Заметьте, что в ней появилось меню. Вы можете пощелкать на пунктах верхнего уровня, чтобы просмотреть меню целиком. Не щелкайте пока на подпунктах — мы сделаем это чуть позже. Обратите внимание, что в меню присутствует много лишних пунктов. Пока мы оставим их на своих местах.

Теперь нам нужно приготовить диалоговые окна File Open и File Save.

1. Щелкните на вкладке Dialogs палитры компонентов. Выберите компонент OpenDialog и поместите его в форму. Значок OpenDialog может находиться в любом месте формы.

2. Измените значение свойства Name диалогового окна на OpenDialog.
3. Измените значение свойства Title на Open a Picture for Viewing.
4. Добавьте к форме компонент SaveDialog.
5. Измените значение свойства Name компонента SaveDialog на SaveDialog, а значение свойства Title — на Save a Picture.

Ваша форма должна выглядеть примерно как на рис. 6.16.

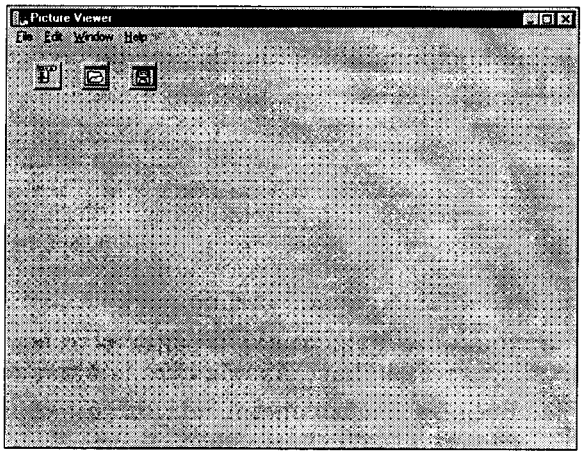


Рис. 6.16
Вид формы на данный момент

Шаг 2. Написание кода для пунктов меню File | Open и File | Save As

Пока все идет хорошо. Давайте теперь напишем код, соответствующий пунктам меню File | Open и File | Save As. C++Builder предоставляет удобный способ написания обработчиков меню, требующий минимальных затрат. Заметьте, мы еще не создавали подчиненных форм MDI, но мы знаем о них достаточно, чтобы написать обработчики меню.

1. Выберите в меню главной формы пункт File | Open. При этом будет создан обработчик данного пункта и на экране появится окно редактора кода.
2. Введите необходимый текст, чтобы обработчик выглядел следующим образом:

```
void __fastcall TMainForm::Open1Click(Tobject *Sender)
{
  if (OpenDialog->Execute())
  {
    TChild* child = new TChild(this);
    child->SetParent(this);
    child->Image->Picture->LoadFromFile(OpenDialog->FileName);
    child->ClientWidth = child->Image->Picture->Width;
    child->ClientHeight = child->Image->Picture->Height;
    child->Caption = OpenDialog->FileName;
    child->Show();
  }
}
```

Этот код прежде всего открывает диалоговое окно File Open и получает имя файла. Если в диалоговом окне была нажата кнопка Cancel, функция не выполняет больше никаких действий. Если же была нажата кнопка OK, создается новый объект TChild (TChild будет именем класса MDI, который мы создадим позже). Файл с изображением передается компоненту Image подчиненной формы; затем рабочая область этой формы подгоняется под размер изображения. В завершение свойству Caption присваивается имя выбранного файла и подчиненное окно отображается на экране.



Помните, мы говорили о необходимости вызывать delete для всех объектов, созданных с помощью оператора new? В приведенном только что коде это правило, на первый взгляд, нарушено. В действительности это не так, потому что VCL берет на себя ответственность за освобождение памяти, выделенной для подчиненных окон MDI. Обратите внимание, что единственным аргументом конструктора TChild является this. Он указывает VCL на то, что владельцем подчиненного окна MDI является главная форма MDI. При уничтожении главной формы (в момент закрытия приложения) будет автоматически освобождена вся память, занятая подчиненными окнами MDI.

3. Нажмите F12, чтобы перейти обратно к форме. Теперь выберите в меню пункт File | Save As. На экране будет отображен обработчик File | Save As.
4. Введите необходимый код, чтобы обработчик пункта File | Save As выглядел следующим образом:

```
void __fastcall TMainForm::SaveAs1Click(Tobject *Sender)
{
  TChild* child = dynamic_cast<TChild*>(ActiveMDIChild);
  if (!child) return;
  if (SaveDialog->Execute())
  {
    child->Image->Picture->SaveToFile(SaveDialog->FileName);
  }
}
```

Как видите, код для пункта File | Save As достаточно прост. В первых двух строках проверяется, активно ли подчиненное окно MDI. Если это так, то отображается диалоговое окно File | Save As. После нажатия кнопки OK изображение сохраняется на диске с помощью метода SaveToFile() класса TPicture.



В приведенном только что коде вы встретились со специальным оператором Си++, dynamic_cast. Этот оператор используется для преобразования указателя на базовый класс в указатель на производный класс. Свойство ActiveMDIChild возвращает указатель на объект TForm. Но в данном случае нам нужен указатель на объект TChild (класс-потомок MDI, произведенный от TForm), чтобы получить доступ к свойству Image подчиненной формы MDI.

Если dynamic_cast не может выполнить приведение типов, он возвращает NULL. Попытка использования указателя NULL приведет к нарушению прав доступа, но отладчик укажет вам ошибочную строку и вы будете точно знать, где возникли проблемы. Это намного лучше, чем использование приведения типов в старой форме, когда ошибка приведения могла повлечь за собой разрушение произвольного участка памяти.

Перед тем, как продолжить, неплохо сохранить проект. Выберите в главном меню пункт File | Save All. Сохраните Unit1 (это имя C++Builder присваивает по умолчанию новому модулю) как ViewMain, а проект — как ViewPict.

Шаг 3. Написание кода для меню Window

Теперь мы создадим код для меню Window. Это не составит большого труда.

1. Вернитесь обратно к форме, нажав F12. Выберите в меню формы пункт Window | Tile.
2. Для этого обработчика вам потребуется ввести всего одну строку кода. Законченный обработчик должен выглядеть следующим образом:

```
void __fastcall TMainForm::Tile1Click(Tobject *Sender)
{
  Tile();
}
```

3. Вернитесь обратно к форме и повторите процесс для пункта Window | Cascade. Законченная функция должна выглядеть следующим образом:

```
void __fastcall TMainForm::Cascade1Click(TObject *Sender)
{
  Cascade();
}
```

4. Выполните аналогичные действия для пункта Window | Arrange All. В тело функции должна быть добавлена единственная строка:

```
ArrangeIcons();
```

На этом разработка главной формы закончена. Мы может перейти к созданию подчиненных форм MDI.

Шаг 4. Создание подчиненной формы MDI

Подчиненные формы MDI могут показаться неожиданно простыми. Фактически, нам вообще не потребуется писать код.

1. Создайте новую форму, нажав на кнопку New Form панели инструментов или выбрав в главном меню пункт File | New Form.
2. Измените значение свойства Name на Child. Свойство Caption можно игнорировать, поскольку мы будем устанавливать его во время выполнения программы.
3. Измените значение свойства FormStyle на fsMDIChild. Это необходимо для того, чтобы форма функционировала как подчиненное окно MDI.

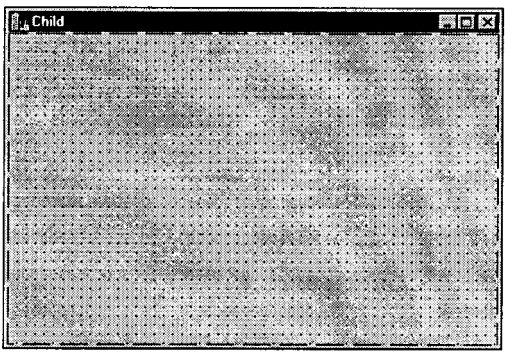
Вот и все, что касается самой формы. Теперь давайте поместим в форму компонент Image, который будет отображать графический файл, выбранный пользователем.

1. Щелкните на вкладке Additional палитры компонентов. Щелкните на кнопке Image и поместите компонент Image в любое место формы.
2. Измените значение свойства Name на Image.
3. Измените значение свойства Stretch на true.

4. Измените значение свойства Align на alClient. Компонент Image будет заполнять всю рабочую область формы.
5. Выберите пункт File | Save и сохраните модуль формы под именем MDIChild.
6. Переключитесь в окно редактора кода (нажмите F12). Щелкните на вкладке ViewMain.cpp. Затем выберите в главном меню пункт File | Include Unit Hdr, отметьте модуль MDIChild и нажмите кнопку OK. Это нужно для того, чтобы компилятор мог распознать ссылку на объект TChild.

Сейчас форма не производит никакого впечатления; она должна выглядеть, как на рис. 6.17.

Рис. 6.17
Подчиненная форма MDI с компонентом Image

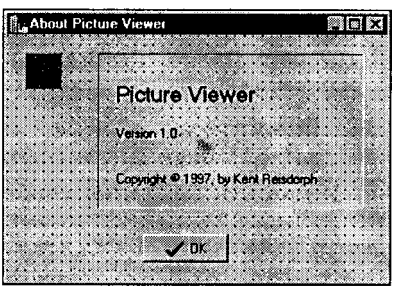


Нам нужно еще создать окно About, но сейчас вам, наверное, не терпится попробовать программу в работе. Нажмите кнопку Run. Спустя некоторое время программа запустится. Вы можете выбрать File | Open и открыть любой графический файл с расширением .bmp, .wmf или .ico. Обратите внимание, что подчиненное окно изменяет свой размер в зависимости от размера изображения. Откройте несколько файлов и попробуйте опции Cascade и Tile в пункте Window. Если хотите, можете сохранить файл под другим именем, используя пункт File | Save As.

Шаг 5. Создание окна About

Сейчас вы знаете о C++Builder достаточно много, чтобы создать окно About самостоятельно. Сделайте его примерно таким, как на рис. 6.18. Если возникнут затруднения, вернитесь назад на несколько страниц и просмотрите еще раз последовательность действий, с помощью которых мы создавали окно About раньше. При желании вы можете оформить это окно по-своему.

Рис. 6.18
Окно About



Создав окно, выполните следующие действия для его вызова из меню:

1. Измените значение свойства Name на AboutBox.
2. Сохраните модуль под именем PVAbout.



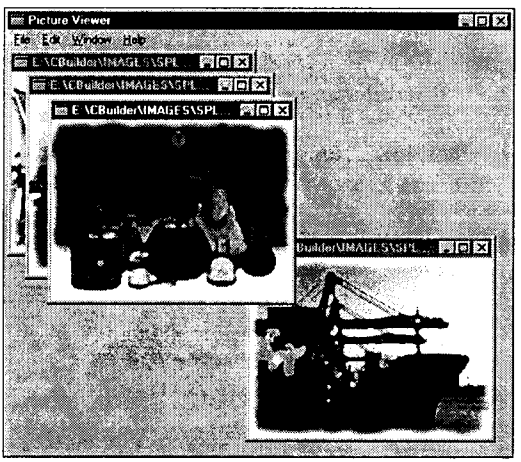
C++Builder полностью поддерживает длинные имена файлов. В этой книге я использую соглашение 8.3 по причинам, связанным с электронной верстkой. В своих приложениях вы можете использовать все преимущества длинных имен.

3. Перейдите к вкладке ViewMain.cpp в окне редактора кода (нажмите F12). Выберите в главном меню пункт File | Include Unit Hdr и включите заголовок PVAbout.
4. Нажмите F12, чтобы перейти обратно к главной форме. Выберите в меню пункт Help | About. Вы окажетесь в окне редактора кода, содержащем обработчик данного пункта меню.
5. Добавьте к тексту обработчика следующую строку:

```
AboutBox->ShowModal();
```

Нажмите кнопку Run и попробуйте вызвать окно About через меню. На рис. 6.19 показана программа Picture Viewer с несколькими подчиненными окнами.

Рис. 6.19
Picture Viewer в действии



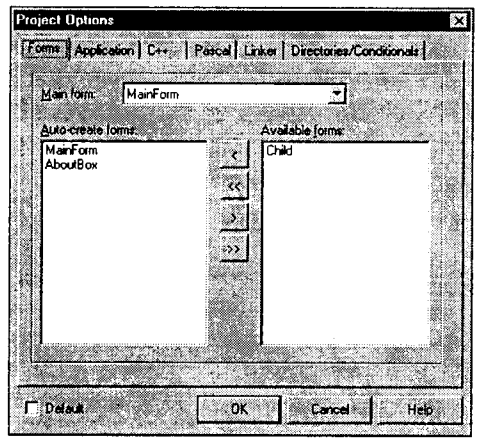
На данном этапе программа уже функционирует, но еще не доведена до нужного уровня по ряду причин. Однако для 30-минутной работы это совсем неплохо! В своем теперешнем виде программа имеет несколько недостатков. Если вы попытаетесь открыть файл, который не содержит графику, то возникнет исключительная ситуация. Мы разберемся с этим позже. В программе также содержатся лишние пункты меню, которые следует убрать. Как это сделать, вы узнаете завтра, когда мы будет подробно изучать редактор меню.

Есть еще одна проблема, которую легко обнаружить и легко устранить. Вы заметили, что после запуска программы сразу отображается пустое подчиненное окно? Дело в том, что приложение C++Builder автоматически создает при запуске все формы, в том числе и подчиненное окно MDI. Мы же

создаем подчиненные окна по мере необходимости, поэтому автоматическое создание в данном случае не требуется.

К счастью, удалить подчиненное окно MDI из списка автоматически создаваемых объектов легко. Выберите в главном меню пункт Project | Options. Появится диалоговое окно Project Options. При необходимости щелкните на вкладке Forms. Вы увидите список автоматически создаваемых форм. Щелкните на форме Child, а затем на кнопке >. Форма будет удалена из списка Auto-create forms (Автоматически создаваемые формы) и помещена в список Available forms (Доступные формы). На рис. 6.20 показано диалоговое окно Project Options после удаления формы Child из списка автоматически создаваемых форм.

Рис. 6.20
Диалоговое окно Project Options



Теперь снова запустите программу. Пустое подчиненное окно больше не будет отображаться.



Если вы удалили форму из списка автоматически создаваемых форм, то должны специально создать ее перед использованием. Если вы не создадите форму, указатель на нее останется неинициализированным, и при попытке его использования возможно нарушение защиты или сбои в программе.

Заключение

Интегрированная среда C++Builder поначалу может казаться непреодолимо сложной. Но если вы поработаете с ней какое-то время, она уже не будет выглядеть столь устрашающе. Сегодня вы научились использовать проекты для создания исполняемых файлов. Вы также узнали больше о формах. Вы увидели, как C++Builder работает с диалоговыми и другими порождаемыми окнами. После этого вы создали программу, которая делает нечто действительно интересное. В завершение этого дня вы ближе познакомились с инспектором объектов. Завтра мы перейдем к изучению редактора форм и редактора меню.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Панель инструментов C++Builder не содержит кнопок для функций, которыми я пользуюсь чаще всего. Могу ли я изменить панель инструментов?

Безусловно. Панель инструментов является полностью настраиваемой. Вы можете добавлять или удалять кнопки по своему усмотрению.

Я поместил в форму несколько компонентов Label и попытался выделить их все. Вместо этого получился еще один компонент Label большего размера. Что я сделал неправильно?

Вы забыли выключить режим размножения компонентов. После размещения нескольких компонентов подряд нужно нажать кнопку со стрелкой на палитре компонентов, чтобы выключить этот режим.

На моей панели размещено несколько компонентов. Я попытался выделить их для перемещения, но в результате сдвинулась вся панель. Что я должен сделать для выделения группы компонентов на панели?

Либо нажимайте Shift, щелкая на компоненте, либо нажмите клавишу Ctrl и растяните вокруг нужных компонентов рамку.

Можно ли написать, откомпилировать и запустить простую программу на Си++, не создавая проект?

Нет. Проект требуется для создания исполняемого файла. Он обеспечивает подключение необходимых библиотечных модулей.

Для чего нужны библиотечные файлы?

Существует стандартный набор функций, использующихся различными программами на Си++. Примером могут служить функции для работы со строками, рассмотренные в первый день, «Первое знакомство». Если ваша программа вызывает какую-либо функцию Си++, эта функция должна быть включена в исполняемый файл. Стандартные функции содержатся в библиотечном файле (.LIB). Компоновщик копирует функцию из библиотечного файла в исполняемый файл вашей программы. То же самое справедливо и для методов VCL.

Почему в C++Builder исходный файл называется unit?

C++Builder был создан на основе Borland Delphi, а в основе Delphi лежит Паскаль. В Паскале для исходных файлов используется термин unit.

Что необходимо сделать для создания MDI-приложения?

Свойство FormStyle для главной формы должно иметь значение fsMDIForm, а для подчиненной формы — fsMDIChild.

Чем

Реально ничем. Форма для диалогового окна может иметь специальную рамку, содержать кнопки OK, Cancel и Help и не иметь кнопок для сворачивания и разворачивания окна, но принципиально она не отличается от других форм, в том числе и от формы для порожденного окна.

Можно ли проверить программу на наличие ошибок, не запуская ее?

Да. Выберите в главном меню пункт Project | Syntax Check. C++Builder откомпилирует все модули, изменившиеся с момента последней компиляции, и сообщит о встретившихся ошибках.

Можно ли с помощью C++Builder создавать OWL- или MFC-приложения?

Да. Поскольку сила C++Builder заключена в использовании VCL, вы, скорее всего, не станете разрабатывать сложные приложения на основе OWL или MFC, но при желании это можно сделать.

Вопросы

1. Как вызвать диалоговое окно редактора панели инструментов?
2. Как добавить кнопки в панель инструментов, используя диалоговое окно редактора?
3. Как можно удалить кнопки из панели инструментов?
4. Как проще всего поместить в форму несколько компонентов одного типа?
5. Как проще всего разместить компонент в центре формы?
6. Перечислите типы файлов, необходимых для построения приложения в C++Builder.
7. Какой метод VCL используется для отображения модальной формы?
8. Какой метод VCL используется для отображения немодальной формы?
9. Как можно связать событие с предварительно определенным обработчиком?
10. Как перебрать возможные значения свойства, используя инспектор объектов?

Упражнения

1. Удалите кнопки Pause, Step Over и Trace из панели инструментов. Добавьте вместо них кнопки Cut, Copy и Paste.
2. Восстановите вид панели инструментов, который она имела по умолчанию.
3. Просмотрите каждую вкладку палитры компонентов. Поместите в форму те компоненты, которые вас заинтересуют, и поэкспериментируйте с ними.
4. Создайте новый каталог на жестком диске. Создайте новое приложение C++Builder. Добавьте к проекту три новых формы (можно оставить их пустыми). Сохраните проект в созданном каталоге и запустите программу. Закройте программу и исследуйте каталог, в котором был сохранен проект. Сравните файлы, которые вы там найдете, с типами файлов, приведенными в таблице 6.1.

5. Запустите программу Picture Viewer, созданную раньше. Откройте несколько графических файлов. Подвигайте подчиненные окна в пределах окна программы. Попробуйте переместить подчиненное окно за пределы главного окна. Что при этом произошло?
6. Сверните все окна, не закрывая программу Picture Viewer. Переместите свернутые окна в разные места экрана и затем выберите в меню пункт Window | Arrange All.
7. Создайте новое приложение. Поместите в форму несколько компонентов. Последовательно выделяя компоненты, просмотрите их свойства через инспектор объектов.
8. Создайте пустую форму. Дважды щелкните в столбце Value рядом со свойством Color для вызова диалогового окна Color. Выберите цвет и нажмите кнопку OK.
9. Отдохните немного. Завтра нам предстоит большой день.

Работа с редактором форм и редактором меню



Кент Рейсдорф

Как вы теперь знаете, в основу C++Builder положена концепция форм, которая позволяет в максимальной степени реализовать возможности визуального программирования. В этой главе вы изучите:

- Редактор форм
- Редактор меню

Чтобы проиллюстрировать использование редактора форм, мы создадим приложение, примерно аналогичное программе Notepad из Windows. При этом вы приобретете ценный опыт работы с редактором форм. В конце этой главе вы подробно изучите редактор меню.

Работа с редактором форм

Редактор форм C++Builder представляет собой мощный инструмент визуального программирования. Он позволяет вам размещать, выделять, передвигать, выравнивать компоненты, изменять их размеры и делать многое другое. Редактор форм также позволяет вам изменять размеры и положение самой формы, добавлять меню и создавать специализированные диалоговые окна — короче, все, что требуется для создания интерфейса пользователя в типичной Windows-программе.

В последующих разделах мы рассмотрим каждую из возможностей, предоставляемых редактором форм. По мере прочтения я призываю вас останавливаться и экспериментировать со всем, что вызывает интерес. Иногда несколько минут подобных упражнений могут научить вас использовать то, о чем вы долгое время имели лишь теоретическое представление.

Контекстное меню редактора форм

Редактор форм, как и большинство других окон C++Builder, имеет свое контекстное меню. В таблице 7.1 перечислены пункты контекстного меню редактора форм и дано их описание.

Таблица 7.1. Пункты контекстного меню редактора форм

Пункт	Описание
Align To Grid	Привязка выделенных компонентов к сетке редактора форм.
Bring To Front	Перемещение выделенных компонентов на передний план.
Send To Back	Перемещение выделенных компонентов на задний план.
Revert to Inherited	При работе с формой, выбранной из репозитория объектов, этот пункт меню возвращает ей первоначальный вид. (Работа с формами, содержащимися в репозитории, рассматривается на девятый день, «Создание приложений в C++Builder».)
Align	Отображение диалогового окна Alignment (Выравнивание).
Size	Отображение диалогового окна Size (Размер).
Scale	Отображение диалогового окна Scale (Масштаб).
Tab Order	Отображение диалогового окна Edit Tab Order (Изменить порядок переключения клавишей Tab).
Creation Order	Отображение диалогового окна Creation Order (Порядок создания).
Add to Repository	Помещение формы в репозиторий объектов. Формы, созданные пользователем, могут быть сохранены для последующего использования. (Репозиторий объектов рассматривается на девятый день.)
View as Text	Отображение в окне редактора кода описания формы в текстовом виде. Вы можете редактировать текстовый вариант формы произвольным образом. Для возврата к форме выберите в контекстном меню редактора кода пункт View as Form. Для переключения между режимами View as Text и View as Form можно также использовать сочетание клавиш Alt+F12.



Для каждой из разрабатываемых форм C++Builder создает файл формы (.DFM) и помещает его в каталог, содержащий проект. Файл формы представляет собой двоичный файл ресурсов, который не может быть непосредственно прочитан человеком. Когда вы выбираете в контекстном меню пункт View as Text, C++Builder преобразует этот файл в текст, доступный для восприятия. При обратном переключении (View as Form) C++Builder заново создает файл формы, в котором учтены внесенные изменения.

Большая часть пунктов контекстного меню рассматривается в последующих разделах. Остальные будут описаны в тех главах, где мы будем изучать соответствующие возможности C++Builder.

Размещение компонентов

Перенос компонента в форму выполняется крайне просто. Вы выбираете компонент из палитры компонентов и щелкаете мышью в том месте формы, где он должен появиться. При этом верхний левый угол компонента будет расположен там, где находился курсор мыши в момент щелчка. Обратите внимание, что после щелчка на кнопке палитры компонентов она остается нажатой. Кнопка возвращается в исходное состояние только после щелчка на форме, показывая тем самым, что операция завершена.



Как вы знаете из предыдущей главы, для размещения в форме нескольких копий компонента нужно нажать и удерживать клавишу Shift при выборе кнопки в палитре компонентов. После этого каждый щелчок на форме приводит к добавлению нового компонента. Для выхода из этого режима нажмите кнопку со стрелкой в палитре компонентов.

Размеры большинства компонентов могут быть изменены. Вы можете сначала поместить компонент в форму, а затем увеличить или уменьшить его. Можно также изменить размер компонента непосредственно при размещении. Для этого нажмите кнопку мыши в том месте, где должен находиться верхний левый угол компонента, и перемещением курсора установите нужный размер. После отпускания кнопки компонент примет заданный размер.



Не все компоненты допускают изменение размера подобным образом. Невизуальные компоненты, например, представлены значками. Хотя вы можете попытаться их растянуть или сжать, новый размер будет игнорирован. Другой пример — однострочные компоненты редактирования, которые допускают только изменение ширины. Высота этих компонентов по умолчанию равна высоте стандартных однострочных элементов редактирования и изменению не подлежит.



Если вы решили не менять размер компонента, размещая его в форме, нажмите клавишу Esc перед отпусканием кнопки мыши. Кнопка в палитре компонентов при этом останется нажатой, и вам потребуется щелкнуть на кнопке со стрелкой для возврата в режим выбора компонентов.

Помещение компонентов в форму настолько просто, что мы не будет уделять этому много времени. Вы приобрели некоторый опыт вчера, поэтому давайте перейдем к другим вещам.

Сетка редактора форм

Редактор форм содержит специальную сетку, которая облегчает разработку форм. По умолчанию эта сетка отображается на экране. Размер ячеек изначально составляет 8 пикселов по горизонтали и 8 по вертикали. В режиме отображения сетки на пересечении ее линий ставятся точки. Компоненты, помещаемые в форму, привязываются к ближайшей точке сетки. *Привязка (snap)* означает, что верхний левый угол компонента автоматически перемещается в ближайшую к нему точку сетки. Это помогает выравнивать группы компонентов по верхнему, нижнему, правому или левому краю. Когда режим Snap to Grid включен, вам достаточно установить курсор мыши примерно в нужном месте и редактор форм автоматически поместит верхний левый угол компонента в ближайшую точку сетки. Это экономит время, позволяя избежать длительной подгонки размера или местоположения компонента.

Параметры сетки могут быть изменены через вкладку Preferences диалогового окна Environment Options. (Я буду подробно рассматривать опции окружения на десятый день, «Подробнее о проектах».) Здесь вы можете изменить размер сетки или выключить режим привязки. Вы можете также

включить или выключить режим отображения сетки. Когда отображение сетки выключено, она остается активной (подразумевается, что режим Snap to Grid включен), но точки, обозначающие узлы сетки, на форме не показываются.

Выделение компонентов

После помещения компонента в форму вам часто нужно выделить его, чтобы изменить каким-либо образом. Выделение требуется для выполнения следующих действий:

- ☐ Перемещения компонента
- ☐ Изменения свойств компонента
- ☐ Выравнивания компонента
- ☐ Изменения размера компонента
- ☐ Копирования или перемещения компонента в буфер обмена
- ☐ Перемещения компонента на передний или задний план
- ☐ Удаления компонента

Выделение отдельных компонентов

Для выделения одиночного компонента просто щелкните на нем. При этом вокруг компонента появятся восемь черных маркеров, указывающих на то, что он выбран. На рис. 7.1 показана форма с выделенным компонентом Button.

Свойства и события выделенного компонента отображаются в окне инспектора объектов. Для отмены выделения щелкните на свободном участке формы. Можно также щелкнуть на самом компоненте, нажимая при этом клавишу Shift. (Сочетание Shift+щелчок будет описано в следующем разделе.)

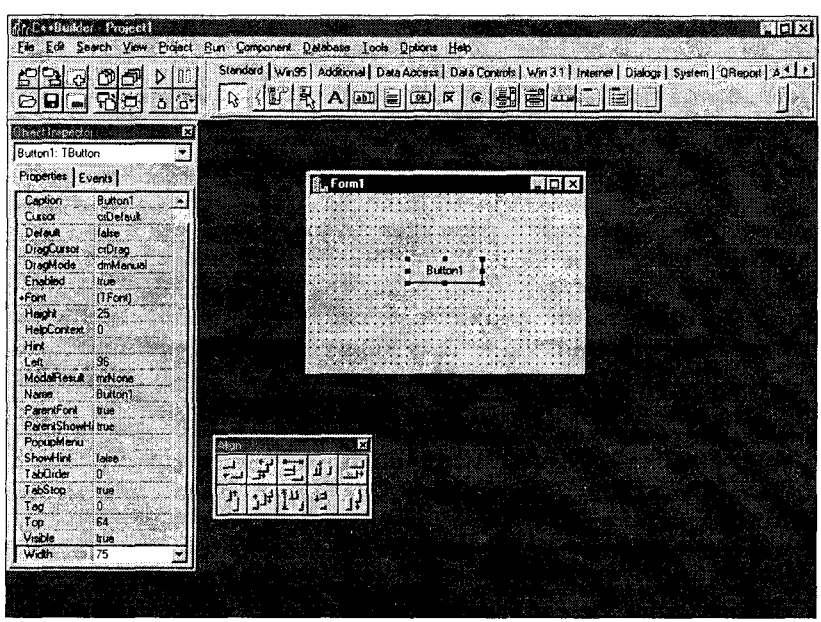


Рис. 7.1 Форма с выделенным компонентом Button



С каждым компонентом по умолчанию связан обработчик определенного события. Именно этот обработчик отображается в окне редактора кода, когда вы дважды щелкаете на компоненте. В большинстве случаев по умолчанию обрабатывается событие OnClick. Вообще говоря, что именно произойдет при двойном щелчке на компоненте, зависит от реализации этого компонента. Например, в случае компонента Image двойной щелчок приводит к открытию диалогового окна редактора изображений.

Выделение группы компонентов

Вы можете выделить несколько компонентов одновременно и работать с ними как с группой. Для группового выделения можно использовать один из трех способов:

- Shift+щелчок с использованием клавиатуры и мыши
- Нажать кнопку мыши и заключить компоненты в рамку
- Выбрать в главном меню пункт Edit | Select All.

Пункт Edit | Select All используется для выделения всех компонентов формы.

Выделение компонентов с помощью сочетания Shift+щелчок

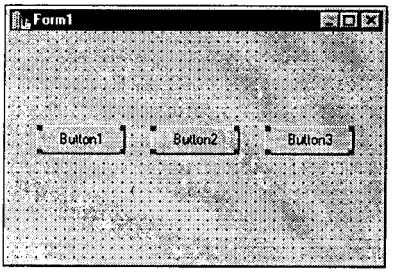
Для использования Shift+щелчок выделите сначала один компонент. Затем, удерживая нажатой клавишу Shift, щелкните на следующем компоненте, который вы хотите объединить в одну группу с первым. Каждый выделенный таким образом компонент окружается четырьмя серыми маркерами, показывающими его принадлежность к группе.

Чтобы удалить компонент из группы, щелкните на нем еще раз, продолжая удерживать клавишу Shift. Таким образом, сочетание Shift+щелчок действует как переключатель.

Чтобы увидеть, как все это работает, создайте новую форму и выполните следующие действия:

1. Поместите в любом месте формы три кнопки. Они будут автоматически названы Button1, Button2 и Button3.
2. Щелкните на кнопке Button1. Вокруг нее появятся черные маркеры.
3. Нажмите и удерживайте клавишу Shift. Щелкните на кнопке Button2. Она будет добавлена к группе. Обе кнопки теперь будут окружены серыми маркерами.
4. Щелкните на кнопке Button3, продолжая удерживать клавишу Shift. Теперь в группу будут включены все три кнопки.
5. Снова щелкните на кнопке Button2 (клавиша Shift остается нажатой). Она будет удалена из группы (серые маркеры исчезнут), но кнопки Button1 и Button3 по-прежнему будут объединены.
6. Щелкните на кнопке Button1, не отпуская Shift. Теперь будет выделена только кнопка Button3. Серые маркеры заменятся на черные.
7. Щелкните на кнопках Button1 и Button2. Все три кнопки снова будут объединены в группу.

Рис. 7.2
Форма с тремя выделенными кнопками



На рис. 7.2 показана форма после окончания всех действий. Имейте в виду, что ваши кнопки могли находиться в других местах.

Держите эту форму под рукой, потому что она потребуется нам для следующего упражнения.



Если вы щелкнете на компоненте, который входит в группу, ничего не произойдет. Чтобы выделить компонент, уже являющийся членом группы, нужно сначала щелкнуть на свободном участке формы или нажать клавишу Esc для отмены группового выделения. Затем можно выделить компонент обычным образом, щелкнув на нем мышью.

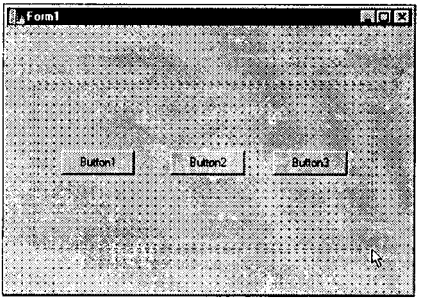
Групповое выделение мышью

Вы можете выделить несколько компонентов, заключив их в ограничивающую рамку. *Ограничивающая рамка (bounding rectangle)* представляет собой серый пунктирный прямоугольник, изменяющий размер при перемещении курсора мыши. На самом деле вам не обязательно полностью заключать компоненты в рамку. Достаточно коснуться рамкой компонента, и он будет включен в группу.

Проследите, чтобы перед началом выделения курсор мыши был помещен на свободный участок формы, а не на компонент. Нажмите левую кнопку мыши и начинайте перемещать курсор. Вы увидите растягивающуюся рамку. Полностью или частично окружите рамкой те компоненты, которые должны входить в группу и отпустите кнопку мыши. Компоненты, которые оказались внутри рамки или касались ее, окажутся выделены.

Для добавления компонентов в созданную группу или удаления их из группы можно использовать технику Shift+щелчок, описанную в предыдущем разделе. Таким образом вы сможете, например, выделить все компоненты в одном из участков формы, за исключением одного.

Рис. 7.3
Выделение компонентов мышью



Вернитесь к форме с тремя кнопками, созданной ранее (если вы уже закрыли ее, создайте новую и поместите в нее три кнопки). Окружите компоненты рамкой, растягивая ее из верхнего левого угла вправо и вниз. Отпустите кнопку мыши, и кнопки окажутся выделены. На рис. 7.3 показаны форма и ограничивающая рамка.



Для объединения групп компонентов, расположенных в разных местах, можно использовать клавишу Shift. Если, например, у вас есть две группы компонентов, расположенные на разных участках формы, растяните рамку вокруг первой группы, затем нажмите и удерживайте клавишу Shift и растяните рамку вокруг второй группы. В результате будут выделены компоненты обеих групп.



Для выделения компонентов можно растягивать рамку в любом направлении, а не только вправо и вниз.

Групповое выделение: компоненты внутри компонентов

С размещением одних компонентов внутри других вы будете сталкиваться довольно часто. Например, компонент Panel обычно используется как основа для других компонентов. Таким путем создаются панели инструментов. Для выделения группы компонентов на панели нужно нажать клавишу Ctrl и заключить их в рамку. (Попробуйте сделать это, не нажимая Ctrl, и посмотрите, что получится.) Вы можете использовать также комбинацию Ctrl+Shift+рамка. (Мне кажется, что разработчики из Borland могли бы подумать об использовании клавиши Alt.)

Для тренировки создайте пустую формы и сделайте следующее:

1. Выберите компонент Panel и поместите его в форму, растянув при этом как можно больше.
2. Выберите компонент Button и поместите на панель шесть кнопок. Ваша форма должна выглядеть как на рис. 7.4.
3. Попробуйте растянуть ограничивающую рамку вокруг компонентов Button1, Button2 и Button3. Вы обнаружите, что это приводит к перемещению панели, а вовсе не к тому результату, который вы ожидали. Передвиньте панель на прежнее место.
4. Нажмите клавишу Ctrl и растяните рамку вокруг компонентов Button1, Button2 и Button3. На этот раз кнопки будут выделены.

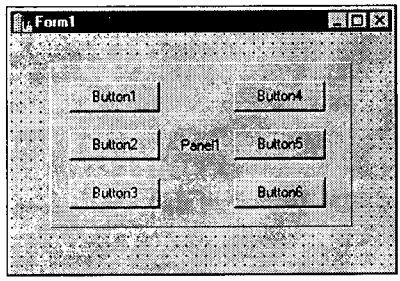


Рис. 7.4
Форма с панелью и шестью кнопками

5. Нажмите одновременно клавиши Ctrl и Shift и растяните рамку вокруг кнопок Button5 и Button6. Теперь окажутся выделенными все компоненты, кроме Button4.

Использование клавиши Ctrl является единственным способом выделения компонентов, расположенных внутри других компонентов, если вы работаете с рамкой. Наряду с этим можно использовать сочетание Shift+щелчок, как и при выделении компонентов в форме.

Перемещение компонентов

Перемещение компонентов является широко распространенным и довольно простым действием. Чтобы передвинуть отдельный компонент, установите на него курсор мыши и начинайте перетаскивание. При этом белый прямоугольник, обозначающий компонент, будет двигаться вместе с курсором. Когда прямоугольник окажется в нужном месте, отпустите кнопку мыши, и компонент займет новое положение.



Когда вы перемещаете элементы управления, используя технику drag-and-drop, свойства Left и Top автоматически обновляются.



Перемещение компонентов с помощью drag-and-drop является самым простым, но не самым точным способом. Если вам нужно разместить компонент в строго определенном месте формы, измените вручную свойства Left и Top. Можно также использовать различные режимы выравнивания, которые будут рассмотрены позже в разделе «Выравнивание компонентов».

Если у вас включен режим Snap to Grid, белый прямоугольник при перемещении будет привязываться к ближайшим узлам сетки.



Если во время перемещения компонента вы изменили свое решение, нажмите клавишу Esc перед тем, как отпустить кнопку мыши. Это приведет к отмене операции и возврату компонента в исходное положение.

Перемещение группы компонентов выполняется аналогичным образом. Установите курсор мыши на любой компонент в группе и начинайте перетаскивание. При этом каждый компонент будет представлен белым прямоугольником, что позволяет определить расположение группы после отпускания кнопки мыши.



Вы не можете перемещать группу компонентов, созданных на основе различных базовых классов. Пусть, например, вы выделили компонент Button в главной форме и компонент SpeedButton на панели. Поскольку эти компоненты произведены от разных классов, их перемещение в составе группы окажется невозможным.



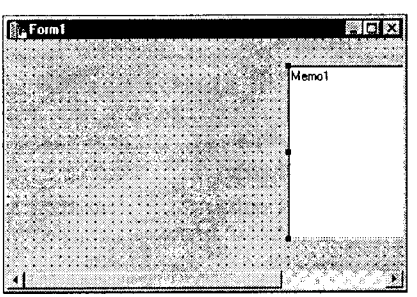
Положение выделенного компонента можно слегка подкорректировать при помощи клавиш со стрелками на клавиатуре, удерживая при этом нажатой клавишу Ctrl. Каждое нажатие будет перемещать компонент на один пиксел в соответствующем направлении. Таким способом можно передвигать как отдельные компоненты, так и группы. Привязка к сетке при этом не работает.
Передвинутый таким способом компонент больше не привязан к узлу сетки, а смещен на некоторое расстояние. Если теперь перемещать компонент обычным способом, используя мышь, это смещение относительно узлов сетки будет сохраняться.



Если вы сдвинули компонент клавишами со стрелками и хотите восстановить его привязку к сетке, выберите пункт Edit | Align to Grid в главном меню или пункт Align to Grid в контекстном меню. При этот верхний левый угол компонента будет совмещен с ближайшим узлом сетки.

Компоненты не могут быть перемещены за пределы породившего их компонента. Если вы попробуете сдвинуть компонент за пределы формы вверх или влево, он будет обрезан на границе формы. Однако выход за правый или нижний край приведет к отображению линеек прокрутки, позволяющих просматривать остальную часть формы. Свойства формы Width и Height при этом не изменяются. Если вы вернете компонент обратно в пределы видимой части формы, линейки прокрутки исчезнут. Такое поведение установлено по умолчанию и сохраняется до тех пор, пока вы не измените значение свойства формы AutoScroll на false. На рис. 7.5 показан компонент Memo, частично сдвинутый за правый край формы. Обратите внимание на линейку прокрутки, появившуюся снизу.

Рис. 7.5
Вид формы при включенном режиме AutoScroll



Защита компонентов от перемещения и изменения размера

Компоненты могут быть зафиксированы на своем месте. Это полезно делать на заключительном этапе разработки формы, чтобы предотвратить их случайное смещение. Выберите в главном меню пункт Edit | Lock Control. После этого положение и размер компонента изменить будет невозможно. Маркеры, окружающие зафиксированный компонент, примут вид серых квадратов с черной рамкой. Для отмены фиксации выберите пункт Edit | Lock Control еще раз.

Упорядочивание компонентов и работа с буфером обмена

Вы можете наложить компоненты друг на друга для достижения определенного визуального эффекта. Например, можно создать эффект тени, разместив белый прямоугольник над черным (речь идет о компонентах Shape). Очевидно, что тень не может располагаться сверху, поэтому вам необходимо каким-то образом изменять порядок расположения компонентов. Давайте сделаем простое упражнение. По ходу дела вы также увидите, как копировать и вставлять компоненты, используя буфер обмена. Начнем, как всегда, с пустой формы.

1. Щелкните на вкладке Additional палитры компонентов и выберите компонент Shape. Поместите его в форму. Вы увидите белый прямоугольник.
2. При желании измените размер прямоугольника (мой имел размер 209 на 129 пикселов).
3. Убедитесь, что компонент Shape выделен. Выберите в главном меню пункт Edit | Copy.
4. Выберите в главном меню пункт Edit | Paste. Копия будет размещена ниже и правее исходного компонента, как раз в том месте, где нам нужно.



Вставленный в результате операции Paste компонент будет выделен.

5. Дважды щелкните на свойстве Brush и измените значение свойства Color, расположенного под Brush, на clBlack. Новый прямоугольник станет черным, но он сейчас располагается над белым. Так быть не должно!
6. Щелкните правой кнопкой мыши и выберите в контекстном меню пункт Send to Back (можно также выбрать Edit | Send to Back в главном меню). Черный прямоугольник расположится под белым. Теперь у вас есть окно с тенью. (С таким же успехом можно было щелкнуть на белом прямоугольнике и использовать операцию Bring to Front, чтобы разместить его сверху.)

Это упражнение демонстрирует две возможности редактора форм. Вы увидели, как изменять порядок расположения компонентов и как использовать операции Copy и Paste для копирования. При копировании и вставке свойства компонента полностью сохраняются. Каждый следующий компонент, вставленный из буфера обмена, размещается правее и ниже предыдущего.



Если перед вставкой был выделен компонент, служащий контейнером для других компонентов, компонент из буфера обмена будет помещен в этот контейнер. Например, вы хотите переместить на панель кнопку из главной формы. Для этого нужно сначала выделить кнопку и выбрать в главном меню пункт Edit | Cut, чтобы удалить ее в буфер обмена. Затем нужно выделить панель и вставить кнопку, используя пункт главного меню Edit | Paste.

Детально объяснять операцию Cut нет необходимости. Когда вы удаляете компонент, он исчезает из формы и размещается в буфере обмена. Затем вы можете вставить компонент из буфера в форму или другой компонент (например, Panel).

Изменение размера компонентов

Некоторые компоненты при помещении в форму принимают размер по умолчанию. Хорошим примером являются кнопки. Стандартная кнопка имеет высоту 25 пикселов и ширину 75 пикселов. Во многих случаях этот размер является наиболее подходящим. Однако для некоторых компонентов размер по умолчанию редко совпадает с требуемым. Например, компонент Memo практически всегда приходится подгонять под размеры конкретной формы, с которой вы работаете.

Изменение размера с помощью мыши

Выделенный компонент окружен восемью черными маркерами. Когда вы установите курсор мыши на один из этих маркеров, он примет форму черной двойной стрелки. После этого можно потянуть за маркер, нажав кнопку мыши. Как именно будет изменяться размер компонента, зависит от того, какой из маркеров вы захватили.

Маркеры, расположенные посередине компонента сверху и снизу, служат для изменения его высоты. Правый и левый центральные маркеры могут быть использованы для изменения размера по горизонтали. Если вы захватите один из маркеров по углам компонента, можно будет изменить высоту и ширину одновременно. Как и при перемещении, во время изменения размера компонент обозначается белым прямоугольником. Растянув или сжав прямоугольник нужным образом, отпустите кнопку мыши, и компонент примет новый размер. На рис. 7.6 показан компонент Memo, растягиваемый с помощью мыши; на рис. 7.7 показана форма после завершения операции.

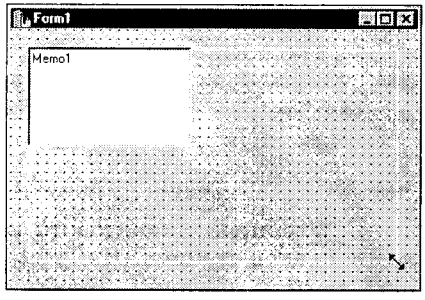


Рис. 7.6
Изменение размера компонента Memo

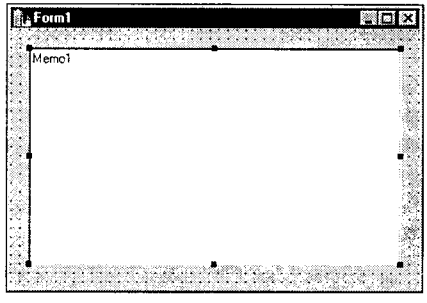


Рис. 7.7
Форма после завершения операции



Изменение размера возможно только для визуальных компонентов. Невизуальные компоненты представляются значками, размер которых фиксирован. При выделении таких компонентов они также окружаются маркерами, но при попытке захватить маркер и потянуть за него результат будет игнорирован.

Размер группы компонентов нельзя изменить с помощью мыши. При выделении более одного компонента черные маркеры заменяются серыми, которые служат только для индикации выделения.



Для одновременного изменения размера всех компонентов группы, измените значения свойств Width или Height в окне инспектора объектов или через диалоговое окно Size, рассмотренное ниже.

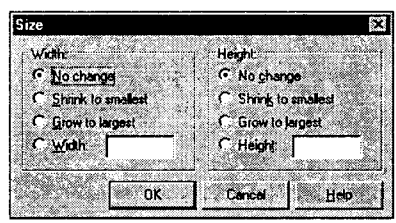


Для изменения размера одного или нескольких компонентов с шагом в один пиксел, используйте клавиши со стрелками на клавиатуре, удерживая при этом нажатой клавишу Shift. Стрелки вверх и вниз изменяют размер по вертикали, а стрелки вправо и влево – размер по горизонтали. Эти операции влияют только на свойства Width и Height. Свойства Top и Left не изменяются.

Изменение размера с помощью диалогового окна Size

Диалоговое окно Size предоставляет альтернативный путь изменения размера компонентов. Вы можете вызвать это окно через главное меню, выбрав пункт Edit | Size. Диалоговое окно Size показано на рис. 7.8.

Рис. 7.8
Диалоговое окно Size



С помощью диалогового окна можно изменить ширину и высоту всех компонентов группы. Например, пусть у вас есть шесть компонентов, имеющих разную ширину. Чтобы улучшить внешний вид формы, имеет смысл привести все компоненты к одному размеру. Для этого нужно сначала использовать групповое выделение, а затем вызвать диалоговое окно Size. Теперь можно выбрать пункт Shrink to Smallest (Сжать до наименьшего) или Grow to Largest (Растянуть до наибольшего) в столбце Width. При этом размер всех компонентов будет подогнан под размер соответственно наименьшего или наибольшего компонента в группе. Вы можете также ввести точное значение ширины в окне Width, оставив при этом для высоты установку No Change (Без изменений). После нажатия кнопки OK компоненты будут иметь одинаковую ширину.

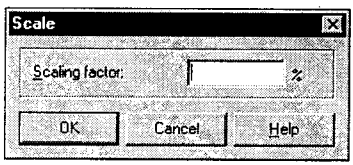


Диалоговое окно Size может быть вызвано также через контекстное меню редактора форм.

Изменение размера с помощью диалогового окна Scale

Другим инструментом для изменения размера является диалоговое окно Scale (Масштаб), показанное на рис. 7.9. Оно позволяет задать степень масштабирования в процентах. Чтобы увеличить компоненты в два раза, наберите в окне Scaling Factor число 200. Для двукратного уменьшения размера наберите число 50. Диалоговое окно Scale удобно для быстрого изменения размера всех компонентов формы. Оно вызывается через пункт Edit | Scale главного меню или пункт Scale контекстного меню редактора форм.

Рис. 7.9
Диалоговое окно Scale



Для изменения размера и расположения компонентов можно также использовать различные опции выравнивания. Давайте перейдем к их рассмотрению.



Не забывайте, что для перемещения компонента всегда можно изменить значения его свойств Left и Top, а для изменения размера — Width и Height.

Выравнивание компонентов

Независимо от того, включен ли режим Snap to Grid, иногда требуется выравнивать компоненты после помещения их в форму. Под выравниванием может подразумеваться выстраивание компонентов вдоль одной линии, их центрирование относительно формы или расположение с определенным шагом. Существуют два различных способа выравнивания компонентов:

- Использование палитры выравнивания (Alignment Palette) или диалогового окна Alignment
- Изменение значения свойства Align



Некоторые компоненты имеют свойство Alignment. Оно определяет способ выравнивания текста внутри компонента (по центру, по левому или правому краю) и не имеет отношения к расположению компонента относительно формы.

Палитра выравнивания и диалоговое окно Alignment

Зачастую необходимо изменить расположение или размер компонентов относительно формы или других компонентов. Эта задача решается с помощью кнопок палитры выравнивания. Диалоговое окно Alignment выполняет

те же действия, но они задаются в другом формате. Для отображения палитры выравнивания выберите в главном меню пункт View | Alignment Palette. На рис. 7.10 показана палитра выравнивания с описанием каждой кнопки.

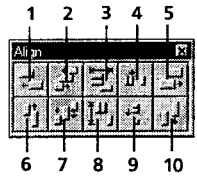


Рис. 7.10
Палитра выравнивания:
1 – выровнять левые края; **2** – выровнять центры по вертикали;
3 – отцентрировать в окне по горизонтали; **4** – установить равный шаг по горизонтали; **5** – выровнять правые края; **6** – выровнять верхние края;
7 – выровнять центры по вертикали; **8** – отцентрировать в окне по вертикали;
9 – установить равный шаг по вертикали; **10** – выровнять нижние края

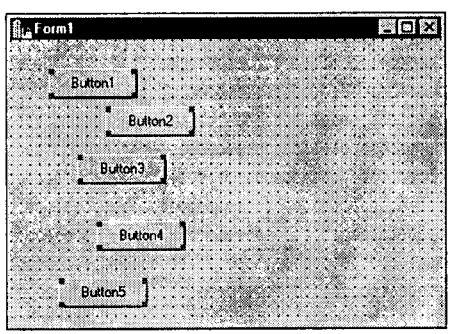


Использование палитры выравнивания сильно упрощает работу. Не тратьте зря время, пытаясь вручную выстроить компоненты вдоль одной линии. Поместите их в форму и сразу воспользуйтесь палитрой выравнивания.

Кнопка Align Left Edges (Выровнять левые края) служит для выравнивания компонентов по левому краю. Создайте пустую форму и сделайте следующее:

1. Поместите в форму пять кнопок, расположив их по вертикали и не заботясь о положении левых краев.
2. Выделите кнопки, окружив их пунктирной рамкой. Форма должна выглядеть примерно как на рис. 7.11.
3. Выберите в главном меню пункт View | Alignnment Palette. На экране появится палитра выравнивания. Передвиньте ее при необходимости так, чтобы она не закрывала форму.
4. Щелкните на кнопке Align Left Edges, чтобы выровнять кнопки в форме.

Рис. 7.11
Форма со случайно расположенными кнопками



Видите, как просто? Пока кнопки выделены, давайте попробуем выровнять их по-другому. Опция Space Equally Vertically (Установить равный шаг по вертикали) позволит нам расположить кнопки равномерно. Щелкните на кнопке Space Equally Vertically палитры выравнивания. Форма должна принять вид, как на рис. 7.12.

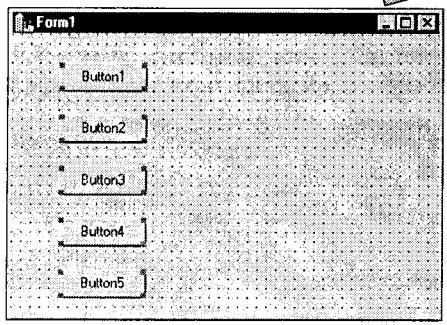


Рис. 7.12
Форма с выровненными и равномерно расположенными кнопками



Опция Space Equally Vertically размещает компоненты с равными интервалами между первым (верхним) и последним (нижним) компонентами столбца. Перед выравниванием убедитесь, что первый и последний компоненты расположены в нужных местах. То же самое верно и для опции Space Equally Horizontally (Установить равный шаг по горизонтали).

Опции Center Horizontally in Window (Отцентрировать в окне по горизонтали) и Center Vertically in Window (Отцентрировать в окне по вертикали) работают в точности так, как это следует из их названий. Они удобны для центрирования отдельных компонентов (например, кнопок) или групп компонентов относительно формы. Пока кнопки в вашей форме остаются выделенными, попробуйте обе опции в действии, щелкнув на соответствующих кнопках палитры выравнивания. Вы увидите, как кнопки будут отцентрированы по горизонтали и вертикали.



Когда вы выделяете несколько компонентов и применяете к ним операцию центрирования, эти компоненты рассматриваются как группа. Если вы будете выделять каждый компонент отдельно и центрировать его по горизонтали и вертикали, все компоненты наложатся друг на друга в середине формы. Предварительное объединение компонентов в группу позволяет сохранить их взаимное расположение.

Сейчас форма должна выглядеть как на рис. 7.13.

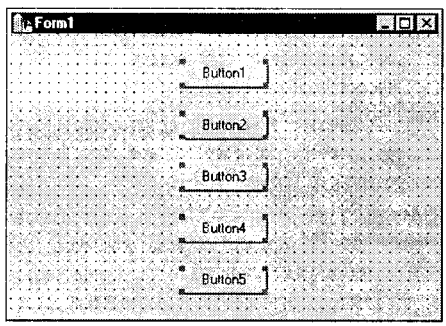


Рис. 7.13
Форма с отцентрированными кнопками



Две опции центрирования относительно окна могут быть использованы для выравнивания компонентов, содержащихся внутри других компонентов, например, кнопок на панели. Центрирование всегда осуществляется относительно порождающего компонента, которым может служить как форма, так и панель.

Опции Align Tops, Align Bottoms и Align Right Edges работают точно так же, как и опция Align Left Edges, рассмотренная выше, поэтому нет особого смысла останавливаться на каждой из них в отдельности.



При выравнивании краев первый выделенный компонент будет являться опорным. Вернитесь к рис. 7.4. Допустим, вы выбрали Button3 и затем добавили к группе остальные кнопки, используя Shift+щелчок. Теперь при выборе опции Align Left Edges кнопка Button3 останется на своем месте, а левые края остальных кнопок будут выровнены по ней.

Опции Align Horizontal Centers (Выровнять центры по вертикали) и Align Vertical Centers (Выровнять центры по горизонтали) могут быть использованы для центрирования компонентов друг относительно друга. Это лучше всего проиллюстрировать с помощью геометрических фигур (компонент Shape). Создайте новую форму (или удалите кнопки с предыдущей) и выполните следующие действия:

1. Щелкните на вкладке Additional палитры компонентов и выберите компонент Shape. Поместите его где-нибудь в верхнем левом углу формы.
2. Измените значение свойства Shape на stCircle.
3. Установите для свойства Width значение 150.
4. Дважды щелкните на свойстве Brush и измените значение свойства Color на clBlack.
5. Поместите в форму еще один компонент Shape.
6. Измените значение свойства Shape второго компонента на stCircle. Теперь на экране присутствуют два круга разного размера — белый и черный.
7. Щелкните на черном круге. Затем, удерживая клавишу Shift, щелкните на белом круге. Оба круга будут выделены.
8. Выберите в главном меню пункт View | Alignment Palette, если палитра выравнивания еще не выведена на экран. Расположите ее так, чтобы можно было наблюдать за обоими кругами.
9. Щелкните на кнопке Align Vertical Centers. Центры кругов будут размещены на одной горизонтальной линии.
10. Щелкните на кнопке Align Horizontal Centers. Центры кругов будут помещены на одну вертикальную линию. Поздравляю — вы изобрели колесо!

В этом примере черный круг остается неподвижен, поскольку он был выделен первым, и служит опорным компонентом. Белый же круг перемещается при нажатии кнопок выравнивания. Вы можете использовать рассмотренные опции для совмещения центров любого количества компонентов. Если выделен только один компонент, нажатие кнопок не вызывает никаких действий.

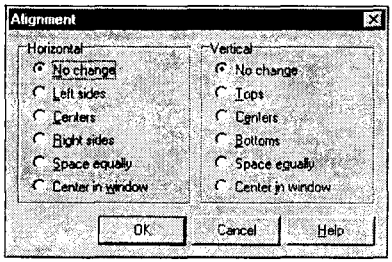
Палитра выравнивания имеет свое контекстное меню. Для его вызова установите курсор мыши на палитру и щелкните правой кнопкой. Описания пунктов контекстного меню приведены в таблице 7.2.

Таблица 7.2. Пункты контекстного меню палитры выравнивания

Пункт меню	Описание
Stay on top	Постоянное размещение палитры выравнивания поверх остальных окон. Это удобно, если вы часто переключаетесь между редактором форм и редактором меню. Окно палитры небольшое и легко может затеряться.
Show Hints	Включение и выключение режима отображения всплывающих подсказок для кнопок палитры.
Hide	Убирает палитру выравнивания с экрана. (Для этого можно использовать также кнопку закрытия окна.) Чтобы снова отобразить палитру, выберите в главном меню пункт View | Alignment Palette.
Help	Вызов справки по C++Builder на странице, посвященной палитре выравнивания.

Диалоговое окно Alignment (рис. 7.14) позволяет выполнить те же действия, что и палитра выравнивания. Для открытия диалогового окна выберите пункт Edit | Align в главном меню или пункт Align в контекстном меню редактора форм.

Рис. 7.14
Диалоговое окно Alignment



В большинстве случаев проще работать с палитрой выравнивания, но при желании вы можете, разумеется, воспользоваться диалоговым окном.

Свойство Align

Свойство Align определяет способ выравнивания компонента в окне. Возможные значения свойства Align и их описания приведены в таблице 7.3.

Таблица 7.3. Возможные значения свойства Align

Значение	Описание
alBottom	Компонент располагается вдоль нижнего края окна. Примером служит строка состояния, размещенная в нижней части главной формы.
alClient	Компонент заполняет всю рабочую область окна. Если в окне размещены другие компоненты, будет заполнена оставшаяся часть рабочей области. В качестве примеров можно привести компоненты Memo, Image и RichEdit.
alLeft	Компонент располагается вдоль левого края окна. Вертикальная панель инструментов является примером такого компонента.
alNone	Компонент размещается в произвольном месте окна. Это значение установлено по умолчанию для большинства компонентов.
alRight	Компонент располагается вдоль правого края окна.
alTop	Компонент располагается вдоль верхнего края окна. Примером может служить панель инструментов.

Давайте познакомимся с различными типами выравнивания на практике. Создайте пустую форму. Затем выполните следующие действия:

1. Щелкните на вкладке Standard палитры компонентов и выберите компонент Panel. Разместите панель в любом месте формы.
2. Найдите свойство Align в окне инспектора объектов (оно находится в самом начале списка). Заметьте, что для него установлено значение alNone. Измените значение этого свойства на alTop. Панель будет размещена вдоль верхнего края формы, заняв всю ширину окна.
3. Попробуйте переместить панель обратно в центр формы. Вы увидите, что это невозможно — панель привязана к верхнему краю.
4. Попробуйте сделать панель уже. Заметьте, что это не удается — ширина панели остается прежней.
5. Измените высоту панели. В отличие от ширины, высота может быть изменена.
6. Измените значение свойства Align на alBottom. Теперь панель «приклеилась» к нижнему краю формы.
7. Измените значение свойства Align сначала на alRight, а затем на alLeft. Обратите внимание, как панель сохраняет свои размеры. Ее ширина теперь такая же, какой была раньше высота. Это эквивалентно повороту панели. Попытки изменить высоту панели в данном случае потерпят неудачу.
8. Измените значение свойства Align на alClient. Панель расширится и займет всю доступную область формы. После этого ее размеры уже нельзя изменить.
9. Измените значение свойства Align на alNone. Панель снова можно перемещать, сжимать и растягивать.

Как вы только что видели, придание свойству Align любого значения, кроме alNone, приводит к фиксации панели на одном из краев формы. В случае alClient панель «приклеивается» ко всем четырем краям. Теперь, чтобы продемонстрировать совместную работу различных компонентов, мы создадим прототип приложения, похожего на Windows Notepad.



Прототип (prototype) — это приложение, которое внешне выглядит как рабочее приложение, но лишено части его функций.



C++Builder идеально подходит для быстрого создания прототипов приложений. Вы можете разработать главное окно и диалоговые окна намного быстрее, чем при использовании традиционных инструментов Си++ для программирования в Windows — MFC или OWL. Это, однако, не означает, что C++Builder годится только для разработки прототипов. С его помощью можно решать все задачи, встающие перед вами при программировании в 32-разрядной среде Windows.

Шаг 1. Создание нового приложения

1. Выберите в главном меню пункт New Application. Если появится запрос на сохранение текущего проекта, ответьте No.
2. Измените значение свойства Name на ScratchPad.

3. Измените значение свойства Caption на ScratchPad 1.0.
4. Выберите в главном меню пункт Options | Project. Щелкните на вкладке Application и введите ScratcPad 1.0 в качестве заголовка приложения. Нажмите кнопку OK, чтобы закрыть диалоговое окно Project Options.

Шаг 2. Создание панели инструментов

Большинство современных приложений Windows имеет панель инструментов. Ее создание проводится в несколько этапов. Сначала мы поместим в верхнюю часть окна разделитель. (Вы поймете его назначение позже, при добавлении меню.) Итак, начнем:

1. Выберите компонент Bevel (вкладка Additional) из палитры компонентов и поместите его в форму.
2. Установите для свойства Height значение 2.
3. Измените значение свойства Align на alTop. Линия расположится вдоль верхнего края рабочей области формы.

Шаг 3. Создание контейнера панели инструментов

Теперь мы можем добавить к форме панель, которая будет служить контейнером для кнопок быстрого доступа:

1. Выберите компонент Panel из палитры компонентов и поместите в любом месте формы.
2. Измените значение свойства Name на SpeedBar.
3. Установите значение свойства Height равным 32.
4. Измените значение свойства BevelOuter на bvNone.
5. Сотрите значение свойства Caption.
6. Измените значение свойства Align на alTop. Панель переместится наверх, но расположится ниже линии, которую мы поместили туда раньше.



Последнее действие иллюстрирует еще одну особенность свойства Align. Если вы установите для него значение alTop, компонент переместится к верхней границе рабочей области или к нижнему краю предыдущего компонента. Это позволяет разместить несколько компонентов в верхней части формы и обеспечить их автоматическую подстройку под размер окна.

Шаг 4. Заполнение панели

Все, что осталось сделать на данном этапе — это разместить на панели одну или две кнопки.

1. Щелкните на вкладке Additional палитры компонентов и выберите компонент SpeedButton.
2. Поместите кнопку быстрого доступа на панель (не в форму!). Не беспокойтесь об ее точном размещении.
3. Измените значение свойства Name на FileOpenBtn.
4. Установите значение свойства Left равным 5.
5. Выберите в главном меню пункт View | Alignment Palette. Щелкните на кнопке Center Vertically in Window.

6. Найдите свойство Glyph и дважды щелкните в столбце Value. Откроется окно редактора изображений.
7. Нажмите кнопку Load. На экране появится диалоговое окно Load picture. Перейдите в подкаталог \Images\Buttons каталога C++Builder и выберите файл fileopen.bmp. Нажмите кнопку OK.
8. Повторите последние семь шагов, чтобы добавить кнопку File Save. Разместите ее справа от кнопки File Open. Дайте ей имя FileSaveBtn и используйте файл filesave.bmp для свойства Glyph.

Сейчас форма должна выглядеть как на рис. 7.15.

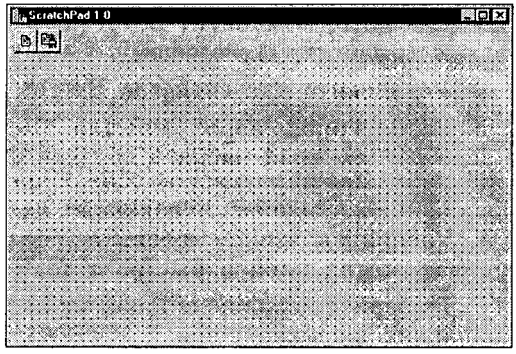


Рис. 7.15
Форма ScratchPad к этому моменту

Шаг 5. Добавление строки состояния

Пока все идет хорошо. Windows Notepad не имеет строки состояния (как, впрочем, и панели инструментов), но мы добавим ее в свое приложение:

1. Щелкните на вкладке Win95 палитры компонентов и выберите компонент StatusBar.
2. Щелкните в любом месте формы. Строка состояния автоматически разместится внизу формы. Свойство Align для нее по умолчанию имеет значение alBottom.
3. Измените значение свойства Name на StatusBar.

Шаг 6. Добавление компонента Memo

Нам нужен какой-то компонент, в котором можно набирать текст, поэтому используем компонент Memo (хотите верьте, хотите нет, но мы уже почти закончили наш прототип):

1. Щелкните на вкладке Standard палитры компонентов и выберите компонент Memo.Поместите его в любом месте рабочей области формы.
2. Измените значение свойства Name на Memo.
3. Измените значение свойства WordWrap на true (если это необходимо).
4. Дважды щелкните в столбце Value рядом со свойством Lines. Откроется окно редактора строк (String List Editor). Сотрите слово Memo и нажмите кнопку OK.
5. Измените значение свойства ScrollBar на ssVertical. (Для начала нам хватит только вертикальной линейки прокрутки.)
6. Установите шрифт Fixedsys (атрибут Name свойства Font). Мы используем системный шрифт, поскольку создаем аналог Notepad'a.

7. Измените значение свойства Align на aClient. Поле компонента заполнит всю рабочую область формы между панелью инструментов и строкой состояния.

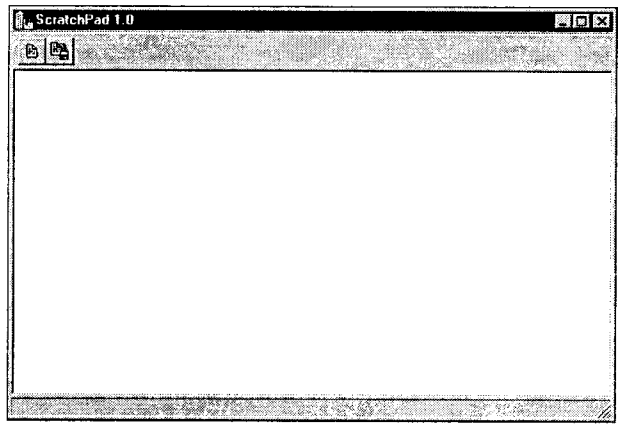
Ну вот, это уже похоже на настоящее приложение! До того, как двинуться дальше, давайте сделаем еще одну вещь, которая продемонстрирует роль свойства Align. Захватите мышью нижний правый угол формы и немного растяните ее. Если вы любите точность, то можете установить для свойств Height и Width значения 375 и 575, соответственно.



Рабочая область нашей формы полностью закрыта компонентами. Это делает невозможным выбор формы простым щелчком на ней. Чтобы сделать форму текущим компонентом в окне инспектора объектов, выберите любой компонент и нажмите клавишу Esc. Можно также выбрать форму в комбинированном списке селектора компонентов.

Обратите внимание, что все компоненты автоматически изменили свои размеры, чтобы сохранить заданное расположение относительно порождающего окна — в данном случае формы. Это является одним из основных достоинств свойства Align. Теперь форма выглядит как на рис. 7.16.

Рис 7.16
Завершенный прототип



Запускаем программу

Щелкните на кнопке Run, чтобы запустить программу. После этого вы можете набирать текст в рабочей области окна и нажимать на кнопки быстрого доступа (хотя они пока ничего не делают). Не забывайте, что это прототип, который выполняет сейчас в основном демонстрационные задачи. К концу главы мы дополним программу.

Сейчас неплохо сохранить проект, чтобы можно было вернуться к нему позже. Выберите в главном меню пункт File | Save All. Дайте исходному модулю главной формы имя SPMain, а проекту — Scratch.

Установка порядка переключения



Порядок переключения (tab order) — это последовательность, в которой компоненты получают *фокус ввода (input focus)* при нажатии клавиши Tab.

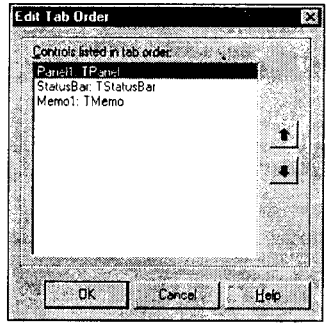
Формы C++Builder автоматически поддерживают перемещение между компонентами с помощью клавиши Tab. Вы можете выбирать следующий компонент, нажимая Tab, и возвращаться обратно, используя сочетание Shift+Tab.



Существуют две разновидности визуальных компонентов. Компоненты *оконного типа (windowed)* принимают фокус ввода; к ним относятся Edit, Memo, ListBox, ComboBox, Button, а также многие другие.
Компоненты *не оконного типа (non-windowed)* не реагируют на клавиатуру. Такими компонентами являются Image, SpeedButton, Label, Shape и другие. Порядок переключения устанавливается только для компонентов оконного типа. Остальные компоненты исключены из этой последовательности.

Порядок переключения изначально устанавливается в соответствии с последовательностью помещения компонентов в форму на этапе разработки. Его можно изменить установкой значения свойства TabOrder для каждого из компонентов. Однако этот способ довольно утомителен, поскольку нужно обращаться к каждому компоненту отдельно. Более легкий путь предоставляет диалоговое окно Edit Tab Order (рис. 7.17).

Рис. 7.17
Диалоговое окно
Edit Tab Order



Диалоговое окно Edit Tab Order содержит список всех переключаемых компонентов, находящихся в форме. Непереключаемые компоненты не отображаются. Чтобы изменить порядок переключения, щелкните на имени компонента и нажатием кнопок со стрелками переместите его на нужное место в списке. Можно также перетащить компонент мышью. После редактирования списка щелкните на кнопке OK, и новый порядок переключения вступит в силу. Вы можете проверить установки, просмотрев свойства TabOrder каждого компонента.



Нумерация компонентов в последовательности переключения начинается с нуля. Номер первого компонента — 0, второго — 1, и т.д.

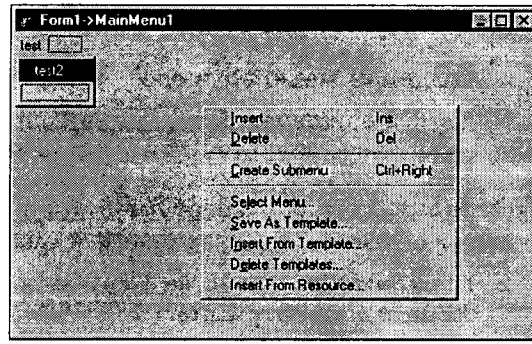
Меню, пожалуйста!

Меню являются важной частью большинства приложений Windows. В некоторых программах меню отсутствует, но таких программ очень мало. В C++Builder для создания меню используется специальный редактор меню, что сильно облегчает работу. Редактор меню обладает следующими возможностями:

- Создание как главного, так и контекстных меню
- Обеспечение оперативного доступа к редактору кода для обработки событий OnClick, соответствующих пунктам меню
- Вставка меню из шаблонов или файлов ресурсов
- Сохранение пользовательских меню в виде шаблонов

Все команды редактора меню доступны через его контекстное меню или из инспектора объектов. Контекстное меню редактора показано на рис. 7.18.

Рис. 7.18
Контекстное меню редактора меню



Основное назначение пунктов меню следует из их названий, поэтому я не буду сейчас объяснять каждый пункт. Вы изучите их на практике. Для начала, давайте добавим к ранее созданной программе ScratchPad главное меню. После этого мы создадим и контекстное меню.

Создание главного меню

Редактор меню позволяет быстро разрабатывать меню любой структуры. Главное меню представлено компонентом MainMenu, которому соответствует класс TMainMenu библиотеки VCL. Каждый пункт меню, в свою очередь, является компонентом MenuItem, инкапсулированным в классе TMenuItem. Вам не требуется знать о том, как функционируют эти классы, поскольку редактор меню берет на себя всю работу по созданию меню. После этого краткого вступления, давайте добавим главное меню к программе ScratchPad.

Добавление главного меню к форме

Прежде всего вы должны добавить к форме компонент MainMenu.



Сейчас у вас есть определенный опыт работы с C++Builder. Начиная с этого момента, я не буду подробно описывать все операции. Например, вы встретите указание «Поместите компонент MainMenu в форму.» вместо «Щелкните на вкладке Standard палитры компонентов. Выберите компонент MainMenu и щелкните на форме для его размещения.» Не беспокойтесь; новые для вас операции будут описаны во всех деталях.

1. Откройте проект ScratchPad, созданный раньше в этой главе.
2. Поместите в форму компонент MainMenu и измените значение его свойства Name на MainMenu. Обратите внимание, что компонент MainMenu

имеет очень мало свойств и совсем не имеет событий. Дело в том, что работа меню полностью определяется компонентами MenuItem.

3. Дважды щелкните на значке главного меню. На экране появится редактор меню.

Редактор меню выглядит как пустая форма без узлов сетки. Вы можете изменить его размер произвольным образом. Размер редактора определяется исключительно из соображений удобства, и не влияет на работу меню во время выполнения программы. Сейчас редактор меню ожидает ваших действий. Создав свое первое меню, вы убедитесь, что это делается достаточно легко и интуитивно понятно.

Создание меню вручную

Хотя существует более легкий способ создания меню File, первый раз вы сделаете это вручную. Редактор меню всегда содержит пустой пункт меню, который обозначает место для вновь создаваемого пункта. При первом запуске редактора меню пустой пункт будет выделен.

1. Измените значение свойства Name на FileMenu.
2. Щелкните на свойстве Caption в окне инспектора объектов, наберите &File и нажмите Enter.



Амперсанд (&) используется для ввода подчеркнутого символа в название пункта меню. Подчеркнутый символ служит для ускоренного выбора пункта меню с клавиатуры путем нажатия соответствующей клавиши в сочетании с Alt. Вы можете поместить амперсанд в любом месте текста. Например, строка для пункта меню Exit имеет вид E&xit. Вам нужно только поместить амперсанд перед нужным символом, а Windows сделает все остальное.

Сейчас произошло сразу несколько вещей. Во-первых, в редакторе меню появился пункт File. Он появился и в главной форме, закрытой редактором меню. (Помните, мы добавили к главной форме ScratchPad компонент Bevel в качестве разделителя? Теперь вы видите, что он служит для визуального разделения меню и панели инструментов.) Во-вторых, под только что созданным пунктом File появился новый пустой пункт, и такой же пустой пункт возник справа. Инспектор объектов содержит пустой компонент MenuItem, ожидая ввода значений для свойств Caption и Name. На рис. 7.19 показан редактор меню к данному моменту.

Давайте продолжим создание меню:

1. Измените значение свойства Name нового пункта на FileNew.
2. Измените значение свойства Caption на &New и нажмите Enter. Редактор меню снова создаст пустой пункт.
3. Повторите шаги 1 и 2 для создания пунктов меню Open, Save и Save As. Если вы не помните, где должен быть расположен амперсанд, обратитесь к рис. 7.20. Не бойтесь ошибиться. Вы всегда можете вернуться назад и исправить любые ошибки.

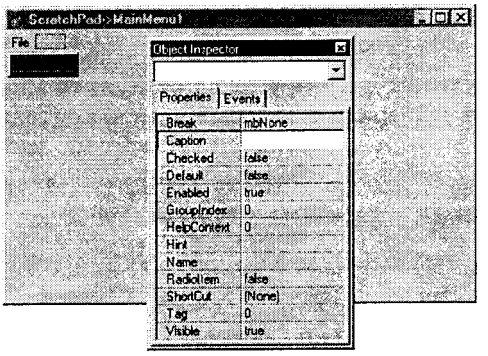


Рис. 7.19
Редактор меню и инспектор объектов после создания меню File



Делайте свои меню по возможности ближе к стандартным. Используйте те же подчеркнутые символы, что и в других программах Windows. Помните также, что многоточие (...) в пункте меню означает, что выбор данного пункта приведет к открытию диалогового окна.

Теперь нам нужен разделитель меню.



Разделитель (separator) — это горизонтальная линия в меню, разделяющая группы пунктов.

Добавление разделителя с помощью редактора меню C++Builder выглядит очень просто. Нужно только установить для свойства Caption значение «-» (дефис). Перейдите к пустому пункту меню под Save As, введите дефис в поле Caption и нажмите Enter. Добавьте в меню остальные пункты, пока оно не будет выглядеть, как на рис. 7.20. Если потребуется изменить какой-либо пункт, просто щелкните на нем и измените его свойства в инспекторе объектов.

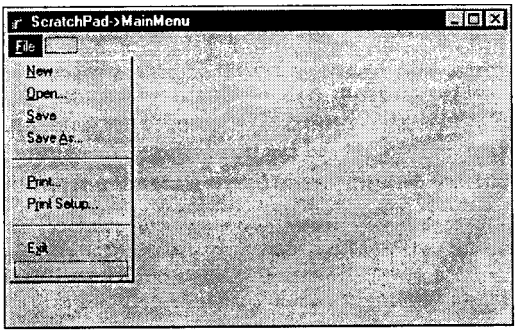


Рис. 7.20
Редактор меню с законченным меню File



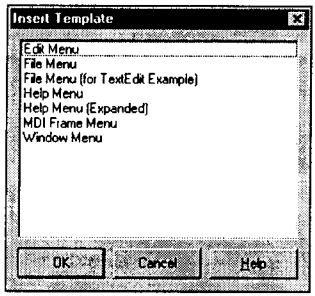
Редактор меню всегда оставляет пустой пункт в конце каждого всплывающего меню и справа от строки меню. Вы не можете их стереть, но это и не нужно — они используются только в редакторе и не отображаются при выполнении программы.

Теперь, когда меню File завершено, остается создать меню Edit и Help.

Вставка меню из шаблона

На этот раз мы используем более легкий способ. Сначала щелкните на пустом пункте справа от пункта File. Затем щелкните правой кнопкой мыши и выберите в открывшемся контекстном меню пункт Insert From Template. На экране появится диалоговое окно Insert Template (рис. 7.21).

Рис. 7.21
Диалоговое окно Insert Template



В диалоговом окне приведен список доступных шаблонов. Вы можете использовать предопределенные шаблоны или создавать свои собственные. Сейчас нам нужно загрузить меню Edit, поэтому выберите из списка Edit Menu и нажмите кнопку OK. В редактор меню сразу же будет вставлено полное меню Edit. На самом деле оно содержит больше пунктов, чем нужно. Мы займемся этим чуть позже.

Давайте заодно вставим и меню Help. Щелкните на пустом пункте справа от Edit. Снова выберите в контекстном меню пункт Insert From Template и вставьте шаблон Help Menu. (Не выберите по ошибке Expanded Help Menu.) Мы приведем в порядок меню Edit и Help в следующем разделе. Обратите внимание, что вновь вставленные пункты уже отображены в главной форме.



Вы можете использовать шаблоны для создания контекстных меню точно так же, как и при создании пунктов главного меню.

Поработав с C++Builder какое-то время, вы наверняка создадите свои собственные шаблоны, чтобы использовать их для столь же быстрого построения меню. Вам по-прежнему останется только дать свойствам Name осмысленные значения, но это намного проще, чем вводить каждый пункт вручную.



Пункт Insert From Resource работает аналогично Insert From Template, за исключением того, что описание меню берется из файла сценария ресурсов (.RC). Этот вариант вставки используется при преобразовании программ на Си или Си++ в формат C++Builder.

Удаление пунктов меню

При создании приложений Windows редко удается получить желаемый результат с первого раза. Пользователи захотят дополнительных возможностей, руководство предъявит свои требования, и кое-что даже придется выкидывать. В результате меню ваших приложений часто будут требовать об-

новления. Например, меню Edit, вставленное ранее, содержит слишком много пунктов. Некоторые из них никогда не будут использоваться в нашей программе. Нет проблем — мы просто сотрем их:

1. Щелкните на пункте Edit.
2. Щелкните на пункте Repeat<command>.
3. Нажмите клавишу Delete или выберите пункт Delete в контекстном меню редактора. Выделенный пункт исчезнет, а оставшиеся сдвинутся вверх.
4. Удалите таким же образом пункт Paste Special.

Сейчас меню Edit еще не совсем готово, но до того, как двинуться дальше, я покажу одну полезную возможность редактора меню. Вы, вероятно, хорошо освоили выбор объектов в Windows-программах с помощью комбинаций Shift+щелчок и Ctrl+щелчок. Эта техника используется, например, в проводнике Windows для выбора файлов. Редактор меню также поддерживает комбинации Shift+щелчок и Ctrl+щелчок, но с одной оговоркой: они могут быть использованы для выделения нескольких пунктов, но не для снятия выделения. Как всегда, с этим проще разобраться на конкретном примере.

1. Меню Edit должно сейчас присутствовать на экране. В противном случае вызовите его щелчком на пункте Edit.
2. Щелкните на пункте Goto.
3. Нажмите клавишу Shift и щелкните на пункте Object. Все пункты между двумя названными будут выделены.
4. Нажмите клавишу Delete, чтобы удалить сразу все пункты.
5. Перейдите к меню Help и удалите два пункта в середине. Должны остаться только пункты Contents и About.

Как вы видели, комбинация Shift+щелчок позволяет быстро удалить ненужные пункты. Теперь мы приведем меню к окончательному виду, который оно должно иметь в приложении ScratchPad.

Вставка пунктов меню

Вставка пунктов меню выполняется очень просто. Щелкните на том пункте, над которым вы хотите разместить новый пункт, и нажмите клавишу Insert (или выберите Insert в контекстном меню редактора). После этого вы можете установить свойства Caption и Name для нового пункта. Давайте вставим еще один пункт в меню Edit:

1. Вызовите меню Edit.
2. Щелкните на пункте Find.
3. Нажмите клавишу Insert. В меню возникнет новый пункт, а все лежащие под ним пункты сдвинутся вниз.
4. Измените значение свойства Name на EditSelectAll, а свойства Caption — на Select &All.
5. Щелкните на пустом пункте в нижней части меню Edit. Добавьте разделитель (вспомните, что для этого достаточно ввести в поле Caption дефис).
6. Снова щелкните на пустом пункте. Установите для свойства Name значение EditWordWrap, а для свойства Caption — &Word Wrap.

Перемещение пунктов меню

При необходимости вы можете легко перемещать пункты меню. При этом возможно перемещение как в пределах одного всплывающего меню, так и между различными меню. Для перемещения пунктов существуют два способа. Первый заключается в использовании буфера обмена. Операции Cut и Paste выполняются как обычно, поэтому я не буду описывать их еще раз. Другой способ заключается в использовании техники drag-and-drop. Давайте попробуем это сделать. Нам нужно разместить пункт Select All под пунктом Undo. Нет проблем — просто перетащите его:

1. Щелкните на пункте Edit для вызова соответствующего меню.
2. Щелкните на пункте Select All и перетащите его, пока разделитель под Undo не окажется выделен.
3. Отпустите кнопку мыши, и пункт займет новое место.

Просто, не правда ли? Но C++Builder для этого и создавался!

Пакетная модификация свойств

Иногда вам нужно изменить свойства сразу нескольких пунктов меню. Например, в меню нашего приложения ScratchPad есть пункты, которые мы пока не готовы реализовать (поддержка печати, вызов справки). Их нужно выключить (запретить):

1. Выберите в редакторе меню пункт Help | Contents.
2. Измените значение свойства Enabled на false.
3. Щелкните на пункте File.
4. Щелкните на пункте Print, затем, удерживая нажатой клавишу Shift, щелкните на пункте Print Setup. Оба пункта окажутся выделены.
5. Измените значение свойства Enabled в окне инспектора объектов на false. Оба пункта будут выключены.
6. Повторите шаги 4 и 5 для выключения пунктов Find и Replace в меню Edit.

Так вы можете изменить несколько пунктов меню за один раз. Просто выделите эти пункты и затем измените значение нужного свойства. Свойства всех выделенных пунктов получат новые значения.

Создание подменю

Создание подменю не имеет каких-либо особенностей. *Подменю (submenu)* — это меню, которое вызывается при выборе пункта вышестоящего меню. Наличие подменю обозначается стрелкой вправо, расположенной за названием пункта. Вы можете создать подменю, выбрав в контекстном меню редактора пункт Create Submenu, или нажав одновременно Ctrl и стрелку вправо на клавиатуре. При создании подменю справа от соответствующего пункта основного меню размещается пустой пункт. Вы можете добавлять в подменю пункты точно так же, как и при создании главного меню. Создать подменю можно также вставкой шаблона.

Добавление сочетаний клавиш

Вы можете легко добавить сочетания клавиш, соответствующие пунктам меню, изменив значение свойства ShortCut в инспекторе объектов. Для меню

Edit, которое мы вставили из шаблона, сочетания клавиш уже определены. Например, пункту Cut соответствует сочетание Ctrl+X. Если вы посмотрите на меню Edit, то увидите, что за названием пункта стоит Ctrl+X. Щелкнув на пункте Cut, вы увидите, что свойство ShortCut в окне инспектора объектов также имеет значение Ctrl+X. Щелкните в столбце Value рядом со свойством ShortCut. Справа вы увидите кнопку со стрелкой. Щелкните на ней для вызова списка возможных значений. Этот список содержит практически все сочетания, которые могут когда-либо понадобиться. Чтобы установить новое сочетание клавиш, просто выберите его из списка.

Стандартным сочетанием клавиш для Select All является Ctrl+A, поэтому давайте установим его для этого пункта меню:

1. Выберите в созданном меню пункт Edit | Select All.
2. Щелкните на свойстве ShortCut в окне инспектора объектов.
3. Выберите Ctrl+A из списка доступных сочетаний. Теперь за названием пункта Select All стоит Ctrl+A.

Это все, что вам нужно сделать; остальное берет на себя C++Builder. Для использования сочетаний клавиш не требуется писать какой-либо код.

Заключительные действия

Давайте сделаем еще несколько вещей. Во-первых, мы по умолчанию включим пункт Word Wrap. Этот пункт предназначен для включения и выключения режима заворачивания слов. Когда режим заворачивания включен, пункт Word Wrap отмечен. Щелкните на пункте Word Wrap и установите для свойства Checked значение true. При этом рядом с названием пункта появится отметка, сообщающая о его включении.

Во-вторых, нам нужно установить значения свойств Name для всех пунктов, вставленных из шаблона. Они имеют имена по умолчанию, которые желательно заменить на более информативные. Выберите пункт Edit | Undo. Измените значение свойства Name с Undo1 на EditUndo. Обратите внимание, что это имя образовано присоединением спереди названия всплывающего меню, Edit, и удалением 1 сзади. Вы можете использовать любое соглашение об именах, но будьте при этом последовательны. Повторите процедуру для пунктов Cut, Copy, Paste, Find и Replace. Потом перейдите к меню Help и измените значения свойств Name для пунктов Contents и About на HelpContents и HelpAbout.

На этом создание меню завершено. Просмотрите его еще раз и устраните замеченные ошибки. Убедившись, что меню построено правильно, закройте окно редактора меню.



Вы можете вызвать редактор кода прямо из редактора меню, дважды щелкнув на любом пункте меню. В окне редактора кода будет находиться обработчик события OnClick для данного пункта, и вы сможете приступить к вводу кода. Однако сейчас мы собираемся вернуться к главной форме и редактировать код оттуда.

Написание кода

Итак, теперь у нас есть все пункты меню, но нет кода, который заставил бы их работать. Вам кажется, что его создание потребует больших усилий? На самом деле это совсем не так. Большая часть кода уже содержится в классе TMemo. Все, что нужно сделать — это вызвать подходящие методы этого класса из обработчиков пунктов меню. Конечно, придется сделать еще кое-что, но с основной частью кода вы встречались раньше.

До того, как начать писать код, необходимо добавить к форме обычные компоненты OpenDialog и SaveDialog:

1. Поместите в форму компонент OpenDialog.
2. Измените значение свойства Name на OpenDialog.
3. Поместите в форму компонент SaveDialog.
4. Измените значение свойства Name на SaveDialog.
5. Разместите значки MainMenu, OpenDialog и SaveDialog на одной линии.

Теперь давайте приступим к созданию кода для пунктов меню. Мы начнем с пункта File | Exit (он самый простой!). Убедитесь, что редактор меню закрыт, чтобы случайно не перепутать его с редактором форм.

1. Выберите в главном меню формы пункт File | Exit. Окно редактора кода переместится на передний план и в нем будет содержаться обработчик события FileExitClick().
2. Курсор расположен в том месте, откуда нужно начинать ввод. Наберите:

```
Close();
```



В шаге 2 я предложил вам использовать функцию Close() для закрытия формы. Она применима здесь, поскольку мы имеем дело с главной формой приложения. Но если вы хотите завершать приложение из любого места программы, следует использовать другую функцию:

```
Application->Terminate();
```

При этом приложение будет завершено независимо от того, какая форма в данный момент открыта.

Вот и все. Давайте напишем код еще для одного пункта, а все остальное вы сделаете самостоятельно.

1. Выберите в главном меню пункт Edit | Cut. Окно редактора кода переместится на передний план и в нем будет содержаться обработчик события EditCutClick().
2. Введите следующий текст, начиная с позиции курсора:

```
Memo->CutToClipboard();
```

Для данного пункта это все! Большую часть работы, о которой вы можете и не знать, берет на себя VCL. В этом и состоит основная идея, лежащая в основе библиотек классов — избавить программиста от груза низкоуровневых деталей.

Одной из интересных особенностей сред типа C++Builder является то, что вы редко видите свою программу целиком. C++Builder показывает вам те части кода, которые соответствуют отдельным событиям, поэтому обычно вы видите лишь небольшие куски программы. Листинг 7.1 содержит заголовок для программы ScratchPad в ее нынешнем виде. Этот файл целиком сгенерирован C++Builder. Вся программа SPMAIN.CPP приведена в листинге 7.2. При написании кода для каждого из оставшихся пунктов меню, ориентируйтесь на только что рассмотренные примеры. Скопируйте код для обработчиков событий OnClick из листинга 7.2. (Строки комментариев служат для объяснения выполняемых действий. Вы не должны вводить их.)



Обработчики событий появляются в исходном файле в той последовательности, в которой они были созданы. Эта последовательность для вашего исходного файла может не совпадать с листингом 7.2. Порядок размещения функций не имеет значения для компилятора.

Листинг 7.1. SPMAIN.H

```
 1: //---------------------------------------------------------------------
 2: #ifndef SPMainH
 3: #define SPMainH
 4: //---------------------------------------------------------------------
 5: #include <Classes.hpp>
 6: #include <Controls.hpp>
 7: #include <StdCtrls.hpp>
 8: #include <Forms.hpp>
 9: #include <ExtCtrls.hpp>
10: #include <Buttons.hpp>
11: #include <ComCtrls.hpp>
12: #include <Menus.hpp>
13: #include <Dialogs.hpp>
14: //---------------------------------------------------------------------
15: class TScratchPad : public Tform
16: {
17: __published:    // Компоненты, управляемые IDE
18:     TPanel *Panel1;
19:     TBevel *Bevel1;
20:     TSpeedButton *FileOpenBtn;
21:     TSpeedButton *FileSaveBtn;
22:     TStatusBar *StatusBar;
23:     TMainMenu *MainMenu;
24:     TMenuItem *FileMenu;
25:     TMenuItem *FileOpen;
26:     TMenuItem *FileSave;
27:     TMenuItem *FileSaveAs;
28:     TMenuItem *N1;
29:     TMenuItem *FilePrintSetup;
30:     TMenuItem *N2;
31:     TMenuItem *FileExit;
32:     TMenuItem *FilePrint;
33:     TMenuItem *Edit1;
34:     TMenuItem *EditReplace;
35:     TMenuItem *EditFind;
```

см. продолжение

Листинг 7.1. (продолжение)

```
36:     TMenuItem *N4;
37:     TMenuItem *EditPaste;
38:     TMenuItem *EditCopy;
39:     TMenuItem *EditCut;
40:     TMenuItem *N5;
41:     TMenuItem *EditUndo;
42:     TMenuItem *Help1;
43:     TMenuItem *HelpAbout;
44:     TMenuItem *HelpContents;
45:     TMenuItem *EditSelectAll;
46:     TMenuItem *N3;
47:     TMenuItem *EditWordWrap;
48:     TOpenDialog *OpenDialog;
49:     TSaveDialog *SaveDialog;
50:     TMenuItem *FileNew;
51:     TMemo *Memo;
52:     TPopupMenu *MemoPopup;
53:     TMenuItem *PopupCut;
54:     TMenuItem *PopupCopy;
55:     TMenuItem *PopupPaste;
56:     void __fastcall FileOpenClick(TObject *Sender);
57:     void __fastcall FileSaveClick(TObject *Sender);
58:     void __fastcall FileSaveAsClick(TObject *Sender);
59:     void __fastcall FileExitClick(TObject *Sender);
60:     void __fastcall EditSelectAllClick(TObject *Sender);
61:     void __fastcall EditCutClick(TObject *Sender);
62:     void __fastcall EditCopyClick(TObject *Sender);
63:     void __fastcall EditPasteClick(TObject *Sender);
64:
65:
66:     void __fastcall EditWordWrapClick(TObject *Sender);
67:     void __fastcall FileNewClick(TObject *Sender);
68:     void __fastcall EditUndoClick(TObject *Sender);
69:   private:        // Объявления пользователя
70:   public:         // Объявления пользователя
71:     virtual __fastcall TScratchPad(TComponent* Owner);
72: };
73: //---------------------------------------------------------------------
74: extern TScratchPad *ScratchPad;
75: //---------------------------------------------------------------------
76: #endif
```

Листинг 7.2. SPMAIN.CPP

```
 1: //---------------------------------------------------------------------
 2: #include <vcl.h>
 3: #pragma hdrstop
 4: #include "SPMain.h"
 5: //---------------------------------------------------------------------
 6: #pragma resource "*.dfm"
 7: TScratchPad *ScratchPad;
 8: //---------------------------------------------------------------------
 9: __fastcall TScratchPad::TScratchPad(TComponent* Owner)
10:     : TForm(Owner)
11: {
```

```
12: }
13: //---------------------------------------------------------------------
14: void __fastcall TScratchPad::FileNewClick(TObject *Sender)
15: {
16:   //
17:   // Открытие нового файла. Перед этим
18:   // проверка необходимости сохранения текущего файла
19:   //
20:   if (Memo->Modified) {
21:     //
22:     // Отображение окна сообщения
23:     //
24:     int result = Application->MessageBox(
25:       "The current file has changed. Save changes?",
26:       "ScratchPad Message", MB_YESNOCANCEL | MB_ICONWARNING);
27:     //
28:     // Если нажата кнопка Yes, сохранение текущего файла
29:     //
30:     if (result == IDYES) FileSaveClick(Sender);
31:     //
32:     // If No was clicked then do nothing.
33:     //
34:     if (result == IDCANCEL) return;
35:   }
36:   //
37:   // Стирание строк в компоненте Memo, если они присутствуют
38:   //
39:   if (Memo->Lines->Count > 0) Memo->Clear();
40:   //
41:   // Установка для свойства FileName диалога Save Dialog
42:   // значения "пустая строка".
43:   // Это будет означать, что файл еще не был сохранен
44:   //
45:   SaveDialog->FileName = "";
46: }
47: //---------------------------------------------------------------------
48: void __fastcall TScratchPad::FileOpenClick(TObject *Sender)
49: {
50:   //
51:   // Открытие файла. Проверка необходимости сохранения
52:   // текущего файла (логика работы аналогична FileNewClick())
53:   //
54:   if (Memo->Modified) {
55:     int result = Application->MessageBox(
56:       "The current file has changed. Save changes?",
57:       "ScratchPad Message", MB_YESNOCANCEL | MB_ICONWARNING);
58:     if (result == IDYES) FileSaveClick(0);
59:     if (result == IDCANCEL) return;
60:   }
61:   //
62:   // Вызов диалога File Open. Если нажата кнопка OK,
63:   // открытие файла с помощью метода LoadFromFile().
64:   // Перед этим очистка свойства FileName
65:   //
66:   OpenDialog->FileName = "";
67:   if (OpenDialog->Execute())
68:   {
```

см. продолжение

Листинг 7.2. (продолжение)

```
69:     if (Memo->Lines->Count > 0) Memo->Clear();
70:     Memo->Lines->LoadFromFile(OpenDialog->FileName);
71:     SaveDialog->FileName = OpenDialog->FileName;
72:   }
73: }
74: //---------------------------------------------------------------------
75: void __fastcall TScratchPad::FileSaveClick(TObject *Sender)
76: {
77:   //
78:   // Если имя файла уже было введено, нет необходимости
79:   // вызывать диалог File Save. Достаточно сохранить файл
80:   // с помощью SaveToFile()
81:   //
82:   if (SaveDialog->FileName != "")
83:   {
84:     Memo->Lines->SaveToFile(SaveDialog->FileName);
85:     //
86:     // Установка Modified в false, т.к. файл уже сохранен
87:     //
88:     Memo->Modified = false;
89:   }
90:   //
91:   // Если имя файла не было указано, выполнение SaveAs()
92:   //
93:   else FileSaveAsClick(Sender);
94: }
95: //---------------------------------------------------------------------
96: void __fastcall TScratchPad::FileSaveAsClick(TObject *Sender)
97: {
98:   //
99:   // Вызов диалога File Save для сохранения файла.
100:  // Установка Modified в false, т.к. файл уже сохранен
101:  //
102:  SaveDialog->Title = "Save As";
103:  if (SaveDialog->Execute())
104:  {
105:    Memo->Lines->SaveToFile(SaveDialog->FileName);
106:    Memo->Modified = false;
107:  }
108: }
109: //---------------------------------------------------------------------
110: void __fastcall TScratchPad::FileExitClick(TObject *Sender)
111: {
112:   //
113:   // Работа завершена. Закрытие формы
114:   //
115:   Close();
116: }
117: //---------------------------------------------------------------------
118: void __fastcall TScratchPad::EditUndoClick(TObject *Sender)
119: {
120:   //
121:   // TMemo не содержит метода Undo, поэтому мы должны послать
122:   // сообщение Windows WM_UNDO для компонента Memo
123:   //
124:   SendMessage(Memo->Handle, WM_UNDO, 0, 0);
125: }
```

```
126: //---------------------------------------------------------------------
127: void __fastcall TScratchPad::EditSelectAllClick(TObject *Sender)
128: {
129:   //
130:   // Вызов TMemo::SelectAll()
131:   //
132:   Memo->SelectAll();
133: }
134: //---------------------------------------------------------------------
135: void __fastcall TScratchPad::EditCutClick(TObject *Sender)
136: {
137:   //
138:   // Вызов TMemo::CutToClipboard()
139:   //
140:   Memo->CutToClipboard();
141: }
142: //---------------------------------------------------------------------
143: void __fastcall TScratchPad::EditCopyClick(TObject *Sender)
144: {
145:   //
146:   // Вызов TMemo::CopyToClipboard()
147:   //
148:   Memo->CopyToClipboard();
149: }
150: //---------------------------------------------------------------------
151: void __fastcall TScratchPad::EditPasteClick(TObject *Sender)
152: {
153:   //
154:   // Вызов TMemo::PasteFromClipboard().
155:   //
156:   Memo->PasteFromClipboard();
157: }
158: //---------------------------------------------------------------------
159: void __fastcall TScratchPad::EditWordWrapClick(TObject *Sender)
160: {
161:   //
162:   // Переключение свойства TMemo::WordWrap. Установка такого же
163:   // значения для свойства Checked пункта меню Word Wrap
164:   //
165:   Memo->WordWrap = !Memo->WordWrap;
166:   EditWordWrap->Checked = Memo->WordWrap;
167:   //
168:   // Если режим WordWrap включен, достаточно только вертикальной
169:   // линейки прокрутки; в противном случае требуются обе линейки
170:   //
171:   if (Memo->WordWrap) Memo->ScrollBars = ssVertical;
172:   else Memo->ScrollBars = ssBoth;
173: }
174: //---------------------------------------------------------------------
```

Момент, которого вы ждали

После создания обработчиков событий для пунктов меню, все готово для запуска программы. Нажмите кнопку Run. Программа должна откомпилироваться и заработать. Если вы получили сообщения об ошибках, тщательно сравните ваш исходный код с листингом 7.2. Внесите необходимые коррек-

тивы и снова нажмите кнопку Run. Может случиться, что вам придется делать это не один раз. Но в конце концов программа заработает (я вам обещаю!).

Поработав с программой, вы обнаружите, что она во многом похожа на Windows Notepad (хотя и не обладает всеми его возможностями). Хотя наша программа еще не вполне завершена, это неплохое начало — особенно если учесть то время, которое потрачено на ее создание. Окно работающей программы ScratchPad показано на рис. 7.22.

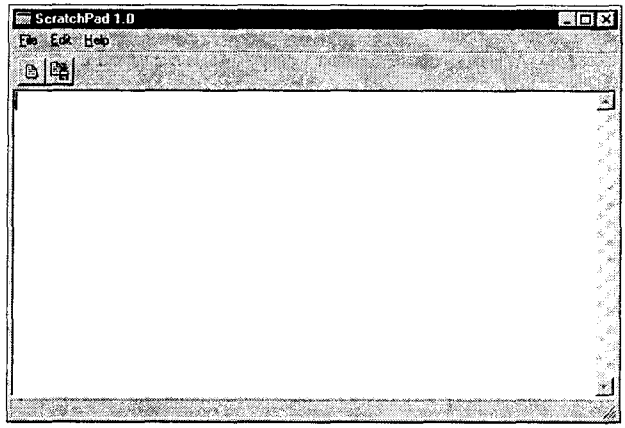


Рис. 7.22
Программа ScratchPad в действии

Контекстные меню

В C++Builder вы можете создавать контекстные меню с такой же легкостью, как и главное меню. C++Builder позволяет сопоставить каждому компоненту свое контекстное меню с помощью свойства PopupMenu. При нажатии правой кнопки мыши на этом компоненте на экране будет автоматически отображено соответствующее меню. Обработчики для пунктов контекстного меню пишутся так же, как и для пунктов главного меню.

В программах обработки текстов принято дублировать операции Cut, Copy и Paste в контекстном меню. Это следует сделать и в программе ScratchPad. Для создания контекстного меню мы немного схитрим и скопируем часть главного меню. Итак, приступаем:

1. Выберите компонент PopupMenu из палитры компонентов и поместите его в форму.
2. Измените значение свойства Name на MemoPopup.
3. Дважды щелкните на значке PopupMenu для запуска редактора меню.
4. Щелкните правой кнопкой мыши для вызова контекстного меню редактора. Выберите в нем пункт Select Menu. В открывшемся диалоговом окне будет показан список меню, доступных вашему приложению. Выберите MainMenu и нажмите кнопку OK.
5. Выберите пункт Edit. Щелкните на пункте Cut, затем, удерживая нажатой клавишу Shift, щелкните на пункте Paste. Cut, Copy и Paste будут выделены.

6. Для копирования выбранных пунктов в буфер обмена, выберите в главном меню C++Builder пункт Edit | Copy (не трогайте пункт Edit | Copy в меню, загруженном в редактор) или нажмите Ctrl+C.
7. Снова выберите пункт Select Menu в контекстном меню редактора. На этот раз щелкните на MemoPopup и нажмите OK. Редактор меню отобразит пустое контекстное меню.
8. Выберите в главном меню пункт Edit | Paste или нажмите Ctrl+V. Пункты Cut, Copy и Paste будут вставлены в контекстное меню.

Теперь еще немного, и мы закончим. Нужно изменить значения свойств Name новых пунктов меню:

1. Для пункта Cut измените значение Name на PopupCut.
2. Для пункта Copy измените значение Name на PopupCopy.
3. Наконец, измените значение Name для пункта Paste на PopupPaste.

Осталось написать обработчики для пунктов контекстного меню. Но у нас уже есть код для пунктов Cut, Copy и Paste главного меню. Было бы досадно его дублировать (даже если в каждом случае это всего одна строка). Хорошо бы использовать обработчики, созданные раньше. Но возможно ли это? Конечно! Нужно только сделать следующее:

1. Щелкните на пункте меню Cut.
2. Щелкните на вкладке Events инспектора объектов.
3. Щелкните на кнопке со стрелкой в правой части столбца Value рядом с событием OnClick (это единственное событие в списке). Вы увидите список обработчиков, созданных на данный момент.
4. Выберите из списка обработчик EditCutClick. Теперь при щелчке на пункте Cut контекстного меню будет вызван обработчик для пункта Edit | Cut главного меню. Никакого дублирования кода не потребовалось.
5. Повторите шаги 1–4 для пунктов Copy и Paste контекстного меню. Закончив, закройте редактор меню.
6. Щелкните на компоненте Memo в главной форме. Измените значение свойства PopupMenu на MemoPopup (выбрав его из списка).

Таким способом вы можете связать любое событие с любым обработчиком этого события. Запустите программу снова и проверьте работу контекстного меню.



Вы можете связать событие OnClick кнопки быстрого доступа с существующим обработчиком точно так же, как это было сделано для контекстного меню. Щелкните на кнопке File Open в вашей форме. Найдите событие OnClick в инспекторе объектов и установите для него обработчик FileOpenClick. Повторите эти действия для кнопки File Save, установив обработчик FileSaveClick.

Создание и сохранение шаблонов меню

C++Builder предоставляет вам несколько шаблонов меню, которые можно вставлять в главные и контекстные меню. Вы можете также сохранять в качестве шаблонов свои собственные меню, чтобы затем использовать их в

других программах. Для начала, запустите редактор меню и создайте новое меню.



Чтобы создать шаблон, вам нужно сначала иметь в форме главное или контекстное меню для запуска редактора меню. Вы можете при желании использовать временную (пустую) форму. Создайте новую форму, поместите в нее компонент MainMenu и дважды щелкните на пиктограмме этого компонента, чтобы запустить редактор меню. Создав шаблон, закройте пустую форму без сохранения.

После создания меню выберите в контекстном меню редактора пункт Save As. Откроется диалоговое окно Save Template. Дайте созданному меню информативное имя и нажмите кнопку OK. Меню будет сохранено как шаблон. Для вставки меню выберите пункт Insert From Template в контекстном меню редактора. Ваши шаблоны будут показаны вместе со стандартными шаблонами C++Builder.

Для удаления ранее созданного шаблона, выберите в контекстном меню редактора пункт Delete Templates. Откроется диалоговое окно Delete Templates и вы сможете выбрать шаблон, подлежащий удалению. После щелчка на кнопке OK выбранный шаблон будет удален. Чтобы закрыть диалоговое окно, не удаляя шаблон, нажмите кнопку Cancel.

Заключение

Поздравляю! Вы только что завершили знакомство с основами визуального программирования в среде C++Builder. Это было и интересно, и познавательно. Редактор форм представляет собой мощный инструмент, позволяющий выполнять максимально возможное количество работы визуально. Если вам не приходилось размещать элементы управления в окне, программируя на Си или Си++, вы можете и не оценить всех достоинств визуального подхода к программированию. Поверьте мне, они очень значительны. Редактор меню также является мощным инструментом, в частности, благодаря возможности импорта меню, что делает их создание легким и приятным. Другая возможность, которую он предоставляет — моментальное обновление существующих меню.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении A.

Вопросы и ответы

Я активно использую палитру выравнивания, но каждый раз, когда приходится переключаться от редактора кода к редактору форм, палитра теряется из виду. Можно ли с этим что-то сделать?

Найдите палитру выравнивания и щелкните правой кнопкой мыши для вызова контекстного меню палитры. Выберите в нем пункт Stay on Top. Теперь палитра выравнивания всегда будет располагаться поверх остальных окон.

Я пытаюсь выделить группу компонентов на панели, заключая их в рамку, но это приводит к перемещению самой панели. Что я делаю неправильно?

Вам нужно удерживать нажатой клавишу Ctrl во время выделения компонентов, расположенных на панели.

Я перемещал компоненты в пределах формы несколько раз и теперь порядок их переключения стал хаотическим. Как можно исправить ситуацию?

Выберите в контекстном меню редактора форм пункт Tab Order. Установите нужный порядок переключения. После нажатия кнопки OK он вступит в силу.

Шаблоны меню, содержащиеся в C++Builder, сделаны отлично, но в них очень много пунктов, которые мне не нужны. Что с этим можно сделать?

Вы можете сделать две вещи. Во-первых, можно импортировать меню и затем просто удалить ненужные пункты. Используя для выделения мышь и клавишу Shift, вы можете избавиться от них в считанные секунды. Удаление пунктов меню, вставленного из шаблона, не влечет за собой никаких нежелательных последствий. Во-вторых, вы можете сохранить измененное меню в качестве нового шаблона. Тем самым вы сохраните существующий шаблон C++Builder и будете иметь свой собственный шаблон.

Можно ли сохранять свои меню как шаблоны?

Да. Сначала создайте меню, а затем выберите в контекстном меню редактора меню пункт Save As Template. Дайте шаблону имя, нажмите OK, и он будет сохранен. Для последующего использования меню достаточно вставить его с помощью пункта Insert From Menu.

Вопросы

1. Когда нужно использовать клавишу Ctrl для выделения компонентов?
2. Какую роль играет первый выделенный компонент при выравнивании группы компонентов?
3. Каков самый быстрый способ выделения группы компонентов?
4. Как можно сделать ширину всех компонентов группы равной ширине самого широкого компонента?
5. Что происходит, когда вы дважды щелкаете на компоненте в форме?
6. Что делает опция alClient свойства Align?
7. Что означает многоточие, следующее за названием пункта меню?
8. Какими двумя способами можно переместить пункт меню?
9. Как сопоставить пункту меню определенное сочетание клавиш?
10. Как изначально запретить пункт меню?

Упражнения

1. Поместите в форму пять компонентов редактирования и расположите их один над другим, выровняв левые края.

2. Выключите режим привязки к сетке (пункт Tools | Options главного меню). Поместите в форму пять любых компонентов и выровняйте их правые края.
3. Поместите компонент ListBox в пустую форму и измените его так, чтобы он постоянно занимал всю рабочую область формы.
4. Добавьте к программе ScratchPad окно About. Используйте палитру выравнивания для быстрого выравнивания текста.
5. Добавьте пункт Undo и разделитель в контекстное меню программы ScratchPad.
6. Создайте новое приложение. Поместите шесть компонентов редактирования в разных местах формы. Теперь измените порядок переключения так, чтобы переход между компонентами выполнялся сверху вниз. Запустите программу и проверьте порядок переключения.
7. Добавьте в панель инструментов программы ScratchPad кнопки Cut, Copy и Paste, а также любые другие по своему желанию. Свяжите с этими кнопками обработчики событий для соответствующих пунктов меню.
8. Добавьте сочетание Ctrl+S для пункта меню File | Save в программе ScratchPad.
9. Откройте проект Picture Viewer, созданный в шестой день, «Изучение IDE C++Builder: проекты и формы». Удалите из меню все неиспользуемые пункты.

Подведение итогов



За эту неделю вы прошли большое количество базового материала. В некотором смысле это была самая трудная часть книги. Си++ нелегко изучать. Но если вы не бросили это дело, то можно с уверенностью сказать, что вы сможете стать программистом на Си++. Не забывайте делать перерывы, как сейчас, так и в дальнейшем. В названии книги предполагается, что вы освоите Borland C++Builder за 21 день, но эти дни не обязаны идти непрерывно! Иногда неплохо отдохнуть несколько дней, чтобы дать новым знаниям устояться.

Если вы путаетесь в синтаксисе Си++, не думайте, что вы одиноки. Указатели и ссылки, * и &, операторы прямого и косвенного доступа... это вполне может поначалу запутать. Но не беспокойтесь: по мере приобретения опыта работы с C++Builder все встанет на свои места. Чего вам действительно не хватает на данном этапе, так это реальных задач. Знания, закрепленные на практике, остаются надолго. Я советую вам придумать и написать какую-нибудь программу. Это не обязательно делать прямо сейчас, но так вы быстрее начнете приобретать необходимый опыт. Вовсе не обязательно писать сразу Word, Netscape Navigator или Excel. Вам нужна небольшая программа, которая поможет связать теорию с практикой.

Первая часть этой недели была посвящена ключевым словам и синтаксису Си++. Конструкции типа циклов и операторов if достаточно легко освоить. Однако не беспокойтесь, если вам придется не один раз возвращаться назад и смотреть на синтаксис. Объем материала довольно большой, поэтому не рассчитывайте запомнить сразу все ключевые слова и правила их использования. Со временем вы выучите их; но, подчеркиваю, на данном этапе это не предполагается.

Ближе к середине недели вы узнали о структурах, а затем и о классах. Классы составляют основу Си++. Те свойства языка, которые рассматривались в первые три дня, перешли в основном из Си. Классы же относятся к «чистому» Си++. Порой вам придется некоторое время соображать, как можно использовать классы в своих программах. Длительное время вы имели дело только с готовыми классами VCL и не пробовали создавать классы самостоятельно. Со временем, вероятно, вы встретитесь с задачами, для решения которых классы подходят лучше всего. К этому моменту вы будете

готовы к написанию своих собственных классов. Вам будет достаточно создать один или два класса, чтобы привыкнуть к работе с ними.

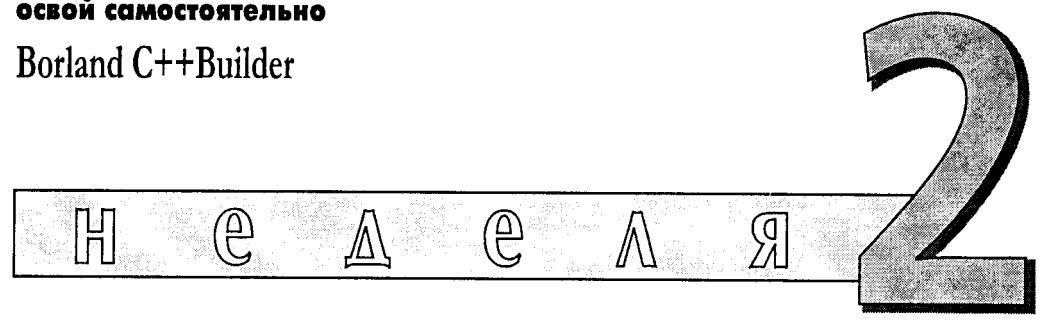
На пятый день вы в общих чертах познакомились с библиотеками классов. Хорошим примером библиотеки классов является VCL. Библиотеки облегчают вашу жизнь, инкапсулируя сложные функции Windows в классах, которые можно использовать достаточно осмысленно. Поверьте мне, иногда API Windows выглядит совершенно иррационально. VCL берет на себя заботу о низкоуровневых деталях, предоставляя в ваше распоряжение объекты более высокого уровня, которые можно с легкостью встраивать в приложения. Хотя VCL тоже достаточно сложна, работать с ней гораздо проще, чем с API Windows. Обсуждая библиотеки, мы рассмотрели модель PME (Properties, Methods and Events). Вы узнали, что такое свойства, методы и события, и как они используются при создании Windows-программ в C++Builder.

В конце первой недели мы подробнее остановились на самой IDE C++Builder. Вы узнали, как настраивать среду по своему вкусу, как использовать палитру компонентов, для чего нужен инспектор объектов и как работать с редактором меню. Вы начали знакомиться с тем, что доставляет удовольствие. Именно так: я считаю программирование во всех его видах большим удовольствием. Именно поэтому я занимаюсь им. Я надеюсь, что вам оно тоже понравится.

Наконец, мы закончили эту неделю подробным изучением редактора форм. Редактор форм представляет собой инструмент, с помощью которого создается основа приложений C++Builder — по крайней мере, их графическая часть. Используя редактор форм, вы можете создавать великолепно выглядящие формы. Вспомните, что форма представляет в вашем приложении окно. Большинство приложений имеет главное окно и несколько диалоговых окон, которые отображаются в процессе взаимодействия пользователя с программой. Достоинства редактора форм нельзя оценить, не попробовав разработать диалоговое окно с помощью традиционных инструментальных программных средств. Даже редакторы ресурсов, поставляемые вместе с Borland C++ и Visual C++, не могут сравниться с C++Builder при разработке форм. Возможность размещать компоненты в формах и изменять их свойства на этапе разработки дает вам ощутимое преимущество, когда дело доходит до рыночной конкуренции. Модель визуального программирования, используемая в C++Builder, мощна и легка в использовании. Что может быть лучше этого?

На седьмой день вы создали простую, но полезную программу — ScratchPad. На примере этой программы вы начали знакомство с разработкой полноценных приложений с помощью C++Builder. Мы собираемся использовать ScratchPad и дальше, на протяжении всей книги. По мере возрастания ваших знаний мы будем расширять возможности этой программы. Если вы будете разрабатывать свое собственное приложение, я призываю вас также добавлять к нему новые возможности по мере их изучения.

Я надеюсь, что вы не очень устали за эту неделю. Отдохните, если нужно, и с новыми силами принимайтесь за дело. Если же эта неделя, наоборот, подбодрила вас и зарядила энергией, продолжайте переворачивать страницы! Я готов повести вас дальше.



Краткий обзор

На этой неделе вам предстоит всерьез приступить к изучению программирования в Windows. Мы начнем с обсуждения компонентов, о которых вы узнаете много нового, помимо того, что вам стало известно из первой недели. Вы познакомитесь со специализированными компонентами и узнаете, как их нужно использовать. Вы потратите некоторое время на чтение и, как я надеюсь, много времени на эксперименты. Не забывайте, что чтение этой книги — не гонка, и пришедший к финишу первым приза не получит. Поэтому при изучении программирования вам лучше быть медлительной черепахой, чем зайцем. Обязательно оставляйте время для экспериментов!



На этой неделе вы научитесь создавать приложения, используя эксперты C++Buider. Эти эксперты помогут вам получить работающую программу в минимально короткое время. После этого вы приступите к изучению техники отладки ваших программ. Ваши программы неизбежно будут содержать ошибки. Но не стоит переживать по этому поводу. Вам нужно научиться находить этих противных «жучков» в своих программах и «давить» их. Отладчик абсолютно необходим при разработке программ, и вы должны уметь им пользоваться. В конечном счете, применение отладчика сохранит вам многие часы вашего времени.







Наконец, в завершение недели, вы познакомитесь с более сложной техникой программирования. Вы узнаете, как работать со строками состояния, кнопками быстрого доступа и печатью. В 14-й день вы узнаете, как организовать контекстную справку и как использовать системный реестр Windows, чтобы держать в нем необходимую для вашей программы информацию. Я думаю, что вам понравится материал этой недели и к ее концу вы наберете такой темп, что вас уже ничто не остановит!







Компоненты VCL



Кент Рейсдорф

Как вы уже знаете, именно компоненты обеспечивают мощь C++Builder. Компоненты разработаны на основе модели свойств, методов и событий. Используя редактор форм, вы можете поместить компонент в форму и изменять его свойства, доступные на этапе разработки. В некоторых случаях этого будет достаточно. При необходимости вы можете управлять компонентом во время выполнения программы, изменяя его свойства и вызывая методы. Далее, каждый компонент реагирует на определенные события. Я рассматривал свойства, методы и события в 5-й день, «Библиотеки классов Си++ и модель визуальных компонентов», и мы не будем повторять это еще раз.

Сегодня вы узнаете о компонентах намного больше. Вы познакомитесь с часто используемыми компонентами и узнаете о классах VCL, представляющих эти компоненты. Если при чтении материала у вас будет возникать желание что-либо проверить, обязательно делайте это. Применение на практике полученных знаний — самый лучший способ обучения, поэтому не стесняйтесь экспериментировать.

Обзор

Давайте вспомним то, что вы уже знаете о компонентах. Но сначала, пользуясь моментом, я хочу остановиться на различиях между компонентами VCL и элементами управления Windows. К элементам управления Windows относятся элементы редактирования, окна списков, комбинированные списки, статические элементы управления (метки) и кнопки, не говоря уже о всех элементах управления Windows 95. Элементы управления Windows, по своей природе, не имеют свойств, методов и событий. Они используют механизм сообщений, указывающих элементу, что нужно делать, или передающих информацию от элемента. Иметь дело с элементами управления на таком уровне — занятие утомительное и тяжелое.

Компонент VCL — это класс, инкапсулирующий элементы управления Windows (хотя не во всех компонентах VCL инкапсулированы элементы управления). В компоненте VCL к элементам управления Windows добавляются свойства, методы и события, которые позволяют сильно упростить работу. Можно сказать, что VCL предлагает новый подход к работе с элементами управления Windows. Можно также сказать, что все компоненты VCL — это

элементы управления, но не все элементы управления являются компонентами. Например, компонент VCL Edit представляет собой элемент управления, но стандартный элемент редактирования в Windows не является компонентом. Итак, благодаря использованию компонентов VCL вы можете поднять работу с элементами управления Windows на более высокий уровень.

В завершение отмечу, что для обозначения компонентов VCL я буду на равных использовать термины *элемент управления* и *компонент* (Но я никогда не назову элемент управления Windows компонентом!)

Визуальные и невизуальные компоненты

Некоторые компоненты являются визуальными, другие — невизуальными.



Визуальный компонент (visual component), как следует из названия, — это такой компонент, который виден пользователю во время разработки программы.

К визуальным компонентам относятся элементы редактирования, кнопки, списки, метки и т.д. Большинство тех компонентов, которые вам придется использовать в приложениях C++Builder, являются визуальными. При разработке программы визуальные компоненты выглядят так, как они будут выглядеть во время ее выполнения.



Невизуальный компонент (nonvisual component) — это компонент, который не может быть виден пользователю в процессе разработки программы.

Невизуальные компоненты работают «за кулисами» и выполняют специфические задачи. Примеры таких компонентов: системные таймеры, компоненты для работы с базами данных и списки изображений. Стандартные диалоговые окна, такие как File Open, File Save, Font и другие, также представлены невизуальными компонентами. Они не видны на этапе разработки и отображаются только во время выполнения программы. Когда вы помещаете невизуальный компонент в форму, C++Builder показывает его в виде значка. Этот значок используется только для изменения свойств компонента на этапе разработки и не выводится на экран при выполнении программы. Невизуальные компоненты, как и визуальные, имеют свойства, методы и события.

Свойство Name

Свойство Name имеет для компонентов жизненно важное значение. Как упоминалось ранее, при помещении компонента в форму C++Builder переходит в фоновый режим, пока вы обдумываете свое следующее действие. Все, что C++Builder делает при этом — создает указатель на компонент и использует свойство Name в качестве имени переменной. Предположим, например, что вы поместили компонент Edit в форму и изменили его свойство Name на MyEdit. В этом случае C++Builder поместит следующую строку в заголовочный файл формы:

```
TEdit* MyEdit;
```

После запуска приложения C++Builder создаст экземпляр класса TEdit, на который будет ссылаться указатель MyEdit. Вы можете использовать этот указатель для обращения к компоненту во время выполнения программы. Чтобы ввести текст в элемент редактирования, вам нужно сделать следующее:

```
MyEdit->Text = "Jenna Lynn";
```

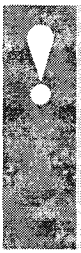
C++Builder использует значение свойства Name также для образования имен обработчиков событий. Допустим, вы хотите обрабатывать событие OnChange компонента Edit. Обычно вы дважды щелкаете в столбце Value рядом с событием OnChange, чтобы C++Builder сгенерировал обработчик события. C++Builder будет использовать для этой функции имя по умолчанию, основанное на значении свойства Name и имени обрабатываемого события. В нашем случае C++Builder создаст функцию с именем MyEditChange().

Вы можете изменять свойство Name в любое время, но только при условии, что вы делаете это через инспектор объектов. Когда при разработке программы вы изменяете свойство Name некоторого компонента, C++Builder просматривает весь код, который он до этого создал, и изменяет имя указателя и имена всех функций-обработчиков событий.



При изменении свойства Name компонента, C++Builder внесет соответствующие изменения в код, сгенерированный им самим, но оставит без изменения ваш текст. О своем коде вы должны заботиться сами. Поэтому изменять свойство Name следует только при первоначальном размещении компонента в форме.

Если в этом же примере вы измените свойство Name элемента редактирования со значения MyEdit на FirstName, то C++Builder даст указателю имя FirstName, а обработчик события OnChange получит имя FirstNameChange(). Все это будет выполнено автоматически; вам нужно только изменить свойство Name и положиться в остальном на C++Builder.



Ни в коем случае не изменяйте свойство Name во время выполнения программы. Никогда не изменяйте вручную имя компонента (то имя, которое C++Builder присваивает указателю на компонент) или имена обработчиков событий в окне редактора кода. Если вы выполните любое из этих действий, то C++Builder потеряет контроль над компонентом, и результат, мягко говоря, будет неудовлетворительным. Возможно, вам даже не удастся загрузить форму. Единственный безопасный способ изменения свойства Name компонента — использовать для этой цели инспектор объектов.

Для каждого компонента, помещаемого в форму, C++Builder устанавливает значение свойства Name по умолчанию. Если вы, например, размещаете компонент Edit, то C++Builder присвоит свойству Name значение Edit1. Если вы поместите в форму второй компонент Edit, он получит имя Edit2, и т. д. Вам следует как можно скорее давать компонентам осмысленные имена, чтобы избежать впоследствии путаницы и лишней работы.



Вы можете оставить имена по умолчанию для тех компонентов, к которым никогда не будете обращаться в программе. Например, если у вас есть несколько компонентов label, которые содержат статический (неизменяемый) текст, то нет необходимости менять их имена, поскольку вам не придется обращаться к этим компонентам во время выполнения программы.

Правила работы со свойством Name

- Изменяйте установленное по умолчанию значение свойства Name на значимое, причем делайте это как можно раньше.
- Компонентам, к которым не происходит обращение во время выполнения программы, можно оставлять имена по умолчанию.
- Ни в коем случае не изменяйте свойство Name в тексте программы или во время ее выполнения.
- Давайте вашим компонентам осмысленные имена, но не делайте их чрезмерно длинными.

Важные общие свойства

Все компоненты имеют некоторые общие свойства. Например, все визуальные компоненты имеют свойства Left и Top, определяющие положение компонента в форме. Свойства типа Left, Top, Height и Width говорят сами за себя, так что я не буду на них останавливаться. Но некоторым общим свойствам следует уделить более пристальное внимание.

Свойство Align

В 7-й день, «Работа с редактором форм и редактором меню», я уже рассматривал свойства Align и Alignment, поэтому я не буду снова вдаваться в детали. Обращайтесь к этой главе за необходимой информацией относительно свойства Align. Нужно только отметить, что не все компоненты отображают свойство Align во время разработки программы. Например, однострочный элемент редактирования имеет стандартную высоту, поэтому для компонентов такого типа свойство Align не имеет смысла. По мере приобретения опыта работы с C++Builder и в зависимости от типа разрабатываемых приложений, вам, возможно, придется часто использовать это свойство.

Свойство Color

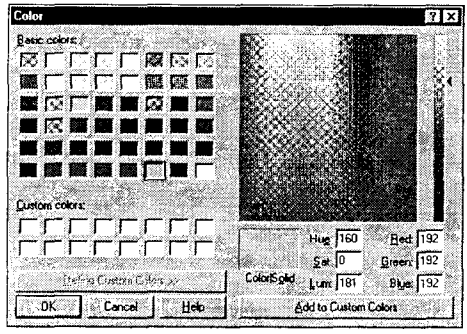
Свойство Color определяет цвет фона компонента. (Цвет текста устанавливается свойством Font). Хотя использование свойства Color очень просто, здесь есть свои особенности, о которых следует сказать.

Порядок работы со свойством Color в инспекторе объектов в некотором смысле уникален. Если вы щелкнете в столбце Value, то увидите кнопку со стрелкой вниз, указывающую на возможность выбора цвета из списка. Но кроме этой возможности, есть еще одна. Если вы щелкнете дважды в том же столбце Value, то на экране появится диалоговое окно Color (Цвет). В этом диалоговом окне вы можете выбрать один из предопределенных цветов или создать ваши собственные цвета, нажав кнопку Define Custom Colors (Определить заказные цвета). На рис. 8.1 показано диалоговое окно Color после нажатия этой кнопки.

То же самое диалоговое окно Color будет отображаться при использовании в приложении компонента ColorDialog.

После выбора цвета в диалоговом окне Color вы увидите, что значение свойства Color изменилось на шестнадцатеричную строку. Эта строка представляет интенсивности красного, зеленого и синего цветов в цветовой модели RGB, необходимых для составления вашего цвета. Если вы знаете точное соотношение R:G:B вашего цвета (что маловероятно), вы можете задать это значение напрямую.

Рис. 8.1
Диалоговое окно Color



В большинстве случаев вы, вероятно, будете выбирать цвет из предлагаемого списка. Когда вы нажмете кнопку со стрелкой, то в появившемся окне увидите две группы значений. Первая группа цветов начинается с clBlack и заканчивается clWhite. Это предопределенные цвета C++Builder; в списке представлены наиболее употребительные из них. Чтобы выбрать один из перечисленных цветов, просто щелкните на нем в списке. Если вас не устраивает ни один предложенный цвет, вы можете вызвать диалоговое окно Color, как говорилось выше.

Вторая группа цветов в списке начинается с цвета clScrollBar. Эта группа содержит системные цвета Windows. Если вы используете цвета из этого списка, ваше приложение будет автоматически менять цвета при изменении цветовой схемы Windows. Если вы хотите, чтобы ваше приложение следовало за цветовой схемой, выбранной пользователем для своей системы, вы должны выбирать цвета только из этой части списка.

К выбору цвета нужно подходить осторожно. Правильно подобранные цвета создают комфортную среду для пользователя. Злоупотребление цветом может сделать программу неприятной в использовании, такая программа будет вызывать раздражение. На начинающих программистов цвет обычно производит магнетическое воздействие. Им так и хочется разбросать по форме большое количество цветов, потому что это забавно и очень легко. Но не стоит развлекаться за счет ваших пользователей.

Правила работы с цветом

- Используйте цвета для акцентирования внимания и выделения.
- Не используйте кричащие цвета — от них устают глаза.
- По возможности используйте в своей программе системные цвета. Если пользователь будет изменять цветовую схему, ваше приложение будет отслеживать эти изменения.
- Будьте последовательны в цветовом оформлении всех ваших форм.

Курсоры

Свойство Cursor определяет вид курсора мыши во время его перемещения через компонент. Для некоторых компонентов Windows автоматически изменяет курсор. Например, Windows изменяет курсор на вертикальный штрих «I», когда он расположен над компонентами Edit, Memo или RichEdit (я перечислил лишь некоторые). Если вы хотите оставить управление курсором за Windows, то не изменяйте значение этого свойства, установленное по умолчанию — crDefault. Если у вас есть специализированные окна (или компоненты), то вы можете определить для них и специальный курсор. Когда курсор мыши окажется над этим компонентом, Windows изменит вид курсора на заданный вами.

Часто требуется изменять курсоры во время выполнения программы. Длительный процесс, например, может быть обозначен для пользователя курсором в виде песочных часов. Затем вы должны придать курсору первоначальный вид. Следующий фрагмент кода демонстрирует, как это можно сделать:

```
TCursor oldCursor = Cursor;
Cursor = TCursor(crHourGlass);
// некоторая длительная операция
Cursor = oldCursor;
```

Этот код гарантирует, что курсор, первоначально использованный в приложении, будет восстановлен правильно.

Другое свойство, DragCursor, определяет курсор, который используется при перетаскивании из компонента в том случае, когда адресат может принять перетаскиваемый объект. Как и в случае с цветами, вы должны проявлять благоразумие при использовании курсоров. Используйте свои курсоры, если это необходимо, но не увлекайтесь этим.

Свойство Enabled

Доступ к компонентам можно разрешить или запретить, используя свойство Enabled. Когда компонент недоступен, он не может получить фокус ввода (щелчок на таком объекте не дает никакого эффекта). Для обозначения такого компонента обычно применяется некоторый визуальный эффект. В случае кнопки, например, ее текст выделяется серым цветом, как если бы на кнопку был наложен растр. Свойство Enabled имеет логический тип и может принимать только два значения: true, если компонент доступен, и false в противном случае. Разрешение и запрещение доступа к окнам (не забывайте, что компоненты оконного типа также являются окнами) — это возможность, предоставляемая самой системой Windows.



Некоторые компоненты (например, BitBtn) показывают недоступное состояние во время разработки программы, но большинство — нет.

Свойство Enabled применяется обычно к компонентам оконного типа, но может использоваться и для других компонентов. Компонент SpeedButton служит примером компонента не оконного типа, который, тем не менее, может быть заблокирован.

Изменение свойства Enabled для компонента Panel имеет некоторые дополнительные следствия. Панели часто используются как контейнеры для других элементов управления. Следовательно, они становятся предками помещенных в них компонентов. Если вы отключите такую панель, то внешний вид компонентов на панели останется прежним, но функционировать не будут, поскольку доступ к их предку (панели) запрещен.

Хотя доступ к компонентам может быть запрещен на этапе разработки, операция разрешения/запрещения доступа обычно выполняется во время работы программы. Пункты меню, например, могут быть включены или выключены в зависимости от того, могут ли они выполняться в данный момент. То же самое относится и к кнопкам. Бывают и другие причины, по которым вы можете отключать те или иные типы элементов управления.

Чтобы запретить доступ к компоненту во время выполнения программы, нужно присвоить значение false его свойству Enabled, а чтобы сделать его доступным, это свойство должно получить значение true. Следующий фрагмент кода включает или выключает пункт меню в зависимости от некоторого условия:

```
if (saveEnabled) FileSave->Enabled = true;
else FileSave->Enabled = false;
```

Этот процесс часто называется *управлением доступом к командам (command enabling)* и является важной частью профессиональных программ для Windows.

Свойство Font

Свойство Font (Шрифт) — одно из основных свойств и, следовательно, должно быть рассмотрено здесь, хотя о нем и не требуется много говорить. Свойство Font — экземпляр класса TFont и имеет собственные свойства. Вы можете определить свойства для Font, дважды щелкнув на имени шрифта в инспекторе объектов (после этого узел Font раскроется, и вы увидите список его вложенных свойств), или вызвав диалоговое окно Font. (Диалоговое окно Font будет подробно рассмотрено далее, в разделе «Диалоговое окно Font».) На рис. 8.2 показано окно инспектора объектов со свойством Font, раскрытым для отображения свойств класса TFont.

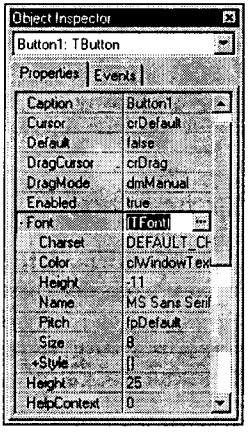


Рис. 8.2
Инспектор объектов с раскрытым свойством Font

Свойство Color определяет цвет шрифта, а свойство Name позволяет вам выбрать гарнитуру шрифта.

Свойства Height (Высота) и Size (Размер) класса TFont заслуживают специального упоминания. Свойство Height используется для определения высоты шрифта в пикселах, в то время как свойство Size определяет высоту шрифта в пунктах. Когда вы изменяете одно из этих свойств, другое изменяется автоматически. Свойство Height часто задается отрицательным числом. Обратившись к встроенной справке по TFont, вы можете узнать, почему это делается именно так.

Свойство Pitch (Шаг) не представляет большой пользы. Я расскажу о нем чуть позже, а сначала мы кратко поговорим о шрифтах. Шрифт может быть пропорциональным или моноширинным (с фиксированной шириной символов). Большинство шрифтов являются пропорциональными. Это означает, что для изображения каждого символа отводится столько места, сколько действительно необходимо. Например, для изображения символа M в верхнем регистре требуется намного больше места, чем для символа i в нижнем регистре. Посмотрите на буквы в этой книге, и вам станет ясно, что я имею в виду. Пропорциональными являются такие шрифты, как Times New Roman, Arial и Bookman. У моноширинного шрифта, наоборот, все символы имеют одинаковую ширину. Такие шрифты удобно использовать в редакторах кода (например, в редакторе кода самого C++Builder) или в любых других окнах, где желательно, чтобы шрифт имел фиксированную ширину. Вероятно, наиболее используемый из таких шрифтов — это Courier New, хотя для Windows выбран шрифт Fixedsys.

Теоретически, свойство Pitch может использоваться для принудительной замены пропорционального шрифта на моноширинный и наоборот. Но проблема состоит в том, что Windows в этом случае может изменить гарнитуру шрифта, и вы не можете точно сказать, что получится в результате. Гораздо надежнее точно выбирать шрифт, который вам требуется, чем полагаться на свойство Pitch.

Наконец, свойство Style (Стиль) класса TFont может использоваться для установки полужирного, курсивного, подчеркнутого или перечеркнутого шрифта. Эти стили не исключают друг друга, так что вы можете смешивать их по своему выбору.



Для изменения свойств шрифта вы можете использовать инспектор объектов, однако диалоговое окно Font имеет одно преимущество: в нем можно видеть, как будет изменяться шрифт по мере выбора вами различных шрифтовых опций. Чтобы просто изменить, например, свойство Style или Size, используйте инспектор объектов. Но если вы занимаетесь подбором нужного шрифта, то диалоговое окно Font — лучший выбор.

Подсказки

Свойство Hint (Подсказка) задает текст подсказки для компонента. Текст подсказки состоит из двух частей. Первая часть иногда называется *короткой подсказкой (short hint)*. Именно этот текст выводится, когда пользователь задерживает курсор над компонентом. Всплывающее окно, в котором отображается текст подсказки, называется *всплывающей подсказкой (tool tip)*.

Вторую часть текста подсказки называют *длинной подсказкой (long hint)*. Длинная подсказка — это необязательный текст, который выводится в строке состояния, когда пользователь устанавливает курсор мыши на компоненте. Тексты короткой и длинной подсказок отделяются друг от друга символом «|». Например, чтобы определить как короткую, так и длинную подсказки для кнопки File Open, вы можете ввести следующий текст:

```
Открыть файл|Открыть файл для редактирования
```

Чтобы выводить краткие подсказки, вы должны установить для свойства ShowHint объекта Application значение true (значение по умолчанию) и присвоить это же значение свойству ShowHint компонента. Вывод полной подсказки в строке состояния требует немного больше работы, и я отложу этот вопрос до завтра.



Вы можете задать только короткую подсказку, только длинную или оба варианта. Символ «|» служит указателем на то, какую подсказку вы вводите. Если вы не используете этот символ, то введенный текст будет использоваться как для короткой, так и для длинной подсказки.

Свойства ParentColor, ParentCtl3D, ParentFont и ParentShowHint

Свойства ParentColor, ParentCtl3D, ParentFont и ParentShowHint работают одинаковым образом, поэтому я буду говорить о них одновременно. Когда эти свойства имеют значение true, компонент наследует свойства Color, Ctl3D, Font и ShowHint от своего предка. Например, для большинства компонентов свойство ParentFont имеет по умолчанию значение true. Это означает, что компонент наследует шрифт, используемый в настоящее время компонентом-предком. В качестве иллюстрации выполните следующее упражнение.

1. Создайте пустую форму. Установите в свойстве Font значение 16 для свойства Size.
2. Поместите в форму компонент Label. Обратите внимание, что в тексте автоматически используется шрифт в 16 пунктов.
3. Поместите в форму компонент Button. Он также будет использовать шрифт в 16 пунктов.

Вы, конечно, можете установить для этого свойства значение false, но это возможно только после помещения компонента в форму, когда уже слишком поздно и все равно нужно менять шрифт вручную.

Свойство Tag

Свойство Tag представляет собой переменную размером в 4 байта, зарезервированную для возможного использования. Вы можете использовать свойство Tag для хранения любых данных, нужных для работы вашего компонента. Это может быть указатель на другой класс, индексное значение или что-то еще. Использование свойства Tag, вероятно, должно рассматриваться при углубленном изучении программирования.

Другие общие свойства

В таблице 8.1 перечислены другие часто используемые свойства. Эти свойства не требуют пояснений, поэтому я только перечислю их для справки. Не все компоненты имеют каждое из перечисленных здесь свойств.

Таблица 8.1. Дополнительные свойства компонентов

Свойство	Описание
BorderStyle	Принимает значения bsSingle или bsNone. Используйте bsNone, если вы хотите, чтобы компонент сливался с фоном.
Caption	Задает надпись на компоненте. Многие компоненты не имеют надписи и для них свойство Caption не задано.
Ctl3D	Указывает, должен ли элемент управления выделяться объемной рамкой. Если для свойства BorderStyle установлено значение bsNone, то значение данного свойства не существенно.
Height	Определяет высоту компонента.
HelpContext	Это свойство используется для связывания индекса в справочном файле с конкретным компонентом.
Left	Определяет x-координату компонента.
PopupMenu	Определяет контекстное (всплывающее) меню, отображаемое при нажатии правой кнопки мыши.
TabOrder	Применяется для компонентов оконного типа. Устанавливает очередь этого компонента при переключении клавишей Tab.
TabStop	Применяется для компонентов оконного типа. Указывает, что на данный компонент можно переключиться клавишей Tab. Если это свойство имеет значение false, то компонент исключается из последовательности переключения.
Top	Определяет y-координату компонента.
Visible	При чтении указывает, является ли компонент в настоящее время видимым. При записи значения в Visible компонент может быть сделан видимым или скрытым.
Width	Определяет ширину компонента.

Основные методы компонентов

Насчитывается более 20 методов, которые используются большинством компонентов. Компоненты оконного типа имеют более 40 общих методов. Однако не все из них используются широко. Многие функциональные возможности компонентов реализуются при помощи свойств. Например, чтобы скрыть компонент, вы можете вызвать метод Hide() или установить для свойства Visible значение false. Кроме того, компоненты обычно имеют свои специфические методы, и именно эти методы для данного компонента используются чаще всего.

Некоторые из методов, перечисленных в таблице 8.2, достойны отдельного внимания. Заметьте, что некоторые из них доступны не всем компонентам. Здесь собраны не общие для всех компонентов методы, которые

часто используются, а просто часто используемые методы. Кроме того, список относится в основном к компонентам, представляющим элементы управления (т.е. компонентам, помещаемым в формы), а не к компонентам-формам. Применение методов форм обсуждалось нами в 6-й день, «Изучение IDE C++Builder: проекты и формы».

Таблица 8.2. Общие методы компонентов

Метод	Описание
Broadcast	Используется для отправки сообщений всем порожденным компонентам оконного типа.
ClientToScreen	Преобразует локальные координаты окна в экранные координаты.
ContainsControl	Возвращает true, если указанный компонент порожден компонентом или формой.
HandleAllocated	Возвращает true, если для компонента было создано свойство Handle. Простое чтение свойства Handle автоматически создает дескриптор, если он еще не существовал, поэтому метод HandleAllocated() можно использовать для проверки существования дескриптора, когда не требуется его создание.
Hide	Скрывает компонент. После этого компонент может быть снова отображен.
Invalidate	Запрашивает перерисовку компонента. Компонент будет перерисован Windows при первой же возможности.
Perform	Позволяет компоненту послать сообщение самому себе напрямую, а не через систему сообщений Windows.
Refresh	Запрашивает немедленную перерисовку компонента, стирая перед этим изображение компонента.
Repaint	Запрашивает немедленную перерисовку компонента. Стирание фона компонента перед этим не производится.
SetBounds	Позволяет вам задать свойства Top, Left, Width и Height одновременно. Это удобнее, чем определять их по отдельности.
SetFocus	Устанавливает фокус ввода в компоненте и делает этот компонент активным. Применяется только к компонентам оконного типа.
Update	Вызывает немедленную принудительную перерисовку компонента. Обычно для этого следует использовать методы Refresh или Repaint.

Теперь давайте рассмотрим некоторые из тех событий, на которые компонентам приходится реагировать чаще всего.

Основные события

Как и в случае свойств и методов, среди событий имеются такие, которые инициируются наиболее часто. Компоненты представляют большое количество возможных элементов управления Windows, и каждый компонент имеет свои индивидуальные особенности. События, специфические для форм, здесь не рассматриваются, поскольку я уже говорил о них в 6-й день. Наиболее часто используемые события перечислены в таблице 8.3.

Таблица 8.3. Основные события компонентов

Событие	Описание
OnChange	Это событие возникает, когда в элементе управления происходят какие-либо изменения. Конкретная реализация зависит от компонента.
OnClick	Происходит при щелчке на компоненте любой кнопкой мыши.
OnDblClick	Происходит при двойном щелке на компоненте.
OnEnter	Происходит, когда компонент оконного типа получает фокус (активизируется).
OnExit	Происходит, когда компонент оконного типа теряет фокус в результате переключения на другой элемент управления. Однако, когда пользователь переключается между формами или приложениями, это событие не возникает.
OnKeyDown	Происходит при нажатии пользователем клавиши, если компонент находится в фокусе. Клавиши включают все алфавитно-цифровые клавиши, а также клавиши-стрелки, Home, End, Ctrl и т.д.
OnKeyPress	Происходит при нажатии одной из алфавитно-цифровых клавиш, клавиши Tab, backspace, Enter или Esc.
OnKeyUp	Происходит каждый раз, когда клавиша отпускается.
OnMouseDown	Событие возникает, когда на компоненте нажимается кнопка мыши. Параметры, передаваемые обработчику события, содержат информацию о том, какой кнопкой мыши щелкнули и какие функциональные клавиши были при этом нажаты (Alt, Shift или Ctrl), а также координаты x и y курсора мыши в момент щелчка.
OnMouseMove	Происходит при перемещении курсора мыши через элемент управления.
OnMouseUp	Событие возникает при отпускании кнопки мыши на элементе управления, если перед этим кнопка была нажата на этом же элементе.
OnPaint	Это событие возникает каждый раз, когда необходимо перерисовать компонент. Вы можете использовать это событие для выполнения специфической для данного компонента процедуры перерисовки.

Особенности событий, связанных с мышью

Обработка связанных с мышью событий имеет свои особенности. Если вам нужно обрабатывать только щелчки мышью на компоненте, то для этого вам достаточно отслеживать события OnClick. Если вас интересуют события OnMouseDown и OnMouseUp, вы должны знать, что вместе с этими событиями будет происходить и событие OnClick. Например, одиночный щелчок кнопкой мыши вызовет следующие события в указанном порядке:

```
OnMouseDown
OnClick
OnMouseUp
```

Аналогично, когда пользователь будет дважды щелкать мышью, то приложение будет получать большее количество событий, чем вы могли подумать. Когда на компоненте дважды щелкнут кнопкой, то будут происходить следующие события:

```
OnMouseDown
OnClick
OnDblClick
OnMouseUp
```

Поэтому я призываю вас к внимательности при обработке событий, связанных с одиночным и двойным щелчком на компоненте. Помните, что в случае двойного щелчка вы должны получить четыре события.

Множественные события происходят и при нажатии клавиш. Например, при нажатии клавиши в окне редактирования произойдут события OnKeyDown, OnKeyPress, OnChange и OnKeyUp.



Компакт-диск, прилагаемый к этой книге, содержит программу EventTst, которая демонстрирует множественные события при щелчках мышью и нажатии клавиш. Запустив эту программу, вы увидите, к каким событиям приводят различные действия пользователя.

Сейчас мы приступим к более подробному рассмотрению отдельных компонентов VCL. Но сначала я хочу познакомить вас с одним классом, который используется некоторыми компонентами.

Класс TStrings

Класс TStrings — это класс VCL, который используется для работы с массивами строк. Некоторые компоненты VCL используют экземпляры класса TStrings для работы со своими данными (обычно текстовыми). Например, в 7-й день вы использовали класс TStrings при создании приложения Scratch-Pad. Вы скажете, что не использовали в этой программе класс TStrings? И будете неправы — вы просто не знали об этом. Вспомните, что при сохранении и загрузке файлов мы использовали строки типа:

```
Memo->Lines->SaveToFile(SaveDialog->FileName);
```

Свойство Lines компонента TMemo является экземпляром класса TStrings. Метод SaveToFile() класса TStrings записывает строки в файл на диске. Загрузка окна списка из дискового файла или сохранение списка в файле производятся похожим образом. В случае класса TListBox, свойство, которое хранит элементы списка, называется Items. В качестве примера выполните следующее упражнение.

1. Создайте новое приложение и поместите в форму компонент ListBox. Если хотите, измените размер списка.
2. Измените значение свойства Name списка на Listbox.
3. Дважды щелкните на самой форме (не на списке). Редактор кода отобразит функцию FormCreate().
4. Дополните функцию FormCreate() следующим кодом:

```
char winDir[256], fileName[256];
GetWindowsDirectory(winDir, sizeof(winDir));
sprintf(fileName, "%s\\win.ini", winDir);
ListBox->Items->LoadFromFile(fileName);
```

5. Щелкните на кнопке Run, чтобы откомпилировать и запустить программу.

После запуска программы в окне списка будет находиться содержимое вашего файла WIN.INI. Этот метод удобно использовать для загрузки данных из любого ASCII-файла. Компонент ComboBox также имеет свойство Items, которое работает точно таким же образом.

Вы можете добавлять, удалять, вставлять и перемещать элементы списка, комбинированного списка или компонента Memo, вызывая методы Add(), Append(), Delete(), Insert() и Move() класса TStrings.



Работа метода Add() зависит от значения свойства Sorted. Если для Sorted установлено значение true, то Add() вставит строку в соответствующее место списка. Если Sorted имеет значение false, то новая строка будет добавлена в конец списка.

Очистить содержимое компонента можно методом Clear(). Получить отдельную строку списка можно с помощью свойства Strings класса TStrings и индексного оператора. Например, чтобы извлечь из списка первую строку, вы можете действовать следующим образом:

```
Edit->Text = ListBox->Items->Strings[0];
```

Каждая строка в массиве TStrings содержит саму строку и 4 дополнительных байта. Доступ к этой дополнительной памяти можно получить при помощи свойства Objects. Вы можете использовать эту память по своему усмотрению. Например, если вы создаете окно списка для вывода растровых изображений, то можете хранить строки обычным способом и дополнительно использовать массив Objects для размещения указателей на объекты TBitmap.



В ряде случаев возникает необходимость работать с массивом строк, который не связан с каким-либо компонентом. В этом случае вы можете использовать класс TStringList. Этот класс работает точно так же, как TStrings, но может использоваться независимо от компонентов.



В действительности класс TStrings является тем, что называется *абстрактным базовым классом*. Абстрактный базовый класс никогда не используется непосредственно, а служит основой для производных классов. Свойство Lines компонента Memo является экземпляром класса TMemoStrings, а не экземпляром класса TStrings, как я утверждал в этом разделе. Я ни в коей мере не хотел ввести вас в заблуждение, но решил отложить обсуждение этого вопроса, чтобы не запутать вас при знакомстве с классом TStrings.

Стандартные элементы управления Windows

Когда-то в доисторические времена жил динозавр по имени Windows 3.0. Однако именно среда Windows 3.0 дала нам однострочный и многострочный элементы редактирования, окна списков, комбинированные списки, кнопки, флажки, радиокнопки и статические элементы управления. Эти элементы оказались довольно удачно придуманы и получили широкое распространение в современных Windows-программах, даже принимая во внимание все новые элементы управления Win32.

Я не собираюсь рассматривать каждый элемент управления Windows и соответствующий ему компонент VCL. Однако вы должны знать несколько важных моментов, которые относятся к стандартным компонентам.

Элементы редактирования

В C++Builder имеется четыре стандартных компонента для управления редактированием. Компоненты Edit, Memo и MaskEdit основаны на стандартном элементе редактирования Windows. Компонент RichEdit основан на элементе управления Win32, который не относится к стандартным элементом управления Windows. Тем не менее, я рассмотрю здесь компонент RichEdit, поскольку он имеет много общего с другими элементами редактирования.

Компонент Edit инкапсулирует базовый однострочный элемент редактирования. Этот компонент не имеет свойств Align и Alignment. Свойство Alignment отсутствует, так как текст в соответствующем элементе редактирования может быть выровнен только по левому краю. Свойства Align нет потому, что этот элемент не может (точнее говоря, не должен) увеличивать свой размер до размеров рабочей области окна.



Если вам все-таки необходимо выравнивать вправо или центрировать текст в элементе редактирования, используйте компонент Memo, установив его высоту равной высоте стандартного компонента Edit. Затем установите для свойства Alignment необходимое значение.



Старайтесь делать формы по возможности стандартными. Если вы сделаете компонент Edit слишком высоким, это может ввести пользователя в заблуждение (он может подумать, что имеет дело с многострочным элементом редактирования).

Компонент MaskEdit представляет собой компонент Edit, к которому добавлен фильтр, или маска вводимых данных. MaskEdit, по существу, не представляет какой-либо элемент управления Windows, а является всего лишь VCL-расширением стандартного элемента редактирования. Маска используется для того, чтобы ограничить вводимые данные определенным диапазоном чисел или символов. Кроме того, маска может содержать специальные символы, которые по умолчанию помещаются в поле редактирования. Например, дата часто записывается в следующем формате:

```
10/25/97
```

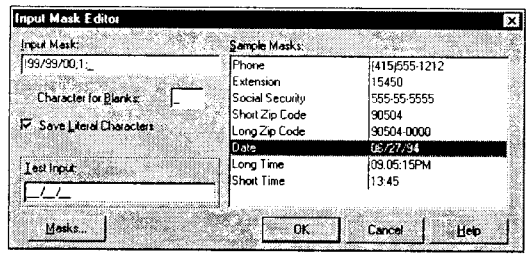
Маска редактирования, предназначенная для ввода даты, может уже содержать наклонные черты в нужных местах, и пользователю останется только ввести числа. Маска редактирования может ограничивать вводимые символы только цифрами, чтобы исключить возможность ввода пользователем нечислового символа.

Свойство EditMask задает используемую маску. Когда вы нажимаете кнопку с многоточием (...) в столбце Value свойства EditMask, появляется окно редактора маски ввода (Input Mask Editor). Это диалоговое окно позволяет вам выбрать одну из предопределенных масок или создать свою собственную. Вы можете выбрать набор масок нужной вам страны. На рис. 8.3 в окне редактора масок ввода показан набор предопределенных масок для Великобритании.

За подробной информацией относительно создания ваших собственных масок обращайтесь к справочной системе C++Builder.

Компонент Memo инкапсулирует многострочный элемент редактирования. Свойство Lines — самое важное из свойств этого компонента. Как упоминалось ранее при обсуждения класса TStrings, свойство Lines позволяет вам сохранять содержимое компонента Memo на диске или загружать в этот компонент текст из файла, а также делать много других вещей. Свойство ScrollBars уникально для компонента Memo. Оно позволяет вам установить только горизонтальную линейку прокрутки, только вертикальную или обе вместе. Вы использовали свойство ScrollBars в 7-й день, когда писали программу ScratchPad. Компонент Memo достаточно универсален, и вы, вероятно, будете часто его использовать.

Рис. 8.3
Редактор масок ввода



Компонент RichEdit — самый большой и самый лучший из всех компонентов редактирования. Он основан на элементе управления Win32 rich edit control. В компоненте RichEdit вы можете изменять шрифты, использовать отступы, задавать полужирный, курсивный или подчеркнутый текст и делать многое другое. Можно сказать, что компонент RichEdit представляет собой маленький текстовый процессор. Компонент RichEdit имеет на удивление мало свойств, используемых на этапе разработки программ, по сравнению с компонентом Memo. К ключевым свойствам времени выполнения относятся SetAttributes и Paragraph. Компонент RichEdit довольно сложен, но использовать его относительно легко. Полное описание компонента RichEdit содержится во встроенной справочной системе C++Builder.

В таблице 8.4 приведен список специфических свойств компонентов, основанных на элементах редактирования.

Таблица 8.4. Свойства элементов редактирования

Свойство	Где применяется	Описание
AutoSelect	Edit, MaskEdit	Когда значение свойства равно true, текст в поле редактирования будет автоматически выделяться при переходе пользователя к этому элементу управления клавишей tab. Значение по умолчанию: true
AutoSize	Edit, MaskEdit	Когда значение свойства равно true, размер поля ввода будет автоматически изменяться при изменении шрифта этого элемента управления. В противном случае размер элемента редактирования не будет изменяться при изменении размера шрифта. Значение по умолчанию: true
CharCase	Edit, MaskEdit	Определяет регистр отображаемого текста: верхний (ecUpperCase), нижний (ecLowerCase) или смешанный (ecNormal). Значение по умолчанию: ecNormal

Свойство	Где применяется	Описание
HideScrollBars	RichEdit	Если установлено значение true, то линейки прокрутки будут отображаться только при необходимости. Когда свойство имеет значение false, отображение линеек прокрутки определяется свойством ScrollBars.
HideSelection	Edit, Memo, RichEdit	Если установлено значение true, выделение выбранного текста не будет показываться, когда пользователь переключается на другой элемент управления. Значение по умолчанию: false
Lines	Memo, RichEdit	Содержит текст компонента. Свойство Lines является экземпляром класса TStrings.
MaxLength	Все	Определяет максимальное число символов, которые может содержать компонент. Если задано значение 0, то количество вводимых символов не ограничивается (ограничение накладывается только системными установками). Когда определено любое отличное от нуля значение, число символов ограничивается этим значением. Значение по умолчанию: 0
OEMConvert	Edit, Memo	Устанавливается в true, когда вводимый текст состоит из имен файлов. Значение по умолчанию: false
PasswordChar	Edit, MaskEdit	Когда это свойство имеет любое значение, отличное от ASCII #0, все вводимые символы отображаются на экране символом с указанным кодом. Введенный фактически текст не изменяется. В большинстве полей, используемых для ввода пароля, применяют символ звездочки (*). Значение по умолчанию: #0
PlainText	RichEdit	Если установлено значение true, RTF-файлы (rich text format) отображаются как простой текст без форматирования символов и абзацев. Если свойство имеет значение false, RTF-файлы будут полностью форматироваться. Значение по умолчанию: false
ReadOnly	Все	Если установлено значение true, компонент будет отображать текст, но его редактирование будет запрещено. Пользователь может, однако, выделить текст и скопировать в буфер обмена. Значение по умолчанию: false
ScrollBars	Memo, RichEdit	Определяет, какие линейки прокрутки будут отображаться. Возможные значения – ssNone, ssBoth, ssHorizontal и ssVertical. Значение по умолчанию: ssNone
Text	Edit, MaskEdit	Содержит текст компонента.
WantReturns	Memo, RichEdit	Если установлено значение true, компонент помещает символы возврата каретки и новой строки в элемент редактирования, когда пользователь нажимает клавишу Enter. В случае значения false, символы возврата каретки передаются форме и не попадают в поле редактирования. Если в вашей форме есть заданная по умолчанию кнопка и свойство WantReturns имеет значение false, то нажатие клавиши Enter приведет к закрытию формы. Значение по умолчанию: true

см. продолжение

Таблица 8.4. Свойства элементов редактирования (продолжение)

Свойство	Где применяется	Описание
WantTabs	Memo, RichEdit	Если установлено значение true, символ табуляции помещается в элемент редактирования при нажатии клавиши Tab. При установке значения false символы табуляции будут передаваться форме; это можно использовать для перехода к следующему элементу управления. Значение по умолчанию: false
WordWrap	Memo, RichEdit	Если установлено значение true, текст будет переноситься на новую строку по достижении правого края поля ввода. При установке значения false вводимый в поле текст будет автоматически прокручиваться. Значение по умолчанию: true
	Свойства времени выполнения	
Modified	Все	Показывает, изменилось ли содержимое элемента редактирования после того, как свойство Modified было установлено в последний раз. После сохранения содержимого компонентов Memo или RichEdit в файле, вы должны установить для свойства Modified значение false.
SelLength	Все	Содержит длину текста, выделенного в настоящий момент в элементе редактирования.
SelStart	Все	Содержит позицию первого символа выделенного текста. Первому символу в элементе редактирования соответствует 0.
SelText	Все	Содержит текст, выделенный в настоящий момент в элементе редактирования.

Элементы редактирования имеют большое количество общих методов, но их слишком много, чтобы привести здесь полный список. Методы CutToClipboard(), CopyToClipboard(), PasteFromClipboard() и Clear() применяются при работе с буфером обмена. Методы GetSelTextBuff() и GetTextBuff() извлекают выделенный текст и весь текст компонента, соответственно. Полный список методов для каждого из компонентов редактирования вы найдете во встроенной справочной системе C++Builder, в разделах, посвященных TEdit, TMaskEdit, TMemo и TRichEdit.

События компонента редактирования, которые могут вас заинтересовать, зависят от типа используемого компонента. Но можно заранее сказать, что наиболее часто вы будете использовать события OnEnter, OnExit, OnChange, OnKeyDown (или OnKeyPress) и OnKeyUp.

Компоненты ListBox и ComboBox

Компоненты ListBox и ComboBox также используются очень широко. Компонент ListBox представляет собой стандартный элемент управления Windows — окно списка, содержащее набор элементов, из которых пользователь может сделать выбор. Если часть элементов списка не помещается в окне, то выводится линейка прокрутки, которая делает доступной остальную часть списка.



Окна списков могут быть *пользовательскими (owner-drawn)*. Способ отображения элементов такого списка определяется самим программистом.

При необходимости вы можете использовать пользовательские списки. Такие списки довольно широко распространены, хотя вы можете просто этого не замечать. В 6-й день я говорил о настройке панели инструментов среды C++Builder. В частности, мы рассмотрели диалоговое окно редактора панели инструментов, показанное на рис.6.4. Давайте вернемся к этому рисунку и посмотрим на него еще раз. Диалоговое окно Speedbar Editor содержит два списка. Список слева — это обычный список. В нем перечислены кнопки, которые вы можете выбирать. А список справа является пользовательским. Он содержит изображения кнопок вместе с описанием их функций.

Комбинированный список — это специализированное окно списка. Фактически, этот компонент является комбинацией окна списка и однострочного элемента редактирования. Пользователь может выбрать значение из списка или ввести нужное значение в поле редактирования. Если пользователь выбирает элемент из списка, то выбранный элемент подставляется в поле редактирования. Существуют три различных типа комбинированных списков. Их краткие описания приведены в таблице 8.5.

Таблица 8.5. Типы комбинированных списков

Тип	Описание
Простой	Простой комбинированный список представляет собой элемент редактирования, помещенный над списком. Пользователь может выбрать значение из списка или ввести текст в поле редактирования.
Выпадающий	Похож на простой тип, за исключением того, что список первоначально не отображается. Для раскрытия списка и выбора элементов необходимо нажать кнопку. Пользователь может также ввести нужные данные в поле редактирования.
Выпадающий список	Этот самый ограниченный вариант комбинированного списка. Как и в случае выпадающего типа, список первоначально не отображается. Пользователь должен нажать на кнопку раскрытия списка, чтобы вывести список и выбрать элемент, но он не может ввести текст в поле редактирования. Используйте этот тип, когда вы хотите ограничить выбор пользователя заранее определенным набором значений.



Компакт-диск, прилагаемый к книге, содержит программу ComboBoxTest, которая демонстрирует различные типы комбинированных списков. На рис. 8.4 показано окно этой программы. Запустите программу и поиграйте с каждым комбинированным списком, чтобы почувствовать, как он работает.

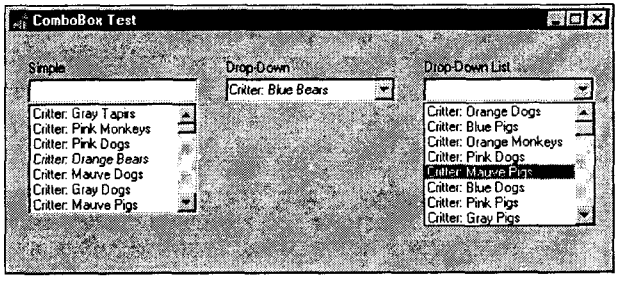


Рис. 8.4
Окно программы ComboBoxText

В таблице 8.6 приведены свойства, общие для окон списков и комбинированных списков.

Таблица 8.6. Свойства окон списков и комбинированных списков

Свойство	Где применяется	Описание
	Свойства	
Columns	ListBox	Содержит число столбцов в списке. Если вы установите для этого свойства значение, большее 1, то получите список из нескольких столбцов. Значение по умолчанию: 0
ExtendedSelection	ListBox	Определяет, разрешено ли расширенное выделение. Расширенное выделение (extended selection) позволяет пользователю выбирать элементы, используя комбинации Shift+щелчок Ctrl+щелчок. Это свойство не играет роли, если MultiSelect имеет значение false. Значение по умолчанию: true
IntegralHeight	ListBox	Если установлено значение true, то высота списка будет меняться, чтобы избежать частичного вывода строки. При значении false строки могут отображаться частично. Значение по умолчанию: false
ItemHeight	Оба компонента	Используется в пользовательских и комбинированных списках. Устанавливает высоту элемента списка. Значение по умолчанию: 13
Items	Оба компонента	Экземпляр класса TStrings, который содержит элементы списка. (Свойства и методы TStrings описаны выше в этой главе.)
MaxLength	ComboBox	Максимальное число символов, которое пользователь может ввести в поле редактирования компонента ComboBox. Аналогично свойству MaxLength элемента редактирования. Значение по умолчанию: 0 (нет ограничений)
MultiSelect	ListBox	Если установлено значение true, то для списка разрешено групповое выделение элементов. Значение по умолчанию: false
Sorted	Оба компонента	Если установлено значение true, то элементы списка сортируются в порядке возрастания. При значении false элементы не сортируются. Значение по умолчанию: false
Style	ComboBox	Определяет тип комбинированного списка. Возможные значения: csSimple, csDropDown, csDropDownList, lbOwnerDrawFixed и csOwnerDrawVariable. (См. описание трех основных типов в таблице 8.5.) Значение по умолчанию: csDropDown
	ListBox	Для окна списка возможны следующие значения: lbStandard, lbOwnerDrawFixed и csOwnerDrawVariable. Значение по умолчанию: lbStandard
TabWidth	ListBox	В списках могут использоваться символы табуляции. Это свойство устанавливает ширину табуляции в пикселах. Значение по умолчанию: 0
Text	ComboBox	Содержит текст из поля ввода комбинированного списка.

Свойство	Где применяется	Описание
Свойства времени выполнения		
ItemIndex	ListBox	Содержит индекс выбранного элемента списка. Индекс первого элемента в списке равен 0. Возвращает -1, если элемент не выбран. При записи, элемент с указанным индексом становится выбранным.
SelCount	ListBox	Содержит количество выбранных элементов в списке с групповым выделением.
Selected	ListBox	Возвращает true, если указанный элемент выбран, и false в противном случае.
SelLength	ComboBox	Содержит длину текста, выделенного в поле ввода кобинированного списка.
SelStart	ComboBox	Содержит позицию первого символа выделенного текста в поле ввода. Первый символ в элементе редактирования имеет номер 0.
SelText	ComboBox	Содержит текст, выделенный в элементе редактирования.
TopIndex	ListBox	Возвращает элемент, находящийся наверху списка. Может использоваться для перемещения определенного элемента списка наверх.

Как и для компонентов редактирования, рассмотренных ранее, я укажу лишь несколько методов для компонентов ListBox и ComboBox. Метод Clear() удаляет все данные из компонента. Метод ItemAtPos() возвращает элемент списка по заданным координатам (x, y). Метод SelectAll() выделяет текст в поле ввода компонента ComboBox.

Самым используемым событием для комбинированных списков и окон списка является OnClick, с помощью которого можно определить момент выбора элемента в списке.



Щелчок в поле ввода компонента ComboBox или на кнопке вызова списка не вызывает события OnClick. Это событие происходит только при щелчке в поле списка компонента ComboBox.

Для обнаружения изменений в поле ввода комбинированного списка, так же, как и в элементах редактирования, может использоваться событие OnChange. Событие OnDropDown служит для обнаружения нажатия кнопки вызова списка в компоненте ComboBox. События OnMeasureItem и OnDrawItem применяются в пользовательских окнах списков и комбинированных списках.

Кнопки

Библиотека VCL содержит несколько типов кнопок, которые вы можете использовать в своих приложениях. И хотя не все из них основаны на стандартном элементе управления Windows — кнопке, я буду рассматривать здесь все типы кнопок. Прежде, чем мы обратимся к конкретным типам компонентов кнопок, давайте рассмотрим некоторые основные моменты.



При установке свойства Caption кнопки используйте амперсанд (&), как и для свойства Caption элементов меню. Следующий за амперсандом символ будет использован для быстрого доступа к этой кнопке и обозначен подчеркиванием.

Основы работы с кнопками

Компоненты кнопок имеют только три свойства, заслуживающих внимания.

Свойство ModalResult

Свойство ModalResult используется для закрытия форм, отображаемых методом ShowModal(). По умолчанию ModalResult имеет значение mrNone (которое определено как константа со значением 0). Используйте это значение для обычных кнопок на форме, которые не должны ее закрывать. Если вы используете для ModalResult любое другое отличное от нуля значение, то нажатие такой кнопки будет приводить к закрытию формы и возврату значения ModalResult. Например, если вы помещаете кнопку в форму и устанавливаете для свойства ModalResult значение mrOk, то при нажатии этой кнопки форма будет закрыта, а ShowModal() возвратит mrOk (т.е. 1). Вы можете использовать это примерно следующим образом:

```
int result = MyForm->ShowModal();
if (result == mrOK) DoSomething();
if (result == mrCancel) return;
```

В таблице 8.7 перечислены константы, определенные в VCL для свойства ModalResult.

Таблица 8.7 Константы VCL для ModalResult

Имя константы	Значение
mrNone	0
mrOk	1
mrCancel	2
mrAbort	3
mrRetry	4
mrIgnore	5
mrYes	6
mrNo	7
mrAll	8



Вы не обязаны использовать одну из предопределенных констант ModalResult в своих кнопках. Используйте любое другое значение, которое вам нравится. Предположим, что у вас есть диалоговое окно, которое может быть закрыто различными кнопками. Вы можете установить для свойства ModalResult каждой кнопки различные значения (скажем, 100, 150 и 200), и тем самым будете знать, какая кнопка закрыла диалоговое окно. Допустимо использование любого отличного от нуля целого числа, вплоть до максимально возможного для типа int.



На прилагаемом к книге компакт-диске находится программа ButtnTst, демонстрирующая использование свойства ModalResult. Программа отображает форму с несколькими кнопками. Когда вы нажимаете одну из кнопок, значение ModalResult возвращается главной форме.

Свойство Default

Свойство Default — другое ключевое свойство кнопок. Windows имеет стандартный механизм работы с диалоговыми окнами. Одно из правил работы этого механизма звучит следующим образом: если фокус ввода с клавиатуры находится в элементе управления, который не является кнопкой, и пользователь нажал на клавиатуре Enter, диалоговое окно будет вести себя так, как если бы пользователь нажал *кнопку по умолчанию (default button)*. Заданная по умолчанию кнопка — это кнопка, для которой установлен стиль BS_DEFPUSHBUTTON (обычно это кнопка OK). Эта особенность Windows в течение многих лет была проклятием программистов и пользователей, занимающихся вводом данных. Свойство Default используется для определения заданной по умолчанию кнопки формы. По умолчанию оно имеет значение false. Чтобы сделать кнопку заданной по умолчанию, установите для свойства Default значение true. Если вы специально не устанавливали свойство Default ни для одной кнопки, форма не будет закрываться при нажатии клавиши Enter.



Если пользователь закрывает форму, нажимая клавишу Enter, то перед закрытием формы будет вызван обработчик события OnClick для кнопки по умолчанию (если такая существует).

Свойство Cancel

Свойство Cancel работает с клавишей Esc аналогично тому, как свойству Default работает с клавишей Enter. Когда пользователь нажимает Esc, чтобы закрыть форму, возвращаемое ShowModal() значение будет значением свойства ModalResult той кнопки, для которой свойство Cancel установлено в true. Если ни у одной кнопки формы свойство Cancel не имеет значения true, то при закрытии формы по клавише Esc будет возвращено значение mrCancel (mrCancel равно 2; см. таблицу 8.7).



Если форма закрывается щелчком на системной кнопке закрытия окна или нажатием Alt+F4, то ShowModal() возвращает значение mrCancel, как и следовало ожидать. Однако нажатие на клавишу Esc закончится возвратом значения ModalResult той кнопки, для которой свойство Cancel равно true. При этом перед закрытием формы будет вызван обработчик события OnClick данной кнопки. При закрытии окна с помощью системной кнопки или сочетания Alt+F4, обработчик события OnClick не вызывается. Постарайтесь предусмотреть в ваших программах все возможные способы работы с формами.



Вы можете иметь несколько кнопок со свойством Default, равным true. То же самое верно и для свойства Cancel. В этом случае при нажатии клавиши Enter будет вызываться кнопка, первая в последовательности переключения по клавише табуляции, свойство Default которой равно true. Аналогично этому, когда пользователь закрывает форму клавишей Esc, ShowModal() будет возвращать значение свойства ModalResult первой кнопки в последовательности переключения по клавише табуляции, имеющей свойство Cancel со значением true.

Свойство Enabled

Ранее я говорил о свойстве Enabled при общем рассмотрении компонентов. У кнопок это свойство часто используется для их включения или выключения, в зависимости от текущего состояния программы или отдельной формы. Когда доступ к кнопке запрещен (свойство Enabled имеет значение false), ее текст становится серым и кнопка не функционирует. В случае кнопок со значками (BitBtn и SpeedButton), эти значки также автоматически станут серыми.

Компоненты-кнопки имеют только один метод, представляющий интерес: метод Click(), который имитирует щелчок мыши. Когда вы вызываете метод Click() кнопки, выполняется обработчик события OnClick, как если бы вы щелкнули на этой кнопке. Что касается событий, то обычно используется только событие OnClick.

Теперь давайте рассмотрим различные компоненты кнопок, которые содержит C++Builder.

Компонент Button

Стандартный компонент Button можно сравнить с актером Денни де Вито — он не слишком красив, но очень трудолюбив. Фактически, мне нечего больше добавить относительно компонента Button. Свойство Height имеет значение по умолчанию в 25 пикселов, а свойство Width — 75. Обычно после размещения кнопки на форме вам придется только обработать событие OnClick.

Компонент BitBtn

BitBtn — хороший пример того, как можно расширить компонент для увеличения его функциональных возможностей. В данном случае к стандартной кнопке добавлена возможность вывода на ее поверхность растрового изображения.

Компонент BitBtn имеет несколько дополнительных свойств по сравнению с компонентом Button. Все вместе эти свойства управляют изображением на кнопке, а также взаимным расположением значка и текста кнопки. Мы рассмотрим эти свойства в следующих подразделах.

Свойство Glyph

Свойство Glyph представляет растровое изображение, помещаемое на кнопку. Значением свойства Glyph является значок (глиф).

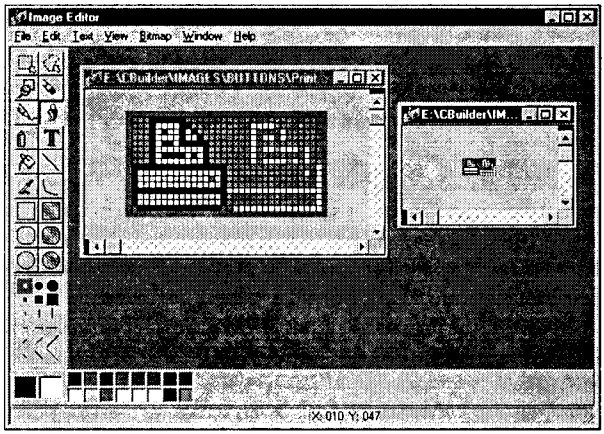


Глиф (glyph) — это изображение, которое обычно имеет формат растрового файла Windows (BMP).

Сам глиф состоит из одного или более растровых изображений, которые соответствуют четырем возможным состояниям кнопки: отжата, нажата, недоступна или зафиксирована в нажатом положении. Если вы станете создавать собственные кнопки, то, вероятно, начнете с создания глифа для одного состояния, который будет изменяться компонентом BitBtn, чтобы представить другие три возможных состояния. Изображение будет сдвигаться вниз и влево при нажатии кнопки, и затеняться серым ее при блокировке. Изображение для «залипшей» кнопки будет тем же, что и для отжатой, только внешний вид самой кнопки изменится, чтобы создать впечатление нажатия.

Если вы определяете более одного глифа, то все они должны иметь одинаковые высоту и ширину и следовать друг за другом, образуя полосу растровых изображений. Изображения, входящие в поставку C++Builder, содержат по два глифа. На рис. 8.5 показан значок для кнопки печати, содержащийся в C++Builder (PRINT.BMP), в натуральную величину и с увеличением, чтобы показать детали. Обратите внимание, что оба глифа имеют одинаковую ширину.

Рис. 8.5
Растровое изображение PRINT.BMP





Цвет пиксела в нижнем левом углу изображения определяет цвет, который будет использоваться в качестве прозрачного. Все пикселы растра, имеющие этот цвет, будут прозрачными при отображении глифа на поверхности кнопки. Вы должны помнить об этом при создании собственных значков. Если вы не используете прозрачность, то цвет пиксела из нижнего левого угла не должен повторяться где-либо еще на изображении.

Чтобы определить глиф для BitBtn, дважды щелкните в столбце Value инспектора объектов напротив свойства Glyph. На экране появится редактор изображений, и вы сможете выбрать значок, который будет использоваться для глифа.



Стандартные глифы кнопок, поставляемые с C++Builder, имеют размер 15x15 пикселов. Этот размер хорошо подходит к высоте стандартной кнопки в 25 пикселов. Ваши глифы, конечно, могут иметь любой другой размер, который вы сочтете нужным, только учтите, что компонент BitBtn не будет предпринимать никаких усилий по подгонке размера кнопки к размеру изображения. Если вы используете большой глиф, то должны сами позаботиться об увеличении размера кнопки.

Свойство Kind

Kind (Вид) — очень удобное свойство компонента BitBtn, которое позволяет вам выбрать один из нескольких предопределенных видов кнопок. По умолчанию свойство Kind имеет значение bkCustom, т.е. вы сами будете определять глиф и другие свойства кнопки. Если вы выберете любой другой из предопределенных типов, то это вызовет следующие пять действий:

- Свойство Glyph автоматически получит значение, соответствующее выбранному типу кнопки.
- Свойства Cancel или Default изменятся согласно типу выбранной кнопки.
- Свойство Caption также изменится в соответствии с типом выбранной кнопки.
- Свойство ModalResult получит значение, соответствующее типу выбранной кнопки.
- Изображение кнопки на форме будет перерисовано, чтобы отразить все эти изменения.



Например, если вы установите для Kind значение bkOK, то получите кнопку типа OK. В качестве глифа будет использовано изображение зеленой галочки, свойство Cancel получит значение false, свойство Default будет иметь значение true, значение свойства ModalResult станет равным mrOk, а свойство Caption получит значение OK. Результат этих изменений вы сразу увидите в форме. Вы всегда можете модифицировать любое из свойств, измененных в результате установки свойства Kind, но обычно это не требуется. На рис. 8.6 показана форма BitBtn Test, которая отображается программой тестирования кнопок, содержащейся на компакт-диске. В этой форме представлены все предопределенные типы кнопок и одна заказная кнопка.

Свойство Layout

Свойство Layout (Размещение) определяет взаимное расположение глифа и текста. Значение по умолчанию — blGlyphLeft. Вы можете также расположить глиф справа от текста, над текстом или под текстом.

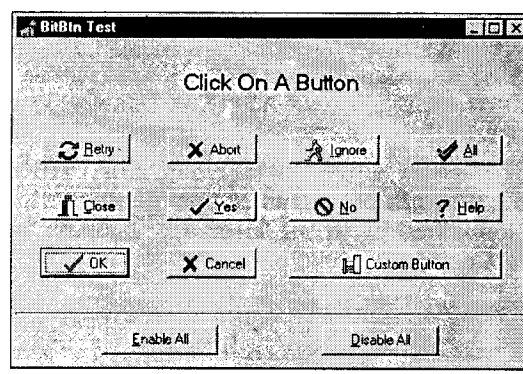


Рис. 8.6
Предопределенные типы BitBtn

Свойство Margin

Свойство Margin (Поле) определяет расстояние между глифом и краем кнопки (какой край кнопки имеется в виду, зависит от значения свойства Layout). По умолчанию установлено значение -1, означающее центровку глифа и текста относительно кнопки. Вы можете ввести любое положительное значение, которое будет задавать абсолютное значение отступа (в пикселах).

Свойство NumGlyphs

Свойство NumGlyphs определяет количество глифов, содержащихся в вашей растровой полосе для данной кнопки. Вы можете задавать от одного

до четырех глифов, как было сказано выше. Глифы должны составлять полосу растровых изображений в следующем порядке: кнопка отжата, недоступна, нажата, зафиксирована в нажатом положении.

Свойство Spacing

Свойство Spacing определяет расстояние в пикселах между глифом и текстом кнопки. Значение по умолчанию — 4 пиксела.

Компонент SpeedButton

Компонент SpeedButton был разработан для использования в оперативных панелях (панелях инструментов). Он отличается от компонентов Button и BitBtn тем, что не является компонентом оконного типа. Это означает, что кнопка быстрого доступа не может получить фокус ввода, и на нее нельзя переключиться по клавише Tab. С другой стороны, компонент SpeedButton имеет несколько общих с компонентом BitBtn черт. Свойство Glyph компонента SpeedButton аналогично одноименному свойству компонента BitBtn, поэтому я не буду обсуждать его снова. Существуют два основных отличия, которыми мы и займемся.

По умолчанию, кнопки быстрого доступа имеют форму квадрата размером 25x25 пикселов. Ваши кнопки могут иметь любой размер и содержать текст, хотя обычно такие кнопки текста не содержат. У кнопок быстрого доступа есть некоторые специфические свойства, о которых вы должны знать. Я расскажу о них в следующих подразделах.

Свойство GroupIndex

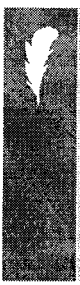
Кнопки быстрого доступа могут быть сгруппированы и вести себя подобно радиокнопкам (радиокнопки будут рассмотрены позже в этой главе, в разделе «Радиокнопки и флажки»). При нажатии одной из кнопок в группе, она фиксируется в нажатом положении, а кнопка, которая была нажата ранее, «выскакивает». Чтобы сгруппировать кнопки быстрого доступа, достаточно присвоить одно и то же значение свойству GroupIndex всех кнопок группы. (Значение по умолчанию 0 указывает на то, что кнопка не входит ни в одну группу.) В качестве примера давайте выполним следующее упражнение.

1. Создайте пустую форму и разместите на ней пять кнопок быстрого доступа. (Я не буду использовать глифы в этом простом упражнении, но при желании вы можете это сделать.)
2. Выделите все кнопки и измените значение свойства GroupIndex на 1. Свойство GroupIndex всех кнопок получит значение 1.
3. Не обязательно: измените значение свойства Down одной из кнопок на true.
4. Нажмите кнопку Run, чтобы откомпилировать и запустить вашу программу.

Во время выполнения программы щелкните на нескольких кнопках. Вы обнаружите, что только одна из кнопок может находиться в нажатом положении. Как видите, при установке отличного от нуля значения для свойства GroupIndex, кнопки быстрого доступа меняют свое поведение. Кнопка с нулевым значением GroupIndex возвращается в исходное состояние, когда вы щелкаете на ней, в то время как кнопка, являющаяся частью группы, остается нажатой.

Свойство AllowAllUp

По умолчанию, одна из кнопок группы должна быть всегда нажата. Вы можете изменить это правило, если установите для свойства AllowAllUp значение true. Установка этого значения у одной из кнопок автоматически изменяет значение свойства AllowAllUp для остальных кнопок группы на true. Теперь вы можете выбрать любую кнопку в группе или не выбирать никакую кнопку вообще.



Иногда удобно иметь кнопку быстрого доступа, которая действует как переключатель. Кнопка-переключатель используется для включения и выключения некоторой опции, при этом она не входит в группу. Чтобы сделать из отдельной кнопки быстрого доступа кнопку-переключатель, установит для ее свойства GroupIndex отличное от нуля значение и одновременно установите значение true для свойства AllowAllUp. Убедитесь в том, что установленное значение свойства GroupIndex не используется никакими другими компонентами формы. Когда пользователь нажимает такую кнопку, она «залипает». Когда кнопку нажимают еще раз, она возвращается в исходное состояние.

Свойство Down

Свойство Down при чтении возвращает true, если кнопка в настоящее время нажата, и false в противном случае. Запись значения в свойство Down может использоваться для перевода кнопки в нажатое или отжатое положение. Присваивание значения свойству Down не даст никакого эффекта, если кнопка быстрого доступа не входит в группу.

Радиокнопки и флажки

Радиокнопка (radio button) и флажок (check box) — это специализированные кнопки, однако они, тем не менее, остаются кнопками. Я не буду тратить много времени на обсуждение этих двух типов кнопок, поскольку их реализация довольно проста. Компоненты RadioButton и CheckBox имеют свойство, называемое Checked, которое используется для задания состояния кнопки и может быть считано для определения текущего состояния.

Радиокнопки обычно используется в составе группы кнопок и, как правило, представляют набор опций, из которых только одна может быть выбрана (подобно группе кнопок быстрого доступа, о которых мы только что говорили). Хотя вы можете использовать радиокнопку в единственном числе, это не рекомендуется, чтобы не ввести в заблуждение вашего пользователя. Если вам действительно нужна одна радиокнопка, используйте вместо нее флажок, который для этого и предназначен.

Любые радиокнопки, помещенные в форму, будут автоматически рассматриваться как часть одной и той же группы. Если вам нужно иметь более одной группы радиокнопок, функционирующих независимо друг от друга, то воспользуйтесь компонентом RadioGroup. Он позволит вам быстро задать группу радиокнопок с заголовком и трехмерной рамкой вокруг кнопок. В качестве иллюстрации давайте сделаем следующее упражнение.

1. Создайте пустую форму или используйте форму из предыдущего упражнения. Поместите в нее компонент RadioGroup.
2. Найдите свойство Items и дважды щелкните в столбце Value.

3. Появится окно редактора списка строк. Введите следующие строки в редакторе:

```
Краснохвостый ястреб
Сокол сапсан
Кречет
Северный ястреб-тетеревятник
```

4. Нажмите OK, чтобы закрыть редактор списка строк. Групповое окно заполнится радиокнопками, содержащими введенный текст.
5. Измените свойство Caption группы радиокнопок на Начинающие сокольничие по закону могут владеть:.
6. Щелкните на кнопке Run, чтобы откомпилировать и запустить программу.

Когда вы нажимаете одну из радиокнопок, то прежде отмеченная кнопка возвращается в исходное состояние. При помощи компонента RadioGroup вы можете поместить в форму более одной группы радиокнопок. Подобно компонентам ListBox и ComboBox, рассмотренным ранее, компонент RadioGroup имеет свойство ItemIndex, прочитав которое во время выполнения программы, вы можете определить, какая из кнопок в группе выбрана. Между прочим, правильный ответ на наше задание — краснохвостый ястреб (ответ «американская пустельга» тоже был бы правильным, но его нет в списке).



Для определения группы радиокнопок вы можете также использовать компонент GroupBox. Компонент GroupBox не так удобен в использовании, как RadioGroup, но зато он более универсален. Вы можете поместить в группу элемент управления любого типа. После того, как элементы управления размещены в групповом окне, они могут перемещаться вместе с этим окном как единое целое во время разработки программы.

Компонент CheckBox используется для предоставления пользователю возможности включить или выключить некоторый режим, или индикации включенного режима. Флажок может иметь до трех состояний, в зависимости от стиля: установлен, сброшен или заблокирован. Если свойство AllowGrayed имеет значение false, то такой флажок может быть только установлен или сброшен. Когда свойство AllowGrayed установлено в true, флажок может быть в любом из трех состояний. Заблокированное (или неопределенное) состояние обрабатывается программно. Иными словами, вы должны решать, что означает это состояние для вашего приложения. Если свойство AllowGrayed установлено в false (значение по умолчанию), вы можете использовать свойство Checked, чтобы определить, установлен флажок или нет. Если свойство AllowGrayed равно true, вы должны использовать свойство State для определения или задания состояния флажка. State может возвращать cbChecked, cbUnchecked или cbGrayed.



Чтобы сделать флажок доступным только для чтения, но не блокировать, поместите его на панель и измените свойство Enabled панели на false.

Компонент Label

Компонент Label (Метка) используется для отображения текста на форме. Иногда текст метки определяется только во время разработки и никогда не

изменяется. В других случаях метка ведет себя динамично и изменяется во время выполнения программы. Для задания текста метки во время выполнения используется свойство Caption. Компонент Label не имеет никаких специализированных методов или событий кроме тех, что имеются у других компонентов. В таблице 8.8 перечислены специфические свойства компонента Label.

Таблица 8.8. Свойства компонента Label

Свойство	Описание
AutoSize	Если установлено значение true, метка будет изменять свой размер в соответствии с текстом, содержащимся в свойстве Caption. При значении false текст будет отсекаться по правому краю метки.
FocusControl	Метка не является компонентом оконного типа, поэтому она не может получить фокус ввода и на нее нельзя переключиться клавишей табуляции. Но иногда метка служит текстом для элемента редактирования. В таких случаях вы можете назначить метке клавишу быстрого доступа (используя символ &) и установить в свойстве FocusControl имя элемента управления, который получит фокус при нажатии этой клавиши.
ShowAccelChar	Установите для этого свойства значение true, если вы хотите, чтобы символ & отображался на метке, а не служил обозначением клавиши быстрого доступа.
Transparent	Когда это свойство имеет значение true, свойство Color игнорируется, и через метку видно то, что находится под ней. Это свойство используется, например, для размещения меток на фоне растровых изображений.
WordWrap	Если установлено значение true, то текст метки переносится на новую строку по достижении правого края метки.

Компонент ScrollBar

Компонент ScrollBar — это автономная линейка прокрутки. Она автономна в том смысле, что не связана с элементом редактирования, списком, формой или чем-нибудь еще. Лично мне не приходится часто использовать ScrollBar. В некоторых видах прикладных программ этот тип компонента используется довольно интенсивно, но применение такого компонента в программах общего использования — дело довольно необычное. Параметры линейки прокрутки определяются свойствами Min, Max, LargeChange и SmallChange. Положение движка может быть задано или получено при помощи свойства Position. Свойство Kind определяет, какую линейку прокрутки вы используете: горизонтальную или вертикальную.



На компакт-диске, прилагаемом к этой книге, находится программа с названием Scroller, которая иллюстрирует возможные варианты использования линеек прокрутки. Запустите эту программу и изучите с ее помощью возможности компонента ScrollBar.

Компонент Panel

Компонент Panel (Панель) — это рабочая лошадка C++Builder. Трудно представить ситуацию, когда бы вы не могли применить этот компонент. Панели могут использоваться для размещения кнопок быстрого доступа и

обычных кнопок, отображения текстовых меток (например, заголовка формы) и графических изображений. Одно из преимуществ панели состоит в том, что компоненты, помещенные на панель, становятся ее потомками. В результате они всегда следуют за панелью, куда бы она не перемещалась. Это может быть очень полезно как во время выполнения программы, так и при ее разработке.

Свойство Align — одно из определяющих свойств компонента Panel. Допустим, что вам нужно отображать заголовок в верхней части формы. Усложним задачу, и потребуем, чтобы заголовок всегда находился по центру, независимо от того, как пользователь изменяет размеры окна. Если вы установите для свойства Align значение alTop, а для свойства Alignment — значение taCenter, ваш заголовок всегда будет отцентрирован. И больше ничего делать не нужно.



Внешний вид панели можно варьировать в широких пределах. Он определяется свойствами BevelInner, BevelOuter, BorderStyle и BorderWidth. Программа PanelEx, которую вы найдете на прилагаемом к этой книге компакт-диске, продемонстрирует вам те визуальные эффекты, которых можно добиться при использовании различных комбинаций стилей. На рис. 8.7 показано окно программы PanelEx.

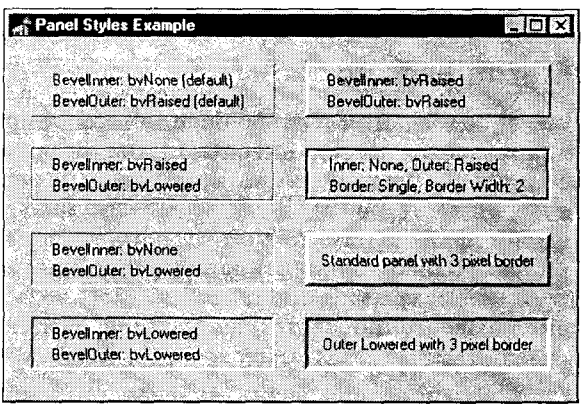


Рис. 8.7
Примеры различного оформления панелей

Компонент Panel настолько универсален, что вам потребуется некоторое время, чтобы обнаружить все его возможные применения.

И это еще не все...

К сожалению, нам не хватит времени и места, чтобы обсудить все компоненты C++Builder. Вам уже знаком компонент Image по 6-му дню, когда вы создавали программу Picture Viewer. О компоненте Bevel вы узнали в тот же день, создавая диалоговое окно About, а о компоненте Shape — в 7-й день, когда выполняли упражнение по выравниванию компонентов. Это были лишь некоторые примеры использования тех компонентов, которые вам предстоит осваивать. Вы обязательно должны сами попрактиковаться с каждым из компонентов, чтобы определить для себя, какие из них вам больше нравятся или понадобятся в дальнейшем.

Однако существует еще одна группа компонентов, которую я должен обсудить с вами прежде, чем мы двинемся дальше — это компоненты Dialog.

Стандартные диалоговые окна

Windows содержит целый набор стандартных диалоговых окон, которые может использовать любое приложение. Сюда относятся следующие окна:

- File Open
- File Save
- Font
- Color
- Print
- Print Setup
- Find
- Replace

Стандартные диалоговые окна вы обнаружите во вкладке Dialogs палитры компонентов. Эти компоненты относятся к невизуальным, поскольку они не имеют визуального интерфейса на этапе разработки программы. В следующих подразделах будет рассмотрено каждое из этих диалоговых окон, за исключением окон Print и Print Setup, описание которых я отложу до того момента, когда буду говорить о печати.

Метод Execute

Для создания и отображения всех стандартных диалоговых окон Windows используется метод Execute(). Диалоговые окна отображаются в модальном режиме, за исключением немодальных окон Find и Replace. Метод Execute() возвращает true, если пользователь нажал кнопку OK, дважды щелкнул на имени файла или нажал клавишу Enter. Если пользователь щелкнул на кнопке Cancel, нажал клавишу Esc или закрыл диалоговое окно системной кнопкой закрытия окна, то Execute() возвращает значение false. Стандартное диалоговое окно часто реализуется следующим образом:

```
if (OpenDialog->Execute()) {
  // пользователь нажал клавишу OK, поэтому используем выбранный файл
  Memo->Lines->LoadFromFile(OpenDialog->FileName);
}
return;
```

Этот код выводит диалоговое окно File Open, в котором пользователь выбирает нужный файл. Если пользователь нажал кнопку OK, то будет выполнен код внутри блока if, в результате чего содержимое файла будет загружено в компонент Memo. Если клавиша OK не была нажата, то код внутри блока if игнорируется и никаких действий предпринято не будет.



Код предыдущего фрагмента — еще один пример сокращенного синтаксиса Си++. Первая строка:

```
if (OpenDialog->Execute()) {
```

эквивалентна следующей:

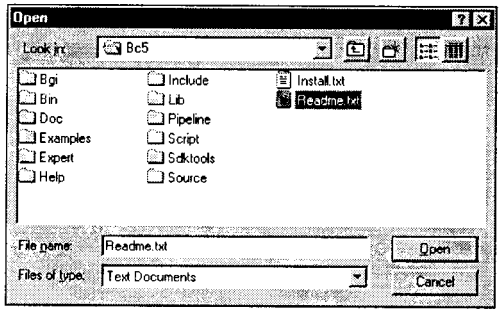
```
if (OpenDialog->Execute() == true) {
```

Вы можете использовать любую запись, но первая более предпочтительна.

Диалоговые окна File Open и File Save

Диалоговые окна File Open (Открыть файл) и File Save (Сохранить файл) имеют несколько общих свойств. Диалог File Open используется для открытия файла в вашем приложении (см. рис. 8.8). Это окно инкапсулировано в компоненте OpenDialog. Через диалоговое окно File Save вы получаете от пользователя имя, под которым должен быть сохранен файл. Это окно используется и как диалоговое окно Save As. Диалоговое окно File Save инкапсулировано компонентом SaveDialog.

Рис. 8.8
Типичное диалоговое окно File Open



Диалоговые окна Windows для работы с файлами достаточно просты в использовании, но имеют несколько особенностей, которые требуют пояснения. В следующих разделах мы поговорим о свойствах, которые являются специфическими для этих окон.



Компоненты OpenDialog и SaveDialog просто возвращают имя файла, введенное пользователем. Написание кода, который выполняет нужные действия над выбранным файлом, ложится на плечи программиста.

Свойство DefaultExt

Свойство DefaultExt определяет расширение, которое по умолчанию будет использоваться в диалоговом окне. *Расширение по умолчанию* — это расширение, которое автоматически добавляется к имени файла, если пользователь не задал расширение явно.

Свойство FileName

Свойство FileName наиболее очевидное из всех свойств диалогового окна Windows для работы с файлами. Оно содержит имя файла, который выбирает пользователь. Установите значение этого свойства перед вызовом диалогового окна, если вы хотите, чтобы при появлении окна на экране поле ввода уже содержало имя файла. После того, как пользователь нажмет OK для закрытия диалогового окна, это свойство будет содержать полный путь и имя выбранного файла.

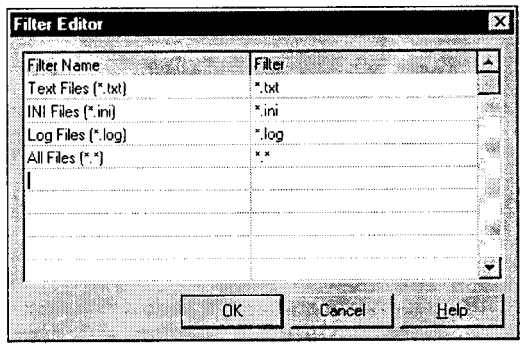
Свойство Files

Свойство Files является экземпляром класса TStrings и доступно только для чтения. Оно содержит список выбранных файлов в том случае, когда разрешено групповое выделение.

Свойство Filter

Свойство Filter содержит список типов файлов, которые пользователь может выбирать. Типы файлов выводятся в комбинированном списке File of type:. С помощью этого свойства вы можете задать типы файлов, которые нужны вашему приложению. Например, простая программа редактирования текстов могла бы иметь фильтры для типов TXT, INI и LOG (я называю только некоторые). При разработке программы вы легко можете задать нужные фильтры, используя диалоговое окно редактора фильтров (Filter Editor). Чтобы вызвать редактор фильтров, дважды щелкните в столбце Value инспектора объектов рядом со свойством Filter. На рис. 8.9 показано окно редактора фильтров для диалогового окна File Open.

Рис. 8.9
Диалоговое окно редактора фильтров



Столбец Filter Name содержит текстовое описание типа файла. В столбце Filter указывается маска, которая будет использоваться при отборе файлов.

Хотя вы можете ввести строку фильтра непосредственно в столбец Value инспектора объектов, удобнее всего использовать редактор фильтров. Если вы используете одиночный фильтр, то можете определить его непосредственно в столбце Value. Отделяйте описание и фильтр друг от друга символом «|». Например, чтобы задать один фильтр для всех типов файлов, вы можете ввести следующую строку:

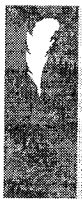
```
All Files (*.*) | *.*
```

Свойство FilterIndex

Свойство FilterIndex используется для установки фильтра, который будет выбран при первом отображении диалогового окна. Значения этого индекса начинаются не с нуля, как вы могли бы ожидать. Первый фильтр в списке имеет номер 1, второй — номер 2, и т. д. В качестве примера обратитесь к рис. 8.9. Если вы хотите, чтобы при открытии окна использовался фильтр All Files, вы должны установить для свойства FilterIndex значение 4.

Свойство InitialDir

Свойство InitialDir определяет каталог, который будет использоваться в качестве начального при отображении диалогового окна File. Если для свойства InitialDir никакого значения не задано, будет использован текущий каталог (как это принято в Windows).



Профессионально сделанные программы для Windows отслеживают последние каталоги, которые пользователь использовал при открытии и сохранении файлов. Обычно эта информация хранится в реестре. Перед открытием диалога File Save или File Open, не забудьте указать в свойстве InitialDir имя того каталога, с которым пользователь работал в последний раз. После того, как пользователь выбрал нужный файл, обновите при необходимости запись в реестре.

Свойство Options

Свойство Options определяет характеристики диалогового окна File. Список всех опций слишком велик, чтобы привести его здесь полностью. Основные опции позволяют вам разрешать или запрещать создание новых файлов или каталогов, показывать или не показывать кнопку Help в диалоговом окне, работать с длинными именами файлов, разрешать или запрещать групповое выделение файлов и многое другое. Полную информацию вы найдете во встроенной справке C++Builder для компонентов OpenDialog и SaveDialog.

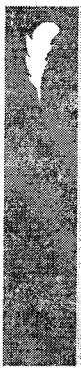
Свойство Title

Свойство Title используется для задания или чтения заголовка диалогового окна. Если заголовок не определен, то используется стандартное значение по умолчанию: Open для компонента OpenDialog и Save для компонента SaveDialog.



Диалоговое окно Save As есть ничто иное, как компонент SaveDialog, свойство Title которого имеет значение Save As.

С диалоговыми окнами File не связаны никакие события.



Вы можете вызвать диалоговое окно File Open (или любое из стандартных диалоговых окон Windows) во время выполнения программы, даже не помещая компонент OpenDialog в форму. Для этого вам нужно создать экземпляр класса TOpenDialog, а затем вызвать метод Execute():

```
TOpenDialog openDlg = new TOpenDialog(this);
if (openDlg->Execute()) {
  // необходимые действия
}
delete openDlg;
```

Если необходимо, вы можете установить значения любых свойств компонента OpenDialog перед вызовом метода Execute().

Диалоговое окно Color

Диалоговое окно Color используется для выбора цвета. После нажатия кнопки OK свойство Color будет содержать выбранное значение цвета. (Диалоговое окно Color было показано на рис. 8.1.) Окно Color, как и файловые диалоговые окна, не имеет связанных с ним событий.

Диалоговое окно Font

С помощью диалогового окна Font пользователь может выбрать один из шрифтов, доступных в его системе. Через свойство Device вы можете выбрать, какие шрифты отображать: экранные, шрифты принтера или оба типа шрифтов. Вы можете ограничить максимальный и минимальный размеры шрифта, которые может выбрать пользователь, изменяя свойства MaxFontSize и MinFontSize. Как и диалоговые окна File, окно Font имеет свойство Options с большим набором опций, задающих различные режимы работы диалога Font.

После нажатия пользователем кнопки OK, свойство Font будет содержать всю информацию, необходимую для использования нового шрифта. На рис. 8.10 показано диалоговое окно Font в конфигурации по умолчанию.

Диалоговое окно Font имеет единственное событие, OnApply, которое происходит при нажатии пользователем кнопки Apply. Кнопка Apply не будет присутствовать в диалоговом окне Font, пока не вы не создадите обработчик события OnApply, содержащий какой-либо исполняемый код.

Диалоговые окна Find и Replace

Диалоговые окна Find и Replace дают пользователю возможность выполнять поиск и замену текста с использованием целого ряда опций. Диалоговое окно Find инкапсулировано в компоненте VCL FindDialog, а диалоговое окно Replace представлено компонентом ReplaceDialog. На рис. 8.11 показано диалоговое окно Replace, которое содержит все элементы окна Find, а также дополнительные возможности по замене.

Рис. 8.10
Диалоговое окно Font

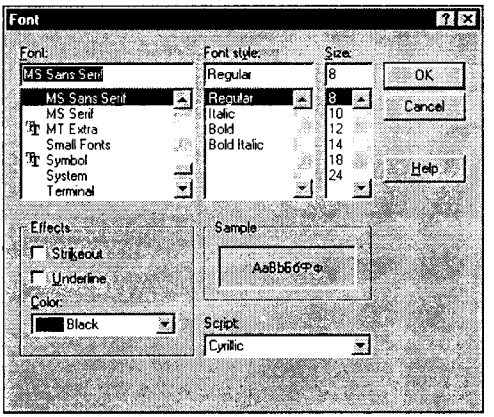
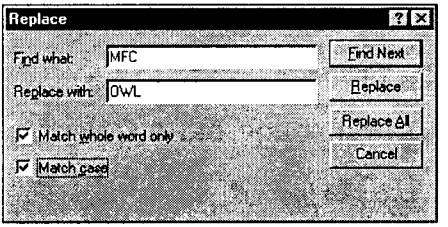


Рис. 8.11
Диалоговое окно Replace



К основным свойствам компонентов FindDialog и ReplaceDialog относятся FindText (текст, который нужно найти), ReplaceText (текст, которым нужно заменить найденный текст) и Options. Очевидно, что компонент FindDialog не

имеет свойства ReplaceText. Свойство Options содержит информацию о режимах, установленных пользователем перед нажатием кнопки Find Next, Replace или Replace All.

Метод Execute() компонентов FindDialog и ReplaceDialog несколько отличается от метода Execute() других стандартных компонентов Dialog. Во-первых, диалоговые окна Find и Replace не являются модальными. Метод Execute() возвращает управление сразу после вывода диалога на экран. Поскольку окно немодальное, то возвращаемое методом Execute() значение не имеет смысла (оно всегда будет true). Для контроля своего состояния диалоговые окна Find и Replace используют события OnFind, OnReplace и свойство Options. Событие OnFind инициируется при нажатии кнопки Find Next. Компонент ReplaceDialog дополнительно к событию OnFind имеет событие OnReplace, связанное с нажатием кнопок Replace и Replace All. Используйте эти события для определения того момента, когда пользователь запросил поиск или замену. Ваши программы должны также считывать свойство Options, чтобы определить заданный пользователем режим поиска или замены.

Заключение

Сегодня вы познакомились с внутренним устройством некоторых из основных компонентов C++Builder. Вы изучили общие свойства компонентов и узнали о некоторых специфических особенностях компонентов, основанных на элементах управления Windows. Это очень важно — разбираться в основных элементах управления Windows и представляющих их компонентах C++Builder. Мы завершили этот день обзором стандартных диалоговых окон Windows.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Если я изменю свойство Name компонента, используя для этого инспектор объектов, изменит ли C++Builder автоматически все ссылки на этот компонент в моей программе?

И да и нет. C++Builder изменит все ссылки на имя этого компонента в той части кода, который был сгенерирован им самим, и оставит без изменений код пользователя.

Очевидно, что компонент OpenDialog является видимым. Почему же он называется невизуальным компонентом?

Потому что он не отображается при разработке программы. Он становится видимым только во время выполнения, когда вы вызываете его при помощи метода Execute().

Почему это так принципиально — изменять свойство Name только через инспектор объектов?

Когда вы работаете с редактором форм, C++Builder генерирует код, опираясь на свойство Name. Если вы затем измените свойство Name, непосредственно редактируя исходные файлы или во время выполнения про-

граммы, то все ссылки на эту форму или компонент станут неверными. Это, скорее всего, приведет к невозможности компиляции вашей программы или ее краху во время выполнения.

Мне кажется, что я использую свойства своих компонентов чаще, чем методы. Это неправильно?

Вовсе нет. Фактически, именно так и разрабатывались компоненты VCL. Хорошо написанный компонент должен в максимальной степени использовать свойства. Именно по этой причине вы можете пользоваться методами компонента гораздо реже, чем свойствами. Используйте методы, когда это действительно необходимо, но во всех остальных случаях управляйте своими компонентами во время выполнения программы через свойства.

Я обрабатываю два события компонента: OnDblClick и OnClick. Каждый раз, когда я дважды щелкаю на компоненте, вызываются обработчики обоих событий. Почему так происходит?

Дело в том, что при двойном щелчке на компоненте Windows генерирует сообщения об одиночном и двойном щелчках. Вы не можете предотвратить эту реакцию, и ваша программа должна уметь разбираться в таких ситуациях.

Я хочу использовать возможности класса TStrings в своей программе для хранения массива строк. Но компилятор не позволяет мне использовать объект TStrings. Что мне делать?

Вам нужен объект TStringList. Класс TStringList разработан именно для этой цели.

Я хочу, чтобы в однострочном элементе редактирования текст был выровнен по правому краю, но не нахожу свойства Alignment у соответствующего компонента. Могу ли я все-таки выровнять текст по правому краю?

Нет, не можете. Все, что можно сделать в этой ситуации — это использовать компонент Memo, придав ему вид обычного элемента редактирования. Установите для свойства WantReturn компонента Memo значение false, высоту задайте равной высоте стандартного элемента редактирования (21 пиксел) и, конечно, для свойства Alignment установите значение taRightJustify. Такой компонент создаст полное впечатление однострочного элемента редактирования с выравниванием по правому краю.

В моей форме расположено несколько кнопок. Когда пользователь закрывает форму клавишей Esc, метод ShowModal() возвращает мне одно значение, а когда пользователь закрывает форму, щелкая на системной кнопке закрытия окна, я получаю от метода ShowModal() совсем другое значение. Почему так происходит?

Это означает, что в вашей форме есть кнопка, у которой свойство Cancel имеет значение true. Когда пользователь нажимает клавишу Esc, то значение ModalResult этой кнопки используется в качестве возвращаемого значения ShowModal(). Когда же пользователь закрывает форму при помощи системной кнопки, вы всегда будете получать значение mrCancel. Вы должны учитывать тот факт, что форма может быть закрыта двумя различными способами.

Вопросы

1. Можно ли изменять свойство Name компонента во время выполнения программы?
2. Какое свойство используется для того, чтобы разрешить или запретить доступ к компоненту?
3. Как можно во время выполнения программы сообщить пользователю, что доступ к кнопке запрещен?
4. В чем состоит различие между длинной и короткой подсказками?
5. Назовите три из четырех методов, которые могут использоваться для принудительной перерисовки элемента управления.
6. Сколько существует типов комбинированных списков?
7. Как используется свойство ModalResult в компонентах кнопок?
8. Какой компонент обычно используется в качестве контейнера для других компонентов?
9. Какое значение возвращает метод Execute() компонента OpenDialog, если пользователь закрывает диалоговое окно нажатием кнопки OK?
10. Как из компонента SaveDialog сделать диалоговое окно Save As?

Упражнения

1. Создайте программу, содержащую два компонента редактирования. Сделайте так, чтобы по мере ввода данных в первый элемент, они появлялись во втором элементе редактирования.
2. Создайте программу с окном списка. Напишите код для загрузки списка из текстового файла перед отображением главного окна программы.
3. Добавьте компонент редактирования в программу из упражнения 2. После того, как пользователь выбрал элемент списка, его текст должен появляться в компоненте редактирования.
4. Добавьте кнопку в программу из упражнений 2 и 3. После нажатия этой кнопки программа должна помещать в список в качестве нового элемента строку из элемента редактирования.
5. Создайте программу, которая использует компонент RadioGroup с четырьмя кнопками в группе. Добавьте компонент Label, текст которого должен изменяться в зависимости от того, какая радиокнопка нажата.
6. Создайте программу, в которой форма имеет заголовок, располагающийся всегда по центру, независимо от размеров окна программы.
7. Измените программу из упражнения 6 так, чтобы нажатием кнопки можно было изменить шрифт заголовка на любой шрифт, доступный в системе.
8. Вернитесь к программе Picture Viewer, созданной вами в 5-й день. Измените компоненты OpenDialog и SaveDialog таким образом, чтобы их свойства Filter позволяли выбирать растровые файлы (*.bmp), метафайлы (*.wmf), значки (*.ico) и все файлы (*.*).

Создание приложений в C++Builder



Кент Рейсдорф

В C++Builder имеется целый ряд вспомогательных инструментов, предназначенных для создания форм, диалоговых окон и приложений. Этот день мы посвятим изучению следующих тем:

- Репозиторий объектов
- Мастер диалогов
- Мастер приложений
- Добавление функций и членов-данных в вашу программу
- Использование ресурсов в приложениях C++Builder

Мы начнем день с рассмотрения репозитория объектов, в котором C++Builder хранит для повторного использования ранее созданные формы и приложения. Следующей темой будут мастера — инструментальные средства, которые шаг за шагом проведут вас через процесс разработки. Вы будете давать необходимые указания, а C++Builder построит на основе этой информации форму или приложение. Мастера — это мощный инструмент для быстрой разработки приложений. Мы закончим этот день обсуждением возможностей использования ресурсов в приложениях C++Builder. Итак, начнем.

Работа с репозиторием объектов

Используя диалоговое окно репозитория объектов, вы можете выбрать нужные вам предопределенные объекты и использовать их в своих приложениях.



Репозиторий объектов — это текстовый файл, который содержит информацию, отображаемую в диалоговом окне Object Repository. Далее я буду называть диалоговое окно репозитория объектов и файл репозитория просто репозиторием объектов.

Репозиторий объектов дает вам возможность делать следующее:

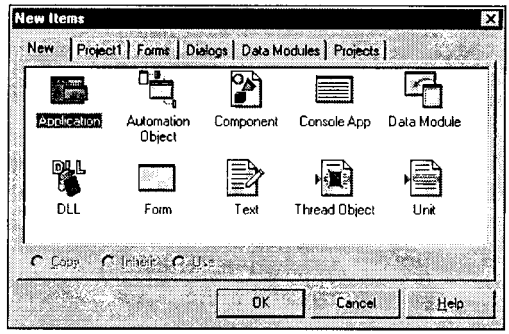
- Выбирать предопределенные приложения, формы или диалоговые окна для использования в своем приложении
- Добавлять в репозиторий разработанные вами формы, диалоговые окна и приложения
- Включать в ваши программы другие объекты, например, ASCII-файлы и дополнительные модули исходного кода

- Управлять модулями данных
- Создавать новые компоненты
- Вызывать мастеров для помощи в создании диалоговых окон или приложений

Страницы и опции репозитория объектов

Репозиторий объектов автоматически вызывается при выборе пункта главного меню File | New. На рис. 9.1 показано окно репозитория объектов после выбора File | New, если ни один проект при этом не открыт.

Рис. 9.1
Окно репозитория объектов





Вам может показаться странным, что диалоговое окно репозитория объектов имеет заголовок New Items (Новые объекты), а окно конфигурации репозитория объектов называется Object Repository (Репозиторий объектов). Зачем это сделано, непонятно.

Окно репозитория объектов имеет несколько страниц, на каждой из которых содержатся различные объекты, которые вы можете включать в приложения. Как видно из рис. 9.1, при выводе окна репозитория объектов выбранной оказывается вкладка New (Новый). В таблице 9.1 перечислены страницы репозитория и описаны объекты, которые вы найдете на каждой из них.

Таблица 9.1. Страницы репозитория объектов

Страница/Вкладка	Описание
New	Позволяет создать новое приложение, консольное приложение, форму или модуль исходного кода, а также более сложные объекты, такие как DLL, компоненты и модули данных.
Forms	Позволяет создавать стандартные формы из предопределенных форм, таких как окно About, форма для работы с базами данных или форма быстрого отчета (Quick Report).
Dialogs	Предоставляет на выбор несколько основных типов диалоговых окон. На этой же странице находится мастер диалогов.
Data Modules	На этой странице вы можете выбрать нужный для вашего приложения тип модуля данных.
Projects	Содержит шаблоны проектов, которые вы можете выбирать для начала работы над приложением. Здесь же находится мастер приложений.



Если вы вызываете репозиторий объектов при открытом проекте, то увидите в окне дополнительную вкладку, которая будет иметь имя вашего проекта. Если вы щелкнете на этой вкладке, то появится страница, содержащая все объекты текущего проекта. Это позволит вам повторно использовать форму или другой объект, просто выбрав его из репозитория объектов.

В нижней части каждой страницы репозитория объектов находятся три радиокнопки. Эти кнопки, с названиями Copy, Inherit и Use, определяют порядок использования выбранного объекта.



В зависимости от выбранного объекта, некоторые (или все) радиокнопки могут быть недоступны. Например, на странице New недоступны все три кнопки. Это происходит потому, что Copy – единственная опция, доступная для объектов на этой странице, поэтому C++Builder блокирует все остальные варианты и автоматически применяет Copy.

Опция Copy

Когда вы выбираете радиокнопку Copy, C++Builder создает копию выбранного объекта и помещает ее в ваше приложение. Затем вы можете изменять этот объект, как сочтете нужным. Когда вы модифицируете новый объект своего приложения, исходный объект в репозитории остается неизменным.

Для иллюстрации, давайте представим, что у вас есть часто используемая форма (имеется в виду форма в традиционном смысле, т.е. бланк) напечатанная на бумаге, например, календарный план работы. Предположим, что вам потребовалось внести в форму новые данные. Вы не станете заполнять оригинал формы, потому что иначе он станет непригоден для последующего использования. Вместо этого вы сделаете копию формы, а оригинал положите куда-нибудь в безопасное место. Затем вы заполните копию формы необходимой информацией. Создание копии объекта из репозитория — это то же самое. Вы свободно можете изменять копию, в то время как оригинал остается лежать в надежном месте. Создание копии — самый безопасный способ использования объектов.

Опция Inherit

Наследование похоже на копирование, но с одним важным отличием: новый объект остается связанным с базовым объектом. Если вы измените базовый объект, созданный на его основе объект также будет изменяться в соответствии с изменениями базового объекта. Обратное, однако, неверно. Изменение нового объекта не оказывает никакого влияния на базовый объект.

Для иллюстрации этого способа использования объектов, рассмотрим следующий пример. При обработке информации, данные из электронной таблицы довольно часто вводятся в текстовый процессор, чтобы затем написать отчет. При этом данные обычно остаются связаны с электронной таблицей при вставке из буфера обмена или импорте таблицы в текстовый процессор. В этом случае любое изменение данных в электронной таблице будет автоматически отражено в документе. Точно так же изменения, внесенные в базовую форму, будут автоматически отражаться во всех производных фор-

мах. Используйте опцию Inherit, когда вы хотите создать несколько форм, основанных на одной стандартной форме, которая впоследствии может быть изменена. Любые изменения в базовой форме вызовут соответствующие изменения во всех производных формах.

Опция Use

Опция Use используется не очень часто. Когда вы выставляете опцию Use, то объект репозитория открывается для редактирования. Эта опция применяется, если вам нужно внести изменения в ранее сохраненный объект. В предыдущем подразделе я говорил о том, что изменения, сделанные в базовой форме, будут отражены во всех производных формах. Для внесения изменений в саму базовую форму, вы должны открыть ее в репозитории объектов с опцией Use.

Что дальше?

Действия, выполняемые после выбора объекта из репозитория, определяются несколькими факторами: типом выбранного объекта, способом его использования (Copy, Inherit или Use) и тем, открыт ли какой-нибудь проект в данный момент. Если вы создаете новое приложение из репозитория объектов в то время, когда у вас уже есть открытое приложение, то перед открытием нового проекта вам будет предложено сохранить текущий проект (если в этом есть необходимость).



Пункт меню File | New Application (Файл | Новое приложение) служит для быстрого открытия нового приложения. Это эквивалентно выбору пункта меню New, а затем объекта Application из репозитория объектов. Аналогичным образом, пункты меню New Form (Новая форма), New Data Module (Новый модуль данных) и New Unit (Новый модуль) являются сокращенными эквивалентами для создания объектов из репозитория.

Новая форма из репозитория объектов создается по-разному, в зависимости от того, открыт ли какой-нибудь проект в настоящее время. Если есть открытый проект, то форма добавляется в приложение в виде пары форма/модуль. Если открытого проекта нет, то новая форма и модуль будут созданы как автономные. Используйте эту возможность для создания новой базовой формы, которую вы хотите включить в репозиторий объектов.

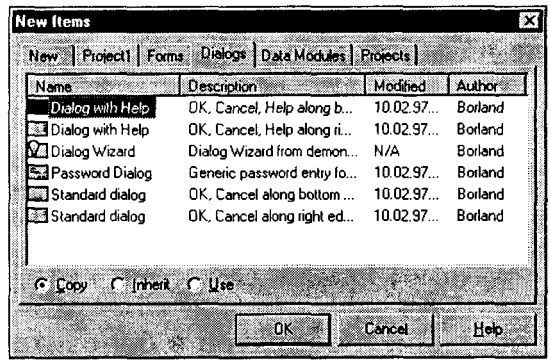
Если вы создаете новый модуль или текстовый файл, то этот файл будет открыт в редакторе кода (если это программный модуль, то он добавляется к текущему проекту). Вам может потребоваться новый текстовый файл по нескольким причинам. Например, предположим, что вы хотите добавить в свое приложение файл конфигурации (.INI-файл). Чтобы создать этот файл, вполне можно использовать репозиторий объектов. Новый модуль создается в том случае, если вам необходимо добавить в проект новый исходный файл, не связанный с формой.

Выбор новой DLL или консольного приложения приведет к созданию нового проекта с соответствующими установками. Если вы создаете новые объекты Automation, Component или Thread, то в дополнительном диалоговом окне вы должны будет определить некоторые параметры, необходимые для создания объекта.

Виды окна репозитория объектов

Окно репозитория объектов — это элемент управления Win32 для просмотра списка, похожий на окно в правой части Windows Explorer (в котором перечисляются файлы). Соответственно, это окно имеет несколько возможных видов: Large Icons (Большие значки), Small Icons (Маленькие значки), List (Список) и Details (Детали). По умолчанию окно имеет вид Large Icons. Чтобы изменить вид окна репозитория объектов, щелкните правой кнопкой на репозитории объектов и выберите нужный пункт в контекстном меню. На рис. 9.2 показано окно репозитория объектов в виде Details с выбранной страницей Dialogs.

Рис. 9.2
Детальный вид окна репозитория объектов



Контекстное меню репозитория объектов позволяет также задать несколько опций сортировки. Вы можете отсортировать объекты по именам, описаниям, датам или по авторам этих объектов.



Когда окно репозитория объектов имеет вид Details, вы можете щелкнуть на заголовке столбца (Name, Description, Date или Author), чтобы немедленно отсортировать объекты по данному признаку.

Создание новых объектов из репозитория

Чаще всего репозиторий объектов используется для создания новых объектов. В качестве примера, давайте создадим простое приложение, в котором используются главная форма, диалоговое окно About и еще одна форма. Выполните последовательно следующие действия:

1. Убедитесь в том, что у вас не осталось открытых приложений. Выберите в главном меню пункт File | New. На экране появится окно репозитория объектов.
2. Щелкните на значке Application и нажмите OK, чтобы создать новое приложение. На экране появится пустая форма нового приложения.
3. Поместите в форму две кнопки. Измените свойство Caption одной из кнопок на About..., а свойство Caption другой кнопки на Display Form2. При желании измените свойства Name.
4. Снова выберите пункт меню File | New. На экране опять появится репозиторий объектов.

5. Щелкните на вкладке Forms в репозитории объектов.
6. Выберите объект — окно About. Убедитесь в том, что нажата радиокнопка Copy, и нажмите OK, чтобы создать новую форму. Измените, если необходимо, свойства формы.
7. Измените по своему усмотрению окно About. (Введите информацию о себе, измените значок, размер и положение окна, и т. д.)
8. Снова выберите пункт меню File | New. Окно репозитория объектов появится в третий раз.
9. Щелкните на вкладке Forms и выберите объект Dual List Box. Нажмите OK, чтобы закрыть репозиторий объектов. Вы увидите на экране форму с двойным списком.
10. Напишите обработчики событий для двух кнопок, которые выводят на экран окно About и вторую форму.
11. Откомпилируйте, запустите и протестируйте вашу программу.

Конечно, эта программа не делает ничего полезного, но она служит иллюстрацией того, как вы можете использовать репозиторий объектов для быстрого создания прототипа приложения. Со временем вы добавите свои собственные объекты в репозиторий, что позволит вам работать действительно эффективно! Давайте посмотрим, как это делается.

Добавление объектов в репозиторий

Репозиторий объектов не был бы таким эффективным инструментом, если бы у вас не было возможности добавлять в него свои собственные объекты. Но вы не должны относиться к этому только как к возможности — вы просто обязаны так поступать. Включая часто используемые объекты в репозиторий, вы сможете работать более эффективно, что, в свою очередь, повышает вашу ценность как программиста. Не имеет смысла каждый раз изобретать колесо. После того, как вы создали приложение, форму или другой объект, помещайте их в репозиторий, чтобы иметь возможность использовать эти объекты повторно. Конечно, не стоит помещать в репозиторий все подряд, но объекты, которые вы используете часто, обязательно должны туда попадать.

Вы можете создать объект специально для помещения в репозиторий, или добавить его туда в процессе разработки приложения. (Термин *объект* довольно широк, поэтому, для большей наглядности, далее я использую конкретный пример.) Предположим, что вы создаете форму окна About для своего приложения. Внезапно вы понимаете, что следовало бы сохранить эту форму, чтобы использовать ее во всех своих программах. В этой форме указано название вашей компании, приводится изображение логотипа, информация об авторских правах, причем все это так удачно расположено, что было бы досадно каждый раз начинать все заново. Никаких проблем — поместите эту форму в репозиторий. Для этого сначала сохраните ее (если вы не сохраните форму, вам будет предложено это сделать прежде, чем вы сможете продолжить указанные действия). Затем щелкните правой кнопкой мыши где-нибудь на форме и выберите пункт Add to Repository (Добавить в репозиторий) в контекстном меню редактора форм. После этого появится диалоговое окно, показанное на рис. 9.3.

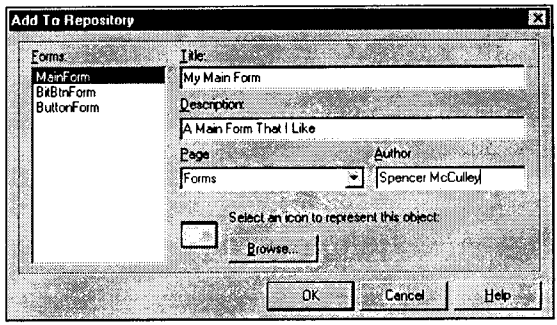


Рис. 9.3
Диалоговое окно Add To Repository

В списке форм в левой части этого диалогового окна перечислены формы и другие объекты приложения (например, модули данных). Выберите ту форму, которую вы хотите добавить в репозиторий объектов.



Активная форма редактора уже будет выделена в списке Forms диалогового окна Add To Repository.

Теперь введите название объекта. Название будет выводиться в репозитории объектов под значком. В поле Description (Описание) вы можете ввести дополнительную информацию об объекте. Это описание будет выводиться на экран, если для окна репозитория объектов выбран вид Details (см. рис. 9.2). В поле Author вы должны ввести имя разработчика данного объекта. Здесь можно ввести свое имя, название компании или любое другое имя.



В предопределенных объектах репозитория, которые поставляются вместе с C++Builder, в качестве имени разработчика используется «Borland».

Поле Page определяет страницу репозитория объектов, на которой будет располагаться новый объект. Вы можете выбрать одну из существующих страниц или ввести имя новой страницы. Если страницы с таким именем не существует, то C++Builder создаст новую страницу. В нижней части диалогового окна находится кнопка Browse... (Просмотр), нажав на которую, вы можете выбрать значок для представления объекта.



Вы можете выбрать подходящий значок из каталогов CBuilder\Images\Icons или CBuilder\Objrepos. В каталоге CBuilder\Objrepos собраны значки, которые C++Builder использует для объектов, помещаемых в репозиторий.

Как только вы заполнили все поля и выбрали значок, можно нажать кнопку OK для добавления объекта в репозиторий. Объект появится в репозитории на той странице, которую вы определили. Теперь вы можете использовать этот объект, как только он вам понадобится. Как видите, добавить объект в репозиторий почти так же просто, как и использовать объект из репозитория.



Когда вы добавляете объект в репозиторий, C++Builder создает раздел с описанием этого объекта в файле репозитория. Эта информация включает путь, где находятся форма и исходный файл объекта. Если вы переместите или удалите форму объекта и/или исходный файл, то не сможете использовать этот объект через репозиторий.

Добавление проектов в репозиторий

Процесс добавления проектов в репозиторий мало отличается от добавления форм. Чтобы включить в репозиторий проект, выберите пункт меню Project | Add To Repository (Проект | Добавить в репозиторий). Появится диалоговое окно Add To Repository, похожее на окно, появляющееся при добавлении объектов в репозиторий, за исключением того, что здесь отсутствует список форм. Заполните поля необходимой информацией (название, описание, автор и т. д.), нажмите кнопку OK, и проект будет добавлен в репозиторий.

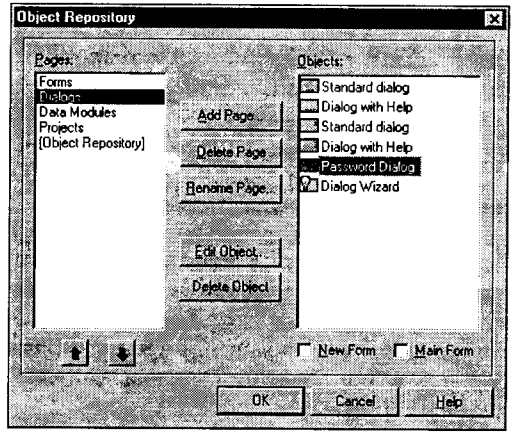
После того, как вы освоите работу с C++Builder, вам следует создать оболочку приложения, в которой будут учтены те особенности, которые вы часто используете в своих приложениях. Каждый раз, когда вы будете начинать работу над новым стандартным приложение, создавайте копию программной оболочки из репозитория объектов. Тем самым вы сразу получите готовые к использованию меню, оперативную панель, окно About и другие стандартные диалоговые окна. После создания нового приложения таким способом, вы можете изменять его, как и любое другое приложение (добавлять новые формы, удалять любые ненужные формы и т. д.).

Конфигурирование репозитория объектов

Вы можете управлять страницами и объектами в репозитории, используя диалоговое окно конфигурации репозитория объектов.

Чтобы вызвать диалоговое окно конфигурации репозитория, выберите в главном меню пункт Options | Repository (Параметры | Репозиторий) или, если окно репозитория открыто, пункт Properties (Свойства) в контекстном меню репозитория. Вы увидите диалоговое окно, показанное на рис. 9.4.

Рис. 9.4
Диалоговое окно конфигурации репозитория объектов



Используя это диалоговое окно, вы можете удалять объекты и страницы из репозитория объектов, перемещать объекты с одной страницы на другую, изменять порядок расположения страниц в репозитории и выполнять другие операции. В левой части окна находится список страниц репозитория объектов, который так и называется — Pages (Страницы). Когда вы выберете одну из страниц списка Pages, в правом списке (с заголовком Objects) появятся объекты, находящиеся на этой странице.



Список Pages имеет две важных особенности. Во-первых, обратите внимание на то, что страница New, которая всегда отображается первой при вызове репозитория объектов, здесь не показана. Параметры страницы New фиксированы и не могут быть изменены. Во-вторых, в этом списке присутствует элемент с названием [Object Repository]. Под этим именем собраны объекты со всех страниц репозитория.

Управление объектами

Чтобы отредактировать, удалить или переместить объект, вы должны сначала выбрать его. Для этого вам нужно щелкнуть на объекте в списке объектов. Выбрав объект, вы можете начать его редактировать, щелкнув на кнопке Edit Object (Редактировать объект). В процессе редактирования вы можете изменить имя объекта, описание и имя автора, равно как и страницу, на которой этот объект расположен.



Чтобы быстро перейти в режим редактирования объекта, дважды щелкните на нем в списке объектов.

Вы можете удалить объект, предварительно выбрав его и нажав затем кнопку Delete Object (Удалить объект). Если вы подтвердите свое намерение, объект будет удален со страницы и из репозитория.



Когда объект удаляется из репозитория, он удаляется из файла репозитория объектов и больше недоступен на страницах репозитория. Однако сам файл формы и исходный файл, которые описывают объект, остаются на вашем жестком диске.

Объекты можно перемещать с одной страницы на другую простым перетаскиванием из списка объектов в список страниц. Отпустите объект на странице, в которую вы хотите его переместить, и он окажется на этой странице.

Управление страницами

В предыдущем подразделе мы рассмотрели редактирование, удаление и перемещение отдельных объектов. В диалоговом окне конфигурации репозитория объектов вы можете также добавлять, удалять или перемещать целые страницы. Для того, чтобы удалить страницу, вы должны предварительно удалить с нее все объекты. После этого, щелкнув на имени страницы в списке и нажав кнопку Delete Page (Удалить страницу), вы сможете удалить эту страницу. Проверив, что страница действительно пуста, C++Builder удалит ее из репозитория объектов.

Щелкнув на кнопке Add Page (Добавить страницу), вы можете добавить в репозиторий новую страницу. В диалоговом окне, отображающемся после нажатия кнопки, вы должны указать имя новой страницы. После задания имени новой страницы, щелкните на кнопке OK, и новая страница появится в списке. Переименование страницы производится, по существу, тем же самым способом. Вы выбираете страницу и нажимаете кнопку Rename Page (Переименовать страницу), после чего появляется диалоговое окно, в котором вы задаете новое имя страницы.

Вы можете изменить порядок расположения страниц в репозитории объектов. Чтобы переместить страницу на новую позицию, щелкните на этой странице, чтобы выбрать ее. После этого нажмите кнопку со стрелкой вниз или кнопку со стрелкой вверх, расположенную под списком страниц, чтобы переместить страницу вниз или вверх в списке. При желании вы можете просто перетащить страницу на новое место.

Задание форм и проектов по умолчанию

В диалоговом окне конфигурации репозитория объектов вы можете определить три объекта, заданных по умолчанию:

- Форма по умолчанию, которая будет использоваться при выборе пункта меню File | New Form.
- Форма по умолчанию, которая будет использоваться в качестве главной формы при выборе пункта меню File | New Application.
- Проект по умолчанию, создаваемый при выборе пункта меню File | New Project.

Вы должны были обратить внимание на то, что в зависимости от выбранного вами объекта, ниже списка объектов появляются один или два флажка. Если вы выбрали форму, то появляются флажки New Form (Новая форма) и Main Form (Главная форма). Если же вы выбрали проект, то появляется флажок New Project (Новый проект). Определить форму или проект по умолчанию очень просто. Предположим, что вы создаете главную форму, которую хотите сделать формой по умолчанию и использовать при создании новых приложений. Выберите форму из списка объектов и установите флажок Main Form. После того, как вы нажмете кнопку OK, эта форма станет формой по умолчанию. Аналогично, если у вас есть проект, который вы хотите сделать проектом по умолчанию, то сначала найдите его в диалоговом окне конфигурации репозитория объектов, щелкните на нем, а затем установите флажок New Project. Теперь при выборе пункта меню File | New Application будет открываться заданный проект.

Создание форм и приложений с помощью мастеров

В C++Builder встроены два мастера, которые предназначены для упрощения процесса создания приложений. Мастер диалогов помогает создавать диалоговые окна, а мастер приложений оказывает помощь в создании основы будущего приложения.



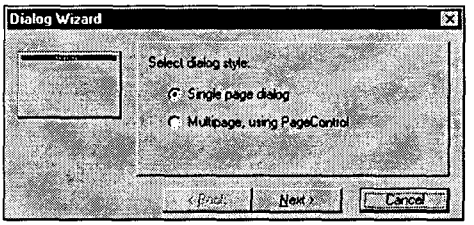
В C++Builder имеется пара других диалоговых окон, которые также называются мастерами. Например, при выборе нового компонента из репозитория объектов, на экране появляется диалоговое окно, в котором вы должны указать сведения, необходимые для создания этого компонента. Поскольку данные окна вряд ли можно считать полноценными мастерами, я не буду их рассматривать вместе с мастерами диалогов и приложений.

Теперь давайте поговорим о каждом из мастеров в отдельности.

Мастер диалогов

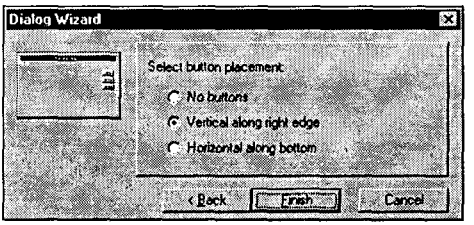
По правде говоря, у мастера диалогов не так уж много работы, потому что практически любое диалоговое окно требует окончательной настройки в редакторе форм. Запускается мастер диалогов из репозитория объектов. Сначала выберите в главном меню пункт File | New, чтобы открыть репозиторий объектов. Затем перейдите на страницу репозитория Dialogs (Диалоги) и дважды щелкните на значке мастера диалогов. Вы увидите окно мастера диалогов, показанное на рис. 9.5.

Рис. 9.5
Мастер диалогов



Вы можете создать диалоговое окно из одной страницы или многостраничное диалоговое окно (с вкладками). Значок в левой части диалогового окна мастера будет показывать внешний вид создаваемого вами окна. Если вы решили создать одностраничное диалоговое окно, то при нажатии кнопки Next вы увидите следующую страницу мастера диалогов (см. рис. 9.6).

Рис. 9.6
Вторая страница мастера диалогов



Эта страница позволяет вам решить, будете ли вы использовать кнопки в диалоговом окне, и если да, то выбрать их расположение: в правой или нижней части окна. Как вы можете догадаться, посмотрев на рис. 9.6 и увидев кнопку Finish (Конец), это последняя страница мастера диалогов в том случае, если вы создаете одностраничное диалоговое окно. Выбрав вариант размещения кнопок, нажмите кнопку Finish, чтобы C++Builder создал для вас диалоговое окно.

Новое диалоговое окно отображается в редакторе форм. Свойство окна BorderStyle будет иметь значение bsDialog, что принято для форм, исполь-

зуемых в качестве диалоговых окон. После того, как мастер диалогов создал основу для вашего окна, вы можете добавить необходимые функциональные возможности в редакторе форм.

Если вы решите создать диалоговое окно с вкладками, то вторая страница мастера будет выглядеть так, как показано на рис. 9.7.

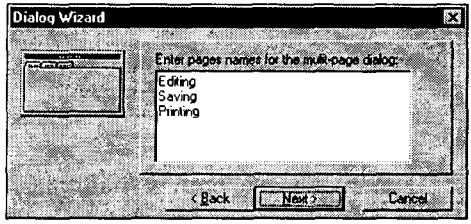


Рис. 9.7
Мастер диалогов в процессе создание многостраничного диалогового окна

На этой странице расположен многострочный элемент редактирования, в котором вы можете задать названия вкладок своего диалогового окна. Текст для каждой вкладки вводится с новой строки, как показано на рис. 9.7. Закончив ввод, нажмите кнопку Next, и вы увидите последнюю страницу мастера диалогов, показанную на рис. 9.6. Выберите, если это необходимо, нужное расположение кнопок и нажмите кнопку Finish, после чего C++Builder создаст многостраничное диалоговое окно.



Мастер диалогов удобнее всего использовать для создания многостраничного диалогового окна. Если вам нужно диалоговое окно из одной страницы, то проще выбрать одно из предопределенных диалоговых окон в репозитории объектов, чем создавать окно с помощью мастера.

Создание приложений с помощью мастера приложений

Мастер приложений — полезный инструмент, который поможет вам быстро разработать оболочку приложения. Чтобы создать новое приложение с помощью мастера приложений, выберите в главном меню пункт File | New. Когда появится окно репозитория объектов, щелкните на вкладке Projects (Проекты), а затем дважды щелкните на значке мастера приложений.



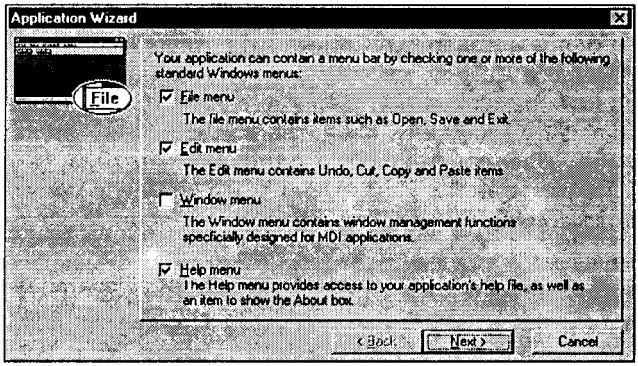
Если вы выберете в главном меню пункт New Application, то новое приложение будет создано на основе заданного по умолчанию проекта. Мастер приложений использоваться не будет.

Давайте последовательно рассмотрим каждую из страниц мастера приложений.

Первая страница: выбор меню

После запуска мастера приложений вы увидите его первую страницу, показанную на рис. 9.8.

Рис. 9.8
Первая страница мастера приложений



Эта страница позволяет вам выбрать элементы главного меню. Вы можете включить в него пункты File, Edit, Window и Help. Установите флажки для тех элементов меню, которые вы хотите увидеть в строке меню своей программы.



Меню Window обычно используются в приложениях MDI, поэтому вы, вероятно, не будете использовать его в программе SDI, кроме каких-то специфических случаев.



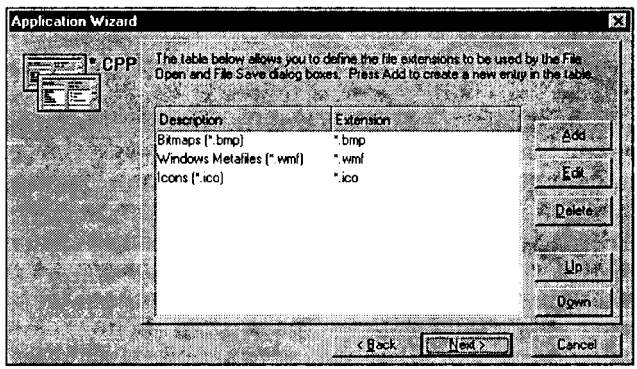
Список меню, который предлагает мастер приложений — это тот набор, который обычно используется в приложениях Windows. Не забывайте, что мастер приложений предназначен для того, чтобы помочь вам создать заготовку приложения. Вы должны затем взять эту основу и превратить ее в полноценное приложение.

После выбора нужных вашему приложению меню, нажмите кнопку Next, чтобы перейти на следующую страницу.

Вторая страница: задание фильтров диалогового окна File

Если вы выбрали на первой странице меню File, то следующей будет страница, представленная на рис. 9.9.

Рис. 9.9
Установка фильтров для диалоговых окон File



Эта страница используется для задания фильтров, которые ваше приложение будет использовать в диалоговых окнах File Open и File Save. (На

рис. 9.9 показано диалоговое окно мастера после того, как в него были добавлены фильтры.) Кнопка Add этого окна добавляет очередной фильтр. После нажатия на эту кнопку появляется диалоговое окно, в котором вы должны задать описание фильтра и сам фильтр. Фильтры вводятся точно так же, как и при задании свойства Filter для компонентов стандартных диалоговых окон. Введите текстовое описание фильтра, а затем маску файла (например, *.bmp). Кнопки Edit, Delete, Up и Down могут использоваться по мере необходимости для редактирования, удаления или перемещения фильтра в списке.

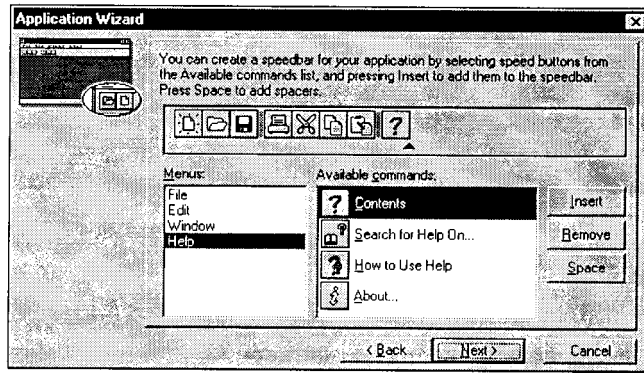


Вторая и третья страницы окна мастера приложений появляются только в том случае, если вы предварительно выбрали меню на первой странице. Например, вторую страницу вы увидите только после выбора на первой странице меню File.

Третья страница: настройка оперативной панели

Третья страница мастера приложений поможет вам настроить оперативную панель своего приложения. Это, вероятно, наиболее полезная возможность мастера приложений (данное утверждение, конечно, не означает, что все остальные возможности бесполезны). На рис. 9.10 показана третья страница мастера приложений после того, как оперативная панель уже создана.

Рис. 9.10
Настройка оперативной панели



Окно списка в левой части страницы, имеющее заголовок Menus, содержит четыре пункта меню, для которых вы можете выбрать кнопки быстрого доступа. Когда вы выбираете один из пунктов, то во втором списке, расположенном справа от списка Menus и имеющем заголовок Available commands (Доступные команды), выводятся доступные для этого меню кнопки. Чтобы добавить кнопку быстрого доступа, щелкните на ней в списке Available commands, а затем нажмите кнопку Insert (Вставить). Кнопка будет добавлена в оперативную панель, образец которой показан в верхней части страницы. Кнопка Space может использоваться для включения в панель разделителя. Используя разделитель, вы можете разбить ваши кнопки на группы. Продолжайте добавлять кнопки и разделители, пока панель не будет завершена. Если вы передумали и хотите удалить кнопку, просто щелкните на ней в панели и нажмите кнопку Remove (Удалить).

Если вы не выбрали для своего приложения какое-либо меню, то соответствующие ему кнопки не будут доступны. Например, если вы не добавляли меню Window, то список Available commands: будет пуст, когда вы щелкнете на элементе Window в списке Menus.

Мастер приложений позаботится даже о кратких подсказках для кнопок быстрого доступа. Вы еще не забыли, как создавали оперативную панель вручную в 7-й день, «Работа с редактором форм и редактором меню»? Сравните этот процесс с процессом создания кнопок быстрого доступа в мастере приложений. Вы согласитесь со мной, что создание оперативной панели с помощью мастера — это самый простой способ решения данной задачи.

Некоторые специфические приложения (а вы, возможно, захотите создать и такое) управляются только кнопками и не имеют меню. Но чтобы создать оперативную панель с помощью мастера приложений, вам обязательно нужно сначала создать меню. Чтобы обойти это ограничение, создайте меню в мастере приложений и скомпонуйте оперативную панель. После генерации приложения вы можете убрать из него меню, удалив компонент MainMenu.

Четвертая страница: установка параметров

На четвертой (и последней) странице мастера приложений вы можете задать имя программы, путь, где должен быть сохранен проект, и несколько заключительных параметров. Последняя страница мастера приложений показана на рис. 9.11.

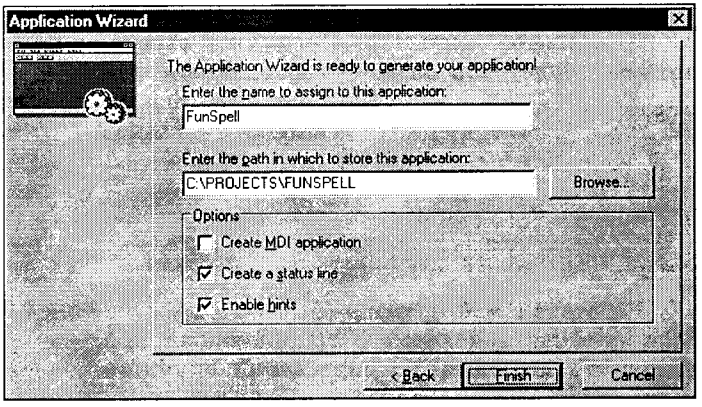


Рис. 9.11
Установка параметров в мастере приложений

В первом поле ввода на этой странице вы определяете имя приложения. Это не то имя, которое появляется в диалоговом окне Project Options, а имя файла, под которым C++Builder сохранит проект. Помимо этого имени, вам нужно будет задать имя приложения в диалоговом окне Project Options. Второе поле используется для определения каталога, в котором должен быть сохранен проект. Если вы не знаете точный путь, нажмите кнопку Browse справа от этого поля и выберите путь с помощью диалогового окна Select Directory (Выбор каталога).



Диалоговое окно Select Directory может использоваться как для создания каталога, так и для выбора уже существующего. Нажмите кнопку Browse, чтобы вызвать диалоговое окно Select Directory. Введите путь для каталога, который вы хотите создать, а затем нажмите кнопку OK или клавишу Enter. Если указанный каталог не существует, то C++Builder запросит у вас подтверждение на создание нового каталога.

В нижней части этой последней страницы вы можете воспользоваться тремя дополнительными опциями. Если вы создаете MDI-приложение, то щелкните на флажке с именем Create MDI Application (Создать MDI-приложение). (Мы рассматривали MDI-приложения в 6-й день, «Изучение IDE C++Builder: проекты и формы».) Два других флажка позволяют включить в приложение строку состояния и подсказки для ваших компонентов.

Если вы уверены, что правильно определили параметры своего приложения, нажмите кнопку Next. C++Builder создаст приложение в соответствии с вашими пожеланиями, сгенерировав максимально возможное количество кода. В результате получится не так много, но некоторая часть основного текста программы уже будет написана за вас. Например, если вы включили в приложение меню File, то обработчик события FileOpenClick() будет выглядеть примерно следующим образом:

```
void __fastcall TMainForm::FileOpen(TObject* Sender)
{
  if (OpenDialog->Execute())
  {
  //-- Add code to open OpenDialog.FileName --
  }
}
```

Диалоговое окно File Open уже вызвано; вам осталось написать код, который будет обрабатывать выбранный файл.



После создания проекта в мастере приложений, вы можете выбрать Project | Save To Repository (Проект | Сохранить в репозитории), чтобы сохранить этот проект для последующего использования. Тогда в следующий раз вам даже не потребуется вызывать мастер приложений. Если хотите, можете добавить окно About перед сохранением проекта в репозитории.

Работать с мастерами — быстро и просто. Конечно, вам придется написать большую часть исходного текста вашей программы, но C++Builder создаст основу приложения, избавив вас от скучного занятия по созданию стандартных элементов. Мастера являются инструментами RAD, как и сам C++Builder. Но можно сказать и больше: мастера C++Buider — это инструменты RAD внутри самого инструмента RAD!

Добавление функций и членов-данных в исходный код

Как вы уже знаете к этому моменту, C++Builder представляет собой мощный инструмент для быстрого создания интерфейса пользователя в при-

ложениях Windows. Он создает за вас обработчики событий — вам остается добавить код, необходимый для того, чтобы программа заработала. Однако достаточно скоро вы обнаружите, что вам нужно добавлять более сложный код в свои приложения. Я имею в виду добавление собственных функций и членов-данных в код, который сгенерировал C++Builder. Например, простое приложение изначально может содержать пару десятков обработчиков событий различных типов. C++Builder создает все эти обработчики событий за вас, вы просто заполняете пустоты своим рабочим кодом. Однако для того, чтобы сделать приложение жизнеспособным, вам, возможно, придется самостоятельно написать еще два десятка функций.

Добавление разработанных вами функций и данных в код, сгенерированный C++Builder — задача не очень сложная, но вы должны знать как, это делается, иначе у вас возникнут проблемы.

Как C++Builder управляет объявлениями классов

Как вы знаете, для каждой новой формы C++Builder создает три файла: файл формы, модуль исходного кода и заголовочный файл модуля. Созданное C++Builder объявление класса в заголовочном файле содержит два раздела. Первый раздел — это та часть объявления класса, которой управляет C++Builder. Второй раздел — это ваша часть. В 7-й день вы создали программу ScratchPad. Если вы делали упражнения в конце той главы, то должны были добавить в программу окно About и несколько кнопок. В листинге 9.1 приведен заголовок главной формы после добавления в приложение этих расширений.

Листинг 9.1. Файл SPMain.h

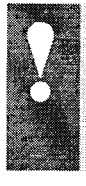
```
 1: class TScratchPad : public TForm
 2: {
 3: __published:  // компоненты, управляемые IDE
 4:     TPanel *Panel1;
 5:     TBevel *Bevel1;
 6:     TSpeedButton *FileOpenBtn;
 7:     TSpeedButton *FileSaveBtn;
 8:     TStatusBar *StatusBar;
 9:     TMainMenu *MainMenu;
10:     TMenuItem *FileMenu;
11:     TMenuItem *FileOpen;
12:     TMenuItem *FileSave;
13:     TMenuItem *FileSaveAs;
14:     TMenuItem *N1;
15:     TMenuItem *FilePrintSetup;
16:     TMenuItem *N2;
17:     TMenuItem *FileExit;
18:     TMenuItem *FilePrint;
19:     TMenuItem *Edit1;
20:     TMenuItem *EditReplace;
21:     TMenuItem *EditFind;
22:     TMenuItem *N4;
23:     TMenuItem *EditPaste;
24:     TMenuItem *EditCopy;
25:     TMenuItem *EditCut;
```

см. продолжение

Листинг 9.1. (продолжение)

```
27:     TMenuItem *EditUndo;
28:     TMenuItem *Help1;
29:     TMenuItem *HelpAbout;
30:     TMenuItem *HelpContents;
31:     TMenuItem *EditSelectAll;
32:     TMenuItem *N3;
33:     TMenuItem *EditWordWrap;
34:     TOpenDialog *OpenDialog;
35:     TSaveDialog *SaveDialog;
36:     TMenuItem *FileNew;
37:     TMemo *Memo;
38:     TPopupMenu *MemoPopup;
39:     TMenuItem *PopupCut;
40:     TMenuItem *PopupCopy;
41:     TMenuItem *PopupPaste;
42:     TSpeedButton *EditCutBtn;
43:     TSpeedButton *SpeedButton2;
44:     TSpeedButton *SpeedButton3;
45:     TSpeedButton *SpeedButton4;
46:     void __fastcall FileOpenClick(TObject *Sender);
47:     void __fastcall FileSaveClick(TObject *Sender);
48:     void __fastcall FileSaveAsClick(TObject *Sender);
49:     void __fastcall FileExitClick(TObject *Sender);
50:     void __fastcall EditSelectAllClick(TObject *Sender);
51:     void __fastcall EditCutClick(TObject *Sender);
52:     void __fastcall EditCopyClick(TObject *Sender);
53:     void __fastcall EditPasteClick(TObject *Sender);
54:     void __fastcall EditWordWrapClick(TObject *Sender);
55:     void __fastcall FileNewClick(TObject *Sender);
56:     void __fastcall EditUndoClick(TObject *Sender);
57:     void __fastcall HelpAboutClick(TObject *Sender);
58:     void __fastcall FormCreate(TObject *Sender);
59: private:       // Объявления пользователя
60: public:        // Объявления пользователя
61:   virtual __fastcall TScratchPad(TComponent* Owner);
62: };
```

Давайте посмотрим на строку под номером 3. Обратите внимание на ключевое слово published и комментарий Компоненты, управляемые IDE. В раздел между ключевым словом published и ключевым словом private (строка 59) не следует вмешиваться. Оставьте управление разделом published для C++Builder.



У вас могут возникнуть неприятности, если вы будете вносить изменения в текст между ключевыми словами published и private. В некоторых случаях вы просто получите сообщения об ошибках от компилятора или компоновщика. В других случаях вашу программу придется восстанавливать. Возьмите за правило остерегаться раздела published.



Если вы старательный студент, то сейчас, возможно, находитесь в некотором недоумении. В первых четырех главах, где мы рассмотрели основы языка Си++, вы узнали о существовании уровней доступа private, protected и public, но ни слова не услышали о ключевом слове published. Объясняется это очень просто: published — не ключевое слово Си++. Это ключевое слово относится к расширениям языка Си++, сделанным фирмой Borland, и отсутствует в стандарте ANSI. Оно было добавлено для реализации в Си++ тех мощных возможностей, которыми обладают компоненты.

Обратите внимание, что строки 59 и 60 в листинге 9.1 имеют комментарий Объявления пользователя. Вы можете без опасений размещать любые члены-данные или объявления функций-членов в разделах private и public.

Несколько слов о строках состояния и подсказках

Чуть позже мы добавим в программу ScratchPad поддержку подсказки, отображаемой в строке состояния. Однако предварительно вы должны познакомиться с тем, как реализованы подсказки.

Когда свойство ShowHint объекта Application имеет значение true (значение по умолчанию), и курсор мыши находится над компонентом, у которого свойство ShowHint также имеет значение true, генерируется событие hint. Объект Application имеет событие, называемое OnHint, которое происходит всякий раз, когда генерируется событие hint. Свойство Hint приложения будет содержать текст подсказки для элемента управления, который вызвал событие hint. Приложение может использовать событие OnHint для вывода подсказки в строке состояния.

Проблема состоит в том, что вы не имеет непосредственного доступа к событию OnHint объекта Application. Единственное, что вы можете сделать — это переопределить значение OnHint, чтобы оно указывало на одну из ваших функций. Когда произойдет событие hint, будет вызван ваш собственный обработчик OnHint. Итак, нужно написать обработчик события OnHint. Этим мы и займемся в следующем разделе.

Включение функций в программу



Чтобы попрактиковаться в добавлении функции к приложению, давайте включим подсказку в программу ScratchPad, которую вы написали ранее. Откройте снова проект ScratchPad. Если вы пропустили домашнее задание в седьмом дне, загрузите проект ScratchPad с компакт-диска, прилагаемого к этой книге.

Сначала мы должны подготовиться. Необходимо определить текст подсказки для каждой из кнопок быстрого доступа и подготовить строку состояния к получению подсказки. Для этого выполните следующие действия:

1. Щелкните на кнопке File Open в оперативной панели главной формы программы ScratchPad.
2. Найдите в инспекторе объектов свойство Hint и введите следующий текст подсказки:

```
New|Create a New File
```

3. Измените значение свойства ShowHint на true.
4. Повторите шаги 2 и 3 для всех кнопок оперативной панели, добавляя соответствующий текст подсказки для каждого типа кнопки.

5. Щелкните на компоненте StatusBar в нижней части главной формы. Измените значение свойства SimplePanel на true. После этой операции строка состояния сможет выводить текстовую строку, используя свойство SimpleText.

Теперь, когда мы выполнили необходимые подготовительные операции, можно приступать к решению нашей задачи. Нам нужно создать обработчик события OnHint, которому мы дадим (что неудивительно) имя OnHint(). Давайте начнем по порядку. Сначала мы добавим объявление функции в объявление класса.

1. Переключитесь в редактор кода и щелкните на вкладке SPMain.cpp.
2. Щелкните правой кнопкой на окне редактора кода и выберите в контекстном меню пункт Open Source/Header File (Открыть исходный/заголовочный файл). Рядом с вкладкой SPMain.cpp появится вкладка SPMain.h, которая станет активным окном редактора.
3. Прокрутите вниз объявление класса TScratchPad, пока не увидите раздел private. Добавьте следующую строку после ключевого слова private:

   void __fastcall OnHint(TObject* Sender);

   В результате последние несколько строк объявления класса должны выглядеть следующим образом:

```
    void __fastcall FormCreate(TObject *Sender);
private:        // User declarations
    void __fastcall OnHint(TObject* Sender);
public:         // User declarations
    virtual __fastcall TScratchPad(TComponent* Owner);
};
```

Пока все идет хорошо. Вы уже добавили объявление для вашей новой функции. Осталось сделать еще два шага и мы будем у цели. Сначала нужно добавить в исходный модуль саму функцию. После этого мы присвоим адрес нашей новой функции событию OnHint объекта Application.

1. Щелкните на вкладке SPMain.cpp и прокрутите этот файл до конца.
2. Введите следующий код:

```
void __fastcall TScratchPad::OnHint(TObject* Sender)
{
  StatusBar->SimpleText = Application->Hint;
}
```

3. Перейдите в инспектор объектов. Выберите главную форму, ScratchPad, в селекторе объектов.
4. Переключитесь на страницу Events в окне инспектора объектов и дважды щелкните в столбце Value рядом с событием OnCreate. Появится окно редактора кода и вы сможете ввести текст обработчика события.
5. Введите следующую строку, начиная с позиции курсора:

```
Application->OnHint = &OnHint;
```

   Функция FormCreate() должна теперь выглядеть следующим образом:

```
void __fastcall TScratchPad::FormCreate(TObject *Sender)
{
  Application->OnHint = &OnHint;
}
```

6. Откомпилируйте и запустите программу. Полный текст подсказки теперь будет отображаться в строке состояния, а короткая подсказка будет выводиться во всплывающем окне рядом с кнопкой.

В шаге 2 мы взяли текст подсказки из свойства Hint объекта Application и присвоили это значение свойству SimpleText класса StatusBar. В шаге 5 мы присвоили адрес созданной функции событию OnHint класса Application. Теперь при каждом возникновении события OnHint будет вызываться ваша функция OnHint(), и текст подсказки будет выводиться в строке состояния.

Добавление членов-данных класса

Добавление членов-данных в классы, сгенерированные C++Builder, выполняется аналогично добавлению функций. Члены-данные, как и функции-члены, должны добавляться только в разделы public или private.

Удаление кода, созданного C++Buider

Может случиться так, что вам нужно будет удалить код, созданный C++Builder. Например, вам может потребоваться удалить кнопку на форме, которая больше не нужна — ваша программа изменилась. Чтобы удалить кнопку, достаточно щелкнуть на ней и нажать на клавишу Delete. И кнопки как не бывало. C++Builder удалит кнопку, но обработчик события OnClick, связанный с этой кнопкой, останется в тексте программы. C++Builder, конечно, знает, что соответствующая кнопка удалена, но он не удаляет этот обработчик события, потому что его могут использовать другие компоненты. Только вы можете решить, удалять обработчик события или нет.

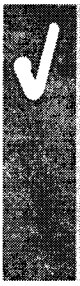
Сама процедура удаления обработчика события выглядит очень просто:

- Удалите определение функции из исходного модуля.
- Удалите объявление функции из заголовочного файла.



Помните, что из любого правила есть исключения: никогда не изменяйте раздел published объявления класса вашей формы.

Перед удалением обработчика события проверьте еще раз, что никакие другие компоненты его не используют. К сожалению, нет простого способа определить, использует ли какой-нибудь другой компонент данный обработчик события или нет. Вы просто должны помнить, как взаимодействуют компоненты вашего приложения.



Некоторые могут посоветовать поступать так: если вы не уверены в том, что обработчик события не используют другие компоненты, то оставьте его в тексте программы. На мой взгляд, это плохое решение. Вы должны знать, что происходит в вашей программе, и избавляться от любых ненужных функций. Хотя код, который не используется, не может причинить никому вреда, он будет увеличивать размер исполняемого файла. В некоторых случаях большое количество лишнего кода может вызвать снижение производительности программы. Будьте внимательны и очищайте вашу программу от неиспользуемого или неэффективного кода.

Использование файлов ресурсов

Каждая программа Windows использует ресурсы.



Ресурсы — это составные части программы, которые используются при ее выполнении, но не являются исполняемым кодом.

К типичным ресурсам Windows-приложений относятся:

- Клавиши быстрого доступа
- Растровые изображения
- Курсоры
- Диалоговые окна
- Значки
- Меню
- Таблицы данных
- Строковые таблицы
- Информация о версии
- Ресурсы, определяемые пользователем (например, звуковые и AVI-файлы)

Ресурсы обычно содержатся в *файле сценария ресурсов* (текстовый файл с расширением .rc), который компилируется компилятором ресурсов и затем связывается с .exe-файлом приложения на этапе компоновки.

Обычно думают, что ресурсы связаны с исполняемым файлом. На самом деле, некоторые ресурсы, типа растровых изображений, строковых таблиц и звуковых файлов, могут помещаться во внешние файлы (.bmp, .txt и .wav); другие могут быть связаны с .exe-файлом и содержаться внутри самого приложения. Вы можете выбирать любой из этих способов. Размещение ресурсов в .exe-файле имеет два основных преимущества:

- Программа работает эффективнее, поскольку найти ресурс в исполняемом файле намного быстрее, чем загрузить его с диска.
- Отпадает необходимость во вспомогательных файлах, так как программа и ресурсы содержатся в одном модуле (.exe-файле).

Неудобство этого подхода заключается в том, что ваш исполняемый файл будет иметь больший размер. Он будет не больше, чем суммарный размер внешних файлов ресурсов и самой программы, но увеличение объема может привести к увеличению времени загрузки.

Способ хранения ваших ресурсов будет, в частности, определяться стоящей перед вами задачей. Вы можете хранить ресурсы как во внешних файлах, так и включенными в .exe-файл (или даже использовать оба варианта в одной и той же программе).

Традиционная программа Windows почти всегда содержит по крайней мере одно диалоговое окно и один значок. Приложение C++Builder, однако, имеет свои особенности. Поскольку в приложениях C++Builder нет настоящих диалоговых окон, то нет и соответствующих ресурсов (формы C++Builder хранятся как ресурсы, но они являются ресурсами RCDATA, а не ресурсами диалоговых окон). Однако приложения C++Builder используют и традиционный ресурс — значки. C++Builder берет на себя заботу о создании файла ресурсов для значка, когда вы создаете приложение. Аналогично, если вы используете растровые изображения для кнопок быстрого доступа, компо-

ненты Image или BitBtn, C++Builder рассматривает растровый файл, который вы выбрали, как часть ресурсов формы. Форма и все ее ресурсы связываются с файлом программы при построении приложения. Все это выполняется более или менее автоматически.

В некоторых случаях, однако, необходимо уйти от обычного в C++Builder использования ресурсов. Например, если вы хотите сделать анимацию, то вам нужно иметь серию изображений, которые должны загружаться таким образом, чтобы обеспечить максимальное быстродействие. В такой ситуации вы должны знать, как связать внешние ресурсы с вашим исполняемым файлом.

Связывание файла ресурсов с исполняемым файлом — достаточно элементарная задача. Намного труднее создавать сами ресурсы. Нетрудно изготовить стандартные ресурсы типа растровых изображений, значков и курсоров с помощью хорошего редактора ресурсов, но создание трехмерных изображений и значков профессионального качества — это уже искусство. Сколько раз вы сталкивались с довольно приличными программами, имеющими ужасные рисунки на кнопках? Я видел множество. Вы можете создавать растровые изображения, значки и курсоры, используя редактор изображений C++Builder. Для работы со строковыми ресурсами, данными пользователя, звуковыми файлами или другими специальными ресурсами вам, вероятно, потребуется редактор ресурсов третьих фирм.



Если у вас есть Borland C++, вы можете использовать его инструмент Resource Workshop для редактирования специализированных ресурсов. После создания ресурсов, полученный .rc-файл можно либо добавить в ваш проект, либо откомпилировать его в .res-файл, воспользовавшись для этой цели компилятором ресурсов фирмы Borland (BRCC32.EXE). Этот компилятор ресурсов поставляется как с Borland C++, так и с C++Builder. В принципе, вы можете создать .rc-файл в любом текстовом редакторе, после чего обработать его компилятором ресурсов; но на практике намного проще использовать редактор ресурсов.

Вы можете добавлять в ваш проект либо .res, либо .rc-файл, используя менеджер проектов. Чтобы добавить файл ресурсов в проект, вам нужно сначала вызвать менеджер проектов, выбрав в главном меню пункт View | Project Manager. Когда появится диалоговое окно Project Manager, нажмите кнопку Add To Project (Добавить в проект). В появившемся диалоговом окне File Open выберите файл ресурсов, который вы хотите добавить в проект, и нажмите кнопку OK. Файл ресурсов появится в окне менеджера проектов вместе с остальными файлами приложения. Я буду подробно говорить о менеджере проектов завтра.



В листингах 9.2 и 9.3 приведены заголовок и модуль главной формы для программы с названием Jumping Jack. Эта программа — простая мультяшка со звуковыми эффектами. На основной форме находятся только две кнопки, компонент Image и компонент Label. Программа Jumping Jack иллюстрирует некоторые аспекты использования ресурсов в приложении C++Builder. (Эту программу вы найдете на компакт-диске, который прилагается к книге.) На примере этой программы вы узнаете, как загрузить растровое изображение, существующее в виде ресурса, как загрузить и вывести строковый ресурс и как проиграть звуковой ресурс. Листинг 9.4 — это часть файла ресурсов, который используется программой Jumping Jack.

Просмотрите внимательно все листинги, а затем мы поговорим о том, как эта программа работает.

Листинг 9.2. Файл JJMain.h

```
 1: //---------------------------------------------------------------------
 2: #ifndef JJMainH
 3: #define JJMainH
 4: //---------------------------------------------------------------------
 5: #include <vcl\Classes.hpp>
 6: #include <vcl\Controls.hpp>
 7: #include <vcl\StdCtrls.hpp>
 8: #include <vcl\Forms.hpp>
 9: #include <vcl\ExtCtrls.hpp>
10: //---------------------------------------------------------------------
11: class TMainForm : public TForm
12: {
13: __published:    // Компоненты, управляемые IDE
14:     TButton *Start;
15:     TButton *Stop;
16:     TImage *Image;
17:     TLabel *Label;
18:     void __fastcall FormCreate(TObject *Sender);
19:
20:     void __fastcall StartClick(TObject *Sender);
21:     void __fastcall StopClick(TObject *Sender);
22: private:      // Объявления пользователя
23:     bool done;
24:     void DrawImage(String& name);
25: public:       // Объявления пользователя
26:     virtual __fastcall TMainForm(TComponent* Owner);
27: };
28: //---------------------------------------------------------------------
29: extern TMainForm *MainForm;
30: //---------------------------------------------------------------------
31: #endif
```

Листинг 9.3. Файл JJMain.cpp

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: //
 4: // Этот #include необходим для функции PlaySound()
 5: //
 6: #include <vcl\mmsystem.hpp>
 7: #pragma hdrstop
 8:
 9: #include "JJMain.h"
10: //---------------------------------------------------------------------
11: #pragma resource "*.dfm"
12: //
13: // Определения для строковых ресурсов
14: //
15: #define IDS_UP   101
16: #define IDS_DOWN 102
17:
18: TMainForm *MainForm;
```

```
19: //---------------------------------------------------------------------
20: __fastcall TMainForm::TMainForm(TComponent* Owner)
21:   : TForm(Owner),
22:   done(false)
23: {
24: }
25: //---------------------------------------------------------------------
26: void __fastcall TMainForm::FormCreate(TObject *Sender)
27: {
28:   //
29:   // Загрузка и вывод первого изображения
30:   //
31:   Image->Picture->Bitmap->
32:     LoadFromResourceName((int)HInstance, "ID_BITMAP1");
33: }
34: //---------------------------------------------------------------------
35: void __fastcall TMainForm::StartClick(TObject *Sender)
36: {
37:   //
38:   // После щелчка на кнопке Start начинается цикл анимации.
39:   // Растровые ресурсы имеют имена с ID_BITMAP1 по
40:   // ID_BITMAP5, поэтому мы начинаем со строки "ID_BITMAP"
41:   // и добавляем по мере необходимости последнюю цифру
42:   //
43:   //
44:   String s = "ID_BITMAP";
45:   //
46:   // Буфер для строковых ресурсов
47:   //
48:   char buff[10];
49:   //
50:   // Флажок, который будет сигнализировать об окончании
51:   //
52:   done = false;
53:   //
54:   // Начало цикла. Цикл выполняется, пока не будет нажата
55:   // кнопка 'Stop'
56:   //
57:   while (!done) {
58:     //
59:     // Цикл по пяти растровым изображениям, начиная с 1
60:     // и заканчивая 5
61:     //
62:     for (int i = 1; i < 6; i++) {
63:       //
64:       // Добавление значения 'i' в конец строки
65:       // для создания строки с именем ресурса
66:       //
67:       String resName = s + String(i);
68:       //
69:       // Вызов функции-члена класса для отображения картинки
70:       //
71:       DrawImage(resName);
72:     }
73:     //
74:     // Загрузка строкового ресурса "Up" с помощью функции WinAPI
75:     // LoadString(), отображение строки и указание Windows
```

см. продолжение

Листинг 9.3. (продолжение)

```
76:     // перерисовать Label
77:     //
78:     LoadString(HInstance, IDS_UP, buff, sizeof(buff));
79:     Label->Caption = buff;
80:     Label->Refresh();
81:     //
82:     // Асинхронное проигрывание звука 'up' с помощью
83:     // функции WinAPI PlaySound()
84:     //
85:     PlaySound("ID_WAVEUP",
86:       HInstance, SND_ASYNC | SND_RESOURCE);
87:     //
88:     // Небольшая пауза в верхней точке прыжка
89:     //
90:     Sleep(200);
91:     //
92:     // Повтор в обратном порядке
93:     //
94:     for (int i = 5; i > 0 ; i--) {
95:       String resName = s + String(i);
96:       DrawImage(resName);
97:     }
98:     PlaySound("ID_WAVEDOWN",
99:       HInstance, SND_ASYNC | SND_RESOURCE);
100:    LoadString(HInstance, IDS_DOWN, buff, sizeof(buff));
101:    Label->Caption = buff;
102:    Label->Refresh();
103:    Sleep(200);
104:  }
105: }
106: //---------------------------------------------------------------------
107: void __fastcall TMainForm::StopClick(TObject *Sender)
108: {
109:   //
110:   // Нажата кнопка Stop, останов цикла
111:   //
112:   done = true;
113: }
114: //---------------------------------------------------------------------
115: //
116: // Функция-член класса, отображающая картинку
117: //
118: void
119: TMainForm::DrawImage(String& name)
120: {
121:   //
122:   // Загрузка растрового изображения из ресурса
123:   // с использованием переданного имени
124:   //
125:   Image->Picture->Bitmap->
126:     LoadFromResourceName((int)HInstance, name);
127:   //
128:   // "Прокачка" очереди сообщений, чтобы Windows могла
129:   // отобразить картинку
130:   //
131:   Application->ProcessMessages();
132:   //
```

```
133:    // Короткая пауза, чтобы анимация не шла слишком быстро
134:    //
135:    Sleep(20);
136: }
```

Листинг 9.4. Начало файла JJRes.rc

```
 1: #define IDS_UP   101
 2: #define IDS_DOWN 102
 3:
 4: STRINGTABLE
 5: {
 6:  IDS_UP, "Up"
 7:  IDS_DOWN, "Down"
 8: }
 9:
10: ID_WAVEUP   WAVE "up.wav"
11: ID_WAVEDOWN WAVE "down.wav"
12:
13: ID_BITMAP1 BITMAP LOADONCALL MOVEABLE DISCARDABLE IMPURE
14: {
15:  '42 4D 76 02 00 00 00 00 00 00 76 00 00 00 28 00'
16:  '00 00 20 00 00 00 20 00 00 00 01 00 04 00 00 00'
17:  //
18:  // остальная часть растрового ресурса опущена
```

Ключ Обратите внимание на строки 23 и 24 объявления класса главной формы в заголовочном файле из листинга 9.2. В строке 23 объявлен член типа bool, который используется для остановки анимации. Функция-член класса, объявленная в строке 24, используется для вывода растрового изображения в компоненте Image.

В листинге 9.3 следует обратить внимание на две функции API Windows, используемые для загрузки строк и звукового файла из соответствующих ресурсов. В строке 78 функция LoadString() загружает строковый ресурс в буфер, используя числовой идентификатор строки (как создаются строковые ресурсы, видно из листинга 9.4). Затем эта строка устанавливается в качестве значения свойства Caption компонента Label. В строке 83 используется функция PlaySound(), которая проигрывает звуковой файл, содержащийся в ресурсе. Флаг SND_ASYNC, используемый в качестве аргумента функции PlaySound(), указывает Windows немедленно возвратить управление программе после начала воспроизведения. Это позволяет продолжать анимацию во время проигрывания звука. Флаг SND_RESOURCE сообщает Windows, что звук хранится в виде ресурса, а не как файл на диске. Обе функции, LoadString() и PlaySound(), используют глобальную переменную HInstance, указывающую Windows, что ресурс нужно искать в исполняемом файле.

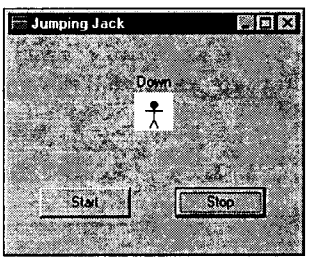
В строках с 1 по 8 листинга 9.4 вы можете видеть, как задается таблица строк в файле сценария ресурсов. Таблицу строк можно создать с помощью любого текстового редактора. В строках 10 и 11 создается звуковой ресурс WAVE на основе двух записанных ранее звуковых файлов, расположенных в каталоге проекта. Когда компилятор ресурсов встречает объявление WAVE, он считывает отдельные звуковые файлы и компилирует их в двоичный файл ресурсов.



Как видно из листинга 9.4, вы можете легко создавать некоторые ресурсы с помощью текстового редактора. Если растровые изображения и звук хранятся во внешних файлах, их можно включить в .rc-файл, как это сделано в листинге 9.4, а затем, используя компилятор ресурсов, откомпилировать в двоичный файл ресурсов. После этого двоичный файл ресурсов можно будет связать с исполняемым файлом вашего приложения.

В листинге 9.4 приведена только часть файла ресурсов. Растровые изображения содержатся в этом файле в числовом виде и занимают довольно много места. Оставшаяся часть описания растровых ресурсов Jumping Jack занимает приблизительно 200 строк, поэтому я решил не приводить их в листинге целиком. На рис. 9.12 показан Jumping Jack в промежуточном состоянии.

Рис. 9.12
Jumping Jack в движении



Хотя создание дополнительных ресурсов для ваших программ — это не техника ракетостроения, здесь есть свои тонкости. Вам потребуется некоторое время, чтобы понять, как все это работает. Возможно, что вам никогда не потребуется создавать дополнительные ресурсы для своих приложений. Но если вам придется это делать, вы будете знать, с чего начинать. Если после чтения этого раздела вы не до конца поняли, о чем шла речь, не расстраивайтесь. Через какое-то время все встанет на свои места.



Растровые изображения, значки и курсоры во многих программах обычно защищены авторским правом. Не используйте без разрешения ресурсы из таких программ. Более того, рассматривайте любую программу как защищенную авторским правом, если она не объявлена как свободно распространяемая (freeware). Что касается растровых изображений, значков и курсоров, поставляемых вместе с C++Builder (см. каталог \CBuilder\Images), то вы можете свободно использовать их в своих приложениях — разрешения от Borland не требуется.

Заключение

Репозиторий объектов — замечательный инструмент, облегчающий повторное использование созданных ранее проектов, форм, диалоговых окон и других объектов. Возможность добавлять в репозиторий свои собственные объекты является огромным преимуществом, когда вы используете технику RAD. Мастера диалогов и приложений еще больше облегчают процесс разработки. Особенно полезен мастер приложений. В середине главы вы узнали, как добавлять функции и члены-данные в классы, сгенерированные

C++Builder. Заключительная часть главы была посвящена различным типам ресурсов, которые вы можете включать в свои приложения, а также их добавлению в проекты C++Builder.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Когда я должен использовать опцию Use репозитория объектов?

Когда вы хотите внести изменения непосредственно в объект, размещенный в репозитории.

Есть ли ограничение на число объектов, размещаемых в репозитории?

Технических ограничений нет. Вы можете поместить туда столько объектов, сколько находите нужным. Помните, однако, что назначение репозитория объектов состоит в том, чтобы помочь вам быстро находить и использовать ваши формы, диалоговые окна и другие объекты. Если вы запишете в репозиторий слишком много редко используемых объектов, то эффективность вашей работы снизится, так как на поиск нужного объекта будет расходоваться значительное время. Больше времени будет также уходить на загрузку и отображение всех объектов репозитория.

У меня в репозитории накопилось много старых объектов, которые я больше не использую. Как мне избавиться от них?

Выберите пункт меню Options | Repository. На экране появится диалоговое окно конфигурации репозитория объектов. Чтобы удалить объект, сначала выберите его в списке объектов, а затем нажмите кнопку Delete Object. Объект будет удален из репозитория.

У меня в репозитории был некоторый объект. Но в последнее время, когда я пытаюсь его использовать, я получаю следующее сообщение: Unable to find both a form and a source file (Невозможно найти форму и исходный файл). Что случилось?

Вы куда-то переместили или удалили исходный файл и/или файл формы этого объекта. В репозитории сохраняется каталог, в котором находится объект. Если вы перемещаете или удаляете объект, то репозиторий объектов не может его найти и выдает сообщение об ошибке.

Могу я добавить объекты на страницу New репозитория объектов?

Нет, не можете. Содержимое страницы New репозитория объектов фиксировано. Объекты этой страницы нельзя удалить или изменить. Вам нужно разместить ваши объекты на другой странице.

Я добавил функцию в класс моей главной формы и теперь не могу откомпилировать программу. В чем дело?

Вероятно, вы случайно добавили объявление функции к разделу published объявления класса. Убедитесь в том, что объявление вашей функции находится в разделах public или private.

У меня есть редактор ресурсов, который позволяет декомпилировать ресурсы, содержащиеся в других программах. Таким образом я «занимаю» нужные мне изображения и другие ресурсы из различных программ. Я могу так поступать?

Вообще говоря, нет. Вы должны исходить из предположения, что все ресурсы других программ защищены законом об авторских правах и не могут свободно использоваться. Проконсультируйтесь с юристом по этому вопросу.

Моя прикладная программа использует большое количество растровых изображений и звуковых файлов. Могу ли я поместить все эти ресурсы в другой файл, а не в исполняемый файл моей программы?

Да. Вы можете сохранить ресурсы в динамически компонуемой библиотеке (DLL).

Вопросы

1. Когда используется опция Inherit при выборе объекта из репозитория объектов?
2. Как сохранить проект в репозитории объектов?
3. Что случится с производными формами, если вы измените базовую форму?
4. В каком месте объявления класса формы размещаются объявления функций пользователя?
5. Где помещается определение функции (тело самой функции) при ее добавлении в программу, сгенерированную C++Builder?
6. Как можно узнать, кто является автором каждого объекта из репозитория?
7. Где можно добавить или удалить страницы репозитория объектов?
8. Что проще — создавать стандартное приложение с самого начала или использовать мастер приложений?
9. Какими двумя способами можно добавить файл ресурсов в проект?
10. Можно ли создать файл сценария ресурсов, содержащий строковую таблицу, используя для этой цели текстовый редактор?

Упражнения

1. Создайте новую форму. Добавьте в нее несколько компонентов по своему выбору. Сохраните форму под именем BaseForm на странице Forms репозитория объектов.

2. Создайте новое приложение. Выберите в меню пункт File | New, чтобы вызвать репозиторий объектов. Перейдите на страницу Forms. Щелкните на радиокнопке Inherit. Выберите объект BaseForm, который вы создали в упражнении 1, и добавьте его в приложение. (Убедитесь в том, что вы установили опцию Inherit.) Сохраните проект и закройте его.
3. Откройте объект BaseForm, который вы создали в упражнении 1. Удалите все компоненты из формы и сохраните ее.
4. Вновь откройте проект, который вы создали в упражнении 2. Откройте форму этого приложения. Обратите внимание, что все компоненты исчезли. (Вспомните, что вы создали этот объект с опцией Inherit, и поэтому все изменения, сделанные в базовой форме, отразились и в производной форме.)
5. Выберите в главном меню пункт Options | Repository. Удалите ранее созданную форму BaseForm.
6. Создайте проект с помощью мастера приложений. Используйте все типы меню и укажите, что приложение будет приложением типа MDI. Создайте оперативную панель.
7. Добавьте многостраничное диалоговое окно в приложение, которое вы создали в упражнении 6. Используйте для этого мастер диалогов.
8. Добавьте в программу из упражнения 6 окно About, используя репозиторий объектов.
9. Создайте в текстовом редакторе ресурс строковой таблицы и откомпилируйте его компилятором ресурсов (BRCC32.EXE). **Дополнительное задание:** напишите программу для загрузки строк и отображения их на форме.



10. Откройте в текстовом редакторе файл JJRes.rc, находящийся на прилагаемом к книге компакт-диске, и исследуйте его содержимое.
11. Напишите 100 раз на черном фоне фразу: «Я не буду использовать растровые изображения из чужих программ».

Подробнее о проектах



Кент Рейсдорф



6-й день, «Изучение IDE C++Builder: проекты и формы», вы получили общее представление о проектах C++Builder. Сегодня мы рассмотрим работу с проектами более подробно.

Сегодня вы также подробнее изучите редактор кода C++Builder и его возможности.

Проекты нужны всем

Проекты — это неотъемлемая часть C++Builder. Вы не сможете создать программу без проекта. Проект содержит информацию, необходимую для построения работающего приложения. В этом разделе мы рассмотрим следующие компоненты интегрированной среды C++Builder:

- Менеджер проектов (Project Manager)
- Проводник проекта (Project Explorer)
- Диалоговое окно Project Options (Опции проекта)
- Редактор кода

Итак, приступим к работе.

Менеджер проектов

На определенном этапе каждый проект нуждается в некоторой организации. Может случиться так, что вам потребуется добавить в проект новый модуль или, возможно, удалить один из модулей. У вас может возникнуть необходимость добавить к проекту файлы других типов, например, двоичный файл ресурсов или файл библиотеки импорта для DLL. Все эти действия выполняются через менеджер проектов.

Окно менеджера проектов

Окно Project Manager показывает текущие файлы, входящие в проект. Чтобы вывести окно менеджера проектов, выберите в главном меню пункт

View | Project Manager. На рис. 10.1 показано окно менеджера проектов для программы ScratchPad, созданной вами в 6-й день.

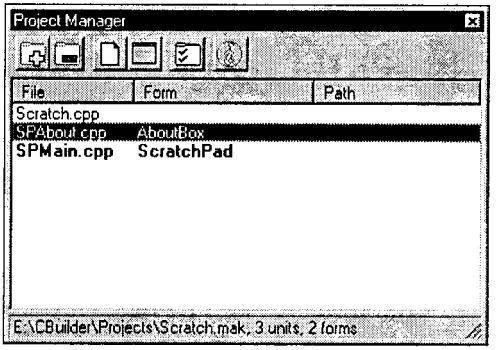
Окно менеджера проектов показывает состояние каждого файла проекта. Файлы, которые не изменялись с момента последнего сохранения проекта, показаны обычным шрифтом; измененные файлы, которые еще не были сохранены, выделены полужирным шрифтом. Это позволяет вам сразу видеть, какие файлы были изменены после последнего сохранения проекта.

Рассматривая рис. 10.1, вы обратите внимание, что список файлов в окне менеджера проектов содержит три столбца. Первый столбец показывает имя исходного модуля, который представляет этот файл. Например, исходный модуль для главной формы программы ScratchPad называется SPMain.cpp.

Второй столбец содержит имя формы, связанной с данным файлом. Имя формы берется из свойства Name формы. В некоторых случаях исходный модуль не имеет связанной с ним формы. В каждый проект входит исходный файл проекта, который содержит код VCL, выполняющий инициализацию приложения. Поскольку с исходным файлом проекта не связано никакой формы, столбец Form для этого файла всегда будет пустым. На рис. 10.1 видно, что исходный файл проекта для программы ScratchPad называется Scratch.cpp, а его столбец Form не заполнен.

Последний столбец списка файлов в окне менеджера проектов содержит путь к данному файлу. Для файлов, которые находятся в собственном каталоге проекта, этот столбец остается пустым.

Рис. 10.1
Окно менеджера проектов





Размеры столбцов в окне менеджера проектов можно изменить. Чтобы изменить ширину столбца, установите курсор мыши на разделитель двух столбцов в заголовке. Когда вы увидите, что курсор мыши изменился на курсор перетаскивания, перетащите разделитель вправо или влево, чтобы расширить или сузить столбец.

Строка состояния окна менеджера проектов показывает рабочий каталог проекта и общее число модулей и форм проекта.

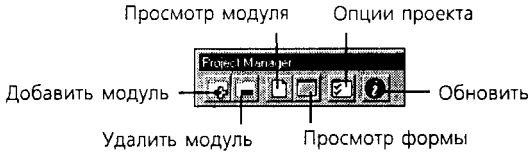


Текст строки состояния довольно неразборчиво считает модули, содержащиеся в проекте. Фактически, в строке состояния учитывается любой файл, независимо от того, является ли он модулем исходного кода или нет. Например, включение в проект текстового файла увеличит счетчик модулей в строке состояния даже в том случае, если этот файл не содержит никакого кода.

Оперативная панель менеджера проектов

Для управления проектом можно использовать оперативную панель менеджера проектов. Кнопки оперативной панели показаны на рис. 10.2.

Рис. 10.2
Оперативная панель менеджера проектов



Add Unit

Кнопка Add Unit используется для включения файлов в проект. Когда вы щелкаете на этой кнопке, появляется диалоговое окно Add To Project (Добавить в проект). Это диалоговое окно содержит фильтры для следующих типов файлов:

- Исходные файлы Си++ (.cpp)
- Исходные файлы Си (.c)
- Исходные файлы Паскаля (.pas)
- Библиотечные файлы (.lib)
- Объектные файлы (.obj)
- Файлы сценария ресурсов (.rc)
- Двоичные файлы ресурсов (.res)

Файлы каждого из этих типов C++Builder обрабатывает определенным образом. Например, если вы добавляете исходный файл Си (.c), то C++Builder будет компилировать его как файл Си, а не как файл Си++ (различия носят достаточно тонкий характер, и большинство пользователей не обращает на них внимания; но для некоторых программистов эти различия существенны). Если вы добавляете исходный файл Паскаля (.pas), то компиляция будет выполнена компилятором Паскаля. Объектный файл (.obj) будет передан компоновщику на стадии компоновки программы.



Вы не можете добавить модуль в проект, если в нем уже существует форма с тем же самым именем. Например, если у вас есть форма с именем MainForm, и вы попытаетесь добавить модуль из другого проекта, который также содержит форму с именем MainForm, то получите сообщение об ошибке даже в том случае, когда имена файлов различаются.



Не рекомендуется включать в проект файлы других типов, которые отличаются от перечисленных в этом разделе. C++Builder будет пытаться выполнить компиляцию любого файла неизвестного типа, что приведет к сообщению об ошибке.



Вы можете добавить в проект более одного исходного файла за один раз. Для этого выделите в диалоговом окне Add To Project все необходимые файлы и нажмите OK. Все выделенные файлы будут добавлены в проект.

Remove Unit

Команда Remove Unit используется для удаления файлов из проекта. Удаленные файлы исключаются из процесса компиляции/компоновки, но остаются на жестком диске.



Соблюдайте осторожность при удалении модулей из своих проектов. Вы должны следить за тем, чтобы не удалить те модули, к которым обращаются другие модули проекта. Если вы это сделаете, то получите сообщение об ошибке во время компиляции или компоновки. Перед удалением модуля обязательно убедитесь в том, что он не используется где-либо в проекте.

Окно менеджера проектов не поддерживает групповое выделение. Если вам нужно удалить несколько модулей, то придется удалять их по одному.

View Unit

Когда вы щелкаете на кнопке View Unit, текст выбранного в данный момент модуля отображается в редакторе кода. Если ни один модуль не выделен или исходный текст для выделенного модуля не существует, то доступ к данной кнопке запрещается. Например, двоичный файл ресурсов (.res) не имеет исходного файла. Если вы щелкнете на двоичном файле ресурсов в окне менеджера проектов, то кнопка View Unit будет недоступна.

View Form

Кнопка View Form используется для отображения формы, связанной с выбранным в данный момент модулем. Форма отображается в редакторе форм. Аналогично кнопке View Unit, кнопка View Form будет недоступна, если для выбранного модуля нет соответствующей формы.



Менеджер проектов имеет сокращенные команды для просмотра исходного текста модуля или формы. Чтобы просмотреть исходный текст модуля, дважды щелкните на имени файла модуля (в столбце File). Чтобы просмотреть форму, дважды щелкните на имени формы в столбце Form. Двойной щелчок на модуле, который не имеет исходного текста, не даст никакого результата.

Project Options

Кнопка Project Options оперативной панели выводит диалоговое окно Project Options для данного проекта. Опции проекта обсуждаются позже в этой главе, в разделе «Опции проекта».

Update

Кнопка Update служит для обновления проекта после изменения одного или нескольких исходных файлов. Обычно эта кнопка недоступна. Если вы вручную изменяете исходный файл проекта, то доступ к кнопке Update разрешается, а все файлы в списке файлов менеджера проектов становятся недоступными (их названия выделяются серым цветом). Щелчок на кнопке Update гарантирует, что все файлы проекта будут согласованы. Доступ к этой кнопке разрешается также после того, как вы меняете опции проекта.



Сохранение проекта автоматически приводит к его согласованию, после чего уже нет необходимости использовать кнопку Update.

Контекстное меню менеджера проектов

Менеджер проектов имеет контекстное меню, которое упрощает управление проектом. Многие из пунктов контекстного меню доступны также через оперативную панель. На рис. 10.3 показано контекстное меню менеджера проектов, а в таблице 10.1 приведен список пунктов меню и их описание.

Рис. 10.3
Контекстное меню менеджера проектов

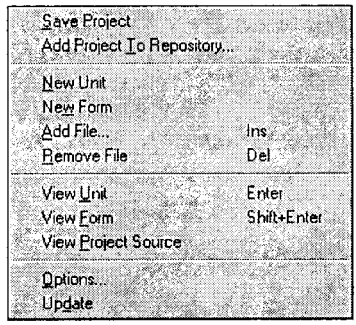


Таблица 10.1. Пункты контекстного меню менеджера проектов

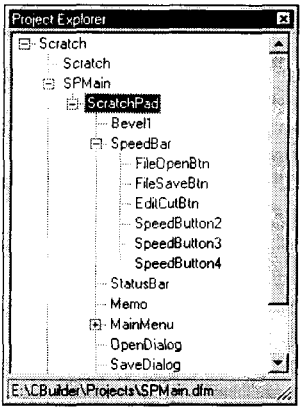
Пункт	Описание
Save Project	Сохранение проекта и всех исходных файлов проекта.
Add Project To Repository	Сохранение проекта в репозитории объектов.
New Unit	Создание пустого модуля исходного кода и его отображение в редакторе кода.
New Form	Создание пустой формы и ее отображение в редакторе форм.
Add File	То же, что и кнопка Add Unit.
Remove File	То же, что и кнопка Remove Unit.
View Unit	Отображение исходного текста выбранного в данный момент модуля в редакторе кода.
View Form	Отображение формы выбранного в данный момент модуля в редакторе форм.
View Project Source	Отображение исходного файла проекта в редакторе кода.
Options	Вывод диалогового окна Project Options.
Update	То же, что и кнопка Update.

Несколько пунктов контекстного меню доступны также в главном меню, в оперативной панели менеджера проектов и имеют эквивалентные сочетания клавиш.

Исследование проектов

Проводник проекта (Project Explorer)* — это компактный инструмент, который позволяет просматривать структуру проекта. На рис. 10.4 показан проводник проекта при исследовании программы ScratchPad.

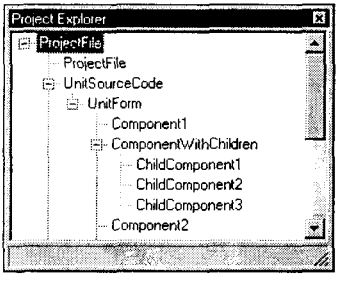
Рис. 10.4
Проводник проекта



Проводник проекта показывает все исходные модули вашего проекта, формы и компоненты, объединяя их в стандартную древовидную структуру. Узлы дерева можно свернуть или раскрыть, чтобы показать структуру ветвей более подробно. Каждый компонент располагается ниже своего предка. Например, посмотрите на узел SpeedBar на рис. 10.4. В этом проекте узел SpeedBar является компонентом Panel, который содержит несколько кнопок быстрого доступа. Эти кнопки показаны ниже панели, чтобы отразить тот факт, что по отношению к ним панель является компонентом-предком.

Проводник проекта представляет собой иерархическое окно. В верхней части этого окна находится файл проекта; под файлом проекта расположены исходные модули проекта. Сразу за исходным файлом проекта следует файл формы этого модуля. Под файлом формы перечислены все компоненты, которые являются прямыми потомками формы. Если, в свою очередь, компонент формы имеет компоненты-потомки, они перечислены под своим предком, и т. д. На рис. 10.5 показано, как будет выглядеть проводник проекта, если заменить имена компонентов и модулей на пояснительный текст.

Рис. 10.5
Иерархия проводника проекта



---

* Проводник проекта присутствовал в предварительной версии C++Builder с кодовым названием Ebony, по которой частично писалась эта книга, и был исключен из основной версии. – *Прим. ред.*

Когда вы щелкаете на каком-либо элементе в проводнике проекта, строка состояния в нижней части окна показывает информацию о данном элементе. Если выбранный элемент является исходным файлом или файлом формы, то строка состояния отображает полный путь и имя данного файла. Если элемент является компонентом, то строка состояния отображает имя класса данного компонента. Например, щелкнув на компоненте кнопки быстрого доступа, вы увидите в строке состояния имя TSpeedButton.

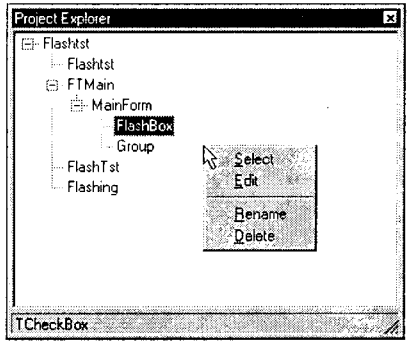
Команды проводника проекта

Работая с проводником проекта, вы можете не только просматривать структуру объектов. Проводник проекта имеет собственное контекстное меню, которое позволяет удалять, переименовывать, редактировать или выделять элемент дерева. Однако не все пункты контекстного меню постоянно доступны. Например, если вы выбрали исходный файл модуля, то пункт контекстного меню Rename (Переименовать) будет недоступен, поскольку вы не можете переименовать исходный файл, находясь в проводнике проекта. Давайте рассмотрим отдельные команды проводника проекта.

Select

Пункт меню Select (Выделить) выделяет текущий объект и отображает его в инспекторе объектов. Действие этой команды аналогично выбору объекта в редакторе форм. Расположив окна инспектора объектов и проводника проекта рядом, вы сможете быстро пройти все объекты проекта, одновременно просматривая их свойства. Эта команда доступна только в том случае, если вы выбрали форму или элемент управления.

Рис. 10.6
Контекстное меню проводника проекта





Обычный выбор элемента управления в дереве проводника проекта не равнозначен его выделению при помощи команды Select. Если вы хотите, чтобы компонент был выведен в окно инспектора объектов, вы должны сначала выбрать компонент в дереве проводника, а затем выбрать в контекстном меню пункт Select.

Edit

Пункт контекстного меню Edit (Правка) позволяет редактировать выбранный объект. Если объектом является файл исходного текста, то команда Edit выведет окно редактора кода с загруженным файлом. Если выбранный объект является формой, на передний план переместится редактор форм, который отобразит данную форму. Если выбранный объект является компонентом

формы, то будет также выведен редактор форм с выделенным компонентом, как если бы вы щелкнули на нем мышью. В случае форм и компонентов состояние инспектора объектов изменяется, чтобы показать выбранный объект.

Rename

Пункт меню Rename (Переименовать) позволяет изменить свойство Name формы или элемента управления из проводника проекта. Например, чтобы изменить имя компонента, сначала выделите компонент в окне проводника, а затем выберите пункт Rename в контекстном меню. После этого вы можете ввести новое имя компонента. Как и при изменении свойства Name через инспектор объектов, все изменения сразу отражаются в исходном тексте вашей программы.



Окно проводника проекта – это стандартный элемент управления Windows (дерево просмотра), и оно имеет те же самые характеристики, что и большинство аналогичных элементов управления. В частности, вы можете изменить текст определенного элемента в проводнике проекта, щелкнув на этом элементе один раз для его выделения, а затем еще раз для начала редактирования. Это называется *оперативным редактированием (in-place editing) или редактированием по месту*. Действуя таким образом, вы вообще можете не использовать пункт Rename контекстного меню. Если вы попытаетесь изменить имя элемента, которое не может быть изменено из проводника проекта (например, имя исходного файла), то результаты оперативного редактирования будут игнорироваться.

Delete

Пункт Delete (Удалить) контекстного меню проводника проекта работает согласно его названию. Если вы выделите объект в проводнике проекта и затем выберете пункт Delete, то этот объект будет удален из проекта.



Будьте внимательны при удалении объектов из окна проводника проекта, поскольку он не имеет возможности отмены команды Delete (Undo или Undelete). Если вы случайно удалите объект в проводнике проекта, вам нужно будет закрыть проект без сохранения, после чего открыть его еще раз.

Работа с проектами через проводник проекта — это одна из тех возможностей, которые легко упустить из виду. Потратьте некоторое время на изучение этого инструмента, и я уверен, что он пригодится вам при разработке приложений.

Опции проекта

Опции проекта — это то, на что часто не обращают внимания. С одной стороны, когда вы только начинаете изучать новую среду программирования, вам вполне хватает значений, установленных по умолчанию. В конце концов, у кого есть время, чтобы беспокоиться обо всех многочисленных опциях компилятора/компоновщика, когда все усилия направлены на освоение самой

среды? Однако на некотором этапе вы начнете в большей степени интересоваться тем, что делают эти опции. Поэтому, когда придет время, неплохо иметь соответствующие ссылки.

В этом разделе мы рассмотрим диалоговое окно Project Options. Вы можете вызвать это окно, выбрав в главном меню пункт Options | Project, нажав Alt+F6 на клавиатуре или выбрав пункт Options в контекстном меню менеджера проектов. Диалоговое окно Project Options содержит следующие вкладки:

- Forms (Формы)
- Application (Приложение)
- C++
- Pascal
- Linker (Компоновщик)
- Directories/Conditionals (Каталоги/Условные определения)

Мы рассмотрим каждую вкладку диалогового окна так, чтобы вы точно понимали, что она делает. Я начну с самых простых вкладок — Forms и Application. После этого мы перейдем к более сложному материалу.



В нижней части каждой вкладки диалогового окна Project Options находится флажок с названием Default (по умолчанию). Если вы хотите, чтобы текущие установки стали установками по умолчанию для всех вновь создаваемых проектов, установите этот флажок. Когда вы нажмете OK, текущие установки станут новыми установками по умолчанию.

Вкладка Forms

Во вкладке Forms диалогового окна Project Options вы определяете, как ваше приложение будет управлять своими формами. Вы видели это диалоговое окно в 5-й день, «Библиотеки классов Си++ и модель визуальных компонентов», когда создавали программу Picture Viewer. На рис. 10.7 показана вкладка Forms диалогового окна Project Options для программы ScratchPad (а для чего же еще?).

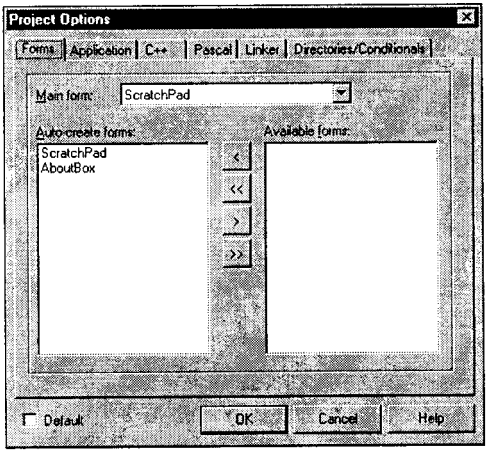


Рис. 10.7
Вкладка Forms диалогового окна Project Options

В верхней части вкладки Forms находится комбинированный список Main form (Главная форма). Здесь вы указываете C++Builder, какая форма должна отображаться сразу после запуска приложения. По умолчанию, главной формой приложения будет являться первая форма, которую вы создаете. Если вы изменяете свой проект таким образом, что главной становится другая форма, то вы должны изменить эту установку соответствующим образом.

В средней части диалогового окна находятся два окна списка. Левое окно списка имеет заголовок Auto-create forms (Автоматически создаваемые формы), а правое — Available forms (Доступные формы). Прежде чем рассматривать применение этих двух списков, давайте поговорим об автоматическом создании форм.

Каждый раз, когда вы создаете форму, C++Builder помещает ее в список автоматически создаваемых форм для данного приложения. Автоматическое создание означает, что C++Builder создает форму во время запуска приложения. Формы, созданные автоматически, отображаются быстрее. Недостаток автоматического создания форм состоит в том, что ваше приложение расходует для этого дополнительную память. Другой недостаток (хотя, возможно, и не очень существенный), состоит в том, что при наличии большого количества автоматически создаваемых форм ваше приложение будет загружаться несколько дольше.



Первая форма списка Auto-create forms всегда является главной формой. Если вы заменяете главную форму, то новая выбранная форма переместится в верхнюю часть списка. Другой способ задания главной формы — перетаскивание любой из форм в окне списка Auto-create forms в верхнюю часть этого списка.

Удобство автоматического создания форм заключается в простоте вывода такой формы. Все, что вы должны сделать — это вызвать функцию Show() или ShowModal() данной формы:

```
AboutBox->ShowModal();
```

Если вы не хотите, чтобы ваши формы автоматически создавались C++Builder, вам необходимо взять на себя ответственность за создание форм перед тем, как они будут использоваться:

```
TAboutBox* aboutBox = new TAboutBox(this);
aboutBox->ShowModal();
delete aboutBox;
```

Этот пример не использует генерируемый C++Builder указатель на окно About. Приведенный код создает локальный указатель, отображает форму, а затем освобождает память, когда форма больше не нужна. Как это часто имеет место в программировании на Си++, существует несколько способов выполнить данную конкретную задачу. Поскольку C++Builder всегда создает указатель на объект формы, я мог бы переписать предыдущий код следующим образом:

```
if (!AboutBox->Handle) {
  AboutBox = new TAboutBox(this);
  AboutBox->SetParent(this);
}
aboutBox->ShowModal();
```

Этот код проверяет, была ли форма уже создана. Если нет, то создается объект и вызывается метод ShowModal(). Этот код также вызывает SetParent(), чтобы установить родительскую форму для вызываемой формы (в большинстве случаев это главная форма). Вы можете выбрать любой из этих способов, но я предпочитаю первый.



Каждый раз, когда вы создаете форму в редакторе форм, C++Builder генерирует указатель на эту форму. Если вы разрешаете C++Builder автоматическое создание формы, вы не должны волноваться относительно допустимости значения указателя. Если вы не используете автоматическое создание формы, указатель на форму будет иметь значение NULL до тех пор, пока вы явно не создадите форму и не инициализируете указатель. Если вы забыли об этом и использовали указатель прежде, чем он был инициализирован, Windows выдаст сообщение о нарушении прав доступа.

Итак, вернемся к диалоговому окну Project Options. Окно списка Auto-create forms содержит список форм, которые будут создаваться автоматически. Если вы не хотите, чтобы форма была создана автоматически, перетащите форму из списка Auto-create forms в окно списка Available forms. Используя технику drag-and-drop, вы можете перемещать сразу несколько форм из одного списка в другой. Чтобы переместить несколько форм за один раз, просто выделите нужные формы (оба окна списка поддерживают групповое выделение), после чего перетащите их вместе.



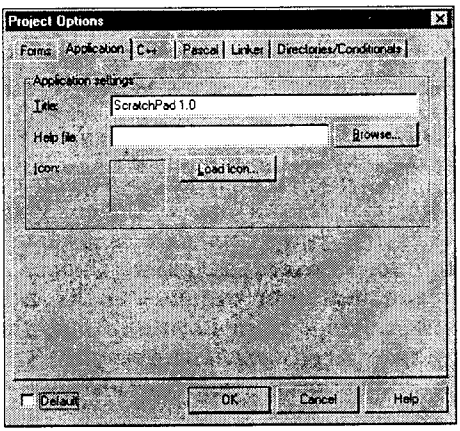
Для перемещения форм из одного списка в другой вы можете использовать кнопки, находящиеся между двумя окнами списка, но техника drag-and-drop обычно более проста.

Вкладка Application

Вкладка Application диалогового окна Project Options очень проста. (См. рис. 10.8.)

В поле Title (Заголовок) данной вкладки задается заголовок приложения. Заголовок — это текст, который выводится в панели задач Windows, когда ваше приложение находится в свернутом состоянии.

Рис. 10.8
Вкладка Application





Заголовок приложения и заголовок главной формы — это две разные вещи. Если вы хотите, чтобы в свернутом состоянии вашей программы показывалось ее имя, вы должны задать заголовок приложения в диалоговом окне Project Options. Если вы не сделали этого, то по умолчанию будет использоваться имя файла проекта.

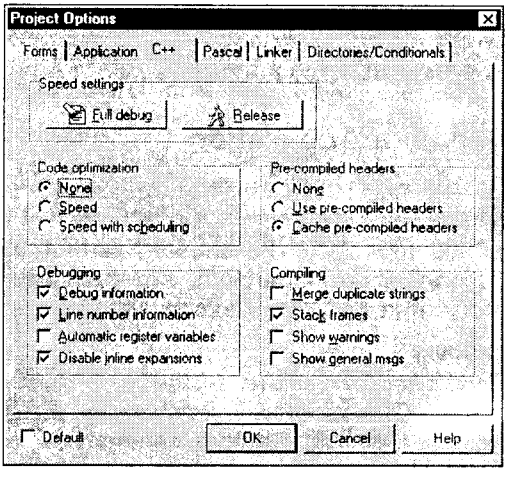
В поле Help file (Справочный файл) вкладки Application задается имя справочного файла, который использует ваше приложение. Этот справочный файл загружается при нажатии клавиши F1 во время выполнения программы. Если вы не помните имя и каталог нужного справочного файла, то можете найти его, нажав кнопку Browse. Если вы не укажете справочный файл, то нажатие клавиши F1 в вашем приложении не даст никакого эффекта.

Опция Icon позволяет выбрать значок для вашего приложения. Этот значок будет отображаться в панели задач Windows во время работы приложения, когда оно находится в свернутом состоянии. Кроме того, этот же значок будет отображаться в строке заголовка главной формы, если вы не задали значок для главной формы явным образом. Чтобы выбрать значок, щелкните на кнопке Load Icon (Загрузить значок) и найдите файл значка (.ico) с помощью диалогового окна Application Icon.

Вкладка C++

Вкладка C++ диалогового окна Project Options содержит опции, которые используются компилятором при построении проекта (см. рис. 10.9).

Рис. 10.9
Вкладка C++ диалогового окна Project Options



В верхней части этой вкладки находится раздел Speed Settings (Установка быстродействия), который содержит две кнопки. Кнопка Full Debug (Полная отладка) устанавливает заданные по умолчанию опции компилятора для типичного сеанса отладки. Скорее всего, вы будете использовать именно эти установки при отладке своих приложений. Кнопка Release (Выпуск) устанавливает опции компилятора для типичного сеанса окончательного построения программы. Используйте установки Release после того, как вы полностью отладили свое приложение и готовы к выпуску конечного продукта.

После изменения установок компилятора обязательно перестройте свой проект командой Build All.



Кнопки Full Debug и Release задают определенный набор опций компилятора для этапа отладки и окончательного выпуска приложения, соответственно. У вас всегда остается возможность изменить отдельные опции после выбора одной из этих кнопок.

Раздел Code optimization

Компилятор может быть сконфигурирован так, чтобы выполнять оптимизацию вашего кода. Когда в разделе Code optimization (Оптимизация кода) вкладки C++ оптимизация выключена (выбрана радиокнопка None), компилятор не делает никаких попыток как-либо оптимизировать ваш код.

Если вы выбираете опцию Speed (Оптимизация по скорости), компилятор генерирует код с максимальным быстродействием, независимо от его конечных размеров. Если вы включили опцию Speed with scheduling (Оптимизация по скорости с планированием), компилятор оптимизирует код таким образом, чтобы использовать преимущества конвейерных инструкций процессора Pentium (Pentium Pipeline Instructions). В большинстве случаев вы должны оставить эту опцию в том состоянии, которое будет установлено после щелчка на кнопке Full Debug или Release.



Результаты от изменения способа оптимизации могут существенно варьироваться. Каждое приложение имеет свои особенности. Иногда оптимизация по размеру существенно отражается на конечном размере исполняемого файла; в других случаях разница незначительна. Тот же самое относится и к оптимизации по скорости.

Раздел Debugging

Раздел Debugging (Отладка) вкладки C++ диалогового окна Project Options управляет генерацией кода для сеанса отладки. Этот раздел имеет четыре опции, которые описываются ниже. (Сам процесс отладки мы подробно рассмотрим завтра.)

Debug Information

Если опция Debug information (Отладочная информация) включена, C++Builder генерирует для проекта отладочную информацию. Эта информация сохраняется в отдельном файле в каталоге проекта. Имя файла, содержащего информацию для отладки, имеет расширение .tds. Например, если ваша программа называется MyApp, C++Builder создаст файл с именем MyApp.tds. Этот файл читается отладчиком во время сеансов отладки. Если вы выключите эту опцию, то во время отладки не сможете останавливаться в контрольных точках и просматривать значения переменных. Иначе говоря, вы не сможете отлаживать свою программу, если не укажете C++Builder генерировать информацию для отладки.

Line Number Information

Опция Line number information (Информация о номерах строк) сообщает C++Builder, нужно ли генерировать для проекта информацию о номерах

строк, которая используется отладчиком для пошагового выполнения кода. Данная опция устанавливается автоматически вместе с опцией Debug Information (даже если флажок это не показывает). Однако можно выключить отладочную информацию, после чего установить опцию Line number information. В этом случае вы сможете устанавливать контрольные точки и выполнять код в пошаговом режиме, но не сможете просматривать какие-либо переменные. Польза от этого заключается в меньшем размере TDS-файла. В действительности маловероятно, что вы будете использовать такое сочетание опций.

Automatic Register Variables

Если опция Automatic Register Variables (Автоматические регистровые переменные) включена, компилятор будет использовать регистровые переменные при каждой подходящей возможности. Использование регистровых переменных намного увеличивает скорость работы программы. С другой стороны, отладка программы с регистровыми переменными может сильно усложниться. Компилятор может оптимизировать ваши переменные во время отладки, сделав их недоступными для просмотра. Если переменная была оптимизирована, в окне переменных наблюдения появится сообщение Variable 'x' has been optimized and is not available (Переменная 'x' была оптимизирована и недоступна), когда вы попытаетесь ее просмотреть.



Чтобы избежать проблем с регистровыми переменными, вы можете сделать одну из двух вещей. Во-первых, вы можете выключить опцию Automatic Register Variables во время отладки своего приложения и включить ее после завершения отладки, когда ваш продукт полностью готов.

Второй способ — это объявить локальную переменной с ключевым словом volatile:

```
volatile int x;
```

Это предотвратит оптимизацию переменной компилятором и, следовательно, она будет доступна для просмотра.

Даже выключив опцию Automatic Register Variables, вы можете заставить компилятор рассматривать определенную переменную как регистровую, объявив ее с ключевым словом register. Например:

```
register int x = 20;
```

Disable Inline Expansions

Опция Disable Inline Expansions (Запретить подстановку функций) управляет тем, как компилятор обрабатывает вызовы подставляемых функций. По умолчанию, тело функции подставляется в месте ее вызова. Если же вы установите эту опцию, подставляемые функции будут обрабатываться как обычные функции. Эта опция редко используется, но при отладке некоторых подставляемых функций она может вам потребоваться.



Если вы изменяете любую из опций вкладки C++, то сразу после этого необходимо выполнить команду Build All. Это гарантирует, что все модули будут построены с одинаковыми установками компилятора.

Pre-compiled Headers

Обратите внимание на раздел Pre-compiled Headers (Прекомпилированные заголовочные файлы) во вкладке C++ диалогового окна Project Options. По существу, *прекомпилированный заголовочный файл* — это образ таблицы имен проекта, сохраненный на диске. Когда вы строите свою программу в первый раз, C++Builder создает прекомпилированный заголовочный файл. Затем C++Builder может загружать его с диска, что намного быстрее, чем компилирование заголовочных файлов при каждом построении программы. Кроме того, вы можете выбрать опцию кэширования в памяти прекомпилированного заголовочного файла. Это еще больше увеличит скорость компиляции, поскольку прекомпилированный заголовочный файл может храниться в оперативной памяти, а не загружаться каждый раз с диска.

Вы можете установить опцию Pre-compiled Headers в состояние None (Нет), Use pre-compiled headers (Использовать прекомпилированные заголовочные файлы) или Cache pre-compiled headers (Кэшировать прекомпилированные заголовочные файлы) в зависимости от ваших потребностей и аппаратных средств, доступных на вашей системе. Вам следует так или иначе использовать эту опцию, поскольку ее выключение почти всегда приводит к намного более медленному построению программы.



Опция кэширования прекомпилированных заголовочных файлов существенно ускоряет компиляцию и построение программы, если ваша система имеет достаточный объем оперативной памяти. Если вы не располагаете запасом RAM, кэширование, наоборот, может замедлить процесс построения программы. Проведите собственные тесты, чтобы определить, ускоряет или замедляет работу вашей системы кэширование прекомпилированных заголовочных файлов. Я бы рекомендовал выключить кэширование, если объем RAM вашей системы меньше 32 Мб.

Раздел Compiling

Группа опций Compiling (Компиляция) вкладки C++ управляет некоторыми аспектами работы компилятора Си++. По большей части, вы должны оставить этот набор опций в состоянии по умолчанию, пока не освоите в достаточной степени C++Builder и Си++ в целом. Назначение этих опций объясняется ниже.

Merge Duplicate Strings

Если опция Merge Duplicate Strings (Объединять дублирующиеся строки) включена, то компилятор будет размещать одинаковые строки в одних ячейках памяти. Это уменьшает размер программы, но может вызвать проблемы при изменении одной из строк.

Stack Frames

При отладке оставляйте опцию Stack frames (Фрейм стека) включенной. Закончив отладку, вы можете выключить эту опцию, чтобы компилятор генерировал более быстрый код меньшего размера, но время компиляции при этом немного увеличится. В большинстве случаев выигрыш в размере и быстродействии не настолько существенен, чтобы оправдать выключение этой опции. Как всегда, проведите свой собственный тест.

Show Warnings

Когда опция Show warnings (Показать предупреждения) включена, любые предупреждения компилятора отображаются в окне сообщений редактора кода. Я всегда оставляю эту опцию включенной. Предупреждения компилятора не стоит игнорировать, поскольку большинство из них может и должно быть устранено.



Сообщения об ошибках компиляции и компоновки всегда выводятся в окно сообщений.

Show General Msgs

Когда опция Show general msgs (Показать общие сообщения) включена, в окне редактора кода отображаются различные сообщения о состоянии компилятора. Например, при компиляции программы ScratchPad с включенной опцией Show general msgs, в окно сообщений будет выведен следующий текст:

```
[C++] Compiling: Scratch.cpp
[C++] Compiling: SPMain.cpp
[C++] Loaded cached pre-compiled headers.
[C++] Compiling: SPAbout.cpp
[Linker] Incremental Linking: D:\Projects\Scratch\Scratch.exe
```

Выключите эту опцию, если вы не хотите видеть сообщения о состоянии компилятора в окне сообщений.

Вкладка Pascal

Вкладка Pascal диалогового окна Project Options используется для установки опций компилятора Паскаля. Компилятор Паскаля используется в том случае, когда вы включили в проект C++Builder модули, написанные на языке Паскаль (.pas). Установок компилятора Паскаля довольно много, и они выходят за рамки нашей темы, так что я не буду касаться их вообще. Обращайтесь к встроенной справочной системе C++Builder, где приведена подробная информация о каждой из опций этой вкладки.

Вкладка Linker

Во вкладке Linker (Компоновщик) диалогового окна Project Options вы устанавливаете опции, которые определяют работу компоновщика. До тех

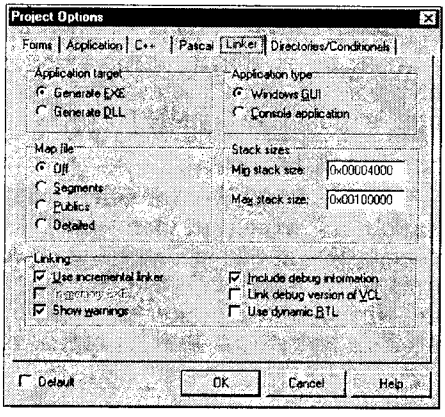


Рис. 10.10
Вкладка Linker диалогового окна Project Options

пор, пока вы детально не освоите C++Builder, вы можете вообще не трогать эту вкладку и принять установки по умолчанию. На рис. 10.10 показана вкладка Linker диалогового окна Project Options. Разделы этой вкладки рассматриваются ниже.

Раздел Application Target

Раздел Application target (Назначение приложения) определяет, будет ли результирующий файл вашего проекта исполняемым файлом или DLL. Если вы создаете свой проект, используя репозиторий объектов, эта опция устанавливается автоматически, и вы не должны беспокоиться о ее состоянии.

Раздел Application Type

Раздел Application type (Тип приложения) позволяет вам задать тип приложения. Доступный выбор — Windows GUI (приложение с графическим интерфейсом) и Console Application (Консольное приложение). Эта опция устанавливается автоматически, как и опция Application target, если вы создаете свой проект через репозиторий объектов.

Раздел Map File

Опция Map file (Файл распределения памяти) определяет, будет ли генерироваться файл с картой распределения памяти и насколько подробным будет его содержание. (Карта памяти — это продвинутое средство отладки и маловероятно, что вы его будете использовать).

Раздел Linking

Раздел Linking (Компоновка) содержит три опции компоновщика. Опция Use incremental linker указывает C++Builder, должен ли он использовать пошаговый компоновщик. Пошаговый компоновщик экономит много времени при разработке приложений. Предположим, например, что ваш проект содержит 20 модулей. Если вы модифицируете одну строку модуля, то этот модуль должен быть повторно откомпилирован и скомпонован. Когда включена опция пошаговой компоновки, повторно компонуется только тот объектный файл, который был изменен. Если пошаговая компоновка выключена, компоновщик должен повторно скомпоновать все двоичные файлы проекта независимо от того, изменялись ли они с момента последней компоновки или нет. Компоновка занимает значительное время, так что пошаговый компоновщик имеет большие преимущества при работе над проектом любого масштаба.

Пошаговая компоновка имеет два недостатка. Во-первых, когда включена опция пошаговой компоновки, первичная компоновка программы занимает большее время. Во-вторых, чтобы выполнить свою задачу, пошаговый компоновщик создает несколько файлов. Эти файлы могут быть очень большими (несколько мегабайт) и, соответственно, занимают много места на диске. Если свободное место на диске является для вас проблемой, вы можете отключить пошаговую компоновку. В противном случае вы, вероятно, захотите оставить ее включенной.



Существует, фактически, и третий недостаток пошаговой компоновки: исполняемый файл имеет несколько большие размеры. Прежде, чем выпустить конечный продукт, вы должны перестроить проект командой Build All с выключенной опцией Use incremental linker.

Опция Show warnings указывает C++Builder выводить любые предупреждения компоновщика в окне сообщений редактора кода.

Опция Link debug version of VCL (компоновать с отладочной версией VCL) компонует версию VCL, в которую встроена отладочная информация. Это позволяет вам войти в исходный код VCL при отладке своего приложения.



Вообще говоря, трассировка кода VCL — не очень плодотворное занятие, особенно если вы не являетесь опытным программистом. Любые проблемы, с которыми вы столкнетесь в своем приложении, с несравненно большей вероятностью будут связаны с вашим кодом, а не с кодом VCL. Иногда вхождение в исходный код VCL полезно, но, по моему опыту, это случается редко. Помните также, что VCL написана на Объектном Паскале и, если ее исходный код напоминает вам иностранный язык, то это именно так и есть.

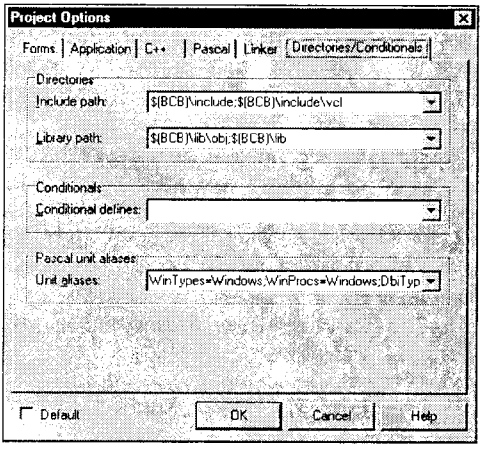
Раздел Stack Sizes

Раздел Stack sizes (Размеры стека) позволяет установить минимальный и максимальный размеры стека для проекта. Как правило, изменять значения этих опций нет необходимости.

Вкладка Directories/Conditionals

Вкладка Directories/Conditionals (Каталоги/Условные определения) диалогового окна Project Options служит для задания каталогов, которые ваш проект использует для поиска библиотечных и заголовочных файлов. Вы можете также задать каталог, в котором компилятор будет размещать выходные файлы. (См. рис. 10.11.) Поля этой вкладки описаны в следующих подразделах.

Рис. 10.11
Вкладка Directories/Conditionals



Include Path

Установка Include Path (Путь для включаемых файлов) задает путь, по которому C++Builder будет искать заголовочные файлы, необходимые для построения вашего приложения (файлы с расширениями .h и .hpp). По умолчанию это поле установлено таким образом, чтобы указывать на различные каталоги C++ Builder, в которых находятся системные заголовочные файлы. Вы должны оставить это поле в состоянии по умолчанию, если только вы не собираетесь использовать библиотеку независимых производителей, которая находится в отдельном каталоге. Если вам нужно добавить каталоги в поле Include Path, вы можете добавить их в конец списка существующих каталогов. Разделяйте каталоги точкой с запятой и убедитесь в том, что вы включили полный путь.

Library Path

Поле Library Path (Путь для библиотек) содержит пути, по которым C++Builder должен искать библиотечные файлы (.lib). Так же, как и для поля Include Path, вы можете добавлять сюда другие каталоги, отделяя каждый путь точкой с запятой.



Не удаляйте список заданных по умолчанию каталогов в полях Include Path и Library Path. Если вам нужно изменить эти поля, добавьте свои каталоги в конец перечисленных, но не удаляйте ни один каталог, заданный по умолчанию. В противном случае ваше приложение не будет компилироваться.

Conditional Defines

В поле Conditional defines (Условные определения) указываются все макроопределения, которые вы хотите добавить на уровне проекта. Например, чтобы добавить поддержку диагностических макросов TRACE и WARN, вы могли бы добавить в поле Conditional defines следующий текст:

```
__TRACE;__WARN
```

Обратите внимание, что каждая директива #define отделяется от другой точкой с запятой.

Pascal Unit Aliases

Поле Pascal Unit Aliases (Псевдонимы модулей Паскаля) связывает имя модуля на Паскале с определенным заголовочным файлом Си++. Псевдонимы разделяются точкой с запятой.

Редактор кода C++Builder

C++Builder, по своей природе, является инструментом визуального программирования. Однако любая программа содержит код, который должен быть написан вручную. После разработки интерфейса пользователя для своего приложения с помощью впечатляющих визуальных средств C++Builder, вам придется провести много времени, работая с редактором кода C++Builder. Редактор кода имеет целый ряд возможностей, которые вы сразу оцените, как только с ними познакомитесь.

В этом разделе вы узнаете о следующем:

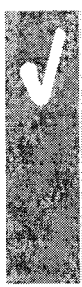
- ☐ Основные операции редактора
- ☐ Специальные возможности редактора
- ☐ Контекстное меню редактора
- ☐ Изменение опций редактора



Редактор кода C++Builder позволяет выбрать одну из четырех схем команд клавиатуры: Default (По умолчанию), IDE Classic (Классическая IDE), BRIEF (эмуляция редактора BRIEF) и Epsilon (эмуляция редактора Epsilon). Далее в этой главе предполагается, что вы пользуетесь схемой Default. Если вы уже знакомы с одной из других схем клавиатуры, то можете игнорировать ссылки на конкретные сочетания клавиш.

Основные операции редактора

Я полагаю, что вы уже знакомы с такими операциями, как ввод и удаление текста, выделение текста мышью, вырезание, копирование, вставка и т. д, так что я не буду тратить на время на этот уровень работы с редактором кода.



Если вы провели много времени за написанием кода, то скорее всего привыкли интенсивно использовать клавиатуру. Если это так, то вы, вероятно, пользуетесь сочетаниями клавиш для таких простых вещей, как вырезание, копирование и вставка. Если у вас мало опыта работы с клавиатурой (или вы просто предпочитаете использовать мышь), то вам следует настроить оперативную панель C++Builder, добавив в нее кнопки для вырезания, копирования и вставки. Если мы с вами похожи, то при написании своих программ вы также будете много вырезать, копировать и вставлять и, независимо от выбранного вами метода выполнения этих операций, он войдет в вашу практику.

В конечном счете, редактор кода C++Builder — это типичный редактор кода. Одной из его отличительных особенностей является выделение синтаксиса, что помогает сразу идентифицировать ключевые слова, строки, числовые константы и комментарии. Немного позже мы рассмотрим различные установки редактора.

Редактор кода — это окно, построенное на основе вкладок. Вы можете открыть столько файлов, сколько вам требуется, и каждый из них будет представлен вкладкой в верхней части окна. В заголовке вкладки отображается имя файла. Чтобы перейти к какому-либо открытому файлу, просто щелкните на соответствующей ему вкладке. Если вы открыли большее количество вкладок, чем может одновременно уместиться в окне редактора, то появятся кнопки прокрутки вкладок.

Строка состояния в нижней части редактора кода содержит информацию о состояния редактора. В левой панели строки состояния отображаются текущий номер строки и позиция курсора в строке. Если файл был изменен со времени последнего сохранения, в центральной панели строки состояния появится текст Modified (Модифицирован). Правая панель строки состояния показывает текущий режим вставки текста: Insert (Вставка) или Overwrite

(Замена). Если файл имеет статус «только для чтения», в этой панели будет показан текст Read only.

Окно редактора имеет левое поле, которое называется *пробельным полем (gutter)*. Пробельное поле используется для отображения специальных значков на различных этапах разработки. Например, когда вы устанавливаете контрольную точку отладчика (это рассматривается завтра), в пробельное поле помещается значок Stop красного цвета. Если вы устанавливаете в данной строке закладку (я скажу об этом ниже), то в пробельное поле помещается значок, представляющий закладку.



Иногда пробельное поле может раздражать. Если вы случайно щелкнете на нем, пытаясь выбрать текст или поместить в него курсор, то обнаружите в этой строке контрольную точку. Чтобы ее убрать, щелкните на пробельном поле в данной строке еще раз.

Открытие и сохранение файлов

В открытии и сохранении файлов в редакторе кода нет ничего особенного. Но, тем не менее, нужно подчеркнуть, что существует различие между открытием проекта и исходного файла. Когда вы выбираете в главном меню пункт File | Open Project, у вас запрашивается имя открываемого файла проекта (.mak). Когда вы выбираете в главном меню пункт File | Open, вы можете открыть любой текстовый файл. (Вы можете также открыть файл формы, но это — совсем другая история.) Оба пункта меню, Open и Open Project, имеют соответствующие кнопки в оперативной панели.



Если вы открываете исходный файл (.cpp), который является модулем исходного кода формы, C++Builder откроет этот файл в редакторе кода, а также откроет форму в редакторе форм.

Вы можете открыть несколько файлов одновременно. Для этого в диалоговом окне Open выделите нужные вам файлы и нажмите OK. Все выделенные файлы будут загружены на отдельные вкладки.



Для открытия файлов можно также использовать технику drag-and-drop. Например, вы можете выделить файл (или группу файлов) в Windows Explorer, после чего перетащить их в редактор кода. Файлы будут открыты в редакторе.

Чтобы сохранить файл, выберите в главном меню пункт File | Save или File | Save As, или нажмите на клавиатуре клавиши Ctrl+S. Если файл не был предварительно сохранен, появится диалоговое окно Save As, где вы сможете ввести имя файла.

Выделение текста

Хотя выделение текста и является базовой операцией, я думаю, что не помешает напомнить вам о нескольких основных способах выделения текста в редакторе кода C++Builder.

Чтобы выделить короткий блок текста, вы можете протащить курсор мыши через выделяемый текст. Выделенный текст можно вырезать, скопи-

ровать или вставить. Чтобы выделить более длинный блок кода, вы можете использовать способ щелчок+Shift+щелчок. Сначала щелкните в начале блока, который вы хотите выделить. Затем нажмите и удерживайте клавишу Shift, после чего щелкните еще раз в конце блока. Весь текст между начальной и конечной точкой будет выделен.

Другая полезная возможность — это быстрое выделение отдельного слова. Чтобы выделить ключевое слово, имя функции или переменной, просто дважды щелкните на этом слове. После этого вы можете выполнять над выделенным словом любые операции по редактированию.



Чтобы выделить одну строку программы с помощью мыши, щелкните в начале строки и протащите курсор прямо вниз до начала следующей строки. Для выделения одной строки программы с помощью клавиатуры, сначала нажмите клавишу Home, чтобы перейти в начало строки, а затем используйте сочетание Shift+стрелка вниз.

Существует множество сочетаний клавиш, которые могут использоваться для выделения текста и выполнения работы по редактированию. Полный список всех доступных сочетаний приведен в справочной системе C++Builder.



При редактировании программы часто приходиться добавлять, удалять или перемещать блоки текста. Иногда вам нужно будет добавить отступ для целого блока кода. В других случаях вам потребуется убрать отступ. Чтобы ввести отступ для блока, выделите этот блок и нажмите на клавиатуре Ctrl+Shift+I. Чтобы убрать отступ, нажмите Ctrl+Shift+U.

Отмена

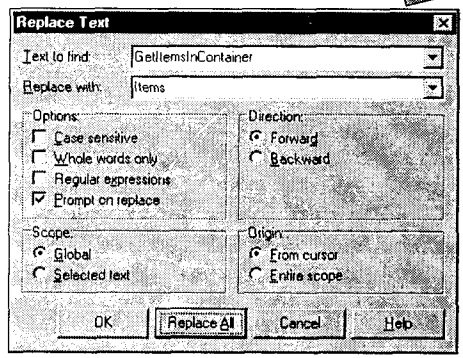
Редактор кода имеет практически неограниченное число уровней отмены (Undo) (32 767 по умолчанию). Обычно вы можете отменять команды только до того состояния, которое было при последнем сохранении файла. Изменив опции редактора, у вас будет возможность отменить прошлые команды даже после сохранения файла. Мы обсудим опции и настройки редактора позже, в разделе этой главы «Изменение опций редактора».

Поиск и замена

Поиск (Find) и замена (Replace) довольно интенсивно используются при написании программ. Команда поиска поможет найти определенный фрагмент кода или определенную переменную в вашем коде. Замена может использоваться для изменения имени переменной или функции. Возможностей бесконечно много.

Диалоговые окна Find и Replace выполняют в C++Builder более или менее стандартные действия по поиску и замене. Для вывода диалогового окна Find выберите в главном меню пункт Search | Find или нажмите Ctrl+F. Для вывода диалогового окна Replace, выберите в главном меню Search | Replace или нажмите Ctrl+R. На рис. 10.12 показано диалоговое окно Replace. За исключением нескольких очевидных различий, окно Find содержит те же самые опции.

Рис. 10.12
Диалоговое окно Replace Text



По большей части, опции диалоговых окон Find и Replace делают все в полном согласии со своими названиями. Например, при выборе опции Case Sensitive (Учитывать регистр) вы должны ввести текст для поиска в точности так, как он появляется в исходном файле.

Единственная опция, которая требует некоторых разъяснений — это опция Whole words only (Только целые слова). Очевидно, что код Си++ не является обычным текстом. Возьмите, например, следующую строку:

```
Memo->Caption = GetCaption();
```

В этом случае Memo, Caption и GetCaption не могут рассматриваться как отдельные слова, поскольку они окружены специальными символами синтаксиса Си++. Если вы ищете слово Memo и включили опцию Whole words only, операция Find проигнорирует символы синтаксиса и найдет слово Memo в этой строке.

Опция Regular expressions (Обычные выражения) также требует объяснений. Когда эта опция включена, вы можете использовать специальные групповые символы при выполнении поиска. Полное описание групповых символов приведено в справочной системе C++Builder, в статье Regular Expressions.

Самое надежное при замене текста — оставить включенной опцию Prompt on replace (Запрос при замене). Если вы выполняете операцию Replace All (Заменить все) и включили эту опцию, редактор выделяет каждое найденное слово и запрашивает у вас подтверждение на его замену. Пользуйтесь операцией Replace All с осторожностью, поскольку при ее выполнении можно легко ошибиться, и не забывайте о возможностях команды Undo.

Остальная часть опций Find and Replace очевидна и не нуждается в дополнительных разъяснениях.



В состав C++Builder входит утилита, которая позволяет искать текст по всем исходным файлам. Утилита называется grep и может быть найдена в каталоге CBuilder\Bin. Эта программа командной строки является мощной утилитой поиска. К сожалению, программа grep не интегрирована в IDE C++Builder, как это сделано в Borland C++. Тем не менее, вы можете запускать grep из командной строки или поискать в сетевых источниках программу grep с интерфейсом Windows от независимых производителей.

Получение справки

Одна из наиболее полезных возможностей редактора кода — его интеграция со справочной системой C++Builder. Просто поместите курсор редактора поверх ключевого слова Си++, свойства или метода VCL, или любого другого специфического для C++Builder текста и нажмите F1. Если для текста под курсором существует статья в справочных файлах C++Builder, будет запущена WinHelp с показом соответствующей страницы. Если для выбранного текста статьи не существует, то будет выведено сообщение об ошибке. Эта возможность чрезвычайно полезна, когда вы не можете запомнить, как использовать определенный элемент C++Builder, Си++ или VCL.

Специальные возможности редактора

Редактор кода C++Builder имеет несколько возможностей, которые являются чрезвычайно полезными при написании большого количества кода. Эти возможности рассмотрены в следующих подразделах.

Использование закладок

Вы можете установить в своем коде закладку (bookmark), чтобы временно отметить нужное место исходного файла. Например, вам часто приходится выходить из блока кода, над которым вы работаете, чтобы просмотреть другую часть кода или скопировать фрагмент кода из другого места программы. Установив закладку на нужной строке, вы всегда сможете вернуться обратно простым нажатием клавиши. Вы можете устанавливать до 10 закладок одновременно.

Чтобы установить закладку в определенном месте, нажмите Ctrl+Shift и номер устанавливаемой закладки. Например, чтобы установить закладку с номером 0 (первая закладка), поместите курсор редактора в нужное место и нажмите Ctrl+Shift+0. В пробельном поле окна редактора кода в той строке, где вы установили закладку, появится значок закладки с ее номером. На рис. 10.13 показан редактор кода с установленной в строке закладкой.

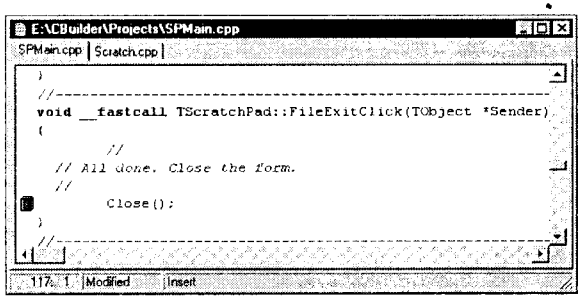


Рис. 10.13
Редактор кода с установленной закладкой

Чтобы вернуться к определенной закладке, нажмите Ctrl вместе с номером закладки. В предыдущем примере, чтобы вернуться к закладке с номером 0, нужно нажать Ctrl+0. Для удаления закладки поместите курсор редактора в строку, содержащую закладку, и снова нажмите Ctrl+Shift+0.



Закладки могут быть установлены для каждого файла, который вы открыли в редакторе кода. Например, вы можете установить закладку 0 в первом файле исходного текста и другую закладку 0 во втором исходном файле. Это означает, что вы не можете искать закладку с одним и тем же номером по всем файлам. Если вы установили закладку 0 в файле Unit1.cpp, вы не можете нажать Ctrl+0 в файле Unit2.cpp и перейти к закладке в Unit1.cpp.

Для иллюстрации работы с закладками сделайте следующее:

1. Откройте любой исходный файл в редакторе кода.
2. Прокрутите файл почти до конца и щелкните на какой-нибудь строке кода.
3. Нажмите Ctrl+Shift+0, чтобы установить закладку. В пробельном поле редактора кода появится значок закладки.
4. Нажмите Ctrl+Home, чтобы перейти в начало исходного файла.
5. Теперь нажмите Ctrl+0, чтобы вернуться к строке с закладкой. Окно редактора кода покажет строку кода с установленной закладкой, а курсор будет помещен в то место строки, где он находился в момент установки закладки.
6. Нажмите Ctrl+Shift+0, чтобы удалить закладку. Закладка будет удалена, и ее значок исчезнет из пробельного поля редактора.

Закладки имеют временный характер. Если вы закроете исходный файл, а затем снова откроете его, то информация о расположении закладок не восстановится.

Инкрементный поиск

Вы можете использовать опцию инкрементного поиска (incremental search) для быстрого поиска короткой последовательности символов. Чтобы начать инкрементный поиск, выберите в главном меню пункт Search | Incremental Search или нажмите Ctrl+E на клавиатуре. Чтобы разобраться в том, как работает инкрементный поиск, проще всего проделать следующее упражнение:

1. Создайте новый текстовый файл из репозитория объектов. (Не имеет значения, открыт ли в данный момент какой-нибудь проект.)
2. Введите следующий текст:

```
Learning to write Windows
programs a bit at a time
is not so bad. Isn't it
time you got back to work?
```

3. Переместите курсор в начало файла (Ctrl+Home).
4. Нажмите Ctrl+E, чтобы начать инкрементный поиск. Вы будете искать слово back. Обратите внимание, что в строке состояния редактора кода появился текст Searching for:.
5. Введите букву b. Вы видите, что в слове bit выделился символ b. Да... похоже, что вы не это искали.
6. Теперь введите букву a. Будет найдено следующее вхождение в текст символов ba, на этот раз — слово bad. Опять промах.
7. Введите c. В слове back будут выделены символы bac. Теперь введите k. В строке состояния редактора кода теперь находится текст Searching for: back и слово back выделено. Поздравляю! Вы нашли то, что искали.

8. Нажмите на клавиатуре Esc (или Enter), чтобы остановить инкрементный поиск. Закройте текстовый файл без сохранения.

Итак, мы рассмотрели все, что относится к инкрементному поиску. Этот тип поиска удобен, когда вы ищете короткие отрывки текста.



Если вы сделаете ошибку, вводя символы при выполнении инкрементного поиска, то можете использовать клавишу Backspace для удаления из строки поиска последнего введенного символа.

Поиск парных фигурных скобок

Как вы уже убедились, код Си++ может часто стать довольно замысловатым, когда вы начинаете использовать вложенные условные операторы if, пары if-else, и т. д. По правде говоря, в них легко заблудиться. Редактор кода имеет возможность поиска фигурной скобки, парной той, на которой в данный момент находится курсор. Чтобы найти парную фигурную скобку, поместите курсор перед фигурной скобкой (не имеет значения, какая это фигурная скобка — открывающая или закрывающая). Теперь нажмите на клавиатуре Ctrl+[. Курсор перейдет к той фигурной скобке, которая соответствует указанной вами. Нажмите Ctrl+[еще раз, и курсор вернется назад к фигурной скобке, с которой вы начали. Вы по-прежнему можете заблудиться в лабиринте фигурных скобок в длинном ряду операторов if, но теперь, по крайней мере, вы знаете, как найти выход.

Контекстное меню редактора кода

Подобно большинству различных окон, с которыми вы сталкиваетесь в C++Builder, редактор кода имеет собственное контекстное меню. Это меню можно разделить на две части: пункты редактора и пункты отладчика. Я оставлю пункты контекстного меню, относящиеся к отладчику, до завтра, когда мы займемся отладкой, а здесь расскажу только о пунктах редактора. В таблице 10.2 приведен список этих пунктов и их описание.

Таблица 10.2. Пункты контекстного меню редактора кода

Пункт	Описание
Open Source/Header File	Если заголовочный файл, соответствующий текущему модулю исходного кода, не открыт в редакторе, то выбор этого пункта меню открывает заголовочный файл, создает для него новую вкладку и делает это окно активным. Когда открыты оба файла, .cpp и .h, выбор этого пункта последовательно переключает фокус ввода между двумя файлами.
Close Page	Закрывает активную вкладку в окне редактора. Если файл данной вкладки был изменен с момента последнего сохранения, появится запрос на сохранение файла.
Open File At Cursor	Открывает файл, на который указывает курсор. Этот пункт работает только в том случае, когда текст под курсором является именем файла исходного кода. Например, если у вас есть заголовочный файл, включенный директивой #include "myclass.h", вы можете установить курсор на имя файла и выбрать этот пункт меню, чтобы открыть данный файл. Файл будет помещен в новое окно редактора, и это окно станет активным.

Пункт	Описание
New Edit Window	Открывает новую копию редактора кода. Это удобно, если вам нужно сравнить два файла.
Topic Search	Выводит справочную статью для элемента, на который указывает курсор (если эта статья существует). Действует аналогично нажатию F1 на клавиатуре.
View As Form	Если активный исходный модуль в редакторе кода имеет соответствующую ему форму, выбор этого пункта меню приведет к переключению на редактор форм, где будет отображена данная форма. (Тот же самое, что нажатие F12.)
Read Only	Переключает режим редактирования текущего активного файла между вариантами read-only (только для чтения) и read/write (чтение/запись). Если установлен режим read-only, то файл нельзя изменить, хотя текст может быть выделен и скопирован в буфер обмена. Строка состояния отображает текст Read Only, чтобы показать, что файл доступен только для чтения. После закрытия и повторного открытия файла снова будет установлен режим read/write.
Message View	Отображает или скрывает окно сообщений C++Builder. Окно сообщений появляется автоматически, когда происходит ошибка компиляции/компоновки или при выводе предупреждения, но можно специально скрыть или показать окно сообщений, используя эту команду.
Properties	Выводит диалоговое окно Environment Options, в котором можно установить опции редактора.

В зависимости от текущего состояния редактора кода и типа открытого файла, некоторые из пунктов, перечисленных в таблице 10.2, могут быть недоступны.

Изменение опций редактора

Опции редактора занимают три вкладки в диалоговом окне Environment Options. Для вывода этого диалогового окна выберите в главном меню пункт Options | Environment.



Чтобы просмотреть опции редактора кода, вы можете также выбрать пункт Properties в контекстном меню редактора. Отличие этого метода состоит в том, что диалоговое окно Environment Options будет содержать только те вкладки, которые имеют отношение к редактору.

Три вкладки в Environment Options, которые относятся к редактору кода — это Editor, Display и Colors. Мы рассмотрим их ниже.

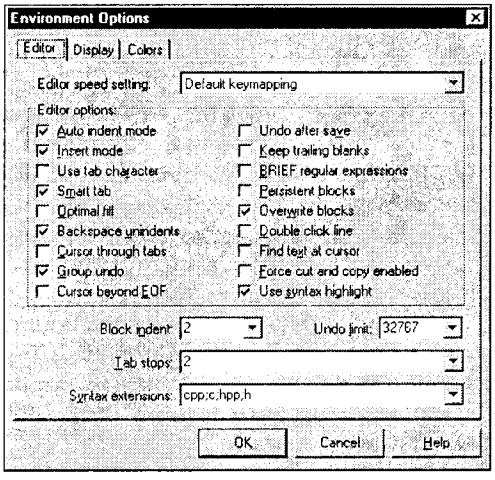
Вкладка Editor

Вкладка Editor (Редактор) диалогового окна Environment Options позволяет управлять работой редактора. Как видно из рис. 10.14, эта вкладка содержит достаточно большое число опций.

В верхней части вкладки находится комбинированный список, озаглавленный Editor speed setting. Здесь вы выбираете один из четырех вариантов установок: Default keymapping (Схема по умолчанию), IDE Classic (Класси-

ческая IDE), BRIEF emulation (Эмуляция редактора BRIEF) или Epsilon emulation (Эмуляция редактора Epsilon). Если вы измените установку этого комбинированного списка, опции редактора также изменятся, чтобы отразить значения по умолчанию для выбранной вами схемы клавиатуры.

Рис. 10.14
Вкладка Editor диалогового окна Environment Options





Если вы только начинаете программировать или уже работали с другими компиляторами Borland, используя для клавиатуры схему по умолчанию, то вы не должны волноваться, что могли что-то пропустить. Те из вас, кто годами работал с определенным типом редактора, будут рады узнать, что могут продолжать пользоваться хорошо знакомыми сочетаниями клавиш и опциями. Для этого нужно просто изменить установку опции Editor speed setting в данной вкладке и во вкладке Display.

Ближе к нижней части экрана вы видите поля Block indent (Отступ блока) и Tab stops (Шаг табуляции). Вы можете использовать эти два поля, чтобы изменять величину отступа для блока кода, или расстояние между соседними позициями табуляции. Отступы блоков обсуждались в разделе «Выделение текста».



Настоящие программисты используют шаг табуляции в два или три символа. (Я использую шаг табуляцию в две позиции.)

По всей видимости, ограничение уровня Undo (отмены) значением 32 767 достаточно в большинстве случаев, поэтому я сомневаюсь, что вам потребуется изменять эту установку. Поле Syntax extensions (Расширения для синтаксиса) позволяет выбрать типы файлов, для которых будет применяться выделение синтаксических элементов. Например, вы вряд ли захотите, чтобы синтаксические элементы выделялись в обычных текстовых файлах (.txt), которые вы открываете в редакторе кода, поэтому по умолчанию этот тип файлов не указан в поле Syntax extensions.

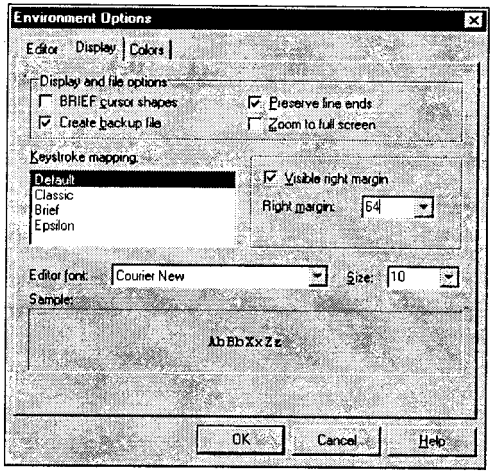
В средней части вкладки Editor вы обнаружите целую группу опций редактора. Здесь так много доступных опций, что трудно определить, какие

из них наиболее важны. Обратитесь к справочной системе C++Builder. Находясь на этой вкладке, просто нажмите F1 или щелкните на кнопке Help, и вы увидите разъяснения по каждой опции. Как и с некоторыми другими опциями, которые мы рассмотрели сегодня, возможно, что вам будет достаточно значений по умолчанию, предлагаемых C++Builder (за исключением шага табуляции и отступа для блоков!)

Вкладка Display

Вкладка Display диалогового окна Environment Options содержит дополнительные опции, которые относятся к непосредственному отображению текста в окне редактора кода. (См. рис. 10.15.)

Рис. 10.15
Вкладка Display



В разделе Display and file options (Опции дисплея и файла) вы найдете опцию BRIEF cursor shapes (Формы курсора редактора BRIEF). Включите эту опцию, если вы предпочитаете видеть в окне редактора горизонтальный курсор, а не вертикальный. Включите опцию Create backup file (Создавать резервную копию файла), если вы хотите, чтобы C++Builder создавал резервную копию при каждом сохранении файла или проекта. Расширения резервных файлов начинаются с тильды (~). Например, имя резервной копии для файла с именем MyApp.cpp выглядело бы как MyApp.~cp.



Мне обычно мешает большое количество резервных файлов, засоряющих каталоги проектов, и я выключаю опцию резервирования. Решайте сами.

Опция Zoom to full screen (Увеличить до полного экрана) определяет действия редактора кода при разворачивании окна. Если эта опция включена, то развернутое окно редактора кода заполнит целиком весь экран. Когда эта опция выключена (по умолчанию), верхняя граница развернутого окна редактора кода совпадет с нижней границей главного окна C++Builder. Иными словами, если эта опция выключена, то главное окно C++Builder всегда будет видимым при разворачивании окна редактора кода.

Вы можете также выбрать, будет ли окно редактора иметь видимое правое поле (опция Visible right margin). Правое поле не ограничивает вас — вы

по-прежнему можете вводить текст, выйдя за правую границу — но оно дает вам визуальный сигнал, что строка становится слишком длинной.

В этой вкладке вы можете также изменить шрифт и размер шрифта редактора кода. Для этой цели предусмотрено два комбинированных списка. В списке шрифтов перечислены только моноширинные экранные шрифты; пропорциональные шрифты и шрифты принтера там отсутствуют. Выберите тот шрифт и размер, которые лучше всего вам подходят. Окно предварительного просмотра (Sample) дает наглядное представление о том, что вы выбрали.

Вкладка Colors

Вкладка Colors (Цвета) диалогового окна Environment Options позволяет выполнить полную настройку цветов окна редактора кода и синтаксических элементов. (См. рис. 10.16.)

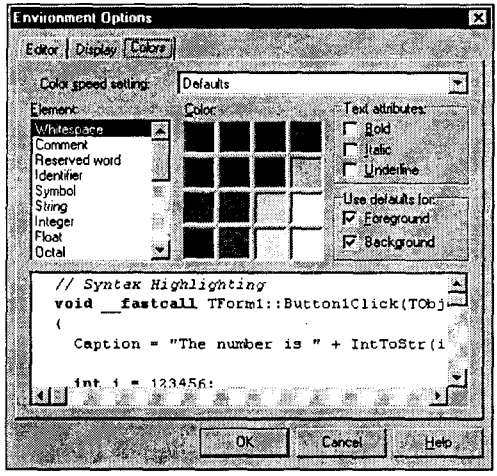


Рис. 10.16
Вкладка Colors диалогового окна Environment Options

В верхней части вкладки расположен комбинированный список Color speed setting. Этот список позволяет вам выбрать одну из четырех цветовых схем. Вы также можете использовать цветовую схему в качестве основы для создания своей собственной цветовой схемы.

Вкладка Colors очень проста в использовании. В нижней части вкладки расположено текстовое окно, которое содержит типовой код. Если вы щелкнете на одном из ключевых элементов кода, этот элемент будет выделен в списке Element, а текущие установки для этого элемента будут отображены в палитре цветов Colors. Чтобы изменить цвет переднего плана, фона и текстовые атрибуты этого элемента, просто выберите те установки, которые вам нравятся. Например, ключевые слова отображаются полужирным шрифтом с передним планом черного цвета и белым цветом фона (для цветовой схемы по умолчанию). Чтобы установить для ключевых слов полужирный шрифт зеленого цвета, щелкните в окне Sample на ключевом слове void, а затем измените цвет переднего плана на зеленый. Цвет текста в окне Sample изменится соответствующим образом. Продолжайте заменять цвета согласно задуманной вами схеме. Когда вы нажмете OK, вид редактора кода изменится согласно новой цветовой схеме.

Заключение

Сегодня вы узнали много таких вещей, которые часто упускаются из виду. Я надеюсь, что вы приняли к сведению некоторые советы, касающиеся работы с проектами и редактором кода C++Builder. Вы также получили разъяснение по некоторым опциям проекта и редактора. Даже если эта глава и не произвела на вас большого впечатления, полученная здесь информация понадобится вам в дальнейшем.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

В окне менеджера проектов имена модулей моего проекта написаны то полужирным, то нормальным текстом. Почему?

Любые модули, которые вы изменили и пока не сохранили, показываются в окне менеджера проектов полужирным текстом. Не устаревшие модули (те, которые не нужно сохранять, поскольку они не изменялись) показаны нормальным текстом.

Каждый раз, когда я пробую изменить имя модуля исходного текста проекта в проводнике проекта, оно возвращается к первоначальному значению. Почему?

Вы не можете изменять имена файлов с исходным текстом в проводнике проекта. В проводнике проекта можно изменять только те объекты, которые имеют свойство Name. Иными словами, вы можете изменять только имена форм и компонентов. Чтобы изменить имя файла исходного текста, выберите пункт File | Save As в главном меню редактора кода.

После запуска моего приложения его главная форма не отображается, а вместо нее выводится одно из диалоговых окон. В чем причина?

Вы случайно установили в качестве главной формы приложения одну из своих диалоговых форм. Откройте диалоговое окно Project Options, щелкните на вкладке Forms и выберите свою главную форму в комбинированном списке Main Form. Запустите программу снова, и вы увидите, что проблемы больше нет: после запуска программы отображается главная форма.

Многочисленные опции компилятора и компоновщика запутали меня. Действительно ли я должен знать о каждой из этих опций, чтобы писать программы в C++Builder?

Нет. Заданные по умолчанию опции проекта хорошо работают почти для всех приложений C++Builder. Со временем вы начнете глубже изучать компилятор и компоновщик, и тогда вам потребуется знание опций проекта. До этого момента можете о них не беспокоиться.

Когда мое приложение свернуто, его значок и заголовок не соответствуют тем, что я установил в главной форме приложения. Почему?

Установка значка и заголовка главной формы не влияет на то, что отображает ваше приложение в свернутом состоянии. Чтобы установить заголовок и значок приложения, перейдите в диалоговое окно Project Options, выберите вкладку Application и введите здесь имя приложения и значок.

Могу ли я открыть одновременно несколько исходных файлов в редакторе кода?

Да. Вы можете либо выбрать пункт меню File | Open и выделить несколько файлов, либо можете выделить группу файлов в Windows Explorer и перетащить их в редактор кода.

Могу ли я найти и заменить имя переменной сразу во всех своих исходных файлах?

Нет. Вам необходимо открыть каждый исходный файл и выполнить поиск и замену через диалоговое окно Replace. Вы можете, однако, использовать клавишу F3, чтобы повторить последнюю команду замены или поиска. Следите за тем, чтобы случайно не изменить имя одной из переменных, созданных C++Builder.

Я обнаружил, что 32 767 уровней отмены недостаточно для моих целей. Что вы могли бы предложить?

Не выкидывайте работу целого дня.

Вопросы

1. Как можно быстро переключаться между формой и исходным кодом модуля, работая в C++Builder?
2. Если вы удаляете файл из своего проекта в менеджере проектов, удаляется ли этот файл с жесткого диска?
3. Как задать главную форму приложения?
4. Что означает отсутствие автоматически создаваемых форм?
5. Какое минимальное количество памяти должен иметь ваш компьютер, чтобы можно было включить опцию кэширования прекомпилированных заголовочных файлов?
6. Какое значение имеет генерация отладочной информации для вашего приложения?
7. Если вы не определили выходной каталог в диалоговом окне Project Options, где C++Builder создаст ваш .exe-файл?
8. Какое сочетание клавиш используется в редакторе кода для сохранения файла?
9. Как установить закладку в окне редактора кода? Сколько закладок вы можете установить?
10. Как перевести файл в редакторе кода в режим read-only (только для чтения)?

Упражнения

1. Создайте новое приложение. Откройте менеджер проектов. Щелкните на кнопке Add Unit, чтобы добавить в проект новый модуль. Найдите каталог \CBuilder\Examples\Apps\Contacts и выберите файл с именем Contacts.cpp. Щелкните OK, чтобы включить файл в проект.
2. Удалите модуль Contacts.cpp из проекта, созданного в упражнении 1.
3. Откройте проект ScratchPad. Замените главную форму на форму AboutBox. Закройте диалоговое окно Project Options и запустите программу. После запуска программы будет выведено окно About. Закройте окно About, чтобы завершить программу, и снова установите форму ScratchPad в качестве главной формы приложения.
4. Откройте приложение RichEdit, которое поставляется вместе с C++Builder. Перейдите в диалоговое окно Project Options и убедитесь в том, что во вкладке C++ установлена оптимизация по скорости. Постройте программу командой Build All. Проверьте размер результирующего .exe-файла. Вернитесь в диалоговое окно Project Options и установите оптимизацию по размеру. Еще раз выполните построение программы командой Build All. Проверьте еще раз размер исполняемого файла и сравните его с предыдущим. Насколько меньше или больше был файл, когда компилятор оптимизировал программу по размеру?
5. Откройте любой исходный файл в редакторе кода. Установите четыре закладки в произвольных строках файла. Переходите от закладки к закладке и наблюдайте результат в редакторе кода. После завершения упражнения удалите все закладки.
6. Откройте проект ScratchPad (или любой другой) и перейдите в редактор кода. Просмотрите исходный файл главной формы проекта. Выберите в главном меню пункт Search | Find. Введите Click в окне Text To Find и нажмите OK, чтобы найти первое вхождение слова Click.
7. Нажмите F3 несколько раз, чтобы повторять поиск до тех пор, пока не будет пройден весь файл.
8. Продолжая работать с тем же проектом, нажмите Ctrl+Home, чтобы перейти в начало файла. Нажмите Ctrl+R, чтобы вывести диалоговое окно Replace Text. Введите Click в окне Text To Find и Test в окне Replace With. Выключите опцию Prompt On Replace и щелкните на кнопке Replace All. Прокрутите файл, чтобы увидеть результат. **Внимание:** выберите затем Edit | Undo, чтобы отменить операцию Replace. Закройте проект без сохранения (просто ради лишней безопасности).
9. Откройте файл в редакторе кода. Выберите пункт Properties в контекстном меню редактора кода. Измените цвет выделения синтаксических элементов на темно-серый для строк, целых чисел и чисел с плавающей точкой. Нажмите OK, чтобы проверить результат своих действий.
10. Верните цветовую схему в состояние по умолчанию.

Работа с отладчиком



Кент Рейсдорф



дной из главных частей IDE C++Builder является встроенный отладчик. Он позволяет вам легко устанавливать контрольные точки, наблюдать за переменными, инспектировать объекты и многое другое. С помощью отладчика вы можете видеть то, что происходит с программой во время ее выполнения. Хороший отладчик просто необходим для эффективной разработки программ.

Отладка относится к тем вещам, на которые легко не обратить внимания. Скажу вам по секрету, что когда я только начинал программировать в Windows (не с C++Builder, конечно), то долгое время игнорировал отладчик, поскольку был полностью загружен уже тем, что учился собственно программированию для Windows. Когда я обнаружил, насколько ценен хороший отладчик, то мне стало обидно, что я так долго не применял этот инструмент. Да, век живи, век учись. У вас есть прекрасная возможность избежать моих ошибок. Итак, сегодня вы узнаете, что может сделать для вас отладчик.

Отладчик IDE предоставляет в ваше распоряжение ряд возможностей и инструментов, которые помогут вам при отладке программ. В этой главе мы рассмотрим следующее:

- ☐ Пункты меню отладчика
- ☐ Использование контрольных точек
- ☐ Просмотр переменных в списке объектов наблюдения (Watch List)
- ☐ Инспектирование объектов с помощью инспектора отладки (Debug Inspector)
- ☐ Другие инструменты отладки
- ☐ Пошаговое выполнение программы
- ☐ Методику отладки

Для чего нужен отладчик?

Краткий ответ — отладчик помогает вам находить ошибки в своей программе. Но процесс отладки служит не только для поиска и устранения ошибок. Отладчик является также инструментом разработки. Несмотря на важность отладки, многие программисты не тратят время на изучение всех возможностей отладчика IDE. В результате они расплачиваются своим временем и деньгами, не говоря уже о переживаниях из-за ошибок, которые не так легко обнаружить.

Сеанс отладки начинается с запуска программы под управлением отладчика. Когда вы щелкаете на кнопке Run оперативной панели, ваша программа автоматически запускается с использованием отладчика. Вы можете также выбрать в главном меню пункт Run | Run или нажать F9 на клавиатуре.

Отладочные пункты меню

Прежде, чем детально изучать отладчик, давайте коротко рассмотрим пункты меню, которые к нему относятся. Некоторые из них находятся в меню Run главного меню, другие — в контекстном меню редактора кода. Большинство этих пунктов будет подробно обсуждаться по мере вашей работы над этой главой, поэтому здесь я только перечислю их, чтобы вы просто с ними познакомились. В таблице 11.1 приведены пункты контекстного меню редактора кода, которые относятся к отладчику, и их описание.

Таблица 11.1. Пункты контекстного меню редактора кода, относящиеся к отладке

Пункт	Сочетание клавиш	Описание
Toggle Breakpoint	F5	Установка или удаление контрольной точки в текущей строке редактора кода.
Run to Cursor	нет	Запуск программы (если необходимо) и ее выполнение до той строки в окне редактора кода, которая содержит курсор.
Inspect	Alt+F5	Вызов инспектора отладки для объекта, на который указывает курсор редактора кода.
Go To Address	нет	Задание определенного адреса в программе, с которого возобновится ее выполнение.
Evaluate/Modify	нет	Просмотр и/или модификация значений переменных во время выполнения программы.
Add Watch at Cursor	Ctrl+F5	Добавление в список объектов наблюдения переменной, на которую указывает курсор.

Пункт Run главного меню имеет несколько подпунктов,которые относятся к отладчику. Они позволяют запустить программу под управлением отладчика, завершить программу, выполняющуюся под управлением отладчика, указать параметры командной строки для вашей программы и т.д. Некоторые из этих пунктов продублированы в контекстном меню редактора кода. В таблице 11.2 перечислены пункты меню Run, управляющие процессом отладки.

Таблица 11.2. Отладочные пункты в меню Run

Пункт	Сочетание клавиш	Описание
Run	F9	Компиляция программы (если необходимо) и ее выполнение под управлением отладчика IDE. То же, что и кнопка Run оперативной панели.
Parameters	нет	Ввод параметров командной строки для вашей программы.
Step Over	F8	Выполнение текущей строки исходного кода с остановкой на следующей строке.
Trace Into	F7	Трассировка функции, вызываемой в текущей точке выполнения.
Trace to Next Source Line	Shift+F7	Перемещение точки выполнения на следующую строку исходного кода.
Run to Cursor	F4	Запуск программы с остановкой на текущей строке исходного кода.
Show Execution Point	нет	Отображение точки выполнения программы в редакторе кода. При необходимости прокрутка текста в окне исходного кода. Работает только в том случае, если выполнение программы приостановлено.
Program Pause	нет	Приостановка выполнения программы, как только точка выполнения входит в исходный код программы.
Program Reset	Ctrl+F2	Прекращение выполнения программы и возврат в IDE C++Builder.
Inspect	нет	Отображение диалогового окна Inspect, где вы можете ввести имя инспектируемого объекта.
Evaluate/Modify	Ctrl+F7	Отображение диалогового окна Evaluate/Modify.
Add Watch	Ctrl+F5	Отображение диалогового окна Watch Properties.
Add Breakpoint	нет	Отображение диалогового окна Edit Breakpoint для добавления контрольной точки.

При отладке своих программ вы будете очень часто использовать эти пункты меню. Вам следует также привыкнуть к использованию сочетаний клавиш для выполнения различных отладочных операций.

Давайте теперь рассмотрим контрольные точки и их применение при отладке программ.

Контрольные точки

Когда вы запускаете свою программу в IDE C++Builder, она выполняется с максимальным быстродействием и останавливается только в тех местах, где вы установили контрольные точки.

Контрольная точка (breakpoint) — это маркер, который указывает отладчику приостановить выполнение программы по достижении данной точки.

Установка и удаление контрольных точек

Чтобы установить контрольную точку, щелкните в окне редактора слева от той строки, на которой вы хотите приостановить выполнение программы. В этом месте появится значок остановки (Stop), а вся строка будет выделена красным цветом. Чтобы удалить контрольную точку, щелкните на значке остановки. Для установки или удаления контрольной точки вы можете также нажать F5 или выбрать пункт Toggle Breakpoint в контекстном меню редактора кода.

Контрольная точка может быть установлена только в той строке, которая генерирует исполняемый код. Контрольные точки не действуют, если они установлены на пустых строках, строках комментариев или объявлений. Вам не запрещается устанавливать контрольные точки в строках перечисленных типов, но отладчик предупредит вас о том, что вы установили контрольную точку в строке, которая не содержит исполняемого кода. Например, установка контрольной точки в строках, приведенных ниже, вызовет предупреждение:

```
// это комментарий с последующей пустой строкой

int x;  // объявление
```

Контрольная точка может быть установлена на операторе return или на закрывающей фигурной скобке функции.

Если вы установили контрольную точку в недопустимой строке, отладчик предупредит вас о том, что эта точка недействительна, но предупреждение не будет выведено до тех пор, пока вы не попытаетесь выполнить программу. На рис. 11.1 показано диалоговое окно с предупреждением, которое выводится при обнаружении отладчиком недействительной контрольной точки.

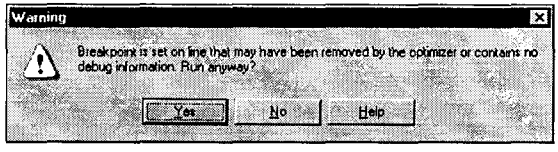


Рис. 11.1
Предупреждающее сообщение для недействительной контрольной точки

Если вы щелкнете на кнопке Yes в диалоговом окне предупреждения, программа начнет выполняться, а все недействительные контрольные точки будут игнорированы. Если вы щелкнете на кнопке No, то вернетесь в редактор кода, где сможете удалить недействительную контрольную точку. Все недействительные контрольные точки будут выделены зеленым цветом, а значок остановки в пробельном поле будет иметь серый цвет.

Программа под управлением отладчика ведет себя так же, как и при обычном запуске, но до тех пор, пока не встретится контрольная точка. В этом случае на переднем плане появляется IDE, а строка исходного кода, содержащая контрольную точку, выделяется. Если вы используете заданные по умолчанию цвета, то строка, на которой программа остановилась, выде-

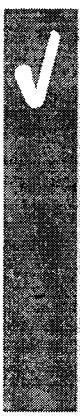
ляется красным цветом, поскольку по умолчанию красный цвет указывает на строку, содержащую контрольную точку.



Точка выполнения (execution point) означает строку исходного кода, которая будет выполнена следующей.

По мере того, как вы продвигаетесь по программе, точка выполнения выделяется синим цветом, а в пробельном поле окна редактора выводится черный символ в виде стрелки-указателя. Нужно понимать, что строка, выделенная синим, еще не была выполнена, но будет выполнена, когда программа продолжит работу.

Как только программа остановилась в контрольной точке, вы можете просматривать переменные, стек вызовов, символы или перемещаться по исходному тексту. Просмотрев интересующие вас переменные и объекты, вы можете продолжить нормальное выполнение программы, щелкнув на кнопке Run. Ваше приложение снова начнет выполняться в обычном режиме, пока не будет достигнута следующая контрольная точка.



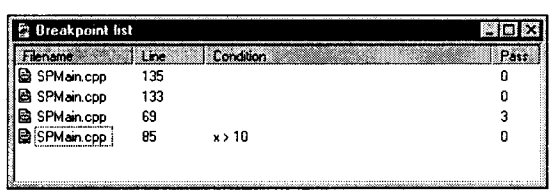
Обычно ошибки в коде программы обнаруживаются после того, как она остановилась в контрольной точке. Если вы измените исходный код во время сеанса отладки, а затем выберете Run, чтобы продолжить выполнение программы, IDE выведет окно сообщения с запросом, хотите ли вы построить программу заново. Если вы выберете Yes, то текущий процесс завершится, после чего исходный код будет перекомпилирован и программа запущена снова.

Проблема с таким подходом к отладке состоит в том, что ваша программа не имеет возможности нормально завершить работу, и использующиеся в настоящий момент ресурсы могут не выгрузиться из памяти, что приведет к ее утечке. Несмотря на то, что Windows 95 и Windows NT отслеживают утечку ресурсов лучше, чем 16-разрядная Windows, предпочтительнее все же сначала завершить программу нормальным образом, а затем выполнить компиляцию исправленного кода.

Окно списка контрольных точек

IDE C++Builder отслеживает установленные вами контрольные точки в окне списка контрольных точек (Breakpoint list). Чтобы просмотреть этот список, выберите в главном меню пункт View | Breakpoints. Появится окно списка контрольных точек, показанное на рис. 11.2.

Рис. 11.2
Окно Breakpoint list



Окно списка контрольных точек имеет четыре столбца. Первый столбец, Filename, показывает имя файла исходного кода, в котором установлена контрольная точка. Второй столбец, Line, показывает номер строки, в которой установлена контрольная точка. Столбец Condition показывает условия останова, которые были установлены для данной контрольной точки, а в столбце

Pass приведено условие на число проходов. (Условия останова и условие на число проходов рассматриваются позже, в разделе «Условные контрольные точки».) Вы можете изменять ширину столбцов, перетаскивая разделительные линии в заголовке столбцов.



Столбец Pass показывает не текущее число проходов контрольной точки, а только условие на число проходов, которое вы установили для данной точки.

Окно Breakpoint list имеет два контекстных меню. В таблице 11.3 перечислены пункты контекстного меню, которое вы увидите, щелкнув правой кнопкой мыши в области столбца Filename. Я буду ссылаться на это меню, как на *главное контекстное меню* данного окна.

Таблица 11.3. Главное контекстное меню окна Breakpoint list

Пункт	Описание
Enable	Включение или выключение контрольной точки. Когда контрольная точка выключена, ее символ в окне списка будет иметь серый цвет. Символ контрольной точки в окне исходного кода также становится серым, а соответствующая строка выделяется зеленым.
Delete	Удаление контрольной точки.
View Source	Прокрутка исходного текста в редакторе кода, чтобы показать строку, содержащую контрольную точку. (Переключения в окно редактора при этом не происходит.)
Edit Source	Установка курсора редактора в строку исходного текста, содержащую контрольную точку, и переключение в окно редактора кода.
Properties	Отображение диалогового окна Edit breakpoint.



Для быстрого редактирования строки исходного кода, в которой установлена контрольная точка, дважды щелкните на ней в столбце Filename окна Breakpoint list. Это эквивалентно выбору Edit Source (Редактировать исходный код) в контекстном меню списка контрольных точек.

Вторичное контекстное меню может быть выведено щелчком правой кнопки мыши в любой части списка контрольных точек, за исключением столбца Filename. Это меню содержит пункты Add (Добавить), Delete All (Удалить все), Disable All (Выключить все) и Enable All (Включить все). Назначение этих пунктов понятно из названий, так что я не стану утруждать себя комментариями.

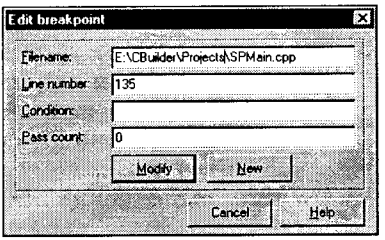


По моему мнению, пункт Add контекстного меню не очень полезен, поскольку намного проще установить контрольную точку в редакторе кода, чем добавлять ее посредством команды Add в окне списка контрольных точек.

В любой момент контрольную точку можно выключить или включить. Вы можете запретить останов в контрольной точке, если вам нужно, чтобы некоторое время программа выполнялась в нормальном режиме; позже вы можете включить эту точку без необходимости вновь ее устанавливать. Выключенные контрольные точки просто игнорируются отладчиком.

Если вам нужно модифицировать контрольную точку, вы можете выбрать пункт Properties в главном контекстном меню списка контрольных точек. При этом появится диалоговое окно Edit breakpoint (Редактировать контрольную точку). (См. рис. 11.3.)

Рис. 11.3
Диалоговое окно Edit breakpoint



Основная причина для модификации контрольной точки состоит в том, чтобы добавить к ней условие. (Условные контрольные точки рассматриваются в разделе «Условные контрольные точки».) Кнопка New в диалоговом окне Edit breakpoint работает довольно любопытным образом. Если вы щелкнете на этой кнопке, контрольная точка будет установлена в той строке, где в настоящий момент находится курсор редактора кода. Это не очень полезная возможность, так что вы с успехом можете игнорировать кнопку New.

Чтобы удалить контрольную точку, вы можете выбрать ее в списке контрольных точек, а затем нажать клавишу Delete на клавиатуре. Чтобы удалить все контрольные точки, щелкните правой кнопкой мыши для вывода вторичного контекстного меню, и выберите в нем пункт Delete All.

Теперь рассмотрим два типа контрольных точек.

Простые контрольные точки

Контрольные точки могут быть простыми или условными. В *простой контрольной точке (simple breakpoint)* выполнение программы приостанавливается. Когда вы первоначально устанавливаете контрольную точку, она по умолчанию относится к простым точкам. Простые контрольные точки действительно не требуют каких-либо дополнительных разъяснений. Когда отладчик встречает такую точку, он приостанавливает выполнение программы и ждет ваших действий. Говорить об этом еще раз нет необходимости.

Условные контрольные точки

В *условной контрольной точке (conditional breakpoint)* программа останавливается только при выполнении заданных условий. Чтобы создать условную контрольную точку, сначала просто установите контрольную точку в редакторе кода. Затем выберите в главном меню пункт View | Breakpoints, чтобы вывести диалоговое окно Breakpoint list. Щелкните правой кнопкой мыши на точке, для которой вы хотите задать условия, и выберите пункт Properties в контекстном меню. Когда будет выведено диалоговое окно Edit breakpoint, задайте требуемые условия.

Условные контрольные точки имеют две разновидности. Первый тип — *контрольная точка с условным выражением (conditional expression breakpoint)*. Введите условное выражение в поле Condition диалогового окна Edit breakpoint (см. рис. 11.3). Во время выполнения программы это условное

выражение будет вычисляться каждый раз при достижении данной контрольной точки. Если значение условного выражения равно true, программа останавливается. Если значение условного выражения равно false, то контрольная точка игнорируется. Для примера вернемся к последней контрольной точке в окне Breakpoint list, показанном на рис. 11.2. Эта контрольная точка содержит условное выражение x > 10. Если в какой-либо момент при выполнении программы значение x станет больше 10, программа остановится в данной точке. Если значение x никогда не превысит 10, выполнение программы остановлено не будет.

Другой тип условной контрольной точки — *контрольная точка с условием на число проходов (pass count breakpoint)*. Для контрольной точки такого типа выполнение программы будет приостановлено только после того, как эта точка будет пройдена заданное число раз. Чтобы определить контрольную точку с условием на число проходов, отредактируйте точку, введя значение в поле Pass count (Число проходов) в диалоговом окне Edit breakpoint. На рис. 11.2 показана контрольная точка, для которой число проходов установлено в 3. Программа остановится в этой точке, когда встретит ее в третий раз.



Счет проходов начинается с 1, а не с 0. Как следует из предыдущего примера, значение 3 для числа проходов означает, что контрольная точка будет действительна только на третий раз, т.е. когда программа в третий раз встретит эту точку.

Используйте контрольные точки с условием на число проходов, если вам нужно, чтобы программа прошла через контрольную точку определенное число раз до начала проверки значений переменных, выполнения части кода в пошаговом режиме или других действий.



Условные контрольные точки замедляют нормальное выполнение программы, поскольку каждый раз требуют дополнительного времени для вычисления заданных условий. Если ваша программа во время отладки работает замедленно, проверьте список контрольных точек и посмотрите, нет ли в нем условных точек, о которых вы могли забыть.



Тот факт, что условные контрольные точки замедляют выполнение программы, время от времени может работать на вас. Если у вас есть процесс, который вы хотите увидеть в замедленном темпе, установите одну или более условных точек в нужной части кода. Задайте условия таким образом, чтобы они никогда не выполнялись, и ваша программа будет работать медленно, но без остановок.

Команда Run to Cursor

Существует еще одна команда отладки, которая заслуживает внимания. Команда Run to Cursor (Выполнить до курсора), которую можно найти в меню Run главного меню и в контекстном меню редактора кода, позволяет выполнять программу до тех пор, пока она не дойдет до строки исходного кода, содержащей курсор редактирования. В этом месте программа останавливается, как если бы здесь была установлена контрольная точка.

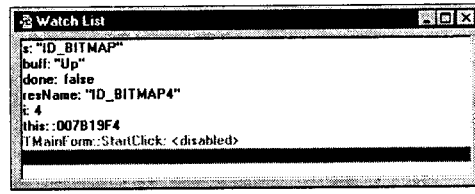
Команда Run to Cursor играет роль временной контрольной точки. Вы можете использовать именно эту команду, а не устанавливать контрольную точку в строке, которую хотите проверить. Все, что от вас требуется — поместить курсор в нужную строку и выбрать пункт Run to Cursor (или нажать F4). Отладчик будет вести себя так же, как и при установке в этой строке контрольной точки. Преимуществом является то, что вам не нужно удалять контрольную точку после отладки данной части кода.

Наблюдение за переменными

Итак, что же вы делаете после остановки программы в контрольной точке? Обычно вы останавливаетесь здесь для того, чтобы проверить значение одной или более переменных. Вам может потребоваться убедиться в том, что конкретная переменная имеет то значение, которое вы предполагаете, или вы вообще можете не знать значения данной переменной и хотите его выяснить. Список объектов наблюдения (Watch List) позволяет вам это сделать.

Список объектов наблюдения имеет единственную функцию: он позволяет проверять значения переменных. Эта простая, но важная возможность часто упускается из виду, поскольку многие программисты не тратят время на полное изучение отладчика. Вы можете добавить в список объектов наблюдения столько переменных, сколько вам требуется. На рис. 11.4 показано окно Watch List во время сеанса отладки.

Рис. 11.4
Список объектов наблюдения во время сеанса отладки



В списке объектов наблюдения выводится имя переменной вместе с ее значением. Способ вывода значения зависит от типа переменной и текущих установок просмотра для данного элемента списка.

Контекстное меню списка объектов наблюдения

Как и любое другое окно C++Builder, окно Watch List имеет свое собственное контекстное меню. (Вы были бы разочарованы, если бы его не было, не так ли?) В таблице 11.4 перечислены пункты этого меню с описаниями.

Таблица 11.4. Контекстное меню списка объектов наблюдения

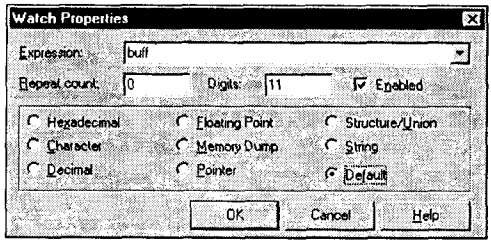
Пункт	Описание
Edit Watch	Редактирование объекта наблюдения в диалоговом окне Watch Properties.
Add Watch	Добавление нового объекта в список.
Enable Watch	Разрешение наблюдения за объектом.
Disable Watch	Запрещение наблюдения за объектом.
Delete Watch	Удаление объекта наблюдения из списка.
Enable All Watches	Разрешение наблюдения за всеми объектами списка.
Disable All Watches	Запрещение наблюдения за всеми объектами списка.
Delete All Watches	Удаление всех объектов из списка.

Оба пункта контекстного меню Edit Watch и Add Watch вызывают диалоговое окно Watch Properties (Свойства объекта наблюдения), поэтому перейдем теперь к его рассмотрению.

Диалоговое окно Watch Properties

Вы обращаетесь к диалоговому окну Watch Properties в тех случаях, когда вам нужно добавить или отредактировать объект наблюдения. На рис. 11.5 показано диалоговое окно Watch Properties в момент редактирования переменной с именем buff.

Рис. 11.5
Диалоговое окно Watch Properties



Поле Expression (Выражение) в верхней части диалогового окна Watch Properties — это то место, где вы вводите имя переменной, которая будет редактироваться или добавляться в список объектов наблюдения. Если вы добавляете объект, выбрав его в окне редактора кода, это поле будет уже заполнено (см. раздел «Добавление переменных в список объектов наблюдения»). Поле Expression является комбинированным списком, поэтому вы можете выбирать объекты наблюдения, использовавшиеся ранее.

Если вы проверяете значение массива, то должны заполнить поле Repeat count (Количество повторений). Предположим, например, что у вас есть массив из 20 целых чисел. Чтобы проверить 10 первых чисел массива, вам нужно ввести его первый элемент в поле Expression (например, array[0]), а затем ввести значение 10 в поле Repeat count. В этом случае в список объектов наблюдения будут выведены первые 10 элементов массива.



Если вы просто добавите имя массива в список объектов наблюдения, то будут выведены все элементы этого массива. Чтобы проверить определенное количество элементов массива, используйте поле Repeat count.

Поле Digits (Цифры) заполняется только при проверке чисел с плавающей точкой. Введите здесь количество значащих цифр, которое будет выводиться в списке объектов наблюдения. При выводе числа с плавающей точкой его младшие разряды не отбрасываются, а округляются. Другое поле в этом диалоговом окне, Enabled, определяет, разрешено ли в настоящее время наблюдение за данным объектом.

Оставшаяся часть диалогового окна Watch Properties содержит различные опции просмотра. Каждый тип данных имеет заданный по умолчанию формат вывода, который устанавливается при выборе опции Default. Эта опция задана по умолчанию. Чтобы вывести данные в другом формате, выберите одну из других опций просмотра. На рис. 11.6 показано окно Watch List, содержащее две переменные, которые выводятся в различных форматах. Переменная buff — это символьный массив, а переменная i — целое число.

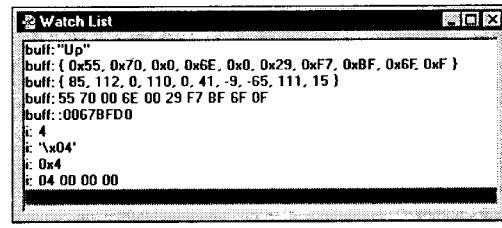


Рис. 11.6
Различные форматы вывода значений переменных

Чтобы модифицировать объект наблюдения, щелкните на нем в списке и выберите в контекстном меню пункт Edit Watch. Появится диалоговое окно Watch Properties, где вы сможете отредактировать этот объект.



Самый быстрый способ отредактировать объект наблюдения — дважды щелкнуть на его имени в списке Watch List.

Разрешение и запрещение наблюдения за объектами

Аналогично контрольным точкам, отдельные объекты в списке Watch List могут быть выключены или включены. Когда наблюдение за объектом запрещено, он отображается серым цветом, а вместо его значения выводится текст <disabled>.

Чтобы запретить наблюдение за объектом, щелкните на его имени в списке и выберите в контекстном меню пункт Disable Watch. Наблюдение разрешается, если вы выберете в контекстном меню пункт Enable Watch.



Имеет смысл запретить наблюдение за объектами, которые не интересуют вас в настоящий момент, но потребуются позже. Наличие большого числа объектов наблюдения в списке Watch List может замедлить выполнение программы во время отладки, поскольку все переменные списка должны модифицироваться каждый раз, когда выполняется очередная строка программы. Чтобы замедлить работу программы, хватит и небольшого количества объектов наблюдения, так что не забывайте удалять или выключать неиспользуемые элементы списка.

Добавление переменных в список объектов наблюдения

Вы можете добавлять переменные в список Watch List одним из нескольких способов. Самый быстрый из них — щелкнуть на имени переменной в окне редактора кода, после чего выбрать в контекстном меню пункт Add Watch at Cursor (Добавить объект наблюдения в позиции курсора) или нажать на клавиатуре Ctrl+F5. После этого появится диалоговое окно Watch Properties, где вы можете либо выбрать опции для этого объекта наблюдения, либо щелкнуть на кнопке OK (или нажать Enter), чтобы оставить все опции в состоянии по умолчанию. Объект наблюдения будет добавлен в список.

Чтобы добавить переменную в список объектов наблюдения, не выполняя ее поиска в исходном файле, выберите в главном меню пункт Run | Add Watch. Когда появится диалоговое окно Watch Properties, введите имя переменной, которую вы хотите добавить в список, и нажмите OK.



Хотя вы и можете добавить экземпляр класса в список объектов наблюдения, я не уверен, что выведенное значение будет полезным. Если добавить в список указатель на объект, то вы сможете определить, является ли значение указателя допустимым, и это само по себе полезно, но вряд ли вы захотите просмотреть детали класса. Для просмотра всех членов-данных класса вы должны использовать инспектор отладки, который мы рассмотрим чуть позже.

Использование списка объектов наблюдения

После достижения контрольной точки, список Watch List отобразит текущие значения всех переменных, которые были в него включены. Если список в данный момент закрыт, вы можете вывести его, выбрав в главном меню пункт View | Watches.

При определенных обстоятельствах рядом с переменной будет выведено сообщение, а не ее значение. Если, например, переменная выходит из области видимости или не найдена, то в списке объектов наблюдения рядом с ее именем появится сообщение Undefined symbol 'x' (Не определен символ 'x'). Пока программа не выполняется или не остановилась в контрольной точке, в списке будет выведено сообщение [process not accessible] (процесс недоступен) для всех объектов наблюдения. Выключенный объект будет сопровождаться сообщением <disabled>. В зависимости от текущего состояния приложения или текущего состояния конкретной переменной могут выводиться и другие сообщения.

Как я уже говорил вчера, иногда вы можете столкнуться с тем, что переменная была оптимизирована и не доступна в списке объектов наблюдения. Это один из незначительных недостатков оптимизирующего компилятора. Если вам нужно просмотреть переменные, которые подлежат оптимизации, либо объявите переменную с модификатором volatile, либо выключите опцию Register Variables во вкладке Compiler диалогового окна Project Options. После отладки удалите модификатор volatile данной переменной.



Список объектов наблюдения может использоваться для быстрого преобразования десятичных значений в шестнадцатеричные и обратно. Чтобы преобразовать шестнадцатеричное число в десятичное, выберите в главном меню пункт Run | Add Watch. Введите шестнадцатеричное число в поле Expression и нажмите OK. В список Watch List будет выведено и шестнадцатеричное число, и его десятичный эквивалент. Чтобы преобразовать десятичное число в шестнадцатеричное, выполните ту же самую процедуру, дополнительно щелкнув радиокнопку Hexadecimal, чтобы изменить формат вывода на шестнадцатеричный. Поскольку поле Expression допускает запись математического выражения, вы также можете использовать Watch List как шестнадцатеричный калькулятор. Вы можете даже смешивать шестнадцатеричные и десятичные значения в одном выражении.

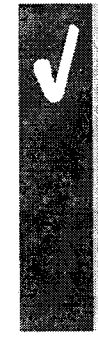
Список объектов наблюдения — это простой, но нужный инструмент для отладки приложений. Чтобы проиллюстрировать процедуру отладки с помощью списка, выполните следующее упражнение.

1. Создайте новое приложение и поместите в форму кнопку. Измените значение свойства Name кнопки на WatchBtn, а свойства Caption — на Watch Test. Измените значение свойства Name формы на DebugMain, а свойства Caption — на любой текст, который вам нравится.

2. Дважды щелкните на кнопке, чтобы вывести обработчик события OnClick в окно редактора кода. Введите следующий код в позиции курсора:

```
String s;
int x = Width;
s = String(x);
int y = Height;
x *= y
s = String(x);
x /= y;
s = String(x);
Width = x;
Height = y;
```

3. Выберите в главном меню пункт Options | Project и щелкните на вкладке C++. Переключите опцию Optimization в состояние None. (См. примечание в разделе «Пошаговое выполнение программы» для объяснения этого действия.)
4. Сохраните проект. Назовите модуль DbgMain, а проект — DebugTst.
5. Установите контрольную точку на второй строке кода, который вы ввели на шаге 2. Запустите программу.
6. Щелкните на кнопке Watch Test. Программа остановится в контрольной точке.
7. Добавьте в список объектов наблюдения переменные s, x и y. (Вместо значений переменных x и y сначала будет выведено сообщение Variable 'x' has been optimized, но не обращайте на это внимание.)
8. Расположите список объектов наблюдения и редактор кода так, чтобы вы могли видеть оба окна.
9. Перейдите в редактор кода и нажмите F8, чтобы выполнить следующую строку. Эта строка будет выполнена, а точка выполнения переместится на следующую строку. Теперь рядом с переменной x показано некоторое значение (возможно, 435).
10. Продолжите пошаговое выполнение программы, нажимая F8. Следите за значениями переменных в окне Watch List.
11. Когда точка выполнения перейдет к последней строке функции, щелкните на кнопке Run оперативной панели, чтобы продолжить выполнение программы.
12. Нажимайте кнопку Watch Test столько раз, сколько вам нужно, чтобы освоиться с работой списка Watch List. На каждом проходе поэкспериментируйте с различными установками для просмотра объектов наблюдения.



Код этого примера получает значения свойств Width и Height формы, выполняет некоторые вычисления, а затем возвращает свойствам Width и Height первоначальные значения, которые они имели в начале программы. В результате ничего не меняется, но существует весомая причина для присвоения значений свойствам Width и Height в конце функции.

Если в программе ничего не делать с переменными x и y, то вы не сможете проверить их значения, поскольку компилятор оптимизирует их, и они станут недоступны для наблюдения. По существу, компилятор может заглянуть вперед, чтобы найти переменные, которые никогда не используются, и тем или иным способом отбросить их. Включение переменных в работу в конце функции предотвращает их отбрасывание при оптимизации программы.

Я повторил все это несколько раз, чтобы быть уверенным в том, что вы понимаете основы работы оптимизирующего компилятора. Когда вы начнете отлаживать свои приложения, это знание поможет вам избежать некоторого недоумения, если в списке Watch List вы начнете получать сообщения типа Variable 'x' has been optimized and is not available (Переменная 'x' была оптимизирована и недоступна).

Инспектор отладки

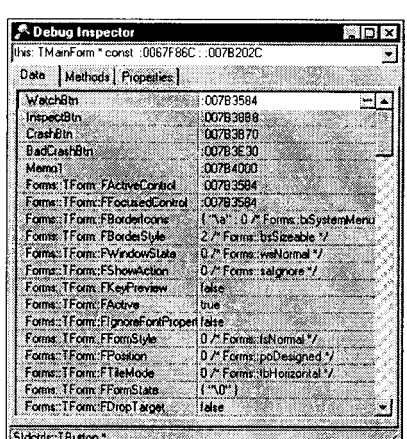
Если не усложнять, то инспектор отладки (Debug Inspector) позволяет просмотреть члены-данные объектов, таких, как классы и компоненты (в конечном счете, компоненты — это просто классы). Вы можете также проверить данные простых типов (целые числа, символьные массивы и т.д.), но лучше всего это делать в списке объектов наблюдения. Инспектор отладки наиболее полезен при исследовании классов и структур.

Вы можете пользоваться инспектором отладки только в том случае, когда выполнение программы приостановлено под управлением отладчика.

Чтобы проверить состояние объекта, щелкните на его имени в исходном файле и выберите пункт Inspect в контекстном меню редактора кода (или нажмите Alt+F5). Вы можете также выбрать пункт Run | Inspect в главном меню.

Когда вы выбираете пункт Inspect в контекстном меню или используете эквивалентное сочетание клавиш Alt+F5, инспектор отладки автоматически покажет состояние объекта под курсором. Если вы выбрали Run | Inspect в главном меню, то сначала выводится диалоговое окно, где вы должны ввести имя объекта, состояние которого хотите проверить. Введите имя переменной и нажмите OK. Будет показан инспектор отладки с выведенными сведениями об объекте.

Рис. 11.7
Представление класса формы в инспекторе отладки



Окно инспектора отладки содержит подробную информацию об указанном объекте. Если объектом является простой тип данных, то в окно инспектора отладки будет выведено текущее значение (в десятичном и шестнадцатеричном форматах для числовых типов), а в строку состояния в нижней части окна — тип переменной. Например, если вы проверяете целую переменную, в окно будет выведено ее значение, а строка состояния покажет тип — int. В верхней части инспектора отладки находится комбинированный список, который изначально содержит описание инспектируемого объекта.

Если вы работаете с классом, то инспектор отладки будет выглядеть примерно так, как показано на рис. 11.7.

Чтобы лучше понять работу инспектора отладки, выполните следующее.

1. Загрузите программу DebugTst, которую вы создали ранее (если она уже не загружена).
2. Установите контрольную точку в каком-либо месте функции WatchBtnClick().
3. Запустите программу и щелкните на кнопке Watch. Отладчик остановит программу в той точке, которую вы установили.
4. Выберите в главном меню пункт Run | Inspect. Появится диалоговое окно Inspect.
5. Введите this в поле Expression и нажмите OK.
6. Появится инспектор отладки.



Вы можете использовать указатель this только внутри функции-члена класса. Если вы установите контрольную точку в обычной функции и попытаетесь ввести this, то получите сообщение о том, что this — недопустимый символ.

Как вы можете видеть, при просмотре классов окно инспектора отладки содержит три вкладки. Вкладка Data показывает все члены-данные класса. Список членов-данных имеет иерархическую структуру. Первыми перечислены члены, принадлежащие непосредственно данному классу. Следующая группа — члены-данные, которые принадлежат классу непосредственного предка этого класса, в данном случае — TForm. Если вы прокрутите список инспектора отладки, то увидите, что после членов-данных класса TForm следуют члены-данные класса TScrollingWinControl (непосредственный предок TForm) и т.д.

Продвигаясь с помощью клавиш курсора по списку членов-данных, вы можете сразу сказать, к какому типу относится каждый из них (смотрите на строку состояния). Для получения более подробной информации о конкретном члене, дважды щелкните в столбце значений на строке, в которой находится интересующий вас член. Откроется второе окно инспектора отладки, где будет выведена более детальная информация о выбранном элементе. На рабочем столе может одновременно находиться несколько открытых окон инспектора отладки.



Инспектор отладки загружает большой объем информации, так что прокрутка его списка на некоторых системах может быть медленной.

Вкладка Methods инспектора отладки отображает методы класса. Как и для членов-данных, список методов имеет иерархическую структуру. Первыми показаны собственные методы класса, после которых следуют методы классов-предков. В некоторых случаях вкладка Methods отсутствует (например, при проверке простых типов данных).

Вкладка Properties инспектора отладки показывает свойства инспектируемого класса. Если вы инспектируете класс, унаследованный из VCL, то проверка его свойств в инспекторе отладки идет очень медленно. В большинстве случаев вы можете выполнить ту же задачу другим способом, просмотрев член класса, связанный с конкретным свойством, во вкладке Data. Проверка членов-данных происходит намного быстрее, чем проверка свойств.



Вкладки Methods и Properties инспектора отладки доступны только в том случае, когда вы инспектируете класс. Для простых типов данных выводится только вкладка Data.

Контекстное меню инспектора отладки имеет несколько пунктов, которые позволяют работать с отдельными переменными. Например, вместо того, чтобы открывать новое окно инспектора отладки для каждого объекта, вы можете выбрать в контекстном меню пункт Descend, чтобы заменить текущий объект в окне инспектора отладки объектом, на который указывает курсор. Этот способ имеет дополнительное преимущество: IDE будет сохранять хронологический список объектов, которые вы инспектируете. Чтобы вернуться к проверявшемуся ранее объекту, достаточно выбрать его из комбинированного списка, расположенного в верхней части окна инспектора отладки. Выбор одного из объектов хронологического списка снова выведет этот объект в окно инспектора отладки.

Пункт Change в контекстном меню инспектора отладки позволит вам изменить значение переменной. Делать это нужно осторожно, поскольку ошибочное изменение не той переменной или присваивание недопустимого значения может привести к краху программы. Пункт Inspect в контекстном меню позволяет открыть второе окно инспектора отладки, содержащее элемент, на который указывает курсор. Пункт New Expression позволяет ввести новое выражение, чтобы проверить его в инспекторе отладки.

Пункт Show Inherited в контекстном меню инспектора отладки является переключателем, который определяет объем информации, выводимой инспектором отладки. Когда опция Show Inherited включена, инспектор отладки показывает все члены-данные, методы и свойства инспектируемого класса, а также члены-данные, методы и свойства всех предков данного класса. Когда опция Show Inherited выключена, показываются члены-данные, методы и свойства только самого класса. Выключение этой опции может значительно ускорить работу инспектора отладки, поскольку ему не придется выводить такой большой объем информации.



Если вы забыли, к какому типу относится конкретный член класса, щелкните на нем, когда программа остановилась в контрольной точке, и нажмите Alt+F5, чтобы вывести окно инспектора отладки. Строка состояния в нижней части окна сообщит вам тип указанной переменной. Это также работает для системных #define типа HINSTANCE, HWND, DWORD и т.д.

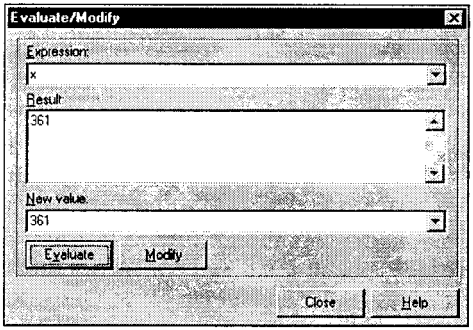
Другие инструменты отладки

C++Builder имеет некоторые дополнительные инструменты отладки, которые помогут вам отслеживать ошибки. Некоторые из этих инструментов по своей природе являются достаточно сложными средствами отладки, и хотя они используются не так часто, как другие, в руках опытного программиста могут представлять значительную силу.

Evaluate/Modify

Диалоговое окно Evaluate/Modify (Вычислить/Модифицировать) позволяет просматривать и изменять текущее значение переменной. В этом диалоговом окне вы можете проверить различные результаты изменения определенной переменной. Это позволяет вам поиграть со своей программой в игру what-if («а что, если») во время ее выполнения. На рис. 11.8 показано диалоговое окно Evaluate/Modify, в котором инспектируется целая переменная с именем x.

Рис. 11.8
Диалоговое окно Evaluate/Modify



Диалоговое окно Evaluate/Modify работает аналогично списку объектов наблюдения или инспектору отладки. Если вы щелкнете на переменной в исходном тексте и выберете пункт Evaluate/Modify в контекстном меню редактора кода, то будет выведено значение этой переменной. Если вам нужна переменная, которая в настоящий момент не показана в исходном тексте, то можете выбрать в главном меню пункт Run | Evaluate/Modify, а затем набрать имя переменной, значение которой нужно вычислить.

В поле Evaluate вводится имя переменной или выражения, значение которого вам нужно вычислить. Когда вы щелкаете кнопку Evaluate (или нажимаете Enter), значение выражения вычисляется, а результат выводится в поле Result.



Диалоговое окно Evaluate/Modify может использоваться как быстрый калькулятор. Вы можете вводить шестнадцатеричные или десятичные числа (или их комбинации) в математической формуле и получать результат вычислений. Например, если вы введете в поле Evaluate

```
0x400 - 256
```

и нажмете клавишу Enter, в поле Result будет выведен результат вычислений — число 768.

Вы можете вводить в поле Evaluate и логические выражения. Например, если вы ввели

```
20 * 20 == 400
```

то в поле Result будет выведено true. Единственная проблема состоит в том, что для функционирования диалогового окна Evaluate/ Modify программа должна быть остановлена в контрольной точке.

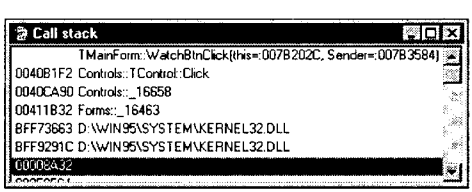
Если вы хотите присвоить переменной другое значение, введите его в поле New Value и нажмите кнопку Modify. Значение переменной будет изменено на вновь введенное. Когда вы нажмете кнопку Run, чтобы перезапустить программу (или продолжить пошаговое выполнение), будет использовано новое значение переменной.

В отличие от списка объектов наблюдения и инспектора отладки, диалоговое окно Evaluate/Modify не обновляется автоматически при пошаговом выполнении программы. Если ваш код изменяет значение переменной, содержащейся в окне Evaluate/Modify, то нужно щелкнуть кнопку Evaluate, чтобы увидеть результат. Обычно при работе с этим диалоговым окном вы вычисляете значение переменной или выражения и сразу закрываете окно.

Просмотр стека вызовов

Во время выполнения своей программы вы можете просматривать стек вызовов (Call Stack), чтобы получить информацию о любых функциях, которые вызывала программа. Выберите в главном меню пункт View | Call Stack, чтобы вывести окно Call stack. Это окно показывает список функций, вызванных вашей программой, в том порядке, в котором они вызывались. Последние вызванные функции будут находиться в верхней части окна. Здесь будут перечислены функции вашей программы, методы VCL и функции, содержащиеся в DLL Windows. На рис. 11.9 показан стек вызовов в том виде, который он имеет после выполнения программы DebugTst, созданной вами ранее в этой главе.

Рис. 11.9
Окно Call stack



В данном случае первой функцией списка является TMainForm::WatchBtnClick(). Далее вы видите вызовы некоторых функций VCL и пару вызовов функций (без имени) из библиотеки Windows KERNEL32.DLL. Не забудьте, что функции перечислены в обратном порядке — функция, выполненная последней, появляется в верхней части списка вызовов.

Если функция принадлежит вашей программе, то дважды щелкнув на ее имени в окне Call stack, вы окажетесь в строке исходного кода, содержащей эту функцию. Если функция содержится в DLL Windows, окно Call stack

будет содержать только адрес и имя DLL. Двойной щелчок на вызове функции, которая содержится в DLL, выведет окно CPU View (это окно рассматривается в следующем разделе).



Если вы компоновали программу с отладочной версией VCL, то двойной щелчок на методе VCL в окне Call stack выведет исходный код этого метода. В противном случае будет отображено окно CPU View.

Просмотр стека вызовов наиболее полезен после системной ошибки Access Violation (Нарушение прав доступа). Просматривая стек вызовов, вы можете увидеть, в каком состоянии находилась ваша программа непосредственно перед тем, как произошла ошибка. Это часто бывает первым шагом в поиске причин краха программы.



Если список вызовов содержит бессмысленную информацию, то вполне возможно, что произошло разрушение стека вызовов. Вероятность такой ситуации в 32-разрядной программе достаточно низка, по сравнению с 16-разрядной, но все же это не исключено.

CPU View

Окно CPU View (Просмотр кода процессора) позволяет просматривать вашу программу на уровне ассемблера. Используя этот инструмент, вы можете выполнять инструкции ассемблера в пошаговом режиме. Кроме того, вы можете выполнить программу до определенной инструкции ассемблера, точно так же, как и до определенной строки исходного кода с обычным отладчиком. Окно CPU View имеет пять подокон: дизассемблера (disassembly), регистров (register), флажков (flags), исходного стека (raw stack) и дампа (dump). Каждое подокно имеет свое контекстное меню, которое содержит все функции, необходимые для работы с данным подокном. Для эффективного применения инструмента CPU View требуется знание ассемблера. Чтобы вывести окно CPU View, выберите в главном меню пункт View | CPU.

Команда Go to Address

Команда Go to Address (Перейти на адрес) также относится к средствам отладки для опытных программистов. Когда в вашей программе происходит сбой, Windows выводит сообщение об ошибке, показывающее адрес, по которому произошел этот сбой. Вы можете использовать команду Go to Address, чтобы попытаться определить то место в вашей программе, где произошел сбой. При получении от Windows сообщения об ошибке Access Violation (Нарушение прав доступа), вы увидите диалоговое окно, подобное тому, которое показано на рис. 11.10.

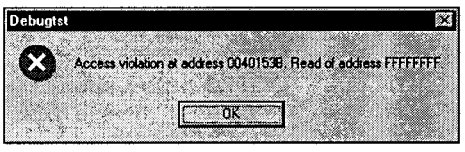


Рис. 11.10
Окно сообщения Windows о нарушении прав доступа

Когда вы видите это сообщение об ошибке, запишите адрес, по которому произошел сбой, а затем выберите в главном меню пункт Search | Go to Address, чтобы вывести диалоговое окно Go to Address. Введите в поле Address тот адрес, который вы только что записали. Когда вы нажмете OK, отладчик попытается найти строку исходного кода, где произошла ошибка. Если ошибка произошла в вашем коде, то курсор будет помещен в ту строку, которая явилась источником ошибки. Если ошибка произошла вне вашего кода, вы увидите окно с сообщением о том, что адрес не найден. Как я уже сказал, для использования команды Go to Address необходим определенный опыт и возможно, что вы ни разу не воспользуетесь этим инструментом.

Пошаговое выполнение программы

Пошаговое выполнение программы — это одна из базовых операций отладки. Тем не менее, мы достаточно подробно рассмотрим ее здесь. Время от времени нужно возвращаться к основам. При этом вы можете узнать то, что раньше ускользнуло от вашего внимания.

После останова в контрольной точке вы можете предпринять различные действия для определения того, что происходит с вашей программой. Вы можете добавить переменные в список объектов наблюдения, проверить состояние объектов в инспекторе отладки или просмотреть стек вызовов. Вы можете также использовать режим пошагового выполнения, чтобы увидеть, как изменяются ваши переменные и объекты после выполнения каждой строки программы. По мере продвижения через код строка, которая будет выполнена следующей, выделяется синим. Если у вас открыты список объектов наблюдения и инспектор отладки, вы сможете немедленно увидеть изменения переменных или объектов.



По умолчанию, опция оптимизации компилятора установлена в состояние Optimize for Speed (Оптимизировать для наибольшего быстродействия). (Опции оптимизации рассматривались вчера.) Когда компилятор выполняет оптимизацию, он перестраивает ваш исходный код, используя некий таинственный способ, о сущности которого простые смертные могут только догадываться. (При этом ваш исходный код остается без изменений, но результирующий ассемблерный код может не соответствовать в точности тексту вашей программы.) В результате программа выполняется быстрее, и это, безусловно, хорошо.

Однако за все приходится платить. Ранее я говорил, что переменные могут быть оптимизированы компилятором, после чего они становятся недоступными для просмотра. Это один из недостатков оптимизации.

Другой интересный побочный эффект оптимизации — реорганизация вашего кода, о чем я уже сказал. В результате этой реорганизации, при пошаговом выполнении кода точка выполнения может не переходить последовательно от строки к строке, как вы могли бы ожидать. Вместо этого она может начать «скакать» по коду, и даже возвращаться к одной и той же строке несколько раз.

Все это совершенно нормально, но может запутать, когда вы только учитесь использовать отладчик. Если вы предпочитаете видеть последовательно передвигающуюся точку выполнения, выключите оптимизацию при отладке. При окончательном построении программы вы можете снова ее включить. Не забудьте о том, что при изменении параметров оптимизации вы должны перестроить все модули, выбрав команду Build All.

Отладчик IDE имеет две основные команды для пошагового выполнения кода: Step Over (буквально — «Перешагнуть через...») и Trace Into (Трассировка). Step Over означает выполнение очередной строки исходного кода и приостановку выполнения на следующей строке. Step Over — это не совсем правильное название для данной команды. Из него можно сделать вывод, что вы можете перешагнуть через исходную строку без ее выполнения. Однако это не так. Step Over означает, что будет выполнена текущая строка, а любые функции, которые вызываются в данной строке, будут выполнены при полном быстродействии. Предположим, например, что вы установили контрольную точку в строке, где вызывается другая функция вашей программы. Когда вы даете отладчику команду перешагнуть через эту функцию, отладчик выполняет функцию и останавливается на следующей строке. (Сравните это с тем, как работает команда Trace Into, о которой вы узнаете буквально через минуту, и все станет гораздо понятнее.) Чтобы выполнить команду Step Over, вы можете либо нажать F8, либо выбрать в главном меню пункт Run | Step Over.



При пошаговом выполнении программы редактор кода автоматически загружает и отображает необходимый модуль (если он еще не открыт).

Команда Trace Into позволяет вам войти в режим трассировки любой функции, которая встречается при пошаговом выполнении программы. В отличие от команды Step Over, которая выполняет функцию и возвращается к следующей строке, команда Trace Into помещает точку выполнения в первую строку исходного кода вызванной функции. После этого вы можете выполнить функцию строка за строкой, используя команды Step Over или Trace Into. Эквивалентная клавиша для команды Trace Into — F7.

После того, как вы просмотрели переменные и выполнили необходимую отладку, вы снова можете запустить программу с полным быстродействием, нажав кнопку Run. Программа будет выполняться в нормальном режиме до следующей контрольной точки.



Если включена опция компоновщика Link Debug Version of VCL (Компоновать отладочную версию VCL), то команда Trace Into, встретив метод VCL, переведет вас к исходному коду этого метода. Внутри исходного кода VCL вы можете просмотреть любые переменные, которые вас интересуют. Если вы включаете эту опцию после построения программы, то должны перестроить все ее модули, выбрав Build All, иначе опция не будет работать. Как я уже говорил ранее, для большинства программистов пошаговое выполнение исходного кода VCL имеет сомнительную пользу.

Другая, менее часто используемая команда отладки — Trace To Next Source Line (Трассировка до следующей строки исходного кода) (Shift+F7 на клавиатуре). Вы тоже, вероятно, не будете активно пользоваться этой командой, пока не освоите как следует отладку и не приобретете опыт программирования в Windows. Некоторые функции API Windows используют *функции обратного вызова* (*callback function*). Это означает, что функция Windows вызывает одну из ваших функций для выполнения некоторого действия. Если точка выполнения находится на вызове функции API Windows, которая использует функцию обратного вызова, то команда Trace To Next

Source Line переведет точку выполнения к первой строке функции обратного вызова. Результат этой команды похож на результат команды Trace Into, но конкретная ситуация, в которой используется Trace To Next Source Line, в целом отличается. Если сказанное не имело для вас особого смысла, не нужно беспокоиться. Это не требуется для понимания того, о чем вы должны сегодня узнать.



Когда вы выполняете функцию в пошаговом режиме, точка выполнения дойдет, в конечном счете, до закрывающей фигурной скобки. Если эта функция возвращает управление Windows, то при нажатии F8 вы выйдете из этой функции и управление вернется к отлаживаемой программе. Поскольку IDE все еще находится в фокусе, отсутствует какой-либо очевидный индикатор того, что программа продолжила работу. Это поведение сначала может несколько озадачить, пока вы не поймете, что случилось. Чтобы снова переключиться на свою программу, просто активизируйте ее так, как вы обычно это делаете с любой другой программой (щелкните на соответствующем символе в панели задач Windows или используйте сочетание Alt+Tab).

Как я уже сказал, пошаговое выполнение кода — это основная методика отладки, которую вы будете постоянно использовать. Из всех сочетаний клавиш, доступных вам в C++Builder, F7 и F8 должны обязательно быть в вашем арсенале.

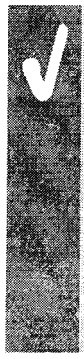
Методика отладки

Я уже коснулся методов отладки, когда мы рассматривали различные возможности отладчика IDE. Здесь мы рассмотрим еще несколько методов, упрощающих отладку ваших программ.

Диагностические макросы: TRACE и WARN

Иногда полезно проследить выполнение своей программы. Например, было бы хорошо иметь регистрационный файл, в который бы заносились определенные события, например, вызов вашей программой некоторых функций. Или вы можете захотеть занести в регистрационный файл значение переменной, чтобы проверить его без остановки программы в контрольной точке. Именно это позволяют сделать диагностические макросы TRACE и WARN. Эти макросы являются удобными средствами отладки, которые, из-за разрозненности документации и недостаточного внимания к данной теме, упускают из виду многие программисты.

Когда вы используете TRACE и WARN в своих программах, сообщения, генерируемые этими макросами, записываются в регистрационный файл с именем OutDbg1.txt. C++Builder автоматически создает этот текстовый файл и отображает его в окне редактора кода, когда встречает оператор TRACE или WARN. Затем вы можете просмотреть регистрационный файл и узнать, что происходило в вашей программе.



Файл OutDbg1.txt имеет пару интересных особенностей. Первая из них состоит в том, что этот файл не рассматривается как часть проекта. Когда вы выбираете пункт Save All, текстовый файл OutDbg1.txt не сохраняется, в отличие от остальных файлов проекта. По той же самой причине, при закрытии этого файла вы не видите запроса на его сохранение. Если вы хотите его сохранить, то должны специально выбрать Save или Save As до закрытия.

Такое поведение может показаться несколько странным, но его смысл станет понятным, если немного подумать. Регистрационный файл почти всегда используется в качестве временного инструмента отладки. Сохранение содержания регистрационного файла требуется не так часто, так что C++Builder просто не отвлекает вас еще одним окном сообщения.

Макрос TRACE просто выводит строку текста в регистрационный файл в редакторе кода. Предположим, например, что ваша функция FormCreate()содержит следующий код:

```
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
  TRACE("Entering FormCreate()");
  // инициализирующий код
  TRACE("Leaving FormCreate()");
}
```

При выполнении этого кода, в файл OutDbg1.txt будут записаны две строки. Вы можете просмотреть сообщения в регистрационном файле в любое время (во время выполнения программы или после ее завершения), просто щелкнув на вкладке OutDbg1.txt в редакторе кода.



Диагностические макросы используют потоки Си++ (помните, мы использовали этот способ в 1-й день, «Первое знакомство»?). Это дает возможность записывать регистрационные сообщения следующим образом:

```
TRACE("Variable x = " << x);
```

Это чрезвычайно облегчает запись самых разнообразных диагностических сообщений.

Макрос WARN аналогичен TRACE за исключением того, что он позволяет выводить сообщения при выполнении заданного условия. В следующем примере сообщение выводится только в том случае, если значение переменной x больше 200:

```
WARN(x > 200, "x = " << x << ": Possible range error");
```

Макрос WARN может уменьшить избыточность в регистрационном файле, отображая сообщения только в тех случаях, когда это необходимо.

Диагностические сообщения, записанные в регистрационный файл, включают имя модуля исходного кода, номер строки исходного кода и заданный текст. Предположим, например, что ваш модуль с именем MyApp.cpp содержит следующий код:

```
int x = 100;
TRACE("x = "  <<  x);
WARN(x == 100, "x is now 100");
```

В этом случае текст, записанный в файл OutDbg1.txt, будет выглядеть следующим образом:

```
Trace MyApp.cpp 18: [Def] x = 100
Warning MyApp.cpp 19: [Def] x is now 100
```

Как вы видите, в регистрационный файл выводится тип сообщения (трассировка или предупреждение), имя модуля, номер строки и текст. Вы можете игнорировать [Def] в регистрационном сообщении, поскольку оно не имеет никакого значения в C++Builder (рассматривайте это как пережиток, сохранившийся от Borland C++).



Если при выполнении операторов TRACE или WARN редактор кода не содержит файла OutDbg1.txt, будет создано окно редактирования OutDbg1.txt, и текст будет выводиться в него. Если файл OutDbg1.txt уже существует в редакторе кода, любое новое диагностическое сообщение будет добавляться в конец существующего файла.

Чтобы использовать диагностические макросы, сначала вы должны определить их как __TRACE и __WARN. Вы можете поместить директивы #define в начале одного из своих исходных файлов, но лучше добавить их на уровне проекта. Чтобы сделать это, выберите в главном меню пункт Options | Project. Когда появится диалоговое окно Project Options, щелкните на вкладке Conditionals/Defines и введите в поле Conditional Defines следующее:

```
__TRACE;__WARN
```

Обратите внимание, что перед обоими определениями стоит двойное подчеркивание, и что они отделяются друг от друга точкой с запятой.

Диагностические макросы объявлены в файле checks.h, поэтому вы также должны добавить в свой исходный код следующую строку:

```
#include <checks.h>
```

Если вы не сделаете этого, то получите сообщение об ошибке компиляции во всех строках, содержащих TRACE или WARN.

Отслеживание GPF

Все программисты Windows при разработке своих приложений сталкиваются с общими нарушениями защиты (General Protection Fault) и общими исключениями защиты (General Protection Exception). Для простоты, я буду ссылаться на то и другое как на GPF.



Термин GPF остался от 16-разрядной среды Windows. Он все еще распространен в 32-разрядном мире программирования Windows, несмотря на то, что 32-разрядная Windows в действительности сообщает о нарушении прав доступа (access violation), а не об общем нарушении защиты.

Отслеживание GPF может быть трудным как для начинающих, так и для опытных программистов в Windows. Обычно по мере приобретения опыта в написании Windows-программ, программисты развивают у себя своего рода шестое чувство причин, вызывающих GPF. В следующих разделах описаны

ситуации, о которых следует знать, пытаясь проследить неуловимую GPF. Это лишь некоторые из наиболее распространенных причин краха программы.

Неинициализированные указатели

Неинициализированный указатель — это указатель, который объявлен в вашей программе, но не содержит адреса какого-либо объекта программы. Неинициализированный указатель будет содержать произвольные данные. В самом лучшем случае он указывает на некоторую безопасную точку памяти. В самых худших ситуациях, неинициализированный указатель указывает на произвольную ячейку памяти, занятую вашей программой. Это может привести к ошибочному поведению программы, поскольку указатель может указывать на различные ячейки памяти после каждого нового запуска программы. Всегда присваивайте указателям значение NULL как до их первого использования, так и после уничтожения соответствующего объекта. Если вы попробуете использовать указатель со значением NULL, ваша программа вызовет GPF, но ошибочная строка исходного кода будет выделена отладчиком, и вы сможете сразу идентифицировать проблемный указатель.

Удаление ранее удаленных указателей

Удаление указателя, который уже был удален, приведет к GPF. Совет, данный для работы с неинициализированными указателями, применим и здесь: присваивайте любому удаляемому указателю значение NULL или 0. В Си++ удаление NULL-указателя совершенно безопасно. Присваивая удаляемым указателям значение NULL, вы можете быть уверены, что случайное удаление указателя во второй раз не вызовет никаких вредных последствий.

Выход за границу массива

Выход за верхнюю границу массива может вызывать GPF. В некоторых случаях запись в занятые ячейки памяти может быть некритичной, проблема проявляется не сразу, а сбой в программе происходит только через некоторое время. Когда это случится, вы, вероятно, будете искать ошибку в той точке, где программа потерпела крах, когда на самом деле ее источник находится в абсолютно другой части программы. В других случаях такое обращение с памятью критично, и программа немедленно вызывает GPF. В экстремальных случаях программа может вызвать даже сбой Windows. Проверьте все массивы, чтобы убедиться, что вы нигде не выходите за их верхние границы.

GPF по завершении программы

Возникновение GPF при нормальном завершении программы обычно является индикатором того, что установлен слишком малый размер стека. Хотя это и маловероятно для 32-разрядной программы, но при определенных обстоятельствах может произойти. Причиной GPF по завершении программы может также быть повторное удаление указателя, о чем я говорил выше.

Короткие советы по отладке

В дополнение ко многим советам, приведенным на предыдущих страницах, вам могут пригодиться и некоторые из следующих:

- Для быстрого просмотра переменной без установки контрольной точки, измените свойство Caption формы, чтобы выводить в него значение интересующей вас переменной. Другой способ заключается в том, чтобы поместить в форму компонент Label, поскольку это очень просто сделать. Измените текст метки таким образом, чтобы показать значение переменной или любую другую информацию, которую вы сочтете нужной при отладке.
- Чтобы временно замедлить выполнение своей программы (например, для просмотра эффектов программы в замедленном темпе), установите условную контрольную точку или контрольную точку, отслеживающую данные. Эти контрольные точки замедляют выполнение программы, поскольку в них проверяется условие останова.
- Во время выполнения программы пользуйтесь диалоговым окном Evaluate/Modify, чтобы временно изменять значение переменной. Это позволит вам увидеть, как влияют на поведение программы различные значения без многократной перекомпиляции кода.
- Обычно вы не можете использовать команду Trace Into с функциями типа inline. Чтобы получить возможность трассировки этих функций, включите опцию Disable Inline Expansion (Запретить подстановку). Выключите эту опцию перед окончательным построением программы.
- Чтобы проинспектировать класс, на котором отладчик остановился в данный момент, выберите в главном меню пункт Run | Inspect и введите this в поле Expression.
- Используйте функцию MessageBeep(-1) в качестве звукового индикатора того, что достигнута определенная точка вашей программы. Эта функция API Windows выдает звуковой сигнал на встроенный динамик PC, если вызывается с аргументом -1.
- Вы можете остановить ошибочную отладку, выбрав в главном меню пункт Run | Program Reset, или нажав Ctrl+F2.
- Используйте временные переменные для разбиения длинных выражений или цепных вызовов функций так, чтобы вы могли исследовать результаты по частям.
- Используйте функцию MessageBox() для вывода информации о трассировке программы.

Заключение

Отладка — это бесконечный процесс. Отладка означает нечто большее, чем просто нахождение ошибок в своей программе. Разумные программисты применяют отладчик уже в начале разработки нового проекта. Отладчик — это инструмент разработки, а не только средство для поиска ошибок. После сегодняшнего дня вы должны, по крайней мере, иметь общее представление о том, как использовать отладчик. Вам придется поработать с ним еще немало времени, чтобы набрать необходимый опыт, но вы уже взяли хороший старт.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Моя программа работала с нормальным быстродействием, когда я запускал ее из IDE. Теперь она «течет» так же медленно, как патока в январе. В чем причина?

Вероятнее всего, установлено либо большое количество контрольных точек, которые вы выключили и забыли о них, либо в вашем коде присутствует одна или более условных контрольных точек. Откройте список контрольных точек и удалите те из них, которые в настоящее время вам не нужны. Также убедитесь в том, что вы не указали слишком большое количество переменных в списке Watch List.

У меня есть переменная, которую я хочу просматривать как в десятичном, так и в шестнадцатеричном форматах. Могу ли я сделать это в списке Watch List?

Да. Сначала добавьте переменную в список объектов наблюдения. Когда появится диалоговое окно Watch Properties, выберите опцию просмотра Decimal (Десятичный формат). Теперь снова добавьте ту же переменную, но на этот раз выберите опцию просмотра Hexadecimal (Шестнадцатеричный формат). В списке будут присутствовать оба элемента, один в десятичном, а другой в шестнадцатеричном формате.

Я пробую добавить свойство Width компонента формы в список объектов наблюдения. В результате я получаю какую-то странную ошибку, типа «Pointer to structure required on left side of something or another»(В левой части того или другого требуется указатель на структуру). Что я делаю неправильно?

Вы не можете просмотреть свойство компонента, используя Watch List. Однако это можно сделать в инспекторе отладки, найдя член FWidth класса компонента, а не свойство Width. (В переменной FWidth хранится значение свойства Width.)

Мне нужно, чтобы программа останавливалась в контрольной точке только в том случае, когда переменная достигает некоторого значения и после того, как контрольная точка была пройдена определенное число раз. Могу ли я это сделать?

Конечно. Введите условное выражение в поле Condition диалогового окна Edit Breakpoint, а в поле Pass Count — нужное значение числа проходов. Когда условие выполнится при заданном количестве проходов, программа остановится в контрольной точке.

При пошаговом выполнении кода я подхожу к функции, которую мне нужно отладить. Когда я нажимаю F8, точка выполнения проскакивает функцию вместо того, чтобы войти в нее. Что мне нужно сделать, чтобы войти в эту функцию?

Когда точка выполнения находится в той строке, где вызывается ваша функция, нажмите F7 (Trace Into) вместо F8. Теперь вы можете выполнять функцию в пошаговом режиме.

При пошаговом выполнении кода, точка выполнения скачет по всей программе, а не продвигается последовательно от строки к строке. В чем причина этого?

Причина одна: оптимизация. Если вы хотите отладить свою программу в последовательном режиме, строка за строкой, выключите всю оптимизацию, после чего перестройте проект командой Build All.

Я выполняю отладку функции в пошаговом режиме. Иногда, когда я дохожу до закрывающей фигурной скобки функции и нажимаю F8 еще раз, ничего не происходит. Почему?

Когда работа данной функции завершается, вашей программе больше нечего выполнять, и она возвращается в неактивное состояние. Фактически, здесь больше нет кода для выполнения очередного шага, поэтому отладчик возвращает управление отлаживаемой программе.

Как я могу использовать инструмент CPU View при отладке?

Чтобы вывести окно CPU View, просто выберите в главном меню пункт View | CPU. Однако работа с этим инструментом — предмет совершенно другого разговора!

Вопросы

1. Как установить контрольную точку в определенной строке программы?
2. Что такое недействительная контрольная точка?
3. Как установить условную контрольную точку?
4. Как можно изменить свойства объекта в списке Watch List?
5. Какой самый быстрый способ включить переменную в список Watch List?
6. Какой инструмент вы будете использовать для просмотра членов-данных и методов класса?
7. Каким образом можно выполнить трассировку функции при пошаговом выполнении программы под управлением отладчика?
8. Как изменить значение переменной во время работы программы?
9. В чем заключается разница между макросами TRACE и WARN?
10. Как просмотреть информацию, выведенную макросами TRACE и WARN?

Упражнения

1. Загрузите программу ScratchPad, которую вы создали несколькими днями ранее. Установите контрольные точки в функциях FileOpenClick() и FileSaveClick(). Запустите программу. Когда выполнение программы будет приостановлено, проинспектируйте соответственно классы OpenDialog и SaveDialog.
2. Продолжая упражнение 1, перейдите в пошаговый режим выполнения после остановки в контрольной точке, и изучите работу программы при пошаговом выполнении функций.
3. Загрузите программу DebugTst, которую вы создали ранее в этой главе. Установите контрольную точку в функции WatchBtnClick(). Добавьте переменные s и x в список объектов наблюдения по четыре раза. Отредактируйте каждый из элементов списка, изменив опции вывода. Запустите программу и выполните функцию в пошаговом режиме, чтобы увидеть результаты в окне Watch List.
4. Добавьте условную контрольную точку в функцию из упражнения 3. Поместите ее в строку, следующую сразу за строкой int x = Width. Задайте условие x == 0 и запустите программу. Что происходит?
5. Продолжая упражнение 4, отредактируйте контрольную точку, изменив условие на x > 400. Запустите программу. Измените размер окна и щелкните на кнопке Watch Test. Повторите это несколько раз, каждый раз меняя размер окна. Что происходит?
6. Загрузите любую программу и перейдите в редактор кода. Установите курсор в любую строку программы и выберите пункт Run to Cursor в контекстном меню редактора. Поэкспериментируйте с программой до тех пор, пока не встретится контрольная точка.
7. Снова загрузите программу DebugTst, которую вы создали ранее. Установите контрольную точку в функции WatchBtnClick() и запустите программу. Когда программа дойдет до этой точки, проверьте WatchBtn с помощью инспектора отладки.
8. Напишите программу, которая делает какие-либо простые вещи при нажатии кнопок (что именно, не важно). Поместите вызовы TRACE в каждую функцию программы. Запустите программу и нажмите каждую кнопку. Закройте программу. Просмотрите файл OutDbg1.txt в редакторе кода, чтобы увидеть результаты работы макроса TRACE.
9. Добавьте в программу, созданную в упражнении 8, несколько вызовов WARN. Запустите программу еще раз и снова просмотрите файл OutDbg1.txt.

Вспомогательные инструменты и опции C++Builder



Кент Рейсдорф

В комплект поставки C++Builder входят несколько вспомогательных инструментов, упрощающих разработку приложений. Некоторые из этих инструментов являются стандартными Windows GUI-приложениями, другие — консольными приложениями Win32. Сюда даже входят несколько DOS-приложений. Сегодня вы познакомитесь со следующими инструментами:

- Редактор изображений
- WinSight
- Инструменты командной строки

С изучения этих инструментов мы и начнем. По мере продвижения к концу этой главы вы узнаете, как включить свои собственные инструменты в меню Tools C++Builder. В конце дня вы познакомитесь с диалоговым окном Environment Options (Опции среды).

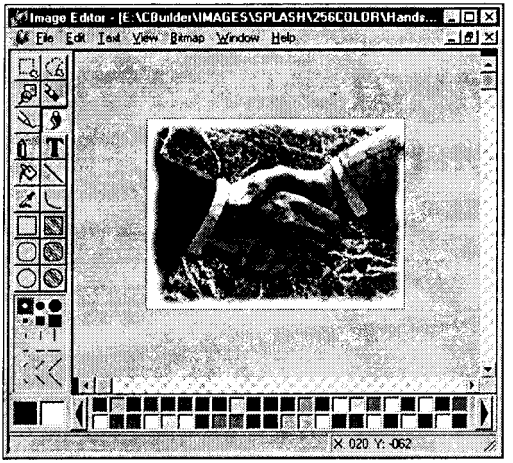


В дополнение к перечисленным выше инструментам, в состав C++Builder входит большое число инструментов, предназначенных для разработки баз данных, например, Database Desktop. Эти инструменты рассматриваются в главах, посвященных работе с базами данных в C++Builder.

Редактор изображений

Редактор изображений (Image Editor) C++Builder — это программа, которая позволяет создавать и редактировать растровые изображения (.bmp), изображения значков (.ico) и курсоров (.cur). Вы также можете создать проект ресурсов, содержащий несколько растровых изображений, значков и курсоров, объединенных в один файл ресурсов (.res). Этот файл можно включить в свой проект C++Builder и использовать ресурсы по мере необходимости. На рис. 12.1 показан редактор изображений во время работы с растровым изображением.

Рис. 12.1
Редактор изображений





Все перечисленные выше изображения являются растровыми независимо от того, собственно ли это растровые изображения в BMP-файлах или значки и курсоры. В этой главе я буду ссылаться на них одинаково, как на растровые изображения.

Чтобы запустить программу редактора изображений, вы можете дважды щелкнуть на соответствующем значке в папке C++Builder или выбрать пункт Tools | Image Editor в главном меню. Редактор изображений является автономной программой, которую можно запускать независимо от среды C++Builder.

Цвета переднего плана и фона

Редактор изображений позволяет создавать изображения с палитрой из 2, 16 или, в случае растрового файла, из 256 цветов. Создавая или редактируя растровое изображение, вы можете выбрать любой доступный вам цвет. В левом нижнем углу редактора находятся два окна, которые показывают текущий цвет переднего плана и фона. (Самое левое окно показывает цвет переднего плана.)

Применяя какой-либо из инструментов рисования, вы можете рисовать либо цветом переднего плана, либо цветом фона. Чтобы нарисовать что-либо цветом переднего плана, используйте левую кнопку мыши. Например, если вы выбрали инструмент Закрашенный прямоугольник и нарисовали прямоугольник в растровом изображении, то он будет закрашен цветом переднего плана. Чтобы нарисовать прямоугольник, закрашенный цветом фона, растяните его, удерживая нажатой правую кнопку мыши. То же самое относится и к большинству других инструментов рисования.



Инструмент Текст работает только с цветом переднего плана. Вы не сможете поместить в изображение текст, окрашенный цветом фона. Если вам нужно использовать цвет, соответствующий в настоящий момент цвету фона, то вы должны поменять цвет переднего плана и только после этого поместить текст в изображение.

Чтобы изменить цвет переднего плана, щелкните левой кнопкой мыши на выбранном вами цвете в цветовой палитре. (Цветовая палитра расположена вдоль нижнего края окна редактора изображений.) Когда вы выберите нужный вам цвет, окно, показывающее текущий цвет переднего плана, изменит свой цвет соответствующим образом. Чтобы изменить цвет фона, щелкните правой кнопкой мыши на выбранном вами цвете в цветовой палитре.



Если вы редактируете 256-цветное изображение, то цветовая палитра будет содержать кнопки прокрутки и вы сможете прокручивать палитру через все доступные цвета.

Если вы загружаете существующее растровое изображение, то цвета, которые появляются в цветовой палитре, определяются самим этим изображением. Если вы создаете новое 256-цветное изображение, то используется палитра из 256 стандартных цветов.

Вы можете задать цвет переднего плана и цвет фона с помощью инструмента Пипетка (Eyedropper). Он позволяет задать цвет переднего плана и цвет фона, выбрав их из существующих цветов самого изображения. Чтобы задать цвет переднего плана с помощью этого инструмента, выберите его из палитры инструментов, поместите тонкий конец в область изображения, содержащую нужный вам цвет, и щелкните левой кнопкой мыши. Цветом переднего плана станет цвет, на который указывал конец Пипетки. Чтобы задать цвет фона, щелкните правой кнопкой мыши вместо левой. Инструмент Пипетка неоценим, когда вам нужно выбрать цвет, в точности соответствующий одному из цветов данного растрового изображения.



Работа с цветом переднего плана и цветом фона в редакторе изображений может отличаться от других грфических редакторов, которыми вы пользуетесь. Например, в некоторых редакторах растровых изображений контур закрашенного прямоугольника может рисоваться цветом переднего плана, а сама заливка выполняться цветом фона. В рассматриваемом редакторе изображений закрашенные объекты не имеют видимого контура. Закрашенный прямоугольник имеет либо цвет переднего плана, либо цвет фона.

Прозрачный и инвертированный цвета

В случае редактирования значков и курсоров вы можете использовать прозрачный цвет (слово *цвет* здесь относительно). Через те области значка, в которых использован прозрачный цвет, вы можете видеть изображение фона. Это может быть фон Windows, если речь идет о значке, или строка заголовка окна вашего приложения. Пользуетесь вы прозрачным цветом или нет, это дело вашего личного вкуса, и все зависит от конкретного значка, который вы создаете. Если это курсор, вы почти всегда будете пользоваться прозрачным цветом в качестве его фона. Курсор, который выглядит как непрозрачный блок, может понадобиться исключительно редко.

Применение инвертированного цвета в изображении значка вызывает инверсию фона под значком (эффект цветного негатива). Инвертированный цвет редко используется, но при необходимости он всегда доступен.

Когда вы редактируете изображение значка или курсора, прозрачный и инвертированный цвета показаны рядом в палитре цветов. Эти цвета представлены квадратами, внутри которых проведена кривая линия.



По умолчанию, ресурсы нового значка и нового курсора имеют фон, закрашенный прозрачным цветом.

Инструменты рисования редактора изображений

Набор инструментов редактора изображений аналогичен большинству других программ рисования. Поскольку эти инструменты общеприняты, я не буду рассматривать каждый из них подробно. Пятнадцать минут экспериментов с редактором принесут вам больше пользы, чем то, что я могу рассказать. Так что запустите редактор изображений, поэкспериментируйте с ним некоторое время, а я подожду.

В верхней части палитры инструментов редактора изображений находятся инструменты Прямоугольная область (Marquee) и Лассо (Lasso). Оба этих инструмента применяются одинаково и служат для выделения области в текущем изображении. Инструмент Прямоугольная область предназначен для выделения прямоугольной области изображения. Выберите его и растяните границы прямоугольника в какой-либо части изображения. Когда вы закончите, границы прямоугольника будут маркировать выделенную область. Инструмент Лассо позволяет выделить область произвольной формы. Для лучшей видимости области, выделенной инструментом Лассо, она заливается наклонной мерцающей штриховкой.



Когда вы рисуете область выделения инструментом Лассо, ее не обязательно замыкать. После того как вы отпустите кнопку мыши, редактор изображений автоматически замкнет область, соединив линией начальную и конечную точки нарисованной кривой.

Как только область выделения задана, вы можете вырезать или скопировать изображение, попадающее внутрь этой области, и вставить его в любое другое место текущего изображения или в другое изображение, с которым вы работаете в настоящий момент (вы можете открыть несколько растровых изображений одновременно). Когда вы выбираете в главном меню пункт Edit | Paste (Правка | Вставить), изображение, попавшее внутрь границ области, будет помещено в левый верхний угол текущего изображения, а его границы будут маркированы. После этого вставленное изображение можно перетащить в нужное место. Когда вы помещаете курсор внутрь выделенной области, он принимает форму руки. После этого вы можете перетащить выделенное изображение и установить его в нужном месте. У вас есть возможность многократно выполнять перетаскивание, пока вы не установите изображение в точности так, как вам нужно. Как только вы это сделаете, щелкните на изображении за границей выделенной области, и она будет объединена с текущим изображением.



Редактор изображений предоставляет более короткий способ для выполнения операций вырезать/вставить. Выделите прямоугольную или произвольную область, поместите курсор внутрь этой области, после чего выделенное изображение можно сразу перетаскивать.

Когда вы вырезаете область или перетаскиваете ее мышью, освободившаяся область изображения будет закрашена цветом фона. По существу, фон «просматривается» через отверстие в изображении, образовавшееся после того, как вы вырезали его часть.



Вы можете скопировать часть одного растрового изображения в другое, используя операции вырезать/вставить. Сначала откройте оба изображения в редакторе. Выделите область в исходном изображении и выберите в главном меню пункт Edit | Copy. Сделайте активным окно другого изображения и выберите в главном меню Edit | Paste. Переместите вставленное изображение в нужное место.



Если в вашем изображении имеется выделенная область, то при вставке изображения из буфера оно будет вставлено внутрь выделенной области и растянуто (или сжато) до ее размеров.

Инструмент Ластик (Eraser) работает противоположным образом в отношении левой и правой кнопок мыши. Если вы используете левую кнопку мыши, то Ластик закрашивает изображение цветом фона, если правую — то цветом переднего плана.

Инструмент Текст (Text) позволяет вам поместить текст в изображение. Для рисования текста используются текущие установки инструмента. Их можно изменить, выбрав пункт Text в главном меню. Здесь вы можете установить выравнивание текста (влево, вправо, по центру) или тип шрифта. Чтобы изменить шрифт, выберите в главном меню Text | Font. Теперь вы можете выбрать новый шрифт или сделать шрифт полужирным, курсивным, подчеркнутым и т.д.

Назначение других инструментов рисования практически очевидно. Как я уже сказал ранее, за несколько минут, проведенных наедине с редактором изображений, вы узнаете почти все, что вам нужно знать об этих инструментах.



Рисуя прямоугольники соответствующим инструментом, вы можете нажать и удерживать клавишу Shift, чтобы свести прямоугольник к квадрату. Вы можете нарисовать точный круг с помощью инструментов Эллипс (Ellipse) или Закрашенный эллипс (Filled Ellipse), также удерживая во время растягивания эллипса клавишу Shift. Сочетание клавиши Shift с инструментом Линия (Line) позволит вам рисовать прямые линии, проходящие строго вертикально, горизонтально или под углами в 45 градусов.

Масштабирование изображения

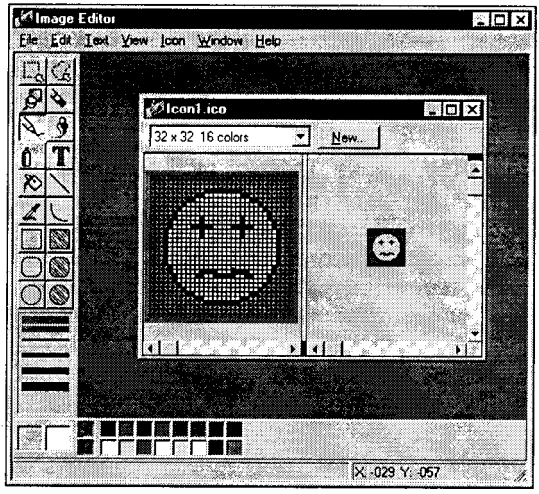
Редактор изображений позволяет вам увеличить масштаб, чтобы вы могли редактировать небольшие детали растрового изображения. Изменять масштаб

можно либо с помощью инструмента Zoom (Масштаб), либо используя пункт меню View (Вид). Чтобы увеличить определенную часть изображения с помощью инструмента Zoom, выберите его в палитре инструментов и растяните границы прямоугольника так, чтобы они охватывали интересующую вас часть изображения. Коэффициент увеличения зависит от размеров выделенной области (чем меньше размеры области, тем больше коэффициент увеличения). Теперь вы можете редактировать самые мелкие детали растрового изображения.

Чтобы увеличить масштаб изображения с помощью меню, выберите пункт View | Zoom In или нажмите комбинацию клавиш Ctrl+I. Каждый раз, когда вы выбираете в меню команду Zoom In, изображение увеличивается в определенное число раз. Чтобы уменьшить масштаб с помощью меню, выберите пункт View | Zoom Out (Ctrl+U) или View | Actual Size (Ctrl+A).

Когда вы создаете курсор или значок, редактор изображений использует разделенное окно. На рис. 12.2 показано окно редактора изображений во время редактирования значка.

Рис. 12.2
Редактирование значка



Хотя вы можете изменять масштаб в любой части разделенного окна, как правило, вы будете работать с увеличенной копией в левой части, используя фактический размер значка справа только для просмотра, как показано на этом рисунке.

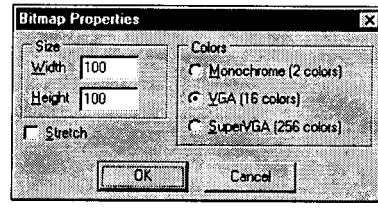
Палитра толщины линии

Палитра толщины линии отображается прямо под палитрой инструментов. В зависимости от выбранного в настоящий момент инструмента, она может отображать либо линии различной толщины, либо различные формы кисти, из которых вы можете выбирать то, что вам нужно в данный момент. Чтобы выбрать толщину линии, просто щелкните на одном из пяти предлагаемых вариантов. Все последующие операции будут использовать выбранную вами толщину линии, пока вы снова ее не измените. Точно так же выбирается и форма кисти. Если вы еще раз посмотрите на рис. 12.1, то увидите палитру толщины линии, отображающую различные формы кисти.

Работа с файлами растровых изображений

Вы можете создать растровое изображение с самого начала или загрузить существующее изображение и изменить его. Чтобы открыть существующий файл растрового изображения, выберите в главном меню пункт File | Open. Для создания нового файла выберите в главном меню пункт File | New, а затем выберите Bitmap File из всплывающего меню. Появится диалоговое окно Bitmap Properties (Свойства растрового изображения), которое показано на рис. 12.3.

Рис. 12.3
Диалоговое окно Bitmap Properties



Здесь вы можете установить начальные размеры растрового изображения (в пикселах), а также глубину цвета. Вы можете создать растровое изображение с палитрой из 2, 16 или 256 цветов.



Данный редактор не поддерживает растровые изображения с палитрой более чем из 256 цветов.

Выберите нужный вам размер изображения, глубину цвета и нажмите кнопку OK. В окне редактора появится пустое растровое изображение. Нарисуйте что-либо. Когда вы закончите редактирование, выберите File | Save или File | Save As, чтобы сохранить файл с растровым изображением на диске.

Во время работы с файлом, содержащим растровое изображение, в главном меню редактора изображений появляется пункт Bitmap. Это меню имеет единственный подпункт — Image Properties (Свойства изображения). После выбора Bitmap | Image Properties появляется практически такое же диалоговое окно Bitmap Properties, как и при создании нового файла с растровым изображением. Выберите новую ширину, высоту или глубину цвета и нажмите OK.



Диалоговое окно Bitmap Properties, которое выводится для уже созданного файла, имеет одно отличие от окна, отображаемого при создании нового растрового изображения. Оно содержит флажок, озаглавленный как Stretch (Растяжение), который используется при изменении размеров растрового изображения. Если опция Stretch выключена, то исходное растровое изображение не будет растянуто или сжато до размеров нового изображения. Если опция Stretch включена, то исходное растровое изображение будет масштабировано по высоте и/или ширине. Процедура изменения размеров растрового изображения не относится к точным операциям, и иногда ее результат может быть не вполне удовлетворительным.

В целом, редактор имеет ограниченный набор возможностей для работы с файлами растровых изображений. Несмотря на то, что их вполне хватает для простых изображений, вы, вероятно, найдете, что для работы со сложной

графикой он не вполне адекватен. Если вам требуются высококачественные изображения, вполне возможно, что вы предпочтете приобрести соответствующий многофункциональный графический пакет или поручить работу по созданию нужного вам изображения художнику по компьютерной графике.



Не забывайте проверять сетевые источники, содержащие информацию о художниках по компьютерной графике. Эти люди знают свое ремесло гораздо лучше других (кроме, возможно, самых одаренных программистов), а предлагаемые ими расценки часто достаточно обоснованы. Попробуйте повторить десять раз следующую фразу: «Я программист, а не художник». (Ну ладно, только два раза, если вам и этого хватает.)

Редактирование значков

Создание значков также является разновидностью художественной графики, но к ним предъявляются меньшие требования, чем к растровым изображениям. В большинстве случаев вы будете создавать свои собственные значки, но и здесь необходим некоторый опыт, чтобы значок действительно хорошо выглядел. Если вы еще раз посмотрите на рис. 12.2, то увидите, как выглядит значок в окне редактора изображений во время редактирования.



Загрузите файлы со значками из любого доступного вам источника, увеличьте их изображения и посмотрите, как создаются лучшие образцы значков. Создание трехмерных значков требует практики.

Значок в 32-разрядной среде Windows — это фактически два значка в одном. Большой значок имеет размер 32x32 пиксела. Такой значок может быть помещен в диалоговое окно, например в окно About. Windows будет использовать этот значок и при создании ярлычка вашего приложения. Большой значок также используется программой Windows Explorer, когда она настроена на вывод списка файлов большими значками.

Малые значки имеют размер 16x16 пикселов и используются Windows в заголовке окна вашего приложения, в панели задач Windows, в диалоговом окне File Open и в программе Windows Explorer, когда она настроена на вывод малых значков. И большой и малый значки хранятся в одном и том же файле значка (.ico).



Вам не нужно разрабатывать большой и малый значки одновременно. Если вы разработаете только большой значок, Windows сожмет его до размеров малого значка, когда это потребуется. Иногда качество значка после этой операции может не отвечать вашим ожиданиям. В этих случаях вы можете создать оба значка самостоятельно, чтобы полностью взять в свои руки управление внешним видом своего приложения, а не полагаться на Windows.

Чтобы создать новый ресурс значка, выберите в главном меню пункт File | New, после чего выберите Icon во всплывающем меню. Если вы создаете новый значок в редакторе изображений, вы увидите диалоговое окно Icon Properties (Свойства значка), которое показано на рис. 12.4.

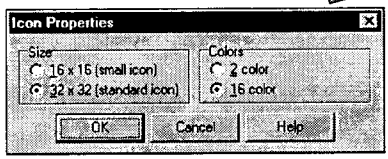


Рис. 12.4
Диалоговое окно Icon Properties

Эта диалоговое окно позволяет выбрать размер создаваемого значка (большой или малый) и количество цветов в палитре. По умолчанию вам предлагается создать стандартный значок (большой) с палитрой из 16 цветов. (В действительности, 2-цветная палитра используется очень редко. Попробуйте вспомнить, когда вы в последний раз видели такой значок?)



Даже если вы создаете значки обоих размеров, для начала работы вы должны выбрать один из них, например, большой. После его завершения вы можете создать вариант с меньшими размерами.

Во время редактирования значка строка меню редактора изображений содержит пункт Icon. Это меню имеет подпункты New (Новый), Delete (Удалить) и Test (Проверить). Пункт New позволяет создать новый большой или малый значок. Например, если вы уже закончили разработку большого значка, вы могли бы выбрать в главном меню Icon | New, чтобы создать малый значок.



Окно редактирования значка редактора изображений содержит кнопку New, которая позволяет быстро выполнить ту же команду, а не выбирать пункты Icon | New в главном меню. (Обратитесь к рис. 12.2.)

Когда вы выбираете New, чтобы создать второй значок, снова появляется диалоговое окно Icon Properties. Если вы уже создали большой значок, то по умолчанию в этом окне будет выбран уменьшенный вариант значка, и вам останется только нажать кнопку OK. В окне редактора появится новый чистый значок. Редактор изображений не позволит вам повторно создать значок того же размера, который уже содержится в данном файле.



Если в редактор одновременно загружены значки обоих размеров, вы можете переключаться на каждый из них, используя комбинированный список в верхней части окна редактирования значка.

Пункт Delete в меню Icon позволяет полностью удалить один из значков из ресурса значка. Но вы не сможете удалить значок, который остался последним в этом ресурсе.

Пункт Test в меню Icon выводит диалоговое окно Icon Tester (Тестер значка), которое показывает, как будет выглядеть данный значок. На рис. 12.5 диалоговое окно Icon Tester показано в действии.

Тестер значка позволяет менять цвет фона, так что вы сможете наглядно увидеть, как влияет различный фон на восприятие вашего значка. Если в настоящий момент вы редактируете большой значок, то в диалоговое окно тестера будет выведен большой значок. Если вы редактируете малый значок, то увидите именно его.

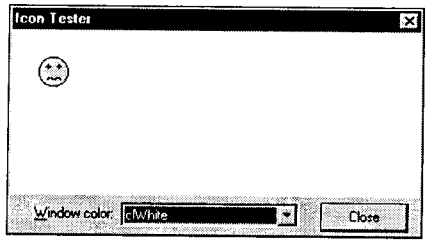


Рис. 12.5
Диалоговое окно Icon Tester

Редактирование курсоров

Работа с курсорами очень незначительно отличается от работы со значками. Курсор имеет только два цвета: белый и черный. (Многоцветные курсоры с анимацией не поддерживаются редактором изображений.) Так что вам нужно только нарисовать свой вариант курсора.



Особенность редактора курсора состоит в том, что прозрачный цвет «окрашен» в цвет фона, установленный на вашей системе, а не в темно-зеленый цвет, использующийся в редакторе значка. Если на вашей системе установлен очень светлый цвет фона Windows, возможно, что вам будет трудно различить, какие части изображения являются прозрачными, а какие белыми. Если вы столкнулись с подобной проблемой, установите другой цвет фона Windows (например, темно-зеленый).

Так же как и при редактировании растровых изображений или значков, в главном меню редактора изображений появляется новый пункт Cursor, когда вы редактируете изображение курсора. Это меню имеет два подпункта: Set Hot Spot (Установить активную точку) и Test (Проверить).

Пункт меню Set Hot Spot позволяет задать активную точку курсора. *Активная точка* — это определенный пиксел в изображении курсора, использующийся Windows для определения координат мыши. Чтобы задать активную точку, выберите в главном меню Cursor | Set Hot Spot. В ответ появится диалоговое окно, где вы можете ввести координаты x и y для активной точки.



Для активной точки вы должны ввести точные значения координат x и y. Предварительно, чтобы упростить задачу, поместите курсор редактора изображений на точку, которую вы хотите сделать активной. В строке состояния редактора изображений будут выведены координаты x и y точки, в которой находится курсор. Запишите значения координат, после чего выберите в главном меню пункт Cursor | Set Hot Spot и введите эти значения в качестве координат x и y активной точки.

Пункт Test в меню Cursor дает возможность проверить работу вашего нового курсора. Выберите в главном меню Cursor | Test. Появится диалоговое окно Cursor Tester (Тестер курсора), которое показано на рис. 12.6.

Нажмите и удерживайте любую кнопку мыши и нарисуйте что-либо в окне тестера курсора. Если вы еще не установили активную точку, то, вероятно, заметите, что активной точкой по умолчанию является правый нижний угол изображения курсора. Вы всегда должны устанавливать активную

точку в таком месте курсора, чтобы это выглядело логичным для пользователей вашего приложения.

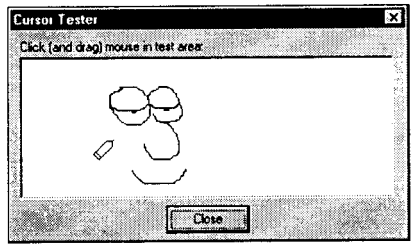


Рис. 12.6
Тестирование курсора

Контекстные меню редактора изображений

Редактор изображений позволяет использовать контекстные меню в любом режиме редактирования (растровый, режимы курсора и значка). Вы могли вспомнить, что правая кнопка мыши используется для рисования, поэтому вы не сможете использовать ее для вывода этих меню, если щелкнуть в пределах изображения. Для вывода контекстных меню редактора изображений используйте щелчок правой кнопкой, когда курсор находится в пределах окна редактора, но вне самого изображения. Контекстные меню содержат пункты, которые можно найти в отдельных меню, обсуждавшихся в предыдущих разделах.

Создание проекта ресурсов

Редактор изображений позволяет также создать файл проекта ресурсов, который может использоваться для хранения всех ваших растровых изображений, значков и курсоров. Чтобы создать проект ресурсов, выберите в главном меню пункт File | New и выберите во всплывающем меню пункт Resource File (Файл ресурсов). Появится окно проекта. Окно проекта — это дерево просмотра (очень напоминает проводник проекта (Project Explorer) C++Builder), которое показывает растровые изображения, значки и курсоры, входящие в проект. На рис. 12.7 показан образец окна проекта.

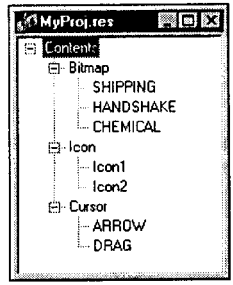


Рис. 12.7
Окно проекта редактора изображений

Окно проекта имеет контекстное меню, которое может использоваться для создания новых ресурсов, редактирования, переименования и удаления ресурсов. Пункты контекстного меню также дублируются в пункте главного меню Resource.

Когда вы сохраняете проект ресурсов, редактор изображений выполняет его компиляцию в двоичный файл ресурсов (.res). После этого вы можете добавить двоичный файл ресурсов к проекту своего приложения.

Создание новых ресурсов

Чтобы создать новый ресурс для вашего проекта, выберите пункт New в контекстном меню окна проекта или Resource | New в главном меню. Теперь вы можете создать новое растровое изображение, значок или курсор, как вы делали это ранее, когда создавали индивидуальный файл ресурсов. Появится окно редактора выбранного ресурса, где вы можете создать ресурс, используя инструменты рисования.

Редактирование ресурсов

После того как вы создали ресурс, вам может понадобиться внести в него какие-либо изменения. Чтобы отредактировать существующий проект ресурсов, найдите данный ресурс в дереве ресурсов проекта и дважды щелкните на его имени. После этого появится окно редактора данного ресурса.

Переименование ресурсов

Переименование ресурсов может быть выполнено непосредственно в дереве ресурсов (редактирование по месту). Просто щелкните на нужном пункте, чтобы его выбрать, и щелкните на нем еще раз, чтобы начать редактирование имени. Другой способ — выбрать пункт Rename (Переименовать) в контекстном меню. После того, как вы введете новое имя ресурса, нажмите Enter или щелкните на другом пункте дерева, и имя ресурса изменится.

Удаление ресурсов

Чтобы удалить ресурс из списка ресурсов проекта, выберите щелчком нужное имя в дереве ресурсов и выберите в контекстном меню пункт Delete (Удалить). Вам будет задан вопрос, действительно ли вы хотите удалить ресурс, и после утвердительного ответа ресурс будет удален. Обратите внимание, что эту операцию нельзя отменить, так что убедитесь в том, что данный ресурс действительно вам не нужен.

Включение ресурсов из других файлов ресурсов

К сожалению, не существует простого способа добавить в проект ресурсов другой ресурс, содержащийся в отдельном файле. Для выполнения этой операции сделайте следующее. Откройте одновременно свой файл проекта и индивидуальный файл, содержащий растровое изображение, значок или курсор, который вы хотите включить в проект. Перейдите в этот файл и выберите в главном меню пункт Edit | Select All, чтобы выбрать ресурс в целом; затем выберите Edit | Copy, чтобы скопировать объект в буфер обмена. Создайте в проекте ресурсов новый ресурс. Когда будет выведено окно редактора ресурсов, выберите в главном меню Edit | Paste, чтобы вставить скопированный ресурс в только что созданный.



Если объект, который вы добавляете в проект ресурсов, является растровым изображением, обязательно проверьте его атрибуты (ширину, высоту и глубину цвета). Эта информация понадобится при создании нового растрового изображения в проекте ресурсов.

Редактор изображений не относится к программам высокого класса, предназначенным для редактирования изображений, но он достаточен для решения многих ваших задач по редактированию изображений. Его просто использовать и он хорошо подходит для создания большинства значков и курсоров.

WinSight: наблюдение за Windows

WinSight — это утилита, которая позволяет вам подсматривать за Windows. WinSight покажет вам любое работающее в данный момент приложение и любое окно, работающее под управлением этого приложения. (Не забудьте, что элементы управления — это тоже окна.) WinSight показывает вам все сообщения, генерируемые Windows (то есть каждое сообщение, посланное окну). Вы можете выбрать просмотр всех сообщений или только тех, которые посланы определенному окну. Чтобы запустить WinSight, выберите в главном меню C++Builder пункт Tools | WinSight. Если в вашем меню Tools нет пункта WinSight, см. раздел данной главы «Конфигурирование меню Tools C++Builder», где описана процедура включения WinSight в меню Tools (имя файла — WS32.exe). Вы можете также запустить WinSight, дважды щелкнув на значке WinSight32 в папке C++Builder. WinSight, подобно редактору изображений, является автономной программой, которая может быть запущена вне оболочки C++Builder. На рис. 12.8 показано окно программы WinSight, которая наблюдает за Windows Explorer (Проводник).

Из рис. 12.8 видно, что окно WinSight разделено на два подокна. В верхнем подокне перечислены активные окна, а нижнее подокно показывает сообщения, посылаемые определенному окну или всем окнам. Размер подокон может корректироваться перемещением разделителя, расположенного между подокнами. По умолчанию окно разделено горизонтально, но вы можете вы-

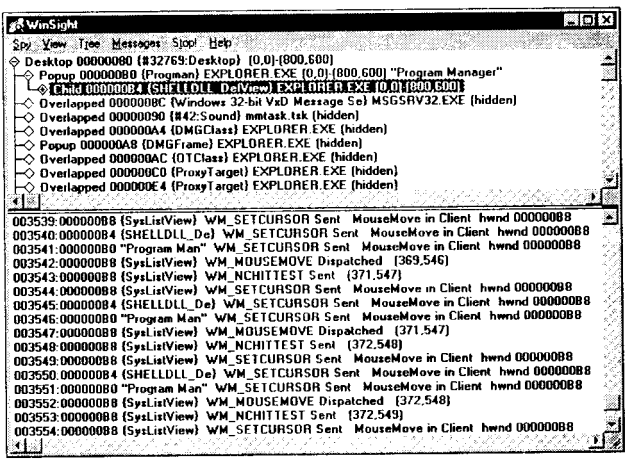


Рис. 12.8
WinSight в действии

брать и вертикальное разделение. Чтобы изменять расположение окон, выберите в меню Edit либо пункт Split Horizontal (Горизонтальное разделение), либо пункт Split Vertical (Вертикальное разделение).

Мы подробнее изучим каждое из двух окон чуть позже, а сначала давайте поговорим о сообщениях Windows.

Система сообщений Windows

От наблюдения за Windows мало пользы, если вы не знаете, что означают те или иные вещи. На самом деле, вы должны потратить большое количество времени, программируя в Windows, прежде чем будете понимать всю информацию, представляемую WinSight.

Программы, подобные C++Builder и Delphi, хороши тем, что позволяют создавать автономные приложения Windows за самое короткое время, которое только возможно. Если у этих сред программирования и есть недостаток, то он заключается в том, что вам не дается возможности узнать, что в действительности заставляет работать программу Windows.

Сообщения — именно они заставляют работать программу Windows. Большое, большое количество сообщений. Windows посылает сообщения окну, чтобы инициировать какие-либо его действия или уведомить окно, что произошло некоторое событие. Например, если окно должно быть перерисовано, Windows посылает ему сообщение WM_PAINT. Сообщение WM_PAINT указывает окну на необходимость перерисовки. Когда изменяются размеры окна, Windows посылает сообщение WM_WINDOWPOSCHANGING, которое уведомляет окно, что его размеры и/или положение изменяются. За этим сообщением следуют сообщения WM_WINDOWPOSCHANGED и WM_SIZE после того, как размеры окна и его положение были изменены. В типичной среде Windows такого рода вещи происходят десятки или даже сотни раз в секунду.

Существует более 100 сообщений, которые Windows может посылать приложению. Я немного упоминал об этом в 5-й день, «Библиотеки классов C++ и модель визуальных компонентов», где обсуждались события в VCL. Многие из событий, на которые реагирует программа C++Builder — это сообщения Windows. Событие OnCreate генерируется в ответ на сообщение WM_CREATE, событие OnSize генерируется в ответ на сообщение WM_SIZE, событие OnMouseDown соответствует двум сообщениям: WM_LBUTTONDOWN и WM_RBUTTONDOWN. Этот список можно продолжать и продолжать. C++Builder позволяет вам иметь дело с сообщениями на более высоком уровне, освобождая время для написания более важных частей приложения.

В конце концов вы захотите узнать, что же заставляет работать Windows. Огромное достоинство C++Builder заключается в том, что вы можете создавать программы Windows прямо сейчас и вместе с тем постепенно знакомиться со всей низкоуровневой «начинкой».



Некоторые очень хорошие Windows-программисты могли бы с этим не согласиться, говоря, что вы должны сначала научиться создавать программы для Windows на Си, как в старые добрые времена. Теоретически, чтобы эффективно программировать под Windows в любой среде, необходимо знать основы. С одной стороны, я согласился бы с тем, что это — оптимальный сценарий. Но с другой стороны, в наши дни очень немногие программисты имеют время на обучение по такой схеме.

После этого введения в сообщения Windows, давайте вернемся к WinSight и посмотрим, как работает эта программа.

Дерево окон

Верхнее подокно программы WinSight называется деревом окон (Window Tree). Дерево окон показывает все открытые в настоящий момент окна. Оно также сообщает некоторые подробности относительно класса данного окна. Например, на рис. 12.8 вы видите, что в дереве окон перечислены строки, содержащие имя программы Windows Explorer. Детали выглядят следующим образом (за исключением того, что строка, приведенная ниже, занимает на экране одну строку):

```
Overlapped 00000D74 {ExploreWClass}
  EXPLORER.EXE (730,14)-(935,460) "Exploring - aTemplate"
```

Первый пункт в этой строке показывает стиль данного окна. В данном случае это перекрытое (overlapped) окно (WS_OVERLAPPED, для старых хакеров Windows). Другие варианты этого пункта включают дочерние (child) и всплывающие (pop-up) окна. (Существуют и другие типы окон, но перекрытое, дочернее и всплывающее встречаются наиболее часто.) Второй столбец показывает дескриптор окна (HWND). Он коррелирует со свойством Handle оконного компонента VCL.

Далее вы видите имя класса окна, заключенное в фигурные скобки. Это имя класса, которое использовано приложением при регистрации данного окна в Windows. Зачастую окна совместно используют одно и тоже имя класса. Например, класс общего элемента управления — кнопки имеет имя Button. В различных приложениях в любой данный момент может существовать множество кнопок, но все они относятся к одному и тому же классу окна. В случае приложений, созданных в среде C++Builder, формы и компоненты будут показывать класс VCL, который соответствует этому компоненту. Для компонента, представляющего кнопку, WinSight покажет имя класса TButton.

После имени класса следует имя модуля процесса, который создал окно. Обычно модулем является исполняемая программа. В данном примере имя модуля — EXPLORER.EXE. После имени модуля следует размер окна и его положение. В конце вы видите текст окна. Для перекрытых окон это обычно тот текст, который появляется в заголовке окна. Для других типов окна значение этого текста различно. Для кнопки, например, это будет текст, отображенный на самой кнопке.



Общая практика коммерческих приложений — создание скрытых окон для выполнения определенной задачи. Для скрытых окон после имени модуля следует только текст (hidden) (скрытое). Поскольку окно скрыто, то для него нет ни информации о размере и положении, ни какого-либо текста.

Дерево окон имеет иерархическую структуру. В самой верхней части дерева находится рабочий стол (Desktop) Windows. Ниже него показаны окна, созданные как потомки рабочего стола. Исполняемый файл, например, появится ниже узла рабочего стола. Данное окно может иметь под собой другие порожденные окна. Предки, их потомки, а также любые элементы одного уровня соединяются линиями.

Обратите внимание, что на рис. 12.8. слева от каждого пункта в окне дерева показан ромб. Если окно имеет дочерние элементы, то внутри ромба находится знак «плюс» или знак «минус». Если в ромбе стоит знак «плюс», это означает, что данный узел находится в свернутом состоянии и может быть раскрыт, чтобы показать дочерние окна. Если в узле стоит знак «минус» — узел полностью раскрыт. Вы можете раскрыть или свернуть узел, щелкнув в каком-либо месте слева от данного пункта. Пустые ромбы указывают на окно, не имеющее порожденных окон.



Если данное окно доступно в дереве окон, то каждый раз, когда оно получает сообщение от Windows, ромб рядом с пунктом будет подсвечиваться.

Окно трассировки сообщений

Окно трассировки сообщений (Message Trace) показывает отдельные сообщения по мере их генерации Windows. Типичный пункт в этом окне выглядит следующим образом:

```
000684:00000854 {TMemo} WM_KEYDOWN Dispatched 48h 72d VK_H Scan 23h Down
```

Не стоит слишком подробно анализировать данное сообщение, поскольку конкретные детали существенно меняются от сообщения к сообщению. В данном случае, компонент Memo получил сообщение WM_KEYDOWN с параметром VK_H. Другими словами, пользователь нажал клавишу h, когда курсор находился в данном компоненте. Как вы видите, часть информации из окна трассировки также показана и в дереве окон. Например, отображаются как дескриптор окна, так и имя класса.

Чтобы начать трассировку сообщений, выберите в главном меню пункт Start!. В окне трассировки начнут появляться сообщения по мере того, как они принимаются окном или окнами, которые вы выбрали. Чтобы остановить трассировку, выберите в главном меню пункт Stop!.



Пункты Stop! и Start! главного меню занимают одно и тоже место в строке меню. Если трассировка выключена, то пункт меню будет называться Start! Если трассировка включена, то пункт меню будет называться Stop!.

Сообщения прокручиваются через окно трассировки сообщений по мере их поступления. Вы всегда можете остановить трассировку и прокрутить список в обратном направлении, чтобы просмотреть любые сообщения, которые вышли из поля зрения. Другая существующая опция — направить вывод сообщений в файл регистрации. Опции WinSight рассматриваются позже в разделе «Опции трассировки сообщений».

Наблюдение за окном

WinSight позволяет просматривать все сообщения, посланные всем окнам или сообщения, посланные только одному окну. Чтобы просматривать все сообщения, выберите в главном меню пункт Messages | All Windows (Сообщения | Все окна). Хотя вы и можете выбрать просмотр всех сообщений, посланных всем окнам, обычно это не очень продуктивно. В системе суще-

ствует так много сообщений, что достаточно трудно поймать именно то, что вы ищете.

Лучший метод состоит в том, чтобы выбрать определенное окно и наблюдать за сообщениями только этого окна. Это снизит нежелательный «шум» в окне трассировки сообщений до допустимого уровня. Чтобы просматривать сообщения для определенного окна, найдите это окно в дереве сообщений и выберите его щелчком. Теперь выберите в главном меню пункт Messages | Selected Windows (Сообщения | Выбранное окно). Выберите Start!, после чего любые сообщения, посланные данному окну, будут показаны в окне трассировки сообщений.

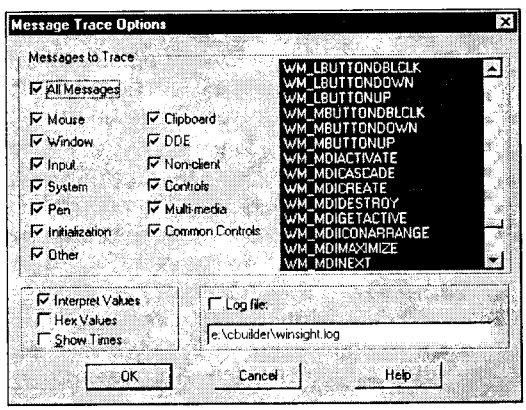


Проще всего работать с WinSight, имея четкое представление о предмете поиска. Поскольку даже одному окну посылается очень много сообщений, то вероятнее всего искомое сообщение пройдет через окно трассировки очень быстро. Чтобы максимизировать эффективность WinSight, выберите окно, которое вас интересует, начните трассировку его сообщений, проделайте с окном те манипуляции, которые вам нужны, и выключите трассировку.

Опции трассировки сообщений

Вы можете управлять сообщениями, которые отображаются в окне трассировки сообщений, с помощью диалогового окна Message Trace Options (Опции трассировки сообщений). (См. рис. 12.9.) Вы можете также изменить способ вывода сообщений. Чтобы вывести диалоговое окно Message Trace Options, выберите в главном меню пункт Messages | Options.

Рис. 12.9
Диалоговое окно Message Trace Options



Большая часть этого диалогового окна — раздел, озаглавленный Messages to Trace (Сообщения для трассировки). В левой части этого раздела показаны несколько групп сообщений, а справа находится окно списка, отображающее все сообщения Windows. Как только вы включите или выключите отметку флажка данной группы сообщений, в окне списка будут выбраны соответствующие сообщения или их выбор будет отменен.



Сообщения в окне списка, которые выведены в верхнем регистре — это стандартные сообщения Windows. Сообщения, выведенные в нижнем регистре — это «недокументированные» сообщения Windows. Последние, как правило, относятся к внутренним сообщениям Windows, которые не приведены в справочных файлах API Windows.

Если опция All Messages (Все сообщения) включена, то WinSight прослеживает все сообщения Windows. Чтобы немного сузить диапазон трассировки и уменьшать «шум» в окне трассировки сообщений, вы можете выбрать просмотр отдельных групп сообщений. Если, например, вас интересуют только сообщения мыши, вы можете сбросить флажок All Messages и установить флажок Mouse. Для дальнейшего сужения диапазона сообщений, вы могли бы выключить все опции и выбрать в правом окне списка диалогового окна только то сообщение, которое вас интересует. Например, если вас интересует сообщение WM_LBUTTONDOWN, вы выключаете все опции и выбираете в окне списка сообщение WM_LBUTTONDOWN. Тогда в окно трассировки сообщений будут выводиться только сообщения WM_LBUTTONDOWN.

Другая группа опций диалогового окна Message Trace Options управляет способом вывода сообщений в окно трассировки. Опция Interpret Value (Интерпретировать значения) переводит WinSight в режим представления параметров каждого сообщения в более подробном формате. Например, приведенное ниже сообщение WM_KEYDOWN:

```
0000309:00000474 {TMemo} WM_KEYDOWN Dispatched
```

будет выведено в формате:

```
0000309:00000474 {TMemo} WM_KEYDOWN Dispatched 44h 68d VK_D Scan 20h Repeat
```

Большую часть времени я предпочитаю видеть более подробные сообщения, так что в основном я оставляю опцию Interpret Values включенной.

Опция Hex Values (Шестнадцатеричные значения) переводит WinSight в режим отображения значений в окне трассировки сообщений в шестнадцатеричном формате. Это полезно в некоторых ситуациях, но, возможно, вы не будете этим пользоваться, пока не наберете соответствующий опыт программирования. Опция Show Times (Показать время) редко используется, поскольку она включает в сообщения значение системного времени, от знания которого не так много пользы.

Опция Log File (Файл регистрации) позволяет вам направить результат трассировки сообщений в дисковый файл. Когда эта опция включена, сообщения в окне трассировки будут отображаться по-прежнему. Эта опция полезна в тех случаях, когда вам нужно сохранить копию генерируемых сообщений.



Создание регистрационного файла имеет и другое преимущество. Создав регистрационный файл и загрузив его в текстовый редактор, вы можете использовать функцию поиска текстового редактора, чтобы найти интересующее вас сообщение или параметры сообщений, которые вы хотите просмотреть.

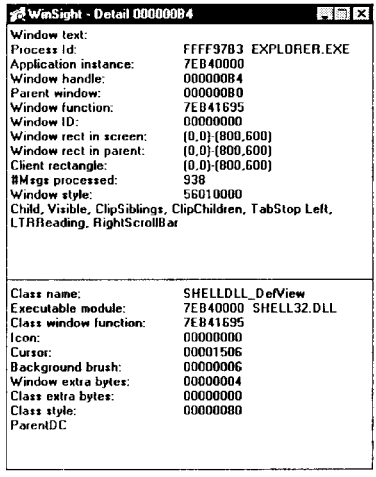
Другие возможности WinSight

WinSight имеет несколько других возможностей, которые упрощают поиск и проверку окон.

Просмотр деталей окна

WinSight имеет опцию Detail (Детали), которая показывает вам подробную информацию об определенном окне. Чтобы вывести детальную информацию, выберите нужное вам окно в дереве окон, а затем выберите в главном меню пункт Spy | Open Detail (Наблюдать | Открыть детали). Вы можете также дважды щелкнуть на интересующем вас окне в дереве сообщений или нажать Enter на клавиатуре. На рис. 12.10 показаны детали окна Windows Explorer.

Рис. 12.10
Окно детальной информации WinSight



Как вы видите, окно деталей показывает вам очень подробную информацию о выбранном окне. (Возможно даже более подробную, чем вам необходимо!)

Слежение за фокусом

Опция Follow to Focus (Следовать за фокусом) позволяет вам выбрать определенное окно приложения, после чего это окно становится активным в дереве окон. Дерево окон обычно содержит большое число окон. Из-за этого часто трудно найти то окно, которое вам нужно, так что возможность следовать за фокусом действительно помогает.

Чтобы использовать эту возможность, выберите в главном меню пункт Spy | Follow to Focus. Затем переключитесь на приложение, содержащее интересующее вас окно, и, если это необходимо, щелкните на определенном элементе управления, за которым вы хотите следить. WinSight автоматически выберет это окно в дереве окон. Чтобы начать слежение за сообщениями этого окна, выберите в главном меню Start!.



Опция Follow to Focus остается включенной, пока вы ее не выключите. За исключением тех случаев, когда вам нужно следить за другими окнами или элементами управления вашего приложения, вам следует сразу выключить опцию Follow to Focus, как только будет найдено интересующее вас окно.

Поиск окна

Опция Find Window (Найти окно) WinSight прямо противоположна опции Follow to Focus. Чтобы использовать опцию Find Window, сначала найдите в дереве окон интересующее вас окно и выберите в главном меню пункт Spy | Find Window. WinSight поместит вокруг окна, которое вы ищете, мигающую утолщенную рамку. Рамка остается вокруг окна до тех пор, пока вы не щелкнете в пределах окна или не выберете другое окно в дереве окон. Утолщенная рамка рисуется поверх всех окон, так что вы сможете найти интересующее вас окно, даже если оно скрыто под другими окнами. Окно, найденное с помощью Find Window, не перемещается поверх других окон и не получает фокус ввода; вы просто сможете его локализовать.



Опция Find Window не может найти скрытые или свернутые окна.

Режим Find Window выключается сразу, как только фокус ввода выходит из дерева окон. Если вы щелкнете где-нибудь еще в WinSight или в другом приложении, режим Find Window выключится.

Переключение фокуса ввода

Возможность Switch To (Переключить на) позволяет вам переключиться на определенное окно приложения. Для этого просто найдите интересующее вас окно в дереве окон и выберите в главном меню пункт Spy | Switch To. Выбранное окно переместится поверх других окон и будет содержать фокус ввода. Если это окно свернуто, то будет восстановлено его предыдущее положение. Опция Switch To не работает со скрытыми окнами.

Потребуется некоторое время работы с WinSight, пока вы не начнете понимать каждое посланное сообщение и его значимость. Но вы, несомненно, разберетесь во всем этом.

Инструменты командной строки

В дополнение к GUI-инструментам, подобным редактору изображений и WinSight, в поставку C++Builder входит также ряд инструментов командной строки. Некоторые из инструментов используются самой средой C++Builder при компиляции вашего проекта. Другие являются дополнительными инструментами, к которым вы можете обращаться по мере необходимости. Несколько из наиболее часто используемых инструментов обсуждаются в следующих разделах. В некоторых случаях инструменты могут быть добавлены к меню Tools так, чтобы вы смогли обращаться к ним из IDE, а не запускать их в сеансе MS-DOS.



Все инструменты командной строки постоянно находятся в каталоге bin C++Builder. Этот каталог был добавлен к пути (PATH) вашей системы при обычной установке C++Builder. Поэтому при запуске утилит из MS-DOS нет необходимости указывать полный путь с именем файла. Если, однако, вы получаете сообщение, что файл не найден, вам может понадобиться ввести точный путь к каталогу bin.

grep.exe

grep (Global Regular Expression Print — Глобальная печать регулярного выражения) — это утилита поиска текста. Вы можете использовать grep для поиска текста в своих исходных файлах (или любых других текстовых файлах). Я часто использую grep для поиска определенных функций, переменных или классов в исходном коде VCL. grep — это удачное сочетание быстродействия и мощности. Эта утилита полезна и удобна, и вы должны знать, как ею пользоваться.

grep необходимо запускать из MS-DOS. Текст, который вы ищете, а также любые опции вводятся в командной строке. Синтаксис командной строки grep выглядит следующим образом:

```
[-опции] строка_поиска [файл(ы) ... ]
```

Если вы воспринимаете подобные вещи так же, как и я, то из приведенной синтаксической конструкции можно извлечь не слишком много, поэтому конкретный пример не помешает. Предположим, что вам нужно найти во всех исходных файлах текущего каталога строку AboutBox. В ответ на приглашение MS-DOS следует ввести следующую строку:

```
grep AboutBox *.cpp
```

Если вам нужно найти строку TComponent в исходных файлах VCL, вы должны ввести следующее:

```
grep TComponent c:\Cbuilder\source\vcl\*.pas
```

Искомая строка, которая включает в себя пробел, должна быть заключена в двойные кавычки. Предположим, например, что вам нужно найти определение константы SS_ETCHEDFRAME в заголовочных файлах Windows. В этом случае вы должны набрать следующую командную строку:

```
grep -d "#define SS_ETCHEDFRAME" c:\CBuilder\include\*.h
```

В этом примере grep будет просматривать все заголовочные файлы в каталоге C++Builder \Include и во всех подкаталогах, поскольку был установлен переключатель -d. Результат любого поиска будет выводиться на консоль. Вы можете ввести единственное имя файла или использовать групповые символы, чтобы просматривать группу файлов за один запуск grep. Вы можете указать путь или не указывать его. Если путь не определен, то будет просмотрен только текущий каталог. Когда вы указываете файлы поиска, применимы все обычные групповые символы.



Поскольку найденный текст может быть довольно длинным, вам может понадобиться перенаправить вывод в файл. Возьмем, например, следующую строку:

```
grep TComponent c:\CBuilder\source\vcl\*.pas > results.txt
```

Эта строка создаст текстовый файл ASCII с именем results.txt, который вы можете просмотреть в любом текстовом редакторе.

В дополнение к искомому тексту и имени файла, вы можете установить в командной строке различные опции. Существует большое количество опций, которые вы можете использовать, но здесь я коснусь только некоторых.

Переключатель -d указывает grep просматривать заданный каталог вместе со всеми его подкаталогами. Если этот переключатель не используется, то поиск будет осуществляться только в указанном каталоге. Переключатель -d увеличивает время поиска, поскольку будет просмотрено большее количество файлов.

Переключатель -i указывает grep не учитывать регистр. По умолчанию регистр искомого текста учитывается. Используйте переключатель -i, когда вы не уверены, что искомый текст набран прописными буквами. Вообще говоря, учет регистра дает более точные результаты, но за счет некоторого возрастания времени поиска.

Переключатель -r разрешает появление регулярных выражений в строке поиска. Я кратко обсуждал регулярные выражения, когда говорил о диалоговых окнах Find и Replace на 10-й день, «Подробнее о проектах». Применение регулярных выражений подробно рассмотрено во встроенном справочном файле C++Builder.

Пользователь может изменить набор опций командной строки grep, который установлен по умолчанию. При запуске grep по умолчанию используется только переключатель -r. Все другие переключатели должны быть введены в командную строку при каждом запуске grep. Чтобы изменить установки grep по умолчанию, необходимо применить переключатель -u. Этот переключатель заставляет grep добавить любые указанные переключатели к заданному по умолчанию списку. Например, если вы хотите постоянно просматривать подкаталоги, вы могли бы добавить к значениям по умолчанию переключатель -d. Для этого в ответ на приглашение DOS нужно ввести

```
grep -u -d mygrep.exe
```

В результате будет создан новый файл с именем mygrep.exe. Как только вы выполните эту командную строку, вы можете удалить старый файл grep.exe и переименовать созданный файл в grep.exe. Теперь каждый раз, когда вы запускаете grep, автоматически будет применен переключатель -d. Чтобы удалить переключатель -d из списка по умолчанию, запустите grep следующим образом:

```
grep -u -d-mygrep.exe
```

После этого вы еще раз должны удалить старый файл grep.exe и переименовать созданный файл в mygrep.exe.

Чтобы увидеть, какие опции командной строки установлены по умолчанию, вы можете запустить grep с вопросительным знаком (?) в командной строке. Опция ? выведет справочный экран с инструкциями. Опции командной строки по умолчанию также перечислены в этом экране и отмечены знаком «плюс» (+), следующим за опцией. Выключенные по умолчанию опции отмечены знаком «минус» (). Командная строка имеет в общей сложности 12 опций. За более подробной информацией относительно других опций командной строки обратитесь к встроенной справке по grep.

implib.exe

Эта утилита создает библиотечный файл импорта (.lib) для динамически компонуемой библиотеки (.dll). Прежде чем вы сможете обращаться к функциям DLL, компоновщик должен иметь возможность «видеть» функции, экспортируемые из DLL. Библиотека импорта читается компоновщиком, кото-

рый определяет из нее, какие функции содержатся в DLL. Библиотеки импорта будут рассмотрены более подробно в 19-й день, «Создание и использование DLL».

Утилита implib проста в использовании. Предположим, что вы создали DLL с именем MyStuff.dll и вам нужно создать для нее библиотечный файл импорта. Откройте окно MS-DOS из Windows и войдите в каталог, в котором находится ваша DLL. В ответ на приглашение MS-DOS введите:

```
implib MyStuff MyStuff.dll
```

В результате будет создан библиотечный файл импорта с именем MyStuff.lib. Теперь вы можете добавить его в любой проект C++Builder, который обращается к данной DLL.



Microsoft и Borland используют различные схемы кодировки имен для своих компиляторов. Если вы покупаете у независимого производителя элемент управления или библиотеку, содержащуюся в DLL-файлах, вы можете обнаружить, что библиотечный файл импорта, предоставленный поставщиком, не работает с вашим проектом C++Builder. Если это случилось, вероятно, что библиотека импорта имеет формат Microsoft COFF, а не Borland OMF. Чтобы создать библиотечный файл импорта, который может использоваться C++Builder, просто запустите implib с DLL, входящими в библиотеку.

Утилита implib имеет несколько опций командной строки, которые вы можете использовать при необходимости. Чтобы увидеть полный список опций, введите в командной строке MS-DOS implib -h (опция -h вызывает справку).

touch.exe

touch может использоваться для установки даты и времени файлов. Если вы посмотрите на файлы C++Builder на CD-ROM, вы обнаружите, что все они имеют одинаковые дату и время создания. (Время файла является своего рода кодом, который идентифицирует версию C++Builder на CD-ROM.) Утилиту touch очень легко использовать. В таблице 12.1 перечислены некоторые из опций командной строки этой утилиты.

Таблица 12.1. Некоторые опции командной строки утилиты touch

Опция	Назначение
-h	Вывод справочного экрана. (-h2 выводит более полный справочный экран.)
-s	Повтор по подкаталогам.
-thh:mm:ss	Установить время для указанных файлов. (Второй параметр необязателен.)
-dmm/dd/yy	Установить дату для указанных файлов.

Предположим, например, что вы выпустили версию 2.12 вашего проекта и хотите установить даты всех файлов на текущую дату, а время на 2:12 а.м. Предположим далее, что у вас есть более чем один каталог и вам нужно модифицировать всех файлы во всех каталогах. В этом случае командная строка выглядела бы следующим образом:

```
touch -t02:12 -s *.*
```

Обратите внимание, что в данном случае вы не указываете дату. Если вы не указали опцию -d, то будет использована текущая дата. То же самое верно и для опции -t.

tdump.exe

Утилиту tdump можно отнести к продвинутым инструментам программирования. tdump выводит структуру исполняемого EXE-файла (а также файлов других типов). По умолчанию, утилита tdump выводит информацию на консоль, но ее легко направить и в текстовый файл, который вы можете позже исследовать, чтобы получить информацию о программе. tdump сообщит вам структуру файла, какие DLL использует программа, к каким функциям в этих DLL она обращается, а также другую информацию. Например, ниже приведена часть результатов анализа файла программы, созданной с помощью C++Builder:

```
Portable Executable (PE) File

Header base:  00000200

CPU type                    80386
Flags                       818E [ executable 32bit ]
DLL flags                   0000 [ ]
Linker Version              2.19
Time stamp                  00000000
O/S Version                 1.0
User Version                0.0
Subsystem Version           4.0
Subsystem                   0002 [ Windows GUI ]
Object count                00000000
Symbols offset              00000000
Symbols count               00000000
Optional header size        00E0
Magic #                     10B
Code size                   0003C000
Init Data size              00009000
Uninit Data size            00001000

...(другая информация)...

Imports from OLEAUT32.dll
    SysAllocStringLen
    SysStringLen
    VariantChangeTypeEx
    VariantClear
    VariantCopyInd

Imports from MPR.dll
    WNetGetConnectionA

Imports from USER32.dll
    AdjustWindowRectEx
    BeginPaint
    CallNextHookEx
    CallWindowProcA
    CharLowerA

...(импорт из других DLL) ...
```

Обычно вы будете запускать tdump из командной строки. Поскольку вывод tdump содержит довольно много информации, вы поступите правильно, если перенаправите вывод в текстовый файл следующим образом:

```
tdump MyApp.exe > dump.txt
```

Теперь у вас есть текстовый файл ASCII, который можно просмотреть в редакторе кода C++Builder или в любом другом текстовом редакторе. Хотя, скорее всего, вы будете пользоваться утилитой tdump не очень часто, в некоторых случаях она может оказаться неоценимой.

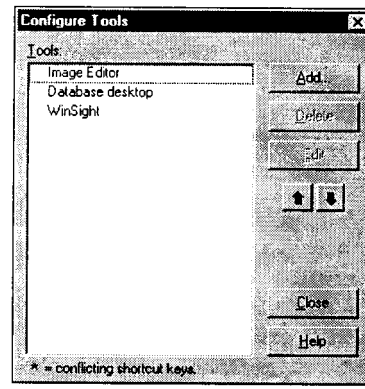
Конфигурирование меню Tools

Конфигурация меню Tools в C++Builder может быть изменена. По умолчанию, меню Tools содержит пункты для рабочего стола базы данных (Database Desktop) и редактора изображений. Вы можете добавить к ним свои собственные инструменты, другие инструменты C++Builder или любую другую программу, которой вы часто пользуетесь. Вы можете также изменить порядок следования пунктов меню Tools или даже удалить пункты совсем.

Диалоговое окно Configure Tools

Почти любое приложение, которым вы часто пользуетесь, может быть добавлено в меню Tools. Добавив пункты в меню Tools, вы можете быстро загрузить любую программу, которую вы часто используете при создании проектов. Добавление инструментов в меню Tools — тривиальная задача и занимает всего несколько минут. Добавление, удаление и перемещение пунктов меню Tools выполняется через диалоговое окно Configure Tools (Конфигурация инструментов). Чтобы вызвать диалоговое окно Configure Tools, выберите в главном меню пункт Tools | Configure (Инструменты | Конфигурация) (См. рис. 12.11.)

Рис. 12.11
Диалоговое окно Configure Tools

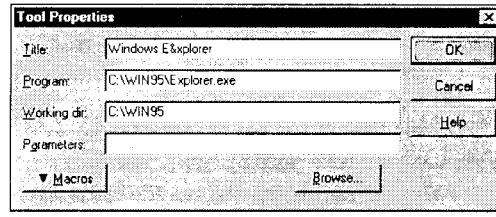


Окно списка с именем Tools показывает текущие инструменты меню Tools. Эти инструменты перечислены в том порядке, в котором они появляются в меню. Чтобы изменить порядок следования пунктов списка, просто щелкните на инструменте, который вы хотите переместить, и щелкните либо на кнопке со стрелкой вверх, либо на кнопке со стрелкой вниз. Чтобы удалить

данный инструмент из меню Tools, выберите его в окне списка и нажмите кнопку Delete (Удалить). Инструмент будет удален.

Чтобы добавить инструмент в меню Tools, нажмите кнопку Add. Появится диалоговое окно Tool Properties (Свойства инструмента), где вы можете ввести свойства добавляемого инструмента. Диалоговое окно Tool Properties со свойствами программы Windows Explorer показано на рис. 12.12.

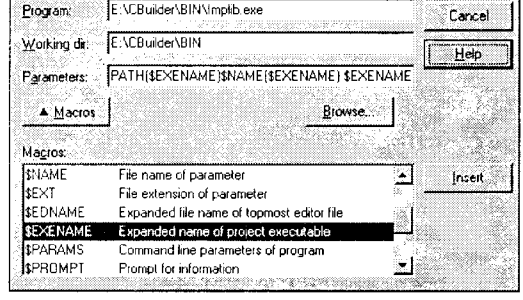
Рис. 12.12
Диалоговое окно Tool Properties



В поле Title вы вводите то название инструмента, которое появится в меню Tools. Вы можете использовать амперсанд, чтобы подчеркнуть символ для быстрого вызова данного пункта. В поле Program вводится путь и имя файла инструмента. Чтобы программа была найдена наверняка, вы должны ввести полный путь. Если вы не знаете точный путь и имя файла, можно нажать кнопку Browse и найти файл самостоятельно. В поле Working dir вводится рабочий каталог инструмента. Если вы использовали кнопку просмотра (Browse) для поиска инструмента, рабочим каталогом автоматически станет каталог, где постоянно находится программа.

Поле Parameters требует некоторых пояснений. В самой простой форме, это поле используется для задания параметров командной строки инструмента. Просто введите здесь любые параметры, необходимые для выполнения программы. В этом поле вы также можете использовать предопределенные макросы C++Builder. Если вы нажмете кнопку Macro в диалоговом окне Tool Properties, то получите список макросов, из которых можете выбрать то, что вам нужно. Чтобы использовать макрос, введите его имя непосредственно или выберите из списка и нажмите кнопку Insert. На рис. 12.13 показано диалоговое окно Tool Properties со списком макросов.

Рис. 12.13
Диалоговое окно Tool Properties с отображенными макросами



Макрос $EXENAME возвращает полный путь исполняемого файла проекта. Например, если ваш проект назывался MyApp и находился в каталоге Projects, параметр $EXENAME передаст программе строку

c:\Projects\MyApp.exe. Некоторые из макросов требуют параметров. Чтобы использовать только путь исполняемой программы вашего проекта без имени файла, вы должны будете определить строку параметров $PATH($EXENAME). В приведенном примере эта комбинация параметров дала бы c:\Projects\. Полный список доступных макросов содержится во встроенной справочной системе C++Builder.

Редактирование инструмента в меню Tools выполняется простым щелчком кнопки Edit диалогового окна Configure Tools. После этого появляется окно Tool Properties, где вы можете при необходимости редактировать любое из полей.

Когда вы закончили добавление, редактирование или реорганизацию инструментов, нажмите кнопку Close. Все изменения будут отражены в меню Tools C++Builder.

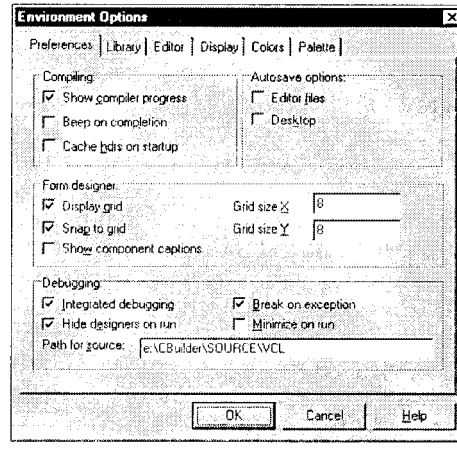
Установка опций среды

Опции среды C++Builder позволяют вам вносить изменения в IDE на глобальном уровне. (Диалоговое окно Project Options управляет установками на уровне проекта.) Для вывода диалогового окна Environment Options, выберите в главном меню пункт Options | Environment (Опции | Среда). Это окно содержит 6 вкладок. Вы познакомились с вкладками Editor, Display и Colors, когда мы рассматривали редактор кода на 10-й день. Здесь мы рассмотрим вкладки Preferences (Общие установки), Library (Библиотека) и Palette (Палитра).

Вкладка Preferences

Во вкладке Preferences диалогового окна Environment Options вы устанавливаете общие параметры IDE C++Builder, определяющие компиляцию, автосохранение и отладку процессов. Здесь вы также можете установить общие параметры редактора форм. (См. рис. 12.14.)

Рис. 12.14
Вкладка общих параметров диалогового окна Environment Options



Раздел Compiling (Компиляция) содержит три опции. Опция Show compiler progress (Показать состояние процесса компиляции) включает/выклю-

чает вывод диалогового окна, показывающего состояние процесса компиляции проекта. Большую часть времени вы будете оставлять эту опцию включенной, чтобы видеть, что происходит с вашим проектом. Выключение этой опции немного ускоряет компиляцию, хотя разница, вероятно, не будет очень существенной. Опция Beep on completion (Звуковой сигнал по завершении) выдает звуковой сигнал через встроенный динамик PC, когда компиляция завершается. Это удобно, если компиляция выполняется длительное время и в фоновом режиме, а вы хотите знать, когда она завершилась. Опция Cache hdrs on startup (Кэшировать заголовочные файлы при запуске) указывает C++Builder, чтобы он загрузил предварительно откомпилированные заголовочные файлы сразу после запуска. Если на вашей системе установлен большой объем RAM, эта опция может уменьшить время компиляции, поскольку заголовки будут находиться в оперативной памяти до начала процесса компиляции. Если эта опция выключена, то заголовочные файлы будут загружаться с диска по мере необходимости. Если ваша система не имеет большого объема RAM (около 48 Мб или более), возможно, что вы не захотите включать эту опцию.

Раздел Autosave (Автосохранение) позволяет вам включить режим автоматического сохранения редактируемых файлов или рабочей кофигурации при каждом выполнении программы. Я не пользуюсь этой опцией, поскольку предпочитаю иметь возможность отказаться от сохранения текущего состояния редактируемых файлов в случае необходимости. Однако я знаю людей, которым нравится эта опция по следующей причине: если ваше приложение плохо себя ведет и нарушает работоспособность IDE C++Builder или Windows, вы знаете, что ваши последние правки были сохранены. Опция Desktop сохраняет текущие размер и расположение окна редактора кода, всех форм, палитры выравнивания, инспектора объектов и т. д.

Назначение раздела Form Designer (Редактор форм) практически очевидно. Опция Display grid (Показать сетку) включает и выключает отображение сетки редактора форм. Она определяет только видимость сетки, а не привязку к ней объектов. Опция Snap to grid (Привязать к сетке) определяет, будут ли помещаемые или перемещаемые компоненты привязываться к узлам сетки. Поля X/Y grid size (шаг сетки по осям X и Y) позволяют вам установить расстояние между узлами сетки. По умолчанию оно составляет 8 пикселов. Опция Show component captions (Показать заголовки компонентов) относится к невизуальным компонентам, помещенным в формы. Когда эта опция включена, редактор форм отображает свойство Name невизуальных компонентов под изображением значка, представляющего данный компонент.

Раздел Debugging (Отладка) управляет отладкой на глобальном уровне. Если опция Integrated debugging (Интегрированная отладка) включена, то при щелчке на кнопке Run используется встроенный отладчик IDE. Когда эта опция выключена, программа выполняется без прогона через отладчик. Опция Hide designers on run (Скрыть редакторы при выполнении) скроет редактор форм, инспектор объектов и другие окна поддержки редактора форм, когда программа запускается кнопкой Run, но главное окно C++Builder и редактор кода остаются видимыми. Опция Minimize on run (Минимизировать при выполнении) работает аналогично. Когда эта опция включена, при отладке программы IDE C++Builder сворачивается.



Когда опция Minimize on run включена, любое действие в программе, которое требует отладки (исключения, контрольные точки и т.д.), приведет к восстановлению прежнего вида IDE C++Builder.

Опция Break on exception (Останов при исключении) управляет действиями отладчика в случаях, когда генерируется исключительная ситуация VCL. (Вы узнаете об исключениях завтра.) Когда эта опция включена, при формировании исключительных ситуаций отладчик приостанавливает выполнение программы. Когда эта опция выключена, исключение VCL будет обработано обычным образом (обычно посредством вывода окна с сообщением, информирующим пользователя о том, какая ошибка произошла в его программе). Поле Path for Source (Путь для исходного кода) указывает каталог, в котором отладчик может найти исходный код VCL. Обычно вы не должны изменять состояние этого поля.

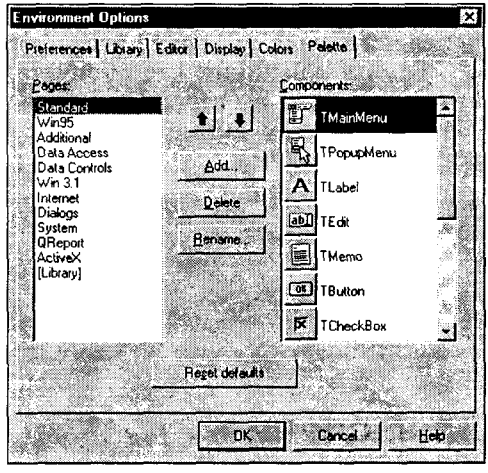
Вкладка Library

Каждый раз, когда вы добавляете компонент в палитру компонентов, C++Builder перестраивает файл с именем cmplib32.lib. Это происходит с любым проектом. Вкладка Library (Библиотека) диалогового окна Environment Options управляет тем, как компилятор и компоновщик ведут себя при построении cmplib32.lib. Опции этой вкладки идентичны опциям, находящимся в диалоговом окне Project Options. Чтобы не повторять все сказанное ранее, я просто предложу вам обратиться к 10-му дню, где рассматривались опции проекта.

Вкладка Palette

Вкладка Palette диалогового окна Environment Options позволяет вам настроить палитру компонентов. Эта вкладка показана на рис. 12.15.

Рис. 12.15
Вкладка Palette диалогового окна Environment Options



Эта вкладка позволяет изменить порядок, в котором появляются вкладки в палитре компонентов. Окно списка Pages в левой части диалогового окна показывает все вкладки, находящиеся в настоящий момент в палитре ком-

понентов. Внизу списка вы видите пункт, озаглавленный [Library]. Этот пункт показывает все установленные компоненты во всех вкладках палитры компонентов. Окно списка Components показывает компоненты, которые появляются во вкладке, выбранной в списке Pages.

Чтобы изменить порядок следования вкладок, просто перетащите вкладку внутри окна списка Pages в положение, которое она будет занимать в палитре компонентов. Например, если вы хотите, чтобы вкладка Samples была первой в палитре компонентов, перетащите ее в верхнюю часть списка в окне Pages. Вы можете также щелкнуть на нужной вкладке и использовать кнопку со стрелкой вверх или вниз, чтобы переместить вкладку на ее новое место.

Вкладки могут быть добавлены, удалены или переименованы, и эти операции выполняются точно так, как вы ожидаете. Чтобы добавить вкладку, просто щелкните на кнопке Add. Появится запрос на ввод имени вкладки. После того как вы введете имя, будет создана новая вкладка. Вы можете добавить вкладку в том случае, если, например, создали свои собственные компоненты и хотите, чтобы они находились все вместе на отдельной вкладке в палитре компонентов. Другая возможная причина создания дополнительной вкладки — ускоренный доступ к тем компонентам VCL, которыми вы часто пользуетесь.



Вы можете перемещать компоненты с одной вкладки на другую, но вы не можете копировать компоненты. Компонент может появляться в палитре компонентов только один раз.

Удаление и переименование вкладок — достаточно простая операция. Как и в репозитории объектов, только свободная от компонентов вкладка может быть удалена.

Компоненты на данной вкладке могут быть реорганизованы точно так же, как и сами вкладки. Чтобы переместить компонент, переместите его в новое положение в окне списка Components или используйте кнопки со стрелками вверх и вниз. Чтобы переместить компонент в новую вкладку, перетащите его на название вкладки в окне списка Pages. Чтобы удалить компонент, сначала выберите его, а затем нажмите кнопку Delete.

Если вы решили, что окончательно испортили палитру компонентов, не отчаивайтесь — вы всегда можете нажать кнопку Restore Defaults. Эта кнопка сбросит палитру компонентов в состояние, которое она имела сразу после установки системы.



Кнопка Restore Defaults (Вернуть в состояние по умолчанию) сбросит палитру компонентов в состояние, в котором она находилась после первоначальной установки C++Builder. Это не означает, что конфигурация палитры компонентов вернется к последнему состоянию, в котором вы ее оставили.

Заключение

Итак, теперь вы знаете немного больше об инструментах, которые включены в пакет C++Builder. Некоторые из этих инструментов просты и на них можно не обратить внимания, так что эта глава поможет вам составить

целостную картину того, что вам доступно. Некоторые из инструментов вы начнете использовать не сразу, но через какое-то время вы можете вернуться и посмотреть еще раз на доступные вам средства. Мы закончили день, рассмотрев диалоговое окно Environment Options. По большей части вы можете оставить его в состоянии по умолчанию. Но когда вам потребуется настроить IDE, вы будете лучше понимать то, для чего предназначена та или иная опция.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Я создаю значок и пробую получить эллипс с черным контуром и желтой заливкой. Я установил черный цвет переднего плана и желтый цвет фона, но все, чего я добился — это круг черного цвета. Что я должен сделать, чтобы нарисовать то, что мне нужно?

В этом отношении редактор изображений отличается от некоторых растровых редакторов. Чтобы нарисовать нужный вам эллипс, сначала нарисуйте черным цветом незакрашенный эллипс, а затем залейте его желтым цветом.

Каким образом в редакторе изображений рисуются правильные окружности или прямые линии?

Чтобы нарисовать правильную окружность или прямую линию, во время рисования удерживайте нажатой клавишу Shift. Тот же самый способ применяется и для рисования правильного квадрата.

Я создал свой собственный курсор в виде стрелки, но, когда я выполняю щелчок, то не могу щелкнуть его острием. В чем заключается проблема?

Вам нужно задать для курсора активную точку. По умолчанию, активная точка находится в правом нижнем углу. Отредактируйте курсор в редакторе изображений, установив активную точку на острие стрелки.

Когда я создаю значок, обязательно ли создавать обе версии значка (большую и малую)?

В этом нет необходимости. Если вы создадите только большой значок, Windows сократит его размеры, когда понадобится уменьшенная версия значка. Если вам не понравился внешний вид вашего значка после того, как Windows его уменьшила, вы можете разработать малый значок вместе с большим. Если малый значок существует, Windows будет его использовать.

Как мне скопировать часть растрового изображения в новое растровое изображение?

Откройте в редакторе изображений исходное растровое изображение в первом окне, а новое — во втором. Растяните прямоугольную область выделения вокруг выбранной части исходного изображения, которую вы

хотите скопировать, и выберите в меню редактора пункт Edit | Copy. Перейдите в окно нового изображения и выберите Edit | Paste. Растровое изображение из буфера обмена будет вставлено в новое изображение.

Мне кажется, что WinSight работает довольно медленно на моей системе. Нормально ли это?

Да, это вполне возможно. Утилита WinSight должна контролировать все, что происходит в Windows, а это требует много процессорного времени. Конечно, на менее производительных системах замедление будет более заметным. Чтобы уменьшить этот эффект в максимально возможной степени, убедитесь, что в данный момент открыты только WinSight и окно, за которым вы следите.

Я ищу определенное сообщение в своей программе, но другие сообщения «забивают» окно трассировки так, что я не могу найти среди них нужное. Что мне делать в такой ситуации?

Две вещи. Во-первых, убедитесь в том, что вы следите за сообщениями только одного определенного окна, которое вас интересует. Выберите это окно в дереве окон, после чего выберите в главном меню пункт Messages | Selected Windows. Во-вторых, вы можете вызвать диалоговое окно Trace Options и выключить все сообщения, кроме тех, которые вас интересуют.

Мне нужно, чтобы некоторые компоненты в палитре компонентов находились как в своих оригинальных вкладках, так в одной общей вкладке, которую я создал. Могу ли я это сделать?

Нет. Компоненты в палитре компонентов могут находиться только в одном месте.

Вопросы

1. Для чего используется прозрачный цвет в значках и курсорах?
2. Как выбрать цвет в редакторе изображений?
3. Как выбрать область растрового изображения для операций вырезать/копировать?
4. Какое максимальное количество цветов может иметь растровое изображение в редакторе изображений?
5. Что такое активная точка курсора?
6. Может ли утилита WinSight наблюдать за скрытым окном?
7. Укажите самый быстрый путь поиска окна в дереве окон.
8. Для чего используется утилита grep?
9. Каким образом сделать так, чтобы файлы, находящиеся в настоящий момент в редакторах, автоматически сохранялись при каждом прогоне вашей программы через отладчик?
10. Каким образом можно вернуть палитру компонентов в состояние по умолчанию?

Упражнения

1. Используя редактор изображений, создайте растровое изображение, которое могло бы использоваться в качестве экрана-заставки для вашей компании или вашего конкретного приложения.
2. Используя редактор изображений, создайте большую и малую версию значка, который имеет прозрачный фон.
3. Создайте ресурс курсора в редакторе изображений. Обязательно установите активную точку и проверьте работу курсора в Cursor Tester.
4. Создайте новый файл ресурсов. Создайте два растровых изображения и курсор. Сохраните файл.
5. Используя файл ресурсов, созданный в упражнении 4, создайте новый ресурс значка и скопируйте значок, созданный в упражнении 2, в этот новый значок.
6. Создайте новую программу в C++Builder. Измените ее свойство Name на Test, а свойство Caption на Test Window. Запустите программу. Теперь запустите WinSight. Найдите программу Test Window и установите трассировку сообщений в состояние Selected Windows. Расположите окна WinSight и Test Window рядом. Выберите пункт Start! в главном меню WinSight. Перейдите в окно Test Window, измените его размеры, щелкните на нем, сверните и т.д. Понаблюдайте за сообщениями в окне трассировки.
7. Продолжая работать над упражнением 6, перейдите в WinSight, выберите в главном меню пункт Messages | Options и выключите все сообщения за исключением группы сообщений Mouse. Еще раз поэкспериментируйте с программой Test Window. Понаблюдайте за сообщениями в окне трассировки.
8. Откройте диалоговое окно Environment Options. Перейдите на вкладку Palette. Создайте новую вкладку в палитре компонентов. Назовите ее MyStuff. Переместите компоненты Label, Image и OpenDialog в новую вкладку. Закройте диалоговое окно Environment Options. Проверьте новый вид палитры компонентов.
9. Продолжая работать над упражнением 8, верните палитру компонентов в состояние по умолчанию.
10. Загрузите программу в C++Builder. Откройте диалоговое окно Environment Options. Во вкладке Preferences включите опцию Minimize on Run. Нажмите кнопку Run, чтобы запустить программу. Обратите внимание на поведение IDE C++Builder.

После изучения основ



Кент Рейсдорф

Сегодня вы узнаете о некоторых вещах, которые помогают превратить хорошее приложение Windows в очень хорошее приложение. Конкретно, мы рассмотрим следующее:

- Оформление окон: оперативные панели и строки состояния
- Управление доступом к командам
- Печать в приложениях C++Builder
- Использование курсоров

Завтра мы продолжим эту тему, когда будем рассматривать реализацию в приложениях C++Builder более сложных аспектов программирования для Windows.

Декоративные элементы окна

Конечно, речь пойдет не о красивых гирляндах вокруг фасадного окна вашего дома. Я имею в виду такие элементы, как оперативные панели и строки состояния. Все это, вместе взятое, часто называют *декорациями окна (window decorations)*. В данном разделе рассматриваются именно эти типы декоративных элементов и способы их реализации в ваших приложениях.

Оперативные панели

Оперативная панель (speedbar), часто называемая также панелью инструментов (toolbar) или панелью управления (control bar) — это почти стандартный элемент современных программ Windows. Пользователи ожидают определенных удобств от интерфейса программы, и оперативная панель является одним из них. Вы уже знаете практически все об оперативных панелях, но давайте повторим еще раз. Итак, оперативная панель может содержать следующие элементы:

- Кнопки, дублирующие некоторые пункты меню приложения.
- Всплывающие подсказки, разъясняющие назначение кнопки. Подсказки появляются, когда пользователь задерживает курсор мыши над кнопкой.

- Дополнительные указания, которые выводятся в строке состояния приложения.
- Другие элементы управления, например, комбинированные списки.

В оперативной панели могут быть реализованы:

- Разрешение и запрещение работы кнопок, когда это требуется по логике работы программы.
- Динамическое добавление и удаление кнопок в зависимости от контекста.
- Необязательная возможность стыковки.

Очевидно, что не все из перечисленного является обязательным для каждой оперативной панели. Большинство этих возможностей очень просто реализовать с помощью C++Builder, однако создание стыкуемой оперативной панели — это нетривиальная задача, и мы не будем пытаться решить ее сегодня. Управление доступом к командам мы рассмотрим немного позже в соответствующем разделе этой главы, и я пока воздержусь от разговора на эту тему.

Обзор оперативной панели

Как я упоминал в 9-й день, «Создание приложений в C++Builder», самый простой способ создания оперативной панели состоит в том, чтобы использовать эксперт приложений. Это можно сделать даже тогда, когда ваше приложение частично написано. Просто выполните генерацию приложения с экспертом приложений, скопируйте оперативную панель в буфер обмена, вновь откройте свое исходное приложение (не беспокойтесь о сохранении приложения, созданного экспертом) и вставьте в него оперативную панель из буфера. Быстро и просто.

Однако эксперт приложений не является универсальным средством, поэтому иногда может возникнуть необходимость создать оперативную панель самостоятельно с самого начала (или, например, вы привыкли проверять все от корки до корки!). Мы делали это и прежде, но здесь, тем не менее, приведены шаги, которые требуется выполнить, чтобы создать оперативную панель (шаг 1 — необязателен, но я думаю, что с ним панель будет выглядеть лучше):

1. Поместите в форму компонент Bevel. Установите для его свойства Height значение 2, а для свойства Align значение alTop.
2. Поместите в форму компонент Panel. Установите для его свойства Height значение 33, для свойства Align значение alTop и для свойства BevelOuter значение alNone. Удалите заголовок, очистив поле свойства Caption. Измените значение свойства Name на что-нибудь значимое (обычное имя — SpeedBar).
3. Разместите кнопки на панели и установите значения их свойств Glyph и Name соответствующим образом. Если вы планируете использовать подсказки для кнопок, установите свойство ShowHint в true и введите текст подсказки в качестве значения свойства Hint.
4. Выровняйте каждый элемент управления, используя клавиши Ctrl+стрелка. При необходимости добавьте разделители для групп кнопок.
5. Свяжите событие OnClick каждой кнопки с обработчиком события OnClick соответствующего пункта меню.

По существу это все. Из 9-го дня вы могли вспомнить, что нам осталось выполнить еще несколько действий, чтобы текст подсказок выводился в строку состояния, но в основном оперативная панель уже функционирует.



В этом упражнении подразумевается, что меню вашего приложения уже создано. Существует хорошая практика размещать на оперативной панели только те кнопки, которым соответствуют пункты меню. Оперативная панель служит альтернативой меню. Поэтому здесь не должны содержаться элементы, которые нельзя найти в другом месте программы. (Для любого правила есть, разумеется, и исключения.)

Добавление и удаление кнопок оперативной панели

В оперативной панели Windows-приложения высокого класса обычно присутствуют только те кнопки, которые соответствуют текущему состоянию программы. Например, любая программа, которая имеет дело с документами, обычно отображает оперативную панель с минимальным набором функций, если ни один документ еще не был открыт. В таких случаях достаточно показать только кнопки File Open, File New, Help и About. После открытия документа будут показаны все остальные кнопки. Достижение этого эффекта потребовало бы некоторой работы при использовании традиционных инструментов программирования на Си++. В C++Builder все делается гораздо проще.

Для динамического изменения состава оперативной панели существует множество способов. Один из них заключается в определении двух массивов кнопок быстрого доступа. Первый массив содержит все доступные кнопки, а другой — только те, которые должны быть показаны в данный момент. Позиции кнопок должны каждый раз вычисляться заново, чтобы они всегда располагались рядом. Чтобы упростить пользователю работу, можно еще больше усложнить задачу, объединив кнопки в группы. Это означает, что кнопки не могут быть просто размещены рядом. Массив ваших кнопок мог бы еще содержать нули в тех местах, где должны находиться разделители... Как видите, задача усложняется все больше и больше.

Но не отчаивайтесь! Существует и более простой способ. Чтобы упростить задачу, вы можете воспользоваться преимуществами компонента Notebook (Записная книжка). Notebook — это компонент-контейнер, который разработан для хранения других компонентов. В этом отношении он подобен компоненту Panel. Однако между ними есть и различие, которое заключается в том, что записная книжка может иметь много страниц. Во время разработки вы размещаете различные группы компонентов на страницах записной книжки, а при выполнении программы переключаете страницы, чтобы отобразить нужную в данный момент группу.

Как вы уже поняли, мы можем использовать компонент Notebook для переключения между оперативными панелями. Достаточно поместить записную книжку в форму и создать отдельную страницу для каждой оперативной панели. Затем на страницах записной книжки мы разместим наборы кнопок быстрого доступа. Начните с пустой формы, а затем шаг за шагом выполните приведенные ниже инструкции.

1. Поместите компонент Notebook в форму (убедитесь, что вы выбрали именно компонент Notebook, а не Tabbed Notebook). Измените значение свойства Height на 33 и установите для свойства Align значение alTop. При этом панель всегда будет располагаться вплотную к верхней границе формы.
2. Найдите свойство Pages в инспекторе объектов и дважды щелкните в столбце Value. Появится диалоговое окно Notebook editor (Редактор записной книжки), показанное на рис. 13.1. Записная книжка всегда имеет страницу с именем Default. Щелкните на кнопке Edit и измените имя этой страницы на ShortSpeedBar (КороткаяОперативнаяПанель).
3. Щелкните на кнопке Add, чтобы создать вторую страницу записной книжки. Назовите эту страницу FullSpeedBar (ПолнаяОперативнаяПанель). После этого нажмите кнопку Close, чтобы закрыть диалоговое окно Notebook editor.
4. В инспекторе объектов измените значение свойства ActivePage на ShortSpeedBar. Добавьте кнопки или другие компоненты в компонент Notebook. Поскольку это всего лишь тест, то не имеет значения, что вы включите в оперативную панель.
5. Убедитесь, что в данный момент выбран компонент Notebook, и измените значение свойства ActivePage на FullSpeedBar. Добавьте другие кнопки или компоненты на эту страницу.
6. Отойдите в сторону и полюбуйтесь на сделанную работу. Модифицируя свойство ActivePage или PageIndex, вы можете чередовать страницы, чтобы редактировать их при необходимости.
7. Напишите код, изменяющий свойство ActivePage или PageIndex во время выполнения программы, чтобы переключать оперативные панели в зависимости от состояния программы.

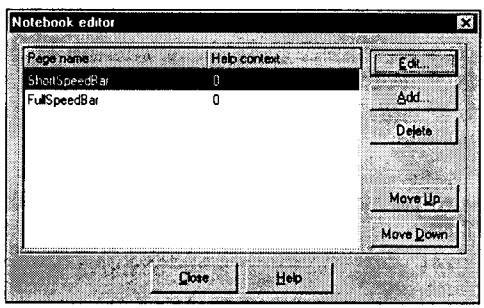


Рис. 13.1
Диалоговое окно Notebook editor

Мы рассмотрели достаточно простое, но эффективное решение проблемы создания и реализации динамических оперативных панелей. Компонент Notebook прост в использовании на этапе разработки, что является большим преимуществом при проектировании несколько панелей.



Когда вы используете записную книжку для переключения оперативных панелей, часть кнопок, вероятно, будет дублироваться на двух или более страницах. Поскольку кнопки быстрого доступа не являются компонентами оконного типа, наличие нескольких дополнительных кнопок не причинит никакого вреда. Иными словами, вы не используете дескриптор окна для каждой кнопки, так что расход ресурсов при таком подходе незначителен.

После настройки записной книжки вы можете легко переключать ее страницы во время выполнения программы. Для этого можно использовать либо свойство ActivePage, либо свойство PageIndex. Следующие две строки кода приводят к одному и тому же эффекту в рассмотренном перед этим примере:

```
Notebook->PageIndex = 0;
Notebook->ActivePage = "ShortSpeedBar";
```



Компакт-диск, прилагаемый к книге, содержит программу SBarTest, которая иллюстрирует рассмотренную методику. Загрузите программу и поработайте с ней, чтобы освоить применение множественных оперативных панелей.

На рис. 13.2 показано окно программы SBarTest с коротким вариантом оперативной панели, а на рис. 13.3 эта панель выведена полностью.

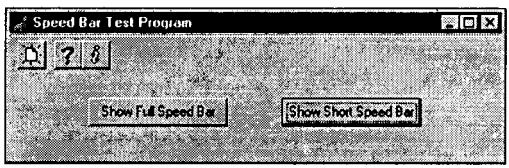


Рис. 13.2
Тестовая программа с короткой оперативной панелью

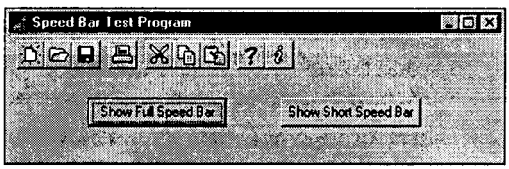


Рис. 13.2
Тестовая программа с полным вариантом оперативной панели

Как вы видите, множественные оперативные панели чрезвычайно просто реализуются в C++Builder. Это превосходный пример, иллюстрирующий возможности визуальной среды программирования. Если вам никогда не приходилось создавать что-либо подобное с использованием только API Windows, то довольно сложно представить себе объем этой работы, но можете поверить мне на слово.

Всплывающие подсказки и указания

Всплывающие подсказки и указания по использованию инструментов также помогут придать вашему приложению законченный вид. Я уже сказал о них практически все, когда мы рассматривали компоненты в 8-й день, «Компоненты VCL», а также в 9-й день, где мы включили поддержку подсказок в программу ScratchPad. Если вам нужно вспомнить, как это делалось, вернитесь к 9-му дню и просмотрите раздел «Добавление функций и членов-данных в исходный код».



Здесь на помощь снова приходит эксперт приложений. Создайте с его помощью оболочку приложения и включите опцию, разрешающую использование подсказок. Эксперт приложений сгенерирует соответствующий код, а вам останется только включить свойство ShowHint и ввести текст подсказки.

Существует одна вещь, о которой я не говорил прежде, а именно, изменение свойств подсказки инструмента. Класс TApplication содержит четыре свойства, которые управляют поведением всплывающих подсказок. В таблице 13.1 приведен список эти свойств и их описания.

Таблица 13.1. Свойства TApplication, относящиеся к всплывающим подсказкам

Свойство	Описание
HintColor	Устанавливает цвет фона окна всплывающей подсказки.
HintHidePause	Устанавливает время (в миллисекундах), по прошествии которого подсказка исчезает, если курсор мыши остается неподвижным, указывая на компонент.
HintPause	Устанавливает интервал времени (в миллисекундах) от момента задержки курсора над компонентом до появления всплывающей подсказки.
HintShortPause	Устанавливает интервал времени между отображениями всплывающих подсказок после того, как одна из них была показана (например, когда пользователь перемещает курсор в пределах группы кнопок оперативной панели).

Установленные для этих свойств значения по умолчанию подходят для большинства приложений. Тем не менее, если вам нужно изменить значение любого из свойств подсказки, у вас есть такая возможность.

Правила работы с оперативными панелями и подсказками

- Не используйте подсказки для тех элементов управления, где подсказка может закрыть нужный пользователю текст. Это относится, в частности, к элементам редактирования и комбинированным спискам. По крайней мере, оставьте пользователю возможность выключить подсказки для таких элементов.
- Делайте всплывающие подсказки краткими и точными.
- Указания, выводимые в строке состояния, должны быть более подробными и содержательными.
- Предоставьте пользователю возможность выключить все подсказки.

Добавление других элементов к оперативной панели

Панель — это настолько универсальный компонент, что не требуется выполнять каких-либо специальных действий для добавления к оперативной панели вашего приложения элементов управления других типов. Наиболее часто в оперативные панели включают комбинированные списки. Они могут использоваться для выбора шрифта, опций конфигурации или установки масштаба — возможностей бесконечно много.

Чтобы добавить какой-либо компонент к оперативной панели, просто поместите его в свою панель SpeedBar, где вы можете обращаться с ним точно так же, как если бы этот компонент был помещен в форму. Если вы никогда не пробовали включить в оперативную панель комбинированный список, используя API Windows, OWL или MFC, то не сможете даже пред-

ставить, сколько работы делает за вас C++Builder. И снова поверьте мне на слово — это очень значительная работа.

Существует множество форм и размеров оперативных панелей, и C++Builder значительно упрощает их создание и реализацию. Работая с C++Builder, вы больше не сможете оправдываться тем, что «это слишком сложно!». Вам даже может понравиться создавать оперативные панели в C++Builder.

Строки состояния

Наличие в приложении строки состояния также делает его более конкурентоспособным. Не все приложения могут извлечь пользу из строки состояния. Но поскольку C++Builder генерирует 32-разрядные приложения, вы можете использовать все достоинства этого 32-разрядного элемента управления Windows. VCL-компонент StatusBar, инкапсулирующий строку состояния Win32, сильно ускоряет создание таких строк. Но сначала позвольте мне перечислить основные свойства компонента StatusBar. (См. таблицу 13.2.)

Таблица 13.2. Свойства компонента StatusBar

Свойство	Описание
Panels	Используется в строках состояния с несколькими панелями. Это свойство определяет отдельные панели.
SimplePanel	Определяет, будет ли в строке состояния отображаться простая панель или несколько панелей.
SimpleText	Содержит текст простой панели строки состояния.
SizeGrip	Определяет, будет ли строка состояния иметь захват изменения размера окна в правом нижнем углу. Захват изменения размера очерчивает область, за которую пользователь может перетаскивать угол окна, меняя его размер. Отсутствие захвата не означает, что размер окна нельзя изменить, но его наличие упрощает эту операцию.

Как видно из таблицы, строка состояния может быть простой или состоять из нескольких панелей. Давайте рассмотрим эти варианты.

Простая или составная?

Строка состояния может быть простой или составной. *Простая строка состояния (simple status bar)* имеет единственную панель, которая занимает всю строку. Если вы хотите создать простую строку состояния, установите для свойства SimplePanel значение true.



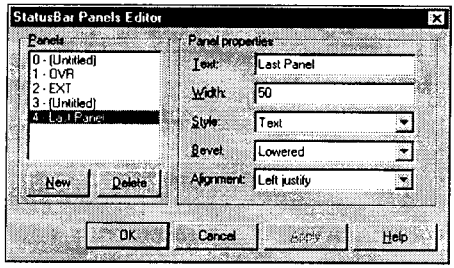
По умолчанию свойство SimplePanel компонента StatusBar установлено в true, поэтому для простой строки вам не нужно его специально устанавливать.

Свойство SimplePanel действует как переключатель. Вы можете переключаться между простой или составной строкой состояния во время выполнения программы, устанавливая SimplePanel в true или false, соответственно.

Составная строка состояния (complex status bar) имеет несколько панелей. Если вы решили использовать составную строку состояния, то можете настроить ее панели в редакторе панелей строки состояния (StatusBar Panels

Editor), который вызывается двойным щелчком в столбце Value свойства Panels. На рис. 13.4 показано окно редактора панелей.

Рис. 13.4
Редактор панелей строки состояния



Текущие панели отображены в окне списка Panels в левой части диалогового окна. Чтобы добавить панель, заполните поля Panel Properties в правой части диалогового окна и нажмите кнопку Add. В окне списка Panels появится новая панель. Чтобы удалить панель, выберите ее в списке Panels и нажмите кнопку Delete. Чтобы отредактировать существующую панель, выберите ее и измените свойства необходимым образом.



Отдельные панели составной строки состояния являются экземплярами класса TStatusPanel.

Назначение большинства свойств очевидно, но два из них требуют некоторых пояснений. Свойство Text содержит текст, который будет отображен на панели. Это свойство может также использоваться для изменения текста панели во время выполнения программы. Установка текста строки состояния рассматривается несколько позже; вам не нужно вводить текст в панель на этапе разработки, если вы собираетесь менять его по ходу программы.

Свойство Style может иметь значения Text или Owner Draw. (Когда вы выбираете один из этих вариантов, редактор панелей строки состояния устанавливает для свойства Style значение psText или psOwnerDraw, соответственно.) Если для Style установлено значение Text, поведение панели достаточно очевидно. Текст выравнивается в зависимости от значения свойства Alignment. Если для Style установлено значение Owner Draw, то создание текста или изображения для вывода на панель является полностью вашим делом. Рисование в панели пользователя (owner-drawn panel) обсуждается немного позже в разделе «Пользовательские панели строки состояния».

Свойства панели Width, Bevel и Alignment говорят сами за себя. Поэкспериментируйте с ними, чтобы увидеть их влияние на внешний вид строки состояния.



Результат изменений, сделанных вами в строке состояния с помощью редактора панелей, можно сразу увидеть в редакторе форм. Расположите редактор панелей так, чтобы во время работы в нем вы могли видеть строку состояния. Каждое внесенное изменение будет отражено в редакторе форм сразу после перехода в следующее поле редактора панелей или нажатия кнопки Apply.



На прилагаемом к книге компакт-диске содержится программа StatTest, которая демонстрирует некоторые способы оформления строк состояния. Выполните программу и исследуйте ее исходный код на

предмет готовых решений, которые вы сможете использовать в строках состояния своих приложений.

Когда вы закончите добавлять панели в строку состояния, нажмите OK, чтобы вернуться в редактор форм.



Когда вы модифицируете свойство Panels компонента StatusBar, редактор форм автоматически устанавливает свойство SimplePanel в состояние false, предполагая, что вам не нужна простая строка состояния, если вы используете несколько панелей.

Изменение текста в строке состояния

Существуют два способа изменить текст в строке состояния:

- Вручную модифицировать свойство SimpleText (для простых строк) или свойство Text отдельной панели (для составных строк).
- Отреагировать на событие onHint объекта Application.

Ручное изменение текста в строке состояния выполняется очень просто, особенно если вы имеете дело с простой строкой. Когда свойство SimplePanel имеет значение true, вы просто устанавливаете в качестве значения свойства SimpleText тот текст, который должен выводиться в строку состояния:

```
StatusBar->SimpleText = "This shows up in the status bar.";
```

Изменить текст в составной строке состояния немного сложнее. Если вы хотите изменить текст в первой панели составной строки, то должны написать код следующего вида:

```
StatusBar->Panels->Items[0]->Text = "Status Bar Text";
```

Хотя эта строка немного растянулась, она не слишком сложна для понимания. Свойство Panels компонента StatusBar имеет свойство, названное Items, которое представляет собой массив панелей строки состояния. Установка свойства Text элемента массива изменяет текст данной панели. Как вы видите, данный массив имеет базовый индекс 0. Т.е., первая панель в строке состояния — это элемент массива с индексом 0.

Изменение текста через событие OnHint требует большей работы, но и эта операция становится довольно простой после того, как вы выполните ее один раз. Вспомните, как на 9-й день мы добавили поддержку указаний в строке состояния к программе ScratchPad. Нам не нужно проходить все это снова, но давайте повторим те шаги, которые мы выполняли:

1. Включите в заголовочный файл главной формы объявление функции для своего собственного события OnHint:

   ```
   void __fastcall OnHint(TObject* Sender);
   ```

2. Добавьте функцию в модуль исходного кода главной формы. В теле функции установите свойство Hint объекта Application в качестве значения свойства SimpleText строки состояния:

   ```
   void __fastcall TMainForm::OnHint(TObject* Sender);
   {
     StatusBar->SimpleText = Application->Hint;
   }
   ```

3. В обработчике события OnCreate главной формы укажите свою собственную функцию OnHint() в качестве обработчика события OnHint объекта Application:

```
void __fastcall TMainForm::FormCreate(TObject* Sender)
{
  Application->OnHint = &OnHint;
}
```

4. Убедитесь в том, что свойства ShowHint всех компонентов, которые должны отображать указания, установлены в true, а в свойствах Hint задан правильный текст.

Все это достаточно легко для простых строк состояния, но если ваше приложение использует составные строки, придется немного изменить функцию OnHint():

```
void __fastcall TMainForm::FormCreate(TObject* Sender)
{
  StatusBar->Panels->Items[0]->Text = Application->Hint;
}
```

Чтобы шагнуть еще дальше, предположим, что ваше приложение переключает простую и составную строки состояния в зависимости от текущего состояния программы. В этом случае вы должны проверить, какая строка состояния, простая или составная, отображается в данный момент, и в зависимости от результата выбрать соответствующий текст. Приведенная ниже функция OnHint() решает эту задачу:

```
void __fastcall TMainForm::OnHint(TObject* Sender)
{
  if (StatusBar->SimplePanel)
    StatusBar->SimpleText = Application->Hint;
  else
    StatusBar->Panels->Items[0]->Text = Application->Hint;
}
```

Теперь не имеет значения, какой тип строки состояния вы используете. Конечно, здесь подразумевается, что указания выводятся в первой панели строки состояния. Если вы используете с этой целью какие-либо другие панели, перепишите этот код соответствующим образом.



Даже при использовании подсказок вы можете вручную изменить текст строки состояния. Вам ничто не помешает это сделать, но не забывайте, что текст будет заменен при следующем событии OnHint.

Пользовательские панели строки состояния

Ранее я говорил, что панель может быть либо текстовой (psText), либо нарисованной пользователем (psOwnerDraw). Если для панели установлен стиль psOwnerDraw, то на вас ложится ответственность за все, что будет отображено в этой панели. Маловероятно, что вы собираетесь создавать себе трудности, выводя в пользовательскую панель обычный текст. Панель пользователя обычно нужна для вывода в строку состояния какого-либо значка

или растрового изображения. Независимо от того, что будет нарисовано на панели, последовательность ее создания остается одной и той же.

1. Установите для свойства Style панели значение psOwnerDraw (обычно через редактор панелей строки состояния).
2. Напишите обработчик события OnDrawPanel.

Очевидно, что в данном случае реальная работа начинается при создании обработчика события OnDrawPanel. Заготовка функции обработчика события OnDrawPanel выглядит следующим образом:

```
void __fastcall TMainForm::StatusBarDrawPanel(TStatusBar* StatusBar,
     TStatusPanel* Panel, const TRect& Rect)
{
}
```

Параметр StatusBar этой функции является указателем на строку состояния. Как правило, у вас уже есть указатель на строку состояния (свойство Name компонента StatusBar), так что от этого параметра не так много пользы. Свойство Panel является указателем на конкретную панель, которую необходимо нарисовать в данный момент. Вы можете использовать этот параметр, чтобы определить, какая панель должна быть выведена, если ваша строка состояния имеет более одной панели пользователя. Параметр Rect содержит размер и положение панели. Этот параметр важен, так как фактически он сообщает вам точные размеры области для рисования.

Обработчик события OnDrawPanel вызывается один раз для каждой панели, свойство Style которой имеет значение psOwnerDraw. Если вам нужно рисовать только на одной панели, то реально необходим лишь параметр Rect. Если же вы выводите изображения на несколько панелей, то сначала вам нужно определить, на какой панели рисовать, и только потом начинать вывод изображения. Разобраться с этим вам поможет конкретный пример.

Ранее я упоминал о программе StatTest. На рис. 13.5 показано ее окно во время работы.

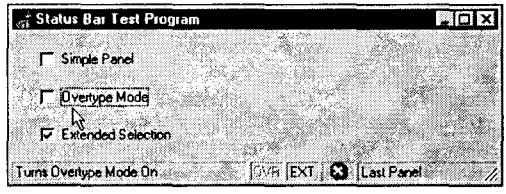


Рис. 13.5
Программа StatTest с пользовательскими панелями строки состояния



Как видите, строка состояния в этой программе имеет несколько панелей. Из них три средние — это панели пользователя. Панели с метками OVR и EXT моделируют строку состояния программы обработки текстов или редактора кода. В таких программах режимы Overtype или Extended Selection могут находиться в состоянии включено/выключено. Если режим включен, текст в панели строки состояния выводится черным цветом. Если режим выключен, текст принимает вид трехмерного текста в недоступном состоянии. Третья панель пользователя содержит один из стандартных значков Windows для иллюстрации вывода графики в строку состояния. Загрузите эту программу с CD-ROM и поэкспериментируйте с ней, чтобы увидеть, как она работает.

Листинг 13.1 содержит обработчик события OnDrawPanel из программы StatTest. С помощью комментариев разберитесь, как он работает.

Листинг 13.1. Функция StatusBarDrawPanel() программы StatTest

```
 1: void __fastcall TMainForm::StatusBarDrawPanel(TStatusBar *StatusBar,
 2:       TStatusPanel *Panel, const TRect &Rect)
 3: {
 4:   //
 5:   // Создание ссылки на холст (Canvas) строки состояния, чтобы
 6:   // не вводить слишком много текста
 7:   //
 8:   TCanvas& c = *StatusBar->Canvas;
 9:   //
10:   // Создание временного объекта TRect. Параметр Rect является
11:   // константой, поэтому мы не можем его изменять
12:   //
13:   TRect temp = Rect;
14:   //
15:   // Проверка: является ли панель 3 той панелью, которую
16:   // нужно нарисовать. Если да, вывод значка на панель
17:   //
18:   if (Panel->Index == 3) {
19:     //
20:     // Загрузка одного из стандартных значков Windows. Здесь
21:     // проще использовать API, чем VCL.
22:     //
23:     HICON icon = LoadIcon(NULL, IDI_HAND);
24:     //
25:     // Вывод значка и сжатие его до размера 15 x 15 пикселов.
26:     // Центровка значка в панели
27:     //
28:     DrawIconEx(c.Handle, Rect.Left + 6, 3, icon,
29:       15, NULL, NULL, DI_NORMAL);
30:     //
31:     // Больше делать нечего
32:     //
33:     return;
34:   }
35:   //
36:   // Этот довольно длинный условный оператор проверяет,
37:   // установлены ли флажки Overtype Mode или Extended Selection.
38:   // Если да, то текст нужно нарисовать дважды:
39:   // сначала белым, затем серым цветом со сдвигом
40:   // на 1 пиксел. В результате получатся объемный текст,
41:   // выглядящий как недоступный
42:   //
43:   if ((Panel->Index == 1 && OvrMode->Checked == false) ||
44:     (Panel->Index == 2 && ExtendedSel->Checked == false)) {
45:     //
46:     // Сдвиг влево и вниз на 1 пиксел
47:     //
48:     temp.Left += 1;
49:     temp.Top += 2;
50:     //
51:     // Изменение цвета текста на белый
52:     //
53:     c.Font->Color = clWhite;
54:     //
```

```
55:      // Установка режима прозрачного фона так, чтобы
56:      // белый текст был виден под серым текстом и
57:      // текст выглядел вдавленным
58:      //
59:      c.Brush->Style = bsClear;
60:      //
61:      // Рисование текста с помощью функции API DrawText().
62:      // Я использую DrawText(), поскольку она позволяет
63:      // центрировать текст как горизонтально, так и вертикально
64:      // в пределах заданного прямоугольника
65:      //
66:      DrawText(c.Handle, Panel->Text.c_str(),
67:        -1, (RECT*)&temp, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
68:      //
69:      // Изменение цвета на серый, поскольку мы собираемся
70:      // нарисовать в данный момент серый текст
71:      //
72:      c.Font->Color = clGray;
73:      //
74:      // Восстановление первоначальных размеров rect
75:      //
76:      temp.Left -= 1;
77:      temp.Top -= 2;
78:    }
79:    //
80:    // Вывод текста. Если режим включен, то для рисования
81:    // текста используется цвет по умолчанию (черный).
82:    // Если режим выключен, то в предыдущем коде цвет
83:    // текста уже установлен на серый
84:    //
85:    DrawText(c.Handle, Panel->Text.c_str(),
86:      -1, (RECT*)&temp, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
87: }
```

Приведенный код может показаться немного устрашающим, но большая его часть — это строки комментариев. Сам код довольно прост. Как объяснено в комментариях, объемный эффект достигается следующим образом: сначала текст рисуется белым цветом, а затем еще раз серым цветом и с небольшим сдвигом. В результате текст выглядит выдавленным на поверхности. Отображение значка осуществляется с помощью функции API Windows DrawIconEx().



Обратите внимание на строку 8 листинга 13.1. Здесь я создал ссылку на свойство Canvas строки состояния, чтобы сэкономить на вводе и сделать код более лаконичным. После создания ссылки, я могу записать код следующим образом:

```
c.Font->Color = clWhite;
```

в отличие от более растянутого варианта:

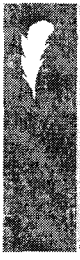
```
StatusBar->Canvas->Font->Color = clWhite;
```

Не спорю, что символ с в качестве имени ссылки несколько озадачивает, но это является иллюстрацией к тому, как можно использовать ссылку в приложении VCL.

Работа с пользовательскими панелями строки состояния сначала требует некоторых усилий, но вы скоро обнаружите, что не все так плохо. Вы можете программировать в течение длительного времени и ни разу не использовать эти панели. Но если когда-нибудь они вам понадобятся, то вы будете знать, что эта задача является вполне разрешимой.

Панели инструментов

Панели инструментов, строго говоря, не относятся к декоративным элементам окна, но они попадают в рамки той же темы, поэтому я кратко их коснусь. *Панель инструментов (toolbox)* — это просто вторичная форма, которая содержит кнопки быстрого доступа. Вы могли заметить, что такие панели инструментов, как палитра выравнивания и инспектор объектов C++Builder, имеют меньшие размеры строки заголовка, чем обычные окна. Такие строки заголовка создаются установкой для свойства BorderStyle значения bsSizeToolWindow (для панелей инструментов с изменяемыми размерами) или bsToolWindow (для панелей инструментов с фиксированными размерами). За исключением этого отличия, панель инструментов представляет собой просто форму другого вида.



У вас может не быть возможности установить ширину панели инструментов во время разработки приложения из-за стандартного размера кнопок, которые выводятся в любую форму. Например, панель инструментов с двумя кнопками, установленными по горизонтали, имеет ширину приблизительно 54 пиксела. Наименьший размер формы, которую можно создать в редакторе форм, составляет 112 пикселов. Единственное, что можно сделать для решения этой проблемы — установить свойство Width панели инструментов перед ее отображением во время выполнения программы.

Управление доступом к командам

Управление доступом к командам (command enabling) — это разрешение или запрещение доступа к кнопкам или пунктам меню в зависимости от текущего состояния программы. Например, если в текстовом редакторе будет разрешен доступ к кнопкам Cut или Copy (или к соответствующим пунктам меню) в то время, когда в нем не выделен текст, это не принесет никакой пользы. Точно так же, если в буфере обмена не содержится какого-либо текста, кнопку Paste следует заблокировать.

Управление доступом к командам не очень сложно, но требует времени. От вас потребуется уделять внимание некоторым деталям.

Проблемы управления доступом к командам

Вы можете управлять доступом к командам в любое время и любым способом. Чтобы разрешить доступ к пункту меню или кнопке оперативной панели, нужно просто установить для свойства Enabled соответствующего компонента значение true. Чтобы запретить доступ к компоненту, измените значение свойства Enabled на false. Само по себе разрешение или запрещение

доступа к компоненту действительно является простой операцией. Основная же работа состоит в том, чтобы создать код, который будет определять, когда и как запрещать или разрешать доступ. Чтобы перевести разговор в практическое русло, вернемся к программе ScratchPad. Мы доработаем ее, добавив управление доступом для пунктов меню Cut, Copy и Paste. (Я сказал для пунктов меню, имея в виду все элементы управления программой: пункты главного меню, кнопки оперативной панели и пункты контекстного меню.)

Прежде, чем мы начнем писать код, который управляет доступом к командам, давайте сначала определим те идеи, которые будут лежать в его основе. В первую очередь примем, что пункты меню Cut и Copy должны быть доступны только в том случае, если в компоненте memo выделен текст (если ничего не выделено, то нечего и вырезать/копировать). Что касается вставки, то нужно сделать так, чтобы доступ к пункту меню Paste был разрешен только при наличии текста в буфере обмена.

Итак, теперь у нас есть набор основных правил, так что можно начинать. Да, еще не все. Остается одна вещь, которую мы должны решить: когда и где мы собираемся разрешать или запрещать доступ?

Мы могли бы разрешать доступ к команде Paste, когда пользователь выбирает в меню пункт Cut или Copy. Если вы помните, у нас есть обработчик событий для пунктов меню Cut, Copy и Paste. Так что мы можем попробовать вставить код в него:

```
void __fastcall TScratchPad::EditCutClick(TObject* Sender)
{
  Memo->CutToClipboard();
  EditPaste->Enabled = true;
}
```

Если пользователь выбирает Edit | Cut, мы знаем, что в буфере обмена что-то находится, и можем разрешить доступ к команде Paste. Но когда мы должны *запретить доступ* к Paste? И что произойдет, если пользователь нажмет комбинацию клавиш Ctrl+X, чтобы вырезать текст, а не воспользуется меню? А если пользователь скопирует текст в буфер обмена из другого приложения и захочет вставить его в ScratchPad? Похоже, что у нас появились некоторые проблемы, которые придется как-то решать. Мое правило — «никогда не делай сегодня то, что можно сделать завтра», поэтому давайте отложим на время эту проблему и посмотрим, что мы можем сделать с Cut и Copy.

Команды Cut и Copy должны быть доступны после выделения текста. Это достаточно просто. Если вы посмотрите справочные файлы VCL, то обнаружите, что TMemo имеет свойство SelLength, которое содержит длину выделенного текста в компоненте memo. Таким образом, если значение SelLength больше 0, это означает, что пользователь выделил некоторый текст. Вы, наверное, подумали: «Отлично! Это действительно очень просто!» Боюсь, что я вас разочарую. У нас нет никакого способа узнать, *когда* пользователь выделил текст. События OnTextSelected не существует, так что в этом случае мы не можем воспользоваться уведомляющими событиями. Мы могли бы использовать системный таймер, но это уж очень некрасивое решение. Кроме того, в системе не так много таймеров и нет никакой гарантии, что мы получим один из них в нужный нам момент. Мы могли бы перехватывать сообщения мыши для компонента memo. Это может сработать. Но что, если пользователь выделит текст клавиатурой, а не мышью? Тогда мы должны перехватывать все сообщения мыши и все сообщения клавиатуры и каждый раз проверять значение SelLength. Да,

решение становится все сложнее и сложнее. А ведь мы еще не разобрались с Paste. Должен же быть какой-то лучший способ! К счастью, есть.

К спасительному OnIdle

Когда ваше приложение заканчивает обработку всех накопившихся сообщений, Windows посылает ему сообщение WM_ENTERIDLE. Фактически, тем самым Windows говорит вашей программе, что «у меня нет сейчас для вас никакой работы, так что расслабьтесь и отдохните некоторое время». Когда приложение C++Builder получает сообщение WM_ENTERIDLE, оно инициирует событие OnIdle объекта Application. Вы можете написать обработчик события OnIdle, который можно связать с событием OnIdle объекта Application. Этот обработчик будет вызываться каждый раз, когда происходит событие OnIdle. На практике это делается точно так же, как и раньше, когда вы связывали обработчик с событием OnHint объекта Application. Создайте функцию в классе вашей главной формы, и присвойте ее адрес событию OnIdle объекта Application.

Какая нам от всего этого польза? Пользой является то, что у нас теперь есть функция, которая будет автоматически вызываться с некоторой периодичностью. Мы можем использовать эту функцию для проверки того, выделен ли какой-нибудь текст в компоненте memo. Итак, теперь мы можем разрешать или запрещать доступ к пунктам меню Cut и Copy по мере необходимости. Замечательно. С этим мы разобрались.

Но ведь проблема с пунктом меню Paste остается, не так ли? Однако здесь тоже есть приемлемое решение. В VCL существует класс с именем TClipboard, который инкапсулирует операции с буфером обмена. Одним из методов TClipboard является HasFormat(). Этот метод проверяет, находятся ли в настоящий момент в буфере обмена данные определенного формата. Если это так, то функция HasFormat() возвращает true. Используя этот метод, мы можем определить, содержится ли текст в буфере обмена:

```
if (Clipboard()->HasFormat(CF_TEXT)) PasteBtn->Enabled = true;
```

Если в буфере обмена находится текст, мы разрешаем доступ к кнопке и пункту меню Paste. Если в буфере обмена нет никакого текста, мы запрещаем доступ к пункту Paste. Еще одна глобальная проблема решена.

Использование функции OnIdle() означает, что наш код управления доступом к команде будет вызываться независимо от того, выделяет ли пользователь текст с помощью мыши, клавиатуры или даже в другом приложении.

От теории к практике

Как я уже сказал, мы должны создать обработчик для события OnIdle. Эта часть довольно проста, так что сначала сделаем ее:

1. Откройте программу ScratchPad и перейдите в редактор кода.
2. Откройте заголовочный файл модуля главной формы (SPMain.h) и добавьте в начало файла после всех #include следующую строку:
   ```
   #include <vcl\Clipbrd.hpp>
   ```
3. Добавьте в раздел private объявления класса следующее:
   ```
   void __fastcall OnIdle(TObject* Sender, bool& Done);
   ```

4. Теперь перейдите в редакторе кода к файлу SPMain.cpp. Добавьте в конец файла следующую функцию:

```
void __fastcall TScratchPad::OnIdle(TObject* Sender, bool& Done)
{
}
```

5. Найдите функцию MainFormCreate() и добавьте в нее следующую строку:

```
Application->OnIdle = &OnIdle;
```

Теперь у вас есть функция, которая будет вызываться каждый раз, когда приложение входит в состояние ожидания. Очевидно, что функция не делает ничего полезного, если в ней нет кода, но мы введем его буквально через минуту. А пока добавим код управления доступом к пунктам меню Save, Save As, Undo и Select All. Для этого мы везде используем OnIdle. В листинге 13.2 приведена законченная функция OnIdle().

Листинг 13.2. Функция OnIdle() программы ScratchPad

```
 1: void __fastcall TScratchPad::OnIdle(TObject* Sender, bool& Done)
 2: {
 3:   //
 4:   // Код управления доступом к командам Cut и Copy. Если текст
 5:   // не выделен, то запрещение доступа к этим пунктам в главном
 6:   // меню, оперативной панели и контекстном меню
 7:   //
 8:   bool textSelected = Memo->SelLength > 0;
 9:   EditCut->Enabled = textSelected;
10:   EditCutBtn->Enabled = textSelected;
11:   PopupCut->Enabled = textSelected;
12:   EditCopy->Enabled = textSelected;
13:   EditCopyBtn->Enabled = textSelected;
14:   PopupCopy->Enabled = textSelected;
15:   //
16:   // Код управления доступом к Paste. Проверка наличия текста в
17:   // буфере обмена. Если текста нет, запрещение доступа к Paste
18:   //
19:   bool cbHasText = Clipboard()->HasFormat(CF_TEXT);
20:   EditPaste->Enabled = cbHasText;
21:   EditPasteBtn->Enabled = cbHasText;
22:   PopupPaste->Enabled = cbHasText;
23:   //delete clipBrd;
24:   //
25:   // Код управления доступом к Select All
26:   //
27:   bool hasText = Memo->Lines->Count > 0;
28:   EditSelectAll->Enabled = hasText;
29:   //
30:   // Код управления доступом к Save и Save As
31:   //
32:   bool modified = Memo->Modified;
33:   FileSave->Enabled = modified && hasText;
34:   FileSaveBtn->Enabled = modified && hasText;
35:   FileSaveAs->Enabled = modified && hasText;
36:   //
37:   // Код управления доступом к Undo
38:   //
39:   EditUndo->Enabled = modified;
40: }
```

Часть этого кода может выглядеть немного странной, если вы не сталкивались с этим прежде. Посмотрите, например, на строку 8. Здесь значение условного выражения Memo->SelLength > 0 присваивается переменной textSelected типа bool. Если значение свойства SelLength компонента memo больше 0, переменной textSelected присваивается значение true. Если значение свойства SelLength меньше или равно 0 (это означает, что текст не выделен), то переменной присваивается значение false. Это же значение затем присваивается свойствам Enabled различных пунктов меню и кнопок Cut и Copy. Остальной код функции работает таким же образом. Изучите листинг 13.2 и прочитайте комментарии, чтобы понять, как работает эта функция. Введите этот код в функцию OnIdle() программы ScratchPad и убедитесь, что все работает так, как ожидалось.



Отладка кода обработчика события OnIdle сложна. Проблема состоит в том, что любые контрольные точки в этом обработчике начнут немедленно срабатывать, как только вы запустите программу. Для отладки кода в обработчике события OnIdle вы будете вынуждены использовать вместо контрольных точек другие средства (например, макросы TRACE и WARN).



Обработчик события OnIdle может вызываться тысячи раз в секунду. По этой причине вы должны следить за тем, чтобы код функции был коротким.

В заключение...

Рассмотренное выше ни в коем случае не является единственным способом управления доступом к командам в приложении C++Builder. Тем не менее, я полагаю, что возникшие проблемы были решены довольно изящно. Каждая новая ситуации отличается от предыдущей, так что не бойтесь модифицировать это частное решение, если оно не вписывается в работу вашего приложения. В некоторых случаях вам может потребоваться прямое управление доступом к командам в ответ на некоторое действие пользователя. Независимо от способа реализации, управление доступом к командам следует использовать во всех подходящих ситуациях.

Печать в приложениях C++Builder

Печать — это ежедневная потребность для большинства пользователей Windows. Хотя и существует много программ, не предоставляющих такой возможности, большая часть приложений Windows поддерживает печать в том или ином виде. В этом разделе мы рассмотрим основы печати.

Обеспечение возможности печати в приложении DOS представляло собой тяжелую задачу. Программа DOS должна была содержать и устанавливать драйверы для каждого из поддерживаемых этой программой принтеров. Это создавало большие сложности для разработчиков программного обеспечения, особенно для небольших компаний и разработчиков условно-бесплатных программ. С появлением Windows все изменилось. Самое главное, что делает Windows — берет на себя бремя общения с различными принтерами, их

драйверами и т.д. Вам остается только направить вывод на принтер точно так же, как и в окно. Мы займемся этим несколько позже.

Печать в приложениях C++Builder имеет несколько разновидностей. Вы, возможно, будете рады узнать, что во многих случаях печать встроена в VCL и происходит почти автоматически. Тем не менее, иногда вам придется заниматься специализированной печатью. Именно здесь вам потребуется более детальное знание этого процесса. Но прежде, чем я начну рассказывать об этом, давайте кратко рассмотрим стандартные диалоговые окна, которые относятся к печати. После этого мы рассмотрим различные способы печати из приложений C++Builder.

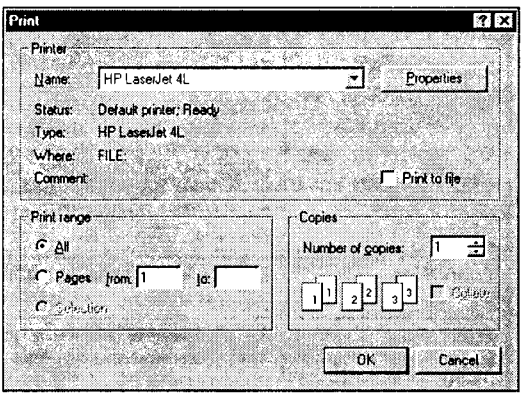
Стандартные диалоговые окна печати

Windows предоставляет в ваше распоряжение стандартные диалоговые окна Print и Print Setup, которые вы можете вызывать из своих приложений. Диалоговое окно Print используется непосредственно перед началом печати, а окно Print Setup служит для настройки принтера. Чтобы вызвать любое из этих диалоговых окон из своего приложения, вы должны сначала добавить в форму соответствующие компоненты.

Диалоговое окно Print

Как уже упоминалось, диалоговое окно Print отображается непосредственно перед началом печати, когда пользователь выбирает в главном меню пункт File | Print. Если пользователь нажимает кнопку OK, печать начинается; если пользователь нажмет кнопку Cancel, печать будет отменена. На рис. 13.6 показано диалоговое окно Print в наиболее общем виде.

Рис. 13.6
Диалоговое окно Print



У меня нет никаких сомнений в том, что вы не первый раз видите это диалоговое окно. Комбинированный список в верхней части окна позволяет выбрать для печати определенный принтер. Щелчок на кнопке Properties выведет диалоговое окно, соответствующее выбранному в данный момент принтеру. Диалоговое окно Properties позволяет установить ориентацию, разрешение и другие свойства, характерные для этого принтера. Раздел Print range (Диапазон печати) позволяет напечатать все страницы, страницы выбранного диапазона или любые объекты и текст, которые выбраны в данный момент в приложении. Раздел Copies (Число копий) позволяет определить число печатаемых копий, а также включить подбор копий по страницам.

Диалоговое окно Print инкапсулировано в компоненте VCL PrintDialog. Как и в случае других стандартных диалоговых окон, окно Print отображается при вызове его метода Execute(). Вероятно, вы не очень огорчитесь, узнав, что многие действия из диалогового окна Print на самом деле выполняются Windows. Выбор принтера, числа копий и включение режима подбора страниц — все это делает Windows, так что вам не нужно беспокоиться об этом. В зависимости от приложения, вы можете предоставить пользователю возможность печати определенного диапазона страниц или выбранного объекта. В этом случае перед началом печати вам нужно будет установить некоторые из свойств PrintDialog.

Компонент PrintDialog содержит только метод Execute() и не имеет событий. Все функциональные возможности компонента PrintDialog реализуются через его свойства (см. таблицу 13.3).

Таблица 13.3. Свойства PrintDialog

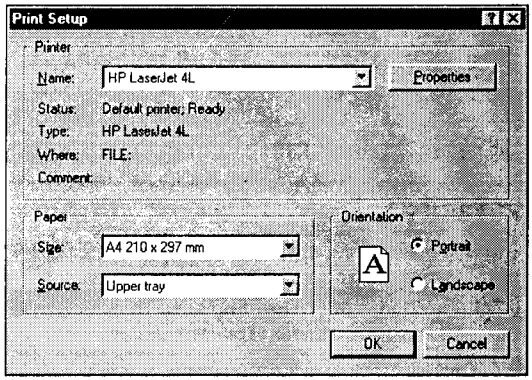
Свойство	Описание
Collate	Если это свойство установлено в true, Windows будет печатать копии с подбором страниц.
Copies	Определяет число печатаемых копий. Вы можете задать значение этого свойства перед вызовом диалогового окна Print, если число печатаемых копий устанавливается в опциях вашего приложения. Windows позаботится о печати правильного количества копий.
FromPage	Определяет начальную страницу, если включена печать диапазона. Приложения, поддерживающие печать диапазона страниц, должны считывать значение этого свойства, чтобы определить, какие страницы должны быть напечатаны.
MaxPage	Определяет максимальный номер страницы, который может быть задан в поле To при печати диапазона страниц. Диалоговое окно Print само заботится о проверке достоверности значений, введенных в поля From и To.
MinPage	Определяет минимальный номер страницы, который может быть задан в поле From при печати диапазона страниц.
Options	Содержит набор опций, определяющих, какие возможности диалогового окна Print доступны. Вы можете выбрать доступность кнопки Help, опции Print to File, опции печати диапазона страниц и печати выбранного объекта.
PrintRange	Определяет, какая из радиокнопок группы Print Range будет выбрана при первом выводе диалогового окна Print.
PrintToFile	Вы считываете значение этого свойства, чтобы определить, выбрал ли пользователь опцию Print to File. Запись в файл — это прерогатива приложения.
ToPage	Определяет номер последней страницы при печати диапазона страниц. Приложения, поддерживающие печать диапазона страниц, должны считывать значение этого свойства, чтобы определить, какие страницы должны быть напечатаны.

В ответ на закрытие диалогового окна Print приложению остается не так много работы, если только не были включены опции Print range и Print to File. Например, если ваше приложение позволяет печатать диапазон страниц, вы должны прочитать значения свойств FromPage и ToPage, чтобы определить, какие страницы печатать. Во всех остальных случаях вы просто начинаете печать, если пользователь выбрал OK.

Диалоговое окно Print Setup

Диалоговое окно Print Setup выводится в тех случаях, когда пользователь хочет сменить принтер, размеры страницы, источник бумаги или ориентацию печати (см. рис. 13.7).

Рис. 13.7
Диалоговое окно Print Setup



Диалоговое окно Print Setup не является необходимым в большинстве приложений, поскольку пользователь может всегда нажать кнопку Properties в диалоговом окне Print, чтобы изменить установки (вернитесь к рис. 13.6). С другой стороны, диалоговое окно Print Setup реализуется настолько просто, что вы также можете включать его в свои приложения. Компонент PrinterSetup не имеет никаких свойств, методов или событий, которые были бы характерны только для него. Как и для компонента PrintDialog, метод Execute() — это единственный метод, который может вас интересовать. Чтобы еще больше упростить дело, Windows управляет всем, что делает диалоговое окно Print Setup. Фактически, функция Execute() даже не возвращает никаких значений, поскольку Windows обрабатывает все за вас. Если пользователь щелкает на кнопке Cancel, Windows не делает ничего. Если пользователь нажимает OK, Windows производит соответствующие изменения для подготовки к печати. Все, что вам нужно — это вывести диалоговое окно Print Setup и забыть о нем. Типичный обработчик события для пункта меню File | Print Setup выглядит следующим образом:

```
void __fastcall TScratchPad::FilePrintSetupClick(TObject* Sender)
{
  PrinterSetupDialog->Execute();
}
```

Это все, что имеет отношение к данному окну. Как я и говорил, работа с диалоговым окном Print Setup настолько проста, что вы вполне можете добавить его в свое приложение.

Простой способ печати

Печать — это задача, зависящая от типа приложения. Эта мысль может показаться не очень глубокой, но это действительно так. В зависимости от типа приложения, которое вы разработали, печать может быть выполнена одной строкой или может повлечь за собой написание сотен строк кода. Позвольте мне сначала рассмотреть самые простые формы печати, и затем мы перейдем к более сложным операциям.

Метод Print() формы

В класс TForm входит метод Print(), который может использоваться для печати содержимого формы. Печатается только рабочая область формы; рамка и строка меню не выводятся. Хотя этот метод довольно хорошо работает при простой распечатке экрана, он имеет существенные ограничения. Вы можете выбирать между тремя опциями печати, которые управляются свойством PrintScaled. В таблице 13.4 приведен список возможностей масштабируемой печати и их описания.

Таблица 13.4. Опции свойства PrintScaled

Свойство	Описание
poNone	Масштабирование не применяется. Вид напечатанной формы будет зависеть от типа принтера.
poProportional	Печать формы с примерным сохранением размеров, которые она имеет на экране.
poPrintToFit	Изображение будет растянуто или сжато таким образом, чтобы точно уместиться в размеры, заданные текущими установками принтера.

Вы можете устанавливать свойство PrintScaled во время выполнения программы или на этапе разработки. Применение метода Print() ограничено простыми распечатками экрана и маловероятно, чтобы он был полезен для какой-либо серьезной печати.

Метод Print() компонента RichEdit

Компонент RichEdit предоставляет большие возможности. Прежде всего, это связано с тем количеством работы, которое выполняется элементом управления расширенным редактированием Windows, лежащим в основе этого компонента. Печать в компоненте RichEdit выполняется через обращение к методу Print(). Эта функция получает единственный аргумент, Caption, который используется менеджером печати, когда он отображает в своем окне выполняемые задания. Печать содержимого компонента RichEdit выполняется очень просто:

```
RichEdit->Print("MyApp.exe - readme.txt");
```

Печатая из компонента RichEdit, вам практически не нужно ни о чем заботиться. Перенос слов на новую строку и разбиение на страницы выполняются автоматически. Если вы используете многострочный элемент редактирования, для которого требуется печать, компонент RichEdit — это то, что вам нужно.



Для печати текстового файла вы можете использовать функцию API Windows ShellExecute(). ShellExecute() используется, кроме всего прочего, для запуска программы в зависимости от расширения имени файла. Например, Windows по умолчанию регистрирует расширение .txt как расширение, принадлежащее файлам программы Notepad Windows. Если вы дважды щелкнете на имени файла с расширением .txt в программе Explorer, Windows будет искать расширение .txt в системном реестре, определит, что для обработки таких файлов зарегистрирована программа Notepad.exe, после чего запустит Notepad. Файл, на котором вы дважды щелкнули, будет загружен автоматически.

Вы можете использовать это поведение в своих интересах. Возьмем, например, следующую строку кода:

```
ShellExecute(Handle, "print", "readme.txt", NULL, NULL,
SW_HIDE);
```

Ее выполнение приведет к загрузке Notepad, печати файла с именем Readme.txt и выходу из Notepad. Фактически, вы даже не увидите главное окно программы Notepad, поскольку в качестве параметра Show определен стиль SW_HIDE. Эта методика предполагает, что пользователь не изменил заданную по умолчанию регистрацию расширения .txt и не удалил Notepad из своей системы.

Печать через QuickReport

Программы, работающие с базами данных, могут использовать компонент QuickReport для печати отчетов. Я упоминаю здесь QuickReport лишь потому, что мы говорим о печати. Это один из компонентов для доступа к базам данных, и я оставлю рассмотрение фактической реализации этого типа печати до той главы, где QuickReport будет описан более подробно.

Сложный способ печати

Не позволяйте заголовку этого раздела ввести вас в заблуждение. Печать в действительности не является таким трудным делом, просто она требует некоторого количества времени и организации. Давайте сначала рассмотрим некоторые вещи, о которых вы должны знать, чтобы выполнять печать из своих приложений. После этого мы займемся собственно написанием кода.

Что такое контекст устройства?

Контекст устройства (DC — device context) — это что-то вроде грифельной доски, на которой могут рисовать программы Windows. Лучшим термином было бы слово *холст (canvas)*. На этом холсте можно нарисовать текст, линии, растровые изображения, прямоугольники, эллипсы и т.д. Тип линии, который используется для рисования в контексте устройства, зависит от текущего пера, выбранного в DC. Текущий цвет заливки и образец заливки определяются кистью, которая в настоящий момент выбрана в контексте устройства. Управление контекстами устройства должно выполняться достаточно аккуратно. Число DC, доступных Windows, ограничено, и вы должны быть внимательными и освобождать контекст устройства, как только закончили с ним работать. Кроме того, если вы правильно не удалите объекты, которые выбираете в контексте устройства, ваша программа вызовет утечку памяти и может даже оставить Windows в опасном состоянии. Как вы уже могли понять, работа с DC может быть довольно сложной.

Хорошая новость состоит в том, что VCL охраняет вас от необходимости знать все подробности контекстов устройств. VCL инкапсулирует DC Windows в классе TCanvas. Каждая форма в C++Builder имеет свой собственный холст, который представлен свойством Canvas. В любой момент, когда вам нужно что-либо нарисовать непосредственно в форме, вы делаете это, обращаясь к свойству Canvas. Например, следующий код проведет в форме диагональную линию из левого верхнего в правый нижний угол:

```
Canvas->MoveTo(0, 0);
Canvas->LineTo(ClientRect.Right, ClientRect.Bottom);
```

Тип рисуемой линии будет определяться текущим пером, которое задано свойством Pen холста.

Приятная особенность свойства Canvas — это то, что вы не должны беспокоиться обо всех мелких подробностях контекстов устройств Windows, которые могут сами начать управлять вами, сводя с ума. VCL позаботится о получении DC, выборе соответствующих объектов в DC и освобождении DC, когда он больше не нужен. Все, что от вас требуется — это нарисовать что-либо на холсте и предоставить VCL остальную работу.

Какое отношение все это имеет к печати? Самое прямое: Windows позволяет вам получить контекст устройства *принтера*, на котором вы можете рисовать текст, графику, линии и т.д. Другими словами, вы рисуете на DC принтера точно так же, как на DC экрана. Эта концепция резко отличается от того подхода к печати, который был реализован в старой доброй DOS. Теперь Windows приходит нам на помощь, позволяя использовать DC принтера. VCL помогает нам еще больше, инкапсулируя контексты устройств в свойстве Canvas. Таким образом, печать становится проще, чем когда-либо прежде.

Класс TPrinter и функция Printer()

VCL помогает выполнять операции печати, предоставляя в ваше распоряжение класс TPrinter. Этот класс инкапсулирует всю печать Windows. TPrinter имеет свойство Canvas, которое может использоваться для вывода линий, текста, графики и других рисуемых объектов на принтер. Я не хочу, чтобы это выглядело слишком просто, но все, что вы должны сделать для печати из своих программ C++Builder — включить заголовочный файл Printers.hpp, а затем написать код наподобие следующего:

```
Printer()->BeginDoc();
Printer()->Canvas->TextOut(20, 20, "Hello There!");
Printer()->EndDoc();
```

Функция Printer() обеспечивает доступ к объекту TPrinter, который полностью настроен и готов к использованию. Вам остается только задействовать его.

Давайте кратко рассмотрим свойства и методы TPrinter. В таблице 13.5 приведен список основных свойств TPrinter, а в таблице 13.6 — список основных методов TPrinter.

Таблица 13.5. Свойства TPrinter

Свойство	Описание
Aborted	Это свойство будет иметь значение true, если печать уже началась, а затем была прервана до нормального завершения.
Canvas	Механизм, посредством которого вы можете рисовать на принтере (контекст устройства принтера).
Fonts	Список шрифтов, поддерживаемых текущим принтером.
Handle	Дескриптор для контекста устройства принтера (HDC). Используйте это свойство, когда вам нужно вызвать функцию API Windows, для которой требуется дескриптор контекста устройства.

Свойство	Описание
Orientation	Ориентация печати (poPortrait или poLandsacpe). Это свойство будет установлено автоматически, когда пользователь выбирает принтер или изменяет его настройки, но вы можете установить его и вручную.
PageHeight	Высота страницы текущего принтера в пикселах. Значение этого свойства меняется от принтера к принтеру. Кроме того, это свойство будет содержать различные значения в зависимости от ориентации. Некоторые принтеры могут использовать несколько разрешений, что также влияет на значение свойства.
PageNumber	Номер страницы, которая печатается в настоящий момент. Значение этого свойства увеличивается на единицу каждый раз, когда вы вызываете NewPage() для печати новой страницы.
PageWidth	Ширина страницы в пикселах. Как и для свойства PageHeight, его значение может изменяться в зависимости от разрешения принтера, ориентации и размера бумаги.
PrinterIndex	Значение индекса принтера, выбранного в настоящий момент в списке доступных принтеров. Укажите -1, чтобы выбрать заданный по умолчанию принтер.
Printers	Список доступных системе принтеров.
Printing	Это свойство будет иметь значение true, если принтер в настоящий момент печатает.
Title	Текст, который идентифицирует выполняемое задание в окне менеджера печати.

Таблица 13.6. Методы TPrinter()

Метод	Описание
Abort	Прерывает печать до ее нормального завершения.
BeginDoc	Начинает процесс печати. Этот метод устанавливает принтер в Windows при подготовке к печати.
EndDoc	Заканчивает процесс печати. Посылает на печать текущую страницу и выполняет очистку печати в Windows.
GetPrinter	Определяет текущий принтер. Лучше использовать свойство Printers, а не данный метод. (Свойство Printers – привилегированный метод доступа к принтерам, поскольку вы можете использовать его как для определения текущего принтера, так и для установки текущего принтера.)
NewPage	Используется для вывода на печать текущей страницы и начала печати следующей. Увеличивает на единицу значение свойства PageNumber.
SetPrinter	Устанавливает текущий принтер. Вместо данного метода лучше использовать свойство Printers.

У класса TPrinter нет интерфейса на стадии разработки. Все процедуры работают только во время выполнения программы.

Практика

Настало время применить только что полученные знания на практике. Еще раз возьмем программу ScratchPad и продолжим приводить ее к достойному виду. В конце концов, что это за текстовый редактор, который не может печатать? В этом разделе мы добавим поддержку печати в старый

добрый ScratchPad. Сначала мы должны немного изменить главную форму. У нас уже есть пункты меню Print и Print Setup, но мы должны задействовать их и добавить к форме соответствующие диалоговые окна. Приступим к делу.

1. Дважды щелкните на компоненте MainMenu, чтобы вывести редактор меню.
2. Выберите пункт File | Print в меню ScratchPad в редакторе меню. Измените значение свойства Enabled на true.
3. Сделайте то же самое для пункта меню File | Print Setup. Закройте редактор меню.
4. Поместите в форму компонент PrintDialog. Измените значение его свойства Name на PrintDialog.
5. Поместите в форму компонент PrintSetupDialog и измените значение его свойства Name на PrintSetupDialog.

Теперь, когда мы завершили работу с формой, нужно изменить код. Для начала добавим пару строк в заголовочный файл SPMain.h. Сделайте следующее:

1. Откройте проект ScratchPad и перейдите в файл SPMain.h.
2. Добавьте после других #include, расположенных в начале файла, следующие:

```
#include <Windows.hpp>
#include <Printers.hpp>
```

3. Добавьте в раздел private объявления класса следующую строку:

```
void PrintFooter(Trect& rect, int lineHeight);
```

Эта строка объявляет функцию, которая будет печатать нижний колонтитул на каждой странице.

Это то, что касается заголовочного файла. Теперь мы должны добавить код в исходный модуль SPMain.cpp.

1. Вернитесь в редактор форм и выберите в главном меню формы пункт File | Print. На экране появится функция FilePrintClick(). Пока оставьте ее пустой.
2. Вернитесь в редактор форм и выберите в главном меню формы пункт File | Print Setup. Введите одну строку, начиная с позиции курсора, чтобы функция FilePrintSetupClick() выглядела следующим образом:

```
void __fastcall TScratchPad::FilePrintSetupClick(TObject* Sender)
{
  PrinterSetupDialog->Execute();
}
```

3. Находясь по-прежнему в редакторе кода, перейдите в конец файла и введите следующее:

```
void TScratchPad::PrintFooter(TRect& rect, int lineHeight)
{
}
```



Теперь мы готовы ввести код функций PrintClick() и PrintFooter(). В листинге 13.3 приведена целиком функция PrintClick(). Вы можете ввести ее вручную или загрузить вместе с программой ScratchPad с компакт-диска. В листинге 13.4 приведена функция PrintFooter(). Введите тело этой функции в свой файл SPMain.cpp.

Листинг 13.3. Функция FilePrintClick()

```
void __fastcall TScratchPad::FilePrintClick(TObject* Sender)
{
  //
  // Отображение диалогового окна Print
  //
  if (PrintDialog->Execute()) {
    //
    // Установка заголовка (title) для объекта принтера
    //
    Printer()->Title = "ScratchPad - " + OpenDialog->FileName;
    //
    // Установка для принтера такого же шрифта, что и в memo
    //
    Printer()->Canvas->Font = Memo->Font;
    //
    // Определение высоты строки. Берется размер (Size) шрифта
    // и с помощью функции MulDiv() вычисляется высота
    // строки с учетом текущего разрешения принтера.
    // Функция abs() используется для получения абсолютного
    // значения, поскольку результат может оказаться отрицательным
    // числом. После этого добавляется 40% на интерлиньяж
    //
    int lineHeight = abs(
      MulDiv(Printer()->Canvas->Font->Size,
      GetDeviceCaps(Printer()->Handle, LOGPIXELSY), 72));
    lineHeight *= 1.4;
    //
    // Определение количества строк на странице. Это значение
    // уменьшается на три строки, чтобы оставить место под нижнее поле
    //
    int linesPerPage = (Printer()->PageHeight/lineHeight) - 4;
    //
    // Начало печати со строки 4, а не строки 0, чтобы оставить
    // место под заголовок и верхнее поле
    //
    int lineCount = 4;
    //
    // Сообщение Windows о начале печати и печать заголовка
    //
    Printer()->BeginDoc();
    TRect rect;
    rect.Top = lineHeight;
    rect.Left = 20;
    rect.Right =  Printer()->PageWidth;
    rect.Bottom = lineHeight * 2;
    DrawText(Printer()->Handle,
      OpenDialog->FileName.c_str(), -1, (RECT*) &rect, DT_CENTER);
    //
    // Печать каждой строки в цикле
    //
    for (int i = 0; i < Memo->Lines->Count; i++) {
      //
      // После достижения конца страницы - сброс счетчика
      // строк, выдача страницы и начало новой
      //
```

см. продолжение

Листинг 13.3. (продолжение)

```
      if (lineCount++ == linesPerPage) {
        PrintFooter(rect, lineHeight);
        lineCount = 4;
        Printer()->NewPage();
      }
      //
      // Печать следующей строки с использованием TextOut()
      //
      String s = Memo->Lines->Strings[i];
      Printer()->Canvas->TextOut
        (0, lineCount * lineHeight, s);
    }
    //
    // Работа завершена
    //
    PrintFooter(rect, lineHeight);
    Printer()->EndDoc();
  }
}
```

Листинг 13.4. Функция PrintFooter()

```
void TScratchPad::PrintFooter(TRect& rect, int lineHeight)
{
  //
  // Компоновка строки для вывода номера страницы. Используется
  // массив char*, а не объект String, поскольку для
  // DrawText() в любом случае нужен char*
  //
  char buff[10];
  sprintf(buff, "Page %d", Printer()->PageNumber);
  //
  // Установка параметров прямоугольника, в котором будет выведен
  // нижний колонтитул. Отступ от низа страницы на две строки
  //
  rect.Top = Printer()->PageHeight - (lineHeight * 2);
  rect.Bottom = rect.Top + lineHeight;
  //
  // Вывод текста с помощью DrawText(), чтобы упростить
  // центрирование текста
  //
  DrawText(Printer()->Handle, buff, -1, (RECT*)&rect, DT_CENTER);
  //
  // Проведение линии через страницу немного выше текста "Page x"
  //
  Printer()->Canvas->MoveTo(0, rect.Top - 2);
  Printer()->Canvas->LineTo(rect.Right, rect.Top - 2);
}
```

Этот код является демонстрацией того, как можно печатать непосредственно через Windows, а не полагаться на встроенную печать VCL. Хотя я всегда выбираю простейший путь, чтобы сделать то или другое, существуют случаи, когда простой способ имеет недостаточную гибкость. Именно в такие моменты полученные знания помогут вам сделать работу без особых проблем.

Использование курсоров

Использование различных курсоров не представляет большой сложности, но я все же затрону эту тему, чтобы вы имели о ней общее представление. В этом разделе речь пойдет о смене курсоров во время выполнения программы. (Чтобы поменять курсор на этапе разработки, нужно просто установить новое значение для свойства Cursor компонента.) Сначала мы рассмотрим основы, а затем я расскажу о загрузке стандартных и специально разработанных курсоров.

Курсоры

Прежде всего, вы можете менять тип курсора как для конкретного компонента или формы, так и для всей рабочей области вашего приложения. В последнем случае вам нужно изменить значение свойства Cursor объекта Screen. Объект Screen соответствует экрану вашего приложения. Модифицируя свойство Cursor объекта Screen, вы обеспечиваете постоянный вид курсора независимо от того, в каком компоненте он находится. Предположим, например, что вы хотите заменить текущий курсор на курсор-песочные часы. Если вы модифицируете свойство Cursor только для формы, курсор примет вид песочных часов непосредственно над формой, но вернется к заданному по умолчанию виду, находясь над любыми другими компонентами формы. Изменив Cursor для объекта Screen, вы получите один и тот же курсор в любом месте.

Управление курсором — это прерогатива объекта Screen. Все доступные вам курсоры содержатся в свойстве Cursors объекта Screen. Обратите внимание, что имя этого свойства — Cursors, и это не то же самое, что свойство Cursor, рассмотренное в предыдущем абзаце. Свойство Cursors является массивом, который содержит список доступных приложению курсоров, а Cursor — это свойство, которое используется для отображения определенного курсора. Хотя поначалу эти имена можно путать, вы достаточно быстро с ними освоитесь. Все, что для этого нужно — небольшой опыт практической работы с курсорами.

Windows содержит несколько встроенных курсоров, которыми вы можете пользоваться в своих приложениях. К этим встроенным курсорам VCL добавляет несколько собственных типов курсоров. Все вместе эти курсоры называются *базовыми курсорами (stock cursors — курсоры, имеющиеся в наличии)*. С каждым из базовых курсоров связана именованная константа. Например, курсор в виде стрелки назван crArrow, курсор в виде песочных часов — crHourglass, а курсор перетаскивания — crDrag. Все эти курсоры хранятся в массиве Cursors и занимают в нем позиции от -17 до 0; курсор по умолчанию имеет в массиве индекс 0 (crDefault), отсутствию курсора соответствует индекс -1 (crNone), курсору стрелки — индекс -2 (crArrow) и т.д. Во встроенной справке по свойству Cursors перечислены все курсоры и имена соответствующих констант, так что обратитесь за полным списком доступных курсоров к справочной системе C++Builder.

Чтобы использовать один из курсоров массива Cursors, присвойте имя выбранного курсора свойству Cursor объекта Screen:

```
Screen->Cursor = TCursor(crHourGlass);
```

Здесь в дело вступает VCL, которая загружает и выводит нужный курсор. Свойство Cursors для вас прозрачно, поскольку вы не обращаетесь к нему непосредственно. Вместо этого вы просто выполняете присваивание значения свойству Cursor, а VCL выполняет поиск соответствующего курсора в массиве Cursors и его отображение. Свойство Cursor компонента и свойство Cursors объекта Screen работают совместно для отображения различных курсоров в вашем приложении.



Предыдущий фрагмент кода мог быть написан следующим образом:

```
Screen->Cursor = crHourGlass;
```

Код в такой записи по-прежнему работает, но компилятор будет генерировать предупреждение Assigning int to Controls::TCursor (Присваивание значения типа int объекту Controls::TCursor). Чтобы компилятор был счастлив, мы создаем временный объект TCursor.

Вам может потребоваться изменять курсоры в своих приложениях по многим причинам: для вывода курсора ожидания (песочные часы), специальных курсоров в своей графической программе или для реализации справочного курсора.

Теперь рассмотрим два типа курсоров, которые вы можете использовать в своих приложениях.

Загрузка и использование базовых курсоров

Windows предоставляет несколько встроенных курсоров, которые вы можете использовать в своем приложении. VCL добавляет к ним еще несколько курсоров. Вы можете использовать эти базовые курсоры в любой момент.

Одной из наиболее очевидных причин замены курсора является необходимость выполнения длительного процесса. Считается плохим стилем не информировать пользователя о том, что программа занята выполнением некоторой задачи. Именно для таких случаев Windows предоставляет вам курсор песочных часов (Windows называет его курсором ожидания). Предположим, например, что в вашем приложении есть цикл обработки, который мог бы занять длительное время. Тогда вы можете сделать что-нибудь вроде следующего:

```
//
// сохранение старого курсора
//
TCursor oldCursor = Cursor;
//
// замена курсора символом песочных часов
//
Screen->Cursor = TCursor(crHourGlass);
//
// начало процесса
//
DoSomeReallyLongProcess();
//
// восстановление исходного курсора
//
Screen->Cursor = oldCursor;
```

Поскольку вы не всегда знаете, какой курсор использует ваше приложение в настоящий момент, имеет смысл сохранять текущий курсор перед каждым изменением свойства Cursor. Закончив работу с данным курсором, вы всегда сможете восстановить старый курсор.



Иногда вы можете изменить свойство Cursor перед длительным циклом обработки, но в результате ничего не произойдет. Причина состоит в том, что Windows не успела изменить курсор перед тем, как ваша программа вошла в цикл. Войдя в цикл, приложение не сможет обрабатывать какие-либо сообщения, в том числе и сообщение о замене курсора. Чтобы предотвратить эту ситуацию, нужно «прокачать» цикл сообщений Windows в ожидании нужного сообщения, находясь при этом в своем цикле:

```
Application->ProcessMessages();
```

Теперь ваше приложение не блокирует поток сообщений, и курсор изменится в самом начале цикла.

Вы можете, конечно, заменить курсор отдельно взятого компонента. Например, программа рисования могла бы менять курсор рабочей области в соответствии с текущим инструментом рисования. В этом случае вам не нужно менять курсор для всего объекта Screen, поскольку курсор должен снова принимать вид стрелки при перемещении в оперативную панель, строку состояния или любой другой компонент, находящийся в форме. В этом случае вы должны установить курсор только для того компонента, который представляет рабочее окно вашего приложения:

```
PaintBox->Cursor = TCursor(crCross);
```

Теперь курсор меняется только для этого компонента, а все остальные компоненты используют свои собственные предопределенные курсоры.

Загрузка и использование курсоров пользователя

Загрузка специально созданных курсоров требует немного большей работы. Как уже говорилось ранее, свойство Cursors класса TScreen содержит список курсоров, доступных вашему приложению. Чтобы использовать курсор пользователя, требуется выполнить несколько шагов:

1. Создайте курсор в редакторе изображений C++Builder или с помощью другого редактора ресурсов.
2. Создайте .rc-файл, который ссылается на курсор, и включите его в свой проект. (Альтернативный способ — выполнить компиляцию ресурса и добавить .res-файл к проекту.)
3. Загрузите курсор в массив Cursors, вызвав функцию LoadCursor().
4. Задействуйте курсор, присвоив соответствующий индекс свойству Cursor формы, свойству Cursor объекта Screen или свойству Cursor любого другого компонента.

Первые два шага этого упражнения описывались в 12-й день, «Вспомогательные инструменты и опции C++Builder», когда я говорил о редакторе изображений, а также при рассмотрении проектов. Как только у вас есть курсор, скомпонованный с .exe, вы можете загрузить его с помощью функции LoadCursor(). Загрузка курсора в массив Cursors выполняется просто:

```
Screen->Cursors[1] = LoadCursor(HInstance, "MYCURSOR");
```

Здесь предполагается, что вы создали курсор с именем MYCURSOR и присваиваете ему позицию с номером 1 в списке курсоров (вспомните, что позиции от -17 до 0 заняты базовыми курсорами). Загрузка курсора должна выполняться только один раз, так что вы, возможно, сделаете это в обработчике события OnCreate вашей главной формы.



Все курсоры загружаются в свойство Cursors объекта Screen независимо от того, собираетесь ли вы использовать курсор с объектом Screen, формой или компонентом. Существует только один массив Cursors, и он принадлежит объекту Screen.

Чтобы использовать курсор, просто присвойте его свойству Cursor объекта Screen или любого компонента:

```
PaintBox->Cursor = TCursor(1);
```

Если у вас есть несколько курсоров, вы можете создать глобальные переменные или макроопределения для каждого курсора, чтобы можно было ссылаться на них, используя значимые имена, а не целые числа, в которых легче запутаться. Для такого метода предыдущий код выглядел бы следующим образом:

```
#define myCursor 1
Screen->Cursors[myCursor] = LoadCursor(HInstance, "MYCURSOR");
// ...
PaintBox->Cursor = TCursor(myCursor);
```

Как видите, загрузка и использование своих собственных курсоров не представляет трудностей. Достаточно один раз увидеть, как это делается.

Заключение

Сегодня вы узнали о некоторых элементах, которые являются неотъемлемой частью Windows-приложений высокого класса, и о том, как реализовывать их в своих программах. Я бы хотел предупредить вас, чтобы вы не переусердствовали с оперативными панелями, строками состояния и курсорами. Применяйте любое декоративное оформление, которое нужно для вашего приложения, но не перегружайте его этими элементами. Сегодня вы также познакомились с печатью из приложений C++Builder. В некоторых случаях поддержка печати встраивается в конкретный компонент, и печать становится невероятно простой. В других случаях вам придется засучить рукава и поработать с классом TPrinter. Но даже тогда печать не является тем, чего нужно бояться.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Могу ли я одновременно запретить доступ ко всем компонентам оперативной панели?

Да. Если вы запретите доступ к панели, которая содержит элементы управления, все они также будут заблокированы. Когда вы используете этот способ, элементы управления не выглядят недоступными (серыми) — они просто не работают.

Я хотел бы создавать стыкуемые оперативные панели, которые видел в других приложениях. Могу ли я сделать это в C++Builder?

Вы можете сделать в C++Builder все, что захотите, вопрос только в том, сколько работы это потребует. В C++Builder нет встроенной поддержки стыкуемых оперативных панелей, так что вам придется создать свой собственный вариант с самого начала. Вы можете поискать в различных сетевых источниках компоненты стыкуемых оперативных панелей, созданные независимыми производителями или распространяемые как условно-бесплатные.

Как мне поместить растровое изображение в строку состояния?

Создайте строку состояния с несколькими панелями. Измените стиль панели, которая должна содержать изображение, на psOwnerDraw. Затем в своем обработчике события OnDrawPanel вызовите метод Draw() класса TCanvas, чтобы вывести изображение в эту панель.

У меня была простая строка состояния, которую я затем преобразовал в составную. Теперь мои подсказки не больше не появляются. В чем ошибка?

Когда вы преобразуете простую строку состояния в составную, необходимо изменить код таким образом, чтобы выводился текст подсказок. Вместо того, чтобы использовать строку

```
StatusBar->SimpleText = Application->Hint;
```

вы должны написать следующее:

```
StatusBar->Panels->Items[0]->Text = Application->Hint
```

В данном случае текст будет выведен в первую панель строки состояния.

Зачем мне беспокоиться об управлении доступом к пунктам меню и кнопкам оперативной панели?

Потому что пользователи ожидают увидеть непротиворечивый интерфейс. Когда определенное действие невозможно (в меню или оперативной панели), соответствующие элементы должны выделяться серым цветом как недоступные. Это дает пользователю визуальную подсказку, что данная команда в принципе возможна, но в настоящий момент неприменима.

Я хочу организовать вывод из большого многострочного элемента редактирования в своем приложении. Какой способ является самым простым?

Простейший способ — это использование компонента RichEdit и его метода Print(), позволяющего напечатать содержимое компонента.

Я вижу, что наряду со свойством Handle объекта Printer существует также свойство Handle свойства Canvas объекта Printer. Чем они отличаются?

В данном случае нет никакого различия. Если вы вызываете функцию API Windows, которая требует дескриптор контекста устройства принтера, вы можете использовать как Printer()->Handle, так и Printer()->Canvas->Handle.

Когда я заменяю курсор своей главной формы, то над формой он имеет правильный вид, но над любой из кнопок снова принимает форму стрелки. Почему?

Вы должны заменить курсор объекта Screen, а не только формы. Изменение курсора объекта Screen гарантирует, что над любой частью вашего приложения будет использоваться новый курсор.

Вопросы

1. Как связать обработчик с событием OnClick кнопки оперативной панели?
2. Можете ли вы поместить в оперативную панель другие элементы управления, кроме кнопок?
3. Какой компонент вы можете использовать в своем приложении, чтобы реализовать множественную оперативную панель?
4. Для чего предназначено свойство SimplePanel компонента StatusBar?
5. Как изменить вручную текст строки состояния?
6. Как разрешить и запретить доступ к пунктам меню и кнопкам?
7. Как получить доступ к принтеру в приложении C++Builder?
8. Какой метод нужно вызвать, чтобы начать печать с классом TPrinter?
9. Какой метод TPrinter вы вызовете для печати новой страницы?
10. Как изменить курсор компонента во время выполнения программы?

Упражнения

1. Создайте приложение, которое имеет три оперативных панели, из которых в любой момент видна только одна. Поместите в форму три радиокнопки. Напишите код так, чтобы при нажатии радиокнопки отображалась соответствующая ей панель.
2. Создайте новую программу. Добавьте в нее оперативную панель с пятью кнопками и строку состояния. Включите режим вывода подсказок так, чтобы кнопки оперативной панели имели как короткие всплывающие подсказки, так и длинные указания в строке состояния.
3. Измените строку состояния программы ScratchPad и добавьте в нее вторую панель. Выведите в эту панель либо Saved, либо Modified, в зависимости от значения Memo->Modified.
4. Сделайте копию программы ScratchPad. Работая с копией, замените компонент Memo на компонент RichEdit и добавьте поддержку печати.

5. Измените окно About программы ScratchPad так, чтобы в него выводился текст Version 1.05. Измените также свойство Title диалогового окна Project Options и свойство Caption главной формы. В конце концов, мы добавили в программу новые возможности, и пользователи должны об этом знать!
6. Создайте свой курсор в редакторе изображений. Напишите программу, которая отображает этот курсор, когда нажата кнопка в главной форме.
7. **Дополнительное задание:** измените программу ScratchPad из упражнения 3 так, чтобы выводить различные растровые изображения в строку состояния в зависимости от того, сохранен ли текущий файл. (**Указание:** просмотрите BMP-файлы led1on.bmp и led1off.bmp в каталоге \Images\Buttons.)
8. **Дополнительное задание:** измените программу ScratchPad так, чтобы пользователь мог определить при печати верхнее, нижнее, правое и левое поля.

Углубленное программирование



Кент Рейсдорф

Сегодня вы узнаете о некоторых более глубоких аспектах программирования для Windows в C++Builder. В этой главе рассматриваются следующие темы:

- Реализация контекстно-зависимой справки
- Обработка исключений
- Использование реестра Windows
- Специальная обработка сообщений

У нас сегодня очень много работы, так что приступим.

Реализация контекстно-зависимой справки

Не так давно вы могли не включать поддержку контекстно-зависимой справки в свои приложения. Фактически, в небольших приложениях можно было вообще обойтись без справочного файла. Однако в условиях современного рынка я бы не рекомендовал это делать. Пользователи становятся все более и более требовательными. Справочный файл сегодня — не дополнение, а необходимый компонент полноценного приложения.

Контекстно-зависимая справка (context-sensitive help) означает, что при нажатии клавиши F1 на экран выводится, в зависимости от текущего контекста, определенная страница справочного файла приложения. Возьмем, например, IDE C++Builder. Предположим, что вы находитесь в диалоговом окне Project Options, где открыта вкладка Application. Когда вы нажимаете клавишу F1 (или кнопку Help в диалоговом окне), запускается программа WinHelp, которая выводит справочную статью для вкладки Application диалогового окна Project Options. Точно так же, если вы нажмете F1, когда на экране находится диалоговое окно Object Repository Options, вы получите справку именно для этого диалогового окна. Контекстная справка работает и для пунктов меню. Если вы выделите пункт меню и нажмете F1, то перейдете на страницу справочного файла, относящуюся к данному пункту.

Чтобы реализовать контекстную справку в своем приложении, вы должны выполнить следующее:

1. Создать справочный файл для своего приложения. Справочный файл должен содержать идентификаторы контекста.
2. Присвоить имя своего справочного файла свойству HelpFile объекта Application.
3. Установить свойство HelpContext всех форм или компонентов, для которых должна быть обеспечена контекстная справка. Значением этого свойства должен быть идентификатор контекста, соответствующий описанию данного компонента в справочном файле.
4. Создать пункты для меню Help, чтобы ваши пользователи могли вызывать справку из меню.

Сейчас мы последовательно рассмотрим каждый из этих шагов.

Создание справочного файла

Создание справочных файлов — это рутинная работа. Нельзя сказать, что я ненавижу ее, но и не жду с нетерпением. Вам повезло, если у вас есть свой отдел документации, который создает справочные файлы.



Даже у программистов с ограниченными финансовыми возможностями могут быть свои отделы документации, которые пишут за них справочные файлы. Часто бывает так, что программист называет автора документации «милая» или «дорогой». Например, программист могла бы сказать своему автору документации: «Милый, когда ты закончишь эту страницу справки, не мог бы ты уложить детей спать?»

Независимо от того, кто пишет справочный файл, должна существовать некоторая координация действий между автором справочного файла и программистом. Идентификаторы контекста в справочном файле должны соответствовать заданным непосредственно в программе.

Справочный файл Windows может конструироваться из нескольких отдельных файлов. Исходный текст для справочного файла Windows называется тематическим файлом (topic file). Тематический файл — это файл в формате .RTF (rich text format), содержащий большое количество специального кода, который понимает компилятор справки. Если ваш справочный файл включает графику, вы можете иметь один или более графических файлов. Графические файлы, использующиеся при создании справочного файла, включают растровые изображения (.BMP), метафайлы Windows (.WMF) и некоторые другие специальные графические файлы. Наконец, у вас должен быть *файл проекта* справки (.HPJ). Файл проекта содержит описание того, как компилятор справки должен объединить тематический файл с графическими файлами и любыми другими специальными файлами, которые требуются для создания результирующего справочного файла. Он также включает раздел [MAP] (карта), который сопоставляет номера идентификаторов контекста определенным темам справки. После того, как файл проекта создан, он обрабатывается компилятором справки, таким как Microsoft Help Workshop (вы можете найти этот компилятор в каталоге \CBuilder\Help\Tools). Help Workshop загружает файл проекта и компилирует его для создания результирующего справочного файла (.HLP).

Хотя вы и можете писать файлы справки в текстовом процессоре, который поддерживает формат RTF, я настоятельно рекомендую вам приобрести программу для автоматизированного создания справки. Это может существенно упростить написание справочных файлов.

Идентификаторы контекста и свойство HelpContext

Независимо от того, каким образом создается справочный файл, вы должны определить номера контекста, связанные с каждой основной темой справочного файла. Номер контекста используется справочной системой Windows WinHelp.exe для отображения определенной страницы справочного файла. Предположим, например, что вы открыли диалоговое окно Options в своем приложении. Когда пользователь нажимает F1 с открытым диалоговым окном Options, ваша программа передает WinHelp идентификатор контекста для этого диалогового окна. WinHelp запускается и выводит страницу справочного файла, которая объясняет порядок работы с диалоговым окном Options. Вам не требуется задавать идентификаторы контекста для каждой из страниц справочного файла, но это следует сделать для страниц с изложением главных тем, описанием диалоговых окон и других основных компонентов своего приложения.

Большинство компонентов (формы, пункты меню, элементы управления) имеют свойство HelpContext. Это свойство содержит идентификатор контекста, который будет передан WinHelp при нажатии клавиши F1, когда данный компонент находится в фокусе. По умолчанию значение свойства HelpContext установлено в 0. Если значение HelpContext компонента равно 0, то этот компонент наследует значение HelpContext родительского окна. Это позволяет устанавливать определенное значение HelpContext для формы в целом. При этом, независимо от того, в каком компоненте формы находится фокус ввода, при нажатии F1 будет использоваться контекст справки формы.



Вы, возможно, отметили, что у компонента SpeedButton нет свойства HelpContext. Это связано с тем, что SpeedButton не является компонентом оконного типа, и он не может получить фокус ввода. Следовательно, клавиша F1 не может использоваться с кнопками быстрого доступа.

Как минимум, вы должны обеспечить контекстную справку для каждой формы своего приложения (диалоговых или вторичных окон). В конечном счете, вам решать, для каких компонентов приложения предоставлять контекстную справку. Но если вы и ошибетесь, то это должна быть ошибка в сторону избытка справочной информации (если такое вообще возможно), а не наоборот.

Реализация контекстно-зависимой справки

Контекстно-зависимая справка относительно просто реализуется в приложениях C++Builder. Как я уже говорил, основная работа состоит в написании справочного файла. Все остальное по сравнению с этим — детские забавы.

Настройка справочного файла

Независимо от того, как вы реализуете контекстную справку, сначала вы должны сообщить Windows имя справочного файла приложения. Для этого нужно присвоить имя справочного файла свойству HelpFile класса Application. Вы можете сделать это двумя способами. Самый простой способ реализуется на этапе разработки через диалоговое окно Project Options. В 10-й день, «Подробнее о проектах», мы рассмотрели опции проекта. Я обратил ваше внимание на то, что вкладка Application диалогового окна Project Options имеет поле Help File, в котором задается имя справочного файла приложения. Просто введите в это поле имя вашего справочного файла, а VCL присвоит его свойству HelpFile. После этого ваше приложение будет использовать указанное имя при каждом запросе справки.

Вы можете также задать имя справочного файла во время выполнения программы. Это может потребоваться в том случае, если вы даете пользователям возможность поместить справочный файл в любой каталог по их выбору. Вы можете сохранить путь к справочному файлу в реестре Windows (реестр рассматривается позже, в разделе «Работа с реестром»), а затем присвоить его вместе с именем файла свойству HelpFile объекта Application. Например, часть вашего обработчика события OnCreate могла бы выглядеть следующим образом:

```
String filename = GetHelpFileName(); // функция, определяемая
                                     // пользователем
Application->HelpFile = filename;
```

При желании вы можете изменять свойство HelpFile во время выполнения программы, например, чтобы подключать различные справочные файлы (хотя это применяется не очень часто). Как только вы задали имя справочного файла, можно переходить к фактической реализации справочной системы.

Добавление поддержки клавиши F1

Чтобы добавить поддержку F1 для форм и компонентов, вам нужно только установить свойство HelpContext так, чтобы его значение совпадало с идентификатором контекста в справочном файле, а VCL возьмет его из этого свойства. Убедитесь в том, что вы присвоили имя справочного файла свойству HelpFile класса Application, а сам справочный файл содержит правильные идентификаторы контекста. После этого все будет работать без вашего участия.

Включение поддержки справочного файла в меню

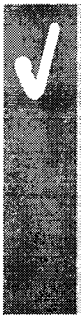
Дополнительно к поддержке клавиши F1, большинство приложений имеет один или два пункта в меню Help (действительно, где же им еще быть?), с помощью которых можно запустить WinHelp. Обычно здесь находится пункт меню Contents (Содержание). Выбор этого пункта выведет страницу содержания справочного файла. Страница содержания обычно служит главной страницей справочного файла. В дополнение к пункту Contents, некоторые приложения имеют в меню Help пункт Help Topics (Темы справки). Выбор этого пункта меню выводит индекс справочного файла (создание индекса является частью процесса разработки справочного файла).

Чтобы реализовать этот и другие пункты меню Help, нужно написать некоторый код. (В каждом случае речь идет только об одной строке.) VCL

содержит метод HelpCommand(), который может выводить WinHelp в одном из нескольких режимов. Если вам нужно реализовать пункт меню Help | Contents, то код будет выглядеть следующим образом:

```
void __fastcall TMainForm::HelpContentsClick(TObject* Sender)
{
  Application->HelpCommand(HELP_CONTENTS, 0);
}
```

Метод HelpCommand() вызывает WinHelp с заданной командой (см. справку API Windows по WinHelp(), где содержится полный список доступных команд). В данном случае WinHelp вызывается с командой HELP_CONTENTS. Эта команда содержит указание WinHelp вывести страницу, которая была обозначена как страница содержания. Последний аргумент метода HelpCommand() передает WinHelp дополнительные данные. Этот параметр не используется с командой HELP_CONTENTS, поэтому он установлен в 0.



Вообще говоря, вам не следует больше пользоваться командой HELP_CONTENTS для вывода страницы содержания. Вместо этого нужно применять команду HELP_FINDER совместно с ID страницы содержания. Если, например, страница содержания имеет идентификатор 1000, вы можете вызвать ее следующим образом:

```
Application->HelpCommand(HELP_FINDER, 1000);
```

Мне нравится команда HELP_CONTENTS, поскольку я могу использовать ее в любом приложении, не заботясь о том, чтобы помнить идентификатор страницы содержания для каждого из своих приложений.

Реализация пункта меню Help Topics практически не отличается от реализации пункта Contents, но содержит другую команду:

```
void __fastcall TMainForm::HelpTopicsClick(TObject* Sender)
{
  Application->HelpCommand(HELP_FINDER, 0);
}
```

Команда HELP_FINDER в сочетании с идентификатором контекста, имеющим значение 0, указывает WinHelp вывести индекс справочного файла. Как я уже сказал, индекс создается вместе со справочным файлом.



Хороший индекс имеет неоценимое значение для вашего справочного файла. Качество справочного файла (и его индекса) обратно пропорционально объему технической поддержки, которую вы должны будете предоставить. Не упустите из виду этот факт.

Контекстно-зависимая справка по запросу

В большинстве своих приложений вам будет достаточно тех двух способов реализации справки, о которых я только что говорил. Но иногда вы должны непосредственно вызывать WinHelp с конкретным идентификатором контекста. Для выполнения этой процедуры VCL предоставляет метод HelpContext(). Этот метод получает в качестве единственного аргумента идентификатор контекста той страницы, которую вы хотите увидеть после запуска WinHelp.

Предположим, например, что нужная вам страница имеет идентификатор 99. Чтобы запустить WinHelp и вывести данную страницу, вы должны сделать следующее:

```
Application->HelpContext(99);
```

Указывая конкретный идентификатор контекста, вы можете заставить WinHelp вывести любую страницу вашего справочного файла по запросу.

Использование заголовочных файлов справки

В большинстве случаев ваши действия сводятся к установке свойства HelpContext форм или компонентов, для которых вы хотите реализовать контекстную справку. Если вам нужно вызывать определенные страницы справки в своем приложении, вы можете использовать директивы #define, чтобы создать константы для идентификаторов справки. Именованную константу намного проще запомнить, чем целое значение определенного ID контекста справки.

В предыдущем разделе я говорил о применении метода HelpContext() для вызова WinHelp с конкретным идентификатором контекста. В качестве примера я привел значение идентификатора, равное 99. Вместо числового значения идентификатора контекста вы могли бы использовать константу:

```
Application->HelpContext(IDH_FILEOPEN);
```

Очевидно, что строку легче запомнить, чем ее числовой эквивалент. Все определения констант (#define) контекстной справки могут храниться в отдельном заголовочном файле, который включается в ваше приложение (с помощью директивы #include), когда это необходимо. В листинге 14.1 приведен текст типичного заголовочного файла, содержащего идентификаторы контекстной справки.

Листинг 14.1. HELP.H

```
#define IDH_FILENEW         1
#define IDH_FILEOPEN        2
#define IDH_FILESAVE        3
#define IDH_FILESAVEAS      4
#define IDH_FILEPRINT       5
#define IDH_FILEPRINTSETUP  6
#define IDH_FILEEXIT        7
#define IDH_EDITUNDO        8
#define IDH_EDITCOPY        9
#define IDH_EDITCUT         10
#define IDH_EDITPASTE       11
#define IDH_EDITSELECTALL   12
#define IDH_EDITWORDWRAP    13
#define IDH_HELPABOUT       14
```

В начале своего исходного файла вы добавляете строку, которая включает в него заголовочный файл справки:

```
#include "help.h"
```

Теперь при необходимости вы можете использовать имя идентификатора контекста, а не его целочисленное значение.

Способ создания заголовочного файла справки зависит от вспомогательных инструментов, с помощью которых вы пишете справочные файлы. Большинство программ автоматизированного создания справки имеет опцию для генерации заголовка некоторого описания, который содержит именованные константы. Конкретная реализация зависит от того, был ли написан справочный файл вами или вашим соавтором, и какой программой создания справки вы пользуетесь. Если вы не используете специализированную программу, то можете просто ввести все директивы #define вручную.

Практика

Настало время применить приобретенные знания на практике. В этом разделе мы добавим контекстную справку в программу ScratchPad. (Ведь вы догадывались, что мы вернемся к нашей старой программе ScratchPad, не так ли?)



Компакт-диск, прилагаемый к книге, содержит простой справочный файл для программы ScratchPad. Скопируйте файл Scratch.hlp с компакт-диска в свой рабочий каталог, чтобы C++Builder мог найти его при построении приложения ScratchPad.

Для реализации контекстной справки требуется не более 10 минут. Итак, мы начинаем.

1. Загрузите проект ScratchPad. Перейдите на вкладку Application диалогового окна Project Options и введите в поле Help File название файла Scratch.hlp. Нажмите OK, чтобы закрыть диалоговое окно. (Убедитесь, что вы переместили файл Scratch.hlp в каталог проекта.)
2. Загрузите главную форму ScratchPad в редактор форм. Дважды щелкните на значке MainMenu, чтобы вызвать редактор меню.
3. В редакторе меню выберите пункт меню File | New компонента MainMenu (не в главном меню C++Builder, конечно). Найдите свойство HelpContext в инспекторе объектов и установите для него значение 1.
4. Повторите то же самое для остальных пунктов меню. В качестве значений свойства HelpContext пунктов меню используйте значения из листинга 14.1.
5. Добавьте пункт Help Topics в меню Help. Измените его свойство Name на HelpTopics. Добавьте разделитель ниже пункта Help Topics, чтобы отделить пункты Help от пункта меню About. Закройте редактор меню.
6. В редакторе форм выберите Help | Contents в главном меню формы. Редактор кода выведет функцию HelpContentsClick(). Введите в нее следующую строку:

```
Application->HelpCommand(HELP_CONTENTS, 0);
```

7. Вернитесь в редактор форм. Выберите Help | Help Topics в главном меню. Когда появится редактор кода, введите следующую строку в позиции курсора:

```
Application->HelpCommand(HELP_FINDER, 0);
```

8. Измените значение свойства HelpContext главной формы на 1000 (идентификатор контекста страницы содержания).

Запустите программу и поэкспериментируйте с ней. Если вы нажмете F1, когда курсор находится в окне компонента memo, будет выведена стра-

ница Contents справочного файла. Если вы выделите какой-либо пункт меню и нажмете F1, будет выведена справка для этого пункта меню. Проверьте также, правильно ли работают пункты главного меню Contents и Help Topics.



Если вы откомпилируете и запустите программу ScratchPad с компакт-диска, то увидите, что она имеет возможности, которые здесь не рассматривались. В этой программе реализован режим справки (Help Mode), в котором можно получить справку по определенным кнопкам оперативной панели и пунктам меню. Чтобы войти в режим справки, щелкните на кнопке Help оперативной панели. Курсор примет вид курсора справки. Теперь выберите любой пункт меню или кнопку оперативной панели. Появится тематическая справка для выбранного пункта. Режим справки автоматически выключается после выбора определенного элемента. Просмотрите исходный код, чтобы понять, как реализован режим справки. Здесь используется специальная обработка сообщений, которую мы рассмотрим позже в этой главе.

Контекстно-зависимая справка больше не является роскошью. Относятся ли ваши пользователи к широкой публике или к вашим коллегам, они все — ваши пользователи. Они ожидают от программы некоторых возможностей, к которым относится и контекстная справка. Поскольку контекстная справка очень просто реализуется средствами C++Builder, нет никакой причины для ее отсутствия в ваших приложениях.

Обработка исключений

Даже в очень тщательно разработанной программе могут происходить такие вещи, которые не предусматривались ее алгоритмом. Пользователи делают ошибки. Например, пользователь может ввести недопустимое значение в поле данных или может открыть файл такого типа, который не обрабатывается вашим приложением. Независимо от сценария, вы всегда должны быть готовы к обработке подобных ошибок, где бы они ни происходили. Вы не можете предугадать каждое действие своих пользователей, но вы можете предупредить и элегантно обработать некоторые грубые ошибки общего вида.

Обработка исключений (исключительных ситуаций) является, по существу, усложненной формой проверки на наличие ошибок. Хотя любая программа может реализовать и использовать обработку исключений, для пользователей компонентов и других классов Си++ обработка исключений особенно важна. Например, если вы используете компонент и в нем происходит нечто из ряда вон выходящее, вы должны знать об этом. Хорошо написанный компонент сформирует исключение, когда что-то пойдет неправильно. Вы можете захватить это исключение и обработать его так, как вам нужно: либо завершить программу, либо дать пользователю возможность исправить ситуацию и повторить свои действия.



Существуют два типа обработки исключений. Прежде всего, это истинная обработка исключений Си++, которую мы здесь и обсуждаем. Кроме того, в 32-разрядной среде Windows реализованы так называемые *структурные исключения (structured exceptions)*. Обработка исключений этих двух типов — не совсем одно и то же. Я не буду вдаваться в подробности, а только хочу подчеркнуть, что вы можете столкнуться с двумя различными типами обработки исключений, когда доберетесь до еще более глубоких аспектов программирования в Windows. Маловероятно, что вы будете использовать структурные исключения, так как Си++ имеет собственный механизм обработки исключений.

Скорее всего, вам не придется очень часто заниматься обработкой исключений в своей повседневной работе. В основном вы будете обрабатывать исключения, формируемые VCL при обнаружении ошибок внутри компонента. Если вы сами создаете компоненты, то, возможно, вы будете чаще заниматься обработкой исключений.



Исключения в ваших программах должны быть зарезервированы для критических ошибок. Вы должны сформировать исключение, если условие ошибки достаточно серьезно, чтобы затруднить дальнейшую работу программы. Большинство ошибок во время выполнения программы может быть обработано путем проверки параметров, проверки правильности ввода данных пользователем и другими, более традиционными способами.

Рассказ об обработке исключений легко может занять всю главу, поэтому я ограничусь рассмотрением только тех исключений, которые формируются компонентами VCL.

Ключевые слова обработки исключений: try, catch и throw

Синтаксис обработки исключений не особенно сложен. Существуют три ключевых слова — try (контролировать), catch (ловить, захватывать) и throw (выбрасывать, формировать, генерировать, посылать). Ключевые слова try и catch используется при обработке исключений, а ключевое слово throw — для формирования исключения. Рассмотрим их более подробно.

Синтаксис

Ключевые слова try и catch:

```
try {
  TryStatements
}
catch (TypeToCatch)
{
  CatchStatements
}
```

Ключевое слово try отмечает начало контролируемого блока. *TryStatements* — это исполняемые операторы. Если какие-либо исключения типа *TypeToCatch* сформированы во время выполнения *TryStatements*, то выполняются операторы *CatchStatements*. Если ни одно исключение не сформировано, операторы *CatchStatements* игнорируются и выполнение программы продолжается с оператора, следующего за блоком catch. *TypeToCatch* может быть одним из встроенных типов данных (int, char, long и т.д.), экземпляром класса или многоточием (...). Если *TypeToCatch* — многоточие, то захватываются все исключения, независимо от их типа.

Захват исключений с try и catch

Прежде, чем я попытаюсь объяснить ключевые слова try и catch, рассмотрим простой пример. Листинг 14.2 содержит короткий фрагмент кода, который иллюстрирует обработку исключений. Этот код пытается загрузить

графический файл (.bmp, .wmf или .ico). Если выбранный пользователем файл не относится к этим форматам, VCL формирует исключение. Вы захватываете исключение и выводите окно сообщения, говорящее пользователю, что выбранный файл не является графическим файлом Windows.

Листинг 14.2. Пример обработки исключений

```
 1: try {
 2:   TImage* image = new TImage(0);
 3:   image->Picture->LoadFromFile(OpenDialog->FileName);
 4:   TChild* child = new TChild(this);
 5:   child->Parent = this;
 6:   child->Image->Picture = image->Picture;
 7:   child->ClientWidth = child->Image->Picture->Width;
 8:   child->ClientHeight = child->Image->Picture->Height;
 9: }
10: catch (...)
11: {
12:   Application->MessageBox(
13:    "This file is not a Windows image file.",
14:    "Picture Viewer Error", MB_ICONHAND | MB_OK);
15: }
```

В этом коде вы видите контролируемый блок и блок захвата. Блок try содержит код, для которого может быть сформировано исключение. Оператор try говорит компилятору: «Проверь работу этих операторов». Если код работает нормально, блок catch игнорируется и выполнение программы продолжается. Если какой-либо из операторов внутри контролируемого блока формирует исключение, код блока catch будет выполнен. Блок catch должен следовать непосредственно после контролируемого блока (между ними не должно быть никакого другого кода).

Важно понимать, что как только исключение сформировано, управление немедленно передается блоку catch. В этом примере формирование исключения наиболее вероятно для кода в строке 3. Если исключение сформировано в строке 3, программа перейдет к строке 12. В этом случае строки 4–8 никогда не будут выполнены.

Формирование исключений

Итак, оператор catch захватывает исключение, которое было сформировано в каком-либо месте программы. В большинстве случаев это будет исключение, посланное VCL.

Исключение формируется с помощью ключевого слова throw и имеет определенный тип. Например, типичный оператор throw может выглядеть следующим образом:

```
if (badParameter) throw("A bad parameter was passed");
```

Оператор throw формирует объект того же типа, который передан ему как параметр. В данном случае посылается char*, но может быть сформирован объект любого другого типа. Предположим, что для конкретной ошибки у вас есть код 111. В этом случае вы можете написать следующий оператор throw:

```
if (badParameter) throw(111);
```

Данный оператор формирует целое число. Компилятор делает копию сформированного объекта и передает ее оператору catch, следующему сразу после контролируемого блока (мы вернемся к этому чуть позже).

Большинство библиотек классов формирует экземпляр класса обработки исключений. В VCL определено несколько таких классов. Существует несколько причин, по которым формируется экземпляр класса. Прежде всего, класс может содержать члены-данные и/или функции, которые описывают все, что вы должны знать для обработки ошибки. Например, базовый класс обработки исключений может содержать члены-данные с именами Error и Msg, которые содержат номер ошибки и текстовую строку, описывающую ошибку. В этом случае вы проверяете номер ошибки, чтобы правильно ее обработать, или выводите строку сообщения (или делаете все вместе).

Захват

Как я уже говорил, вероятнее всего вам придется заниматься захватом исключений VCL. Если в компоненте VCL что-то идет не так, VCL сформирует исключение. Если вы не обработаете это исключение, VCL обработает его заданным по умолчанию способом. Обычно это означает, что будет выведено окно сообщения с описанием возникшей ошибки. Захватывая эти исключения в своем коде, вы можете сами решить, как они должны быть обработаны, а не принимать заданное по умолчанию поведение.

Взгляните на листинг 14.2 еще раз. В строке 10 вы видите следующее:

```
catch (...)
```

Эта запись сообщает компилятору, что вы хотите захватывать все исключения, независимо от их типа. Если в блоке try сформирован какой-либо тип исключения, будет выполнен блок catch. Это удобно, если вы не знаете, какое исключение сформирует определенный фрагмент кода, или вы хотите обрабатывать все исключения. Скорее всего, в реальности вы будете сужать диапазон захватываемых исключений.

В качестве примера снова вернемся к листингу 14.2. Код в строке 3 может сформировать исключение, если файл, который пользователь пробует открыть, не является графическим. В этом случае VCL формирует исключение EInvalidGraphic. Если говорить точнее, то VCL сгенерирует исключение и сформирует экземпляр класса EInvalidGraphic. Вы можете захватывать только этот тип исключений, а остальные исключения будут обработаны заданным по умолчанию способом:

```
catch (EInvalidGraphic&)
{
  Application->MessageBox(
    "This file is not a Windows image file.",
    "Picture Viewer Error", MB_ICONHAND | MB_OK);
}
```

Обратите внимание, что здесь захватывается ссылка на класс EInvalidGraphic. Это более предпочтительный метод захвата исключений. Захват класса, переданного по ссылке, эквивалентен передаче класса по ссылке (см. День 4, «Классы и объектно-ориентированное программирование».) Любые исключения данного типа будут теперь захватываться блоком

catch, а все другие типы исключений будут обработаны заданным по умолчанию способом.



VCL автоматически обрабатывает многие типы исключений, но не может предусмотреть все случаи. Исключение, которое не может быть обработано, называется *необрабатываемым исключением (unhandled exception)*. Если возникает необрабатываемое исключение, Windows генерирует сообщение об ошибке, и работа вашего приложения завершается. Попытайтесь предусмотреть исключения, которые могли бы произойти в вашем приложении, и примите меры для их обработки.

Вы можете включить в программу несколько блоков catch, следующих после оператора try. Предположим, например, что вы написали код, который вызывает некоторые методы VCL и некоторые функции вашей программы, которые могут сформировать исключение. Если возникает VCL-исключение EInvalidGraphic, вы намерены его обработать. Кроме этого, ваш собственный код может сформировать исключение типа MyOwnException (класс, который вы написали для обработки исключений). В таком случае вы можете написать следующий код:

```
try {
  OneOfMyFunctions();
  Form->Image->Picture->LoadFromFile(filename);
  AnotherOfMyFunctions();
}
  catch (EInvalidGraphic&) {
  // какие-то действия
}
catch (MyOwnException&) {
  // какие-то действия
}
```

Здесь вы подготовились к захвату либо исключения VCL, либо исключения собственного типа. Каждый тип исключения нужно обрабатывать раздельно, чтобы вы могли захватывать эти исключения независимо.

Непосредственно после оператора try вы можете поставить столько операторов catch, сколько вам нужно. Однако у вас должен быть как минимум один оператор catch, следующий сразу за контролируемым блоком. При наличии нескольких операторов catch, они должны следовать непосредственно друг за другом, без какого-либо постороннего кода между соседними catch. Компилятор выдаст сообщение об ошибке, если вы нарушите синтаксис try/catch.

Обратите внимание, что в предшествующих примерах мы захватываем ссылку на класс исключения VCL, но ничего не делаем с экземпляром класса, переданным блоку catch. Здесь мы действительно не заботимся о том, какую информацию содержит класс EInvalidGraphic, поскольку нам просто нужно знать, что исключение произошло. Вам следует просмотреть интерактивную справку по интересующему вас конкретному классу VCL, обрабатывающему исключения, для получения более детальной информации о свойствах и методах, доступных в этом классе.



Все классы исключений VCL имеют свойство Message, содержащее текстовое описание возникшего исключения. Вы можете использовать это свойство, чтобы вывести соответствующее сообщение пользователю:

```
catch (EInvalidGraphic& info)
{
  // необходимые действия после возникновения
  // ошибки, затем...
  Application->MessageBox(
   info.Message.c_str(),
   "My Program - Error", MB_ICONHAND | MB_OK);
}
```

Иногда сообщения VCL могут показаться недостаточно информативными. В таких случаях вы можете написать свои собственные сообщения об ошибках, но, как правило, свойства Message бывает достаточно.

Захват необрабатываемых исключений на уровне приложения

Вы можете также обрабатывать исключения на уровне приложения. TApplication имеет событие с именем OnException, которое будет происходить при формировании каждого необрабатываемого исключения. Реагируя на это событие, вы можете захватить любое исключение, сформированное вашим приложением.



Это событие инициируется при формировании необрабатываемого исключения. Любые исключение, которые вы захватываете с помощью try и catch, обрабатываются; следовательно, для этих исключений событие OnException происходить не будет.

Вы перехватываете событие OnException точно так же, как мы это делали ранее, перехватывая события OnHint или OnIdle:

1. Включите объявление функции в заголовок главной формы:

```
void __fastcall OnException(TObject* Sender, Exception* E);
```

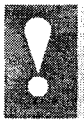
2. Добавьте в исходный модуль главной формы определение функции:

```
void __fastcall TMainForm::OnException(TObject* Sender, Exception*
E)
{
  // сделайте здесь все, что нужно для обработки исключения
}
```

3. Присвойте адрес своей функции OnException() событию OnException объекта Application:

```
Application->OnException = &OnException;
```

Теперь функция OnException будет вызываться при возникновении необрабатываемых исключений.



Если вы неправильно обработаете исключение, то можете блокировать свою программу или даже вызвать крах Windows. Обычно лучше всего предоставить обработку исключений VCL и Windows, если вы не знаете точно, как это делается.

Функция ShowException() может использоваться для отображения окна сообщения с описанием возникшей ошибки. Обычно эта функция вызывается обработчиком по умолчанию для OnException, но вы можете с равным успехом использовать ее в своих приложениях. Один из параметров обработчика события OnException — это указатель на объект Exception (Exception — это базовый класс VCL для обработки исключений). Чтобы вывести окно сообщения об ошибке, передайте объект Exception функции ShowException():

```
void __fastcall TMainForm::OnException(TObject* Sender, Exception* E)
{
  // сделайте здесь то, что нужно для обработки исключения,
  // а затем выведите сообщение об ошибке
  Application->ShowException(E);
}
```

Теперь окно с сообщением об ошибке будет отображаться так же, как и при реакции VCL на необрабатываемое исключение.

Как видите, обработка исключений на уровне приложения достаточно сложна, поэтому лучше не заниматься этим до тех пор, пока вы не будете абсолютно уверены в своих действиях.

Отладка с обработкой исключений

Если в программе применяется обработка исключений, то ее отладка может несколько усложниться. Каждый раз, когда формируется исключение, отладчик приостанавливает выполнение программы на блоке catch, как если бы в этой строке была установлена контрольная точка. Если блок catch находится в вашем коде, будет выведена точка выполнения, как и при обычной остановке в контрольной точке. Вы можете снова запустить программу, щелкнув на кнопке Run, или перейти к пошаговому выполнению кода.



Вы можете столкнуться с тем, что после формирования исключения будет невозможно продолжить выполнение программы. Возможность продолжения отладки зависит от того, какая ошибка возникла в вашей программе.

Иногда блок catch находится в коде VCL. Это будет иметь место для тех исключений VCL, которые вы не обрабатываете в своем коде. В таком случае точку выполнения можно увидеть с помощью CPU View.

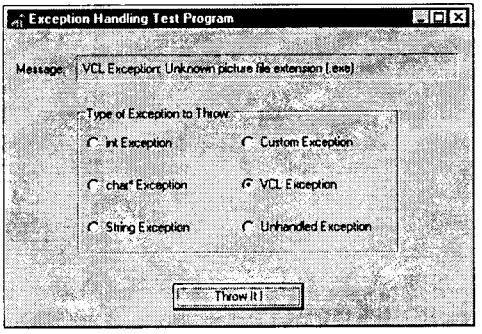
Другой аспект отладки с исключениями состоит в том, что окно сообщения для исключения VCL будет выведено даже в том случае, если вы самостоятельно обрабатываете исключение. Это обстоятельство может путать и несколько раздражать, если вы не сталкивались с этим прежде. Чтобы предотвратить вывод окна сообщения VCL, а также остановку программы при формировании исключения, откройте вкладку Preferences диалогового окна Environment Options и сбросьте флажок Break on Exceptions (Прерывание при исключении). В этом случае отладчик не будет приостанавливать выполнение программы при формировании исключения.

Как и со многими другими аспектами C++Builder и VCL, изучение обработки исключений требует некоторого времени. Не забывайте про обработку исключений, когда это действительно может быть необходимо, но для менее существенных ошибок используйте традиционные методы обработки.



Компакт-диск, прилагаемый к книге, содержит программу EHTest. Эта программа демонстрирует формирование и обработку нескольких видов исключений. Обратите внимание, что в этом проекте флажок Break on Exceptions сброшен. На рис. 14.1 показана работающая программа EHTest.

Рис. 14.1
Работающая программа EHTest



Работа с реестром

С самого начала программы Windows использовали файлы конфигурации (файлы с расширением .INI) для сохранения информации, относящейся к приложению. Главный конфигурационный файл, как вы, возможно, знаете, называется WIN.INI. Приложения могли сохранять общесистемную информацию в WIN.INI, а относящиеся к приложению данные в собственном .INI -файле. Этот подход имеет свои достоинства, но вместе с тем и некоторые недостатки.

По ходу дела кто-то более сообразительный (возможно), чем я, нашел новый способ хранения информации, относящейся к приложению — реестр (Registry). Во всем королевстве слуги поклонились королю и сказали: «Реестр — это хорошо». Плох он или хорош, но руководствуясь здравым смыслом, нужно прекращать использование .INI-файлов и переходить к работе с реестром.

Термин *Registry* (*реестр*) является сокращенным названием для Windows Registration Database (регистрационная база данных Windows). Реестр содержит разнообразную информацию о конфигурации вашей системы. В реестре сохраняется почти каждая опция и установка Windows. В дополнение к системной информации вы найдете здесь данные, относящиеся к установленным приложениям. Тип информации, хранимой для каждого приложения, полностью определяется этим приложением, но сюда могут включаться такие вещи, как последний размер и положение окна, список последних открытых файлов, последний каталог, использованный при открытии файла и т.д. Возможностей бесконечно много.

Вместе с Windows 95 и NT поставляется программа REGEDIT.EXE, предназначенная для просмотра и изменения данных реестра. Эта программа

называется Registry Editor (редактор реестра). На рис. 14.2 показан редактор реестра с выведенными опциями редактора форм C++Builder.

Как видите, реестр имеет иерархическую структуру. Вы можете использовать для себя аналогию с каталогами и файлами жесткого диска.

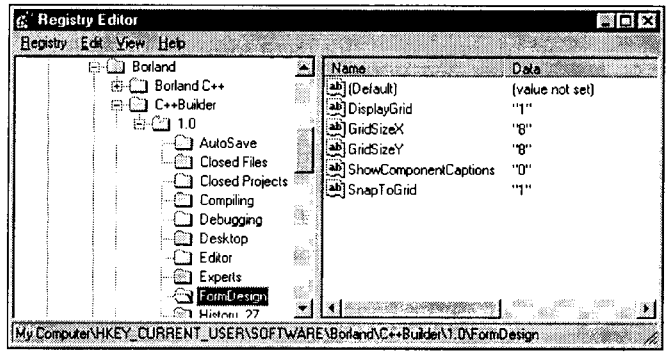


Рис. 14.2
Окно редактора реестра

Ключи реестра

Каждый пункт реестра называется *ключом*. Аналогом ключа может служить каталог на вашем жестком диске. Чтобы обратиться к определенному ключу, сначала его нужно открыть. После того, как ключ открыт, вы можете читать или записывать в него данные. Посмотрите на рис. 14.2. В данный момент в окне отображается ключ

```
\HKEY_CURRENT_USER\Software\Borland\C++Builder\1.0\FormDesign
```

Вы не можете видеть каждую ветвь дерева реестра, но если вы посмотрите на строку состояния редактора реестра, то увидите ключ, который отображается в данный момент. Также обратите внимание на то, что ключ C++Builder\1.0 имеет несколько *подключей (subkeys)*. Вы можете создать столько ключей и подключей для своего приложения, сколько вам требуется.

Данные отдельных ключей хранятся в *элементах данных (data items)*. Каждый ключ имеет элемент данных с именем (Default). Значение по умолчанию обычно не используется, поскольку вы почти всегда будете создавать для ключа свои собственные элементы данных. Рассматривая рис. 14.2, вы можете заметить, что ключ FormDesign содержит следующие элементы:

```
DisplayGrid
GridSizeX
GridSizeY
ShowComponentCaptions
SnapToGrid
```

Если вы были внимательны, то сможете вспомнить, что эти элементы данных соответствуют опциям редактора форм во вкладке Preferences диалогового окна Environment Options. Каждый элемент данных имеет значение, связанное со значением соответствующей опции. Вы можете изменить значение элемента данных в реестре или просто прочитать это значение.

Типы данных реестра

В реестре можно хранить данные нескольких типов. Для вас наибольший интерес представляют двоичные данные, строки и данные целого типа. *Двоичный тип данных (binary data type)* может использоваться для хранения любых двоичных данных. Например, в двоичном элементе данных вы можете сохранить массив целых чисел. Возможно, вы не очень часто будете использовать двоичный тип непосредственно.

Скорее всего, в большинстве случаев вам будет достаточно считывать и записывать строки или целые значения. Как видно из рис. 14.2, даже числовые данные могут быть сохранены в виде строк. Вы сами должны решать, в каком виде сохранять данные в реестре.

До сих пор мы обсуждали то, что вы можете делать с реестром, но не касались самой процедуры работы с реестром. Давайте рассмотрим это следующим шагом.

Класс TRegistry

API Windows предоставляет несколько функций для манипуляций с реестром. К ним относятся функции RegCreateKey(), RegOpenKey(), RegQueryValue(), RegSetValue(), RegDeleteKey() и множество других. Работа с реестром на уровне API немного утомительна. Я благодарен (и вы, конечно, тоже) людям из Borland, создавшим класс VCL с названием TRegistry, который инкапсулирует операции с реестром. Этот класс (наряду с производным классом TRegIniFile) предоставляет вам все, что нужно для чтения и записи в реестр. Прежде, чем мы рассмотрим работу с классом TRegistry, давайте пройдемся по его свойствам и методам.

Свойства TRegistry

TRegistry имеет четыре свойства. Свойство CurrentKey содержит значение текущего ключа. Это целое число, которое идентифицирует ключ. Когда вы вызываете метод класса TRegistry, этот метод работает с текущим ключом. Установка свойства CurrentKey происходит автоматически, когда вы открываете ключ. Вы можете прочитать значение ключа, но от этого не так много пользы.

Свойства RootKey и CurrentPath работают вместе, формируя текстовую строку для текущего ключа. Свойство CurrentPath содержит текстовое описание пути к текущему ключу, исключая значение RootKey. Например, текущий путь ключа, открытого на рис. 14.2 — Software\Borland\C++Builder\1.0\FormDesign. Свойство RootKey в этом примере содержало бы значение \HKEY_CURRENT_USER.



По умолчанию RootKey имеет значение \HKEY_CURRENT_USER. Именно здесь вы должны сохранять данные, относящиеся к приложению, поэтому, как правило, нет необходимости изменять корневой ключ. Если вам понадобилось изменить корневой ключ, просто установите для свойства RootKey новое значение.

Свойство LazyWrite определяет, как приложение записывает данные в заданный ключ. Если LazyWrite имеет значение true, то управление немедленно возвращается приложению, когда вы закрываете ключ. Иными словами,

после начала записи ключа ваше приложение возвращается к текущей работе. Если LazyWrite имеете значение false, управление не будет возвращено приложению, пока запись ключа не завершится. По умолчанию LazyWrite имеет значение true, и вы не должны менять его, если только вам не требуется сохранить некоторые важные данные, которые должны быть записаны прежде, чем ваше приложение продолжит работу.

Методы TRegistry

Класс TRegistry содержит несколько методов, которые применяются для чтения и записи в реестр. В таблице 14.1 приведен список основных методов и их описания.

Таблица 14.1. Основные методы TRegistry

Метод	Описание
CloseKey	Закрывает ключ и записывает в него данные. Ключ следует закрывать сразу после окончания работы с ним, но вы не должны специально вызывать метод CloseKey(), поскольку деструктор TRegistry закроет ключ самостоятельно.
CreateKey	Создает ключ, но не открывает его для выполнения операций. Вместо CreateKey()лучше использовать метод OpenKey().
DeleteKey	Удаляет указанный ключ. Вы можете указать здесь любой ключ. Чтобы удалить текущий ключ, передайте DeleteKey() пустую строку.
GetKeyNames	Возвращает объекту TStrings все подключи текущего ключа. Вы можете использовать этот метод, если нужно выполнить циклическую операцию для всех подключей данного ключа.
GetValueNames	Возвращает имена всех элементов данных текущего ключа. Используйте этот метод, если вам нужно выполнить циклическую операцию для всех элементов данного ключа.
KeyExists	Возвращает true, если указанный ключ существует, и false в противном случае. Вы можете вызвать этот метод для проверки существования ключа перед попыткой его чтения.
LoadKey	Загружает ключ, предварительно сохраненный на диске. За подробностями обращайтесь к справочной системе C++Builder.
OpenKey	Открывает указанный ключ. Если ключ не существует, значение параметра CanCreate определяет, будет ли данный ключ создан автоматически. Используйте этот метод вместо CreateKey(), если вы собираетесь создать ключ и начать записывать в него данные, поскольку OpenKey() создает ключ, а затем его открывает.
ReadBinaryData	Читает двоичные данные из указанного элемента данных.
ReadBool	Читает булево значение из указанного элемента данных.
ReadDateTime	Читает значение даты и времени из определенного элемента данных. Возвращаемое значение является экземпляром класса TDateTime. Чтобы получить только значение даты, используйте ReadDate(); чтобы получить только значение времени, используйте ReadTime().
ReadFloat	Читает значение с плавающей точкой из указанного элемента данных.
ReadInteger	Читает целое значение из указанного элемента данных.
ReadString	Читает строковое значение из указанного элемента данных.
SaveKey	Сохраняет ключ на диске для последующей загрузки методом LoadKey(). Подробное описание содержится в справочной системе C++Builder.
ValueExists	Возвращает true, если указанный элемент данных существует.

Метод	Описание
WriteBinaryData	Записывает двоичный элемент данных в указанный ключ. Используйте этот метод, чтобы сохранять массивы или двоичные данные других типов.
WriteBool	Записывает булево значение в указанный элемент данных. Значение преобразуется в целое число, а затем сохраняется в элементе данных.
WriteDateTime	Записывает объект TDateTime в определенный элемент данных. Чтобы сохранить только дату, используйте WriteDate(). Чтобы сохранить только время, используйте WriteTime(). Непосредственно перед сохранением объект TDateTime преобразуется в двоичный тип данных.
WriteFloat	Записывает число с плавающей точкой в указанный элемент данных, предварительно преобразовав его в двоичные данные.
WriteInteger	Записывает целое число в указанный элемент данных.
WriteString	Записывает строку в указанный элемент данных.

Хотя здесь перечислено большое количество методов, многие из их выполняют одинаковые операции, но над различными типами данных. Узнав, как применять один из этих методов, вы тем самым узнаете, как использовать все остальные. Обратите внимание, что некоторые методы преобразуют переданное значение в двоичные данные, а затем сохраняют их в реестре.

Использование TRegistry

Работа с TRegistry очень проста. В большинстве случаев ваше взаимодействие с реестром сводится к выполнению следующих четырех шагов:

1. Создайте экземпляр класса TRegistry.
2. Откройте ключ, вызвав метод OpenKey().
3. Прочитайте или запишите данные, вызвав необходимое количество раз функцию Read() или Write().
4. Закройте ключ, вызвав функцию CloseKey().



Перед использованием TRegistry необходимо включить в ваш исходный файл заголовок для классов реестра. Добавьте следующую строку в модуль главной формы под другими перечисленными там заголовками:

```
#include <Registry.hpp>
```

Если вы этого не сделаете, то при компиляции получите сообщение об ошибке Typename expected or Undefined symbol.

Следующий код иллюстрирует эти шаги.

```
TRegistry& regKey = *new TRegistry();
bool keyGood = regKey.OpenKey(
    "Software\\SAMS\\C++Builder in 21 Days", false);
if (keyGood) {
  int top = regKey.ReadInteger("Top");
  int left = regKey.ReadInteger("Left");
  int width = regKey.ReadInteger("Width");
  int height = regKey.ReadInteger("Height");
  SetBounds(left, top, width, height);
}
delete &regKey;
```

Этот код открывает ключ и читает значение координат верхнего левого угла формы, а также ее ширину и высоту. Затем вызывается функция SetBounds(), чтобы переместить окно и/или изменить его размеры. Я хочу обратить ваше внимание на пару моментов. Прежде всего, я использую ссылку при создании экземпляра TRegistry. Это уменьшает беспорядок в остальной части кода. Мне не очень нравится оператор косвенного доступа (->), так что по возможности я стараюсь его избегать.

Во-вторых, обратите внимание, что результат, возвращаемый методом OpenKey(), присваивается переменной типа bool. OpenKey() возвращает true, если ключ был успешно открыт и false, если этого не произошло. Если ключ был успешно открыт, то читаются отдельные элементы данных. Вы должны всегда проверять значение, возвращаемое OpenKey(), если сомневаетесь в том, что ключ может быть открыт.



При неудачной попытке чтения или записи ключа, VCL генерирует исключения. Вы получите исключительную ситуацию, если попытаетесь читать данные из неоткрытого ключа. Либо подготовьтесь к обработке исключения, либо всегда проверяйте значение, возвращаемое OpenKey(), перед чтением или записью элементов данных.

Наконец, обратите внимание, что в приведенном коде ключ не был явно закрыт. Если вы не закроете ключ, деструктор TRegistry сделает это за вас. В данном случае деструктор будет вызван (и ключ будет закрыт) при разрушении объекта regKey, так что нет необходимости явно обращаться к CloseKey().



Функция OpenKey() автоматически предполагает, что корневым ключом является HKEY_CURRENT_USER, поэтому вам не нужно указывать корень, открывая ключ.

Здесь мы поставили телегу немного впереди лошади, поскольку говорим о чтении из реестра, ничего в него не записав. Это непринципиально — запись в реестр выполняется так же просто:

```
TRegistry& regKey = *new TRegistry();
regKey.OpenKey(
    "Software\\SAMS\\C++Builder in 21 Days", true);
regKey.WriteInteger("Top", Top);
regKey.WriteInteger("Left", Left);
regKey.WriteInteger("Width", Width);
regKey.WriteInteger("Height", Height);
delete &regKey;
```

Этот код просто открывает ключ и записывает в него свойства формы Top, Left, Width и Height, используя метод WriteInteger(). Обратите внимание, что последним аргументом метода OpenKey() является true. Это служит указанием на то, что ключ должен быть создан, если он еще не существует. При использовании такой конструкции вам вообще никогда не потребуется вызывать метод CreateKey().

В основном это все, что имеет отношение к чтению значений из реестра и записи в реестр. Методы чтения и записи данных других типов являются лишь вариациями предыдущего фрагмента кода. Листинги 14.3 и 14.4 содержат программу, которая использует реестр для сохранения относящихся

к приложению данных. Эта программа сохраняет в реестре следующую информацию: последний размер и положение окна, состояние окна (нормальное, свернутое или развернутое), последний каталог, последний файл и индекс последнего фильтра, который использовался при открытии файла через диалоговое окно File Open, дату и время последнего запуска программы. Чтобы очистить ключ реестра, созданный программой RegTest, вы можете щелкнуть на кнопке Delete Key главной формы (см. рис. 14.3).



Эта программа содержится также на компакт-диске, прилагаемом к книге.

Листинг 14.3. RegMain.h

```
 1: //---------------------------------------------------------------------
 2: #ifndef RegMainH
 3: #define RegMainH
 4: //---------------------------------------------------------------------
 5: #include <vcl\Classes.hpp>
 6: #include <vcl\Controls.hpp>
 7: #include <vcl\StdCtrls.hpp>
 8: #include <vcl\Forms.hpp>
 9: #include <vcl\Menus.hpp>
10: #include <vcl\Dialogs.hpp>
11: #include <vcl\Registry.hpp>
12: #include <vcl\ExtCtrls.hpp>
13: //---------------------------------------------------------------------
14: class TMainForm : public TForm
15: {
16: __published:    // Компоненты, управляемые IDE
17:   TMainMenu *MainMenu;
18:   TMenuItem *File1;
19:   TMenuItem *FileExit;
20:   TMenuItem *FileOpen;
21:   TOpenDialog *OpenDialog;
22:   TPanel *Panel1;
23:   TLabel *Label1;
24:   TButton *DeleteKey;
25:   TPanel *Panel2;
26:   TLabel *Label2;
27:   TLabel *Label3;
28:   TLabel *TimeLabel;
29:   TLabel *DateLabel;
30:   TLabel *Label4;
31:   void __fastcall FileOpenClick(TObject *Sender);
32:   void __fastcall FormCreate(TObject *Sender);
33:   void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
34:   void __fastcall FileExitClick(TObject *Sender);
35:   void __fastcall DeleteKeyClick(TObject *Sender);
36: private:        // Объявления пользователя
37:   void ParseFileName(String& path, String& fname);
38:   bool keyDeleted;
39: public:         // Объявления пользователя
40:   virtual __fastcall TMainForm(TComponent* Owner);
41: };
42: //---------------------------------------------------------------------
43: extern TMainForm *MainForm;
44: //---------------------------------------------------------------------
45: #endif
```

Листинг 14.4. RegMain.cpp

```
//---------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "RegMain.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------
__fastcall TMainForm::TMainForm(TComponent* Owner) : TForm(Owner)
{
  //
  // Инициализация переменной keyDeleted значением false. Эта
  // переменная используется, если пользователь удаляет ключ
  // из программы. См. функцию MainFormClose()
  //
  keyDeleted = false;
}
//---------------------------------------------------------------------
void __fastcall TMainForm::FormCreate(TObject *Sender)
{
  //
  // Создание объекта TRegistry для доступа к реестру
  //
  TRegistry& regKey = *new TRegistry();
  //
  // Открытие ключа
  //
  bool keyGood = regKey.OpenKey(
    "Software\\SAMS\\C++Builder in 21 Days", false);
  //
  // Проверка, открыт ли ключ. Если нет, это означает, что программа
  // выполняется первый раз, так что пока не нужно ничего делать.
  // Если ключ открыт, чтение всех элементов данных, имеющих
  // отношение к инициализации приложения
  //
  if (!keyGood) return;
  int top = regKey.ReadInteger("Top");
  int left = regKey.ReadInteger("Left");
  int width = regKey.ReadInteger("Width");
  int height = regKey.ReadInteger("Height");
  SetBounds(left, top, width, height);
  WindowState = TWindowState(regKey.ReadInteger("WindowState"));
  //
  // Класс TDateTime удобен для операций с датой и временем
  //
  TDateTime dt;
  dt = regKey.ReadDate("Date and Time");
  DateLabel->Caption = dt.DateString();
  TimeLabel->Caption = dt.TimeString();
  delete &regKey;
}
//---------------------------------------------------------------------
void __fastcall TMainForm::FileOpenClick(TObject *Sender)
{
  //
  // Эта функция отображает диалог File Open, но не открывает
```

```
59:    // файл. Последнее имя файла, фильтр и путь к файлу
60:    // записываются в реестр, когда пользователь нажимает OK
61:    //
62:    // Создание объекта TRegistry для доступа к реестру
63:    //
64:    TRegistry& regKey = *new TRegistry();
65:    //
66:    // Открытие ключа
67:    //
68:    regKey.OpenKey(
69:      "Software\\SAMS\\C++Builder in 21 Days", true);
70:    //
71:    // Чтение значений. Если эти элементы еще не были
72:    // созданы, мы получим исключение
73:    //
74:    try {
75:      OpenDialog->InitialDir = regKey.ReadString("LastDir");
76:      OpenDialog->FileName = regKey.ReadString("LastFile");
77:      OpenDialog->FilterIndex = regKey.ReadInteger("FilterIndex");
78:    }
79:    //
80:    // Мы знаем, что исключение сформировано, но нам это не мешает,
81:    // так что мы просто захватим его и проигнорируем
82:    //
83:    catch (...) {}
84:    //
85:    // Отображение диалога File Open. Если пользователь нажал OK,
86:    // запись в реестр пути, имени файла и фильтра
87:    //
88:    if (OpenDialog->Execute()) {
89:      String path;
90:      String fname;
91:      //
92:      // Функция для разделения пути и имени файла
93:      //
94:      ParseFileName(path, fname);
95:      regKey.WriteString("LastDir", path);
96:      regKey.WriteString("LastFile", fname);
97:      regKey.WriteInteger("FilterIndex", OpenDialog->FilterIndex);
98:    }
99:    delete &regKey;
100: }
101:
102: void TMainForm::ParseFileName(String& path, String& fname)
103: {
104:    //
105:    // Отделение пути от имени файла
106:    //
107:    int pos = 0;
108:    String s = OpenDialog->FileName;
109:    int x = 1;
110:    while (x) {
111:      x = s.Pos("\\");
112:      s = s.SubString(x + 1, s.Length() - x);
113:      pos += x;
114:    }
115:    fname = s;
```

см. продолжение

Листинг 14.4. (продолжение)

```
116:   path = OpenDialog->FileName.SubString(0, pos);
117: }
118: //---------------------------------------------------------------------
119: void __fastcall TMainForm::
120: FormClose(TObject *Sender, TCloseAction &Action)
121: {
122:   //
123:   // Если пользователь нажал кнопку для удаления ключа,
124:   // то информация в него не записывается
125:   //
126:   if (keyDeleted) return;
127:   //
128:   // Создание объекта TRegistry для доступа к реестру
129:   //
130:   TRegistry& regKey = *new TRegistry();
131:   //
132:   // Открытие ключа
133:   //
134:   regKey.OpenKey(
135:     "Software\\SAMS\\C++Builder in 21 Days", true);
136:   //
137:   // Запись значений
138:   //
139:   regKey.WriteInteger("WindowState", WindowState);
140:   if (WindowState != wsMaximized) {
141:     regKey.WriteInteger("Top", Top);
142:     regKey.WriteInteger("Left", Left);
143:     regKey.WriteInteger("Width", Width);
144:     regKey.WriteInteger("Height", Height);
145:   }
146:   TDateTime dt;
147:   regKey.WriteDate("Date and Time", dt.CurrentDateTime());
148:   delete &regKey;
149: }
150: //---------------------------------------------------------------------
151: void __fastcall TMainForm::FileExitClick(TObject *Sender)
152: {
153:   //
154:   // До свидания!
155:   //
156:   Close();
157: }
158: //---------------------------------------------------------------------
159: void __fastcall TMainForm::DeleteKeyClick(TObject *Sender)
160: {
161:   //
162:   // Удаление ключа, поскольку нажата кнопка Delete Key.
163:   // Установка флажка, чтобы не создавать ключ заново
164:   // при закрытии программы
165:   //
166:   TRegistry& regKey = *new TRegistry();
167:   regKey.DeleteKey("Software\\SAMS");
168:   keyDeleted = true;
169:   delete &regKey;
170: }
171: //---------------------------------------------------------------------
```

Изучая этот листинг и выполняя программу RegTest, вы можете многое узнать про использование реестра в своих приложениях. На рис. 14.3 показана работающая программа RegTest, а рис. 14.4 показывает ключ реестра, созданный этой программой.

Рис. 14.3
Работающая программа RegTest

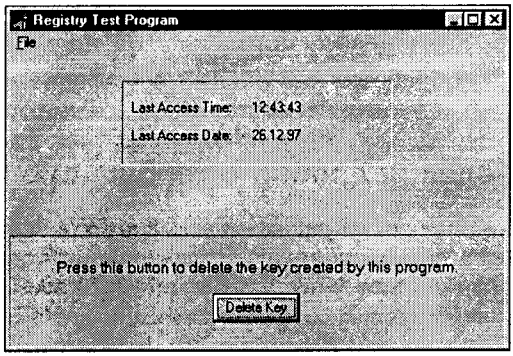
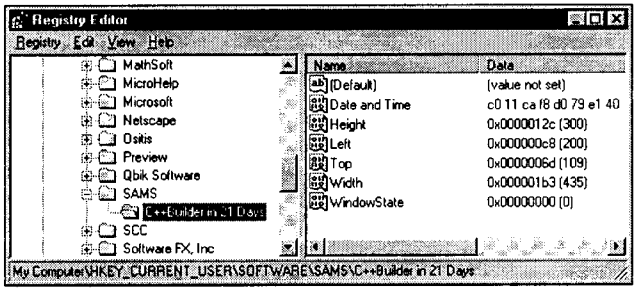


Рис. 14.4
Редактор реестра, показывающий ключ RegTest





Существует специальный класс TRegIniFile, предназначенный для манипуляций с реестром в приложениях, переходящих от сохранения данных в .INI-файле к сохранению данных в реестре. Подробная информация о TRegIniFile содержится во встроенной справочной системе.



Хотя логичнее всего сохранять относящиеся к приложению данные в реестре, вы можете по-прежнему использовать .INI -файлы, если они вам нравятся. Эти файлы имеют то преимущество, что менее искушенные пользователи при необходимости могут редактировать их в текстовом редакторе (это же является и недостатком в некоторых случаях). Чтобы упростить использование .INI -файлов, в VCL включен класс TIniFile. Обязательно ознакомьтесь с возможностями этого класса, если вы предполагаете использовать в своем приложении файлы конфигурации.

Специальная обработка сообщений

О сообщениях Windows говорилось в 11-й день, «Работа с отладчиком», когда мы рассматривали программу WinSight. В большинстве случаев обработка сообщений в C++Builder выполняется через события. Событие обычно

генерируется в ответ на сообщение Windows, посланное приложению. Однако бывают ситуации, когда вам может потребоваться обработать сообщение самостоятельно. Существуют два основных типа сообщений, которые требуют обработки в обход обычной системы сообщений C++Builder:

- Сообщения Windows, не обрабатываемые VCL
- Сообщения, определяемые пользователем

Обработка сообщений собственными силами требует применения некоторой дополнительной техники, о которой вы узнаете из этого раздела.

Более подробно о сообщениях Windows

Каким образом возникают сообщения Windows? Некоторые сообщения посылаются непосредственно Windows для указания окну на необходимость выполнения каких-либо действий или уведомления окна о некотором событии в системе. В других случаях сообщения посылаются программистом или библиотекой, которую он использует (VCL, OWL или MFC). Независимо от того, кто посылает сообщение, можете не сомневаться, что каждую миллисекунду в системе циркулирует множество сообщений.

Типы сообщений

В основном, сообщения делятся на две категории:

- Командные сообщения
- Уведомляющие сообщения

Командные сообщения вызывают какое-то действие либо в Windows, либо внутри определенного элемента управления. Уведомляющие сообщения посылаются Windows для того, чтобы информировать вас о некотором событии. Чтобы проиллюстрировать различие между этими двумя типами сообщений, посмотрим на стандартный элемент редактирования.

Стандартный элемент редактирования Windows имеет почти 80 командных сообщений и около 20 уведомляющих сообщений. Удивлены? Это действительно так, и при этом не забудьте, что элемент редактирования — это только один из множества элементов управления Windows.



Я был несколько неточен, когда говорил о событиях в 5-й день, «Библиотеки классов Си++ и модель визуальных компонентов». Тогда я сказал, что существует более 175 сообщений, которые Windows может послать приложению, и это действительно правда. Но когда вы рассматриваете сообщения, относящиеся к элементам управления, вместе с сообщениями, которые могут быть посланы главным окнам, общее их число достигает приблизительно 700. И сюда включены только командные сообщения, без уведомляющих! После прочтения этого раздела вы, возможно, еще больше оцените VCL.

Программист, пишущий приложение для Windows на Си, должен посылать множество сообщений для выполнения стоящей перед ним задачи. Например, чтобы получить значение длины выбранного в данный момент текста в элементе редактирования, необходим следующий код:

```
int start;
int end;
long result = SendMessage(hWndEdit, EM_GETSEL, 0, 0);
```

```
start = LOWORD(result);
end = HIWORD(result);
int length = end - start;
```

Именно на это тратят свое время программисты на Си (или программисты на Си++, не использующие библиотеки). Теперь сопоставьте приведенную процедуру с тем, как делает то же самое VCL:

```
int length = Edit->SelLength;
```

Что вы предпочтете? Независимо от своего источника, командные сообщения используются очень интенсивно как программистами, так и самой Windows.

Уведомляющие сообщения, с другой стороны, посылаются только самой средой Windows. Эти сообщения говорят Windows, что что-то произошло внутри элемента управления. Например, сообщение EN_CHANGE посылается после изменения содержимого элемента редактирования. Для компонентов Edit, MaskEdit и Memo в VCL определено событие OnChange, которое генерируется в ответ на уведомляющее сообщение EN_CHANGE (а также на некоторые другие). Программисты, использующие традиционные инструментальные средства программирования в Windows, должны перехватывать эти сообщения и обрабатывать их так, как им это необходимо. Все это переводит нас к следующей теме: параметры сообщения.



Традиционные программы на Си для Windows содержат длинный оператор switch, который используется для определения того, какое сообщение было послано окну.

WPARAM, LPARAM и раскрытие сообщений

Каждое сообщение Windows имеет два параметра: WPARAM (сокращение от word parameter — параметр с длиной в одно машинное слово) и LPARAM (сокращение от long parameter — параметр типа long).



В 32-разрядной среде Windows оба эти параметра являются 32-разрядными числами. В 16-разрядной Windows, WPARAM — это 16-разрядное число (слово), а LPARAM — это 32-разрядное число (двойное слово, или long). На этом закончим экскурс в историю имен WPARAM и LPARAM и вернемся к нашим программам.

Эти два параметра могут рассматриваться аналогично параметрам функции. После получения сообщения Windows, параметры можно проверить на предмет извлечения информации, относящейся к посланному сообщению. Например, сообщение WM_LBUTTONDOWN — это уведомляющее сообщение, которое посылается после щелчка левой кнопкой мыши в пределах окна. WPARAM для сообщения WM_LBUTTONDOWN содержит специальный код, который сообщает, какие другие кнопки мыши, а также клавиши на клавиатуре, были нажаты, когда произошло данное событие. LPARAM содержит координаты курсора мыши в момент щелчка. Координата x содержится в младшем слове, а координата y — в старшем слове.

Чтобы извлечь необходимую информацию, сообщение должно быть *раскрыто (cracked, буквально — расколото)*. Код для обработки сообщения WM_LBUTTONDOWN мог бы выглядеть следующим образом:

```
int Shift = wParam;
int X = LOWORD(lParam);
int Y = HIWORD(lParam);
```

Это довольно простой пример, но он показывает то, что происходит при каждой обработке сообщения в приложении Windows.

Я рад вам сообщить, что для всех событий VCL сообщения раскрываются без вашего участия. Например, если вы пишете обработчик сообщения для события OnMouseDown, сгенерированная C++Builder функция будет выглядеть следующим образом:

```
void __fastcall TScratchPad::FormMouseDown(TObject *Sender,
TMouseButton Button, TShiftState Shift, Integer X, Integer Y)
{
  // введите здесь свой код
}
```

Как видите, обработчик события, сгенерированный C++Builder, содержит всю нужную вам информацию. VCL раскроет WPARAM и LPARAM и вручит вам по частям, чтобы вы могли сразу начать работу. То же самое происходит с каждым сообщением, для которого VCL инициирует событие.



В случае кнопок мыши VCL выполняет еще больше предварительной работы с сообщениями. Фактически Windows генерирует до трех разных сообщений: одно для левой кнопки, одно для средней кнопки (если она существует) и одно для правой кнопки. VCL обрабатывает все эти сообщения в одном событии. Параметр TMouseButton события OnMouseDown сообщает вам, какая из трех кнопок был нажата.

После этого небольшого введения в систему сообщений Windows, давайте рассмотрим, как же посылаются сообщения.

Отправление и пересылка

API Windows содержит две функции для посылки сообщений: PostMessage() и SendMessage(). Функция PostMessage() отправляет сообщение в очередь сообщений Windows и немедленно возвращает управление. Эта функция просто вручает сообщение Windows, после чего программа продолжает работу. PostMessage() возвращает 1, если вызов функции прошел успешно, и 0 в противном случае. (Практически единственная причина, по которой PostMessage() может быть не выполнена — сообщение направлено недействительному окну.)

Функция SendMessage(), напротив, посылает сообщение Windows и не возвращает управление до тех пор, пока сообщение не будет доведено до получателя. Значение, возвращаемое SendMessage(), зависит от пересылаемого сообщения. Иногда основной причиной для использования SendMessage() вместо PostMessage() является необходимость получить возвращаемое значение от конкретного сообщения.



Различие в ситуациях, когда вы используете PostMessage() и SendMessage(), относится к достаточно тонкой сфере. Например, из-за особенностей синхронизации внутри Windows существуют случаи, когда пересылка сообщения не приведет к тому эффекту, который вы ожидали. В этих случаях вы должны будете использовать PostMessage(). Обратное также верно: иногда PostMessage() не работает в данной ситуации, и вы вынуждены использовать SendMessage(). Хотя это и не понадобится вам прямо сейчас, запомните сказанное для своей будущей работы.

Вы можете использовать в своих приложениях как PostMessage(), так и SendMessage(). Например, чтобы отправить себе сообщение, вы можете написать что-нибудь наподобие следующего:

```
PostMessage(Handle, WM_QUIT, 0, 0);
```

Первый параметр в PostMessage() и в SendMessage() — это дескриптор окна, которому вы хотите послать сообщение. В данном случае мы посылаем сообщение главной форме (предполагая, что эта строка содержится в ее исходном модуле).

В дополнение к функциям API Windows, VCL содержит метод с названием Perform(), который вы можете вызывать для посылки сообщений любому окну VCL. Perform() обходит систему передачи сообщений Windows и направляет сообщение непосредственно механизму обработки сообщений данного окна. Эквивалент предыдущей строки с использованием Perform() выглядит следующим образом:

```
Perform(WM_QUIT, 0, 0);
```

Чтобы посылать сообщения своему приложению и другим окнам внутри приложения, вы можете использовать любую из этих трех функций.

Обработка событий

Мы уже обсуждали обработку событий в VCL, но краткий обзор не помешает. Когда определенный компонент получает сообщение Windows, он выясняет, существует ли обработчик для события, соответствующего этому сообщению. Если обработчик события существует, он вызывается. Если данному событию не был назначен обработчик, сообщение обрабатывается заданным по умолчанию способом. Вы можете обрабатывать любые сообщения, которые вам требуются, и игнорировать остальные.

То, что происходит в ваших обработчиках событий, зависит от ряда факторов: специфики обрабатываемого сообщения, действий вашей программы в ответ на сообщение, а также, изменяете ли вы входящее сообщение или только используете обработчик события для уведомления о том, что событие произошло. Когда вы полнее познакомитесь с программированием в Windows, вы увидите, что существуют буквально сотни вещей, которые можно сделать в ответ на событие.

В большинстве случаев вы будете использовать обработчик события для уведомления о том, что произошло определенное событие. Возьмем, например, событие OnMouseDown. Обрабатывая его, вы просто хотите знать, что пользователь щелкнул мышью на компоненте (не забудьте, что формы — это тоже компоненты). Вы, возможно, не собираетесь как-либо менять параметры этого сообщения; вы только хотите знать, что событие произошло. Большая часть ваших обработчиков событий будет использоваться именно для целей уведомления.

Иногда вам может потребоваться изменить один или более параметров сообщения перед пересылкой этого сообщения получателю. Предположим, например, что вам требуется перевести в верхний регистр текст определенного элемента редактирования. (Конечно, для этого можно установить свойство CharCase на ecUpperCase, но сейчас это нас не интересует.) Чтобы перевести все нажимаемые клавиши в верхний регистр, вы могли бы сделать в своем обработчике события OnKeyPress следующее:

```
void __fastcall TScratchPad::EditKeyPress(TObject* Sender, Char& Key)
{
  if (Key >= 'a') Key -= 32;
}
```

Таким образом, вы изменяете сообщение прежде, чем оно будет послано VCL для обработки. Именно по этой причине параметры обработчика события часто передаются по ссылке.



C++Builder в некоторой степени ограждает вас от ошибок, передавая параметры сообщения, которые вы не должны изменять, по значению, а не по ссылке. Это относится и к тем параметрам сообщения, изменение которых не дает никакого эффекта. Возьмем событие OnMouseDown, которое мы обсуждали ранее. Это событие служит только для уведомления. Не имеется никакого смысла модифицировать параметры Button, Shift, X или Y, поэтому все они передаются по значению.

Модифицируете ли вы один или несколько посылаемых параметров, полностью зависит от сообщения и того, что вы предполагаете делать с этим сообщением. Через какое-то время вы, несомненно, столкнетесь с ситуациями, когда потребуется изменить конкретное сообщение, чтобы заставить Windows вести себя определенным образом. В других случаях вы вообще не будете изменять параметры, но проверите их значения для определения того, что происходит с вашим приложением. Поскольку существует неисчислимое количество возможностей, я оставлю дальнейшее исследование параметров сообщения в качестве упражнения для читателя.

Обработка других сообщений Windows

В определенный момент, несомненно, вы столкнетесь с ситуацией, когда нужно будет ответить на сообщение Windows, для которого в VCL нет соответствующего события. Когда это произойдет, вам нужно будет знать, как обработать такое сообщение. Именно этому и посвящен данный раздел.

VCL имеет события лишь для наиболее часто используемых сообщений Windows. Очевидно, что невозможно обеспечить событиями более 700 сообщений Windows. Теория 80/20 говорит, помимо всего прочего, что 20 процентов людей делает 80 процентов работы. Вероятно, то же самое верно и для сообщений Windows. VCL может обеспечить событиями только 10 или 20 процентов сообщений Windows, но это те сообщения, которые вы используете в 90 процентах случаев. Таким образом, остается много сообщений, для которых в VCL нет адекватных событий, и вы должны знать, как их обрабатывать.

Я думаю, вы будете рады узнать, что можно обработать практически любое сообщение Windows, если только знать как. Когда вы изучили основы, каждое новое сообщение — это только вариация на ту же тему. Механизм, который C++Builder использует для обработки сообщений, не имеющих VCL событий, называется *таблицей отображения сообщений (message map table)*. Таблица отображения сообщений связывает некоторое сообщение Windows с определенной функцией в вашем коде. Когда ваше окно получает это сообщение, вызывается соответствующая ему функция. Похоже на события, не так ли? Действительно, некоторая аналогия существует.

Обработка сообщений на этом уровне реализуется в три шага.

1. Создайте таблицу отображения сообщений в объявлении класса и добавьте в нее сообщение, которое вы хотите обработать.
2. Добавьте объявление функции обработчика сообщения в объявление класса.
3. Добавьте определение обработчика сообщения в модуль исходного кода вашей формы.

Сначала позвольте мне показать код, реализующий отображение сообщений для вашего класса; затем мы рассмотрим функцию обработки сообщения.

Таблица отображения сообщений C++Builder

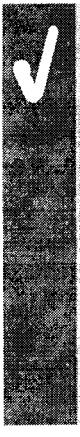
Таблица отображения сообщений располагается в конце определения вашего класса. Она начинается с BEGIN_MESSAGE_MAP и заканчивается END_MESSAGE_MAP. Между этими двумя элементами находится одно или несколько объявлений MESSAGE_HANDLER. Вы передаете имя базового класса в качестве единственного параметра в END_MESSAGE_MAP. Даже при сегодняшней инфляции картинка все еще стоит тысячи слов, так что позвольте мне продемонстрировать вам типовую таблицу отображения сообщений. Это выглядит следующим образом:

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_NCHITTEST, TWMNCHitTest, WMNCHitTest)
  MESSAGE_HANDLER(WM_ERASEBKGND, TWMEraseBkgnd, WMEraseBkgnd)
END_MESSAGE_MAP(TForm)
```

Эта таблица настраивает программу на обработку двух сообщений Windows: WM_NCHITTEST и WM_ERASEBKGND.



Сообщение WM_ERASEBKGND следует непосредственно перед сообщением WM_PAINT (последнее посылается, когда Windows перерисовывает окно). В программировании с VCL это означает, что WM_ERASEBKGND принимается перед тем, как произойдет событие OnPaint. Вы можете перехватить сообщение WM_ERASEBKGND, чтобы нарисовать что-либо в окне независимо от обработчика события OnPaint. (Когда вы реагируете на сообщение Windows, это называется перехватом (catch) или обработкой (handle) сообщения.)



Вы могли бы задать вопрос, «Почему все буквы заглавные?». Причина состоит в том, что BEGIN_MESSAGE_MAP, END_MESSAGE_MAP и MESSAGE_HANDLER являются макросами, а имена макросов традиционно записываются в верхнем регистре. Макросы — это текстовые метки-заполнители, которые расширяются до фактического кода Си++, когда синтаксический анализатор (часть компилятора) с ними сталкивается. Макроопределения для таблицы отображения сообщений вынесены в исходный код VCL.

Для тех из вас, кому не терпится узнать как можно больше, рассказываю дальше. Все вместе макросы таблицы отображения сообщений создают функцию, которая называется Dispatch(). Поскольку таблица отображения сообщений содержится в объявлении класса, это автоматически делает функцию подставляемой.

Если раскрыть все макросы предыдущего примера, то это выглядело бы примерно так:

```
virtual void __fastcall Dispatch(void* Message)
{
  switch (((PMessage)Message)->Msg) {
    case WM_NCHITTEST :
      OnNCHitTest((TWMNCHitTest&)*Message);
      break;
    case WM_ERASEBKGND :
      OnEraseBkgnd((TWMEraseBkgnd&)*Message);
      break;
    default:
      TForm::Dispatch(Message);
      break;
  }
}
```

В раскрытом виде все выглядит уже не так таинственно. Если вы хотите узнать, как выглядят макросы, загляните в файл \CBuilder\include\vcl\Sysdefs.h.

Давайте раскроем макрос обработчика MESSAGE_HANDLER и исследуем его. Здесь я буду использовать одну из записей синтаксиса, которые я очень не люблю:

```
MESSAGE_HANDLER(<message>, <message struct>, <message handler>)
```

Параметр message — это сообщение Windows, которое вы хотите обработать. Имена сообщений Windows всегда пишутся в верхнем регистре (во всяком случае, имена документированных сообщений).

Параметр message struct используется для передачи имени структуры, которая будет содержать параметры сообщения после его раскрытия VCL. VCL содержит объявления соответствующих структур для большинства сообщений Windows. Имя структуры — это имя сообщения Windows с предшествующим символом T и без символа подчеркивания. Например, именем структуры для сообщения Windows WM_ERASEBKGND является TWMEraseBkgnd. Вы могли предположить, что имя структуры также будет записано в верхнем регистре, но в большинстве случаев существующая запись полностью оправдана. Эта структура передается обработчику сообщения (подробнее об этом чуть позже).

Параметр message handler — это фактическое имя функции, обрабатывающей сообщение. Вы можете использовать любое имя, которое вам нравится, но по традиции (я ввожу традицию прямо сейчас) вы удаляете из имени структуры приставку TWM и добавляете предлог On. Если опять обратиться к WM_ERASEBKGND, то имя функции обработчика сообщения должно быть OnEraseBkgnd.

Чтобы усвоить эту информации на будущее, вы должны увидеть полное объявление класса, в котором реализована таблица отображения сообщений. В листинге 14.5 показано типичное объявление класса главной формы, который использует таблицу отображения сообщений.

Листинг 14.5. Message.h

```
1: //---------------------------------------------------------------------
2: #ifndef MsgMainH
3: #define MsgMainH
4: //---------------------------------------------------------------------
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
```

```
 7: #include <vcl\StdCtrls.hpp>
 8: #include <vcl\Forms.hpp>
 9: #include <vcl\ExtCtrls.hpp>
10:
11: #define MYMESSAGE WM_USER + 1
12:
13: //---------------------------------------------------------------------
14: class TMainForm : public TForm
15: {
16: __published:    // Компоненты, управляемые IDE
17:   TButton* ShowTBar;
18:   TButton* Button1;
19:   TLabel* Label;
20:   TBevel* Bevel1;
21:   TCheckBox* Hatched;
22:   TCheckBox* LetVCLHandle;
23:   void __fastcall ShowTBarClick(TObject* Sender);
24:   void __fastcall Button1Click(TObject* Sender);
25:   void __fastcall HatchedClick(TObject* Sender);
26: private:         // Объявления пользователя
27:   void __fastcall OnNCHitTest(TWMNCHitTest& Message);
28:   void __fastcall OnEraseBkgnd(TWMEraseBkgnd& Message);
29:   void __fastcall OnGetMinMaxInfo(TWMGetMinMaxInfo& Message);
30:   void __fastcall OnMyMessage(TMessage& Message);
31: public:          // Объявления пользователя
32:   virtual __fastcall TMainForm(TComponent* Owner);
33: BEGIN_MESSAGE_MAP
34:   MESSAGE_HANDLER(MYMESSAGE, TMessage, OnMyMessage)
35:   MESSAGE_HANDLER(WM_NCHITTEST, TWMNCHitTest, OnNCHitTest)
36:   MESSAGE_HANDLER(
37:     WM_GETMINMAXINFO, TWMGetMinMaxInfo, OnGetMinMaxInfo)
38:   MESSAGE_HANDLER(WM_ERASEBKGND, TWMEraseBkgnd, OnEraseBkgnd)
39:   END_MESSAGE_MAP(TForm)
40: };
41: //---------------------------------------------------------------------
42: extern TMainForm* MainForm;
43: //---------------------------------------------------------------------
44: #endif
```

Обратите внимание на таблицу отображения сообщений в конце объявления класса и на объявление функций для обработки сообщений. (Не принимайте во внимание строки 11, 30 и 34. В них реализуется определяемое пользователем сообщение, о чем я буду говорить немного позже.)

Функция обработки сообщения

Функция обработки сообщения (или просто *обработчик сообщения*) — это функция, которая будет вызываться каждый раз, когда сообщение, на которое вы реагируете, получено вашим приложением. Обработчик сообщения имеет единственный параметр — структуру, раскрывающую сообщение, о которой я говорил раньше. Обработчик для сообщения WM_ERASEBKGND может выглядеть следующим образом:

```
void __fastcall TMainForm::OnEraseBkgnd(TWMEraseBkgnd& Message)
{
  // ваш код обработки сообщения
}
```

Как я уже говорил, раскрывающая структура будет содержать все параметры, необходимые для обработки сообщения. Структура, раскрывающая сообщение WM_ERASEBKGND, имеет следующий вид:

```
struct TWMEraseBkgnd
{
  unsigned int Msg;
  HDC DC;
  long Unused;
  long Result;
};
```

Все раскрывающие структуры имеют, как правило, два члена-данных: Msg и Result. Член Msg содержит сообщение, которое было послано. Этот параметр используется VCL и не относится непосредственно к вам.

Член Result более важен. Он содержит возвращаемое значение для сообщения, которое вы обрабатываете. Возвращаемое значение изменяется от сообщения к сообщению. Например, значение, возвращаемое вашим обработчиком сообщения WM_ERASEBKGND, должно быть true, если вы стираете фон до начала рисования, или false, если вы не стираете фон. (Чтобы определить, какие значения может принимать Result, обратитесь к интерактивной справке API Win32 по конкретному сообщению, которое вы обрабатываете.) При необходимости установите для члена Result нужное значение:

```
void __fastcall TMainForm::OnEraseBkgnd(TWMEraseBkgnd& Message)
{
  // какие-то действия
  Message.Result = false;
}
```

Любые другие члены-данные раскрывающей структуры меняются от сообщения к сообщению.

Иногда в дополнение к вашей собственной обработке, вам придется вызывать заданный по умолчанию обработчик конкретного сообщения. В этом случае вы можете вызвать DefaultHandler() или метод базового класса Dispatch(). Какой из них использовать — зависит от последующих действий. Например, при одних обстоятельствах вам может потребоваться закрасить фон вашего окна, а при других — нет. Если вы не закрашиваете фон, то оставьте эту работу VCL, которая закрасит фон заданным по умолчанию способом. В этом случае можно сделать следующее:

```
void __fastcall TMainForm::OnEraseBkgnd(TWMEraseBkgnd& Message)
{
  if (LetVCLDoIt) {
    TForm::Dispatch(&Message); // передача работы VCL
  return;
  }
  else {
    // закрашивание фона
  }
  Message.Result = false;
}
```

В других случаях вы будете вызывать DefaultHandler() для выполнения некоторой обработки по умолчанию. Вызываете ли вы DefaultHandler() до своей обработки или после нее, зависит от того, что вы хотите получить.

Сообщения, определяемые пользователем

Помимо нормальных сообщений Windows, вы можете создать *сообщение, определяемое пользователем (user-defined message)*. Определяемое пользователем сообщение — это частное сообщение, которое вы можете послать самому себе или одному из окон вашего приложения. Существует ряд причин, по которым вам могут потребоваться такие сообщения. События VCL уменьшают потребность в использовании своих собственных сообщений, но все еще могут встретиться случаи, когда без них не обойтись. Предположим, например, что вы выполняете длинное вычисление в отдельном процессе. (Процессы не будут сегодня рассматриваться.) Если там происходит что-то такое, о чем приложение должно знать, процесс может послать приложению специальное сообщение.

Реализация и перехват определяемого пользователем сообщения почти идентичны обработке сообщения Windows. Единственное отличие состоит в том, что вы должны сначала определить сообщение. Это делается следующим образом:

```
#define MYMESSAGE WM_USER + 1
```

Эта строка устанавливает определяемое пользователем сообщение с именем MYMESSAGE.



Символ WM_USER — это специальный символ, отмечающий начало диапазона чисел, которые могут использоваться для определяемых пользователем сообщений. Вы можете использовать диапазон чисел примерно до WM_USER + 31743. Я обычно выбираю что-нибудь в диапазоне WM_USER + 100. Точное число, которое вы используете, не важно; только убедитесь в том, что вы случайно не установили одно и то же значение для двух разных сообщений.

Вернитесь к листингу 14.5 и обратите внимание на объявление сообщения в строке 11. Как только вы определили сообщение, вы можете добавить его в таблицу отображения сообщений следующим образом:

MESSAGE_HANDLER(MYMESSAGE, TMessage, OnMyMessage)

Заметьте, что раскрывающая структура сообщения имеет тип TMessage. Это структура общего вида, которая определена следующим образом:

```
struct TMessage
{
  unsigned int Msg;
  long WParam;
  long Lparam;
  long Result;
};
```



Фактическое объявление TMessage выглядит немного по-другому. Структура, показанная здесь, представляет собой конечный результат.

Когда вы посылаете сообщение, определяемое пользователем, то передаете любые параметры, которые вам нужны, через члены WParam и LParam. Предположим, например, что вы послали сообщение, указывающее, что произошла ошибка с кодом 124 на 1019 итерации цикла обработки. Тогда вызов Perform() мог бы выглядеть следующим образом:

```
int result = MainForm->Perform(MYMESSAGE, 124, 1019);
```

Конечно, для посылки сообщения вы также могли бы использовать PostMessage() или SendMessage(). Для определяемых пользователем сообщений вы, возможно, будете использовать Perform() в большинстве случаев.

Итак, сообщение определено и послано. Теперь вы должны написать код для его обработки. Обработчик сообщения для MYMESSAGE мог бы выглядеть следующим образом:

```
void __fastcall TMainForm::OnMyMessage(TMessage& Message)
{
  char msg[100];
  sprintf(msg,
    "Error #%d occurred on iteration number %d.",
    Message.WParam, Message.LParam);
  Application->MessageBox(msg, "Error Message", MB_OK);
  Message.Result = true;
}
```

Значение, возвращаемое Perform(), будет значением члена Result структуры TMessage. Будете ли вы посылать параметры, и будет ли ваш обработчик сообщения возвращать результат, полностью зависит от вас.



Компакт-диск, прилагаемый к книге, содержит программу с названием Message, которая иллюстрирует обработку сообщений Windows, не охваченных событиями VCL. Она также демонстрирует применение сообщения, определяемого пользователем. В этой программе реализованы некоторые трюки, использующиеся при программирования в Windows: вы можете перетаскивать окно, нажав кнопку мыши в любой точке рабочей области.

Заключение

Итак, сегодня было рассмотрено много базового материала. Мы начали с контекстно-зависимой справки и ее включения в программы. Несмотря на то, что создание контекстной справки не обязательно будет простой операцией, вам следует ее использовать. После этого вы узнали, как захватывать и обрабатывать исключения VCL. Вы также получили некоторое представление о реестре Windows. Реестр нужно и должно использовать для сохранения данных, относящихся к приложению. Знание того, как использовать реестр, может помочь вам реализовать приятные для пользователей мелочи. Мы завершили день обработкой сообщений без использования событий. В целом это был долгий, но полезный день.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Создание справочных файлов в текстовом процессоре очень утомительно. Что вы можете предложить?

Возьмите коммерческую программу для автоматизированного создания справки. Эти программы берут на себя ту мелкую работу, которая сводит с ума, когда вы пишете справочный файл вручную. Создание справочных

файлов — не очень веселое занятие для большинства из нас, но хорошая программа создания справки действительно может помочь.

Должен ли я включать в свой справочный файл идентификаторы контекста?

Не обязательно. Вы можете предоставить своим пользователям только вызов справки из главного меню. Но если вы не поместите идентификаторы контекста в справочный файл, то не сможете обеспечить контекстно-зависимую справку.

Для чего я должен беспокоиться об обработке исключений?

Обработка исключений позволяет вам в максимальной степени контролировать работу своей программы при возникновении ошибок.

Я перехватил исключение VCL. Как мне воспроизвести окно сообщения об ошибке, которое отображает VCL при формировании исключения?

Ответ: Вызовите Application->ShowException(), и VCL выведет сообщение об ошибке.

Должен ли я использовать реестр для сохранения установок своей программы?

Нет. Вы может использовать .INI-файлы или любой другой тип конфигурационного файла. Однако реестр — это более предпочтительное место для сохранения данных, относящихся к приложению. Работу с реестром упрощает использование класса TRegistry.

Я получаю исключения каждый раз, когда пробую создать ключ, а затем использовать WriteString() для записи элемента данных в этот ключ. В чем может состоять ошибка?

Возможно, вы используете CreateKey() для создания ключа. CreateKey() создает ключ, но не открывает его. Используйте для создания ключа OpenKey(), а не CreateKey().

Что такое сообщение, определяемое пользователем?

Определяемое пользователем сообщение — это сообщение, определенное вами, пользователем, для использования в пределах своего приложения. Противопоставьте это сообщениям Windows, которые определены и используются Windows на глобальном уровне.

Что я должен сделать, чтобы получить заданное по умолчанию поведение при обработке конкретного сообщения Windows?

Вызовите метод Dispatch():

```
TForm::Dispatch(&Message);
```

Вопросы

1. Как задать справочный файл, который будет использовать ваше приложение?
2. Как реализовать поддержку клавиши F1 для конкретной формы или диалогового окна?
3. Какую функцию вы вызовете, чтобы вывести индекс вашего справочного файла?

4. Какие типы объектов могут формироваться в исключительных ситуациях?
5. Допускается ли иметь более одного оператора catch, следующих за оператором try?
6. Как сформировать исключение?
7. Какой корневой ключ (RootKey) установлен по умолчанию для класса TRegistry?
8. Должны ли вы вызывать CloseKey() после завершения работы с ключом?
9. В чем заключается разница между SendMessage() и PostMessage()?
10. Как называется функция VCL, которая посылает сообщение непосредственно компоненту?

Упражнения

1. Исследуйте доступность средств для автоматизированного создания справочных файлов. Хотя это упражнение может показаться странным, оно могло бы стать самым полезным по теме справочных файлов.
2. Создайте новый проект. Добавьте в главную форму несколько компонентов. Присвойте каждому компоненту различные значения HelpContext.
3. Свяжите с проектом справочный файл (подойдет любой справочный файл). Если у вас есть программа создания справки, создайте простой справочный файл специально для этой программы. Запустите программу и нажмите кнопку Fl, когда компонент находится в фокусе.
4. Создайте новый проект. Добавьте в главную форму кнопку. Введите в обработчик события OnClick кнопки следующий код:

```
TImageList* list;
delete list;
```

Запустите программу и щелкните на кнопке. Что при этом происходит?

5. Введите пару операторов try/catch для двух строк кода из упражнения 4, чтобы захватывать любые исключения. Выведите сообщение, когда происходит ошибка, но не прерывайте после этого выполнение программы.
6. Модифицируйте программу ScratchPad таким образом, чтобы в ней использовался реестр Windows. Сохраните в реестре имя и путь файла, который был открыт последним.
7. Модифицируйте программу ScratchPad таким образом, чтобы она использовала имя и путь файла, сохраненные в реестре, при выводе диалоговых окон File Open и File Save.
8. Напишите программу, которая посылает себе сообщение, определенное пользователем, после щелчка на кнопке. Выведите информационное окно после получения сообщения.
9. Добавьте в программу из упражнения 8 обработчик сообщения Windows MW_MOVE. При перемещении окна программа должна выдавать звуковой сигнал и выводить в форме новые координаты.
10. **Дополнительное задание**: модифицируйте программу PictureViewer из 6-го дня, «Изучение IDE C++Builder: проекты и формы», чтобы захватывать исключение в тех случаях, когда пользователь пытается открыть файл, не являющийся графическим.

Подведение итогов



Ну как, много информации, да? Но вы получили удовольствие? Или вас начало лихорадить? Держу пари, что к этому моменту колеса в вашей голове действительно провернулись! Вероятно, вы уже представили себе одно или два собственных приложения. Возможно, вы даже начали работать над приложением. Я надеюсь, что это так, поскольку я много раз говорил, каким образом можно действительно чему-то научиться. Если у вас еще не появилась идея для какого-либо приложения, не волнуйтесь: она придет в свое время.

Эта неделя включала разнородный материал. В самом начале вы разобрались с основами построения приложений C++Builder. Вы можете сколько угодно размещать компоненты в форме, но когда-нибудь нужно начать писать код. Это немного страшно — пускаться в самостоятельное путешествие. До этого момента C++Builder в каком-то смысле вел вас за руку. Но настало время оставить гнездо. Вы выяснили, как включить в свой код собственные функции и члены-данные. Вы выяснили, как включить в свою программу такие ресурсы, как растровые изображения и звук. Это хороший багаж. Скоро вы будете делать все это самостоятельно, как профи.

Несколько менее захватывающий материал этой недели — подробное изучение проектов C++Builder и отладчика интегрированной среды. Эти главы могли не произвести особого впечатления, но они содержали ту информацию, которая необходима вам при разработке приложений. Все это имеет отношение к оптимизации затрат вашего времени. Изучение отладчика требует нескольких часов или даже дней, но он, в конечном счете, сэкономит вам недели работы. Поверьте мне! Если вы знаете, как использовать отладчик, то легко сможете выяснить, что происходит, когда вы сталкиваетесь с ошибкой в своей программе. Если вы не задержались на отладчике, я настоятельно советую вам вернуться и просмотреть 11-й день. Как я уже говорил, отладчик — это не самое захватывающее из того, о чем вы узнаете из этой книги, но, без сомнения, он будет одной их самых полезных вещей. Изучение эффективного использования ваших проектов относится к той же теме. Рациональное управление проектом, в конечном счете, также сэкономит вам время.

Ближе к концу недели вы узнали о некоторых вещах, которые могут превратить среднюю программу в программу высокого класса. Конечно, сами по себе декоративные элементы окна, такие как строки состояния и оперативные панели, не могут компенсировать слабую идею, лежащую в основе

программы. Никакое количество приспособлений не может это сделать. Но если у вас с самого начала есть хорошая программа, то различные «примочки» могут заставить ее действительно оторваться от конкурентов. Ничто не может препятствовать включению этого типа украшений в свои приложения, поскольку в C++Builder они реализуются очень просто! Это не реклама, а чистейшая правда.

Вы также узнали, как выполнять печать из своих приложений. Печать, по своей природе, нельзя визуализировать на стадии разработки. Развитые визуальные средства программирования C++Builder не могут здесь помочь. Тем не менее, VCL делает печать намного удобнее, чем при прямом использовании API. Как только вы приступите к печати, то обнаружите, что она совсем нетрудна. И еще раз, экспериментируйте столько, сколько можете. Обыгрывание различных сценариев может быть самым лучшим учителем.

Наконец, вы закончили неделю, осваивая более сложную технику программирования. Реализация контекстно-зависимой справки — это то, что вы вынуждены делать при сегодняшней конкуренции на рынке. Хорошая новость состоит в том, что в C++Builder это делается проще. Плохая же новость — не существует способа укоротить процедуру написания хороших справочных файлов. Этому нельзя помочь. Воспользуйтесь моим советом и приобретите хорошую программу для автоматизации создания справки. Такая программа может взять на себя часть неприятной работы по созданию справочных файлов. Если вы пишете справочные файлы самым трудным способом — в текстовом редакторе — вы, возможно, заскучаете и начнете пренебрегать подробностями. Хорошая программа создания справки поможет вам избежать и этой ситуации. Помните, что справочный файл так же важен, как и сама программа. В конце концов, как вы выясните, насколько хорош Мазерати, если не сможете разобраться, как включить первую передачу?

Использование реестра также может стать плюсом вашей программы. Конечно, это не изменит внешнего вида программы, но предоставление пользователям дополнительных опций увеличивает ее ценность. Сохранение этих опций в реестре также упрощает вашу работу. Не существует никакого оправдания, если вы не предоставите пользователям те возможности, которых они ожидают от программы. Конечно, это потребует некоторого времени и внимания к деталям, но другого пути здесь нет.

Мы завершили неделю рассмотрением обработки сообщений. Это наиболее низкий уровень программирования, с которым вы до сих пор сталкивались. Вам, вероятно, не часто придется этим заниматься, но в какой-то момент полученные знания сыграют свою роль. Надеюсь, что раздел по обработке сообщений принесет вам пользу, если и не сразу, то спустя некоторое время.

Приведенные ниже листинги — это код главной формы программы ScratchPad. Листинг R2.1 содержит заголовочный файл SPMain.h, а R2.2 — файл SPMain.cpp. Просмотрите эти листинги, чтобы глубже понять те идеи, которым была посвящена эта неделя.

Листинг R2.1. SPMain.h.

```
1: //---------------------------------------------------------------------
2: #ifndef SPMainH
3: #define SPMainH
4: //---------------------------------------------------------------------
5: #include <Classes.hpp>
6: #include <Controls.hpp>
```

```
 7: #include <StdCtrls.hpp>
 8: #include <Forms.hpp>
 9: #include <ExtCtrls.hpp>
10: #include <Buttons.hpp>
11: #include <ComCtrls.hpp>
12: #include <Menus.hpp>
13: #include <Dialogs.hpp>
14: #include <Clipbrd.hpp>
15: #include <Windows.hpp>
16: #include <Printers.hpp>
17: //---------------------------------------------------------------------
18: class TScratchPad : public TForm
19: {
20: __published:    // Компоненты, управляемые IDE
21:     TPanel *SpeedBar;
22:     TBevel *Bevel1;
23:     TSpeedButton *FileOpenBtn;
24:     TSpeedButton *FileSaveBtn;
25:     TStatusBar *StatusBar;
26:     TMainMenu *MainMenu;
27:     TMenuItem *FileMenu;
28:     TMenuItem *FileOpen;
29:     TMenuItem *FileSave;
30:     TMenuItem *FileSaveAs;
31:     TMenuItem *N1;
32:     TMenuItem *FilePrintSetup;
33:     TMenuItem *N2;
34:     TMenuItem *FileExit;
35:     TMenuItem *FilePrint;
36:     TMenuItem *Edit1;
37:     TMenuItem *EditReplace;
38:     TMenuItem *EditFind;
39:     TMenuItem *N4;
40:     TMenuItem *EditPaste;
41:     TMenuItem *EditCopy;
42:     TMenuItem *EditCut;
43:     TMenuItem *N5;
44:     TMenuItem *EditUndo;
45:     TMenuItem *Help1;
46:     TMenuItem *HelpAbout;
47:     TMenuItem *HelpContents;
48:     TMenuItem *EditSelectAll;
49:     TMenuItem *N3;
50:     TMenuItem *EditWordWrap;
51:     TOpenDialog *OpenDialog;
52:     TSaveDialog *SaveDialog;
53:     TMenuItem *FileNew;
54:     TPopupMenu *MemoPopup;
55:     TMenuItem *PopupCut;
56:     TMenuItem *PopupCopy;
57:     TMenuItem *PopupPaste;
58:     TSpeedButton *EditCutBtn;
59:     TSpeedButton *EditCopyBtn;
60:     TSpeedButton *EditPasteBtn;
61:     TSpeedButton *AboutBtn;
62:     TPrintDialog *PrintDialog;
63:     TPrinterSetupDialog *PrinterSetupDialog;
64:     TMemo *Memo;
```

см. продолжение

Листинг R2.1. (продолжение)

```
 65:     TSpeedButton *HelpBtn;
 66:     void __fastcall FileOpenClick(TObject *Sender);
 67:     void __fastcall FileSaveClick(TObject *Sender);
 68:     void __fastcall FileSaveAsClick(TObject *Sender);
 69:     void __fastcall FileExitClick(TObject *Sender);
 70:     void __fastcall EditSelectAllClick(TObject *Sender);
 71:     void __fastcall EditCutClick(TObject *Sender);
 72:     void __fastcall EditCopyClick(TObject *Sender);
 73:     void __fastcall EditPasteClick(TObject *Sender);
 74:     void __fastcall EditWordWrapClick(TObject *Sender);
 75:     void __fastcall FileNewClick(TObject *Sender);
 76:     void __fastcall EditUndoClick(TObject *Sender);
 77:     void __fastcall HelpAboutClick(TObject *Sender);
 78:     void __fastcall FormCreate(TObject *Sender);
 79:
 80:     void __fastcall FilePrintClick(TObject *Sender);
 81:     void __fastcall FilePrintSetupClick(TObject *Sender);
 82:     void __fastcall HelpBtnClick(TObject *Sender);
 83:
 84:     void __fastcall HelpContentsClick(TObject *Sender);
 85:     void __fastcall HelpTopicsClick(TObject *Sender);
 86: private:          // Объявления пользователя
 87:     void __fastcall OnSetCursor(TWMSetCursor& Message);
 88:     void __fastcall OnHint(TObject* Sender);
 89:     void __fastcall OnIdle(TObject* Sender, bool& Done);
 90:     void PrintFooter(TRect& rect, int lineHeight);
 91:     void DoHelp(int contextID);
 92:     bool HelpMode;
 93: public:           // Объявления пользователя
 94:     virtual __fastcall TScratchPad(TComponent* Owner);
 95:
 96:   BEGIN_MESSAGE_MAP
 97:     MESSAGE_HANDLER(WM_SETCURSOR, TWMSetCursor, OnSetCursor)
 98:   END_MESSAGE_MAP(TForm)
 99: };
100: //---------------------------------------------------------------------
101: extern TScratchPad *ScratchPad;
102: //---------------------------------------------------------------------
103: #endif
```

Листинг R2.2. SPMain.cpp.

```
  1: //---------------------------------------------------------------------
  2: #include <vcl.h>
  3: #pragma hdrstop
  4:
  5: #include "SPMain.h"
  6: #include "SPAbout.h"
  7: #include "SPHelp.h"
  8: //---------------------------------------------------------------------
  9: #pragma resource "*.dfm"
 10: #define MYMESSAGE WM_USER + 1
 11:
 12: TScratchPad *ScratchPad;
 13: //---------------------------------------------------------------------
```

```
14: __fastcall TScratchPad::TScratchPad(TComponent* Owner)
15:   : TForm(Owner)
16: {
17:   //
18:   // Выключение режима справки
19:   //
20:   HelpMode = false;
21: }
22: //---------------------------------------------------------------------
23: void __fastcall TScratchPad::FileNewClick(TObject *Sender)
24: {
25:   // Проверка каждого пункта меню на включение режима HelpMode.
26:   // Если да, вызов DoHelp() для вывода соответствующего
27:   // справочного файла. Мы должны это делать таким образом, чтобы
28:   // режим справки работал как для быстрых кнопок, так
29:   // и для пунктов меню. Если режим справки включен, то
30:   // пользователь не собирается выбирать данный пункт,
31:   // и после вызова WinHelp следует возврат
32:   //
33:   if (HelpMode) {
34:     DoHelp(IDH_FILENEW);
35:     return;
36:   }
37:   //
38:   // Открытие файла. Перед этим проверка, нужно ли сохранять
39:   // текущий файл
40:   //
41:   if (Memo->Modified) {
42:     //
43:     // Отображение окна сообщения
44:     //
45:     int result = Application->MessageBox(
46:       "The current file has changed. Save changes?",
47:       "ScratchPad Message", MB_YESNOCANCEL | MB_ICONWARNING);
48:     //
49:     // Если нажата кнопка Yes, сохранение текущего файла
50:     //
51:     if (result == IDYES) FileSaveClick(Sender);
52:     //
53:     // Если нажата кнопка No, ничего не делать
54:     //
55:     if (result == IDCANCEL) return;
56:   }
57:   //
58:   // Удаление строк в компоненте Memo, если они существуют
59:   //
60:   if (Memo->Lines->Count > 0) Memo->Clear();
61:   //
62:   // Установка для свойства FileName диалога Save Dialog
63:   // пустой строки. Тем самым мы будем знать, что файл
64:   // еще не был сохранен
65:   //
66:   SaveDialog->FileName = "";
67: }
68: //---------------------------------------------------------------------
69: void __fastcall TScratchPad::FileOpenClick(TObject *Sender)
70: {
71:   if (HelpMode) {
```

см. продолжение

Листинг R2.2. (продолжение)

```
 72:     DoHelp(IDH_FILEOPEN);
 73:     return;
 74:   }
 75:   //
 76:   // Открытие файла. Перед этим проверка, нужно ли сохранять
 77:   // текущий файл. Та же логика, что и в FileNewClick() выше
 78:   //
 79:   if (Memo->Modified) {
 80:     int result = Application->MessageBox(
 81:       "The current file has changed. Save changes?",
 82:       "ScratchPad Message", MB_YESNOCANCEL | MB_ICONWARNING);
 83:     if (result == IDYES) FileSaveClick(0);
 84:     if (result == IDCANCEL) return;
 85:   }
 86:   //
 87:   // Вызов диалога File Open. Если нажата кнопка OK, то
 88:   // открытие файла методом LoadFromFile(). Перед этим
 89:   // очистка свойства FileName
 90:   //
 91:   OpenDialog->FileName = "";
 92:   if (OpenDialog->Execute())
 93:   {
 94:     if (Memo->Lines->Count > 0) Memo->Clear();
 95:     Memo->Lines->LoadFromFile(OpenDialog->FileName);
 96:     SaveDialog->FileName = OpenDialog->FileName;
 97:   }
 98: }
 99: //---------------------------------------------------------------------
100: void __fastcall TScratchPad::FileSaveClick(TObject *Sender)
101: {
102:   if (HelpMode) {
103:     DoHelp(IDH_FILESAVE);
104:     return;
105:   }
106:   //
107:   // Если имя файла уже есть, то нет необходимости
108:   // вызывать диалог File Save. Сохранение
109:   // файла с помощью SaveToFile()
110:   //
111:   if (SaveDialog->FileName != "")
112:   {
113:     Memo->Lines->SaveToFile(SaveDialog->FileName);
114:     //
115:     // Установка Modified в false, т.к. мы только что сохранили
116:     //
117:     Memo->Modified = false;
118:   }
119:   //
120:   // Если имени файла нет, выполнение SaveAs()
121:   //
122:   else FileSaveAsClick(Sender);
123: }
124: //---------------------------------------------------------------------
125: void __fastcall TScratchPad::FileSaveAsClick(TObject *Sender)
126: {
127:   if (HelpMode) {
128:     DoHelp(IDH_FILESAVEAS);
```

```
129:     return;
130:   }
131:   //
132:   // Вывод диалога File Save для сохранения файла
133:   // и установка Modified в false
134:   //
135:   SaveDialog->Title = "Save As";
136:   if (SaveDialog->Execute())
137:   {
138:     Memo->Lines->SaveToFile(SaveDialog->FileName);
139:     Memo->Modified = false;
140:   }
141: }
142: //---------------------------------------------------------------------
143: void __fastcall TScratchPad::FileExitClick(TObject *Sender)
144: {
145:   if (HelpMode) {
146:     DoHelp(IDH_FILEEXIT);
147:     return;
148:   }
149:   //
150:   // Работа завершена. Закрытие формы
151:   //
152:   Close();
153: }
154: //---------------------------------------------------------------------
155: void __fastcall TScratchPad::EditUndoClick(TObject *Sender)
156: {
157:   if (HelpMode) {
158:     DoHelp(IDH_EDITUNDO);
159:     return;
160:   }
161:   //
162:   // В TMemo нет метода Undo, поэтому мы должны послать
163:   // компоненту Memo сообщение Windows WM_UNDO
164:   //
165:   SendMessage(Memo->Handle, WM_UNDO, 0, 0);
166: }
167: //---------------------------------------------------------------------
168: void __fastcall
169: TScratchPad::EditSelectAllClick(TObject *Sender)
170: {
171:   if (HelpMode) {
172:     DoHelp(IDH_EDITSELECTALL);
173:     return;
174:   }
175:   //
176:   // Просто вызов TMemo::SelectAll()
177:   //
178:   Memo->SelectAll();
179: }
180: //---------------------------------------------------------------------
181: void __fastcall TScratchPad::EditCutClick(TObject *Sender)
182: {
183:   if (HelpMode) {
184:     DoHelp(IDH_EDITCUT);
185:     return;
186:   }
```

см. продолжение

Листинг R2.2. (продолжение)

```
187:    //
188:    // Вызов TMemo::CutToClipboard()
189:    //
190:    Memo->CutToClipboard();
191: }
192: //---------------------------------------------------------------------
193: void __fastcall TScratchPad::EditCopyClick(TObject *Sender)
194: {
195:    if (HelpMode) {
196:      DoHelp(IDH_EDITCOPY);
197:      return;
198:    }
199:    //
200:    // Вызов TMemo::CopyToClipboard()
201:    //
202:    Memo->CopyToClipboard();
203: }
204: //---------------------------------------------------------------------
205: void __fastcall TScratchPad::EditPasteClick(TObject *Sender)
206: {
207:    if (HelpMode) {
208:      DoHelp(IDH_EDITPASTE);
209:      return;
210:    }
211:    //
212:    // Вызов TMemo::PasteFromClipboard()
213:    //
214:    Memo->PasteFromClipboard();
215: }
216: //---------------------------------------------------------------------
217: void __fastcall TScratchPad::EditWordWrapClick(TObject *Sender)
218: {
219:    if (HelpMode) {
220:      DoHelp(IDH_EDITWORDWRAP);
221:      return;
222:    }
223:    //
224:    // Переключение свойства TMemo::WordWrap. Установка для свойства
225:    // Checked пункта меню того же значения, что и у WordWrap
226:    //
227:    Memo->WordWrap = !Memo->WordWrap;
228:    EditWordWrap->Checked = Memo->WordWrap;
229:    //
230:    // Если режим WordWrap включен, нужна только линейка
231:    // вертикальной прокрутки. Иначе нужны обе линейки
232:    //
233:    if (Memo->WordWrap) Memo->ScrollBars = ssVertical;
234:    else Memo->ScrollBars = ssBoth;
235: }
236: //---------------------------------------------------------------------
237:
238: void __fastcall TScratchPad::HelpAboutClick(TObject *Sender)
239: {
240:    if (HelpMode) {
241:      DoHelp(IDH_HELPABOUT);
242:      return;
```

```
243:    }
244:    //
245:    // Показать окно About
246:    //
247:    AboutBox->ShowModal();
248: }
249: //---------------------------------------------------------------------
250: void __fastcall TScratchPad::FormCreate(TObject *Sender)
251: {
252:    Application->OnIdle = &OnIdle;
253:    Application->OnHint = &OnHint;
254: }
255: //---------------------------------------------------------------------
256: void __fastcall TScratchPad::OnHint(TObject* Sender)
257: {
258:    StatusBar->SimpleText = Application->Hint;
259: }
260:
261: void __fastcall
262: TScratchPad::OnIdle(TObject* Sender, bool& Done)
263: {
264:    //
265:    // Если режим справки включен, разрешение доступа ко всем
266:    // кнопкам и пунктам меню, чтобы пользователь мог выбрать
267:    // любой элемент для получения справки
268:    //
269:    if (HelpMode) {
270:      EditCut->Enabled = true;
271:      EditCutBtn->Enabled = true;
272:      PopupCut->Enabled = true;
273:      EditCopy->Enabled = true;
274:      EditCopyBtn->Enabled = true;
275:      PopupCopy->Enabled = true;
276:      EditPaste->Enabled = true;
277:      EditPasteBtn->Enabled = true;
278:      PopupPaste->Enabled = true;
279:      EditSelectAll->Enabled = true;
280:      FileSave->Enabled = true;
281:      FileSaveBtn->Enabled = true;
282:      FileSaveAs->Enabled = true;
283:      EditUndo->Enabled = true;
284:      return;
285:    }
286:    //
287:    // Код управления доступом для Cut и Copy. Если нет
288:    // выделенного текста, запрещение доступа в главном меню,
289:    // оперативной панели и контекстном меню
290:    //
291:    bool textSelected = Memo->SelLength > 0;
292:    EditCut->Enabled = textSelected;
293:    EditCutBtn->Enabled = textSelected;
294:    PopupCut->Enabled = textSelected;
295:    EditCopy->Enabled = textSelected;
296:    EditCopyBtn->Enabled = textSelected;
297:    PopupCopy->Enabled = textSelected;
298:    //
299:    // Код управления доступом для Paste. Если в буфере обмена
```

см. продолжение

Листинг R2.2. (продолжение)

```
300:    // нет текста, запретить Paste
301:    //
302:    bool cbHasText = Clipboard()->HasFormat(CF_TEXT);
303:    EditPaste->Enabled = cbHasText;
304:    EditPasteBtn->Enabled = cbHasText;
305:    PopupPaste->Enabled = cbHasText;
306:    //delete clipBrd;
307:    //
308:    // Код управления доступом для Select All
309:    //
310:    bool hasText = Memo->Lines->Count > 0;
311:    EditSelectAll->Enabled = hasText;
312:    //
313:    // Код управления доступом для Save
314:    //
315:    bool modified = Memo->Modified;
316:    FileSave->Enabled = modified && hasText;
317:    FileSaveBtn->Enabled = modified && hasText;
318:    //
319:    // Код управления доступом для Undo
320:    //
321:    EditUndo->Enabled = modified;
322: }
323: void __fastcall TScratchPad::FilePrintClick(TObject *Sender)
324: {
325:    if (HelpMode) {
326:      DoHelp(IDH_FILEPRINT);
327:      return;
328:    }
329:    //
330:    // Вывод диалога Print
331:    //
332:    if (PrintDialog->Execute()) {
333:      //
334:      // Установка заголовка для объекта принтера
335:      //
336:      Printer()->Title =
337:        "ScratchPad - " + OpenDialog->FileName;
338:      //
339:      // Установка такого же шрифта принтера, как и в Memo
340:      //
341:      Printer()->Canvas->Font = Memo->Font;
342:      //
343:      // Определение высоты строки. Берется размер шрифта Size
344:      // и вызывается функция MulDiv() для вычисления
345:      // высоты строки с учетом текущего разрешения
346:      // принтера. Функция abs() используется для получения
347:      // абсолютного значения, поскольку результат может быть
348:      // отрицательным числом. После этого добавляется 40%
349:      // на интерлиньяж
350:      int lineHeight = abs(
351:        MulDiv(Printer()->Canvas->Font->Size,
352:        GetDeviceCaps(Printer()->Handle, LOGPIXELSY), 72));
353:      lineHeight *= 1.4;
354:      //
355:      // Определение количества строк на странице. Уменьшение
356:     // результата на три строки, чтобы оставить место под нижнее поле
```

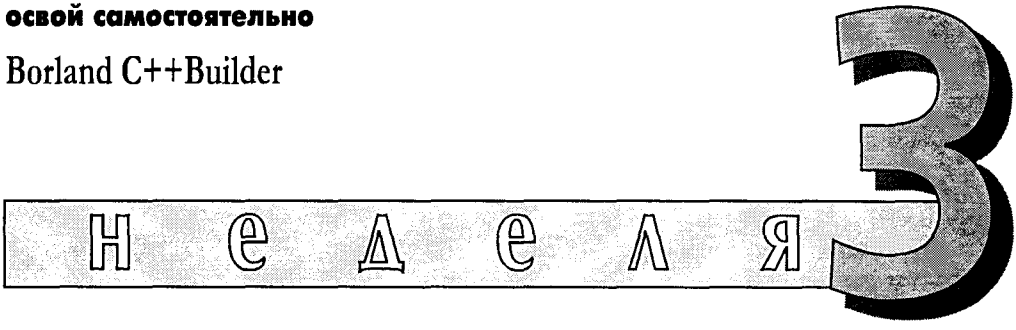
```
357:      //
358:      int linesPerPage =
359:        (Printer()->PageHeight/lineHeight) - 4;
360:      //
361:      // Начало печати со строки 4, а не со строки 0, чтобы
362:      // оставить место под заголовок и верхнее поле
363:      //
364:      int lineCount = 4;
365:      //
366:      // Сообщение Windows о начале печати и печать заголовка
367:      //
368:      Printer()->BeginDoc();
369:      TRect rect;
370:      rect.Top = lineHeight;
371:      rect.Left = 20;
372:      rect.Right =  Printer()->PageWidth;
373:      rect.Bottom = lineHeight * 2;
374:      DrawText(Printer()->Handle,
375:        OpenDialog->FileName.c_str(), -1, (RECT*)&rect, DT_CENTER);
376:      //
377:      // Цикл по всем строкам и печать каждой строки
378:      //
379:      for (int i = 0; i < Memo->Lines->Count; i++) {
380:        //
381:        // Когда мы дойдем до низа страницы, сброс счетчика
382:        // строк, выдача страницы и начало новой
383:        //
384:        if (lineCount++ == linesPerPage) {
385:          PrintFooter(rect, lineHeight);
386:          lineCount = 4;
387:          Printer()->NewPage();
388:        }
389:        //
390:        // Получение следующей строки и ее печать через TextOut()
391:        //
392:        String s = Memo->Lines->Strings[i];
393:        Printer()->Canvas->TextOut
394:          (0, lineCount * lineHeight, s);
395:      }
396:      //
397:      // Все сделано
398:      //
399:      PrintFooter(rect, lineHeight);
400:      Printer()->EndDoc();
401:    }
402: }
403:
404: void TScratchPad::PrintFooter(TRect& rect, int lineHeight)
405: {
406:    //
407:    // Формирование строки для вывода номера страницы. Используется
408:    // указатель char*, а не объект String, поскольку для
409:    // DrawText() в любом случае нужен char*
410:    //
411:    char buff[10];
412:    sprintf(buff, "Page %d", Printer()->PageNumber);
413:    //
```

см. продолжение

Листинг R2.2. (продолжение)

```
414:    // Установка прямоугольника, в который будет выведен нижний
415:    // колонтитул. Определение низа страницы и отступ на две
416:    // строки
417:    //
418:    rect.Top = Printer()->PageHeight - (lineHeight * 2);
419:    rect.Bottom = rect.Top + lineHeight;
420:    //
421:    // Вывод текста через DrawText, чтобы можно было центрировать
422:    // текст без особого труда
423:    //
424:    DrawText(Printer()->Handle, buff, -1, (RECT*)&rect, DT_CENTER);
425:    //
426:    // Проведение линии через страницу немного выше текста Page x
427:    //
428:    Printer()->Canvas->MoveTo(0, rect.Top - 2);
429:    Printer()->Canvas->LineTo(rect.Right, rect.Top - 2);
430: }
431: //---------------------------------------------------------------
432: void __fastcall
433: TScratchPad::FilePrintSetupClick(TObject *Sender)
434: {
435:   if (HelpMode) {
436:     DoHelp(IDH_FILEPRINTSETUP);
437:     return;
438:   }
439:   //
440:   // Просто вывод диалога Printer Setup. Windows возьмет
441:   // его отсюда
442:   //
443:   PrinterSetupDialog->Execute();
444: }
445: //---------------------------------------------------------------
446: void __fastcall TScratchPad::HelpBtnClick(TObject *Sender)
447: {
448:   //
449:   // Переключение режима справки
450:   //
451:   HelpMode = !HelpMode;
452: }
453: //---------------------------------------------------------------
454: void TScratchPad::DoHelp(int contextID)
455: {
456:   //
457:   // Вывод WinHelp с переданным ID контекста
458:   //
459:   Application->HelpContext(contextID);
460:   //
461:   // Выключение режима справки, как только пользователь выбрал
462:   // пункт
464:   HelpBtn->Down = false;
465:   HelpMode = false;
466: }
467: void __fastcall TScratchPad::HelpContentsClick(TObject *Sender)
468: {
469:   //
470:   // Вывод страницы содержания справочного файла
```

```
471:   //
472:   Application->HelpCommand(HELP_CONTENTS, 0);
473: }
474: //---------------------------------------------------------------------
475: void __fastcall TScratchPad::HelpTopicsClick(TObject *Sender)
476: {
477:   //
478:   // Вывод индекса справочного файла
479:   //
480:   Application->HelpCommand(HELP_FINDER, 0);
481: }
482: //---------------------------------------------------------------------
483:
484: //---------------------------------------------------------------------
485: // ЗАМЕЧАНИЕ: ниже используется вызов Windows::SetCursor()
486: // для доступа к функции Win API SetCursor(). Это вызов можно
487: // заменить, чтобы не использовать пространство имен 'Windows'.
488: // Если код не компилируется, то замените Windows::SetCursor() на
489: // ::SetCursor()
490: //
491: //---------------------------------------------------------------------
492: void __fastcall
493: TScratchPad::OnSetCursor(TWMSetCursor& Message)
494: {
495:   //
496:   // Мы будем сами управлять изменением курсора, чтобы
497:   // не зависеть от VCL. Это нужно для того, чтобы
498:   // получить курсор справки поверх меню
499:   //
500:   if (HelpMode) {
501:     //
502:     // Если курсор находится в рабочей области окна, то
503:     // вывод курсора 'NO' (невидимый курсор)
504:     //
505:     if (Message.CursorWnd == Memo->Handle)
506:       ::SetCursor(LoadCursor(NULL, IDC_NO));
507:     else
508:     //
509:     // Иначе вывод курсора справки
510:     //
511:       ::SetCursor(LoadCursor(NULL, IDC_HELP));
512:     Message.Result = true;
513:   }
514:   //
515:   // Если режим справки выключен, обработка сообщения
516:   // предопределенным образом
517:   //
518:   else TForm::Dispatch(&Message);
519: }
```



Краткий обзор

В начале этой недели мы рассмотрим элементы управления ActiveX, которые приобретают сейчас все большую популярность, а также технологию OLE. Затем мы приступим к изучению баз данных. Вы узнаете об архитектуре доступа к базам данных и о том, как реализованы операции с базами данных в C++Builder и VCL. Вы начнете с самых основ и будете постепенно осваивать более сложную технику программирования баз данных.



После баз данных мы обратимся к динамически компонуемым библиотекам (DLL).



DLL рассматриваются здесь для общего ознакомления. Их использование зависит от вашего желания, но важно, чтобы вы могли принять обоснованное решение. Для этого нужно понять, что представляют собой DLL и как они могут быть использованы в ваших приложениях.



День 19 целиком посвящен рассмотрению DLL. По его завершении вы сможете решать, подходят ли DLL для ваших целей.



Занимаясь программированием, рано или поздно вы столкнетесь с ситуациями, где использование DLL будет оправдано.

На двадцатый день вы узнаете о том, как создаются компоненты. Создание компонентов — это настоящее программирование. От вас потребуется более глубокое понимание свойств, методов и событий, чем раньше.



В завершение вы узнаете, как использовать C++Builder совместно с другими системами программирования.



Одним из главных достоинств форм C++Builder является то, что вы можете вызывать их практически из любой среды программирования. Тем самым для вас открывается широкий спектр новых возможностей.



Итак, впереди еще одна неделя. Вы готовы? Тогда начнем!

OLE, ActiveX и OCX



Кен Хендерсон

Не так давно аббревиатура OLE имела очень простое значение: связывание и внедрение объектов (Object Linking and Embedding). В настоящее время, однако, OLE представляет целую сеть взаимодействующих коммуникационных систем. Многие из этих систем могут быть использованы независимо от других. Я расскажу вам о нескольких подсистемах, составляющих OLE, и о возможностях их использования в приложениях.



Необходимо сделать замечание относительно терминологии. В настоящее время OLE является устаревшим термином; вместо него используется ActiveX. Microsoft предпочитает использовать для документов OLE название ActiveDocs. И вместо элементов управления OLE (OCX) я буду теперь ссылаться на элементы управления ActiveX. Используйте те ключевые слова, которые вам больше нравятся — технология, которую они обозначают, одна и та же.

В C++Builder встроена поддержка конструирования контейнеров и автоматизированных серверов OLE. Все контейнеры OLE могут вмещать объекты OLE таким образом, что они выглядят как принадлежащие своим базовым приложениям. Автоматизированный сервер OLE имеет интерфейс с внешним миром, через который можно управлять сервером и направлять к нему запросы. Автоматизированные объекты OLE предоставляются серверами OLE для использования другими процессами. Автоматизированный объект может быть встроен в контейнер OLE или связан с ним, и использоваться для управления своим базовым сервером. Далее я приведу обзор контейнеров и автоматизированных серверов OLE.

Лучше всего осваивать OLE на конкретных примерах. Чтобы не конструировать приложения OLE «с нуля», мы можем взять готовые демонстрационные программы, которые поставляются вместе с C++Builder. На их примере вы убедитесь, как легко работать с OLE в C++Builder.

Autosrv и Autocon

Первые два приложения, которые мы рассмотрим, иллюстрируют средства автоматизации OLE. Autosrv является автоматизированным сервером, который поддерживает запросы и управление извне. В Autocon создается экземпляр автоматизированного объекта OLE, определенного в Autosrv для установки и получения значения компонента TEdit, принадлежащего Autosrv.

Чтобы увидеть, как все это работает, сделайте следующее:

1. Запустите C++Builder и загрузите проект Autosrv. Он должен быть расположен в подкаталоге \Examples\Apps\Autosrv главного каталога C++Builder.
2. Нажмите Ctrl+F9 для компиляции проекта без последующего запуска.
3. После завершения компиляции загрузите проект Autocon. Он должен находиться в подкаталоге \Examples\Apps\Autocon главного каталога C++Builder.
4. Оставив приложение Autocon загруженным в C++Builder, запустите приложение Autosrv извне. Чтобы сделать это в Windows 95 и Windows NT 4.0, выберите в стартовом меню пункт Run, наберите C:\Program-Files\Borland\CBuilder\Examples\Apps\Autosrv\Autosrv.exe и нажмите Enter. (В вашей системе полный путь может быть другим.) После запуска Autosrv переместите его окно так, чтобы оно не было закрыто при обратном переключении к C++Builder.
5. Вернитесь в C++Builder и нажмите F9 для запуска Autocon. Вид экрана должен быть примерно таким, как на рис. 15.1. После того, как оба приложения запущены, пощелкайте на кнопках в Autocon и посмотрите, что при этом происходит в Autosrv.

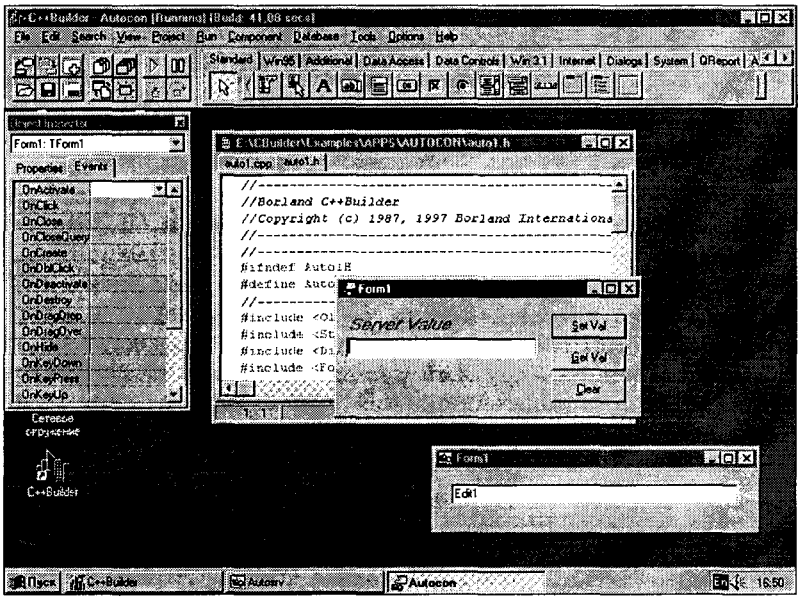


Рис. 15.1
Приложения OLE Autosrv и Autocon

Как вы только что убедились, Autocon способен управлять Autosrv посредством OLE. Давайте обратимся к коду Autocon, чтобы выяснить, как это происходит. Листинг 15.1 содержит исходный код формы Autocon.

Листинг 15.1. Единственная форма приложения Autocon

```
//---------------------------------------------------------------------
//Borland C++ Builder
//Copyright (c) 1987 Borland International Inc. All Rights Reserved.
//---------------------------------------------------------------------
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "auto1.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent *Owner)
  : TForm(Owner)
{
  try
  {
      AutoServer = CreateOleObject("BCBAutoSrv.EditServer");
  }
  catch (...)
  {
    ShowMessage("Please build and run the AutoSrv example before this one.");
    Application->Terminate();
  }
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  AutoServer.OlePropertySet("EditStr", Edit1->Text);
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  Edit1->Text = AutoServer.OlePropertyGet("EditStr");
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  AutoServer.OleProcedure("Clear");
}
//---------------------------------------------------------------------
```

Следующая строка кода является ключевой:

```
AutoServer = CreateOleObject("BCBAutoSrv.EditServer");
```

AutoServer — это переменная типа Variant, определенная в файле Auto1.h. Autocon устанавливает соединение с Autosrv путем создания экземпляра автоматизированного объекта OLE BCBAutoSrv.EditServer. Откуда берется BCBAutoSrv.EditServer? Загрузите проект Autosrv и исследуйте модуль Auto2.cpp. Исходный текст приведен в листинге 15.2.

Листинг 15.2. Исходный код автоматизированного объекта OLE в Autosrv (Auto2.cpp)

```
1: //---------------------------------------------------------------------
2: //Borland C++ Builder
3: //Copyright (c) 1987 Borland International Inc.
4: // All Rights Reserved.
5: //---------------------------------------------------------------------
6: //---------------------------------------------------------------------
7: #include <vcl.h>
8: #pragma hdrstop
9: #undef RegisterClass
10:
11: #include "Auto2.h"
12: #include "Auto1.h"
13: //---------------------------------------------------------------------
14: int Initialization();
15: static int Initializer = Initialization();
16: //---------------------------------------------------------------------
17: __fastcall ButtonServer::ButtonServer()
18:   : TAutoObject()
19: {
20: }
21: //---------------------------------------------------------------------
22: String __fastcall ButtonServer::GetEditStr()
23: {
24:   return Form1->Edit1->Text;
25: }
26: //---------------------------------------------------------------------
27: void __fastcall ButtonServer::SetEditStr(AnsiString NewVal)
28: {
29:   Form1->Edit1->Text = NewVal;
30: }
31: //---------------------------------------------------------------------
32: int __fastcall ButtonServer::GetEditNum()
33: {
34:   int val;
35:
36:   sscanf(Form1->Edit1->Text.c_str(), "%d", &val);
37:   return val;
38: }
39: //---------------------------------------------------------------------
40: void __fastcall ButtonServer::SetEditNum(int NewVal)
41: {
42:   Form1->Edit1->Text = NewVal;
43: }
44: //---------------------------------------------------------------------
45: void __fastcall ButtonServer::Clear()
46: {
47:   Form1->Edit1->Text = "";
48: }
49: //---------------------------------------------------------------------
50: void __fastcall ButtonServer::SetThreeStr(AnsiString s1,
51:     AnsiString s2, AnsiString s3)
52: {
53:   Form1->Edit1->Text = s1 + ", " + s2 + ", " + s3;
54: }
55: //---------------------------------------------------------------------
56: void __fastcall ButtonServer::SetThreeNum(int n1,
57:                                 int n2, int n3)
```

```
58: {
59:   AnsiString s1(n1), s2(n2), s3(n3);
60:
61:   Form1->Edit1->Text = s1 + ", " + s2 + ", " + s3;
62: }
63: //---------------------------------------------------------------------
64: void __fastcall RegisterButtonServer()
65: {
66:   TAutoClassInfo AutoClassInfo;
67:
68:   AutoClassInfo.AutoClass = __classid(ButtonServer);
69:   AutoClassInfo.ProgID = "BCBAutoSrv.EditServer";
70:   AutoClassInfo.ClassID = "{61E124E1-C869-11CF-9EA7-00A02429B18A}";
71:   AutoClassInfo.Description =
72:                 "Borland C++Builder AutoSrv Example Server Class";
73:   AutoClassInfo.Instancing = acMultiInstance;
74:
75:   Automation->RegisterClass(AutoClassInfo);
76: }
77: //---------------------------------------------------------------------
78: int Initialization()
79: {
80:   RegisterButtonServer();
81:   return 0;
82: }
83: //---------------------------------------------------------------------
```

Если вы внимательно посмотрите на исходный текст Auto2.cpp, то найдете в нем определение EditServer. Обратите внимание на ссылку на класс TAutoObject в следующей строке:

```
__fastcall ButtonServer::ButtonServer()
  : TAutoObject()
```

Класс ButtonServer включает в себя несколько методов. Методы GetEditStr(), SetEditStr() и Clear() вызываются Autocon для осуществления связи с Autosrv. Например, реализация метода GetEditStr() в Autosrv выглядит следующим образом:

```
String __fastcall ButtonServer::GetEditStr()
{
  return Form1->Edit1->Text;
}
```

Для вызова GetEditStr() в Autocon используется следующий код:

```
Edit1->Text = AutoServer.OlePropertyGet("EditStr");
```

Поскольку функция GetEditStr() определена как метод чтения для свойства EditStr (см. Auto2.h), запрос EditStr приводит к выполнению GetEditStr() и возврату требуемого значения. Имя свойства передается в качестве параметра методу OlePropertyGet(), принадлежащему Autosrv. Когда EditStr запрашивается в базовом автоматизированном объекте OLE, в Autosrv вызывается функция GetEditStr и возвращается значение свойства Autosrv Edit1->Text. После этого процесс повторяется в обратном порядке. Значение, возвращенное GetEditStr, передается Autocon и присваивается свойству Text компонента Edit1, принадлежащего Autocon. В результате Autocon может ссылаться на значения в Autosrv так же, как и на свои собственные. Это

возможно благодаря использованию автоматизированного объекта OLE, определенного в Autosrv.

Функция Initialization() регистрирует EditServer в Windows, чтобы приложение, поддерживающее OLE, могло его использовать. Этот шаг необходим для работы с автоматизированными объектами OLE. Initialization() вызывает RegisterButtonServer(), который устанавливает связь между классом ButtonServer и BCBAutoSrv.EditServer. Исходный текст RegisterButtonServer() приведен в листинге 15.3.

Листинг 15.3. RegisterButtonServer()

```
 1: void __fastcall RegisterButtonServer()
 2: {
 3:   TAutoClassInfo AutoClassInfo;
 4:
 5:   AutoClassInfo.AutoClass = __classid(ButtonServer);
 6:   AutoClassInfo.ProgID = "BCBAutoSrv.EditServer";
 7:   AutoClassInfo.ClassID = "{61E124E1-C869-11CF-9EA7-00A02429B18A}";
 8:   AutoClassInfo.Description =
 9:                "Borland C++Builder AutoSrv Example Server Class";
10:   AutoClassInfo.Instancing = acMultiInstance;
11:
12:   Automation->RegisterClass(AutoClassInfo);
13: }
```

Обратите внимание на присваивания значений членам AutoClass и ProgID. Эти присваивания связывают BCBAutoSrv.EditServer и ButtonServer, позволяя OLE-приложениям ссылаться на автоматизированный объект ButtonServer, используя BCBAutoSrv.EditServer.

Создание собственного автоматизированного сервера

Вы можете создавать свои собственные автоматизированные серверы OLE, используя мастер автоматизированного объекта (Automation Object Wizard), включенный в C++Builder. Создание простого автоматизированного сервера выполняется следующим образом:

1. Выберите в главном меню C++Builder пункт File | New Application для создания нового приложения. Это приложение станет вашим автоматизированным сервером OLE. Для управления сервером и направления к нему запросов другие приложения будут использовать объект Automation.
2. Измените значение свойства Caption главной формы нового приложения на My OLE Automation Server. Это сделает назначение вашего приложения немного более понятным.
3. Выберите в главном меню пункт File | Save Project As и сохраните новое приложение. Сохраните главную форму как unit1.cpp, а сам проект как MyAuto.mak.
4. Поместите в форму один компонент TEdit.
5. Выберите в главном меню пункт File | New..., затем дважды щелкните на Automation Object в диалоговом окне New Items. После этого вы должны увидеть диалоговое окно, показанное на рис. 15.2.

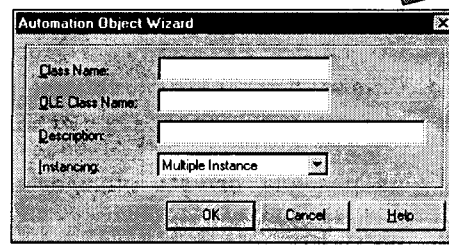


Рис. 15.2
Диалоговое окно мастера автоматизированного объекта OLE

6. Первое, что вам нужно указать — это имя класса нового объекта. Имя должно следовать соглашениям, принятым в Си++. Наберите TMyOLEEditServer и нажмите Tab.
7. Следующим пунктом стоит имя класса OLE, которое будет использоваться другими приложениями для создания экземпляров вашего автоматизированного объекта. Это имя всегда следует составлять из имени проекта и имени класса Си++ (без префикса T), разделенных точкой. В нашем случае оно будет выглядеть как MyAuto.MyOLEEditServer. Для продолжения нажмите Tab.
8. Затем вам нужно ввести описание нового объекта. Это значение будет сохранено в системном реестре. Введите в поле Description строку A Test OLE Automation Server.
9. Последним пунктом является Instancing (создание экземпляров). Для серверов, реализованных в виде DLL, вы будете использовать значение Multiple Instance. Для приложений обычно устанавливается Single Instance. Пока давайте не будем изменять этот пункт. Нажмите кнопку OK для генерации исходного кода автоматизированного объекта. На рис. 15.3 показано заполненное диалоговое окно.

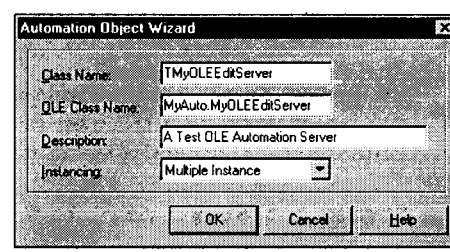


Рис. 15.3
Заполненное диалоговое окно мастера автоматизированного объекта

Теперь, когда код для нового автоматизированного объекта создан, можно добавлять к нему методы, которые будут использоваться другими приложениями посредством механизма OLE.

Целесообразно скопировать в ваш новый автоматизированный сервер часть кода из демонстрационной программы Autosrv. Вам потребуется код, поддерживающий работу со свойством EditStr. Для копирования EditStr в ваш новый автоматизированный объект выполните следующие действия:

1. Установите курсор в модуле unit2.cpp на имени unit2.h в строке #include "unit2.h" и нажмите Ctrl+Enter для загрузки этого файла в редактор кода.
2. Загрузите в редактор кода исходный файл Auto2.cpp из подкаталога Examples\Apps\Autosrv главного каталога C++Builder.

3. Установите курсор на Auto2.h в строке #include "Auto2.h" и нажмите Ctrl+Enter для загрузки этого файла в редактор кода.
4. Скопируйте следующие строки:

```
AnsiString __fastcall GetEditStr();
void __fastcall SetEditStr(AnsiString NewVal);
```

из раздела private определения класса ButtonServer (файл Auto2.h) в раздел private определения вашего класса TMyOLEEditServer (файл unit2.h).

5. Скопируйте следующую строку:

```
__property AnsiString EditStr = {read=GetEditStr, write=SetEditStr};
```

из раздела __automated определения класса ButtonServer (файл Auto2.h) в раздел __automated определения класса TMyOLEEditServer (файл unit2.h).

6. Добавьте следующую строку в модуль unit2.cpp сразу за строкой #include "unit2.h":

```
#include "unit1.h"
```

Это сделает возможным доступ к форме unit1 (а именно, к ее компоненту Edit1) из модуля unit2, в котором определен ваш автоматизированный объект.

7. Скопируйте следующие строки:

```
//---------------------------------------------------------------------
String __fastcall ButtonServer::GetEditStr()
{
  return Form1->Edit1->Text;
}
//---------------------------------------------------------------------
void __fastcall ButtonServer::SetEditStr(AnsiString NewVal)
{
  Form1->Edit1->Text = NewVal;
}
```

из реализации класса ButtonServer (модуль Auto2.cpp) в реализацию класса TMyOLEEditServer (модуль unit2.cpp). При этом замените имя ButtonServer на TMyOLEEditServer, как здесь:

```
//---------------------------------------------------------------------
String __fastcall TMyOLEEditServer::GetEditStr()
{
  return Form1->Edit1->Text;
}
//---------------------------------------------------------------------
void fastcall TMyOLEEditServer::SetEditStr(AnsiString NewVal)
{
  Form1->Edit1->Text = NewVal;
}
```

8. Сохраните проект и нажмите Ctrl+F9 для его компиляции.

Для проверки работы вашего нового сервера OLE потребуется программа-клиент, которая создаст экземпляр автоматизированного объекта и будет вызывать его методы. Чтобы не создавать новое приложение целиком, можно снова использовать демонстрационную программу C++Builder. Загрузите про-

ект Autocon, который вы компилировали и запускали ранее. После загрузки сделайте копию строки

```
AutoServer = CreateOleObject("BCBTestOLE.EditServer");
```

и закомментируйте ее. Данный фрагмент кода должен выглядеть следующим образом:

```
AutoServer = CreateOleObject("BCBTestOLE.EditServer");
//AutoServer = CreateOleObject("BCBTestOLE.EditServer");
```

Затем замените имя BCBTestOLE.EditServer (в незакомментированной строке) на MyAuto.MyOLEEditServer:

```
AutoServer = CreateOleObject("MyAuto.MyOLEEditServer");
//AutoServer = CreateOleObject("BCBTestOLE.EditServer");
```

В результате приложение Autocon создаст экземпляр вашего автоматизированного объекта вместо того объекта, который был определен в Autosrv.

Окончив вносить изменения в Autocon, запустите приложение MyAuto извне C++Builder, а затем запустите Autocon. Разместите окна приложений так, чтобы их можно было видеть одновременно.

Теперь вы можете проверить, действительно ли Autocon может вызывать методы, определенные в TMyOLEEditServer. Используйте для этого кнопки в Autocon. Щелкните на кнопке Get Val для получения текущего содержимого компонента Edit1 в MyAuto. Теперь наберите текст в окне редактирования TEdit приложения Autocon и щелкните на кнопке Set Val. Вы увидите, как введенный текст появится в окне Edit1 приложения MyAuto. В завершение щелкните на кнопке Clear приложения Autocon.

При нажатии на кнопку Clear вы получите сообщение об ошибке, поскольку мы не реализовали метод Clear(), когда копировали код из Autosrv в MyAuto. Так как реализация OLE в C++Builder основана на позднем связывании (late binding), вы часто будете узнавать о подобных промахах только при тестировании приложения.

Вы завершили создание своего автоматизированного сервера OLE. Как видите, этот процесс действительно прост, если пройти через него один или два раза.

Листинги 15.4–15.8 содержат полный исходный код приложения MyAuto.

Листинг 15.4. Исходный код unit1.h

```
 1: //---------------------------------------------------------------------
 2: #ifndef unit1H
 3: #define unit1H
 4: //---------------------------------------------------------------------
 5: #include <vcl\Classes.hpp>
 6: #include <vcl\Controls.hpp>
 7: #include <vcl\StdCtrls.hpp>
 8: #include <vcl\Forms.hpp>
 9: //---------------------------------------------------------------------
10: class TForm1 : public Tform
11: {
12:    __published:    // Компоненты, управляемые IDE
```

см. продолжение

Листинг 15.4. (продолжение)

```
13:     TEdit *Edit1;
14:   private:          // Объявления пользователя
15:   public:           // Объявления пользователя
16:     virtual __fastcall TForm1(TComponent* Owner);
17: };
18: //---------------------------------------------------------------------
19: extern TForm1 *Form1;
20: //---------------------------------------------------------------------
21: #endif
```

Листинг 15.5. Исходный код unit1.cpp

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: #pragma hdrstop
 4:
 5: #include "unit1.h"
 6: //---------------------------------------------------------------------
 7: #pragma resource "*.dfm"
 8: TForm1 *Form1;
 9: //---------------------------------------------------------------------
10: __fastcall TForm1::TForm1(TComponent* Owner)
11:     : TForm(Owner)
12: {
13: }
14: //---------------------------------------------------------------------
```

Листинг 15.6. Исходный код unit1.dfm

```
 1: object Form1: TForm1
 2:   Left = 200
 3:   Top = 108
 4:   Width = 435
 5:   Height = 300
 6:   Caption = 'My OLE Automation Server'
 7:   Font.Charset = DEFAULT_CHARSET
 8:   Font.Color = clWindowText
 9:   Font.Height = -11
10:   Font.Name = 'MS Sans Serif'
11:   Font.Style = []
12:   PixelsPerInch = 96
13:   TextHeight = 13
14:   object Edit1: Tedit
15:     Left = 128
16:     Top = 72
17:     Width = 121
18:     Height = 21
19:     TabOrder = 0
20:     Text = 'Edit1'
21:   end
22: end
```

Листинг 15.7. Исходный код unit2.h

```
 1: //---------------------------------------------------------------------
 2: #ifndef unit2H
 3: #define unit2H
 4: //---------------------------------------------------------------------
 5: #include <vcl\oleauto.hpp>
 6: #include <vcl\Classes.hpp>
 7: //---------------------------------------------------------------------
 8: class TMyOLEEditServer : public TAutoObject
 9: {
10:   private:
11:     AnsiString __fastcall GetEditStr();
12:     void __fastcall SetEditStr(AnsiString NewVal);
13:   public:
14:     virtual __fastcall TMyOLEEditServer();
15:   __automated:
16:     __property AnsiString EditStr =
17:       {read=GetEditStr, write=SetEditStr};
18: };
19: //---------------------------------------------------------------------
20: #endif
```

Листинг 15.8. Исходный текст unit2.cpp

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: #pragma hdrstop
 4:
 5: #include "unit2.h"
 6: #include "unit1.h"
 7: //---------------------------------------------------------------------
 8: __fastcall TMyOLEEditServer::TMyOLEEditServer()
 9:     : TAutoObject()
10: {
11: }
12: //---------------------------------------------------------------------
13: String __fastcall TMyOLEEditServer::GetEditStr()
14: {
15:   return Form1->Edit1->Text;
16: }
17: //---------------------------------------------------------------------
18: void __fastcall TMyOLEEditServer::SetEditStr(AnsiString NewVal)
19: {
20:   Form1->Edit1->Text = NewVal;
21: }
22: //---------------------------------------------------------------------
23: void __fastcall RegisterTMyOLEEditServer()
24: {
25:   TAutoClassInfo AutoClassInfo;
26:
27:   AutoClassInfo.AutoClass = __classid(TMyOLEEditServer);
28:   AutoClassInfo.ProgID = "MyAuto.MyOLEEditServer";
29:   AutoClassInfo.ClassID = "{1B0BA200-7C66-11D0-B9C3-0080C82BD9D5}";
30:   AutoClassInfo.Description = "A Test OLE Automation Server";
```

см. продолжение

Листинг 15.8. (продолжение)

```
31:   AutoClassInfo.Instancing = acMultiInstance;
32:
33:   Automation->RegisterClass(AutoClassInfo);
34: }
35: //---------------------------------------------------------------------
36: #pragma startup RegisterTMyOLEEditServer
37: //---------------------------------------------------------------------
```

Olemdi и Olesdi

Теперь, когда вы познакомились с работой автоматизированных серверов OLE, давайте рассмотрим контейнеры OLE. Контейнер OLE позволяет приложению работать с объектом, принадлежащим другому приложению. Объект может быть вставлен либо как значок (и просматриваться/редактироваться в своем исходном приложении), либо с сохранением своего оригинального вида, что позволяет просматривать или редактировать его непосредственно. При работе с документами OLE основное внимание уделяется не исходному приложению, а самому документу.

В C++Builder для поддержки контейнеров OLE используется компонент TOLEContainer, который предоставляет специальные методы для создания объектов OLE. Вы можете использовать CreateObject или CreateObjectFromFile для создания внедренного объекта OLE, тогда как CreateLinkToFile создает связанный объект. Внедренный объект OLE реально содержится в своем базовом документе. Связанный объект, напротив, располагается на диске — базовый документ содержит только ссылку на него. Преимущество внедренных объектов заключается в том, что их базовые документы являются одновременно и контейнерами — в этом случае не требуются никакие дополнительные или служебные файлы. Вы можете пересылать документ с внедренными объектами OLE, не беспокоясь о включении дополнительных файлов. При работе со связанными объектами требуется включение исходных документов. В этом случае размер базового документа получается меньше, поскольку он содержит только ссылки на связанные объекты, а не сами эти объекты.

Для демонстрации возможностей контейнеров OLE в C++Builder включены два приложения: Olemdi (OLE multiple-document interface) и Olesdi (OLE single-document interface). Они показывают, как легко можно внедрять объекты из различных источников в приложения C++Builder.

Для наших целей более важно разобрать приложение Olemdi, поэтому откройте проект Olemdi.mak (он расположен в подкаталоге \Examples\Doc\Olectnrs главного каталога C++Builder).

Для запуска Olemdi нажмите клавишу F9. После того, как приложение заработает, щелкните на кнопке Open в его панели инструментов и откройте несколько документов OLE (например, документ Microsoft Word или таблицу Microsoft Excel). На рис. 15.4. показано приложение Olemdi с загруженным документом Word и HTML-файлом.

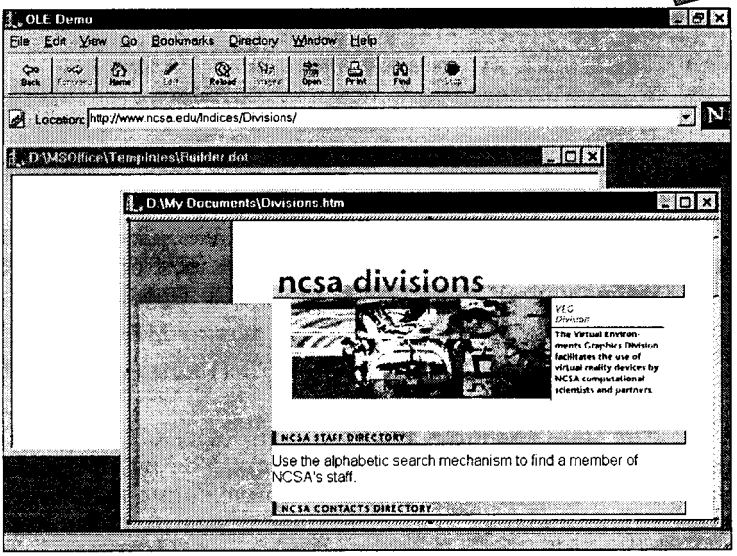


Рис. 15.4
Olemdi может содержать объекты OLE из других приложений



Возможность открытия HTML-файлов (Web-страниц) зависит от Web-браузера, который установлен в вашей системе по умолчанию. В моем случае это Netscape Navigator. Если вы используете Microsoft Internet Explorer 3.0, то просмотр HTML-файлов в Olemdi будет невозможен. Дело в том, что Internet Explorer 3.0 не поддерживает in-place-вызовы документов OLE, хотя и является продуктом Microsoft. Вместо этого объекты будут представлены значками. После двойного щелчка на значке, связанный HTML-файл будет загружен в Internet Explorer вне приложения-контейнера OLE.

После загрузки документов OLE вы заметите, что двойной щелчок на них вызывает существенное изменение вида Olemdi. Во время просмотра выбранного документа к Olemdi добавляются меню и панель инструментов исходного приложения. Например, когда вы просматриваете документ Word, в верхней части окна Olemdi появляются панель инструментов и система меню Microsoft Word. Создается впечатление, что загружен сам Word, но в строке заголовка окна по-прежнему написано Olemdi. Если вы затем переключитесь на таблицу Excel, меню и панель инструментов станут такими же, как в Microsoft Excel. Поскольку у меня в качестве Web-браузера по умолчанию стоит Netscape Navigator, при просмотре HTML-файлов я увижу меню и панель инструментов Netscape Navigator. Идея внедрения и связывания объектов заключается в том, что вы можете чувствовать себя «как дома» при просмотре «чужих» документов в приложении-контейнере OLE. Объекты OLE несут в себе всю информацию, необходимую для их просмотра и редактирования.

Olemdi изнутри

Несмотря на свою мощь, Olemdi в действительности очень простая программа. Большая часть ее функций реализуется компонентом TOLEContainer, помещенным в подчиненную форму MDI. Например, с компонентом OpenButton главной формы связан следующий код:

```
void __fastcall TMainForm::Open1Click(Tobject *Sender)
{
  if (OpenDialog1->Execute()) {
    TMDIChildForm* ch = new TMDIChildForm(this);
    ch->CreateFromFile(OpenDialog1->FileName, False);
    CutButton->Enabled = True;
    CopyButton->Enabled = True;
  }
}
```

Здесь выполняются три основных действия:

- Отображается диалоговое окно File Open.
- Создается новая подчиненная форма MDI.
- Вызывается метод CreateFromFile() компонента TOLEContainer для вставки объекта OLE.

Этот код является центральной частью приложения Olemdi. Для связывания и внедрения объектов применяется очень прямолинейный подход. Вы можете создать свое собственное приложение-контейнер OLE, просто поместив в форму компонент TOLEContainer и вызывая при необходимости его методы CreateObject(), CreateFromFile() и CreateLinkToFile().

Элементы управления OCX

Элемент управления OCX (или OLE, или ActiveX) похож на компонент C++Builder в том, что он использует аналогичный интерфейс для обращения к блоку кода. Для поддержки OCX в C++Builder используется специальный класс VCL, благодаря которому импортированные элементы управления OCX выглядят и работают подобно обычным компонентам. Для импорта элемента управления OCX выберите в главном меню C++Builder пункт Component | Install, а затем щелкните на кнопке ActiveX. Например, для импорта OCX, который выполняет большинство функций Web-браузера Microsoft Internet Explorer 3.0, сделайте следующее:

1. Выберите в главном меню C++Builder пункт Component | Install.
2. Щелкните на кнопке ActiveX.
3. После появления диалогового окна Import OLE Control (Импортировать элемент управления OLE), выберите в списке Registered Controls (Зарегистрированные элементы) пункт Microsoft Internet Controls.
4. В поле Palette page должно быть установлено значение Internet, чтобы новый компонент был помещен во вкладку Internet палитры компонентов.
5. Нажмите кнопку OK, затем еще раз OK. Вы должны увидеть в палитре компонентов новый компонент с названием TWebBrowser. Теперь вы можете создавать свои собственные приложения с теми же основными функциями, что и у Internet Explorer 3.0.

Заключение

Сегодня мы рассмотрели средства поддержки OLE и связанных с ним технологий в C++Builder. C++Builder имеет встроенные средства поддержки

автоматизированных серверов и контейнеров OLE, а также элементов управления OCX (известных еще как элементы управления OLE или ActiveX) в ваших приложениях.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Поддерживает ли C++Builder тип данных Variant для работы с OLE?

Да, в качестве примера можно привести приложение Autocon, рассмотренное сегодня.

Какой компонент нужно использовать для связывания и внедрения объектов OLE в приложениях?

Компонент TOLEContainer.

Как вызвать мастер автоматизированного объекта OLE?

Выберите в главном меню C++Builder пункт File | New..., затем дважды щелкните на Automation Object.

Вопросы

1. Что происходит при вызове OLE-приложением метода, не существующего в автоматизированном объекте OLE?
2. Какой пункт меню нужно использовать для установки компонента OCX в палитру компонентов C++Builder?

Упражнения

1. Добавьте автоматизированный объект OLE к существующему приложению C++Builder и создайте методы для запроса/установки различных элементов приложения посредством механизма OLE.
2. Импортируйте элемент управления OLE и установите его в палитру компонентов C++Builder.
3. Используя приложение Olemdi или Olesdi, откройте несколько различных документов OLE и посмотрите, как изменяется вид меню и панели инструментов базового приложения.

Архитектура доступа к базам данных в C++Builder



Кен Хендерсон

Сегодня мы будем изучать иерархию компонентов C++Builder, относящихся к базам данных. Я подробно опишу классы, составляющие архитектуру доступа к базам данных в C++Builder, выделяя ключевые свойства, методы и события каждого из них. К концу дня вы будете иметь общее представление о том, как взаимодействуют различные классы и как они используются для построения приложений.

Говоря более конкретно, вы узнаете:

- Как устроена архитектура доступа к базам данных в C++Builder
- Какие компоненты необходимы для построения приложений, работающих с базами данных
- Для чего предназначен компонент TSession
- Ключевые элементы компонента TDatabase
- Как использовать компонент TDataSource
- Свойства, события и основные методы компонентов TTable, TQuery и TStoredProc
- Как использовать компонент TBatchMove для копирования строк и создания таблиц

Некоторые термины, которые необходимо знать

На протяжении всей главы я буду ссылаться на различные составляющие архитектуры баз данных C++Builder. В связи с этим необходимо предварительно определить некоторые термины, чтобы вы понимали, о чем идет речь. Одни из них относятся к C++Builder, другие — к базам данных. С частью этих терминов вы можете быть знакомы, какие-то, вероятно, будут для вас абсолютно новыми. Ключевые термины сегодняшнего дня сведены в таблицу 16.1.

Таблица 16.1. Стандартные термины, относящиеся к базам данных в C++Builder

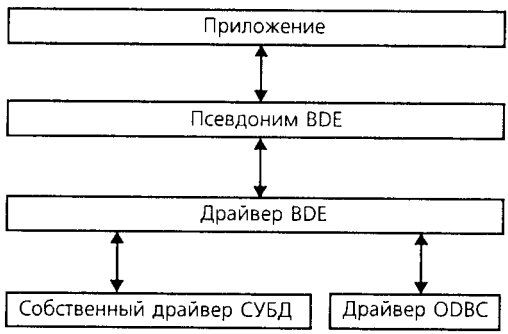
Термин	Описание
Таблица (table)	Совокупность строк (или объектов) в базе данных. Например, вы можете создать таблицу INVOICE для хранения счетов.
Строка (row)	Запись или объект таблицы. Например, таблица CUSTOMER может содержать строки данных о покупателях. Каждая строка в этой таблице будет соответствовать отдельному покупателю.
Столбец (column)	Поле или атрибут, содержащийся в строках таблицы. Например, ваша таблица INVOICE может содержать столбец CustomerNumber (код покупателя). Поле CustomerNumber будет присутствовать в каждой строке таблицы.
Borland Database Engine	Набор DLL и вспомогательных файлов, которые обеспечивают доступ к базам данных в C++Builder и других продуктах фирмы Borland. Borland Database Engine (BDE) сильно облегчает работу с базами данных, предоставляя высокоуровневый интерфейс прикладного программирования баз данных (database API), одинаковый для всех СУБД, которые он поддерживает. Этот дружественный по отношению к разработчику интерфейс обеспечивает в C++Builder управление базами данных таким образом, что вам редко придется работать напрямую с самим BDE.
IDAPI	Независимый интерфейс прикладного программирования баз данных, разработанный фирмой Borland (Independent Database Application Programming Interface). Различные приложения (в том числе приложения C++Builder) взаимодействуют с BDE через этот интерфейс. Поскольку почти все необходимые вызовы IDAPI выполняются за вас компонентами C++Builder, вам редко придется писать код, напрямую обращающийся к IDAPI. Вместо этого вы будете иметь дело с методами, свойствами и событиями компонентов C++Builder, предназначенных для работы с базами данных.
Драйвер BDE (BDE Driver)	DLL (или набор DLL), обеспечивающая взаимодействие BDE с конкретной СУБД. C++Builder в версии клиент/сервер включает в себя драйверы для работы с Sybase, Microsoft, Oracle, InterBase, Informix, DB2, Paradox, dBASE и любыми 32-разрядными источниками данных ODBC. Программы C++Builder не работают напрямую с драйверами BDE. Вместо этого они используют псевдонимы (aliases) BDE, которые основаны на драйверах BDE.
Псевдоним BDE (BDE Alias)	Совокупность конфигурационных параметров, которые сообщают BDE, как обращаться к конкретной базе данных. Псевдонимы основаны на драйверах баз данных BDE. Вы можете создавать псевдонимы с помощью конфигурационной программы BDE или Database Explorer C++Builder. Для каждой базы данных обычно используются отдельный псевдоним. Например, вы можете создать один псевдоним для обращения к базе данных Northwind в формате Microsoft Access, а другой – для обращения к базе данных Orders в том же формате. Оба псевдонима будут основаны на драйвере Access ODBC, поскольку они работают с базой данных Access. Однако эти псевдонимы не будут одинаковыми, т.к. используются для обращения к различным базам данных Access. В этом и заключается разница между драйверами и псевдонимами BDE – драйверы обращаются к СУБД, а псевдонимы – к конкретной базе данных, с которой работает СУБД.

Термин	Описание
Драйверы SQL Links	Высокопроизводительные драйверы для доступа к базам данных, используемые BDE для работы с СУБД типа клиент/сервер. C++Builder в версии клиент/сервер включает в себя SQL-драйверы для СУБД Sybase, Microsoft, Oracle, InterBase, Informix и DB2. Поскольку эти драйверы включены в C++Builder, вам не требуется использовать альтернативные методы типа ODBC для доступа к перечисленным СУБД, хотя это и не возбраняется.
Драйверы ODBC	Драйверы для доступа к базам данных, основанные на спецификации Open Database Connectivity фирмы Microsoft. C++Builder может использовать 32-разрядные драйверы ODBC для связи с внутренними компонентами баз данных. Настройка источников данных ODBC (которые похожи на псевдонимы BDE) и управление ими осуществляется через стандартную программу ODBC Administrator в панели управления Windows.
Компонент доступа к данным (data access control)	Невизуальный (невидимый во время выполнения программы) компонент, который обеспечивает вашему приложению доступ к базе данных. Компоненты доступа к данным расположены во вкладке Data Access палитры компонентов C++Builder. К ним относятся TDatabase, TTable и TDataSource.
TDataSet	Класс C++Builder, обеспечивающий доступ к таблицам баз данных, а также результатам запросов, имеющих структуру таблицы. Поскольку компоненты TTable, TQuery и TStoredProc косвенно произведены от класса TDataSet, я часто буду ссылаться на них вместе как на TDataSet.
TTable	Компонент C++Builder, обеспечивающий доступ к таблицам базы данных. Свойство TableName этого компонента используется для выбора конкретной таблицы, к которой вы хотите обратиться.
TQuery	Компонент C++Builder, позволяющий создавать, выполнять и обрабатывать ваши собственные запросы SQL.
TStoredProc	Компонент C++Builder, позволяющий запускать откомпилированные процедуры SQL, которые находятся на сервере базы данных. Такие процедуры называются *хранимыми процедурами (stored procedures)*.
Интерфейсный элемент (data-aware control)	Визуальный (видимый во время выполнения программы) компонент, который позволяет пользователю просматривать и изменять данные в базе, используя компоненты доступа к данным. Интерфейсные компоненты расположены во вкладке Data Controls палитры компонентов C++Builder. В большинстве случаев они будут казаться вам «интеллектуальными» версиями компонентов из вкладки Standard. Примерами интерфейсных компонентов служат TDBGrid, TDBNavigator и TDBEdit.
TDataSource	Компонент C++Builder, который обеспечивает связь компонентов TDataSet с интерфейсными элементами. Интерфейсные элементы обращаются к компонентам TDataSource, которые, в свою очередь, обращаются к компонентам TDataSet.
TField	Класс C++Builder, обеспечивающий доступ к полям таблицы базы данных. C++Builder создает производные этого класса, такие как TStringField и TIntegerField, когда вы используете редактор полей для добавления нового поля в форму. Владельцем компонентов TField, помещенных в форму, является сама форма, а не связанный с ними TDataSet.

Обзор архитектуры

Для работы с локальными и удаленными базами данных приложения C++Builder используют механизм доступа к СУБД, называемый Borland Database Engine (BDE). В случае локальных баз данных, таких как таблицы dBase или Paradox, BDE использует свои собственные встроенные драйверы СУБД. Для доступа к удаленным базам данных, например, Oracle или Sybase, BDE связывается с их серверами, используя драйверы SQL Links и/или драйверы ODBC. Эти драйверы часто обращаются к библиотекам драйверов самой СУБД. Схематически взаимодействие этих компонентов показано на рис. 16.1.

Рис. 16.1
В C++Builder реализована простая, но гибкая архитектура доступа к базам данных



Внутри вашего приложения интерфейсные компоненты обращаются к компонентам TDataSource. Обычно в форме содержится небольшое количество компонентов TDataSource, хотя интерфейсных компонентов может быть достаточно много. Компоненты TDataSource, в свою очередь, обращаются к одному или нескольким компонентам TDataSet. В некоторых случаях форма может включать только один компонент TDataSet и один TDataSource. Рис.16.2 иллюстрирует взаимосвязь названных компонентов.

Гибкость, присущая этой многоуровневой архитектуре, позволяет достаточно легко разрабатывать приложения для работы с базами данных, отличающиеся не только надежностью, но и возможностью расширения и изменения. Благодаря разделению внутренних драйверов BDE и компонентов приложения, возможно (по крайней мере, теоретически) изменять внутреннюю структуру базы данных даже без перекомпиляции приложения. Модульность архитектуры позволяет заменять отдельные участки, не разрабатывая заново все приложение.

Теперь позвольте сделать несколько утверждений общего характера, которые помогут лучше усвоить только что рассмотренные концепции. Возможно, вы скажете: «Эта архитектура выглядит отлично, но как ее использовать? Какое отношение она имеет ко мне? Что это все означает на практике?» Если вы действительно готовы задать такие вопросы, надеюсь, что следующие советы помогут вам.

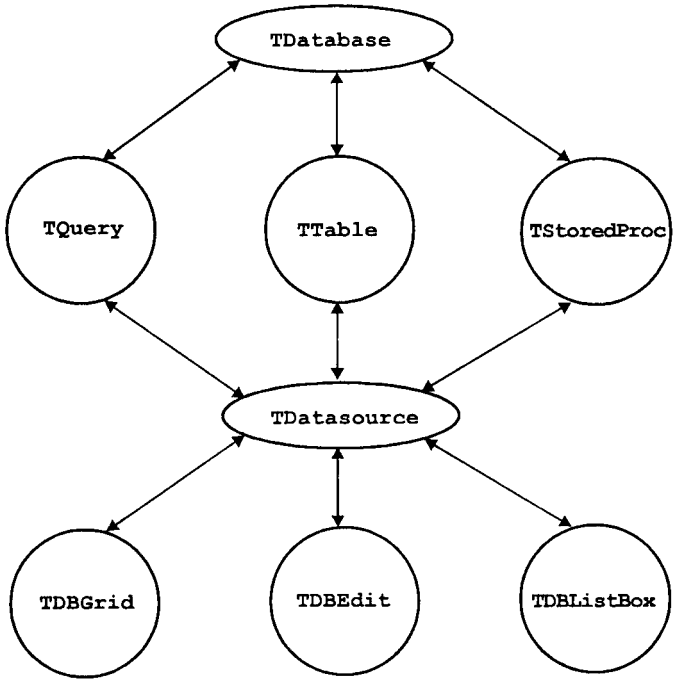


Рис. 16.2
Доступ к базе данных с точки зрения приложения C++Builder

- Вам не обязательно использовать компонент TDatabase для обращения к базам данных. Этот компонент предоставляет некоторые дополнительные возможности, которые вы можете использовать или не использовать, но он не является неотъемлемой частью приложений C++Builder, работающих с базами данных.
- Вероятно, вам не придется обращаться к компоненту TSession до тех пор, пока вы не станете разрабатывать многопоточные приложения для работы с базами данных. Многопоточные приложения открывают одновременно несколько каналов выполнения. Это означает, что несколько операций могут выполняться в одно и то же время. Обычные приложения, работающие с базами данных, не являются многопоточными, поэтому вам, как правило, не нужно беспокоиться о создании и использовании TSession. При запуске приложения, использующего базу данных, C++Builder автоматически создает экземпляр TSession (хранимый в глобальной переменной с именем Session). В случае однопоточного приложения это означает, что при необходимости использовать свойства или методы TSession вы можете просто обращаться к переменной Session.
- Вам не потребуются компоненты TQuery или TStoredProc до тех пор, пока вы не начнете писать свои собственные SQL-процедуры или обращаться к процедурам, хранимым на сервере. Вы можете открывать таблицы баз данных в любом из локальных или удаленных форматов, поддерживаемых C++Builder, используя только компонент TTable.
- Компонент TTable обычно используется для приема/передачи данных из базы. Как уже говорилось, TTable является центральной частью схемы доступа к базам данных в C++Builder. Вы будете использовать его для обращения к таблицам и обмена данными с интерфейсными компонентами.

- Вкладка Data Controls содержит визуальные интерфейсные компоненты, которые отображают данные на экране и позволяют их изменять. Они представляют собой «интеллектуальные» версии тех элементов управления, с которыми часто приходится иметь дело в приложениях Windows. Вы будете использовать эти компоненты для создания интерфейса пользователя в приложениях, работающих с базами данных. Доступ пользователя к базе данных становится возможен в результате взаимодействия интерфейсных компонентов с компонентами доступа к данным, такими как TTable.
- Производные TDataSet (например, TTable, TQuery и TStoredProc) предназначены для поиска данных в базе, но они не могут передавать эти данные непосредственно интерфейсным компонентам (таким, как TDBEdit). Передача данных осуществляется через компоненты TDataSource. Таким образом, интерфейсные компоненты наподобие TDBEdit не обращаются непосредственно к TDataSet, обеспечивающим доступ к данным. Вместо этого происходит обращение сначала к компонентам TDataSource, которые, в свою очередь, взаимодействуют с TDataSet.
- Для создания простой формы вам нужны три компонента: TTable, TDataSource и один из интерфейсных (TDBEdit, TDBMemo и т.д.).

Обзор компонентов C++Builder, обеспечивающих работу с базами данных

Теперь, когда вы получили общее представление об архитектуре доступа к базам данных в C++Builder, давайте рассмотрим соответствующие компоненты. Я буду брать компоненты по очереди и останавливаться на ключевых свойствах каждого из них. Тем самым мы изучим архитектуру доступа к базам данных более подробно.

Я намеренно опускаю компоненты, которые выступают в роли вспомогательных классов. Если класс в действительности не является компонентом, а служит только предком компонента — я не включаю его в обзор. Это относится, в частности, к классам TDataSet и TDBDataSet, которые являются предками компонентов TTable, TQuery и TStoreProc. Основное внимание будет уделено тем компонентам, которыми можно манипулировать с помощью визуальных инструментов C++Builder. Хотя иерархия классов-предков также важна, большинство свойств, событий и методов, о которых вам нужно знать, реализованы в компонентах C++Builder.

Описание каждого компонента состоит из трех частей:

- Описание компонента, название его исходного модуля и класса-предка, а также таблица основных свойств, методов и событий
- Ключевые элементы
- Основные задачи

Таблицы основных свойств, методов и событий не являются исчерпывающими — за дополнительной информацией обращайтесь к документации по C++Builder, а еще лучше — к исходным текстам VCL. Основная идея заключается в том, чтобы познакомить вас с архитектурой доступа к базам данных в C++Builder, не вдаваясь во все подробности VCL.

TSession

Модуль: DB Класс-предок: TComponent

Компонент TSession автоматически создается C++Builder при каждом запуске вашего приложения, поэтому обычно нет необходимости делать это вручную. Встроенный TSession доступен через глобальную переменную Session.

Как правило, для приложения достаточно одного компонента TSession. Единственным исключением служат многопоточные приложения. В этом случае может возникнуть необходимость в дополнительных компонентах TSession, чтобы открыть несколько независимых соединений с сервером для предотвращения интерференции между отдельными потоками. Вы можете помещать в форму или модуль данных столько компонентов TSession, сколько необходимо, и затем использовать их в компонентах TDatabase и TDataSet. В многопоточных приложениях обычно создается по одному TSession для каждого потока. Имейте в виду, что в C++Builder определена еще одна глобальная переменная, Sessions, для отслеживания нескольких компонентов TSession в одном приложении. Sessions имеет тип TSessionList и содержит запись для каждого компонента TSession.

В таблицах 16.2, 16.3 и 16.4 перечислены соответственно основные свойства, методы и события компонента TSession.

Таблица 16.2. Основные свойства TSession

Свойство	Описание
DatabaseCount	Возвращает значение счетчика активных TDatabase.
Databases	Возвращает массив активных TDatabase.
Handle	Обеспечивает доступ к дескриптору BDE; используется для прямых вызовов BDE.
KeepConnections	Определяет, нужно ли сохранять неактивные соединения.
NetFileDir	Определяет расположение файла PDOXUSRS.NET.
PrivateDir	Определяет расположение временных файлов.
SessionName	Содержит имя сеанса, передаваемое другим компонентам.

Таблица 16.3. Основные методы TSession

Метод	Функция
AddPassword	Устанавливает пароль для текущего сеанса (для соединений в Paradox).
CloseDatabase	Явно закрывает компонент TDatabase.
DropConnections	Разрывает все неактивные соединения TDatabase.
FindDatabase	Осуществляет поиск TDatabase по имени.
GetAliasNames	Возвращает доступные псевдонимы BDE.
GetAliasParams	Возвращает параметры данного псевдонима.
GetDatabaseNames	Выдает список всех доступных псевдонимов, включая локальные.

см. продолжение

Таблица 16.3. (продолжение)

Метод	Функция
GetDriverNames	Выдает список доступных драйверов BDE.
GetDriverParams	Возвращает параметры указанного драйвера.
GetPassword	Запрашивает пароль; возвращает true в случае правильного ввода.
GetTableNames	Выдает список всех таблиц указанной базы данных.
GetStoredProcNames	Выдает список всех хранимых процедур для указанной базы данных.
OpenDatabase	Явно открывает TDatabase.
RemoveAllPasswords	Удаляет все пароли Paradox.
RemovePassword	Удаляет данный пароль Paradox.

Таблица 16.4. Основные события TSession

Событие	Условия возникновения
OnPassword	Запрос BDE на ввод пароля для таблицы Paradox.
OnStartup	Активизация сеанса.

Ключевые элементы

TSession является основным компонентом, управляющим соединением приложения с базами данных. Свойство DatabaseCount служит для определения количества активных TDatabase; свойство Databases используется для доступа к ним по индексу.

Свойство SessionName компонента TSession, аналогичное свойству DatabaseName компонента TDatabase, позволяет вам указать имя сеанса, которое будет передаваться другим компонентам. В многопоточных приложениях вы должны установить одинаковые значения SessionName для компонента TSession и соответствующих ему компонентов TDatabase и TDataSet (TTable, TQuery или TStoreProc). При этом имя сеанса можно выбирать произвольным образом. Установка в компонентах TDatabase и, скажем, TQuery того же значения SessionName, что и в TSession, позволяет указать, какое соединение с базой данных должен использовать компонент TQuery. Более того, если вы будете затем взаимодействовать с TQuery из потока, вы должны явно указать, какое соединение использует поток. Для некоторых серверов это требование является необходимым, поскольку различные потоки могут интерферировать между собой.

Свойство KeepConnections определяет, будут ли неактивные соединения с базой данных сохраняться для временных компонентов TDatabase. Для разрыва всех неактивных соединений используйте свойство DropConnections. Учтите, что если все текущие соединения с сервером базы данных неактивны и вы разорвали их, в следующий раз придется входить на сервер заново. Можно, однако, установить KeepConnections в false и при этом избежать ввода имени пользователя и пароля при повторном соединении. (Эта возможность рассматривается в разделе, посвященном компоненту TDatabase.)

Расположение рабочего сетевого каталога BDE определяется свойством NetFileDir. Обратите внимание, что это свойство используется только для таблиц Paradox. Путь к временным файлам определяется свойством PrivateDir.

Основные задачи

Компонент TSession используется для обеспечения доступа к внутренним структурам BDE. Вы можете получить списки псевдонимов и их параметров, а также установки драйверов. С помощью свойства Handle осуществляются прямые обращения к API BDE.

Для вызова методов TSession можно использовать предопределенную переменную Session. Следующие три строки кода служат примерами таких вызовов.

В результате вызова этого метода содержимое ListBox1.Items заменяется на список определенных в данный момент псевдонимов BDE:

```
Session->GetAliasNames(ListBox1->Items);
```

Вызов этого метода заменяет содержимое ListBox1.Items на список всех псевдонимов, относящихся к BDE и приложению:

```
Session->GetDatabaseNames(ListBox1->Items);
```

А это обращение приводит к разрыву неактивных временных соединений с базой данных:

```
Session->KeepConnections = false;
```

Обратите внимание, что разрываются только временные соединения BDE, а не те, которые были созданы вами явно. Базы данных, имеющие свой собственный компонент TDatabase, используют свойство KeepConnection этого компонента.

Основная польза от разрыва неактивных соединений состоит в сохранении пропускной способности сети. Для локальных сетей это может практически не ощущаться. Однако при использовании глобальных сетей или удаленного доступа разница может быть очень существенна. Закрытие неиспользуемых соединений освобождает также ресурсы сервера и локальной машины, хотя это редко оправдывает необходимость повторного входа.

Вы можете обрабатывать событие OnLogin компонента TDatabase таким образом, чтобы пользователю не приходилось вводить имя и пароль при повторном соединении. С учетом этого, разрыв неактивных соединений может оказаться не столь нежелателен, как на первый взгляд. (Как это сделать, описано в разделе «Основные задачи» для компонента TDatabase.)

TDatabase

Модуль: DB Класс-предок: TComponent

Для доступа к базам данных явное использование компонента TDatabase не требуется. Однако к определенным параметрам соединения вы можете обращаться только с его помощью. Обычно в приложении используется только один компонент TDatabase. Если вы явно не включили компонент TDatabase в форму, C++Builder автоматически создает его временный экземпляр.

В таблицах 16.5, 16.6 и 16.7 перечислены соответственно основные свойства, методы и события компонента TDatabase.

Таблица 16.5. Основные свойства TDatabase

Свойство	Описание
AliasName	Ссылается на псевдоним, используемый BDE.
Connected	Определяет, открыт ли TDatabase.
DatabaseName	Определяет псевдоним базы данных, используемый внутри приложения.
DriverName	Указывает тип используемого драйвера.
KeepConnection	Определяет, нужно ли сохранять неактивные соединения.
LoginPrompt	Определяет, нужно ли выдавать приглашение на ввод имени пользователя и пароля.

Таблица 16.6. Основные методы TDatabase

Метод	Функция
Open	Явно открывает соединение с базой данных.
Close	Явно закрывает соединение с базой данных.

Таблица 16.7. Основные события TDatabase

Событие	Условия возникновения
OnLogin	Открытие SQL TDatabase при условии, что LoginPrompt содержит true.

Ключевые элементы

Свойство DatabaseName используется для определения зависящего от приложения, или локального, псевдонима BDE. Указанное здесь имя (при желании оно может совпадать с именем компонента) заносится в список возможных значений свойств DatabaseName компонентов TDataSet (например, TTable и TQuery). Затем вы можете выбрать его из списка для связывания компонента TDataSet с TDatabase.

Свойство AliasName содержит псевдоним BDE, который должен использоваться данным компонентом TDatabase для получения установок по умолчанию. Это псевдоним, который вы предварительно определили с помощью утилиты конфигурирования BDE. Обратите внимание, что данное свойство и свойство DriverName являются взаимоисключающими. Установка одного из них автоматически сбрасывает другое.

Если вы не используете AliasName, необходимо указать драйвер BDE с помощью свойства DriverName. Для локальных таблиц (dBASE и Paradox) используется драйвер STANDARD, а для SQL-серверов — драйверы INTERBASE, SYBASE, ORACLE или MSSQL. Повторю еще раз, что свойства DriverName и DatabaseName являются взаимоисключающими.

Свойство Connected открывает и закрывает соединение с базой данных. Вы можете установить для него значение true через инспектор объектов, чтобы открыть соединение во время разработки. Если вы открываете компонент TDataSet, который обращается к TDatabase, соединение открывается

автоматически. При закрытии TDatabase все связанные с ним компоненты TDataSet также закрываются.



Если вы определили в своем приложении AliasName, то для открытия компонентов TDataSet должны быть загружены форма или модуль с соответствующим компонентом TDatabase.

Чтобы избежать процедуры входа на сервер при каждом открытии базы данных, устанавливайте для свойства KeepConnection значение true.

Если для свойства LoginPrompt установлено значение true, при соединении с сервером базы данных у пользователя будет запрашиваться входная информация. Выдачи приглашения можно избежать, если использовать событие OnLogin (см. следующий раздел).

Основные задачи

Ваше приложение должно включать в себя компонент TDatabase для выполнения следующих операций:

- Установления постоянного соединения с базой данных
- Задания локального, используемого внутри приложения псевдонима базы данных
- Изменения параметров входа на сервер
- Управления процессом транзакции на сервере

Установление соединения с базой данных

Приложения C++Builder устанавливают соединения с SQL-серверами, используя драйверы SQL Links, входящие в Borland Database Engine. Эти драйверы обеспечивают доступ к СУБД InterBase, Sybase, Oracle и Microsoft.

Как правило, вы будете использовать Database Explorer или утилиту конфигурирования BDE для создания *псевдонимов* баз данных, с помощью которых ваше приложение будет устанавливать соединение с серверами. Псевдоним BDE представляет собой не более чем поименованный список параметров соединения, которые используются BDE для связи с базой данных. Созданный псевдоним появляется в списке возможных значений свойства DatabaseName компонентов TDataSet (например, TTable или TQuery). Вы можете переустановить значения по умолчанию, содержащиеся в псевдониме BDE, изменив свойство Params компонента TDatabase.

Сохранение соединений с базой данных

Если свойство KeepConnection имеет значение true, соединение с базой данных будет сохраняться даже при отсутствии открытых компонентов TDataSet. Это необходимо, если вы не хотите заново выполнять процедуру входа на сервер при следующем обращении к базе данных.



Не путайте свойство KeepConnection компонента TDatabase со свойством KeepConnections компонента TSession. Свойство KeepConnections компонента TSession влияет только на временные компоненты TDatabase. Установка этого свойства не сохраняет соединений для тех компонентов, которые вы создали явно.

Изменение параметров входа на сервер

Вы можете использовать обработчик события OnLogin компонента TDatabase, чтобы предотвратить отображение диалогового окна входа при установлении соединения. OnLogin принимает два аргумента: компонент TDatabase, который указывает на базу данных, в которую пользователь намерен войти, и объект TStrings, в котором хранятся параметры входа. Ниже приведен заголовок типичного обработчика OnLogin:

```
void __fastcall TForm1::Database1Login(TDatabase *Database, TStrings
*LoginParams)
```

Внутри обработчика вы можете использовать индексируемое свойство Values объекта TString для доступа к отдельным параметрам:

```
LoginParams->Values["SERVER NAME"] = "VH1";
LoginParams->Values["USER NAME"] = "dave";
LoginParams->Values["PASSWORD"] = "unreally";
```

Чтобы диалоговое окно входа не отображалось, необходимо установить по крайней мере параметр PASSWORD. Вы можете взять необходимые параметры из диалогового окна при первом входе, получить их от другого компонента TDatabase или вставить в текст программы. Если в обработчике OnLogin присутствуют какие-либо параметры, C++Builder попытается использовать их для установления соединения.

Выполнение транзакций под управлением приложения

Транзакция (transaction) — это операция над базой данных, заключающаяся в изменении каких-либо данных и рассматриваемая как единое целое. Транзакция может состоять как из одной команды на изменение данных, так и из тысячи. Когда вы объединяете серию изменений данных в одну транзакцию, будут выполнены либо все изменения, либо ни одно. Если транзакция состоит из 1000 изменений и 999-ое было неудачным, все изменения отменяются и база данных остается в исходном состоянии. Обычно C++Builder автоматически начинает и завершает транзакцию, когда приложение пытается внести изменения в базу данных. Если этого уровня управления недостаточно, вы можете сами организовать процесс транзакции, используя свойство TransIsolation и методы StartTransaction, Commit и Rollback.

Свойство TransIsolation определяет уровень изоляции транзакции (TIL — Transaction Isolation Level) на сервере базы данных. TIL управляет доступом одних конкурирующих транзакций к изменениям, сделанным другими.

TransIsolation может принимать одно из трех возможных значений: tiDirtyRead, tiReadCommitted и tiRepeatableRead (по умолчанию устанавливается tiReadCommitted). Установка этих значений имеет следующий эффект:

- tiDirtyRead — изменения, выполняемые другими транзакциям, видимы во время их выполнения.
- tiReadCommitted — изменения, выполняемые другими транзакциями, видимы только после их завершения.
- tiRepeatableRead — изменения другими транзакциями ранее считанных данных не видимы; это означает, что при каждом считывании данной записи транзакция получает *точно такую же* запись.

Метод StartTransaction отмечает начало группы изменений данных, которую вы хотите рассматривать как одно целое. Изменения либо произойдут в полном объеме, либо ни одно из них не будет выполнено.

Метод Commit делает постоянными те изменения в базе данных, которые произошли после начала транзакции. Можете рассматриать его как команду Save для базы данных.

Метод Rollback отменяет изменения, сделанные после начала транзакции. Можете считать его командой Undo применительно к базе данных.



Вы можете также управлять выполнением транзакции на сервере, используя Passthrough SQL. Для этого нужно использовать команды SQL, которые изменяют процесс транзакции. Имейте в виду, что если при этом для SQLPASSTHRUMODE установлено значение SHARED AUTOCOMMIT или SHARED NOAUTOCOMMIT, то новые установки TIL могут повлиять на другие транзакции, инициированные вашим приложением.

TTable

Модуль: `DBTables` Класс-предок: `TDBDataSet`

TTable является прямым потомком класса TDBDataSet и косвенным потомком класса TDataSet. С помощью компонента TTable осуществляется доступ к таблицам баз данных. Открывая TTable, вы устанавливаете соединение между вашим приложением и таблицей. Компонент TTable используется для добавления, изменения и удаления строк в таблице базы данных.

В таблицах 16.8, 16.9 и 16.10 перечислены соответственно основные свойства, методы и события компонента TTable.

Таблица 16.8. Основные свойства TTable

Свойство	Описание
Active	Открывает и закрывает TDataSet.
AutoCalcFields	Определяет способ вычислений с использованием полей.
Bof	Показывает, находится ли TDataSet на начале.
CachedUpdates	Определяет, будут ли кэшироваться обновленные данные.
Database	Идентифицирует TDatabase, используемый TDataSet.
DatabaseName	Определяет псевдоним, используемый для соединения с базой данных.
Eof	Показывает, достигнут ли конец TDataSet.
Exclusive	Определяет, могут ли другие пользователи обращаться к TDataSet.
FieldCount	Возвращает количество полей в TDataSet.
FieldDefs	Содержит информацию о полях TDataSet.
Fields	(Индексируемое) возвращает указанное поле TDataSet.
Filter	Определяет выражение, используемое для фильтрации записей.
Filtered	Определяет, какая фильтрация используется: указанная в Filter или в OnFilterRecord.
FilterOptions	Управляет работой фильтров.
IndexDefs	Выдает информацию об индексах TDataSet.

см. продолжение

Таблица 16.8. (продолжение)

Свойство	Описание
IndexFieldCount	Возвращает число полей в текущем индексном ключе.
IndexFieldNames	Указывает поля, используемые в качестве индексного ключа.
IndexName	Указывает имя используемого индекса.
IndexFields	(Индексируемое) возвращает указанное индексное поле TDataSet.
KeyExclusive	Инвертирует действие функций поиска и установки диапазона.
KeyFieldCount	Указывает количество ключевых полей, используемых для поиска.
MasterFields	Указывает главные поля в отношении главный/подчиненный.
MasterSource	Указывает главный DataSource в отношении главный/подчиненный.
Modified	Показывает, была ли данная запись изменена с момента последнего Post или Cancel.
ReadOnly	Определяет возможность изменения TDataSet.
RecordCount	Возвращает количество строк в TDataSet.
SessionName	Указывает, какой компонент TSession используется для соединения с базой данных.
State	Возвращает состояние TDataSet (например, dsEdit или dsBrowse).
TableName	Указывает физическое имя сопоставленной таблицы.
TableType	Указывает тип (локальной) таблицы.
UpdateMode	Определяет тип SQL, используемого для изменения данных.
UpdateObject	Определяет компонент UpdateSQL для использования совместно с кэшируемыми изменениями.

Таблица 16.9. Основные методы TTable

Метод	Функция
AddIndex	Создает новый индекс
Append	Добавляет пустую строку к TDataSet и переводит его в режим редактирования.
AppendRecord	Добавляет строку к TDataSet, используя указанные значения.
ApplyRange	Активизирует диапазон, установленный методами Set/EditRange.
ApplyUpdates	Сохраняет кэшированные изменения в базе данных.
BatchMove	Копирует группу строк из одного TDataSet в другой.
Cancel	Отменяет ожидаемое изменение текущей строки.
CancelRange	Отменяет установки, сделанные методами Set/EditRange.
CancelUpdates	Отменяет ждущие изменения, занесенные в кэш.
ClearFields	Устанавливает для полей текущей строки значения по умолчанию.
Close	Закрывает TDataSet.
CreateTable	Создает новую таблицу.
Delete	Удаляет текущую запись.

Метод	Функция
DeleteIndex	Удаляет вторичный индекс.
DeleteTable	Удаляет соответствующую физическую таблицу в базе данных.
Edit	Переводит TDataSet в режим редактирования.
EditKey	Позволяет изменять значения ключа поиска.
EditRangeEnd	Позволяет изменить верхний ключевой предел диапазона.
EditRangeStart	Позволяет изменить нижний ключевой предел диапазона.
EmptyTable	Удаляет все строки в TDataSet.
EnableControls	Разрешает использование интерфейсных компонентов.
FetchAll	Считывает все ожидающие своей очереди строки базы данных.
FieldByName	Возвращает TField, используя имя поля в базе данных.
FindFirst	Осуществляет поиск записи, используя указанное условие фильтрации.
FindNext	Осуществляет поиск следующей записи, соответствующей заданному критерию.
FindKey	Выполняет точный поиск в TDataSet.
FindNearest	Выполняет приближенный поиск в TDataSet.
GetFieldNames	Возвращает список полей в TDataSet.
GetIndexNames	Возвращает список индексов TDataSet.
GotoKey	Выполняет точный поиск в TDataSet по ключу.
GotoNearest	Выполняет приближенный поиск в TDataSet по ключу.
Insert	Вставляет пустую строку и позволяет ее редактировать.
InsertRecord	Вставляет строку, используя заданные значения столбцов.
Locate	Осуществляет поиск записи в TDataSet.
LockTable	Блокирует локальную таблицу.
Lookup	Осуществляет поиск записи в TDataSet и возвращает значения ее полей.
MoveBy	Перемещает курсор TDataSet на заданное количество строк.
Open	Открывает TDataSet.
Post	Сохраняет ждущие изменения текущей строки.
RenameTable	Переименовывает локальную таблицу.
RevertRecord	Отменяет кэшированные изменения текущей строки.
SetKey	Переводит TDataSet в режим поиска по ключу.
SetRange	Переводит TDataSet в режим поиска по диапазону.
SetRangeEnd	Устанавливает верхний предел диапазона.
SetRangeStart	Устанавливает нижний предел диапазона.
UnlockTable	Разблокирует локальную таблицу.

Таблица 16.10. Основные события TTable

Событие	Условия возникновения
AfterCancel	После вызова Cancel.
AfterClose	После закрытия TDataSet.
AfterDelete	После вызова Delete.
AfterEdit	После вызова Edit.
AfterInsert	После вызова Insert.
AfterOpen	После открытия TDataSet.
AfterPost	После вызова Post.
BeforeCancel	Перед выполнением Cancel.
BeforeClose	Перед закрытием TDataSet.
BeforeDelete	Перед выполнением Delete.
BeforeEdit	Перед выполнением Edit.
BeforeInsert	Перед выполнением Insert или Append.
BeforeOpen	Перед открытием TDataSet.
BeforePost	Перед выполнением Post.
OnCalcField	При необходимости ввода значений в вычисляемые поля.
OnDeleteError	При ошибке удаления записи.
OnEditError	При ошибке редактирования записи.
OnFilterRecord	В режиме фильтрации, когда TDataSet требует строку.
OnNewRecord	При добавлении новой записи к TDataSet.
OnPostError	При ошибке сохранения записи.
OnUpdateError	При ошибке во время сохранения кэшированных изменений.
OnUpdateRecord	При сохранении каждой строки с помощью ApplyUpdates.

Ключевые элементы

Свойство DatabaseName используется для указания базы данных, к которой вы хотите получить доступ. Здесь указывается либо псевдоним, используемый внутри приложения, либо псевдоним, определенный с помощью Database Explorer или утилиты конфигурирования BDE.

Свойство TableName указывает физическую таблицу базы данных. Для некоторых СУБД оно может также включать имя базы данных, содержащей таблицу и/или имя владельца таблицы.

Для использования вторичного индекса установите свойство IndexName или IndexFields. Для установления связи с другой таблицей, свойство MasterSource должно ссылаться на компонент TDataSet, использующий тот же ключ. Присвоив значение свойству MasterSource, укажите ключевые поля в главном TDataSource, используя свойство MasterFields. Эти ключи должны соответствовать ключам текущего индекса, определенным в IndexName или IndexFields.

Обратите внимание, что двойным щелчком на свойстве MasterFields вы можете вызвать редактор связи полей (Field Link Designer), который позволит вам визуально установить отношения главный/подчиненный.

Установка для свойства Active значения true эквивалентна вызову метода Open компонента TDataSet и приводит к открытию TDataSet. Аналогично, установка для Active значения false эквивалентна вызову Close и приводит к закрытию TDataSet.

Вы можете узнать текущее состояние TDataSet с помощью свойства State. Оно принимает следующие значения:

- dsInactive — TDataSet закрыт.
- dsBrowse — TDataSet находится в режиме Browse (Обзор). Данные можно просматривать, но их изменение невозможно, пока State не примет значение dsEdit.
- dsEdit — TDataSet находится в режиме Edit (Редактирование), позволяя изменять данные.
- dsInsert — TDataSet находится в режиме Insert (Вставка).
- dsSetKey — TDataSet находится в режиме SetKey (Установка ключа) после вызова метода SetKey. Значения, присвоенные столбцам в этом режиме, интерпретируются как ключевые. Последующий вызов GotoKey осуществляет поиск записи с использованием данных значений.
- dsCalcFields — вызван обработчик события OnCalcField.

Основные задачи

Метод First перемещает указатель текущей записи (называемый также курсором) в начало TDataSet, а метод Last — в конец. Методы Prior и Next перемещают курсор соответственно на предыдущую или следующую строку. Используя метод MoveBy, можно переместиться на любое число строк вперед или назад.

Методы SetKey, FindKey, GotoKey, FindNearest и GotoNearest могут быть использованы для поиска в TDataSet заданного набора значений полей.

Свойство Bof определяет, находится ли курсор в начале TDataSet. Чтобы определить, достигнут ли конец TDataSet, используйте свойство Eof. Эти два свойства полезны при организации циклов. Вот пример простого цикла, который перебирает строки и отображает их поля:

```
Table1->First();
while (!(Table1->Eof)) {
  ShowMessage("Category is: "+Table1->FieldByName("Category")->Value);
  Table1->Next();
}
```



Будьте осторожны, чтобы не сделать неверных предположений о свойствах Bof и Eof. Нельзя считать, что Bof имеет значение true только потому, что вы находитесь на первой строке таблицы. Нельзя также считать, что Eof имеет значение true, когда вы находитесь на последней строке. Обычно требуется дополнительный вызов Prior или Next, чтобы установить Bof или Eof в true. Например, последовательность First, Next, Prior не установит Bof в true, тогда как First, Next, Prior, Prior сделает это. Обратите внимание, что Bof принимает значение true немедленно после открытия таблицы или вызова метода First, а Eof — после вызова метода Last.

Методы Append и Insert используются для добавления пустых строк к TDataSet. Append добавляет запись в конец TDataSet, а Insert вставляет

запись в текущей позиции курсора. Как Append, так и Insert переводят TDataSet в режим dsEdit. Методы AppendRecord и InsertRecord используются для добавления к TDataSet заполненных строк, поля которых содержат указанные значения.

Метод Delete удаляет строку в текущей позиции курсора. Метод Edit позволяет модифицировать строки, переводя TDataSet в режим редактирования. Метод Post сохраняет эти изменения в базе данных, тогда как Cancel отменяет их. Это справедливо также и для методов Append и Insert — вы можете сохранить или отменить результат их вызова.

Локальные фильтры

Свойства Filter, Filtered и FilterOptions служат для установки локальных фильтров в TDataSet. Локальная фильтрация осуществляется внутри приложения. Это дает определенные преимущества для TDataSet с малым количеством строк; в этом случае вся таблица располагается в кэше BDE и обращение к серверу базы данных не требуется.

Filter позволяет вам указать выражение, которое ограничивает количество строк, видимых в TDataSet. Синтаксис, используемый в выражениях, похож на синтаксис оператора WHERE языка SQL. Поля можно сравнивать друг с другом и с постоянными величинами. В фильтрующих выражениях можно использовать следующие операторы:

Оператор	Значение
<	Меньше
>	Больше
>=	Больше или равно
<=	Меньше или равно
=	Равно
<>	Не равно
()	Окружает отдельный элемент составного выражения
[]	Окружает имена полей с пробелами
AND, OR, NOT	Объединяет отдельные элементы составных выражений

Вы можете также фильтровать записи, используя событие OnFilterRecord. Обработчик этого события выглядит следующим образом:

```
void __fastcall TForm1::Table1FilterRecord(TDataSet *TDataSet,
      Boolean &Accept)
{
  Accept = (dlr == vhlead);
}
```

Обработчик устанавливает параметр Accept в зависимости от того, удовлетворяет ли строка заданному критерию. Обратите внимание, что TDataSet, к которому относится фильтр, также передается в качестве параметра. Заметьте, что для установки параметра Accept используется вариантное свойство Value.

С помощью методов FindFirst, FindNext, FindPrior и FindLast вы можете осуществлять поиск в TDataSet, используя выражения-фильтры. FindFirst

находит первую строку, удовлетворяющую критерию фильтрации, FindNext находит следующую строку. FindPrior находит предыдущую строку, соответствующую заданному критерию, а FindLast — последнюю.

FilterOptions представляет собой набор, который может включать два возможных элемента:

Элемент	Значение
foCaseInsensitive	Фильтрация осуществляется без учета регистра
foNoPartialCompare	Частичное соответствие полей не допускается

Диапазоны

Методы SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, ApplyRange и SetRange позволяют ограничить диапазон строк, которые видны приложению. В отличие от более гибких локальных фильтров C++Builder, строки внутри диапазона должны соответствовать последовательному набору ключей индекса, если вы имеете дело с локальными таблицами. Для SQL-таблиц можно использовать любые поля, перечисленные в свойстве IndexFieldNames. Метод CancelRange позволяет снова сделать видимыми все строки.

Locate/Lookup

Методы Locate и Lookup позволяют вам осуществлять поиск строк в таблице. Они обладают гораздо большей гибкостью, чем семейство функций FindKey/SetKey, поскольку не требуют использования индекса. В силу этого их можно использовать также с компонентами TQuery и TStoredProc. Вы просто выбираете нужные данные, а BDE находит оптимальный способ доступа к ним.

Locate

Метод Locate имеет три параметра: строку, указывающую поле (или поля), в которых должен осуществляться поиск, вариантный список значений для поиска и переменную TLocateOptions, представляющую собой набор опций поиска. Locate имеет следующий синтаксис:

```
System::Boolean __fastcall Locate(const System::AnsiString KeyFields,
const System::Variant &KeyValues, TLocateOptions Options);
```

Имена полей разделяются в параметре KeyFields точкой с запятой. Для передачи их значений используется вариантный массив KeyValue. Параметр Options имеет тип TLocateOptions и позволяет указывать опции, управляющие поиском. Этот параметр является набором и имеет два возможных значения, loCaseInsensitive и loPartialKey. Первая опция, loCaseInsensitive, указывает Locate не учитывать регистр. Вторая, loPartialKey, разрешает поиск частичных совпадений. Вы можете передать либо одну, либо обе опции, присваивая их переменной типа set:

```
TLocateOptions SearchOpts
SearchOpts << loPartialKey;
Table->Locate("NAME", "Cri", SearchOpts);
```

Locate использует наискорейший способ поиска. Если существующий индекс подходит для реализации поиска по заданному критерию, Locate использует его. Если такого индекса не существует, создается фильтр BDE. В любом случае, будет выбран самый быстрый путь для получения нужных данных.

Locate возвращает true, если запрашиваемые данные найдены, и false в противном случае.

`Lookup`

Похожая на функцию Locate, функция Lookup имеет три параметра: строку, содержащую список столбцов, в которых должен производиться поиск, вариантную переменную или массив со значениями для поиска, и строку со списком имен столбцов, возвращаемых в качестве результата. Lookup имеет следующий синтаксис:

```
System::Boolean __fastcall Lookup(const System::AnsiString KeyFields,
const System::Variant &KeyValues, const System::AnsiString ResultCo-
lumns);
```

Кроме выполнения поиска в TDataSet, Lookup возвращает значения. Если соответствующая строка не может быть найдена, возвращается NULL. Сначала Lookup просматривает поля KeyFields; затем возвращаются значения полей, перечисленных в ResultColumns. Если список ResultColumns содержит несколько полей, результатом будет вариантный массив; в противном случае возвращается простая вариантная переменная.

Кэшируемые изменения

Механизм кэширования изменений C++Builder обеспечивает задержку записи измененной величины в базу данных. Вы можете сами решить, когда произвести обновление данных в базе, и записать все измененные данные за одно обращение. Обычно сделанные изменения записываются сразу. Однако в случае приложений клиент/сервер это увеличивает загрузку сети, поскольку ваше приложение должно каждый раз обращаться к серверу. Кэширование позволяет вам контролировать моменты и частоту этих обращений. Поскольку кэширование осуществляется локально, оно существенно увеличивает производительность, особенно при работе с очень медленными глобальными сетями (WAN — Wide Area Network).

Дополнительное преимущество кэширования заключается в возможности изменения компонентов TDataSet, предназначенных только для чтения. Наличие контроля за SQL, сгенерированными для обновления TDataSet, позволяет вам написать код, модифицирующий данные.

Для работы с кэшируемыми изменениями предназначены четыре метода: ApplyUpdates, CancelUpdates, CommitUpdates и RevertRecord. Их функции сведены в следующую таблицу:

Метод	Функция
ApplyUpdates	Сохраняет кэшированные изменения в базе данных.
CancelUpdates	Отменяет кэшированные изменения.
CommitUpdates	Уведомляет кэш о сохранении изменений.
RevertUpdates	Возвращает строку в состояние, которое она имела до изменения, занесенного в кэш.

Существует также несколько свойств, относящихся непосредственно к кэшируемым изменениям:

Свойство	Описание
CachedUpdates	Включает и выключает режим кэширования изменений.
UpdateRecordType	Определяет строки, видимые в кэше.

Использование кэша в приложении выглядит следующим образом:

1. Установите значение true для свойства CachedUpdates того компонента TDataSet, изменения в котором вы хотите кэшировать.
2. С помощью свойства UpdateRecordTypes определите строки, которые будут кэшироваться. Это свойство представляет собой набор и может иметь следующие значения: rtModified, rtInserted, rtDeleted и rtUnmodified. Перечисленные значения определяют тип строк, видимых в том TDataSet, изменения которого будут кэшироваться.
3. Установите обработчик события OnUpdateError для обработки ошибок, которые могут возникнуть при вызове ApplyUpdates.
4. Измените данные TDataSet.
5. Вызовите метод ApplyUpdates для сохранения изменений или метод CancelUpdates для их отмены.

Кэширование изменений находит хорошее применение в формах, предназначенных для ввода данных. Существуют три основных типа форм, работающих с базами данных: формы поддержки принятия решений, формы обработки транзакций и формы ввода данных. Поскольку пользователи форм ввода данных обычно добавляют новые строки, имеет смысл кэшировать их локально и затем сохранять за одно обращение к базе данных. Это уменьшит время, в течение которого таблица недоступна, и ускорит работу приложения.

Изменение TDataSet типа read-only будет рассмотрено позже, при обсуждении компонента UpdateSQL.

On...Error

События OnEditError, OnDeleteError и OnPostError позволяют вам реагировать на ошибки, происходящие при модификации данных в TDataSet. Обработчикам всех этих событий передаются три параметра: TDataSet, в котором произошла ошибка, класс исключения, вызванного ошибкой и параметр var, позволяющий вам указать действие, которое должно быть выполнено после вызова обработчика. Далее приводится пример обработчика, сгенерированного C++Builder для событий On...Error:

```
void __fastcall TForm1::Table1DeleteError(TDataSet *TDataSet,
EDatabaseError *E, TDataAction &Action)
{
}
```

Вы можете установить для Action любое из трех значений: daFail, daAbort или daRetry.

TQuery

Модуль: DBTables Класс-предок: TDBDataSet

Так же, как и TTable, TQuery является прямым потомком класса DBDataSet и косвенным потомком класса TDataSet. TQuery используется для посылки явных операторов SQL системе управления базой данных. Эти SQL выполняются над локальными таблицами или передаются серверу базы данных. Для выполнения запроса и возврата результата, нужно использовать метод Open или установить для свойства Active значение true. Предполагая, что запрос удовлетворяет ограничениям C++Builder на «активные» запросы, вы можете затем рассматривать результат как таблицу, что примерно соот-

ветствует работе оператора SQL VIEW на многих серверах баз данных. Вы можете изменить, добавить и удалить строки этой таблицы точно так же, как при использовании компонента TTable.

В таблицах 16.11, 16.12 и 16.13 перечислены соответственно основные свойства, методы и события компонента TQuery.

Таблица 16.11. Основные свойства TQuery

Свойство	Описание
Active	Открывает и закрывает TDataSet.
AutoCalcFields	Определяет способ вычислений с использованием полей.
Bof	Показывает, находится ли TDataSet на начале.
CachedUpdates	Определяет, будут ли кэшироваться обновленные данные.
Constrained	Контролирует допустимые изменения для активных результатов.
Database	Идентифицирует TDatabase, используемый TDataSet.
DatabaseName	Устанавливает псевдоним, используемый для соединения с базой данных.
DataSource	Определяет TDataSource для получения параметров запроса.
DBHandle	Возвращает дескриптор низкоуровневого соединения BDE.
Eof	Показывает, достигнут ли конец TDataSet.
FieldCount	Возвращает количество полей в TDataSet.
FieldDefs	Содержит информацию о полях TDataSet.
Fields	(Индексируемое) возвращает указанное поле TDataSet.
Filter	Определяет выражение, используемое для фильтрации записей.
Filtered	Определяет, какая фильтрация используется: указанная в Filter или в OnFilterRecord.
FilterOptions	Управляет работой фильтров.
Handle	Возвращает дескриптор низкоуровневого курсора BDE.
Modified	Показывает, была ли данная запись изменена с момента последнего Post или Cancel.
ParamCount	Показывает количество параметров SQL-запроса.
Params	Определяет параметры, используемые для SQL-запроса.
Prepared	Показывает, готов ли запрос.
RecordCount	Возвращает количество строк в TDataSet.
RequestLive	Определяет, будут ли результаты запроса изменяемыми.
SessionName	Указывает, какой компонент TSession используется для соединения с базой данных.
SQL	Определяет SQL-операторы для выполнения на сервере.
State	Возвращает состояние TDataSet (например, dsEdit или dsBrowse).
StmtHandle	Возвращает низкоуровневый дескриптор BDE для последнего результата запроса.
UniDirectional	Устанавливает движение курсора только в одном направлении.
UpdateMode	Определяет тип SQL, используемого для изменения данных.
UpdateObject	Определяет компонент UpdateSQL для использования совместно с кэшируемыми изменениями.

Таблица 16.12. Основные методы TQuery

Метод	Функция
Append	Добавляет пустую строку к TDataSet и переводит его в режим редактирования.
AppendRecord	Добавляет строку к TDataSet, используя указанные значения.
ApplyUpdates	Сохраняет кэшированные изменения в базе данных.
Cancel	Отменяет ждущее изменение текущей строки.
CancelUpdates	Отменяет ждущие изменения, занесенные в кэш.
ClearFields	Устанавливает для полей текущей строки значения по умолчанию.
Close	Закрывает TDataSet.
Delete	Удаляет текущую запись.
Edit	Переводит TDataSet в режим редактирования.
ExecSQL	Выполняет SQL без возврата курсора.
FetchAll	Считывает все строки базы данных, ожидающие своей очереди.
FieldByName	Возвращает TField, используя имя поля в базе данных.
FindKey	Выполняет точный поиск в TDataSet.
GetFieldNames	Возвращает список полей в TDataSet.
Insert	Вставляет пустую строку и позволяет ее редактировать.
InsertRecord	Вставляет строку, используя заданные значения столбцов.
Locate	Осуществляет поиск записи в TDataSet.
Lookup	Осуществляет поиск записи в TDataSet и возвращает значения ее полей.
MoveBy	Перемещает курсор TDataSet на заданное количество строк.
Open	Открывает TDataSet.
ParamByName	Возвращает параметр запроса по его имени.
Post	Сохраняет ждущие изменения текущей строки.
RevertRecord	Отменяет кэшированные изменения текущей строки.

Таблица 16.13. Основные события TQuery

Событие	Условие возникновения
AfterCancel	После вызова Cancel.
AfterClose	После закрытия TDataSet.
AfterDelete	После вызова Delete.
AfterEdit	После вызова Edit.
AfterInsert	После вызова Insert.
AfterOpen	После открытия TDataSet.
AfterPost	После вызова Post.
BeforeCancel	Перед выполнением Cancel.
BeforeClose	Перед закрытием TDataSet.

см. продолжение

Таблица 16.13. (продолжение)

Событие	Условие возникновения
BeforeDelete	Перед выполнением Delete.
BeforeEdit	Перед выполнением Edit.
BeforeInsert	Перед выполнением Insert или Append.
BeforeOpen	Перед открытием TDataSet.
BeforePost	Перед выполнением Post.
OnCalcField	При необходимости ввода значений в вычисляемые поля.
OnDeleteError	При ошибке удаления записи.
OnEditError	При ошибке редактирования записи.
OnFilterRecord	В режиме фильтрации, когда TDataSet требует строку.
OnNewRecord	При добавлении новой записи к TDataSet.
OnPostError	При ошибке сохранения записи.
OnUpdateError	При ошибке во время сохранения обновленного содержимого кэша.
OnUpdateRecord	При сохранении каждой строки с помощью ApplyUpdates.

Ключевые элементы

Вы можете использовать операторы SQL, не возвращающие значений. К ним относятся, например, INSERT, UPDATE и DELETE. Для запросов такого типа используется метод ExecSQL.

Свойство DatabaseName определяет базу данных, к которой вы хотите направить запрос. Свойство SQL служит для указания одиночного оператора SQL, который будет использоваться в запросе. При обращении к локальной таблице используйте локальный SQL. При обращении к таблицам на сервере можно использовать любой синтаксис, поддерживаемый этим сервером, пока вы не намереваетесь изменять результаты запроса. Для изменения результатов необходимо использовать синтаксис локального SQL, чтобы СУБД могла определить, какая из таблиц базы данных будет обновляться.

Оператор SQL может быть как статическим, так и включать параметры, которые будут динамически заменяться фактическими значениями. Параметры запроса, использующего заменяемые параметры (называемого *динамическим* запросом SQL) выделяются двоеточием:

```
SELECT * FROM ORDERS
WHERE CustomerNumber=:CustNo
```

В этом примере CustNo является именем заменяемого параметра. Эти параметры указываются в свойстве Params компонента TQuery.



При редактировании свойства SQL компонента TQuery вы можете использовать редактор кода C++Builder. Для его вызова щелкните на кнопке Code Editor в редакторе свойства SQL. Вы увидите, что редактор кода C++Builder намного универсальнее, чем компонент TMemo, используемый в качестве редактора по умолчанию.

Свойство Constrained позволяет вам указать, какие изменения могут быть сделаны в активных результатах запроса. Если вы установили для Constrained значение true, то изменения, результатом которых будет удаление строки из набора данных, станут невозможны. Пусть, например, компонент TQuery должен возвращать данные только о тех покупателях, чьи фамилии начинаются на A. В этом случае попытка изменить столбец LastName так, чтобы фамилии начинались на B, потерпит неудачу. Это во многом похоже на опцию WITH CHECK SQL-оператора VIEW.

Основные задачи

Для получения активных, или изменяемых, результатов запроса необходимо выполнение двух условий. Во-первых, свойство RequestLive должно иметь значение true. Во-вторых, используемый SQL должен соответствовать определенным требованиям. Набор требований зависит от того, к каким таблицам вы обращаетесь — локальным или расположенным на сервере. Для локальных таблиц SQL должен:

- ☐ Использовать только локальный синтаксис SQL.
- ☐ Работать только с одной таблицей.
- ☐ Не иметь оператора ORDER BY.
- ☐ Не содержать составных функций.
- ☐ Не содержать вычисляемых полей.
- ☐ Использовать в операторе WHERE сравнения имен столбцов только со скалярными константами. К поддерживаемым операторам относятся LIKE, >, <, >= и <=. Отдельные элементы оператора могут быть объединены также с помощью AND или OR.

Для таблиц, расположенных на сервере, SQL должен:

- ☐ Использовать только локальный синтаксис SQL.
- ☐ Работать только с одной таблицей.
- ☐ Не содержать составных функций.



Компонент TLiveQuery, находящийся на прилагаемом к этой книге CD-ROM, предоставляет альтернативный способ изменения результатов запроса. Для этого на сервере базы данных создается и открывается временный файл просмотра, который можно изменять так же, как и обычную таблицу. В этом случае возможности изменения результатов зависят только от ограничений, которые сервер накладывает на изменяемые файлы просмотра.

Метод First устанавливает TDataSet на начало, а метод Last — в конец. Методы Prior и Next перемещают курсор на предыдущую и следующую строку, соответственно. Метод MoveBy используется для перемещения на определенное число строк вперед или назад относительно текущей строки.

С помощью свойств Bof и Eof можно определить, находится ли курсор TDataSet на начале или в конце таблицы. Эти два свойства полезны при организации циклов, в которых просматриваются строки.



Будьте осторожны, чтобы не сделать неверных предположений о свойствах Bof и Eof. Нельзя считать, что Bof имеет значение true только потому, что вы находитесь на первой строке таблицы. Нельзя также считать, что Eof имеет значение true, когда вы находитесь на последней строке. Обычно требуется дополнительный вызов Prior или Next, чтобы установить Bof или Eof в true. Например, последовательность First, Next, Prior не установит Bof в true, тогда как First, Next, Prior, Prior сделает это. Обратите внимание, что Bof принимает значение true немедленно после открытия таблицы или вызова метода First, а Eof — после вызова метода Last.

Методы Append и Insert используются для добавления пустых строк в TDataSet. Append добавляет запись в конец таблицы, а Insert вставляет ее в текущей позиции курсора. AppendRecord и InsertRecord используются для добавления строк, поля которых содержат заданные значения.

Метод Delete удаляет строку в текущей позиции курсора. Метод Edit позволяет изменять строки, переводя TDataSet в режим редактирования. Метод Post сохраняет изменения в базе данных, а Cancel — отменяет их.

Другие свойства, методы и события описаны в предыдущем разделе, посвященном компоненту TTable.

TStoredProc

Модуль: `DBTables` Класс-предок: `TDBDataSet`

Так же, как TTable и TQuery, TStoredProc является прямым потомком класса DBDataSet и косвенным потомком класса TDataSet. Это означает, что в дополнение к методам, свойствам и событиям, определенным в самом классе, TStoreProc наследует некоторые члены от класса DBDataSet. Таким образом, существует много общего между тремя компонентами, основанными на TDataSet: TTable, TQuery и TStoredProc.

Компонент TStoredProc используется для выполнения хранимых процедур в приложениях C++Builder. Хранимые процедуры представляют собой откомпилированный набор операторов SQL, выполняемых как одна программа. TStoredProc обеспечивает интерактивную работу с результатами, которые возвращают хранимые процедуры.

В таблицах 16.14, 16.15 и 16.16 перечислены соответственно основные свойства, методы и события компонента TStoredProc.

Таблица 16.14. Основные свойства TStoredProc

Свойство	Описание
Active	Открывает и закрывает TDataSet.
AutoCalcFields	Определяет способ вычислений с использованием полей.
Bof	Показывает, находится ли TDataSet на начале.
CachedUpdates	Определяет, будут ли кэшироваться обновленные данные.
Database	Идентифицирует TDatabase, используемый TDataSet.
DatabaseName	Устанавливает псевдоним, используемый для соединения с базой данных.
Eof	Показывает, достигнут ли конец TDataSet.

Свойство	Описание
FieldCount	Возвращает количество полей в TDataSet.
FieldDefs	Содержит информацию о полях TDataSet.
Fields	Возвращает указанное поле TDataSet. (Требует индекс в качестве аргумента.)
Filter	Определяет выражение, используемое для фильтрации записей.
Filtered	Определяет, какая фильтрация используется: указанная в Filter или в OnFilterRecord.
FilterOptions	Управляет работой фильтров.
Modified	Показывает, была ли данная запись изменена с момента последнего Post или Cancel.
Overload	Указывает перегруженную процедуру для использования в СУБД Oracle.
ParamBinMode	Определяет способ связи Params с параметрами процедуры.
ParamCount	Показывает количество параметров SQL-запроса.
Params	Определяет параметры, используемые для SQL-запроса.
Prepared	Показывает, готов ли запрос.
RecordCount	Возвращает количество строк в TDataSet.
SessionName	Указывает, какой компонент TSession используется для соединения с базой данных.
State	Возвращает состояние TDataSet (например, dsEdit или dsBrowse).
StmtHandle	Возвращает низкоуровневый дескриптор BDE для последнего результата запроса.
StoredProcName	Указывает имя процедуры, подлежащей выполнению.
UpdateObject	Определяет компонент UpdateSQL для использования совместно с кэшируемыми изменениями.

Таблица 16.15. Основные методы TStoredProc

Метод	Функция
Append	Добавляет пустую строку к TDataSet и переводит его в режим редактирования.
AppendRecord	Добавляет строку к TDataSet, используя указанные значения.
ApplyUpdates	Сохраняет кэшированные изменения в базе данных.
Cancel	Отменяет ждущее изменение текущей строки.
CancelUpdates	Отменяет ждущие изменения, занесенные в кэш.
Close	Закрывает TDataSet.
Delete	Удаляет текущую запись.
Edit	Переводит TDataSet в режим редактирования.
ExecProc	Выполняет хранимую процедуру.
FetchAll	Считывает все строки базы данных, ожидающие своей очереди.
FieldByName	Возвращает TField, используя имя поля в базе данных.

см. продолжение

Таблица 16.15. (продолжение)

FindKey	Выполняет точный поиск в TDataSet.
GetFieldNames	Возвращает список полей в TDataSet.
GetResults	Возвращает выходные параметры хранимой процедуры Sybase.
Insert	Вставляет пустую строку и позволяет ее редактировать.
InsertRecord	Вставляет строку, используя указанные значения столбцов.
Locate	Осуществляет поиск записи в TDataSet.
Lookup	Осуществляет поиск записи в TDataSet и возвращает значения ее полей.
MoveBy	Перемещает курсор TDataSet на заданное количество строк.
Open	Открывает TDataSet.
ParamByName	Возвращает параметр запроса по его имени.
Post	Сохраняет ждущие изменения текущей строки.

Таблица 16.16. Основные события TStoredProc

Событие	**Условие возникновения**
AfterCancel	После вызова Cancel.
AfterClose	После закрытия TDataSet.
AfterDelete	После вызова Delete.
AfterEdit	После вызова Edit.
AfterInsert	После вызова Insert.
AfterOpen	После открытия TDataSet.
AfterPost	После вызова Post.
BeforeCancel	Перед выполнением Cancel.
BeforeClose	Перед закрытием TDataSet.
BeforeDelete	Перед выполнением Delete.
BeforeEdit	Перед выполнением Edit.
BeforeInsert	Перед выполнением Insert или Append.
BeforeOpen	Перед открытием TDataSet.
BeforePost	Перед выполнением Post.
OnCalcField	При необходимости ввода значений в вычисляемые поля.
OnDeleteError	При ошибке удаления записи.
OnEditError	При ошибке редактирования записи.
OnFilterRecord	В режиме фильтрации, когда TDataSet требует строку.
OnNewRecord	При добавлении новой записи к TDataSet.
OnPostError	При ошибке сохранением записи.
OnUpdateError	При ошибке во время сохранения обновленного содержимого кэша.
OnUpdateRecord	При сохранении каждой строки с помощью ApplyUpdates.

Ключевые элементы

Свойство DatabaseName используется для указания базы данных, к которой вы хотите получить доступ. Оно может представлять собой как локальный, используемый внутри приложения псевдоним, так и псевдоним, определенный с помощью Database Explorer или утилиты конфигурирования BDE.

Свойство StoredProcName указывает хранимую на сервере процедуру, которая должна быть выполнена.

Свойство Params позволяет задавать параметры хранимой процедуры. Вы можете редактировать их во время разработки, используя инспектор объектов C++Builder. Если необходимая информация может быть получена с сервера, инспектор объектов выводит список параметров, ожидаемых процедурой.

Вы можете задать параметры процедуры во время выполнения программы, присваивая значения свойству Params. Например, в следующем фрагменте присваивается значение параметру CustomerNumber хранимой процедуры, сопоставленной компоненту TStoredProc:

```
StoredProc1->ParamByName("CustomerNumber")->AsString = "123";
```

Обратите внимание, что доступ к возвращаемым значениям хранимой процедуры осуществляется также через свойство Params. Таким образом, если вы определили выходной параметр с именем Balance в свойстве Params, к нему можно обратиться после выполнения процедуры, используя ParamByName("Balance")->AsFloat.



Если вы намереваетесь использовать возвращаемые параметры хранимых процедур Sybase, которые возвращают также результат запроса, необходимо вызывать метод GetResults. Обычно компонент TStoredProc делает это автоматически, но сервер SQL Sybase не возвращает выходные параметры хранимой процедуры, пока не будут прочитаны все результаты запроса, поэтому вам нужно вызывать GetResults самостоятельно.

Основные задачи

Если хранимая процедура возвращает только одну строку, или не возвращает строк вообще, используйте метод ExecProc. В случае возврата нескольких строк, используйте вместо него метод Open.

Обратите внимание, что хранимая процедура должна быть подготовлена к выполнению. Во время работы программы это делается методом Prepare, а во время разработки — путем редактирования свойства Params.

Метод First перемещает курсор на начало набора результатов запроса; метод Last перемещает курсор в конец. Методы Next и Prior используются для перемещения на одну строку вперед или назад, соответственно. С помощью MoveBy можно переместиться вперед или назад на заданное число строк относительно текущей строки.

Методы Append и Insert используются для добавления пустых строк к результату запроса компонента TStoredProc. AppendRecord и InsertRecord вставляют в набор результатов строку, поля которой содержат заданные значения.

Метод Delete удаляет строку в текущей позиции курсора. Метод Edit позволяет изменять строку в текущей позиции, переводя набор результатов в режим редактирования. Метод Post сохраняет изменения в базе данных, тогда как Cancel отменяет их.

Кэшируемые изменения

Выше уже говорилось, что единственный способ изменения результатов запроса состоит в использовании кэширования и компонента TUpdateSQL. Как правило, вам достаточно установить свойства InsertSQL, DeleteSQL и ModifySQL компонента TUpdateSQL, а остальную работу этот компонент берет на себя. После вызова метода ApplyUpdates компонента TStoredProc выполняется релевантный SQL. Поскольку вы можете контролировать SQL, сгенерированный для обновления TDataSet, можно написать код, модифицирующий результаты запроса, которые в противном случае были бы доступны только для чтения.

В компоненте TUpdateSQL определены операторы SQL для вставки, изменения и удаления строк. Эти операторы SQL могут представлять собой составные запросы, обновляющие несколько таблиц или даже результаты вызовы других хранимых процедур. Для ссылок на объекты TUpdateSQL служит свойство UpdateObjects компонента TStoredProc.

TStoredProc имеет два метода, которые непосредственно относятся к кэшируемым изменениям: ApplyUpdates и CancelUpdates. ApplyUpdates сохраняет изменения, сделанные вами в базе данных. Фактически это сводится к выполнению релевантных SQL-операторов INSERT, UPDATE или DELETE в соответствующем компоненте UpdateSQL. Обратите внимание, что вы можете установить обработчик события OnUpdateError для реакции на ошибки, которые могут возникнуть при вызове ApplyUpdates.

Существуют также свойства, относящиеся к использованию кэшируемых изменений с хранимыми процедурами: CacheUpdates и UpdateRecordTypes. CacheUpdates включает и выключает поддержку кэширования изменений для компонента StoredProc. Для изменения результатов, возвращенных хранимой процедурой, необходимо разрешить кэширование, присоединить и настроить компонент UpdateSQL. UpdateRecordTypes определяет тип изменений, которые будут видимы в TDataSet после установки для CacheUpdates значения true.

Изменение TDataSet, изначально доступных только для чтения, будет рассмотрено далее в разделе, посвященном компоненту UpdateSQL.

TBatchMove

Модуль: `DBTables` Класс-предок: `TComponent`

Компонент TBatchMove позволяет вам переносить группы записей из одной таблицы в другую. Эти группы могут состоять как из нескольких записей, так и из всех записей TDataSet. Работая с TBatchMove, вы указываете исходную таблицу и таблицу назначения. Можно добавлять, изменять и стирать строки в таблице, или даже заменять ее целиком. Операция, выполняемая при вызове метода Execute, определяется установкой свойства Mode.

В таблицах 16.17 и 16.18 перечислены соответственно основные свойства и методы компонента TBatchMove. События для этого компонента не определены.

Таблица 16.17. Основные свойства TBatchMove

Свойство	Описание
Destination	Указывает таблицу-получатель в операции группового перемещения.
Mapping	Устанавливает взаимное соответствие между столбцами Source и Destination в том случае, когда исходная таблица и таблица назначения не идентичны. Если вы пренебрежете установкой этого свойства, групповое перемещение между таблицами разной структуры будет невозможно.
Mode	Определяет тип перемещения (например, batAppendUpdate или batCopy).
Source	Определяет таблицу-источник в операции группового перемещения.

Таблица 16.18. Основные методы TBatchMove

Метод	Функция
Execute	Инициирует начало групповой операции.

Ключевые элементы

В свойстве Source указывается TDataSet, из которого будет осуществляться копирование, а в свойстве Destination — TDataset назначения. В зависимости от выполняемой операции для свойства Mode устанавливается одно из следующих значений:

- batAppend — добавление строк к существующему TDataSet.
- batUpdate — замена строк в существующем TDataSet на соответствующие строки таблицы-источника. Этот режим требует указания индекса, определяющего положение изменяемых строк.
- batAppendUpdate — добавление новых строк к существующему TDataSet и замена уже существующих строк. Этот режим требует указания индекса, определяющего положение изменяемых строк.
- batCopy — копирование строк в TDataSet назначения. При этом создается новая таблица, содержащая скопированные строки, т.е. существующая таблица перезаписывается. Имейте в виду, что все объекты, связанные с прежней таблицей (индексы и присоединенные процедуры), также будут уничтожены. *Учтите, что эти вторичные объекты не восстанавливаются.*
- batDelete — удаляет те записи в таблице назначения, которые повторяют записи исходной таблицы. Этот режим требует указания индекса, определяющего положение удаляемых строк.

Основные задачи

Для выполнения копирования вызывается метод Execute. Предварительно свойство Mode должно быть установлено соответствующим образом. Если при копировании возникли ошибки, реакция TBatchMove будет зависеть от установок различных свойств:

- Если свойство AbortOnProblem имеет значение true, копирование будет прервано при возникновении любой ошибки.

- Если свойство AbortOnKeyViol имеет значение true, операция будет прекращена при нарушении ключа.
- Если свойство ProblemTableName имеет значение true, то в него будут помещены все строки, вызвавшие ошибки. Очевидно, что при AbortOnProblem, установленном в true, эта таблица будет состоять не более чем из одной строки.
- Если свойству KeyViolTableName присвоено определенное значение, то в указанную таким образом таблицу будут помещены все строки, вызвавшие нарушение ключа.
- Если свойству ChangeTableName присвоено определенное значение, TBatchMove переместит в указанную таким образом таблицу измененные строки таблицы назначения вместо их уничтожения.

TDataSource

Модуль: `DBTables` Класс-предок: `TComponent`

Компонент TDataSource осуществляет связь между интерфейсными компонентами и компонентами TDataSet (TTable, TQuery и TStoredProc). Именно он позволяет интерфейсным компонентам взаимодействовать с физическими объектами базы данных.

Интерфейсные компоненты обращаются к TDataSource при помощи своих свойств DataSource. Компонент TDataSource, в свою очередь, обращается к TDataSet за необходимыми данными. Данные от TDataSet передается через TDataSource к интерфейсному компоненту. После модификации данные передаются TDataSource, который передает их компоненту TDataSet.

Абстрагирование уровня управления данными от уровня TDataSet упрощает взаимодействие между TDataSet и интерфейсными компонентами. Вы можете, например, сменить TDataSet для нескольких компонентов, не меняя их по отдельности. Иными словами, если вы хотите, чтобы интерфейсные компоненты формы обращались к другому TDataSet, вам не нужно изменять сами эти компоненты. Вместо этого достаточно изменить свойство TDataSet того компонента TDataSource, к которому они обращаются. Такая трехуровневая схема позволяет лучше организовать доступ группы интерфейсных компонентов к данному TDataSet.

В таблицах 16.19, 16.20 и 16.21 перечислены соответственно основные свойства, методы и события компонента TDataSource.

Таблица 16.19. Основные свойства TDataSource

Свойство	Описание
AutoEdit	Определяет, будет ли изменение данных в интерфейсном компоненте приводить к автоматическому переходу в режим Edit.
DataSet	Указывает TDataSet, связанный с данным TDataSource.
Enabled	Показывает, отображаются ли данные в интерфейсных компонентах, связанных с TDataSource.
State	Возвращает состояние компонента TDataSet.

Таблица 16.20. Основные методы TDataSource

Метод	Функция
Edit	Переключает TDataSet, связанный с данным TDataSource, в режим редактирования.

Таблица 16.21. Основные события TDataSource

Событие	Условие возникновения
OnDataChange	При изменении данных или перемещении указателя записи.
OnStateChange	При изменении свойства State.
OnUpdateData	При вызове Post или UpdateRecord.

Ключевые элементы

Свойство DataSet идентифицирует TTable, TQuery или TStoredProc, обеспечивающие компонент данными. Свойство AutoEdit определяет, будет ли компонент TDataSet автоматически переключаться в режим Edit при изменении содержимого интерфейсных компонентов, разрешая тем самым изменение данных в базе.

Основные задачи

Вы можете контролировать изменения данных в TDataSet и связанных с ним интерфейсных компонентах, установив обработчик для события OnDataChange.

Событие OnStateChange возникает при изменении состояния TDataSet — например, если вы переключили его из режима dsBrowse в режим dsEdit, вызвав метод Edit. Поскольку OnStateChange может происходить и для пустого TDataSet, проверьте TDataSet на наличие данных перед тем, как обращаться к нему.

Если вы хотите изменить данные в текущей строке до ее сохранения, установите обработчик события OnUpdateData. Это событие происходит при вызове Post или UpdateRecord.

TUpdateSQL

Модуль: `DBTables` Класс-предок: `TDataSetUpdateObject`

Компонент TUpdateSQL дает вам возможность управлять способом изменения данных в TDataSet. Имея полный контроль над этим процессом, вы можете изменять даже те TDataSet, которые предназначены только для чтения. Все это делается с помощью свойств InsertSQL, DeleteSQL и ModifySQL, которые позволяют указать SQL, выполняемые для вставки, удаления и изменения строки.

В таблицах 16.22 и 16.23 перечислены соответственно основные свойства и методы компонента TUpdateSQL. События для этого компонента не определены.

Таблица 16.22. Основные свойства TUpdateSQL

Свойство	Описание
DeleteSQL	Определяет SQL, выполняемый при удалении строки.
InsertSQL	Определяет SQL, выполняемый при добавлении строки.
ModifySQL	Определяет SQL, выполняемый при изменении строки.

Таблица 16.23. Основные методы TUpdateSQL

Метод	Функция
Apply	Подставляет параметры и вызывает указанный метод (DeleteSQL, InsertSQL или ModifySQL). Вызывает методы SetParams и ExecSQL.
ExecSQL	Выполняет DeleteSQL, InsertSQL или ModifySQL.
SetParams	Подставляет параметры в DeleteSQL, Insert SQL или ModifySQL указанным способом.

Ключевые элементы

Свойства DeleteSQL, InsertSQL или ModifySQL предоставляют средства для изменения компонентов TDataSet, которые могут быть набором результатов TTable, TQuery или TStoredProc. Указанный вами SQL может состоять из простого SQL-оператора INSERT, UPDATE или DELETE, или представлять собой комплексный SQL-запрос, включающий даже вызов хранимой процедуры. Такая гибкость позволяет изменять наборы результатов практически любого типа.

Основные задачи

Для использования компонента TUpdateSQL необходимо выполнить следующее:

1. Поместить компонент TUpdateSQL в форму и задать операторы InsertSQL, UpdateSQL и ModifySQL для изменения объекта (или объектов) базы данных, на которые ссылается TDataSet.
2. Установить свойство UpdateObject компонента TDataSet для указания на компонент TUpdateSQL.
3. Установить для свойства CachedUpdates компонента TDataSet значение true.
4. Вызвать команду ApplyUpdates в том месте приложения, где вы хотите выполнить SQL, указанный в компоненте TUpdateSQL.

Изменения, выполняемые через DeleteSQL и ModifySQL, требуют, разумеется, уточнения с помощью SQL-оператора WHERE. Оба эти свойства поддерживают специальное расширение SQL, позволяющее вам обращаться к исходному значению поля. Для этого перед именем поля должен быть указан префикс Old_. Это похоже на использование контекстной переменной Old в СУБД InterBase. Например, свойство ModifySQL, определенное для таблицы CUSTOMER, может выглядеть следующим образом:

```
UPDATE CUSTOMER SET Name=:Name
WHERE CustomerNo=:Old_CustomerNo
```

Хотя этот запрос не приводит к реальному изменению CustomerNo, неплохо взять за правило всегда использовать префикс Old_, поскольку некоторые изменения без него невозможны.



Для редактирования SQL, связанных с компонентом TUpdateSQL, вы можете использовать полноценный редактор кода C++Builder. Для этого нужно дважды щелкнуть на кнопке Code Editor, находясь в редакторе свойств InsertSQL, DeleteSQL и ModifySQL. Редактор кода C++Builder гораздо удобнее, чем компонент TMemo, используемый по умолчанию для редактирования этих трех свойств.

Событие OnUpdateRecord используется для дополнительной обработки строк перед их передачей компоненту TUpdateSQL. После завершения дополнительной обработки можно вызвать метод Apply компонента TUpdateSQL для подстановки параметров, встроенных в ваш SQL, и выполнения основных операций над базой данных. Вот пример обработчика OnUpdateRecord:

```
void __fastcall TForm1::Table1UpdateError(TDataSet *TDataSet,
EDatabaseError *E, TUpdateKind, TUpdateAction &UpdateAction)
{
#define DEFAULTRATE 5.00
  if (UpdateKind == ukInsert)
    StoredProc1->FieldByName("Rate")->Value = DEFAULTRATE;
  UpdateSQL1->Apply(UpdateKind);
  UpdateAction = uaApplied;
}
```

Обратите внимание на установку параметра UpdateAction, который сообщает процедуре ApplyUpdates о том, что никакие дальнейшие действия не требуются.

TField

Модуль: DBTables Класс-предок: TComponent

Компонент TField используется для доступа к столбцам в строках TDataSet. Все, что C++Builder позволяет вам делать на уровне полей, реализуется компонентом TField. С его помощью вы можете включать и выключать отображение отдельных полей, проверять возможность изменения значений и правильность ввода.

В таблицах 16.24, 16.25 и 16.26 перечислены соответственно основные свойства, методы и события компонента TField.

Таблица 16.24. Основные свойства TField

Свойство	Описание
Calculated	Показывает, является ли поле вычисляемым.
DataSet	Возвращает TDataSet, которому принадлежит TField.
EditMask	Определяет входную маску для ограничения текста, вводимого в интерфейсном компоненте.
FieldName	Указывает имя поля базы данных, связанного с TField.
Lookup	Показывает, является ли поле поисковым.
Value	Возвращает значение, содержащееся в поле, как вариантную переменную.
Visible	Определяет, отображается ли TField (по умолчанию) в интерфейсном компоненте DBGrid.

Таблица 16.25. Основные методы TField

Метод	Функция
Assign	Копирует содержимое одного поля в другое.
AssignValue	Присваивает TField символьное значение.
Clear	Очищает TField.
GetData	Возвращает данные из поля в исходном формате.
SetData	Присваивает полю значение в исходном формате.

Таблица 16.26. Основные события TField

Событие	Условие возникновения
OnChange	При любом изменении TField.
OnValidate	При изменении значения поля.

Ключевые элементы

Если вы специально не создаете набор компонентов TField, используя редактор полей (Field Editor) C++Builder, они автоматически создаются при каждом открытии TDataSet. Созданные таким образом компоненты TField соответствуют столбцам таблицы TDataSet.

Создание своих собственных компонентов TField с помощью редактора полей гарантирует, что ваше приложение будет обращаться именно к тем столбцам, к которым оно должно обращаться. В противном случае, смена таблицы базы данных автоматически изменяет столбцы, с которыми работает приложение. Использование компонентов TField делает приложение нечувствительным к изменению порядка столбцов и приводит к генерации исключения при изменении имени столбца или типа данных. Единственный случай, когда вам не следует использовать TField — создание программы для просмотра различных типов таблиц. Во всех остальных случаях желательно, чтобы для работы с TDataSet использовались компоненты TField.

Когда вы определяете свой собственный набор TField, обращение к полю, которое было переименовано или удалено из таблицы, вызовет генерацию исключения. Таким образом, приложение получает возможность работать с неверными данными.

Сам компонент TField в действительности никогда не создается в приложениях C++Builder — он является абстрактным классом и, следовательно, его члены должны быть замещены перед созданием конкретного экземпляра. Чтобы создать такой экземпляр, вы должны создать объект-потомок TField и заполнить некоторые из его абстрактных промежутков. На языке ООП это называется *наследованием (inheritance)*. Класс TField может также быть использован для приведения типов и манипулирования его потомками. В ООП для этого используется термин *полиморфизм (polymorphism)*. Классы-потомки TField перечислены в таблице 16.27.

Таблица 16.27. Потомки TField

Потомок TField	Тип хранимых данных
TStringField	Текстовые данные с фиксированной длиной до 255 символов.
TIntegerField	Целые числа от -2 147 483 648 до 2 147 483 647.
TSmallintField	Целые числа от -32 768 до 32 767.
TWordField	Целые числа от 0 до 65 535.
TFloatField	Действительные числа от $5,0*10^{-324}$ до $1,7*10^{308}$.
TCurrencyField	Денежные значения с точностью от 15 до 16 цифр, представленные двоичными числами в диапазоне от $5,0*10^{-324}$ до $1,7*10^{308}$.
TBCDField	Двоично-кодированные десятичные числа с точностью до 18 цифр.
TBooleanField	Логические величины.
TDateTimeField	Значения даты и времени.
TDateField	Значения даты.
TTimeField	Значения времени.
TBlobField	Поле переменной длины без ограничения размера.
TBytesField	Поле переменной длины без ограничения размера.
TVarBytesField	Поле переменной длины до 65 535 символов.
TMemoField	Текстовое поле переменной длины без ограничения размера.
TGraphicField	Графическое поле переменной длины без ограничения размера.

Потомки TField никогда специально не помещаются в форму. Как уже говорилось, они создаются для вас либо через редактор полей, либо автоматически при открытии TDataSet.

Основные задачи

TField поддерживает некоторые установки, относящиеся к столбцам, которые вы можете использовать для настройки приложения. Это делается либо через редактор полей, либо через TDataSet. Для доступа к редактору полей щелкните правой кнопкой мыши на компоненте TDataSet и выберите в контекстном меню пункт Fields Editor.

Например, для предохранения поля от изменения, откройте редактор полей и установите в инспекторе объектов для свойства ReadOnly данного поля значение true. Чтобы сделать поле невидимым в DBGrid, установите для свойства Visible значение false. Если вы хотите контролировать допустимые типы символов, вводимых в поле, укажите маску в свойстве EditMask. Для изменения поля базы данных, с которым связан компонент TField, измените значение свойства FieldName на имя того поля, к которому вы хотите обращаться.



Обратите внимание, что свойство Visible компонента TField управляет отображением поля в компоненте DBGrid только в том случае, если в свойстве Columns этого компонента не указаны никакие столбцы.



Вы можете перенести TField из редактора полей прямо в форму. Это приведет к созданию в форме интерфейсного компонента с соответствующим заголовком. Вы можете задать тип компонента, помещаемого в форму, используя свойство TControlClass в Database Explorer.

TField поддерживает также неявные преобразования данных. Иными словами, вам не нужно знать, данные какого типа в действительности хранятся в TField, чтобы преобразовать их к другому типу. Это возможно благодаря свойству Value компонента TField. Value имеет вариантный тип, который неявно преобразуется при выполнении присваивания. Например, вы можете присвоить строковое значение булеву полю, используя его свойство Value:

```
TBooleanField MyBoolean;
...
MyBoolean->Value = "T";
```

Вы можете также присвоить числовое поле строковому элементу редактирования, используя его свойство Value:

```
TCurrencyField MyMoney;
...
Edit1->Text = MyMoney->Value;
```

Потрясающая способность TField выполнять неявные преобразования типов делает его похожим на хамелеона, когда речь идет о работе с различными данными. Это упрощает приложения и уменьшает количество работы по их адаптации к различным структурам данных.

Заключение

Как вы могли видеть, иерархия классов C++Builder, относящихся к базам данных, довольно богата, однако легка для использования. Большое внимание было уделено тому, чтобы сделать ее не только обширной, но и логически последовательной. В классах VCL, обеспечивающих доступ к базам данных, достигнут хороший баланс между функциональностью и легкостью использования.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Какое свойство компонента TQuery определяет, будут ли результаты запроса изменяемыми?

Свойство RequestLive.

Вопрос: Какое свойство компонента TDatabase используется для указания псевдонима, используемого в приложении?

Свойство DatabaseName.

Какой минимальный набор компонентов вкладки Data Access должен быть помещен в приложение C++Builder, требующее включения интерфейсных компонентов?

Компоненты TTable и TDataSource.

Вопросы

1. Для чего предназначен компонент TSession?
2. Какие преимущества дает использование компонента TDatabase в ваших приложениях?
3. Какие условия должны быть выполнены для того, чтобы результаты запроса TQuery стали изменяемыми?
4. Какой компонент используется для копирования строк из одного TDataSet в другой?

Упражнения

1. Создайте простую форму, используя компоненты TTable, TDataSource и интерфейсные компоненты из вкладки Data Controls палитры компонентов.
2. Измените форму из упражнения 1, включив в нее компонент TDatabase.
3. Создайте новую таблицу, скопировав строки из уже существующей с помощью компонента TBatchMove.
4. Настройте компонент TQuery для считывания строк из таблицы BIOLIFE в демонстрационной базе данных BCDEMOS.
5. Сконфигурируйте компонент TQuery из предыдущего упражнения так, чтобы результаты запроса были изменяемыми.

Создание форм для работы с базами данных



Кен Хендерсон

Вчера вы кратко познакомились с архитектурой доступа к базам данных в C++Builder. Сегодня я покажу вам, как использовать компоненты C++Builder для создания форм, работающих с базами данных. Мы применим в реальных приложениях те знания, которые вы получили в шестнадцатый день, «Архитектура доступа к базам данных в C++Builder».

Сегодня вы узнаете о следующем:

- Как создавать формы для работы с базами данных, используя мастер форм баз данных (Database Form Wizard)
- Каковы существенные свойства компонентов TTable, TDataSource и TDBEdit
- Как создавать формы для работы с базами данных, используя традиционную технику разработки форм
- Почему интерфейсные компоненты должны обращаться к компонентам TDataSet косвенно — через компонент TDataSource

Простая форма

Давайте начнем сегодняшний урок с построения простой формы для работы с базой данных. Пройдя на практике через процесс создания такой формы, вы увидите, как архитектура доступа к базам данных проявляет себя в реальных приложениях.

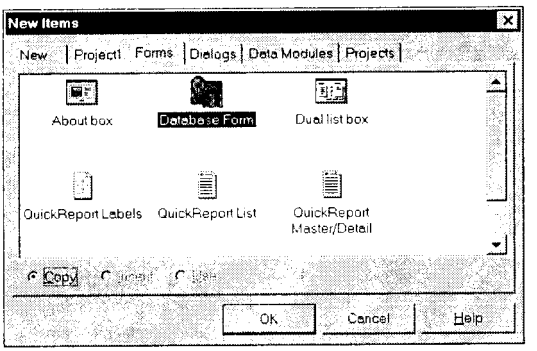
Существуют два основных способа построения формы для работы с базами данных. Вы можете сделать это вручную, по очереди помещая компоненты в новую форму и связывая их друг с другом, или можете предоставить это дело мастеру форм баз данных. Давайте начнем с использования мастера. Запустите C++Builder, если вы еще не сделали это, и откройте новый проект.

Мастер форм баз данных

Мастер форм предназначен для конструирования формы на основе ваших ответов на простые вопросы. Выберите в меню File пункт New и щелкните

на вкладке Forms в диалоговом окне New Items. Затем дважды щелкните на значке Database Form. На рис. 17.1 показано диалоговое окно New Items.

Рис. 17.1
Вы можете вызвать мастер форм баз данных через диалоговое окно New Items

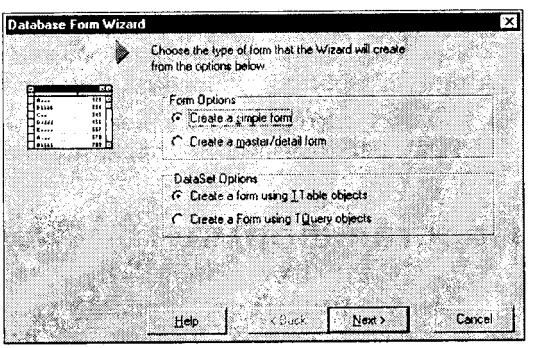




Вы можете также запустить мастер форм баз данных из меню C++Builder. Для этого просто выберите в главном меню пункт Database | Form Wizard.

В первом диалоговом окне мастера форм запрашивается, хотите ли вы строить простую форму или форму типа главный/подчиненный и на каких компонентах должна быть основана новая форма — TTable или TQuery. Оставьте пока установки по умолчанию и нажмите кнопку Next. Рис.17.2 иллюстрирует первый шаг процесса создания формы с помощью мастера форм баз данных.

Рис. 17.2
Выберите тип и основу создаваемой формы



Затем мастер спросит, какую таблицу вы хотели бы использовать в форме. Начните с выбора псевдонима BCDEMOS в выпадающем списке Drive or Alias name (Имя диска или псевдонима). Тем самым вы укажете сервер базы данных или место на диске, где расположены таблицы, к которым предполагается доступ. После выбора BCDEMOS, в списке имен таблиц будут перечислены все таблицы, расположенные в том каталоге, на который ссылается псевдоним BCDEMOS. Дважды щелкните на таблице BIOLIFE в верхней части списка. Второе диалоговое окно мастера форм показано на рис. 17.3.

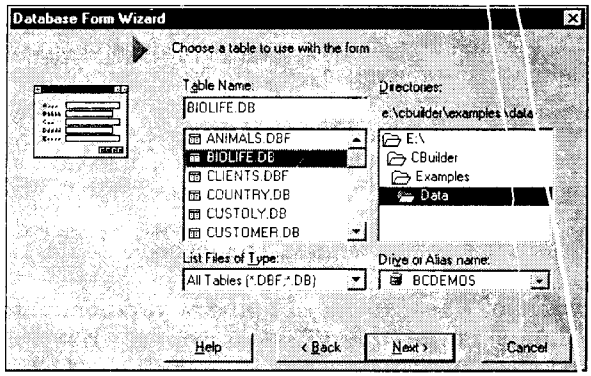


Рис. 17.3
Выберите таблицу, которая будет использоваться в новой форме

Затем вы увидите список полей таблицы BIOLIFE. Вы можете выбирать их по одному с помощью кнопки >, или все сразу, используя кнопку >>. Щелкните на кнопке >>, затем щелкните на кнопке Next для продолжения. Процесс выбора полей показан на рис. 17.4.

В следующем окне мастер попросит выбрать ориентацию полей в новой форме. Вам предлагаются три варианта: Horizontal (Горизонтально), Vertical (Вертикально) и Grid (Сетка). Выберите тот, который установлен по умолчанию (Horizontal), и нажмите кнопку Next (см. рис. 17.5).

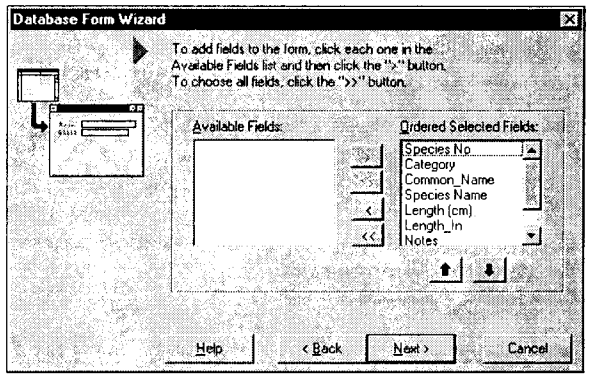


Рис. 17.4
Выбор полей в мастере форм

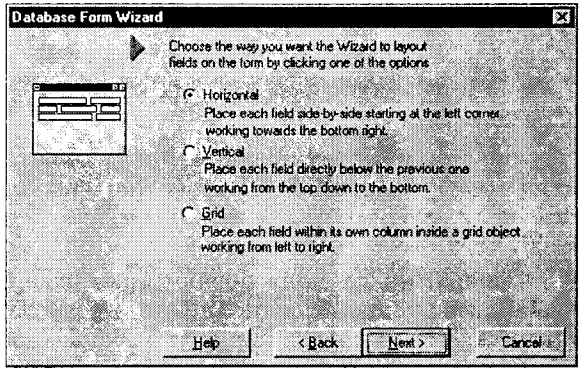
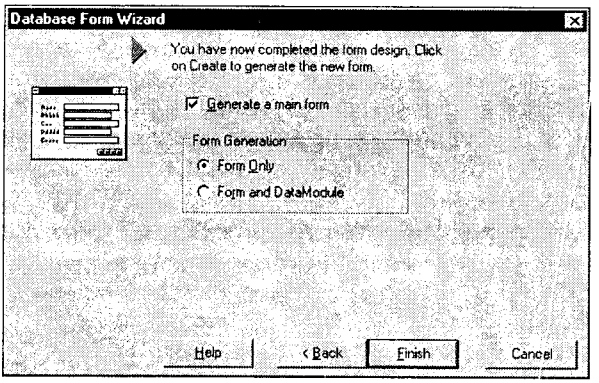


Рис. 17.5
Используйте это диалоговое окно для выбора ориентации новой формы

В следующем — и последнем — окне мастер форм спрашивает, хотите ли вы сделать форму главной формой приложения. Как вы помните, главная форма приложения — это та форма, которая отображается на экране сразу после запуска. Обратите внимание на флажок в верхней части окна. Сейчас мы примем установку по умолчанию и разрешим новой форме стать главной формой приложения. Очевидно, что добавляя формы к приложению, вы не захотите делать каждую из них главной. Чтобы избежать этого, просто сбросте флажок.

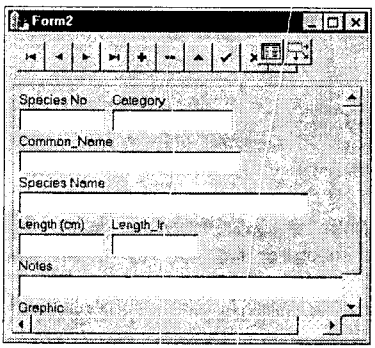
В этом последнем диалоговом окне также спрашивается, хотите ли вы создать новую форму вместе с соответствующим модулем данных или отдельно. Давайте снова примем установку по умолчанию и оставим выбранной опцию Form Only (Только форма). На рис. 17.6 показан завершающий шаг процесса создания формы.

Рис. 17.6
Вы можете указать, создавать ли главную форму отдельно или включить модуль данных



Теперь, когда вы ответили на все запросы мастера форм баз данных, щелкните на кнопке Finish. Вы должны увидеть новую форму, загруженную в редактор форм C++Builder. Форма имеет свои собственные компоненты TTable и TDataSource, а также набор визуальных интерфейсных компонентов. На рис. 17.7 показано, как должна выглядеть новая форма.

Рис. 17.7
Ваша новая форма, загруженная в редактор форм



Теперь, когда ваша новая форма завершена, давайте посмотрим, как она выглядит во время выполнения программы. Нажмите F9 для построения проекта. Поскольку новая форма создавалась как главная форма приложения, вы должны увидеть ее сразу после запуска. На рис. 17.8 показано, как это должно выглядеть.

Пока приложение на экране, вы можете перемещаться по таблице BIOLIFE, используя элемент управления DBNavigator в верхней части формы. С его помощью вы можете добавлять, изменять и стирать строки в таблице. Щелкните на кнопке + в панели DBNavigator, чтобы добавить запись; щелкните на кнопке –, чтобы ее стереть. Введите что-нибудь в одно из полей, начав тем самым его редактирование. Щелкните на кнопке Post (кнопка с галочкой) для сохранения сделанных изменений. При минимальной затрате усилий, вы только что создали полностью работоспособную форму для ввода данных в таблицу BIOLIFE.

Окончив просмотр новой формы, закройте ее, чтобы вернуться в C++Builder.

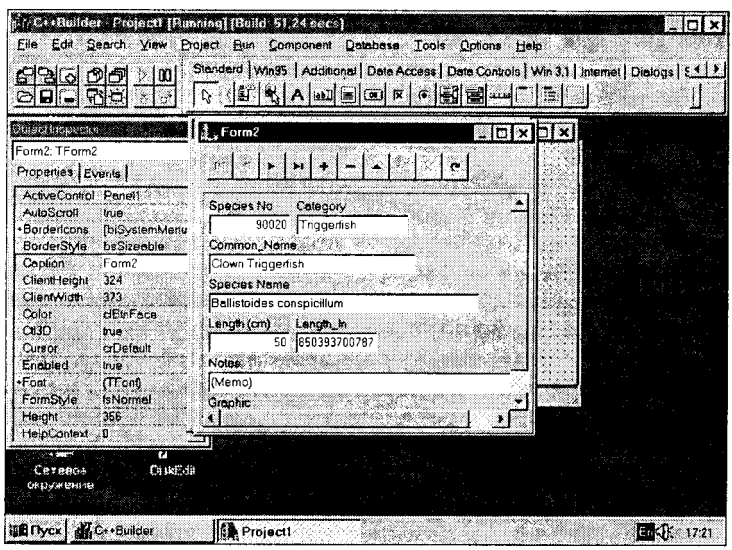


Рис. 17.8
Вид построенной вами формы после запуска приложения



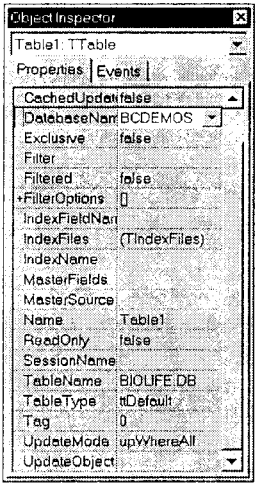
Имейте в виду, что вы можете изменять формы, сгенерированные мастером форм баз данных. Они являются формами C++Builder в полном смысле этого слова. В дальнейшем вы можете обнаружить, что наибольший смысл имеет создание форм с помощью мастера и последующая настройка их под свои конкретные задачи.

Изучение созданной формы

Перед тем, как двинуться дальше, давайте рассмотрим только что созданную форму, чтобы понять, как она работает. Щелкните на компоненте TTable (он должен находиться вблизи верхнего края формы), чтобы выделить его. Теперь нажмите F11 для перехода в окно инспектора объектов. Здесь нужно отметить два ключевых свойства. Во-первых, обратите внимание на свойство DatabaseName. Оно содержит значение BCDEMOS — тот псевдоним BDE, который вы выбрали в мастере форм. Это свойство предоставляет BDE информацию, необходимую для поиска таблицы, указанной во втором ключевом свойстве, TableName. Свойство TableName содержит имя таблицы

BIOLIFE, выбранной в мастере форм. На рис. 17.9 показано, что вы должны увидеть в инспекторе объектов.

Рис. 17.9
Компонент TTable при ближайшем рассмотрении

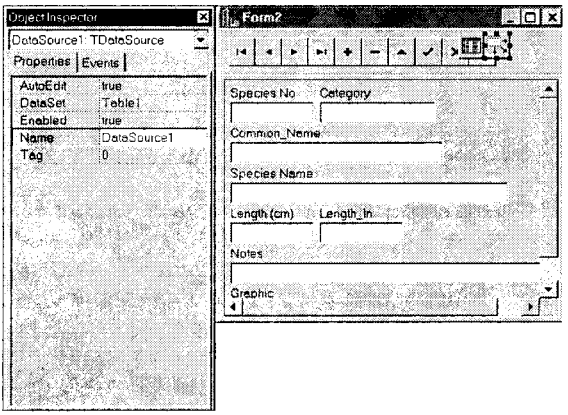


Исследовав компонент TTable, давайте таким же образом рассмотрим компонент TDataSource. Он должен находиться в форме рядом с TTable. Выделите его, щелкнув на нем мышью.

Как и компонент TTable, TDataSource имеет два ключевых свойства, на которые вам следует обратить внимание. Первое из них — это свойство TDataSet, которое ссылается на компонент TTable нашей формы, Table1. Это означает, что при обмене данными с интерфейсными компонентами TDataSource будет посылать или принимать эти данные от таблицы, связанной с Table1.

Во-вторых, обратите внимание на свойство AutoEdit. Для него установлено значение true (которое является установкой по умолчанию). Как говорилось вчера, установка этого свойства приводит к автоматическому переключению TDataSource в режим редактирования при изменении данных в соответствующем интерфейсном компоненте. На рис. 17.10 показаны свойства компонента TDataSource в инспекторе объектов C++Builder.

Рис. 17.10
Ваш компонент TDataSource «под микроскопом»



Последние компоненты формы, которые требуют дополнительного изучения — это интерфейсные компоненты. Они отвечают за визуальный ввод и модификацию данных. Щелкните на самом верхнем компоненте TDBEdit, чтобы выделить его; затем выделите вместе с ним пару других компонентов TDBEdit. После этого нажмите клавишу F11, чтобы вызвать инспектор объектов C++Builder.



Выделив один компонент, вы можете добавить к нему другие, щелкая на них левой кнопкой мыши при нажатой клавише Shift.

Как и для компонентов TTable и TDataSource, мы остановимся на двух ключевых свойствах TDBEdit. Во-первых, обратите внимание на свойство DataSource. Оно содержит имя компонента формы TDataSource. Это означает, что данные, которые вы вводите в визуальные компоненты формы, передаются соответствующей таблице через компонент DataSource1. Во-вторых, посмотрите на свойство DataField. Оно указывает, *какое* поле в таблице обслуживает данный интерфейсный компонент.

Теперь, когда мы подробнее рассмотрели компоненты TTable, TDataSource и TDBEdit, взаимосвязь между ними должна стать более очевидной. Компоненты TTable обеспечивают базовую связь между формой и таблицей базы данных. Функцией компонентов TDataSource является передача данных между TTable и соответствующими интерфейсными компонентами. Интерфейсные компоненты, такие как TDBEdit, посылают и принимают данные от компонентов TDataSource. Таким образом, в приложениях C++Builder реализуется гибкий трехступенчатый доступ к базам данных.

Вы можете удивиться, зачем вообще нужен компонент TDataSource. Рассмотрим следующую ситуацию: пусть компоненты типа TDBEdit могут непосредственно обращаться к компонентам TDataSet, и вам требуется, чтобы все интерфейсные компоненты формы обращались к другому TDataSet. В этом случае вам придется выделить каждый из них и изменить соответствующим образом значение свойства DataSet. А к каким последствия приведет замена TDataSet, если интерфейсные компоненты связаны только с TDataSource? Вы просто измените значение свойства DataSet компонента TDataSource. Это сделать гораздо проще.

Вы можете возразить, что достаточно было бы выделить сразу все интерфейсные компоненты и изменить их свойства DataSet с такой же легкостью. Но как быть во время выполнения программы? Вам придется изменять каждый компонент в отдельности. Можно, конечно, написать для этого цикл, но в целом ситуация не изменится. Вероятно, теперь вы смогли увидеть некоторую логику в архитектуре доступа к данным C++Builder.

Создание простой формы сложным способом

Теперь, когда вы создали базовую форму простым способом, давайте сделаем то же самое вручную, чтобы увидеть детали этого процесса. Нажмите кнопку New form в панели инструментов C++Builder для начала создания новой формы.

Поместите в форму два компонента TPanel. Один из них разместите рядом с верхним краем формы и установите для его свойства Align значение alTop. В результате панель займет самую верхнюю часть формы. При этом

неважно, какие размеры имеет форма — панель все равно будет занимать ее верхнюю область. Второй компонент TPanel поместите у нижнего края формы и установите для его свойства Align значение alClient. Панель займет всю оставшуюся часть формы. При этом размер формы опять-таки не играет роли — панель будет занимать всю площадь, не занятую первым компонентом TPanel.

Затем давайте удалим текст, который отображается в каждой панели по умолчанию. Нам не нужны никакие подписи, поэтому свойства Caption необходимо сбросить. Выделите обе панели и нажмите F11 для переключения в окно инспектора объектов. Сотрите содержимое общего свойства Caption. На рис. 17.11 показана форма с выделенными панелями и пустым свойством Caption.

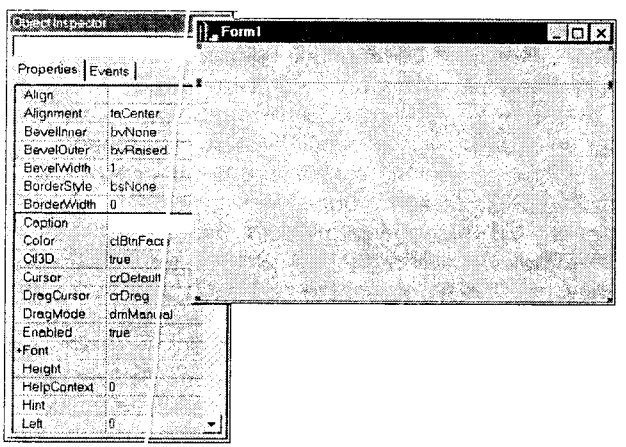


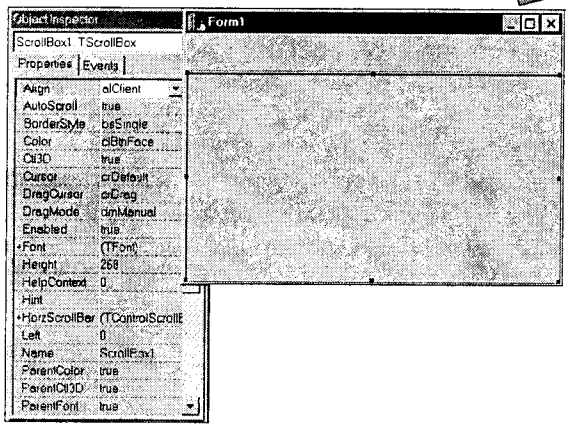
Рис. 17.11
В окне инспектора объектов очистите свойство Caption обоих компонентов TPanel

Следующее, что нужно сделать — поместить на нижнюю панель компонент TScrollBox, который позволит отображать больше интерфейсных компонентов, чем вмещает форма. Если таблица имеет много полей, соответствующие интерфейсные компоненты могут не поместиться в окне. TScrollBox обеспечивает доступ к ним путем прокрутки содержимого окна во время выполнения программы. Щелкните на нижней панели для ее выделения; затем поместите на панель элемент управления TScrollBox. Установите для его свойства Align значение alClient. В результате TScrollBox займет все пространство базовой панели, свободное от других компонентов. Поскольку в данном случае TScrollBox является единственным компонентом на панели, он будет занимать всю ее рабочую область. На рис. 17.2 показано, как должна выглядеть новая форма с компонентом TScrollBox.

Теперь, когда основа формы подготовлена, можно приступить к размещению компонентов, обеспечивающих доступ к базе данных. Давайте начнем с двух невизуальных компонентов: TTable и TDataSource. Поместите каждый из них в произвольном месте формы. В данном случае положение не играет роли, поскольку оба эти компонента не будут видимы во время выполнения программы. Я советую, однако, поместить их в верхней части формы, чтобы обеспечить к ним легкий доступ независимо от того, какой размер будет иметь форма.

Установите для свойства DatabaseName компонента TTable псевдоним BCDEMOS. Свойству TableName присвойте имя таблицы BIOLIFE.DB, расположение которой определяется псевдонимом BCDEMOS.

Рис. 17.12
Ваша новая форма с помещенным в нее TScrollBox



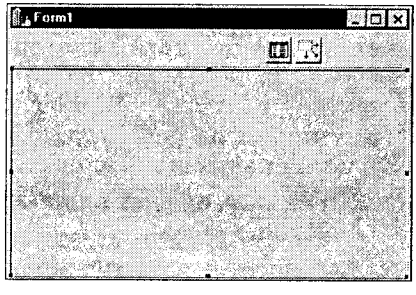
Затем укажите в свойстве DataSet компонента TDataSource имя компонента TTable, помещенного в форму (Table1 по умолчанию). Это устанавливает связь между TTable и интерфейсными компонентами, которые обслуживает TDataSource.



Для быстрой установки свойства DataSet компонента TDataSource вы можете дважды щелкнуть на нем. Если в форме находится только один потомок класса TDataSet (в данном случае TTable), C++Builder автоматически подставит его имя в качестве значения свойства DataSet.

На рис. 17.13 показаны размещенные компоненты TTable и TDataSource.

Рис. 17.13
Ваша новая форма с невизуальными компонентами



Теперь, когда невизуальные компоненты размещены, можно обратиться к интерфейсным компонентам, которые обеспечивают взаимодействие пользователя с таблицами базы данных. Они похожи на те элементы управления, которые вы обычно видите в приложениях Windows, но разработаны специально для обмена информацией с базами данных. Например, вы часто встречаетесь с элементом управления TEdit, используемым в приложениях для редактирования строк того или иного типа. В C++Builder включена версия этого элемента для работы с базами данных, называемая TDBEdit. TDBEdit работает точно так же, как и TEdit, но предназначен только для обслуживания конкретного столбца в таблице базы данных. Он отображает данные, содержащиеся в столбце, и передает обратно все изменения, вносимые пользователем.

Первым интерфейсным компонентом, который вы поместите в форму, будет TDBNavigator. Он находится во вкладке Data Controls палитры компонентов. Выделите верхнюю панель формы и поместите на нее TDBNavigator. Укажите в его свойстве DataSource имя компонента TDataSource, находящегося в форме (см. рис. 17.14).

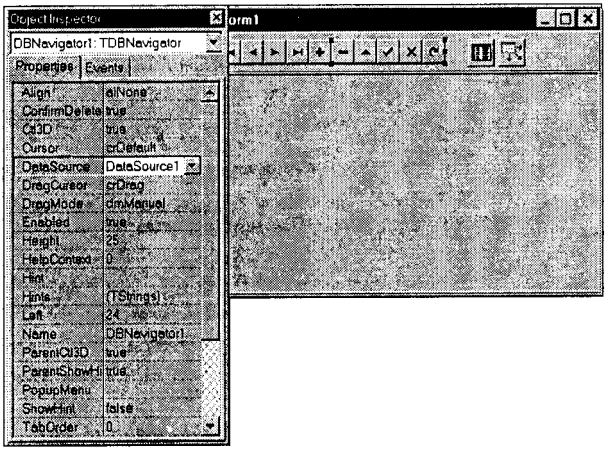


Рис. 17.14
Форма с размещенным компонентом TDBNavigator

После настройки TDBNavigator можно начать размещение других интерфейсных компонентов. Для этого существуют два основных способа. Первый является наиболее утомительным. Он заключается в последовательном помещении компонентов (TDBEdit, TDBMemo и других) в форму и установке свойств DataSource и DataField для каждого из них в отдельности. После того, как все компоненты окажутся на своих местах, вам нужно будет разместить подписи к ним (компоненты TLabel). Однако все это можно сделать намного проще, хотя некоторой работы вам все равно не избежать.

Второй, и намного более быстрый способ, подразумевает использование редактора полей C++Builder. Вы сможете переместить компоненты полей в форму прямо из редактора. C++Builder создаст соответствующие интерфейсные компоненты и подписи к ним. Для этого выполните следующие действия:

1. Щелкните правой кнопкой мыши на компоненте TTable для вызова контекстного меню.
2. Выберите в контекстном меню пункт Fields Editor.
3. Щелкните правой кнопкой на редакторе полей и выберите пункт Add fields (Добавить поля).
4. После отображения списка полей нажмите кнопку OK, чтобы добавить все поля.
5. Не снимая выделения полей в редакторе, переместите их в форму как группу. После этого вы должны увидеть для каждого поля соответствующий интерфейсный компонент с компонентом TLabel.

На рис. 17.15 показано, как должна выглядеть форма в результате этих действий.

На этом ваша форма в основном завершена. Вы только что сделали вручную то, что ранее автоматически делал мастер форм. Вероятно, это поможет вам понять, сколько времени сохраняет использование мастера при построении новых форм.

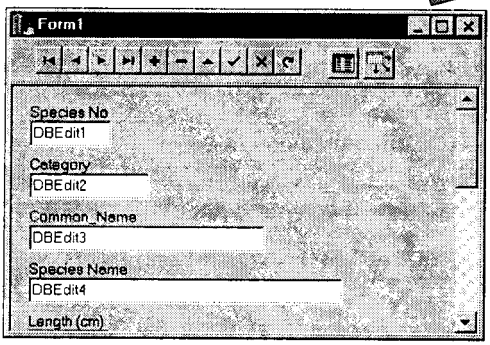


Рис. 17.15
Ваша новая форма после размещения интерфейсных компонентов

Единственное, что осталось сделать с новой формой — это решить, как и когда открывать TTable. Для этого существуют два простых способа. Во-первых, вы можете включить свойство Active компонента TTable на этапе разработки. Установка для этого свойства значения true приведет к открытию TTable уже во время работы над ним, и это будет сделано точно так же, как и при выполнении программы. Во-вторых, вы можете установить обработчик события формы OnCreate или OnShow, который будет открывать TTable во время работы приложения. Фактически, именно так поступает мастер форм — для открытия таблицы во время выполнения программы он генерирует обработчик события OnCreate.

Давайте сымитируем работу мастера форм и напишем обработчик события OnCreate, который будет открывать таблицу при создании формы. Нажмите F11 для переключения в окно инспектора объектов и выберите форму в выпадающем списке селектора компонентов. Затем щелкните на вкладке Events и дважды щелкните на событии OnCreate. Введите следующий код в окне редактора кода:

```
Table1->Open();
```

Это приведет к открытию таблицы при создании формы. По умолчанию, формы приложения автоматически создаются после его запуска, хотя вы можете изменить этот порядок работы.

Запуск новой формы

Сейчас ваша форма завершена и готова к запуску. Чтобы эта форма отображалась первой после запуска приложения, нужно определить ее как главную форму приложения. Для этого выполните следующие действия:

1. Выберите в главном меню C++Builder пункт Options | Project.
2. Во вкладке Forms диалогового окна Options вызовите выпадающий список Main form и выберите в нем свою форму. Тем самым она будет сделана главной формой приложения.
3. Нажмите кнопку OK для сохранения сделанных изменений.

Теперь, когда ваша новая форма является главной формой приложения, нажмите F9. После компиляции и запуска программы вы увидите новую форму, которая должна выглядеть как на рис. 17.16.

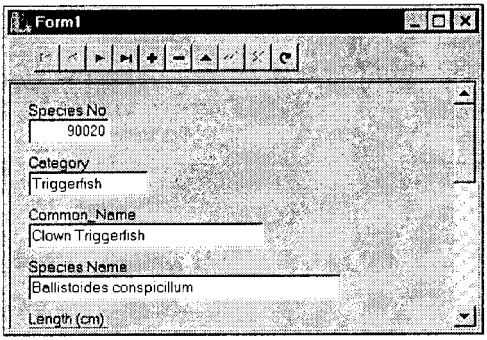


Рис. 17.16
Вид вашей новой формы во время выполнения программы

Когда вы закончите просматривать форму, закройте ее, чтобы вернуться в C++Builder.

Заключение

В этой главе вы научились строить формы для работы с базами данных, используя как мастер форм, так и традиционную технику конструирования форм. Вы увидели, что для этого существуют два пути — простой и сложный. Более важно то, что при этом вы узнали, почему определенные элементы архитектуры доступа к базам данных сконструированы именно так, а не иначе. Вы также научились использовать эту архитектуру при создании базовых форм для ввода и редактирования данных.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Какое свойство компонента TQuery определяет, будут ли результаты запроса изменяемыми?

Свойство RequestLive.

Какое свойство компонента TDatabase используется для указания псевдонима, используемого в приложении?

Свойство DatabaseName.

Какой минимальный набор компонентов должно содержать приложение C++Builder, в которое вы хотите включить интерфейсные элементы для работы с базой данных?

Один из компонентов TTable, TQuery или TStoredProc вместе с компонентом TDataSource.

Вопросы

1. Какими двумя способами можно запустить мастер форм баз данных C++Builder?
2. Каковы ключевые свойства компонента TTable?
3. Каковы ключевые свойства компонента TDataSource?
4. Какой компонент необходим для обращения интерфейсных компонентов к потомкам TDataSet, таким как TTable?

Упражнения

1. Создайте форму для работы с базами данных полностью вручную, не используя технику drag-and-drop редактора полей, которую я показал вам сегодня.
2. Создайте форму для работы с базами данных, устанавливающую связь между двумя таблицами.
3. Создайте новую форму с помощью мастера форм и настройте ее по своему вкусу.
4. Создайте новую форму с помощью мастера форм, используя в качестве ее основы компонент TQuery, а не TTable.

Построение приложений для работы с базами данных



Кен Хендерсон

Вчера вы изучили основы построения форм для работы с базами данных. Сегодня вы получите дополнительные сведения о построении таких форм и создании приложений, использующих базы данных. Мы рассмотрим следующие темы:

- ☐ Использование компонента TDatabase в приложениях
- ☐ Определение псевдонимов BDE
- ☐ Создание модулей данных
- ☐ Создание форм типа главный/подчиненный
- ☐ Использование компонента TQuery
- ☐ Разработка отчетов

Компонент TDatabase

Запустите C++Builder, если вы еще не сделали это, и загрузите вторую из форм, созданных вчера. Мы начнем сегодня с изменения этой формы для использования компонента TDatabase.

Использование компонента TDatabase в приложениях, работающих с базами данных, не является обязательным, но это лучший способ управлять доступом к базе данных из одной ключевой точки. Например, для переключения уровня изоляции транзакции вы можете изменить значение свойства TransIsolation компонента TDatabase. Для изменения последовательности входа в базу данных, установленной по умолчанию, можно связать код с событием OnLogin. И наконец, для указания параметров драйвера вы можете добавить пункты к свойству Params.

Найдите компонент TDatabase во вкладке Data Access палитры компонентов и поместите его в форму. Как и для компонентов TTable и TDataSource, положение TDatabase относительно формы несущественно. Это невизуальный компонент — он не будет отображаться во время выполнения программы.

Установите для свойства AliasName компонента TDatabase псевдоним BCDEMOS, который ранее уже использовался в форме. Установите для свойства DatabaseName значение dbTest. DatabaseName определяет имя, по которому другие компоненты будут обращаться к TDatabase. Таким образом, для использования TDatabase совместно с TTable, вам нужно указать в свойстве DatabaseName компонента TTable то же имя, которое указано в свойстве DatabaseName компонента TDatabase.

После установки свойств компонента TDatabase нужно изменить TTable так, чтобы он мог обращаться к TDatabase. Перейдите в окно инспектора объектов и вызовите выпадающий список рядом со свойством DatabaseName компонента TTable. Вы должны увидеть в этом списке имя dbTest. Выберите его, щелкнув на нем мышью. Теперь ваша форма будет осуществлять доступ к базе данных через один компонент — TDatabase. На рис. 18.1 показана форма с настроенным компонентом TDatabase.

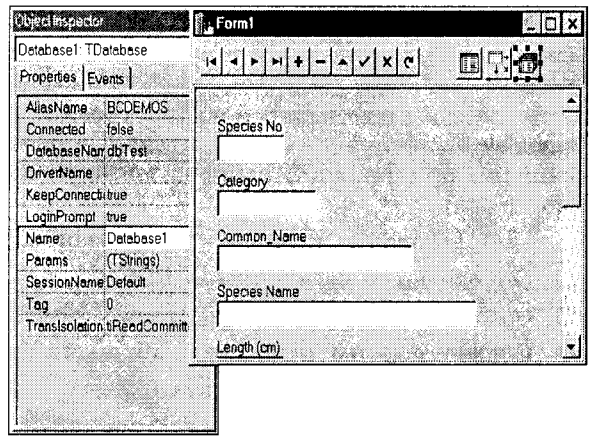


Рис. 18.1
Вы можете осуществлять доступ к базе данных через компонент TDatabase

Псевдонимы BDE

Утилита Database Explorer, входящая в C++Builder, имеет множество применений. Здесь мы используем ее для создания псевдонимов Borland Database Engine (BDE). Псевдонимы BDE требуются для доступа к объектам баз данных из C++Builder.

Мне никогда не нравился термин «псевдоним». Псевдоним — это слово для обозначения ненастоящего имени. Псевдоним же BDE аналогичен источнику данных ODBC. Это набор параметров, определяющих способ установления связи с базой данных или СУБД.

Вчера мы использовали встроенный псевдоним BCDEMOS. Он включен в C++Builder для поддержки некоторых приложений, демонстрирующих работу с базами данных. Сегодня я покажу вам, как определять свои собственные псевдонимы BDE. Они наверняка потребуются вам при создании реальных приложений.

Выберите в главном меню C++Builder пункт Database | Explore для запуска Database Explorer. Щелкните на вкладке Database, если она еще не выбрана, а затем на элементе Databases в расположенном ниже дереве. Щелк-

ните на этом элементе правой кнопкой мыши и выберите в контекстном меню пункт New.

Вам будет предложено выбрать драйвер базы данных для нового псевдонима BDE. Псевдоним, который мы создадим сегодня, будет предназначен для работы с БД Paradox, поэтому выберите драйвер Standard и нажмите кнопку OK.

Затем вам будет предложено ввести имя нового псевдонима базы данных. Наберите PDOXDEMO и нажмите Enter. После этого щелкните на пункте Path в панели справа, введите имя каталога, где находятся демонстрационные файлы баз данных C++Builder (в моем случае это C:\ProgramFiles\Borland\CBuilder\Examples\Data) и нажмите Enter.

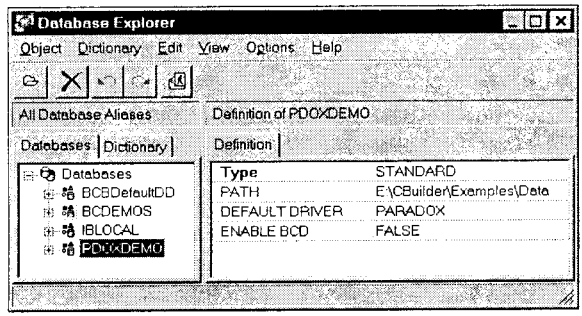


Указывая в новом псевдониме каталог C++Builder с демонстрационными файлами, вы дублируете псевдоним BCDEMOS. Это облегчает тестирование нового псевдонима, поскольку таблицы уже находятся в данном каталоге. В своих собственных приложениях вы, конечно, будете указывать другой каталог.

В завершение нажмите кнопку Apply в панели инструментов Database Explorer. На ней изображена синяя стрелка, показывающая вправо. Тем самым вы сохраните определение нового псевдонима.

На рис. 18.2 показано, как должен выглядеть экран после завершения всех действий.

Рис. 18.2
Определение нового псевдонима



Теперь, когда вы создали новый псевдоним BDE, давайте изменим форму так, чтобы она могла его использовать. Для этого выполните следующие действия:

1. Закройте Database Explorer и вернитесь к редактору форм C++Builder.
2. Щелкните на вашем компоненте TDatabase, затем нажмите F11 для перехода к инспектору объектов.
3. Измените значение свойства AliasName так, чтобы оно ссылалось на ваш новый псевдоним.

На рис. 18.3 показано, что вы должны видеть в инспекторе объектов.

Сохраните проект и запустите приложение. Вы увидите, что открытие таблицы Paradox BIOLIFE происходит так же, как и раньше. Окончив работу с приложением, закройте его и вернитесь в C++Builder.

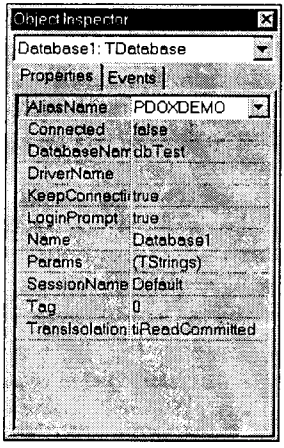


Рис. 18.3
Компонент TDatabase ссылается на новый псевдоним BDE

Модули данных



Модуль данных (data module) — это форма специального типа, содержащая невизуальные компоненты, такие как компоненты доступа к данным. Модуль данных позволяет вам разместить все невизуальные компоненты в одном месте для облегчения доступа и управления.

Я рекомендую создавать модуль данных для каждой базы данных, к которой предполагается обращение из приложений C++Builder. Вы не обязаны помещать все таблицы базы данных в один модуль данных. Однако я советую по возможности поступать именно так, чтобы впоследствии сохранить модуль данных в репозитории объектов C++Builder и использовать его при построении других приложений. Это поможет обеспечить единообразие в работе приложений и позволит управлять доступом к базам данных из одной ключевой точки — репозитория объектов.

Итак, выберите File | New Data Module. Перед вами появится небольшая форма, в которую вы можете помещать невизуальные компоненты. Возможно, перед продолжением работы с модулем данных стоит сделать его немного больше. Размер, установленный по умолчанию, достаточно мал.

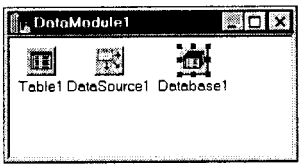
Теперь, имея модуль данных, мы переместим в него из вашей формы невизуальные компоненты доступа к данным. При создании полноценного приложения, вам следовало бы переместить в модуль данных как можно больше таких компонентов.

Начните с выделения компонентов TTable, TDataSource и TDatabase в вашей форме BIOLIFE. Затем нажмите Ctrl+X для удаления их в буфер обмена Windows. После этого выделите ваш новый модуль данных и нажмите Ctrl+V для вставки в него содержимого буфера обмена. Теперь компоненты доступа к данным перемещены в модуль данных. На рис. 18.4 показано, как должен выглядеть этот модуль.

После окончания операций удаления и вставки, вы должны вернуться к форме BIOLIFE и восстановить ссылки на компоненты доступа к данным, которые были разрушены при перемещении этих компонентов. Ранее интерфейсные компоненты (например, TDBEdit) обращались к TDataSource, кото-

рый теперь удален из формы. Ссылка на этот компонент больше недействительна и C++Builder удалил ее. Вам нужно восстановить разрушенные таким образом ссылки перед продолжением работы.

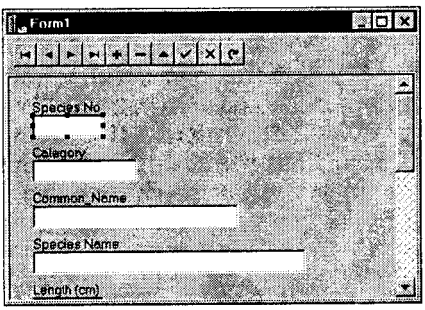
Рис. 18.4
Ваш новый модуль данных, содержащий компоненты доступа к данным



Перед тем, как начать использовать компоненты из нового модуля данных, вы должны включить заголовочный файл формы этого модуля в исходный код вашей формы, чтобы форма могла «видеть» компоненты модуля данных. Для этого загрузите форму BIOLIFE в редактор форм и через опцию Include Unit Hdr в меню File выберите исходный файл модуля данных. Теперь объекты модуля доступны в вашей форме так же, как если бы они были помещены непосредственно в форму.

Затем выделите все интерфейсные компоненты вашей формы (включая TDBNavigator) и нажмите F11 для перехода к инспектору объектов. Щелкните на кнопке вызова выпадающего списка рядом со свойством DataSource и выберите в списке компонент TDataSource, расположенный в вашем модуле данных. Вы заметите, что имя компонента TDataSource теперь дополнено именем исходного файла модуля данных. На рис. 18.5 показана форма, измененная для обращения к новому модулю данных.

Рис. 18.5
Ваша форма теперь обращается к созданному модулю данных





Вы можете быстро выделить несколько компонентов, заключив их в ограничивающую рамку. Для этого установите курсор мыши левее и выше самого левого компонента, нажмите левую кнопку и растягивайте рамку, пока она не будет окружать все компоненты, которые вы хотели выделить. После отпускания кнопки компоненты внутри рамки окажутся выделенными.

Поскольку мы открывали TTable при создании формы программным образом, необходимо изменить также ссылку в исходном тексте. Нажмите F11 для перехода к инспектору объектов; затем выберите вашу форму в выпадающем списке компонентов. Щелкните на вкладке Events и дважды щелкните на событии OnCreate. Вы должны увидеть следующий код:

```
Table1->Open();
```

Компонент Table1 больше не содержится в форме, поэтому этот код не будет компилироваться. Вы могли бы изменить строку на

```
DataModule1->Table1->Open();
```

Но это создаст межмодульную зависимость, которой следует по возможности избегать. В этом случае вы должны были бы гарантировать, что ваш модуль данных будет создан раньше, чем форма BIOLIFE, иначе обращение к методу, открывающему таблицу Table1, вызовет нарушение прав доступа. Лучше всего вообще не открывать таблицу через код. Гораздо проще открыть таблицу во время разработки, чем вызывать для этого соответствующий метод. Открывая таблицу на этапе разработки, вы можете просматривать данные во время работы над формой, которая к ним обращается, а также избегаете явного открытия TTable в исходном коде программы. Этот вариант проще и более гибок.

Сотрите строку Table1->Open() из обработчика события OnCreate и сохраните форму. Затем загрузите ваш модуль данных в редактор форм C++Builder и выделите компонент TTable. Теперь нажмите F11 для перехода к инспектору объектов и дважды щелкните на свойстве Active, чтобы открыть TTable. Сделав это, сохраните проект.

Вы можете запустить свое приложение прямо сейчас и убедиться, что оно работает, как и ожидалось. Закончив, закройте его для возврата в C++Builder.

Теперь, когда модуль данных завершен, давайте сохраним его в репозитории объектов. Вы часто будете убеждаться в том, что сохранение разработанной формы, будь это модуль данных или «обычная» форма, имеет большой смысл. Форму, которая была сохранена в репозитории объектов, можно использовать впоследствии как основу для создания другой формы или непосредственно включать в другие проекты.

Щелкните правой кнопкой мыши на модуле данных и выберите в контекстном меню пункт Add To Repository (Добавить в репозиторий). Введите dmBIOLIFE в поле Title (Заголовок) и BIOLIFE data module form в поле Description (Описание). Выберите в выпадающем списке Page пункт Data Modules и введите свое имя в поле Author, после чего нажмите кнопку OK. Диалоговое окно Add To Repository показано на рис. 18.6.

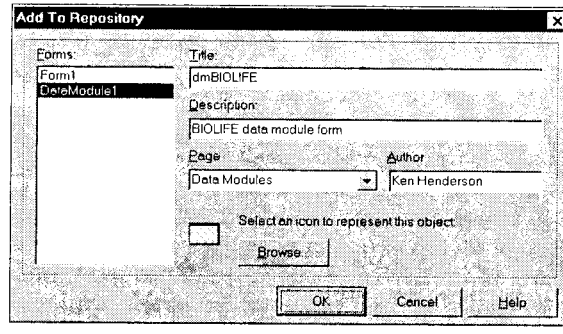


Рис. 18.6
Заполненное диалоговое окно Add To Repository



Для представления своего модуля данных в репозитории объектов вы можете также выбрать значок. Присваивание каждому модулю данных определенного изображения поможет легче различать модули, когда их станет достаточно много. Чтобы выбрать специальный значок для вашего нового модуля данных, щелкните на кнопке Browse в диалоговом окне Add To Repository.

Если вы еще не сохраняли форму вашего модуля данных, то перед ее добавлением в репозиторий объектов получите соответствующий запрос. В этом случае щелкните на кнопке Yes и введите путь для файла.

Теперь, когда вы добавили модуль данных dmBIOLIFE в репозиторий объектов, можно использовать его в любом приложении, которому необходим доступ к таблице BIOLIFE.

Создание формы главный/подчиненный



Форма типа *главный/подчиненный (master/detail)* обычно содержит как минимум две таблицы, связанные друг с другом. Как правило, эти таблицы отображаются в разных частях формы. Например, вы можете просматривать таблицу ORDERS в верхней части формы главный/подчиненный, а связанную с ней таблицу LINEITEM — в нижней. Обращение к каждой строке в ORDERS будет приводить к отображению связанных с ней строк в LINEITEM. Формы такого типа довольно часто встречаются в реальных приложениях, работающих с базами данных.

Конструирование формы главный/подчиненный станет следующей темой, которую мы рассмотрим сегодня. Я покажу вам, как она создается и используется в приложениях.

Вы также познакомитесь с компонентом TQuery. Все формы баз данных, которые вы создавали до сих пор, были основаны на компоненте TTable. Форма главный/подчиненный, которой мы займемся сейчас, будет использовать вместо него компонент TQuery.

Начните с запуска мастера форм баз данных (для этого выберите в главном меню пункт Database | Form Wizard). Щелкните на радиокнопках Create a master/detail form (Создать форму главный/подчиненный) и Create a Form using TQuery objects (Создать форму с использованием объектов TQuery); затем щелкните на кнопке Next.

Во втором диалоговом окне мастера форм вызовите выпадающий список Drive or Alias name (Имя диска или псевдонима) и выберите псевдоним PDOXDEMO, который вы определили раньше. Затем дважды щелкните на таблице CUSTOMER.DB в списке Table Name (Имя таблицы). Тем самым вы выберете в качестве главной таблицы формы главный/подчиненный таблицу CUSTOMER (в формате Paradox) из каталога с демонстрационными данными C++Builder.

После двойного щелчка на таблице CUSTOMER мастер форм автоматически перейдет к следующему шагу. Теперь вы можете выбрать, какие поля таблицы CUSTOMER включить в форму. Щелкните на кнопке >> для выбора всех полей, после чего нажмите кнопку Next.

В следующем диалоговом окне мастера форм баз данных запрашивается, каким образом вы хотите расположить поля главной таблицы. Возможны варианты Horizontal (Горизонтально), Vertical (Вертикально) и Grid (Сетка). Щелкните на кнопке Next для выбора варианта по умолчанию (Horizontal).

Теперь, когда вы выбрали главную таблицу, можно приступать к выбору подчиненной таблицы. Псевдоним, выбранный ранее для главной таблицы, будет использоваться по умолчанию и для подчиненной — вам останется только выбрать саму таблицу. Дважды щелкните на таблице ORDERS, чтобы выбрать ее в качестве подчиненной таблицы вашей формы.

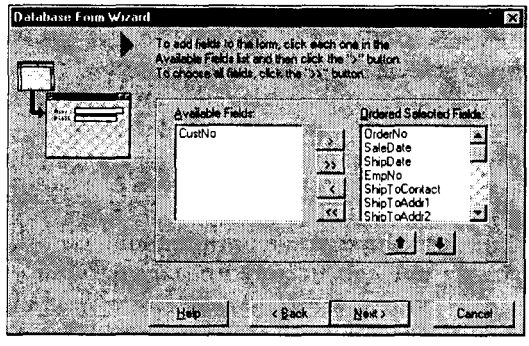
После выбора подчиненной таблицы, мастер форм баз данных спросит у вас, какие поля вы хотите включить в свою форму. Щелкните на кнопке >> для выбора всех полей, затем щелкните на поле CustNo в списке Ordered Selected Fields (Выбранные поля по порядку) и нажмите кнопку < для удаления этого поля из списка. Поскольку поле CustNo главной таблицы будет отображаться в верхней части формы, нет смысла дублировать его внизу. Щелкните на кнопке Next для продолжения.



В этом диалоговом окне вы можете также выбрать порядок отображения полей. Для перемещения полей в списке можно использовать технику drag-and-drop или воспользоваться кнопками со стрелками под списком Ordered Selected Fields. Каждое нажатие на кнопку перемещает выделенное поле на одну позицию.

На рис. 18.7 показано, как должно выглядеть диалоговое окно перед тем, как вы нажмете кнопку Next.

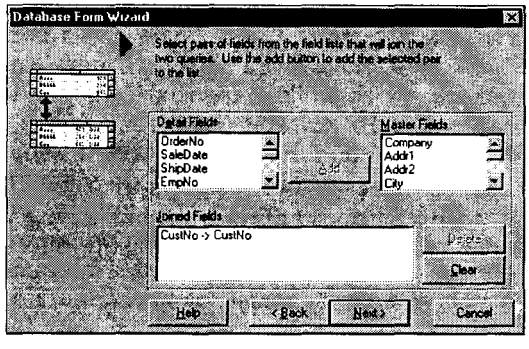
Рис. 18.7
Выбор всех полей за исключением CustNo



Как и для главной таблицы, в следующем шаге мастер форм спросит, какое расположение полей вы хотите использовать для подчиненной таблицы. По умолчанию устанавливается Grid (Сетка). Щелкните на кнопке Next для подтверждения этой установки и продолжения.

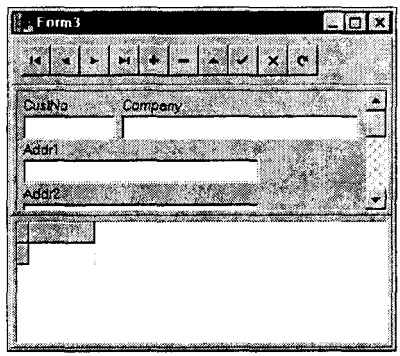
Теперь вам нужно выбрать пары полей, которые будут связаны в главной и подчиненной таблицах. При интуитивно-понятном дизайне баз данных эти поля в обеих таблицах имеют одинаковые имена. В данном случае вы будете связывать главную и подчиненную таблицы, используя столбцы CustNo. Щелкните на столбцах CustNo в обоих списках и нажмите кнопку Add. На рис. 18.8 показано, как должно выглядеть диалоговое окно после определения связанных полей. Закончив установку связи, щелкните на кнопке Next для продолжения.

Рис. 18.8
Связывание двух таблиц с использованием столбцов CustNo



Последнее диалоговое окно мастера форм позволяет вам выбрать, нужно ли генерировать только главную форму или главную форму вместе с модулем данных. Щелкните на радиокнопке Form and DataModule (Форма и модуль данных), затем нажмите кнопку Finish. Ваша новая форма будет сделана главной формой приложения, поскольку по умолчанию мастер форм генерирует именно главную форму. На рис. 18.9 показана законченная форма в редакторе форм C++Builder.

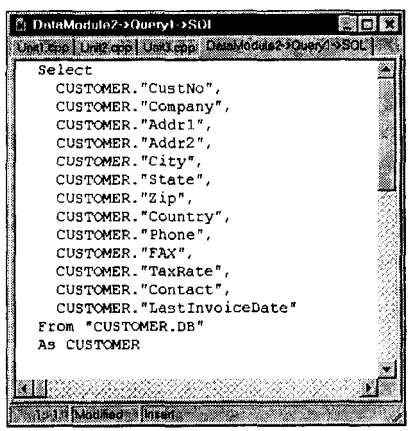
Рис. 18.9
Законченная форма главный/подчиненный



Перед запуском программы давайте немного изучим новую форму. Понимание того, как работает сгенерированная форма, поможет вам создавать свои собственные формы типа главный/подчиненный.

Вы должны были заметить, что после окончания работы мастера форм появились две новых формы. Это форма главный/подчиненный и форма модуля данных. Загрузите модуль данных в редактор форм и щелкните на самом левом компоненте TQuery. Нажмите F11 для перехода к инспектору объектов и дважды щелкните на свойстве SQL. Вы можете щелкнуть на кнопке вызова редактора кода для просмотра SQL в редакторе кода C++Builder. На рис. 18.10 показано, что вы должны увидеть.

Рис. 18.10
SQL, сгенерированный мастером форм баз данных



Этот запрос SQL предназначен для получения всех столбцов и строк таблицы CUSTOMER.DB. Обращение идет к каждому столбцу в отдельности, но поскольку вы выбрали их все в мастере форм, реально возвращаются все столбцы таблицы. SQL не содержит уточняющего оператора WHERE, поэтому

возвращаются также все строки таблицы. Просмотрев SQL целиком, нажмите Ctrl+F4 для закрытия окна редактора кода.

Теперь, когда вы увидели SQL для главной таблицы, давайте взглянем на SQL, сгенерированный для подчиненной таблицы. Щелкните на правом компоненте TQuery и затем дважды щелкните на его свойстве SQL в инспекторе объектов. Вы должны увидеть следующий SQL:

```
Select
  ORDERS."OrderNo",
  ORDERS."SaleDate",
  ORDERS."ShipDate",
  ORDERS."EmpNo",
  ORDERS."ShipToContact",
  ORDERS."ShipToAddr1",
  OREDRS."ShipToAddr2",
  ORDERS."ShipToCity",
  ORDERS."ShipToState",
  ORDERS."ShipToZip",
  ORDERS."ShipToCountry",
  ORDERS."ShipToPhone",
  ORDERS."ShipVIA",
  ORDERS."PO",
  ORDERS."Terms",
  ORDERS."PaymentMethod",
  ORDERS."ItemsTotal",
  ORDERS."TaxRate",
  ORDERS."Freight",
  ORDERS."AmountPaid",
From "ORDERS.DB"
As ORDERS
Where
  "ORDERS"."CustNo" =:"CustNo"
```

В этом SQL нужно отметить оператор WHERE. Он ограничивает строки, возвращаемые из таблицы ORDERS, только теми, у которых столбец CustNo совпадает с переменной :CustNo. Где же определена переменная :CustNo?

Чтобы найти ответ на этот вопрос, вернитесь к инспектору объектов и посмотрите на свойство DataSource компонента TQuery. Вы увидите, что оно указывает на TDataSource компонента TQuery главной формы. Это означает, что при изменении поля CustNo в главной таблице (например, при перемещении указателя текущей строки), подчиненный TQuery будет повторно выполнен в соответствии с новым значением. Таким образом, C++Builder автоматически синхронизирует эти два запроса. В подчиненной таблице будут видимы только те строки, которые соответствуют текущей строке в главной таблице.

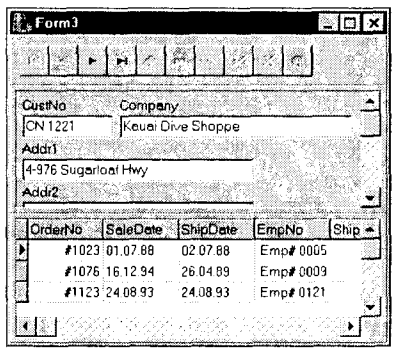


Рис. 18.11
Ваша форма главный/подчиненный во время выполнения программы

Чтобы проверить, как все это работает, давайте запустим приложение. На рис. 18.11 показано, что вы должны увидеть.

Закончив просмотр новой формы, закройте ее и вернитесь в C++Builder.

Отчеты

C++Builder обеспечивает два простых способа создания отчетов. Вы можете использовать компонент TQuickReport, который расположен во вкладке QReport палитры компонентов. Вы можете также создавать отчеты с помощью ReportSmith — генератора отчетов клиент/сервер, входящего в состав C++Builder. Сегодня я покажу вам процесс создания отчета с помощью компонента TQuickReport. За информацией о работе с ReportSmith обращайтесь к документации по C++Builder.

В C++Builder включены два удобных шаблона отчетов, которые вы можете приспособить для своих целей. Вы найдете их во вкладке Forms репозитория объектов. Выберите в меню File пункт New и щелкните на вкладке Forms. Найдите форму QuickReport List и дважды щелкните на ней. На рис. 18.12 показано, что вы должны увидеть.

С помощью пункта меню File | Include Unit Hdr выберите модуль, содержащий модуль данных BIOLIFE, созданный ранее. Это сделает TDataSource, определенный в модуле данных, доступным в отчете.

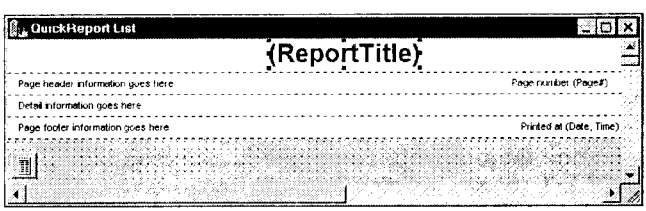


Рис. 18.12
Для создания нового отчета вы можете использовать шаблоны отчетов C++Builder

Затем щелкните на компоненте TQuickReport, содержащемся в форме отчета, и нажмите F11, чтобы перейти к инспектору объектов и установить свойство TDataSource этого компонента равным TDataSource модуля данных BIOLIFE. Щелкните на свойстве ReportTitle и установите для него значение BIOLIFE.

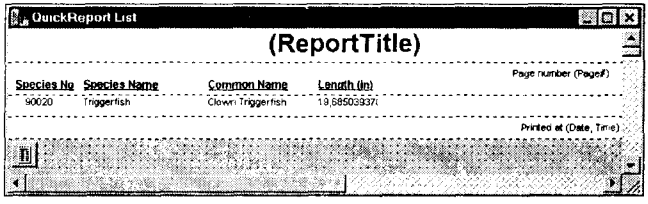
Теперь, когда компонент TQuickReport настроен, выделите три компонента-заполнителя TQRLabel в левой части формы отчета и удалите их. (Текст в каждом компоненте оканчивается словами ...goes here.)

После этого, поместите четыре компонента TQRDBText в строку отчета Detail (среднюю) и расположите их с равными интервалами вдоль строки. Выделите все четыре компонента и вызовите инспектор объектов, чтобы установить их общее свойство TDataSource равным свойству TDataSource вашего модуля данных. Затем выделите первый компонент TQRDBText и установите для его свойства DataField значение Species No. Щелкните на втором компоненте TQRDBText и установите для его свойства DataField значение Category BIOLIFE. Установите в третьем TQRDBText ссылку на поле Common_Name, а в четвертом — на поле Length_in.

Когда все поля установлены, поместите в строку отчета PageHeader (верхнюю) компоненты TQRLabel, соответствующие каждому TQRDBText. Эти компоненты будут служить заголовками столбцов, поэтому установите в свойстве

Caption каждого из них название соответствующего поля строки Detail. Чтобы выделить заголовки, вы можете использовать для них полужирный или подчеркнутый шрифт. На рис. 18.3 показано, как должен выглядеть ваш отчет.

Рис. 18.3
Ваш отчет после размещения столбцов с заголовками



Итак, вы закончили базовый отчет, который мы создавали для просмотра таблицы BIOLIFE. Теперь можно присоединить его к вашему приложению, чтобы отображать и выводить на печать во время выполнения программы. Загрузите в редактор форм форму BIOLIFE, которая была создана вчера, и выделите ее верхний компонент TPanel. Найдите компонент TButton во вкладке Standard палитры компонентов и поместите его в правой части TPanel. Измените значение свойства Caption на &Report и дважды щелкните на кнопке для установки события OnClick. Введите в редакторе кода следующий текст:

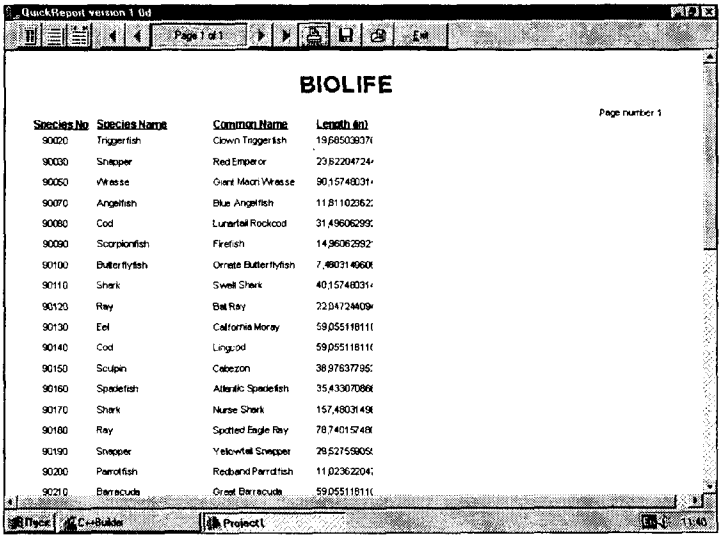
```
QRListForm->QuickReport->Preview();
```

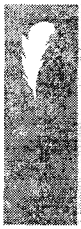
Затем выберите в меню File пункт Include Unit Hdr и дважды щелкните на исходном файле вашей формы отчета, чтобы добавить его в список включаемых файлов формы BIOLIFE. Это сделает форму отчета QRListForm и ее компонент TQuickReport видимыми в форме BIOLIFE, позволив тем самым вызывать метод Preview().

Выберите в меню Options пункт Project и установите форму BIOLIFE главной формой приложения. Таким образом, при запуске приложения форма BIOLIFE будет отображаться первой.

Теперь все готово для тестирования отчета. Нажмите F9 для запуска приложения и щелкните на кнопке Report в форме BIOLIFE. На рис. 18.14 показано, что вы должны увидеть.

Рис. 18.14
Вид вашего отчета во время выполнения программы





Обращение к форме отчета QuickReport, которая использует TDataSource, уже задействованный в другой форме, приводит к нежелательному побочному эффекту – прокручиванию TDataSource по мере помещения строк в отчет. Это означает, что форма также будет прокручиваться. После завершения печати отчета вы можете оказаться на другой строке. Обойти это можно, используя методы GetBookmark() и GotoBookmark() компонента TTable. За дополнительной информацией обращайтесь к документации по C++Builder.

Окончив просматривать отчет, закройте его вместе с приложением и вернитесь в C++Builder.

Заключение

В этой главе вы научились использовать модули данных в приложениях, работающих с базами данных. Вы также узнали, как определять свои собственные псевдонимы BDE и использовать их в приложениях. Вы научились создавать формы типа главный/подчиненный и разрабатывать простые отчеты, используя компоненты QuickReport.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Какое свойство используется для установки связей в C++Builder?

Свойство DataSource.

Какое средство C++Builder можно использовать для определения псевдонимов BDE?

Database (или SQL) Explorer.

Какие два способа разработки отчетов предоставляет C++Builder?

Использование компонентов QuickReport и использование ReportSmith.

Как лучше всего создавать таблицы и другие объекты баз данных для использования в моих приложениях?

СУБД Database Desktop (включенная в C++Builder) является упрощенной версией популярной СУБД Paradox для Windows фирмы Borland. Она хорошо подходит для создания таблиц, индексов и других элементов баз данных.

Вопросы

1. Какое свойство компонента TDatabase определяет имя, которое другие компоненты будут использовать для обращения к данному TDatabase?
2. Какой пункт меню File позволяет обеспечить обращение одного модуля к другому?
3. Где расположены шаблоны форм отчетов C++Builder?
4. Каковы преимущества использования модулей данных?

Упражнения

1. Создайте псевдоним BDE, используя утилиту конфигурирования BDE вместо Database Explorer.
2. Сконструируйте отчет типа главный/подчиненный, используя соответствующий шаблон отчета C++Builder и таблицы CUSTOMER и ORDERS, с которыми вы работали сегодня.

Создание и использование DLL



Кен Хендерсон

Если вы работаете в Windows, то вам неизбежно придется столкнуться с DLL. Достаточно только заглянуть в каталоги \Windows и \Windows\System. В своей системе я насчитал там более 350 динамических библиотек. Поэтому DLL следует безоговорочно признать фактом повседневной жизни в Windows.

Вопрос состоит в том, нужно ли *вам* создавать и использовать DLL? Сегодня вы найдете на него ответ. Мы рассмотрим следующие темы:

- Что такое DLL и почему их следует использовать
- Вызов функций и использование классов в DLL
- Написание DLL
- Использование DLL ресурсов

К концу дня вы будете лучше представлять, подходят ли для вас DLL. Я полагаю, что рано или поздно DLL войдут в вашу жизнь.

Обзор DLL

DLL — очень полезные создания. Чтобы аргументировать это утверждение, мне придется рассказать о той выгоде, которую дает их использование, и какое отношение это имеет к вам, программисту в C++Builder. Но перед этим давайте уточним, что же такое DLL.

Что представляет собой динамически компонуемая библиотека?

Говоря кратко, DLL (Dynamic Link Library) — это один или несколько участков кода, хранимых в файле с расширением .dll. Код, содержащийся в DLL, может быть вызван из исполняемой программы, но сама DLL не является автономной программой. DLL можно рассматривать как вспомогательный файл для главной программы.

Существуют два различных типа DLL: DLL кода и DLL ресурсов. Это деление не является однозначным. Вы можете без проблем иметь как код, так и ресурсы в одной и той же DLL. Однако в некоторых случаях удобно хранить код в одной DLL, а ресурсы — в другой. Немного позже я скажу, *почему* вам это может понадобиться (в разделе «Использование ресурсов в DLL»).

DLL могут содержать код в двух основных формах. Первая форма — это отдельные функции, которые можно вызывать из главного приложения. Вы уже делали это несколько раз, изучая предыдущий материал. Вспомните этот фрагмент кода из четырнадцатого дня, «Углубленное программирование»:

```
Screen->Cursors[myCursor] = LoadCursor(HInstance, "MYCURSOR");
```

Или этот, тоже из четырнадцатого дня:

```
DrawText(c.Handle, Panel->Text.c_str(),
  -1, temp, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
```

Функции LoadCursor() и DrawText() относятся к API Windows. Возможно, вы это знали. Но знали ли вы, что эти функции вызываются из DLL? Ведь они должны где-то храниться. Но где? В данном случае как LoadCursor(), так и DrawText() содержатся в DLL с именем User32.dll.

А как насчет всех тех стандартных диалоговых окон (типа File Open, Print, Printer Setup), которые мы использовали? Они должны откуда-то браться. Эти диалоговые окна хранились в виде ресурсов в файле с именем Comctl32.dll. Таким образом, вы все время использовали DLL, сами того не подозревая.

Вторая форма кода, который может содержаться в DLL — это классы Си++. Вместо доступа к отдельным автономным функциям, вы можете создать экземпляр класса и вызывать функции-члены этого класса, находящиеся в DLL. Я буду рассматривать эту сторону DLL немного позже, в разделе «Импорт и экспорт функций и классов».

Может создаться впечатление, что свои собственные DLL должны использоваться каким-то особым образом. Это не так. Создав DLL, вы можете вызывать ее функции так же, как и функции API Windows. Создавать DLL не слишком сложно, поэтому нет никаких серьезных препятствий для их использования в своей работе.

Почему стоит использовать DLL?

Это хороший вопрос. Что же дает вам DLL, кроме исключения части кода из исполняемого файла? Использование DLL имеет следующие преимущества:

- ☐ Эффективное многократное использование кода
- ☐ Разделение кода между многими приложениями
- ☐ Секционирование кода
- ☐ Обеспечение многоязыковой поддержки
- ☐ Эффективное использование ресурсов Windows

Давайте рассмотрим каждый из этих пунктов в отдельности, и тогда станет ясно, смог ли я убедить вас в полезности DLL.

Эффективное многократное использование кода

Многократное использование кода является существенной частью объектно-ориентированного программирования. DLL могут оказать здесь большую помощь. Пусть, к примеру, у вас есть набор классов, написанных для поддержки мультимедиа в ваших программах. Эти классы могут включать в себя базовый класс, выполняющий основные операции, и производные классы, выполняющие специализированную обработку файлов WAV и MIDI, аудио CD и видео AVI. Для создания таких классов вам пришлось бы затратить много усилий. Было бы неплохо использовать весь этот код в любых приложениях, которым он требуется. Именно такую возможность предоставляют вам DLL.

Вы должны откомпилировать все свои классы в DLL. Когда это сделано, для создания экземпляров класса достаточно включить в программу заголовочный файл для классов, содержащихся в DLL. (На самом деле требуется выполнить еще пару действий, но они просты и я не буду сейчас забегать вперед.) Что может быть проще?

Теперь вы имеете свои классы под рукой в любое время. Кроме этого, вы можете передать DLL своему лучшему другу-программисту и он также сможет использовать эти классы. Более того, вы можете продать DLL другим программистам. Ведь вы вложили в нее свой труд, и теперь можно рассчитывать на получение отдачи!



Вы может даже создавать DLL, которые могут быть вызваны из приложений, написанных в Delphi, Borland C++, Visual Basic, Visual C++ или других средах программирования. Естественно, это зависит от кода, содержащегося в DLL, и способа обращения к нему, но такая возможность существует.

Вы увидели, в чем суть? Один раз написав DLL, вы легко можете использовать ее в любое время, везде, где захотите. Для этого достаточно только несколько раз щелкнуть мышью.

Разделение кода между приложениями

Разделение кода связано с его многократным использованием, но идет немного дальше. Предположим, что вы работаете программистом в большой корпорации. Вы можете иметь сотни пользователей, каждого со своей собственной системой (в этом примере мы игнорируем сетевые соглашения). Допустим, что вы написали пять различных приложений для этих пользователей. Пусть каждое из этих пяти приложений использует ваши классы для работы с мультимедиа (которые обсуждались выше), а объем кода после компиляции классов составляет 100 Кб (по самой скромной оценке). Если вы не используете DLL, эти 100 Кб кода придется продублировать пять раз (для каждого из приложений), что даст в итоге 500 Кб. Это явная избыточность.

Лучший подход заключается в помещении мультимедиа-классов в DLL. Каждая из пяти программ может использовать одну и ту же DLL, когда дело дойдет до работы с мультимедиа. Это одно из достоинств DLL: будучи написанными один раз, они доступны всем вашим приложениям. Каждый пользователь получит на свою машину дополнительные 100 Кб, *но* при этом каждое из пяти приложений будет уменьшено в размере на 100 Кб. Экономия дискового пространства составит 400 Кб для каждого пользователя. Сохра-

нение 400 Кб может показаться не таким уж значительным, но если вы умножите их на сотни пользователей, то это начинает становиться заметным в рамках корпорации (те, кто считает деньги, обычно обращают внимание на такие вещи). И это еще достаточно простой пример. В реальных ситуациях, используя DLL, вы легко можете сохранить несколько мегабайт в расчете на одного пользователя.

Но что, если три из пяти программ запущены одновременно? Нет проблем. Каждая программа берет код из DLL по мере необходимости, и конфликтов не возникает. Windows отслеживает, кто кого вызывает, и тому подобные вещи. Вам нужно только написать DLL и начать ее использовать.

Секционирование кода

Секционируя свой код, вы можете упростить его обновление, когда до этого дойдет дело. Я не сторонник написания отдельных DLL для каждой части программы, но там, где имеет смысл разделить программу на отдельные связанные участки кода, вам следует это делать. В разделении приложения на DLL нет другой пользы, кроме секционирования кода. Но если у вас есть свои библиотеки классов, их стоит поместить в отдельные DLL. Это наиболее очевидно при создании библиотек, которые могут использоваться любыми вашими программами.



При разработке класса или компонента всегда следует подумать о том, как организовать многократное обращение к нему в будущем. Иными словами, планируйте наперед и с самого начала разрабатывайте код для многократного использования. Даже если вам кажется, что код, над которым вы работаете сегодня, вряд ли будет снова использован в будущем, пишите его, имея в виду эту возможность. Вы никогда не знаете, когда вам потребуется код такого же типа. Если вы приняли объектно-ориентированный образ мышления, программирование для многократного использования должно войти в привычку.

Преимущество такого подхода состоит в том, что ваше приложение при необходимости может быть легче модифицировано. Давайте говорить прямо: даже лучшие программы выходят на рынок с ошибками. Основной вопрос заключается в том, сколько их и насколько они серьезны, а не в том, есть ли ошибки вообще. Если вы обнаружили ошибки в своей DLL, то можете исправить их и переслать пользователям только новую DLL вместо компиляции и пересылки всего исполняемого файла. Для тех приложений, которые вы писали раньше, это преимущество может показаться не таким уж большим. Но с другой стороны, со временем вы можете начать писать программы размером в несколько мегабайт, состоящие из десятков исходных модулей. В таких случаях секционирование имеет гораздо больший смысл.

Многоязыковая поддержка

Еще несколько лет назад вы могли писать программу, особенно не беспокоясь о международном рынке. Вы писали программу и создавали пункты меню, диалоговые окна, подсказки, сообщения об ошибках и прочие элементы на своем родном языке, и этого было достаточно. Вы выпускали программу на рынок и больше не думали об этом.

Но вы заметили, что мир становится меньше? Из-за взрывного развития Internet и World Wide Web, положение вещей полностью изменилось по

сравнению с тем, каким оно было всего пару лет назад. Вы можете создать демонстрационную или условно-бесплатную версию своей программы и поместить ее в Сеть. Спустя часы, даже минуты, люди по всему миру смогут получить доступ к вашей программе. Это одновременно и воодушевляет, и пугает. Наличие такого феномена означает, что вы должны заранее подумать о переводе программы на другие языки.

Один из способов заключается в создании DLL ресурсов, содержащих строковые ресурсы на разных языках. Вы можете иметь отдельные DLL для каждого языка, или объединить все строки в одной DLL и загружать нужный вариант строки во время выполнения программы. Во время выполнения вы можете использовать для загрузки строк и их присваивания различным компонентам функцию LoadString() из API Windows.



Одним из недостатков модели программирования C++Builder является то, что в ней не используются традиционные ресурсы, как в других средах программирования. Элементы программы наподобие меню не загружаются как отдельные ресурсы, а вместо этого содержатся в общем ресурсе формы. Это затрудняет введение многоязыковой поддержки и является одним из случаев, когда данная модель работает против вас.

Как и раньше, предварительное планирование может впоследствии сберечь много усилий. Если вы задумываете свое приложение, имея в виду введение многоязыковой поддержки, то перевод на другие языки может оказаться намного проще, когда в этом возникнет необходимость.

Эффективное использование ресурсов Windows

Современные системы работают быстрее, имеют больше оперативной памяти и дискового пространства. Легко начать рассуждать примерно так: «Я использую всего 2 Мб системного ОЗУ, зачем мне думать об экономии?» Однако вы всегда должны следить за тем, какое количество системных ресурсов используете. Здесь снова могут помочь DLL.

На самом деле это продолжение разговора о разделении кода между различными приложениями. Давайте вернемся к примеру, который рассматривался раньше. (Вспомните, что у вас было пять приложений, использующих некоторый общий код.) Если вы не используете DLL, то при одновременном запуске нескольких программ в память будут загружены одинаковые фрагменты кода. Вы перерасходуете системные ресурсы, поскольку каждая программа загружает свою копию одного и того же кода. Чтобы избежать повторной загрузки кода, вы можете использовать DLL и загрузить код только один раз. Он будет использоваться всеми вашими приложениями, и тем самым «утечка» памяти в системе будет уменьшена.

Вызов функций и использование классов в DLL

Я намереваюсь рассказать о том, как вызывать функции, содержащиеся в DLL. Но перед использованием функции из DLL вы должны сначала загрузить DLL в память. Давайте кратко рассмотрим, как происходит загрузка DLL.

Загрузка DLL

Загрузка DLL может быть выполнена двумя способами — статически или динамически. Оба эти способа имеют свои достоинства и недостатки.

Статическая загрузка (static loading) означает, что ваша DLL автоматически загружается при запуске приложения, обращающегося к этой DLL. Ваша DLL содержит экспортируемые функции. Описание этих функций находится в специальном файле, называемом *файлом библиотеки импорта (import library file)*. Файл библиотеки импорта имеет имя, совпадающее с именем соответствующей DLL, и расширение .lib.

Для использования статической загрузки вы должны подключить к проекту на этапе компоновки .lib-файл для DLL. В приложениях C++Builder это сводится к добавлению .lib-файла к проекту через менеджер проектов. DLL загружается при загрузке приложения и вы можете вызывать экспортируемые функции так же, как и любые другие функции. Это самый легкий путь использования кода, содержащегося в DLL. Недостаток такого подхода состоит в том, что при отсутствии необходимой DLL программа откажется загружаться.

Динамическая загрузка (dynamic loading) означает, что вы загружаете DLL при необходимости и выгружаете ее по окончании использования. Такой способ загрузки DLL также имеет свои достоинства и недостатки. Одним из достоинств является то, что DLL находится в памяти только до тех пор, пока она нужна, что приводит к более эффективному использованию памяти. Другое достоинство — более быстрая загрузка приложения, поскольку не требуется загружать сразу весь необходимый код.

Главным недостатком динамической загрузки является то, что от вас требуется гораздо больше работы. Сначала вы должны загрузить DLL, используя функцию LoadLibrary() из API Windows. Затем, окончив работу с DLL, вы должны выгрузить ее с помощью функции FreeLibrary(). Кроме того (здесь и начинается основная работа), вам нужно использовать функцию GetProcAddress() для получения указателя на ту функцию, которую вы хотите вызвать. Для иллюстрации, давайте посмотрим на то, как вызываются функции из DLL.

Вызов функций, расположенных в DLL

Способ вызова функции из DLL зависит от того, как была загружена эта DLL — статически или динамически. Вызов функций из динамически загруженных DLL не доставляет большого удовольствия. Вам требуется установить указатель на функцию в DLL, а указатели на функции можно перепутать. Допустим, у вас есть DLL-функция с именем SayHello(). Она может выглядеть следующим образом в исходном тексте DLL:

```
void __export __stdcall SayHello(HWND hWnd)
{
  MessageBox(hWnd, "Hello There!!",
    "Message From a DLL", MB_OK);
}
```

Для вызова этой функции из вашей программы сначала необходимо объявить указатель на функцию:

```
void (__stdcall *SayHello)(HWND);
```

Когда это сделано, вы может загрузить DLL, использовать GetProcAddress() для получения указателя на функцию, вызвать функцию и, наконец, выгрузить DLL. Это выглядит примерно так:

```
// загрузка DLL
HINSTANCE dllInstance = LoadLibrary("mystuff.dll");
// получение адреса функции
SayHello = (void(__stdcall*)(HWND))
  GetProcAddress(dllInstance, "_SayHello");
// вызов функции
SayHello(Application->Handle);
// выгрузка DLL
FreeLibrary(dllInstance);
```

Указатели на функции являются одной из наиболее запутанных сторон Си++, если рассматривать их с точки зрения синтаксиса. Тем не менее, при необходимости загрузить DLL во время выполнения программы вам не остается ничего другого. Следует заметить, что этот фрагмент немного сокращен. Практически всегда вам нужно будет добавлять код для проверки ошибок, чтобы убедиться в правильной загрузке DLL и возврате нормального адреса функцией GetProcAddress(). Скорее всего, вы не станете использовать динамическую загрузку DLL до тех пор, пока без нее нельзя будет обойтись.

Вызов функций из статически загруженных DLL выглядит после этого совсем просто. Сначала вы должны объявить функцию. После этого она вызывается как любая другая функция:

```
extern "C" __import void SayHello(HWND);
// затем...
SayHello(Application->Handle);
```

Пока вам действительно не понадобится та гибкость, которую обеспечивает динамическая загрузка, вы почти всегда будете делать выбор в пользу статической загрузки.



Некоторые инструментальные средства и библиотеки независимых производителей поставляются в виде DLL. Для использования этих инструментов или библиотек вам нужно поместить соответствующие DLL в рабочий каталог приложения. Затем с помощью директивы #include необходимо включить в исходный текст программы заголовочный файл, поставляемый вместе с библиотекой. К каждой библиотеке должна прилагаться полная инструкция по работе с этой библиотекой.

Создание DLL

Создание DLL не является особенно трудной задачей. Есть несколько специальных вещей, о которых вам необходимо знать, но в основном это обычное программирование на Си++. По всей вероятности, чаще всего вы будет использовать статическую загрузку. Для статически загружаемых DLL вам потребуется не менее четырех файлов:

- Один или несколько исходных файлов. Эти файлы не требуется поставлять вместе с DLL.
- Один или несколько заголовочных файлов. Эти файлы будут содержать объявления или классы, которые могут быть вызваны из DLL.

- Файл библиотеки импорта. Этот файл позволяет компоновщику связать вызовы в приложении с адресами в DLL.
- Сама DLL.

Из этих файлов в комплект поставки DLL должны входить заголовок (.h), библиотека импорта (.lib) и сама динамически компонуемая библиотека (.dll).



Для DLL, загружаемых с помощью LoadLibrary(), библиотека импорта не требуется.



Если вы имели опыт работы с DLL, то для вас могло стать неожиданным то, что C++Builder не поддерживает файлы определения модуля (DEF-файлы). Используйте библиотеку импорта и специально экспортируйте все функции, которые должны вызываться из DLL.

Давайте начнем с основ построения DLL. После этого мы вместе создадим DLL.

Импорт и экспорт функций и классов

Функции в DLL делятся на три основные категории:

- Функции, вызываемые внутри DLL
- Функции, вызываемые извне DLL
- Функции, которые являются членами классов, содержащихся в DLL

Функции, вызываемые внутри DLL, не требуют специального оформления. Вы объявляете их так же, как и обычные функции. Такие функции могут быть вызваны другими функциями в пределах DLL, но не могут вызываться извне DLL. Иными словами, приложение не имеет доступа к этим функциям. В этом отношении они похожи на закрытые функции-члены класса. Ваше приложение не будет «видеть» эти функции и даже не будет знать об их существовании.



Попытка использования функции, закрытой по отношению к DLL, может не привести к ошибке компиляции, но обязательно вызовет ошибку компоновки. Компоновщик выдаст примерно следующее сообщение: Unresolved external 'MyFunction()' referenced from module unit1.cpp (Нераспознанная внешняя ссылка 'MyFunction' в модуле unit1.cpp).



В дополнение к функциям, DLL может содержать глобальные данные, к которым могут обращаться все функции этой DLL. В 16-разрядной среде Windows глобальные данные разделяются всеми экземплярами DLL. Иными словами, если одна программа присвоила глобальной переменной x значение 100, x будет иметь значение 100 во всех приложениях, вызывающих эту DLL. В 32-разрядной среде Windows это не так. В Win32 для каждого процесса, загружающего DLL, создаются отдельные копии глобальных данных DLL.

Экспорт функций и классов

Другим классом функций являются те, которые могут быть вызваны извне DLL. Эти функции сделаны открытыми в результате *экспорта (exporting)*. Они могут вызываться как функциями самой DLL, так и из приложения.



Функции DLL могут быть вызваны как выполняемыми приложениями, так и другими DLL. Иными словами, одна DLL может вызывать функции из другой DLL.

После того, как функция экспортирована, вы можете вызывать ее из своего приложения. Для экспорта функции при создании DLL используется ключевое слово __export. Исходный код DLL для простой функции может выглядеть примерно следующим образом:

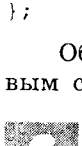
```
void __export SayHello(HWND hWnd)
{
  MessageBox(hWnd, "Hello from a DLL!",
    "DLL Message Box", MB_OK | MB_ICONEXCLAMATION);
}
```

Эта функция теперь экспортирована и может быть вызвана из любого приложения.

Экспорт классов осуществляется в основном так же, за исключением того, что модификатор __export помещается в объявлении класса. Объявление для экспортируемого класса выглядит примерно следующим образом:

```
class __export MyClass {
  public:
    MyClass(HWND hwnd);
    void SaySomething(char* text);
    void Beep();
  private:
    HWND hWnd;
};
```

Обратите внимание, что модификатор __export размещен между ключевым словом class и именем класса.



Если у вас возникают проблемы с экспортом функций или классов, вы можете обработать DLL утилитой TDUMP. Информация, выданная TDUMP, будет содержать раздел со списком имен, экспортируемых из DLL. Изучив этот раздел, вы сможете лучше понять, где находится ошибка.

Импорт функций и классов

Экспорт функции — это только половина дела. При создании приложения вы должны *импортировать* функцию, вызываемую из DLL. Как вы могли догадаться, импорт функции осуществляется с помощью модификатора __import.



Ключевые слова __import и __export имеют впереди два знака подчеркивания. Такие обозначения широко используются производителями компиляторов, чтобы отличать специальные ключевые слова компилятора от ключевых слов языка Си++.

Для импорта функции, содержащейся в DLL, вы должны использовать в объявлении функции модификатор __import. Например, для вызова функции SayHello() из предыдущего примера, приложение может использовать следующий код:

```
extern "C" void __import SayHello(HWND hWnd)
//
// затем...
//
SayHello(Handle);
```

Модификатор __import указывает компилятору, что ссылка на объявленную функцию содержится в файле библиотеки импорта. Собственно вызов функции не отличается от вызова любой другой функции.



В этом коде второй раз появляется модификатор extern "C". Он используется для предотвращения искажения имени (name mangling) функции. Вы должны использовать модификатор extern "C" для каждой автономной функции в DLL. Это обеспечит импорт функции любым приложением, в том числе откомпилированным с помощью компилятора, отличного от компиляторов фирмы Borland.

Импорт классов осуществляется аналогично. Перед использованием класса, находящегося в DLL, вы должны объявить его с модификатором __import:

```
class __import MyClass {
  public:
    MyClass(HWND hwnd);
    void SaySomething(char* text);
    void Beep();
  private:
    HWND hWnd;
};
```

После этого вы можете создавать экземпляры этого класса и обращаться к его открытым функциям и членам-данным по мере необходимости:

```
MyClass Billy(Handle);
Billy.SaySomething("Hi, there!");
Billy.Beep();
```

Таким образом, главное при написании и использовании DLL — это правильно выполнить экспорт и импорт. После этого работа с DLL не представляет никаких трудностей.

Забудьте все это

Забудьте все то, что я сейчас говорил про __import и __export. Ладно, я слегка шучу. Модификаторы __import и __export играют большую роль при написании и использовании DLL, и вы должны понимать, как они применяются. Но для приложений C++Builder разработан новый способ импорта и экспорта классов и функций. Вместо использования __import и __export лучше применять ключевое слово __declspec().

Это ключевое слово дает вам большую гибкость там, где вы должны применять модификаторы __import и __export. Раньше вы должны были убедиться, что поместили модификатор __import или __export точно в нужном месте определения функции, иначе компилятор выдавал сообщение об ошибке. Используя ключевое слово __declspec(), вы можете помещать моди-

фикатор в любом месте объявления функции (в пределах разумного, конечно). Например, обе следующие строки правильны:

```
__declspec(dllexport) void SaySomething(HWND hWnd);
void __declspec(dllexport) SaySomething(HWND hWnd);
```

Если бы вы использовали __export, его нужно было поместить между типом возвращаемого значения и именем функции. Используя __declspec(), вы не должны об этом беспокоиться.



Ключевое слово __declspec(), так же как __import и __export, имеет впереди два знака подчеркивания. Поскольку __declspec() является ключевым словом, оно будет выделено полужирным шрифтом после правильного ввода. Выделение синтаксических конструкций – характерная черта редактора кода C++Builder. Если вы используете только один знак подчеркивания, выделения не произойдет. Таким образом вы можете проверять правильность ввода ключевых слов.

Из приведенного выше примера вы заметили, что __declspec() имеет один параметр. Для экспорта функции нужно использовать __declspec(dllexport); для импорта используйте __declspec(dllimport). Вот и все. (Не думайте, что раньше я вел вас по ложному пути. Знание модификаторов __import и __export никогда не помешает. Если вы будете работать с компилятором, отличным от C++Builder, вам, возможно, придется использовать __import и __export.

Секрет заголовков DLL

Вы могли заметить одну потенциальную проблему при чтении предыдущего раздела. Если вам требуется экспортировать функцию или класс при создании DLL и импортировать их при создании .exe, необходимо иметь два отдельных заголовка: один со всеми экспортируемыми функциями, объявленными с модификатором __export, а другой — со всеми импортируемыми функциями, объявленными с модификатором __import. Но на самом деле это не обязательно. Вы можете обойтись только одним заголовком, используя макросы.

Заголовок практически любой DLL будет содержать примерно следующие строки:

```
#if defined(BUILD_DLL)
#  define DLL_EXP __declspec(dllexport)
#else
#  if defined(BUILD_APP)
#    define DLL_EXP __declspec(dllimport)
#  else
#    define DLL_EXP
#  endif
#endif

extern "C" void DLL_EXP SayHello(HWND);

class DLL_EXP MyClass {
  public:
    MyClass(HWND hwnd);
    void SaySomething(char* text);
    void Beep();
  private:
    HWND hWnd;
};
```

Группа директив препроцессора в верхней части заголовка работает следующим образом: если определена константа BUILD_DLL, имя DLL_EXP раскрывается в текст __declspec(dllexport). Если определена константа BUILD_APP, DLL_EXP раскрывается в __declspec(dllimport).

Это означает, что вы можете использовать один и тот же заголовок для DLL и приложения. Работая с DLL, вы делаете следующее:

```
#define BUILD_DLL
#include "mydll.h"
```

В этом случае все классы и функции, использующие DLL_EXP, будут экспортируемыми. При создании приложения вы используете следующий код:

```
#define BUILD_APP
#include "mydll.h"
```

Здесь DLL_EXP раскрывается в __declspec(dllimport), и тогда все функции и классы, использующие DLL_EXP, будут импортируемыми. Это позволяет вам использовать один и тот же заголовок как для создания приложения, вызывающего DLL, так и для самой DLL.

Создание DLL с помощью репозитория объектов

DLL в C++Builder создаются через репозиторий объектов. Выберите в главном меню пункт File | New для отображения репозитория. Затем щелкните на значке DLL для создания новой DLL и нажмите кнопку OK. На экране появится редактор кода со следующим файлом в окне редактирования:

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: #pragma hdrstop
 4: //---------------------------------------------------------------------
 5: //   Important note about DLL memory management:
 6: //
 7: //   If your DLL exports any functions that pass String objects (or structs/classes
 8: //   containing nested Strings) as parameter or function results, you will need
 9: //   to add the library BCBMM.DLL to both the DLL project and any EXE projects
10: //   that use the DLL. This will change the DLL and its calling EXE's to use
11: //   the BCBMM.DLL as their memory manager. In these cases, the file BCBMM.DLL
12: //   should be deployed along with your DLL. To avoid using BCBMM.DLL, pass
13: //   string information using "char *" or ShortString parameters.
14: //---------------------------------------------------------------------
15: USERES("Project1.res");
16: //---------------------------------------------------------------------
17: int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
18: {
19:     return 1;
20: }
21: //---------------------------------------------------------------------
```

Прежде всего давайте поговорим о том, что означают комментарии в строках с 5 по 13. В том случае, когда ваша DLL содержит экспортируемые функции, принимающие в качестве аргумента экземпляр класса String, вы должны сделать следующее:

- Добавить к проектам DLL и .exe библиотеку BCBmm.dll.
- Обеспечить поставку файла BCBmm.dll вместе с вашей DLL.

Чтобы избежать этих требований, достаточно не использовать в качестве параметров функций экземпляры класса String. Например, вместо

```
MyFunction(String& s)
{
}
```

используйте

```
MyFunction(char* buff)
{
}
```

Это усложнение довольно легко обойти, и вам никогда не потребуется BCBmm.dll. Вы только должны знать об ограничениях, накладываемых на использование класса String в функциях DLL. Обратите внимание, что вы можете использовать объекты String внутри самой DLL, не прибегая к BCBmm.dll. Ограничение относится только к экспортируемым функциям, имеющим String в качестве параметра или возвращаемого значения.

Теперь взгляните на строку 2. Обратите внимание, что сюда автоматически вставлено #include <vcl\vcl.h>. Это сделано для удобства работы и не является обязательным требованием. Если DLL содержит только ваш собственный код (без классов или форм VCL), эту директиву #include следует убрать. То же самое относится и к коду в строке 15. Если вы не используете формы VCL в своей DLL, можете удалить эту строку. (Чуть позже я расскажу о формах VCL в DLL.)

Наконец, обратите внимание на функцию DllEntryPoint(). Эта функция является точкой входа в DLL. Она вызывается при начальной загрузке DLL, а также в том случае, когда другой процесс (программа или DLL) обращается к данной DLL. В дальнейшем эта функция будет вызываться при отсоединении процесса от DLL и, в конечном итоге, при выгрузке DLL. Может возникнуть необходимость следить за всем этим, но большую часть времени вы не должны беспокоиться. Если вам нужно отслеживать подобные вещи, используйте параметр reason для определения того, что происходит с DLL. Чаще всего вы можете просто оставить функцию DllEntryPoint() в том виде, который она имела по умолчанию.



Первым параметром функции DllEntryPoint() является hinst. Это дескриптор экземпляра DLL (HINSTANCE). Если какая-либо функция требует этот дескриптор (например, LoadIcon() или LoadCursor()), вы можете использовать параметр hinst для создания глобальной переменной. Затем можно обращаться к этой глобальной переменной каждый раз, когда возникает необходимость передать дескриптор. (Использование дескриптора экземпляра описано ниже, в разделе «Использование ресурсов DLL».)

Теперь, когда редактор кода отображает исходный текст DLL, вам остается только начать ввод кода. Убедитесь, что вы использовали заголо-

вочный файл для DLL и следуйте тем процедурам, о которых я говорил в связи с экспортом функций и классов. Напишите все автономные функции или классы, которые должны находиться в DLL. После этого вы можете выбрать пункт Run | Build All или Run | Make для компиляции и построения DLL.

Давайте рассмотрим эту последовательность по шагам, чтобы она стала более понятной.

1. Выберите в главном меню пункт File | New. После отображения репозитория объектов, дважды щелкните на значке DLL для создания нового проекта DLL. Для вас будет создан новый пустой проект.
2. В редакторе кода измените строку #include <vcl\vcl.h> на
   ```
   #include <windows.h>
   ```
   (Мы не будем использовать классы VCL в этой DLL, но нам потребуется файл Windows.h.)
3. Сотрите все строки комментариев, относящиеся к BCBmm.dll.
4. Добавьте следующие две строки перед функцией DllEntryPoint():
   ```
   #define BUILD_DLL
   #include "MyDll.h"
   ```
5. Добавьте после функции DllEntryPoint() код из листинга 19.1 (Функция DllEntryPoint() и весь текст выше нее уже будут находиться в окне редактора кода.)
6. Сохраните проект под именем MyDll.

Листинг 19.1. StaticLd.cpp

```
//---------------------------------------------------------------
#include <windows.h>
#pragma hdrstop

#define BUILD_DLL
#include "StaticLd.h"

int WINAPI DllEntryPoint(HINSTANCE, unsigned long, void*)
{
  return 1;
}

//---------------------------------------------------------------
void DLL_EXP SayHello(HWND hWnd)
{
  MessageBox(hWnd, "Hello from a DLL!",
    "DLL Message Box", MB_OK | MB_ICONEXCLAMATION);
}

MyClass::MyClass(HWND hwnd)
{
  hWnd = hwnd;
}
```

```
void MyClass::SaySomething(char* text)
{
  MessageBox(hWnd, text,
    "A Text Message From a DLL Class", MB_OK);
}

void MyClass::Beep()
{
  MessageBeep(-1);
}
```



Этот файл называется StaticLd.cpp, поскольку соответствующая ему DLL будет загружаться статически. На CD-ROM, прилагаемом к этой книге, содержится программа CallDll.mak. Она вызывает функции статической DLL, а также функции другой DLL, загружаемой динамически. Имена файлов сделаны настолько информативными, насколько это возможно в рамках 8-символьного соглашения. Помните, что в своих программах и DLL вы можете использовать длинные имена, но я использую соглашение 8.3 по причинам, связанным с электронной верстkой.

Теперь мы должны создать заголовочный файл для DLL.

1. Выберите в главном меню пункт File | New. После открытия репозитория объектов, выберите Text для создания нового текстового файла и нажмите OK. В окне редактора кода будет отображен новый текстовый файл.
2. Выберите в главном меню File | Save. Сохраните файл как StaticLd.h. (Мы сохраняем файл в самом начале, чтобы впоследствии синтаксис был выделен. Выделение синтаксиса не работает для файлов с расширением .txt.)
3. Введите код из листинга 19.2.
4. Выберите в главном меню File | Save All.

Листинг 19.2. StaticLd.h

```
#ifndef _STATICLD_H
#define _STATICLD_H

#if defined(BUILD_DLL)
#  define DLL_EXP __declspec(dllexport)
#else
#  if defined(BUILD_APP)
#    define DLL_EXP __declspec(dllimport)
#  else
#    define DLL_EXP
#  endif
#endif

extern "C" void DLL_EXP SayHello(HWND);

class DLL_EXP MyClass {
  public:
    MyClass(HWND hwnd);
```

см. продолжение

Листинг 19.2. (продолжение)

```
    void SaySomething(char* text);
    void Beep();
  private:
    HWND hWnd;
};

#endif
```

Теперь, когда вы ввели исходный код и заголовок, можно строить проект. Для этого выберите в главном меню пункт Project | Build All.



Функции DLL не требуют для успешной компиляции предварительного объявления. Если вы обнаружите, что функция не экспортируется, проверьте, не забыли ли вы включить ее в заголовочный файл DLL.

Создание библиотеки импорта

После компиляции DLL вам нужно создать для нее библиотеку импорта (если вы не хотите использовать динамическую загрузку). Библиотека импорта — это файл, который содержит список функций, экспортируемых из DLL. Этот файл необходим компоновщику, чтобы связать ссылки на функции в объектном коде с реальными адресами в исполняемом файле. Для создания библиотеки импорта вы должны использовать утилиту командной строки IMPLIB. Сделайте следующее:

1. Откройте в Windows окно MS-DOS.
2. Перейдите в каталог, содержащий вашу DLL.
3. Наберите в командной строке следующий текст и нажмите Enter:

   ```
   \CBuilder\bin\implib MyDll.lib MyDll.dll
   ```

   (Проверьте правильность ввода пути к каталогу с C++Builder.)



Если вы раньше работали со средой программирования типа Borland C++, ручное построение библиотеки импорта может показаться не слишком удобным. Эта процедура будет усовершенствована в следующих версиях C++Builder.



При желании вы можете сконфигурировать меню Tools для запуска IMPLIB. Более подробная информация о конфигурировании меню Tools приведена в двенадцатом дне, «Вспомогательные инструменты и опции C++Builder».

В настоящий момент существует проблема с макросом $NAME, который используется в инструментах конфигурирования. Это делает невозможным вызов IMPLIB из меню Tools. По этой причине я написал простое консольное приложение, которое вызывает IMPLIB и создает .lib-файл для текущего проекта DLL. Вы можете добавить этот инструмент к меню Tools, чтобы сделать создание .lib-файлов более удобным.

Программа называется MakeLib.exe и может быть найдена на CD-ROM, прилагаемом к этой книге. Для добавления MakeLib в меню Tools сделайте следующее:

1. Скопируйте MakeLib.exe с CD-ROM в каталог \CBuilder\bin\.
2. Выберите в главном меню пункт Tools | Configure Tools.
3. Щелкните на кнопке Add для добавления нового инструмента.
4. Введите данные, как показано на рис. 19.1. Вы должны ввести тот диск и каталог, где у вас инсталлирован C++Builder.
5. Щелкните на кнопке OK для закрытия диалогового окна Tool Properties и затем нажмите кнопку Close, чтобы закрыть диалоговое окно Configure Tools.

Теперь вы можете выбирать Tools | MakeLib из основного каталога и, если проект DLL открыт, .lib-файл для вашей DLL будет создан в каталоге проекта.

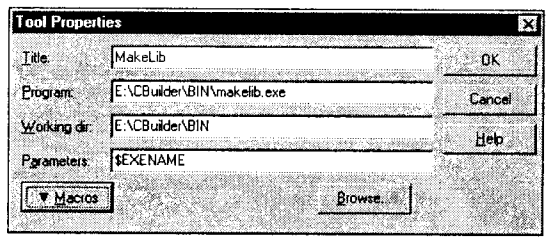


Рис. 19.1
Свойства инструмента для MakeLib.exe

Создание вызывающего приложения

Итак, вы получили работающую (будем надеяться) DLL. DLL не слишком полезны без приложения, которое будет к ним обращаться, поэтому давайте приступим к его созданию. Создание приложения, использующего DLL, требует только двух дополнительных шагов по сравнению с процедурой создания обычного приложения (предполагая статическую загрузку):

- Добавьте #include для заголовочного файла DLL.
- Добавьте библиотеку импорта DLL к проекту.

Как я говорил, при правильно созданном заголовке остальное не вызывает большого труда. Чтобы убедиться в этом, давайте создадим программу, которая использует нашу DLL.

1. Создайте новое приложение.
2. Поместите в форму кнопку и измените значение свойства Caption кнопки на Say Hello.
3. Дважды щелкните на кнопке для отображения обработчика события OnClick этой кнопки. Введите следующую строку:

```
SayHello(Handle);
```

4. Поместите в форму другую кнопку. Назовите ее MyClass Test. Дважды щелкните на кнопке и введите следующий текст:

```
MyClass mc(Application->Handle);
mc.SaySomething("Uh... Hello, there.");
mc.Beep();
```

Это позволит протестировать класс, который мы экспортируем из DLL.

5. Сохраните модуль как CDMain.cpp, а проект — как CallDll.mak.

Ну вот, вызывающее приложение почти написано. Однако перед компиляцией файла нам нужно добавить #include для заголовка DLL.

Добавление #include для заголовка DLL

Добавить #include для заголовка DLL очень просто:

1. Перейдите в верхнюю часть файла и введите следующие строки сразу под строкой #pragma hdrstop:

```
#define BUILD_APP
#include "MyDll.h"
```

2. Сохраните проект.

Это все, что нужно сделать в вызывающем приложении. Еще один шаг, и мы закончим.



Вы можете иметь несколько заголовков для одной DLL. Например, если ваша DLL экспортирует пять различных классов, вы можете написать пять заголовков, по одному для каждого класса. Только следите за тем, чтобы включить все необходимые заголовки. Впрочем, если вы забудете один, не беспокойтесь — компилятор вам напомнит.

Вы должны включать заголовок в каждый из модулей, которые используют функции или классы DLL. Теперь давайте добавим к проекту библиотеку импорта.

Добавление файла библиотеки импорта

Вы можете добавить библиотеку импорта для DLL к проекту несколькими способами. Несомненно, что самый простой из них состоит в использовании *менеджера проектов*:

1. Выберите в главном меню пункт View | Project Manager.
2. Щелкните на кнопке Add Unit (Добавить модуль) или выберите пункт Add File (Добавить файл) в контекстном меню менеджера проектов.
3. После появления диалогового окна Add to Project (Добавить к проекту) выберите файл MyDll.lib и нажмите кнопку OK.
4. Библиотечный файл будет отображен в окне менеджера проектов.

Как видите, использование DLL не очень сложно, если знать соответствующие правила.

Вы можете также добавить библиотечный файл прямо к исходному файлу проекта. Если вы просмотрите исходный файл проекта после добавления к нему .lib-файла (View | Project Source), то увидите следующую строку:

```
USELIB("MyDll.lib");
```

Когда вы добавляете .lib-файл через менеджер проектов, C++Builder автоматически вставляет эту строку в исходный файл проекта. Если хотите, можете добавить аналогичную строку самостоятельно. Однако делать это через менеджер проектов проще; кроме того, в этом случае вы можете выбрать файл, а не вводить его по памяти.

Наконец, вы можете добавить .lib-файл прямо в информационный файл проекта (project makefile). В действительности нет причин, по которым вам нужно было бы это делать, но такая возможность существует. Если вы посмотрите на информационный файл (View | Project Makefile), то увидите строку, которая выглядит примерно так:

```
LIBFILES =
```

Достаточно добавить имя библиотеки импорта в конец этой строки:

```
LIBFILES = mydll.lib
```

Как я уже сказал, самый легкий способ добавить .lib-файл состоит в использовании менеджера проектов.

Теперь все готово к компиляции и запуску программы. Щелкните на кнопке Run и проверьте ее работу. При нажатии кнопок должны выполняться функции, содержащиеся в DLL.

Использование форм в DLL

Вы можете помещать в DLL формы наравне с кодом. В действительности, создание такой формы не сильно отличается от создания формы в приложении. Сначала вы создаете новую DLL через репозиторий объектов, как и раньше. После отображении нового проекта, создайте новую форму щелчком на кнопке New Form или с помощью репозитория объектов. Добавьте в форму компоненты и напишите код для исходного модуля формы, как это делалось при создании формы в приложении.

Существуют два различных типа приложений, из которых вы можете вызывать формы, содержащиеся в DLL:

- Приложения C++Builder
- Приложения не-C++Builder

В каждой из этих ситуаций нужно поступать по-разному. Для начала давайте рассмотрим простейшую задачу: вызов формы DLL из приложения C++Builder.

Вызов формы DLL из приложения C++Builder

Если вы хотите вызывать формы, содержащиеся в DLL, вам нужно обеспечить экспорт класса формы из DLL. Для этого вы должны использовать технику экспорта классов, описанную ранее (в разделе «Экспорт функций и классов»). Говоря конкретнее, вы должны будете добавить макрос, который экспортирует или импортирует базовый класс формы в зависимости от того, создаете ли вы DLL или вызывающее приложение. Это означает необходимость ручного редактирования заголовка и исходного модуля формы.

Написание DLL и заголовка DLL

Прежде всего вам нужно добавить макрос для импорта или экспорта класса. Откройте заголовочный файл формы и добавьте перед объявлением класса код, похожий на приведенный ниже (я сказал *похожий*, поскольку вместо DLL_EXP вы можете использовать любой другой идентификатор):

```
#if defined(BUILD_DLL)
#  define DLL_EXP __declspec(dllexport)
#else
#  if defined(BUILD_APP)
#    define DLL_EXP __declspec(dllimport)
#  else
#    define DLL_EXP
#  endif
#endif
```



В данном коде используется символическое имя DLL_EXP. Я позаимствовал его из демонстрационной программы, поставляемой вместе с Borland C++. Это имя может быть любым. Например, вы можете посчитать имя BUILD_TYPE более информативным. Используйте любое имя, которое вам больше нравится.

Теперь вам нужно добавить имя макроса к объявлению класса формы. Объявление класса начинается со следующей строки:

```
class TMyForm : public TForm
```

Поместите имя макроса между ключевым словом class и именем формы:

```
class DLL_EXP TMyForm : public TForm
```

Теперь, когда заголовок готов, перейдите к исходному модулю формы. Найдите строку, включающую заголовок модуля, и добавьте над ней следующую строку:

```
#define BUILD_DLL
```

Сейчас эта секция должны выглядить примерно следующим образом:

```
#define BUILD_DLL
#include "MyForm.h"
```

Итак, у вас есть класс формы, который будет экспортироваться из DLL. Теперь можно построить саму DLL. Не забудьте после этого создать библиотеку импорта.



При желании можно не экспортировать класс формы, а поступать так же, как и при вызове формы не из приложений C++Builder (это будет обсуждаться в разделе «Вызов формы не из приложений C++Builder»). В этом случае вы должны передавать указатель на форму this той функции DLL, которая создает и отображает форму. Это обеспечит правильное порождение формы DLL главной формой вызывающего приложения.

Написание вызывающего приложения

В своем вызывающем приложении вы должны выполнить те же два шага, что и раньше, плюс третий шаг:

1. Добавьте директиву #include для заголовочного файла DLL.
2. Добавьте файл библиотеки импорта к проекту.
3. Создайте форму перед ее отображением.

Действия по включению заголовка формы и добавлению .lib-файла к проекту идентичны описанным ранее. Только убедитесь в том, что вы добавили #define BUILD_APP перед строкой, включающей заголовок формы. После этого добавьте к проекту .lib-файл.

На десятый день, «Подробнее о проектах», я рассказал об автоматическом создании форм. Поскольку форма DLL не является частью проекта вызывающего приложения, она не может быть автоматически создана при загрузке программы. Вы должны явно создать форму перед ее отображением. Я уже рассказывал об этом, но в другом контексте, поэтому давайте повторим еще раз. Код для отображения и запуска формы выглядит следующим образом:

```
TMyForm* form = new TMyForm(this)
form->ShowModal();
delete form;    // не обязательно
```

Помимо того факта, что вы должны создать форму перед ее отображением, между формой в файле приложения и формой в DLL с точки зрения вызывающего приложения разницы не существует.



В приведенном фрагменте кода указатель на форму уничтожается после окончания ее работы. Это не требуется, но при желании вы можете стереть указатель, чтобы освободить память. Если вы не уничтожите форму, это сделает вызывающая форма, когда сама будет уничтожена (обычно при завершении работы приложения). Когда вы указываете this в конструкторе формы, вы делаете форму потомком главной формы, а объекты-предки VCL несут ответственность за уничтожение порожденных ими объектов.

Вызов формы не из приложения C++Builder

Если вы хотите вызывать форму из других приложений, потребуется немного больше работы. Поскольку вызывающее приложение ничего не знает о VCL, вы не можете импортировать класс VCL и вызвать его метод ShowModal(), как это делалось при вызове формы из приложения C++Builder. Необходимо создать в DLL автономную функцию, которая будет вызываться приложением. Эта функция может быть вызвана из любой программы, если вы объявите ее с модификатором extern "C". Затем внутри тела функции вы сможете создать и отобразить форму.

Форма в DLL создается так, как описано раньше, за исключением того, что вы не должны экспортировать класс формы. Вместо этого в исходном модуле главной формы должна быть определена функция наподобие следующей:

```
void ShowForm()
{
  TMyForm* form = new TMyForm(0);
  form->ShowModal();
  delete form;    // это необходимо сделать
}
```

Обратите внимание, что предку формы передается NULL. Причина этого в том, что другие приложения не имеют объекта VCL, который может быть предком формы. В этом случае вы должны уничтожить объект TMyForm, поскольку форма не имеет порождающего объекта VCL.

Затем вы должны создать отдельный заголовок для DLL. Помните, что класс формы не будет экспортироваться, поэтому заголовок формы не имеет значения для вызывающего приложения. Однако все вызывающие приложения хотят видеть в заголовке функцию ShowForm(). Создайте заголовок, содержащий объявление функции ShowForm(), а затем включите его в исходный файл вызывающего приложения. (Не забудьте написать макрос для обеспечения правильного импорта и экспорта функции.) Как и раньше, вы должны будете скомпоновать .lib-файл для DLL с вызывающим приложением при его построении. При желании вы можете также использовать динамическую загрузку. Используемый метод зависит от ваших предпочтений и языка, на котором написано вызывающее приложение.

Использование ресурсов DLL

Иногда удобно помещать в DLL ресурсы. Я уже упоминал о многоязыковой поддержке и о том, как можно использовать DLL для упрощения перевода ваших программ на другие языки. Пусть, например, у вас есть экран с инструкциями и эти инструкции занимают пять строк в DLL. Строки могут иметь имена IDS_INSTRUCTION1, IDS_INSTRUCTION2 и т.д. Вы могли бы загружать и отображать эти строки следующим образом:

```
char buff[256];
LoadString(dllInstance, IDS_INSTRUCTION1, buff, sizeof(buff));
InstructionLabel1->Caption = buff;
```

Первый аргумент функции LoadString() содержит дескриптор экземпляра (HINSTANCE) того модуля, в котором находится строка. Вы могли бы создать DLL на нескольких языках и загружать нужную DLL в зависимости от выбора пользователя. Соответствующий код мог бы выглядеть следующим образом:

```
String dllName;
switch (language) {
  case french : dllName = "french.dll"; break;
  case german : dllName = "german.dll"; break;
  case spanish : dllName = "spanish.dll"; break;
  case english : dllName = "english.dll"; break;
}
dllInstance = LoadLibrary(dllName.c_str());
```

Вы должны только загрузить нужную DLL; остальной код остается без изменения. Это лишь один пример использования ресурсов, содержащихся в DLL. Вы можете найти много применений этой технике.

Создание DLL ресурсов

Вы можете создавать DLL, которые содержат только ресурсы, или смешивать код и ресурсы в одной и той же DLL. В этом отношении ограничений нет.

Если ваша DLL содержит только ресурсы, следует удалить нижеприведенные строки из исходного модуля DLL, создаваемого по умолчанию:

```
#include <vcl\vcl.h>
#pragma hdrstop

USERES("MyDll.res");
```

Можно также удалить из DLL функцию DllEntryPoint(), в зависимости от того, собираетесь ли вы загружать DLL статически или динамически (подробнее об этом будет сказано позже). Вы должны оставить #include для файла Windows.h, если удаляется #include для vcl.h.

Размещение ресурсов в DLL по существу то же самое, что и добавление ресурсов к приложению. В менеджере проектов, щелкните на кнопке Add Unit и добавьте имя сценария ресурсов (.rc) или двоичного файла ресурсов (.res), содержащего необходимые ресурсы. Эти ресурсы будут присоединены к DLL при ее построении.

При статической загрузке ресурсов (во время загрузки приложения) вы должны выполнить еще один шаг. Как было видно из предыдущего фрагмента кода, перед доступом к ресурсам DLL необходимо получить дескриптор экземпляра этой DLL. При динамической загрузке это делается довольно просто, поскольку дескриптор экземпляра возвращает функция LoadLibrary():

```
int dllInstance = LoadLibrary("resource.dll");
```

После этого вы можете использовать дескриптор экземпляра везде, где он требуется. В данном случае использование динамической загрузки более предпочтительно. При статической загрузке вы должны каким-то способом получить дескриптор экземпляра DLL (помните, что DLL будет загружена при запуске приложения). Для подготовки к статической загрузке вы должны выполнить следующие действия при создании DLL:

1. Объявить в DLL глобальную переменную, которая будет содержать дескриптор экземпляра.
2. В функции DllEntryPoint() присвоить дескриптор экземпляра DLL глобальной переменной.
3. Написать функцию, возвращающую дескриптор экземпляра вызывающему приложению.

Вместо подробного объяснения каждого шага, позвольте мне привести сразу полный код для DLL, использующей ресурсы (см. листинг 19.3).

Листинг 19.3. Resource.cpp

```
 1: //---------------------------------------------------------------------
 2: #include <window.h>
 3: #pragma hdrstop
 4:
 5: #define BUILD_DLL
 6: #include "resource.h"
 7:
 8: HINSTANCE hInstance;
 9:
10: //---------------------------------------------------------------------
```

см. продолжение

Листинг 19.3. (продолжение)

```
11: int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long, void*)
12: {
13:   hInstance = hinst;
14:   return 1;
15: }
16: //---------------------------------------------------------------------
17:
18: HINSTANCE GetDllInstance()
19: {
20:   return hInstance;
21: }
```

Как видите, я удалил многое из того, что по умолчанию содержалось в исходном файле DLL C++Builder после его создания с использованием репозитория объектов. Обратите внимание, что в строках 5 и 6 я включил заголовочный файл для DLL. В этом случае заголовок будет содержать объявление только одной функции. Затем, в строке 8, я объявил глобальную переменную hInstance. В строке 13 этой переменной присваивается дескриптор экземпляра, передаваемый функции DllEntryPoint().



Дескриптор экземпляра DLL создается Windows и передается функции DllEntryPoint() при загрузке DLL.

Наконец, строки с 18 по 21 содержат функцию с именем GetDllInstance(), которая возвращает дескриптор экземпляра DLL вызывающему приложению. Эта функция может быть использована в вызывающем приложении каждый раз, когда требуется дескриптор экземпляра DLL:

```
LoadString(
  GetDllInstance(), IDS_TEXT, buff, sizeof(buff));
```

Приведение типа HINSTANCE (возвращаемого DLL) к типу int требуется в связи с тем, что VCL и Windows обрабатывают тип HINSTANCE по-разному.

Не забывайте создавать статическую библиотеку для вашей DLL при использовании статической загрузки. Если вы не используете статическую загрузку, создание .lib-файла не требуется.

Использование ресурсов DLL

В действительности, осталось сказать совсем немного. Я уделил достаточно внимания использованию ресурсов, содержащихся в DLL, когда речь шла о создании самих DLL ресурсов. Тем не менее, кратко повторим основные положения еще раз. Вы знаете, что можно загружать DLL ресурсов статически или динамически. При статической загрузке для получения доступа к ресурсам вы должны использовать экспортируемую функцию, чтобы получить дескриптор экземпляра DLL. При динамической загрузке вы можете получить дескриптор экземпляра через функцию LoadLibrary(). Не забывайте вызывать FreeLibrary() по окончании работы с DLL или перед закрытием приложения.

Для DLL ресурсов динамическая загрузка проще, чем статическая, поскольку при желании вы можете вообще не писать никакой код. Она также проще потому, что вам не требуется создавать для DLL библиотеку импорта. Достаточно только использовать LoadLibrary(), и динамическая библиотека будет в вашем распоряжении.



Использование динамической загрузки имеет дополнительное преимущество в виде ускорения загрузки ваших приложений. Во многих случаях DLL ресурсов загружаются только при необходимости, и выгружаются, когда потребность в них отпадает. В результате ваше приложение использует меньше памяти, чем при размещении ресурсов в исполняемом файле. Недостаток по сравнению со статической загрузкой состоит в том, что ваши пользователи могут заметить небольшую паузу во время загрузки DLL ресурсов. Постарайтесь по возможности опередить события и загрузить DLL в тот момент, когда это будет наименее заметно.

При использовании статической загрузки вы не должны забывать о добавлении .lib-файла к проекту вызывающего приложения, а также включать директиву #include для заголовка DLL. Преимущество статической загрузки состоит в том, что вам не нужно беспокоиться о LoadLibrary()/FreeLibrary(). Недостаток — немного больше работы для инициализации. Ваши программы также будут медленнее загружаться и занимать больше места в памяти.



Помните JumpingJack из 9-го дня, «Создание приложений в C++Builder»? CD-ROM, прилагаемый к этой книге, содержит версию JumpingJack, которая загружает видео, звуковые и строковые ресурсы из DLL. Разберите эту программу, как пример использования ресурсов DLL.

Заключение

Использование динамически компонуемых библиотек не слишком сложно, но оно может довести до исступления, пока вы не получите правильную комбинацию импорта и экспорта. Это делается проще с помощью ключевого слова __declspec, которое описано в этой главе. Помещение форм VCL в DLL и последующий их вызов из приложений, созданных не в C++Builder, является мощной возможностью. Это означает, что вы можете создавать формы, которые другие смогут вызывать практически из любых приложений Windows, независимо от того, написаны ли они с помощью OWL, MFC, обычного Си, Visual Basic, Delphi и так далее. Использование ресурсов DLL эффективно в том случае, когда ваше приложение содержит много ресурсов и вы хотите управлять тем, когда и куда эти ресурсы загружаются.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

У меня получилась относительно небольшая программа и я не вижу реальной необходимости использовать DLL. Должен ли я пересмотреть структуру программы?

Возможно, нет. Для маленьких приложений DLL часто не нужны. Если у вас есть классы, которые могут использоваться многократно, вы можете успешно использовать DLL, но пусть поиски возможностей использования DLL в небольшом приложении не сбивают вас с собственного пути.

Я начинаю работать над достаточно большим классом. Следует ли мне планировать его помещение в DLL?

Независимо от того, дойдет ли дело в конце концов до DLL, вам всегда следует писать класс, имея в виду его многократное использование. Если вы будете делать это с самого начала, последующее помещение класса в DLL будет представлять собой тривиальную задачу.

Эта GetProcAddress() немного странновата. Должен ли я использовать ее для вызова функций из DLL?

Нет. Достаточно использовать статическую загрузку, и вам не придется беспокоиться о функциях LoadLibrary(), GetProcAddress() и FreeLibrary(). Использование статической загрузки потребует для вашей DLL библиотеки импорта (.lib), но в конечном счете это намного проще.

Моя программа превосходно компилируется, но я получаю от компоновщика сообщение об ошибке Undefined external MyFunction() in module MyUnit.cpp (Не определена внешняя MyFunction() в модуле MyUnit.cpp). Что неправильно?

Либо вы не экспортировали функцию из DLL, либо неправильно импортировали ее из DLL. Возможно, вы просто забыли добавить .lib-файл для DLL в проект или сделали ошибку в заголовке вашей DLL, и что-то оказалось неправильно при построении DLL или вызывающего приложения.

Моя DLL содержит форму, а мое вызывающее приложение является программой C++Builder. Размер DLL довольно велик. Можно ли как-нибудь его уменьшить?

К сожалению, нет. В этом случае мощь программирования в C++Builder работает немного против вас. Вызывающее приложение и DLL содержат некоторое количество одинакового кода VCL. Иными словами, код дублируется в .exe и .dll. Если вы используете формы в DLL, вам остается только смириться с большим размером программы.

Мое приложение содержит большое количество звуковых файлов. Сейчас все они находятся в отдельных файлах, но я бы хотел поместить их в один файл. Подходит ли здесь DLL ресурсов?

Безусловно. Единственным недостатком является то, что ради одного звукового файла вам нужно загружать всю DLL. Однако при использовании динамической загрузки вы можете загружать и выгружать DLL по мере необходимости. Функция PlaySound() делает воспроизведение звука из DLL достаточно простым.

Я живу во франко-говорящей стране. Почему я должен беспокоиться о многоязыковой поддержке в своей программе?

Это полностью зависит от того, на кого рассчитано ваше приложение. Если вы точно знаете, что программа никогда не будет использоваться людьми в тех странах, где французский не является основным языком, вы можете вообще не беспокоиться о многоязыковой поддержке. Но если есть даже малейшая возможность, что ваша программа будет продаваться в других странах, вы должны планировать многоязыковую поддержку с самого начала. Намного легче сделать это сразу, чем вносить исправления потом.

Будет ли C++Builder автоматически создавать файл библиотеки импорта при создании DLL?

Нет. Для этого вы должны обработать DLL утилитой IMPLIB.

Вопросы

1. Как загрузить DLL статически?
2. Как загрузить DLL динамически?
3. Как вызвать функцию из DLL, которая загружена статически?
4. Какие действия необходимо предпринять, чтобы гарантировать вызов функции, содержащейся в DLL, извне?
5. Можете ли вы выгрузить динамически загруженную DLL в любое время или только при закрытии программы?
6. Что вы должны сделать, чтобы вызвать форму C++Builder, содержащуюся в DLL, из приложения, созданного не в C++Builder?
7. Каковы преимущества использования макроса импорта/экспорта при построении DLL и вызывающих их приложений?
8. Как добавить ресурсы к DLL?
9. Должна ли DLL ресурсов обязательно содержать код наравне с ресурсами?
10. Может ли DLL, содержащая ресурсы, быть загружена статически (при загрузке программы)?

Упражнения

1. Создайте DLL, которая содержит функцию, отображающую при вызове окно сообщения.
2. Создайте приложение, вызывающее DLL из упражнения 1.
3. Создайте DLL, содержащую форму, и вызывающее приложение, которое будет отображать эту форму.
4. Создайте DLL, которая содержит два ресурса в виде растровых изображений.
5. Создайте программу, которая будет отображать по требованию любой из двух видеоресурсов вашей новой DLL. (**Указание**: используйте компонент TImage и метод TLoadFromResource().)
6. **Дополнительное задание**: Напишите пять DLL, каждая из которых содержит строки одинакового содержания, но на разных языках.
7. **Дополнительное задание**: Напишите вызывающее приложение, которое отображает строки, созданные в упражнении 6. Дайте возможность пользователю выбрать язык, который будет использоваться в приложении. Выведите строки на выбранном языке.

Создание компонентов



Кент Рейсдорф

C++Builder предоставляет вам для использования в своих приложениях большое разнообразие компонентов. К ним относятся как базовые элементы управления Windows, так и специальные компоненты, не относящиеся к самой Windows. Тем не менее, вы можете оказаться в ситуации, когда для решения стоящей перед вами задачи будет недостаточно предустановленных компонентов. В таком случае вы можете написать компонент самостоятельно. Для создания компонента нужно выполнить следующие действия:

1. Использовать мастер компонентов (Component Wizard) для начала создания компонента (или создать компонент с «чистого листа», используя пустой файл редактора).
2. Добавить свойства, методы и события к классу компонента.
3. Протестировать компонент.
4. Добавить компонент в палитру компонентов.

Сегодня вы научитесь создавать компоненты. Как и большая часть того, с чем вы встречались в C++Builder, это не так сложно после некоторой тренировки.

Предварительные сведения

Написание компонентов требует более глубокого знания программирования, чем требовалось вам до сих пор. Прежде всего, вы должны создать класс для вашего нового компонента. Класс должен быть разработан так, чтобы некоторые его свойства могли отображаться в инспекторе объектов (предполагается, что вы пишете визуальный компонент). Затем вы должны придать компоненту свойства. В дополнение, вам потребуется написать для вашего компонента методы. Некоторые из них будут закрытыми, другие — открытыми для обеспечения к ним доступа пользователей. Наконец, вы должны будете написать для компонента события.

Как видите, создание компонента требует некоторой работы. Несмотря на все свои достоинства, визуальная среда программирования C++Builder не сможет вам здесь ничем помочь. Написание компонентов — это программирование в чистом виде.

Существуют два основных способа создания компонентов:

- Модификация существующего компонента
- Написание компонента заново

Выбор метода зависит от типа компонента, который вы пишете. Допустим, что вы хотите создать компонент label, который выводит мигающий текст (назовем этот компонент FlashingLabel). В этом случае стандартный компонент Label содержит все, что вам нужно для начала, поэтому можно начать с TLabel и изменить его для создания эффекта мигания. Если, с другой стороны, вы хотите написать компонент label, который позволит создавать ярлыки Windows, вам следует писать его заново, поскольку VCL не содержит компонента, который можно было бы использовать в качестве базового.



C++Builder предоставляет вам несколько классов, которые можно использовать в качестве базовых для новых компонентов. Имена этих классов начинаются с TCustom. Например, базовым классом для TLabel является TCustomLabel. Ваши собственные элементы управления могут быть производными одного из этих классов. Пользовательские классы уже содержат свойства, которые с наибольшей вероятностью потребуются для компонента соответствующего типа (в данном случае компонента пользователя label). Все, что вы должны сделать — это переобъявить свойства в разделе published объявления класса вашего компонента. Это будет подробно объяснено далее, в разделе «TFlashingLabel: пример компонента».

Для изменения существующего компонента используется мастер компонентов. Он вызывается через репозиторий объектов. Для запуска мастера компонентов выберите в главном меню C++Builder пункт File | New. После отображения репозитория объектов дважды щелкните на значке Component, и мастер компонентов появится на экране. После этого вы можете выбрать компонент, который будет использован в качестве базового. (Мастер компонентов подробно рассматривается в соответствующем разделе.) Если вы выберете имя класса, который уже существует в библиотеке компонентов, мастер компонентов выдаст предупреждение. Вы должны будете выбрать уникальное имя класса перед тем, как продолжить работу.



Нет никакой причины начинать имя класса с T; это обычно свойственно только классам фирмы Borland. (Использование T в классах Borland является традицией, идущей со времен появления Турбо-Паскаля. Эта буква используется в OWL и VCL.) Некоторые используют T при создании классов на основе классов Borland, но не используют ее в своих собственных классах. Можете поступать по своему усмотрению.

Поле Ancestor Type (Тип предка) мастера компонентов используется для указания базового класса компонента. Это поле представляет собой комбинированный список, содержащий имена всех классов, зарегистрированных в библиотеке компонентов (около 150 классов, инсталлированных по умолчанию, плюс добавленные вами). Вы можете выбрать класс из списка или сразу ввести имя, если вы уже его знаете. Выбор имени базового класса из списка предпочтительнее, поскольку это уменьшает вероятность ошибки при вводе.

Изменение существующих компонентов

Для изменения существующего компонента используется возможность Си++, называемая *наследованием (inheritance)*. Прошло немало времени с тех пор, как я говорил о наследовании, поэтому при необходимости обратитесь к главе «Классы и объектно-ориентированное программирование», которая изучалась на четвертый день. По существу, наследование от компонента означает, что вы берете все его свойства и добавляете что-то свое. Класс, от которого производится наследование, называется *базовым классом (base class)*, а новый класс — *производным классом (derived class)*. В предыдущем примере компонент TLabel был бы базовым классом, а TFlashingLabel — производным классом. Создавая компонент на основе уже существующего, вы можете использовать мастер компонентов для начальной генерации исходного файла и заголовка (подробнее об этом чуть позже).

Создание компонентов заново

Изменение существующих компонентов является самым легким способом создания нового компонента. Но иногда вы должны создавать компонент с самого начала, не пользуясь теми преимуществами, которые дает наследование. Если вы создаете компонент заново, требуется выполнить следующее:

1. Создать заголовок, содержащий объявление класса компонента.
2. Создать модуль исходного кода для компонента.
3. Зарегистрировать компонент в системе C++Builder.

Затем вы можете продолжить написание компонента, протестировать его и добавить к палитре компонентов. Новый исходный файл и заголовок создаются обычным образом.



Если вы должны создавать компонент заново, можете использовать заголовок существующего компонента в качестве образца. Скопируйте объявление одного из классов VCL и вставьте его в новое окно редактора. Уберите весь код, сохранив основную структуру объявления.

Регистрация компонента

Регистрация компонента необходима для того, чтобы C++Builder знал, какие компоненты помещать в библиотеку компонентов. Этот шаг выполняется автоматически, когда вы создаете новый компонент с помощью мастера компонентов. При создании компонента вручную, вы должны зарегистрировать его самостоятельно.

Для регистрации компонента поместите в его исходный код функцию Register(). Типичная функция Register() выглядит следующим образом:

```
namespace Mycomponent {
void __fastcall Register()
{
  TComponentClass classes[1] = {__classid(TMyComponent)};
  RegisterComponents("Samples", classes, 0);
}
}
```

Прежде всего, обратите внимание, что функция Register() помещена в *пространство имен*.

Пространство имен (namespace) — это элемент Си++, позволяющий различать блоки кода по именам.

Пространства имен являются элементами Си++, о которых я еще не говорил. Пространство имен используется для отделения данной функции Register() от функций Register() других компонентов, входящих в библиотеку компонентов.



Имя блока для компонента C++Builder должно быть образовано по специальным правилам. Оно состоит из имени файла, в котором содержится компонент, набранного в нижнем регистре за исключением первого символа. Пусть, например, у вас есть компонент TFancyEdit, содержащийся в файле FancyEdit.cpp. Имя блока для функции Register() в этом случае будет Fancyedit.

При использовании мастера компонентов вы не должны беспокоиться об имени блока. Оно будет создано автоматически при сохранении файла и будет изменяться, если вы измените имя файла с помощью File | Save As.

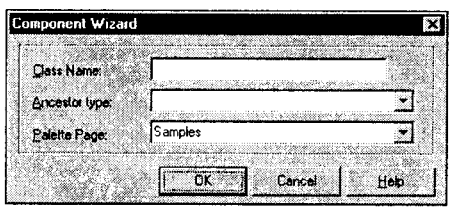
В первой строке функции Register() создается массив компонентов, подлежащих регистрации, после чего компонент помещается в этот массив. Если вы создаете целую библиотеку компонентов, их можно зарегистрировать за один раз, поместив в массив TComponentClass. Однако в большинстве случаев вы будете работать с компонентами по отдельности.

Вторая строка функции Register() сообщает C++Builder, на какой вкладке палитры компонентов должен быть размещен значок нового компонента. После того, как компонент написан и полностью протестирован, вы можете добавить его в библиотеку компонентов. Помещение компонента в библиотеку будет подробно рассмотрено в разделе «Добавление компонента в библиотеку компонентов».

Мастер компонентов

Мастер компонентов (Component Wizard) дает возможность начать написание компонента. Он вызывается из репозитория объектов. Чтобы запустить мастер компонентов, выберите File | New для отображения репозитория объектов и затем дважды щелкните на значке Component. На экране появится диалоговое окно мастера компонентов, показанное на рис. 20.1.

Рис. 20.1
Диалоговое окно мастера компонентов



Введите имя вашего класса в поле Class Name. Имя класса следует начинать с T и делать его достаточно информативным. Например, возвращаясь к приведенному ранее примеру, если у вас есть компонент, отображающий мигающий текст, назовите его TFlashingLabel.



Вы можете быстро перейти к определенному имени класса в комбинированном списке Ancestor Type. Сначала щелкните на кнопке со стрелкой для вызова списка имен классов, а затем наберите первые два-три символа имени класса, который вам нужен. Список имен классов будет прокручиваться по мере ввода символов.

Поле Palette Page позволяет вам указать вкладку палитры компонентов, на которой должен появиться значок компонента. (Значок компонента не будет отображаться, пока вы не инсталлируете компонент.)

Как вы могли догадаться, сейчас мы переходим к написанию простого компонента. Мы создадим компонент TFlashingLabel, который я использовал в качестве примера в разделе «Предварительные сведения». Этот компонент представляет собой обычный компонент Label, который отображает на экране мигающий (или вспыхивающий, если вам больше нравится) текст. Далее я расскажу вам о деталях, но пока вы должны понять основную идею того, что мы попытаемся осуществить. Итак, давайте начнем создание компонента.

Создание компонента

В данном случае мы собираемся создавать компонент на основе существующего компонента. Разумеется, мы используем для этого мастер компонентов. Выполните следующие действия:

1. Выберите в главном меню пункт File | New для вызова репозитория объектов.
2. Дважды щелкните на значке Component в репозитории объектов. На экране появится диалоговое окно мастера компонентов.
3. Введите TFlashingLabel в поле Class Name.
4. В комбинированном списке Ancestor Type выберите класс TCustomLabel, который будет служить в качестве базового класса.
5. Поле Palette Page содержит Samples. Оставьте это поле без изменений. Новый компонент при инсталляции будет добавлен к вкладке Samples палитры компонентов.
6. Нажмите кнопку OK, чтобы закрыть диалоговое окно мастера компонентов. Редактор кода переместится на передний план и будет содержать новый модуль исходного кода.
7. Сохраните модуль под именем FlashingLabel.cpp.



Вам следует сразу же сохранить модуль, сгенерированный мастером компонентов. При этом будет обновлена ссылка на заголовок модуля, которая служит идентификатором пространства имен.

Листинги 20.1 и 20.2 содержат исходный модуль и заголовок в том виде, который они имеют на данный момент.

Листинг 20.1. FlashingLabel.h

```
//---------------------------------------------------------------------------
#ifndef FlashingLabelH
#define FlashingLabelH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
#include <vcl\StdCtrls.hpp>
//---------------------------------------------------------------------------
class TFlashingLabel : public TCustomLabel
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

Листинг 20.2. FlashingLabel.cpp

```
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "FlashingLabel.h"
//---------------------------------------------------------------------------
static inline TFlashingLabel* ValidCtrCheck()
{
  return new TFlashingLabel(NULL);
}
//---------------------------------------------------------------------------
__fastcall TFlashingLabel::TFlashingLabel(TComponent* Owner)
  : TLabel(Owner)
{
}
//---------------------------------------------------------------------------
namespace Flashinglabel
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TFlashingLabel)};
    RegisterComponents("Samples", classes, 0);
  }
}
//---------------------------------------------------------------------------
```

Из листинга 20.1 видно, что класс TFlashingLabel является производным от TCustomLabel. Объявление класса содержит только ключевые слова (private, public, protected и __published). Мы заполним пустые места немного позже, когда рассмотрим устройство компонента подробнее.

Как видите, мастер компонентов предоставляет вам заготовку, заполняя часть заголовка и исходного модуля. Вам по-прежнему остается самая трудная

работа, но по крайней мере, функция Register() уже написана и объявление класса начато.

Перед тем, как продолжить, я должен посвятить некоторое время объяснению того, из чего состоит компонент. После этого мы вернемся к написанию компонента TFlashingLabel. Держите только что созданный компонент под рукой, потому что он скоро вам понадобится.

Написание свойств

Написание свойств является значительной частью всей работы по созданию компонента. До этого момента вы лишь использовали свойства и знаете о них как пользователь. Теперь вам нужно знать свойства с точки зрения создателя компонента. Прежде чем вы сможете написать компонент, вы должны понимать, чем являются и чем не являются свойства.

Прежде всего, свойства — это не члены-данные. Естественно думать о свойствах как о членах-данных класса, к которому они принадлежат. Кроме того, вы рассматриваете свойства как члены-данные при выполнении действий типа следующих:

```
int width = Width;
Height = width * .7;
```

Но свойства не являются членами-данными, и вам нужно иметь это в виду при написании компонентов. Свойства во многом отличаются от членов-данных, но у них есть по крайней мере одна общая черта: определенный тип. Тип свойства может быть одним из встроенных типов (int, long, double, enum и т.д.), классом (String, TFont и т.д.) или структурой (например, TRect).

Таким образом, свойства являются объектами специального типа, удовлетворяющими следующим критериям:

- ☐ Свойства имеют базовые члены-данные, хранящие значения этих свойств.
- ☐ Свойства могут включать метод записи.
- ☐ Свойства могут включать метод чтения.
- ☐ Свойства могут использовать прямой доступ вместо методов записи и чтения.
- ☐ Свойства могут быть доступны только для записи или только для чтения.
- ☐ Свойства могут иметь значения по умолчанию.
- ☐ Свойства могут быть отображаемыми или неотображаемыми.

Чтобы лучше понять все это, давайте рассмотрим характерные черты свойств по отдельности.

Свойства имеют базовые члены-данные

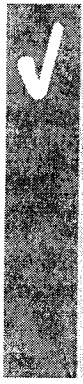
С каждым свойством связан соответствующий член класса, хранящий значение этого свойства. Рассмотрим, например, простое присваивание:

```
Label->Caption = "McGarry & Mahar";
```

Этот оператор присваивает строку свойству Caption компонента Label. То, что при этом происходит — больше, чем просто присваивание. Поскольку

свойство Caption имеет тип String, оно имеет в своей основе объект String. Именно этот объект получает значение присваиваемой строки. Использование базовых членов-данных необходимо, поскольку свойство само по себе не способно хранить данные.

Базовые члены-данные могут иметь любые имена, но по традиции их имена образуются из имени соответствующего свойства и буквы F впереди. Например, член, связанный со свойством Caption, был бы назван FCaption.



Связь между свойствами и их базовыми членами-данными может стать источником многих недоразумений, когда вы начнете писать компоненты. Дело не в трудности понимания, а просто в том, что вы можете перепутать имена. Например, вы можете случайно написать

```
Left = 20;
```

имея в виду

```
FLeft = 20;
```

Это может приводить к самому любопытному поведению ваших компонентов. Вы увидите, почему это так, когда я буду рассматривать методы записи в следующем разделе.

Свойства имеют методы записи

Когда вы присваиваете свойству значение, многое остается за кадром. Что именно происходит, зависит от конкретного свойства. Например, данный код выглядит очень просто:

```
Left = 20;
```

Но при выполнении этой строки происходит несколько вещей. Прежде всего, член FLeft получает новое значение. Затем форма (предполагая, что код выполняется для формы) перемещается в новое положение с помощью функции MoveWindow(), входящей в API Windows. Наконец, форма перерисовывается функцией Invalidate(). Все это осуществляется через метод записи свойства Left. Метод записи — это метод, который вызывается при каждом присваивании свойству значения. Вы можете использовать метод записи для проверки правильности присваиваемого значения или выполнения специальных действий.

Метод записи объявляется при объявлении свойства. Вот пример типичного объявления свойства:

```
__property int FlashRate = {read=FFlashRate, write=SetFlashRate};
```

В этом объявлении используется синтаксис, с которым вы еще не встречались, поскольку он предназначен специально для свойств и не существует в стандартном Си++. Прежде всего, обратите внимание на ключевое слово __property и следующий за объявлением знак равенства с дополнительным кодом. Этот код сообщает компилятору, что метод записи для свойства называется SetFlashRate(). Вы можете дать методу записи любое имя, но по традиции это должно быть имя свойства с предшествующим словом Set. (Обратите внимание, что в этом объявлении также присваивается значение методу чтения. Я перейду к методу чтения чуть позже.)

Когда происходит запись в свойство (присваивание ему значения), автоматически вызывается связанный с этим свойством метод записи. Метод записи должен возвращать void. Он имеет один параметр, и этот параметр должен иметь тот же тип, что и само свойство. Например, объявление метода записи для свойства FlashRate может выглядеть следующим образом:

```
void __fastcall SetFlashRate(int AFlashRate);
```

Значение, передаваемое методу записи, является тем значением, которое присваивается свойству. Таким образом, в результате выполнения строки

```
FlashingLabel->FlashRate = 1000;
```

значение 1000 будет передано функции SetFlashRate(). Что вы будете делать с этим значением внутри метода записи, зависит от многих факторов. Как минимум, вы должны присвоить это значение соответствующему члену. Иными словами, метод записи должен выглядеть примерно следующим образом:

```
void __fastcall TFlashingLabel::SetFlashRate(int AFlashRate)
{
  FFlashRate = AFlashRate;    // теперь FFlashRate равно 1000
  // другой код
}
```



Использование буквы A перед именем параметра в методе записи также является традицией.

Вы почти всегда будет делать больше, чем просто присваивать значение соответствующему члену. Если же вы только присваиваете значение, метод записи для свойства вообще не нужен. Я объясню это чуть позже, а пока давайте рассмотрим методы чтения.



Аргумент может быть передан методу записи по значению или по ссылке. Структуры и классы следует передавать по ссылке, чтобы уменьшить накладные расходы. Встроенные типы данных можно передавать по значению.

Свойства имеют методы чтения

Метод чтения работает точно так же, как метод записи (за исключением очевидного различия). Он выполняется при считывании значения свойства и возвращает значение соответствующего члена.

Имя метода чтения образуется из имени свойства добавлением впереди слова Get. Метод чтения не принимает аргументы и возвращает значение, тип которого совпадает с типом свойства. Например, если вы хотите использовать метод чтения для свойства FlashRate, его следует объявить таким образом:

```
int __fastcall GetFlashRate()
```

Метод чтения может выполнять некоторые действия и только потом возвращать значение базового члена, в данном случае FFlashRate. Чтение свой-

ства имеет место во многих случаях. Иногда это результат присваивания значения переменной:

```
int rate = FlashingLabel->FlashRate;
```

В другой раз оно используется внутри оператора Си++:

```
switch (FlashingLabel->FlashRate) {
  case 1000 : SpeedUp(); break;
  // и т.д.
}
```

Метод чтения вызывается при каждом считывании значения свойства, независимо от того, как это происходит.

Обратите внимание на слово *если*, которое я произнес, говоря об использовании метода чтения. Часто вам вообще не потребуется метод чтения. Вместо него вы будете использовать для получения значение соответствующего члена прямой доступ. К его рассмотрению мы сейчас и перейдем.

Свойства могут использовать прямой доступ

Можно вообще не использовать для свойств методы чтения и записи. Если вы просто присваиваете значение базовому члену или считываете его значение, достаточно использовать прямой доступ. Объявление свойства, использующего прямой доступ, выглядит следующим образом:

```
__property int FlashRate = {read=FFlashRate, write=FFlashRate};
```

В этом коде вместо вызова метода чтения или записи спецификаторам read и write присваивается сам член FFlashRate. При записи происходит изменение этого члена и ничего больше. При чтении возвращается значение базового члена. Все очень просто.



Обычно для чтения свойства используется прямой доступ, а для записи — метод записи. Запись в свойство часто сопровождается дополнительными действиями, о которых я говорил в предыдущем разделе. Фактически, способность порождать целый ряд дополнительных действий является сильной стороной свойств. Для выполнения этих действий при записи, вы должны использовать соответствующий метод. Результатом чтения свойств, с другой стороны, обычно является только возврат значений базовых членов-данных. В этом случае использование прямого доступа имеет больший смысл.

Свойства могут быть доступны только для записи или только для чтения

Вы можете установить для свойства тип read-only или write-only. Возможность сделать свойство доступным только для чтения очень полезна. (VCL содержит очень много свойств типа read-only.) Например, вы можете иметь свойство, значение которого пользователю разрешено считывать, но не изменять. Прямая модификация свойства может повлечь нежелательные последствия для компонента, поэтому вам следует принять меры против этого.

Сделать свойство доступным только для чтения легко: вы просто опускаете спецификатор записи в объявлении свойства:

```
__property int FlashRate = {read=FFlashRate};
```

Если пользователь предпримет попытку записи в свойство, объявленное как read-only, компилятор выдаст сообщение об ошибке: TFlashingLabel::FlashRate is not accessible (TFlashingLabel::FlashRate недоступен). Как видите, сделать свойство доступным только для чтения действительно очень просто.

Вы можете сделать свойство доступным только для записи, опустив спецификатор чтения в объявлении свойства. Трудно представить себе использование свойства, которое можно только записывать, но не считывать, но вы можете написать такое свойство, если когда-либо в этом возникнет необходимость.

Свойства могут иметь значения по умолчанию

Значение по умолчанию (default value) — другая полезная особенность свойств. Вы можете заметить, что при помещении компонента в форму многие свойства, отображаемые в окне инспектора объектов, уже имеют определенные значения. Это значения по умолчанию, которые были определены разработчиком компонента. Присваивание значений по умолчанию облегчает жизнь пользователю этого компонента. Все свойства по возможности должны иметь значения по умолчанию. Это дает возможность пользователю изменять только те свойства, которые он (или она) хочет, и не трогать остальные. Большинство свойств имеет значения по умолчанию, но для свойств некоторых типов такие значения отсутствуют.

Подобно методам чтения и записи, значение по умолчанию устанавливается при объявлении свойства. Давайте вернемся к свойству FlashRate нашего компонента. Объявление свойства FlashRate со значением по умолчанию выглядит следующим образом:

```
__property int FlashRate =
    {read=FFlashRate, write=SetFlashRate, default=800};
```

Теперь при отображении компонента FlashingLabel в инспекторе объектов для свойства FlashRate уже будет установлено значение 800 (в данном случае это миллисекунды).



Установка значения по умолчанию для свойства приводит только к отображению этого значения в инспекторе объектов. Она не устанавливает значение базового члена. Вы по-прежнему должны присваивать значение по умолчанию этому члену в конструкторе компонента. Например, конструктор для компонента FlashingLabel мог бы выглядеть следующим образом:

```
__fastcall TFlashingLabel::TFlashingLabel(TComponent* AOwner)
  : TCustomLabel(AOwner)
{
  FFlashRate = 800;
  // остальной код
}
```

Убедитесь, что вы установили нужные значения для всех членов-данных, соответствующих свойствам со значениями по умолчанию.

Если вы не хотите использовать для свойства значение по умолчанию, просто опустите спецификатор default в объявлении свойства.

Свойства могут быть отображаемыми или неотображаемыми

Некоторые свойства доступны на этапе разработки. Их можно изменять через инспектор объектов, а также изменять и считывать во время выполнения программы. Другие свойства имеют тип *runtime-only* и недоступны на этапе разработки (не отображаются в инспекторе объектов). Свойства, доступные во время разработки, называются *отображаемыми (published)*, а свойства, недоступные во время разработки — *неотображаемыми (nonpublished)*.

Проще говоря, отображаемое свойство — это то свойство, которое появляется в инспекторе объектов на этапе разработки. Вы можете сделать свойство отображаемым, поместив его объявление в раздел __published объявления класса компонента. Чтобы свойство было доступно только во время выполнения программы, поместите его объявление в раздел public. Все свойства, размещенные в разделе __published, будут отображаться в инспекторе объектов.

Написание методов для компонентов

Написание методов для компонентов практически не отличаются от написания методов для любого класса Си++. Единственное различие заключается в том, что методы компонентов должны использовать модификатор __fastcall. Кроме этого, методы ваших компонентов могут быть закрытыми, защищенными или открытыми. Давайте повторим то, что вы знаете об этих уровнях доступа.

Уровни доступа для компонентов

На четвертый день я говорил об уровнях доступа для классов. При этом были рассмотрены три уровня доступа: закрытый, защищенный и открытый. C++Builder расширяет Си++ в этом отношении, вводя отображаемый раздел. Я только что кратко рассказал об этом разделе. Давайте сделаем краткий обзор всех уровней доступа. В таблице 20.1 перечислены уровни доступа и даны краткие описания каждого из них.

Таблица 20.1. Уровни доступа класса компонента

Уровень доступа	Описание
Закрытый (Private)	Закрытый уровень имеет наибольшее число ограничений. Метод доступен для использования только в пределах данного класса. Ни пользователи компонента, ни производные классы не могут вызывать такие методы. Закрытые методы доступны только другим методам данного класса.
Защищенный (Protected)	Метод не может быть вызван пользователями компонента, но классы, произведенные от данного компонента, могут вызывать защищенные методы.
Открытый (Public)	Открытый метод доступен всем и может быть вызван пользователями класса или другими методами класса.
Отображаемый (Published)	Спецификатор __published относится только к свойствам и не применим к методам. Отображаемое свойство показывается в инспекторе объектов на этапе разработки.

В эту таблицу сведено все, что вам нужно знать об уровнях доступа. Она поможет вам при написании компонентов. Определить, какие методы должны быть открытыми, достаточно просто. Открытый метод — это тот, который пользователи вашего компонента будут вызывать для выполнения компонентом определенного действия.



Сводите использование методов в вашем компоненте к минимуму. Старайтесь использовать свойства для выполнения каких-либо действий. Если же это будет невозможно, используйте открытый метод.

Немного труднее разобраться в использовании закрытых и защищенных методов. После приобретения некоторого опыта вы сможете лучше распознавать ситуации, которые требуют использования защищенного уровня доступа вместо закрытого.



Методы чтения и записи для свойств обычно делают защищенными. Тем самым вы позволяете классам, произведенным от данного компонента, изменять поведение методов чтения и записи посредством их замещения.

Как я уже сказал, методы компонентов по своей сути остаются методами и в большинстве случаев могут рассматриваться как обычные функции-члены класса.

TFlashingLabel: пример компонента

Немного позже я буду говорить о событиях и о том, как они пишутся, но на данный момент вы имеете достаточно информации, чтобы написать свой первый компонент. Как уже было сказано, TFlashingLabel представляет собой компонент, который выводит на экран мигающий текст. Он имеет следующие характерные черты:

- Свойство FlashRate, которое управляет частотой мигания.
- Свойство FlashEnabled, которое включает и выключает режим мигания.
- Методы записи для двух свойств.
- Значения по умолчанию для двух свойств.
- Закрытый член класса (экземпляр класса TTimer) для управления периодом мигания.
- Все характеристики обычного компонента Label.

Сначала я покажу вам полные тексты заголовка и исходного модуля. (См. листинги 20.3 и 20.4.) После этого мы разберемся, как этот код работает.

Листинг 20.3. FlashingLabel.h

```
1: //---------------------------------------------------------------------
2: #ifndef FlashingH
3: #define FlashingH
4: //---------------------------------------------------------------------
5: #include <vcl\sysutils.hpp>
6: #include <vcl\controls.hpp>
```

см. продолжение

Листинг 20.3. (продолжение)

```
 7: #include <vcl\classes.hpp>
 8: #include <vcl\forms.hpp>
 9: #include <vcl\StdCtrls.hpp>
10: //---------------------------------------------------------------------
11: class TFlashingLabel : public TCustomLabel
12: {
13: private:
14:   //
15:   // Закрытые члены-данные
16:   //
17:   bool  FFlashEnabled;
18:   int   FFlashRate;
19:   TTimer* Timer;
20: protected:
21:   //
22:   // Защищенные методы записи для свойств
23:   //
24:   void __fastcall SetFlashEnabled(bool AFlashEnabled);
25:   void __fastcall SetFlashRate(int AFlashRate);
26:   //
27:   // Обработчик события OnTimer
28:   //
29:   void __fastcall OnTimer(TObject* Sender);
30: public:
31:   __fastcall virtualTFlashingLabel(TComponent* AOwner);
32: __published:
33:   //
34:   // Свойства компонента
35:   //
36:   __property bool FlashEnabled =
37:     {read=FFlashEnabled, write=SetFlashEnabled, default=true};
38:   __property int FlashRate =
39:     {read=FFlashRate, write=SetFlashRate, default=800};
40: };
41: //---------------------------------------------------------------------
42: #endif
```

Листинг 20.4. FlashingLabel.cpp

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: #pragma hdrstop
 4: #include "FlashingLabel.h"
 5: //---------------------------------------------------------------------
 6: static inline TFlashingLabel *ValidCtrCheck()
 7: {
 8:   return new TFlashingLabel(NULL);
 9: }
10: //---------------------------------------------------------------------
11: __fastcall TFlashingLabel::TFlashingLabel(TComponent* AOwner)
12:   : TLabel(AOwner)
13: {
14:   //
15:   // Присваивание членам-данным значений по умолчанию
16:   //
```

```
17:   FFlashEnabled = true;
18:   FFlashRate = 800;
19:   //
20:   // Инициализация таймера
21:   //
22:   Timer = new TTimer(this);
23:   //
24:   // Установка интервала таймера
25:   //
26:   Timer->Interval = FlashRate;
27:   //
28:   // Присваивание нашего собственного обработчика
29:   // событию OnTimer компонента TTimer
30:   //
31:   Timer->OnTimer = &OnTimer;
32: }
33: void __fastcall
34: TFlashingLabel::SetFlashEnabled(bool AFlashEnabled)
35: {
36:   //
37:   // Установка члена FFlashEnabled
38:   //
39:   FFlashEnabled = AFlashEnabled;
40:   //
41:   // Не запускать таймер в режиме разработки.
42:   // Вместо этого возврат
43:   //
44:   if (ComponentState.Contains(csDesigning)) return;
45:   //
46:   // Запуск таймера
47:   //
48:   Timer->Enabled = FFlashEnabled;
49:   //
50:   // Если мигание было выключено, необходимо
51:   // гарантировать видимость компонента
52:   //
53:   if (!FFlashEnabled) Visible = true;
54: }
55: void __fastcall TFlashingLabel::SetFlashRate(int AFlashRate)
56: {
57:   //
58:   // Одновременная установка члена FFlashRate и
59:   // интервала таймера
60:   //
61:   Timer->Interval = FFlashRate = AFlashRate;
62: }
63: void __fastcall TFlashingLabel::OnTimer(TObject* Sender)
64: {
65:   //
66:   // Если форма в режиме разработки,
67:   // остановка таймера и возврат
68:   //
69:   if (ComponentState.Contains(csDesigning)) {
70:     Timer->Enabled = false;
71:     return;
72:   }
73:   //
74:   // Переключение свойства Visible при каждом
```

см. продолжение

Листинг 20.4. (продолжение)

```
75:    // событии таймера
76:    //
77:    Visible = !Visible;
78: }
79: namespace FlashingLabel {
80: //
81: // Регистрация компонента
82: //
83: void __fastcall Register()
84: {
85:    TComponentClass classes[1] = {__classid(TFlashingLabel)};
86:    RegisterComponents("Samples", classes, 0);
87: }
88: }
```



Прежде всего, давайте посмотрим на заголовок (листинг 20.3). Обратите внимание на раздел private, начинающийся со строки 13. В нем объявлены три члена-данных. Первые два, FFlashEnabled и FFlashRate, связаны со свойствами FlashEnabled и FlashRate. Последнее объявление, в строке 19, выглядит следующим образом:

```
TTimer* Timer;
```

Здесь объявлен указатель на объект TTimer. Этот объект будет использоваться для регулировки частоты мигания.



Мы еще не рассматривали таймеры. Таймер устанавливается так, чтобы срабатывать через определенный интервал (в миллисекундах). При каждом срабатывании таймера посылается сообщение WM_TIMER тому окну, которое является владельцем таймера. В результате возникает событие OnTimer. Например, если вы установите интервал таймера равным 1000 миллисекунд, событие OnTimer будет вызываться приблизительно каждую секунду. Я должен отметить, что сообщение WM_TIMER имеет низкий приоритет и может быть игнорировано, если система занята. По этой причине нельзя использовать обычный таймер для критических операций. Однако для измерения некритичных интервалов TTimer вполне подходит. (Более подробная информация о таймерах приведена во встроенной справочной системе C++Builder, в разделах, посвященных TTimer или сообщению WM_TIMER.)

Раздел protected объявления класса, начинающийся со строки 20, содержит методы записи для свойств FlashRate и FlashEnabled. В этом разделе также объявлена функция OnTimer(). Эта функция будет вызываться при каждом возникновении события таймера. Она является защищенной и объявлена как virtual, чтобы производные классы могли осуществлять ее замещение для специальной обработки событий таймера. Например, производный класс может каждый раз при отображении текста изменять его цвет. Сделав метод виртуальным, мы делаем возможным его замещение, что позволяет дополнительно изменять поведение компонента.

Наконец, обратите внимание на раздел __published, начинающийся со строки 32. В строке 36 объявлено свойство FlashEnabled. Спецификатор read использует прямой доступ, спецификатору write присвоен метод SetFlashEn-

abled(), а значением по умолчанию является true. Свойство FlashRate использует аналогичную конструкцию.

Теперь обратите свое внимание на исходный модуль компонента, содержащийся в листинге 20.4. Прежде всего, вы видите, что в строках 17 и 18 членам-данным, представляющим свойства FlashEnabled и FlashRate, присваиваются значения по умолчанию. Вспомните, что мы объявили для этих свойств значения по умолчанию, но это влияет только на способ отображения свойства в инспекторе объектов. Поэтому в конструкторе мы должны присвоить действительные значения.

Теперь посмотрите на эти три строки, начинающиеся со строки 22 (комментарии удалены):

```
Timer = new TTimer(this);
Timer->Interval = FlashRate;
Timer->OnTimer = &OnTimer;
```

В первой строке создается экземпляр объекта TTimer. Во второй строке значение свойства FlashRate присваивается свойству Interval компонента Timer. Это устанавливает интервал мигания текста. Наконец, в последней строке адрес функции OnTimer() присваивается событию OnTimer объекта Timer. Тем самым гарантируется, что вы получите уведомление о возникновении события таймера.

Функция SetFlashEnabled() (начинающаяся со строки 34) представляет собой метод записи для свойства FlashEnabled. В строке 39 члену FFlashEnabled присваивается значение AFlashEnabled. В строке 48 свойство Enabled объекта Timer устанавливается в соответствии со значением AFlashEnabled. Записью в свойство FlashEnabled пользователь нашего компонента может включить или выключить мигание. Эта функция также содержит строку кода, которая устанавливает для свойства Visible значение true после выключения мигания. Тем самым исключается ситуация, когда текст останется скрытым при выключении мигания в середине интервала.

Начиная со строки 55, вы видите функцию SetFlashRate(). Это метод записи для свойства FlashRate. Пользователь может присвоить значение свойству FlashRate для управления периодом мигания. (1200 соответствует достаточно медленному миганию, а 150 — очень быстрому.) Эта функция содержит всего одну строку:

```
Timer->Interval = FFlashRate = AFlashRate;
```

С синтаксисом этой строки вы могли еще не встречаться. Это базовый синтаксис Си++ для последовательного присваивания значения AFlashRate сначала члену FFlashRate, а затем свойству Interval объекта Timer.

Функция OnTimer() не требует особых объяснений. Она вызывается в качестве реакции на событие OnTimer. Все, что нужно делать — это переключать состояние видимости компонента при каждом возникновении события. Иными словами, если FlashRate было установлено в 1000, события таймера будут происходить приблизительно каждую секунду. При первом событии компонент скрывается. Секунду спустя возникает следующее событие и компонент отображается. Это продолжается до тех пор, пока для свойства FlashEnabled не будет установлено значение false или приложение не будет закрыто. Наконец, строки с 83 по 88 содержат код для регистрации компонента.

Введите код из листингов 20.3 и 20.4 в файлы FlashingLabel.h и FlashingLabel.cpp, созданные ранее мастером компонентов. Не утруждайте себя набором строк комментариев, если у вас нет желания это делать. Не забудьте про метод cut-and-paste, когда нужно будет добавить все отображаемые свойства TCustomLabel. Объявление класса TCustomLabel можно найти в файле Stdctrls.hpp. После того, как вы введете код, мы протестируем компонент, чтобы убедиться в его работоспособности.

Предварительное планирование

Сейчас я немного забегу вперед, чтобы обратить ваше внимание на код, который не был затронут при анализе листинга 20.4. В функции SetFlashEnabled() есть одна важная строка, которая требует некоторого пояснения. В строке 44 вы увидите следующий код:

```
if (ComponentState.Contains(csDesigning)) return;
```

Эта строка проверяет, не используется ли компонент в форме на этапе разработки. Если форма находится в режиме разработки, вам нужно предотвратить запуск таймера. Когда компонент помещается в форму, находящуюся в редакторе форм, он не обязан полностью функционировать, поскольку редактор форм не может полностью смоделировать работающую программу. Вам не требуется запускать таймер, пока компонент используется в режиме разработки.

Все визуальные компоненты имеют свойство с именем ComponentState. Оно показывает, в частности, находится ли компонент в режиме разработки. Флаг csDesigning может быть либо установлен (on), либо сброшен (off). Если этот флаг установлен, компонент используется в форме, загруженной в редактор форм. Чтобы определить состояние флага csDesigning, вы можете использовать функцию Contains(). Если флаг установлен, вы можете выполнить возврат из функции OnTimer() без запуска таймера.



В Delphi свойство ComponentState реализовано в виде набора (набор — это элемент языка Паскаль). Набор представляет собой собрание объектов. В качестве простой аналогии можно рассмотреть корзину с фруктами. Корзина (набор) может содержать яблоки, апельсины, персики или сливы. В ней может быть один или несколько таких объектов, или все сразу. Свойство ComponentState, таким образом, является набором флагов, которые индицируют текущее состояние компонента.

Язык Си++ не поддерживает наборы. Вместо этого разработчики C++Builder реализовали наборы как шаблоны Си++. Я не буду сейчас углубляться в детали устройства шаблонов, а просто скажу, что вы можете использовать функцию Contains() для определения того, существует ли отдельный объект в наборе.

В строках с 69 по 72 листинга 20.4 проверяется, не используется ли компонент в форме, загруженной в редактор форм. Если компонент используется в редакторе форм, таймер выключается и происходит возврат из функции без выполнения оставшегося кода. По умолчанию таймер запускается в конструкторе TFlashingLabel, поэтому он должен быть немедленно выключен, если компонент используется на этапе разработки.

Тестирование компонента

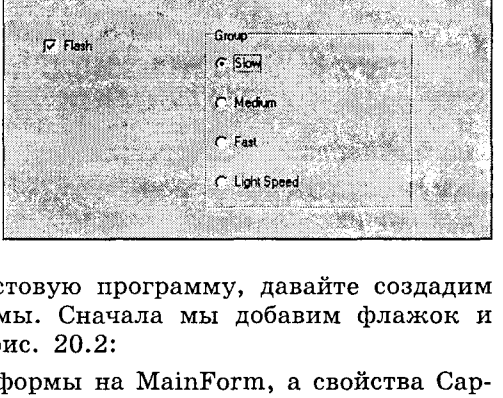
В конце концов, вы добавите вновь созданный компонент в библиотеку компонентов и палитру компонентов. Однако сначала вы должны протестировать свой компонент, чтобы убедиться в его работоспособности и в том, что он функционирует, как предполагалось.



Когда вы добавляете компонент к библиотеке компонентов, C++Builder перестраивает всю библиотеку VCL. В зависимости от аппаратуры, при первом добавлении компонента это может занять довольно много времени (на моем старом 486 требуется около 10 минут). После начального построения, последующие занимают только пару минут. В любом случае, перед добавлением компонента в библиотеку тщательно протестируйте его.

Для тестирования компонента вам нужно написать приложение, которое будет служить испытательным стендом. Поскольку вы не можете поместить компонент в форму из палитры компонентов, его придется создавать вручную. В нашем случае компонент FlashingLabel имеет два свойства и мы хотим убедиться, что они работают. Для этого тестовая программа должна включать и выключать режим мигания. Кроме того, программа должна позволить нам установить несколько частот мигания, чтобы убедиться в правильной работе свойства FlashRate. На рис. 20.2 показана работающая тестовая программа. Этот рисунок даст вам общее представление о том, что мы собираемся сделать.

Рис. 20.2
Работающая тестовая программа



Теперь, когда вы взглянули на тестовую программу, давайте создадим ее. Как всегда, начните с пустой формы. Сначала мы добавим флажок и группу радиокнопок в соответствии с рис. 20.2:

1. Измените значение свойства Name формы на MainForm, а свойства Caption — на FlashingLabel Test Program.
2. Используя рис. 20.2 в качестве образца, добавьте к форме компонент CheckBox. Измените значение его свойства Name на FlashBox, свойства Caption — на Flash, а свойства Checked — на true.
3. Перейдите на вкладку Events инспектора объектов и дважды щелкните в столбце Value рядом с событием OnClick. Редактор кода покажет обработчик OnClick для компонента FlashBox. Введите следующий код, начиная с позиции курсора (имя компонента FlashingLabel будет Flasher):

```
Flasher->FlashEnabled = FlashBox->Checked;
```

Это приведет к разрешению или запрещению мигания в зависимости от состояния флажка.

4. Поместите в форму компонент RadioGroup. Измените значение его свойства Name на Group, а свойства Caption — на Flash Speed.
5. Дважды щелкните в столбце Value рядом со свойством Items. После отображения редактора строк наберите следующие строки:

```
Slow
Medium
Fast
LightSpeed
```

Щелкните на кнопке OK для закрытия редактора строк. Введенные строки будут отображены в групповом окне как радиокнопки.

6. Установите для свойства ItemIndex значение 1. Радиокнопка Medium будет нажата.
7. Выделив радиогруппу, перейдите к вкладке Events инспектора объектов. Дважды щелкните в столбце Value рядом с событием OnClick. Редактор кода покажет обработчик события OnClick для группового окна.
8. Введите следующий код, начиная с позиции курсора:

```
switch (Group->ItemIndex) {
  case 0 : Flasher->FlashRate = 1200; break;
  case 1 : Flasher->FlashRate = 800; break;
  case 2 : Flasher->FlashRate = 400; break;
  case 3 : flasher->FlashRate = 150; break;
}
```

Это будет устанавливать значение FlashRate в зависимости от того, какая из радиокнопок нажата.

9. Сохраните проект в том же каталоге, где находится ваш компонент TFlashingLabel. Дайте модулю главной формы имя FTMain.cpp, а проекту — FlashTst.mak.

Теперь нам нужно добавить сам компонент. Поскольку этот компонент пока не является визуальным (мы не можем взять его из палитры компонентов), нужно добавить его вручную. Это делается следующим образом:

1. Щелкните на кнопке Add to Project (в панели инструментов, из главного меню или окна менеджера проектов). После появления диалогового окна Add to Project выберите файл FlashingLabel.cpp и щелкните на кнопке OK.
2. Перейдите к исходному модулю главной формы в редакторе кода и выберите в главном меню пункт File | Include Unit Hdr. Выберите FlashingLabel в диалоговом окне Include Unit и нажмите OK.
3. Перейдите к файлу FTMain.h и добавьте следующее объявление в раздел private класса TMainForm:

```
TFlashingLabel* Flasher;
```

4. Найдите в редакторе кода конструктор главной формы. Введите в конструктор следующий код:

```
Flasher = new TFlashingLabel(this);
Flasher->Parent = this;
Flasher->SetBounds(20, 20, 200, 20);
```

```
Flasher->Font->Size = 16;
Flasher->Caption = "This is a test";
Flasher->FlashRate = 800;
```

Теперь все готово для тестирования компонента. Нажмите кнопку Run для компиляции и запуска тестовой программы. Если вы получите сообщения об ошибках, тщательно проверьте код и исправьте ошибки, на которые указывает компилятор.

Когда программа заработает, вы можете щелчком на флажке Flash включить и выключить мигание. Измените частоту мигания, выбрав одну из радиокнопок в групповом окне. Поздравляю, вы написали свой первый компонент!

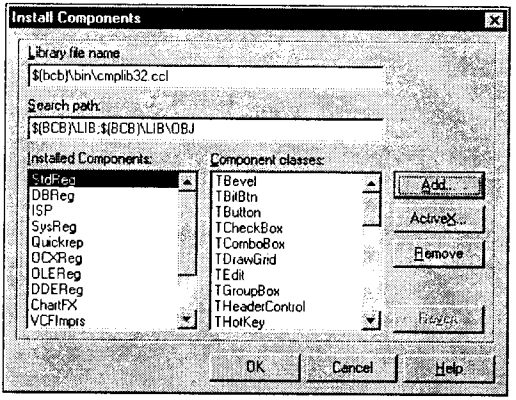
Добавление компонента в библиотеку компонентов

После того, как компонент заработает и вы будете им удовлетворены, можно добавить его в библиотеку компонентов. При этом он будет добавлен и в палитру компонентов. Однако прежде чем инсталлировать компонент FlashingLabel, давайте коротко рассмотрим диалоговое окно Install Components.

Диалоговое окно Install Components

Для добавления компонента в библиотеку компонентов, выберите в главном меню пункт Component | Install. На экране появится диалоговое окно Install Components (Инсталлировать компоненты). Это диалоговое окно позволяет вам добавлять и удалять компоненты, а также добавлять элементы управления OCX. Диалоговое окно Install Components показано на рис. 20.3.

Рис. 20.3
Диалоговое окно Install Components



Первое поле содержит имя исходного файла библиотеки. Вам никогда не потребуется его изменять. Следующее поле содержит путь, который C++Builder будет использовать для поиска компонентов, подлежащих компиляции при перестройке библиотеки компонентов. Имейте в виду, что библиотека компонентов может содержать компоненты из различных источни-

ков: компоненты, входящие в поставку C++Builder, компоненты от независимых разработчиков, ваши собственные компоненты и т.д. При каждой перекомпиляции библиотеки C++Builder должен знать, где расположены все инсталлированные компоненты.



Вам не потребуется изменять поле Search Path непосредственно. При добавлении нового компонента с помощью кнопки Add, его расположение будет автоматически добавлено к путям поиска. Соблюдайте осторожность при изменении этого поля.

В центре диалогового окна Install Components расположены два окна списков. Левое окно содержит зарегистрированные группы компонентов. Когда вы щелкаете на одной из этих групп, в правом окне отображаются классы компонентов, входящие в данную регистрационную группу.

Для удаления инсталлированного компонента, выделите его в списке Component Classes и щелкните на кнопке Remove (Удалить). Компонент будет удален из списка инсталлированных компонентов.



Каждый раз, когда вы делаете изменения в диалоговом окне Install Components, библиотека компонентов автоматически перестраивается. Это происходит как при добавлении, так и при удалении компонента. Удаленный компонент не будет в действительности удален до тех пор, пока библиотека не будет перестроена. Точно так же заново инсталлированный компонент не появится в палитре компонентов, пока C++Builder не закончит построение библиотеки.

Если вы хотите вернуться к первоначальной конфигурации, щелкните на кнопке Revert (Обратить) и затем на кнопке OK. Библиотека будет приведена к тому виду, который она имела при инсталляции C++Builder.

Однако о чем вы действительно захотите узнать, так это о кнопке Add (Добавить). Кнопка Add, как можно было ожидать, позволяет вам инсталлировать новый компонент в библиотеку компонентов. Давайте рассмотрим этот процесс на примере компонента FlashingLabel.



Возможно, вам стоит поместить все созданные вами компоненты в одно место. Собрав все компоненты в одном каталоге, вы избежите необходимости охотиться за ними, если когда-нибудь потребуется изменить компонент.

Инсталляция компонента FlashingLabel

Теперь мы готовы инсталлировать компонент FlashingLabel. Первым делом, однако, закройте все открытые проекты.



Если у вас открыт проект, C++Builder закроет его перед продолжением инсталляции компонента. Если проект требовал сохранения, C++Builder предложит вам это сделать.

Итак, мы готовы начать. Выполните следующие действия:

1. Выберите в главном меню пункт Component | Install. На экране появится диалоговое окно Install Components.

2. Щелкните на кнопке Add. Появится диалоговое окно Add Module (Добавить модуль). Нажмите кнопку Browse и найдите файл FlashingLabel.cpp. Нажмите кнопку OK для выбора файла и нажмите ее еще раз, чтобы закрыть диалоговое окно Add Module.
3. Теперь компонент FlashingLabel показан в нижней части окна списка Installed Components. Все готово для инсталляции компонента.
4. Щелкните на кнопке OK, чтобы закрыть диалоговое окно Install Components. C++Builder начнет перестраивать библиотеку компонентов.
5. Сядьте и сделайте дыхательные упражнения, пока C++Builder перестраивает библиотеку (в большинстве систем это займет всего пару минут).

Если вы тщательно протестировали свой компонент, построение библиотеки пройдет без задержек. При появлении ошибок, в окне состояния компилятора будет указано их количество. Кроме того, откроется окно Messages (специальное окно, которое появляется только при возникновении ошибок в процессе построения библиотеки компонентов). Прежде, чем вы сможете продолжить построение библиотеки, необходимо исправить все ошибки. Вот почему следует полностью протестировать компоненты перед их инсталляцией в библиотеку компонентов.

Когда C++Builder закончит перестраивать библиотеку компонентов, вы можете поискать свой новый компонент. Проверьте вкладку Samples — там должна быть кнопка, выглядящая как кнопка компонента Label или, возможно, кнопка, которую C++Builder по умолчанию присваивает компоненту. Если вы задержите курсор мыши на кнопке, всплывающая подсказка должна сообщить: FlashingLabel. Начните новый проект и протестируйте компонент FlashingLabel, поместив его в форму. Обратите внимание, что наряду со свойствами FlashRate и FlashEnabled в инспекторе объектов присутствуют все обычные свойства компонента Label. Заметьте, что для новых свойств отображаются значения по умолчанию, которые мы указали.

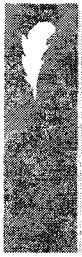
Добавление значка для кнопки компонента

Я не хочу умалять ваших заслуг, но вы могли заметить одну проблему. Кнопка для компонента FlashingLabel в палитре компонентов выглядит так же, как и для обычного компонента Label. Этого быть не должно!



Если C++Builder не может найти значок для нового компонента, он использует значок базового класса компонента или значок палитры компонентов, устанавливаемый по умолчанию.

Вы можете при желании указать значок для вашего нового компонента (обычно так и делается). Он должен содержаться в файле .dcr и следовать специальному соглашению об именах. (Файл .dcr представляет собой откомпилированный файл ресурсов, аналогичный файлу .res). Файл должен иметь то же имя, что и исходный файл компонента, и находиться в том же каталоге. Кроме того, файл .dcr должен содержать ресурс в виде растрового изображения, имя которого совпадает с именем класса компонента. Например, чтобы создать значок для кнопки компонента FlashingLabel, вы должны создать файл с именем FlashingLabel.dcr, содержащий растровый ресурс с именем TFLASHINGLABEL. В имени ресурса регистр букв не имеет значения, но по традиции используется верхний.



Иногда вы захотите взять кнопку для базового класса и слегка ее изменить, чтобы представить новый компонент. В таком случае можно запустить редактор изображений и открыть один из файлов .dcr в каталоге \CBuilder\Lib\Obj. (Вам придется немного поискать нужный значок.) Например, значок для компонента Label находится в файле с именем Stdreg.dcr. Откройте этот файл, скопируйте значок TLABEL в буфер обмена, откройте новый .dcr-файл и вставьте содержимое буфера в новый растровый ресурс с именем TFLASHINGLABEL. Измените изображение нужным образом и сохраните проект ресурса.

Значок для кнопки компонента должен иметь размер 24 на 24 пиксела. В большинстве случаев достаточно 16 цветов. В C++Builder темно-желтый цвет используется как прозрачный, и вы должны иметь это в виду при разработке значков для своих кнопок.

Если вы хотите добавить значок для кнопки своего компонента, как описано в этом разделе, то сначала создайте файл .dcr. Поместите этот файл в один каталог с файлом FlashingLabel.cpp и перестройте библиотеку компонентов (для этого нужно выбрать в главном меню пункт Component | Rebuild). Если кнопка не отображается нужным образом, вы можете сначала удалить компонент FlashingLabel, а затем повторно его инсталлировать. Если все сделано правильно, созданный вами значок будет показан в палитре компонентов.

Написание событий для компонентов

Написание событий требует некоторого планирования. Когда я говорил о событиях, то описал две различных возможности. Иногда событие возникает как результат сообщения Windows, а иногда — как результат изменения в компоненте. Событие, инициируемое сообщением Windows, не может вами полностью контролироваться. Вы можете реагировать на события этого типа, но обычно не инициируете их. События второго типа инициируются самим компонентом. Иными словами, вы можете ими управлять, будучи разработчиком компонента.

Работа с событиями на этом уровне поначалу может вызвать сильную путаницу. Я постараюсь обойти все подводные камни и покажу вам, как можно использовать события на практике. Для этого мы добавим событие к классу TFlashingLabel. Однако предварительно давайте рассмотрим некоторые основные положения, касающиеся событий.

Обзор событий

Прежде всего, вы должны понимать, что события являются свойствами. А раз это так, то они имеют все черты, присущие «обычным» свойствам: например, вы можете присвоить значения спецификаторам чтения и записи. События, как и остальные свойства, также используют базовые члены-данные для хранения своих значений. В случае событий, базовый член содержит адрес функции, которая будет вызвана при возникновении этого события. Как и свойства, события могут быть отображаемыми или неотображаемыми. Отображаемые события показываются в инспекторе объектов точно так же, как и отображаемые свойства.

Во-вторых, события являются *клаузами (closures)*. Клауза — это специальный тип указателя на функцию, который содержит адрес функции, вызываемой при возникновении события. Клаузы — это что-то вроде суперуказателей на функции, которые могут указывать не только на функцию в экземпляре класса, но и на функцию в экземпляре любого другого класса, не связанного с данным. Пока сохраняется соответствие объявлению функции (тот же тип возвращаемого значения и те же параметры), клауза вызывает функцию независимо от ее расположения. Например, событие OnClick объекта TLable может указывать на функцию-обработчик события в объекте TEdit, объекте TForm, объекте TListBox и т.д. Клаузы на самом деле немного сложнее, но нам не нужно вдаваться в эти подробности. (Кстати, информация для тех, кто удивлен: клаузы не содержатся в стандартном Си++.)



События всегда имеют возвращаемый тип void. В зависимости от типа, событие может принимать один или несколько аргументов, но оно не может возвращать значение. Вы можете, однако, получить информацию обратно от обработчика события, передавая один или более аргументов по ссылке и позволяя пользователю изменять эти параметры для достижения определенного эффекта.

Вы можете работать с событиями на нескольких уровнях. Например, вы можете заместить обработчик одного из событий базового класса, чтобы добавить дополнительные функции. Пусть вы хотите, чтобы при щелчке мышью на вашем компоненте выполнялось какое-то специальное действие. В таком случае не нужно проходить весь путь создания нового события для щелчка, поскольку оно уже существует в виде события OnClick базового класса. Вам достаточно дополнить существующее событие, а не создавать новое.

Другой способ работы с событиями заключается в создании такого события, которое можно инициировать из компонента. С событием именно такого типа мы будет иметь дело сначала. Как я говорил, мы собираемся добавить событие к компоненту TFlashingLabel, созданному ранее. Добавление этого события потребует также добавления нового свойства. Событие будет названо OnLimitReached, а новое свойство — FlashLimit. Это событие будет инициироваться после того, как компонент мигнет определенное число раз, указанное в FlashLimit. Если FlashLimit равно 0 (по умолчанию), событие OnLimitReached никогда не будет инициировано. Итак, давайте приступим к его созданию.

План работы

Написание события, определенного пользователем, можно разбить на пять основных задач:

1. Определение типа события.
2. Объявление события.
3. Объявление базового члена.
4. Создание виртуальной функции, которая вызывается при инициировании события.
5. Написание кода, инициирующего событие.

Давайте рассмотрим каждый из этих пунктов, чтобы вы лучше поняли их содержание.

Определение типа события

Ранее, при обсуждении свойств, я сказал, что свойство имеет определенный тип. То же самое верно и для событий. Однако в случае событий тип определяется указателем на функцию, который включает описание параметров обработчика события. Вчера, при рассмотрении DLL, я немного говорил об указателях на функцию (раздел «Вызов функций, расположенных в DLL»).

Как уже было сказано несколько раз, существуют два основных типа событий. Один из них — это *уведомляющее событие (notification event)*. Такое событие просто сообщает вам, что что-то произошло, не давая подробностей. Объявление функции-обработчика уведомляющего события выглядит следующим образом:

```
void __fastcall Clicked(TObject* Sender);
```

Единственная информация, содержащаяся в уведомляющем событии — это источник события. Для уведомляющих событий C++Builder предоставляет тип TNotifyEvent. В действительности это typedef, эквивалентный указателю на функцию. Не задумывайтесь об этом сейчас, просто отметьте для себя, что события уведомляющего типа, которые вы создаете, должны иметь тип TNotifyEvent.

Другой тип событий — это события, которые имеют более одного параметра и действительно передают информацию обработчику. При желании вы можете создавать для своих компонентов и такие события. Например, пусть вы хотите использовать обработчик события, объявленный следующим образом:

```
void __fastcall LimitReached(TObject* Sender, bool& Stop);
```

Использование обработчика этого типа позволяет пользователю изменить параметр Stop, посылая таким образом информацию о событии обратно к компоненту. Если вы собираетесь создавать события, имеющие параметры, вам потребуется объявить свой собственный тип функции. Например, пусть вам нужно написать тип для предыдущего объявления функции, имеющий имя TMyEvent. Его определение будет выглядеть следующим образом:

```
typedef void __fastcall
  (__closure *TMyEvent)(System::TObject* Sender, bool& Stop);
```

Хотя это выглядит запутанно, вам нужно только скопировать данный образец и затем добавить или удалить нужные параметры. После этого вы можете объявлять событие как имеющее тип TMyEvent. (Все это не будет иметь для вас большого смысла, пока мы не пройдем через весь процесс, поэтому запаситесь терпением.)



Объявите новый тип события перед объявлением класса в заголовочном файле компонента. Можете подсмотреть в листинг 20.5.

Постарайтесь как можно лучше определить, событие какого типа потребуется для вашего компонента. Если вам нужно только уведомляющее событие, достаточно использовать тип TNotifyEvent. Если же ваше событие будет передавать параметры обработчику, вам потребуется определить свой собственный тип события.

Объявление события

Теперь, когда вы определили тип события, можно объявить это событие. Объявление события для компонента выглядит почти аналогично объявлению любого другого свойства. Вот типичное объявление события:

```
__property TNotifyEvent OnSomeEvent =
    {read=FOnSomeEvent, write=FOnSomeEvent};
```

В основном это похоже на объявления свойств, с которым вы имели дело раньше. Обратите внимание на отсутствие спецификатора default. Заметьте также, что оба спецификатора чтения и записи указывают на член FOnSomeEvent. Это говорит о том, что события используют прямой доступ и не используют методы чтения и записи. Наконец, обратите внимание на то, что в этом примере событие имеет тип TNotifyEvent. Если ваше событие передает параметры, вам потребуется определить свой собственный тип, как описывалось в разделе «TFlashingLabel: пример компонента».

Наше событие OnLimitReached будет передавать параметры, поэтому нам нужно определить специальный тип и присвоить его событию. С учетом этого, объявление события OnLimitReached будет выглядеть следующим образом:

```
__property TMyEvent OnLimitReached =
    {read=FOnLimitReached, write=FOnLimitReached};
```

Немного позже я покажу вам полный заголовок и вы сможете увидеть, как все это выглядит в целом. (Если хотите забежать вперед, можете посмотреть листинг 20.5.)

Объявление базового члена

Объявление базовых членов-данных выполняется очень просто. Все, что мы должны сделать — это объявить закрытый член класса, имеющий тот же тип, что и событие. Вот как это выглядит:

```
TMyEvent FOnLimitReached;
```

Как и для свойств, имя члена образуется из имени события добавлением впереди буквы F.

Создание виртуальной функции для инициирования события

Создание виртуальной функции для инициирования события является действием, требующим некоторого разъяснения. Вы будете инициировать событие в результате определенного изменения в компоненте. В данном случае мы намереваемся это делать, когда компонент мигнет столько раз, сколько указано в свойстве FlashLimit. Вы можете инициировать событие простым его вызовом:

```
bool Stop = false;
OnLimitReached(this, Stop);
```

Инициировать событие можно из нескольких мест компонента, в зависимости от различных факторов. Чтобы свести все в одно место, мы создадим виртуальную функцию, которая и будет инициировать событие. Эта виртуальная функция будет иметь то же имя, что и событие, за исключением приставки On. Сначала мы объявим функцию в объявлении класса:

```
virtual void __fastcall LimitReached();
```

Затем мы должны написать тело функции, в котором будет инициироваться событие:

```
void __fastcall TFlashingLabel::LimitReached()
{
  //
  // Установка значения по умолчанию для параметра Stop.
  // Если пользователь не изменит это значение,
  // оно будет возвращено
  //
  bool Stop = false;
  //
  // Если с событием был связан обработчик,
  // вызов этого обработчика (событие инициируется здесь)
  //
  if (OnLimitReached) OnLimitReached(this, Stop);
  //
  // Установка значения свойства FlashEnabled в зависимости
  // от значения параметра Stop
  //
  FlashEnabled = !Stop;
}
```

Здесь нужно отметить несколько моментов. Во-первых, обратите внимание, что мы устанавливаем для параметра Stop значение по умолчанию. Если пользователь не изменит этот параметр, Stop будет иметь значение false. Параметр Stop используется для определения того, должно ли мигание прекратиться по достижении значения FlashLimit. В обработчике события (в приложении, которое использует это событие) пользователь может установить для Stop значение true, чтобы остановить мигание, или оставить параметр Stop как есть, позволив тем самым миганию продолжаться.

Затем обратите внимание на строку, инициирующую событие:

```
if (OnLimitReached) OnLimitReached(this, Stop);
```

Если пользователь компонента связал с событием обработчик, вы вызываете этот обработчик. Если обработчик не был определен, вы выполняете для события обработку по умолчанию. (В данном случае такая обработка отсутствует.) Важно, чтобы пользователь при желании мог игнорировать событие. Данный код позволяет так поступать.



Одной из трудно усваиваемых вещей при написании компонентов является то, что сам компонент не предоставляет вам обработчик события. Обработчик содержится в приложении, вызывающем компонент; мы только обеспечиваем механизм, посредством которого этот обработчик будет вызван при возникновении события.

Как упоминалось ранее, эта функция является виртуальной. Она сделана виртуальной для того, чтобы производные классы могли переопределить поведение события, которое оно имеет по умолчанию. Теперь любой компонент, произведенный от нашего компонента, должен только заместить функцию LimitReached(), чтобы изменить обработку события, которая выполняется по умолчанию. Это позволяет легко менять поведение события, не просматривая весь код для поиска мест, из которых оно инициируется. Событие инициируется только в функции LimitReached().

Написание кода, инициирующего событие

Где-то в компоненте должен быть код, вызывающий функцию LimitReached() (которая, в конечном итоге, инициирует событие). В случае компонента TFlashingLabel, мы будем вызывать LimitReached() из функции OnTimer(). Вот как выглядит эта функция после соответствующих изменений:

```
void __fastcall TFlashingLabel::OnTimer(TObject* Sender)
{
  //
  // Если компонент в режиме разработки,
  // остановка таймера и возврат
  //
  if (ComponentState.Contains(csDesigning)) {
    Timer->Enabled = false;
    return;
  }
  //
  // Переключение свойства Visible при каждом
  // возникновении события
  //
  Visible = !Visible;
  //
  // Инициирование события при необходимости. Увеличение счетчика
  // только при видимом (включенном) тексте
  //
  if (FFlashLimit && Visible) {
    //
    // Увеличение члена FlashCount
    //
    FlashCount++;
    //
    // Если FlashCount больше или равен значению
    // свойства FlashLimit, то сброс FlashCount в 0
    // и инициирование события
    //
    if (FlashCount >= FFlashLimit) {
      FlashCount = 0;
      LimitReached();
    }
  }
}
```

Как видите, при достижении значения FlashLimit вызывается функция LimitReached() и событие инициируется. Кстати, считается только каждое второе событие OnTimer(). Если вы будет увеличивать счетчик при каждом вызове функции OnTimer(), это будет неправильно, поскольку событие OnTimer() происходит дважды за период (включение и выключение).

Замещение событий базового класса

В предыдущем разделе было затронуто кое-что еще, о чем я хотел кратко упомянуть. Если вы хотите изменить поведение одного из событий, определенных в базовом классе, все, что нужно сделать — это заместить инициирующую функцию, как было описано выше. Например, пусть вам нужно изменить предопределенное событие OnClick так, чтобы динамик издавал звук при щелчке на компоненте. Для этого требуется заместить функцию Click() базового класса, как здесь:

```
void __fastcall TFlashingLabel::Click()
{
  //
  // Подача звукового сигнала и затем вызов метода Click()
  // базового класса для обработки по умолчанию
  //
  MessageBeep(-1);
  TCustomLabel::Click();
}
```

Поскольку эта функция объявлена в базовом классе как виртуальная, она будет автоматически вызываться при каждом щелчке мышью на компоненте (щелкая на мигающем компоненте, нужно иметь в виду, что все это будет работать, только если компонент видим). Мы заставляем динамик издавать звук лишь затем, чтобы проверить работу функции.

Собираем все вместе

Вы уже видели большую часть нового и усовершенствованного компонента TFlashingLabel на предыдущих страницах, но давайте теперь посмотрим на все вместе. В листинге 20.5 приведен заголовок TFlashingLabel, а листинг 20.6 содержит исходный файл для этого компонента. Изучите реализацию события OnLimitReached, чтобы лучше понять, как это делать в своих компонентах. В листинге 20.7 приведен исходный модуль тестовой программы FlashTst с обработчиком события OnLimitReached.

Листинг 20.5. FlashingLabel.h (новый и усовершенствованный)

```
 1: //---------------------------------------------------------------------
 2: #ifndef FlashingH
 3: #define FlashingH
 4: //---------------------------------------------------------------------
 5: #include <vcl\sysutils.hpp>
 6: #include <vcl\controls.hpp>
 7: #include <vcl\classes.hpp>
 8: #include <vcl\forms.hpp>
 9: #include <vcl\StdCtrls.hpp>
10: //---------------------------------------------------------------------
11:
12: //
13: // Определение нового типа для нашего обработчика события.
14: // Этот тип принимает TObject* и ссылку bool
15: //
16: typedef void __fastcall
17:   (__closure *TMyEvent)(System::TObject* Sender, bool& Stop);
18:
19: class TFlashingLabel : public Tlabel
20: {
21: private:
22:   //
23:   // Закрытые члены-данные
24:   //
25:   bool  FFlashEnabled;
26:   int   FFlashRate;
27:   int   FFlashLimit;
```

```
28:    int   FlashCount;
29:    TMyEvent FOnLimitReached;
30:    TTimer* Timer;
31:  protected:
32:    //
33:    // Защищенные методы записи для свойств
34:    //
35:    void __fastcall SetFlashEnabled(bool AFlashEnabled);
36:    void __fastcall SetFlashRate(int AFlashRate);
37:    //
38:    // Функция LimitReached(), инициирующая событие
39:    //
40:    virtual void __fastcall LimitReached();
41:    //
42:    // Замещенный метод Click() только как пример
43:    // замещения события базового класса. Эта функция
44:    // будет вызываться при каждом щелчке на компоненте,
45:    // поскольку она является виртуальной
46:    //
47:    virtual void __fastcall Click()
48:    {
49:      //
50:      // Подача звукового сигнала и вызов метода Click()
51:      // базового класса для обработки по умолчанию
52:      //
53:      MessageBeep(-1);
54:      TLabel::Click();
55:    }
56:    //
57:    // Обработчик события OnClick
58:    //
59:    virtual void __fastcall OnTimer(TObject* Sender);
60:  public:
61:    __fastcall virtual TFlashingLabel(TComponent* AOwner);
62:  __published:
63:    //
64:    // Свойства компонента
65:    //
66:    __property bool FlashEnabled =
67:      {read=FFlashEnabled, write=SetFlashEnabled, default=true};
68:    __property int FlashRate =
69:      {read=FFlashRate, write=SetFlashRate, default=800};
70:    __property int FlashLimit =
71:      {read=FFlashLimit, write=FFlashLimit, default=0};
72:    //
73:    // Событие OnLimitReached
74:    //
75:    __property TMyEvent OnLimitReached =
76:      {read=FOnLimitReached, write=FOnLimitReached};
77:  };
78:  //---------------------------------------------------------------------
79:  #endif
```

Листинг 20.6. FlashingLabel.cpp (новый и усовершенствованный)

```
//---------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "FlashingLabel.h"
//---------------------------------------------------------------------
static inline TFlashingLabel *ValidCtrCheck()
{
  return new TFlashingLabel(NULL);
}
//---------------------------------------------------------------------
__fastcall TFlashingLabel::TFlashingLabel(TComponent* AOwner)
  : TLabel(AOwner)
{
  //
  // Присваивание членам-данным значений по умолчанию
  //
  FFlashEnabled = true;
  FFlashRate = 800;
  FFlashLimit = 0;
  FlashCount = 0;
  //
  // Инициализация таймера
  //
  Timer = new TTimer(this);
  //
  // Установка интервала таймера
  //
  Timer->Interval = FlashRate;
  //
  // Присваивание событию OnTimer компонента TTimer
  // нашего собственного обработчика
  //
  Timer->OnTimer = &OnTimer;
}

void __fastcall
  TFlashingLabel::SetFlashEnabled(bool AFlashEnabled)
{
  //
  // Set FFlashEnabled data member.
  //
  FFlashEnabled = AFlashEnabled;
  //
  // Если компонент в режиме разработки, таймер не запускается.
  // Вместо этого возврат
  //
  if (ComponentState.Contains(csDesigning)) return;
  //
  // Запуск таймера
  //
  Timer->Enabled = FFlashEnabled;
  //
  // Если мигание было выключено, необходимо
  // гарантировать видимость компонента
  //
  if (!FFlashEnabled) Visible = true;
```

```
58: }
59:
60: void __fastcall TFlashingLabel::SetFlashRate(int AFlashRate)
61: {
62:   //
63:   // Одновременная установка члена FFlashRate
64:   // и интервала таймера
65:   //
66:   Timer->Interval = FFlashRate = AFlashRate;
67: }
68:
69: void __fastcall TFlashingLabel::OnTimer(TObject* Sender)
70: {
71:   //
72:   // Если компонент в режиме разработки,
73:   // остановка таймера и возврат
74:   //
75:   if (ComponentState.Contains(csDesigning)) {
76:     Timer->Enabled = false;
77:     return;
78:   }
79:   //
80:   // Переключение свойства Visible при каждом
81:   // событии таймера
82:   //
83:   Visible = !Visible;
84:   //
85:   // Инициирование события при необходимости.
86:   // Если текст видим (включен), только увеличение счетчика
87:   //
88:   if (FFlashLimit && Visible)  {
89:     //
90:     // Инкремент члена FlashCount
91:     //
92:     FlashCount++;
93:     //
94:     // Если FlashCount больше или равен
95:     // значению свойства FlashLimit, сброс
96:     // FlashCount в 0 и инициирование события
97:     //
98:     if (FlashCount >= FFlashLimit) {
99:       FlashCount = 0;
100:      LimitReached();
101:    }
102:  }
103: }
104:
105: void __fastcall TFlashingLabel::LimitReached()
106: {
107:   //
108:   // Установка значения по умолчанию для параметра Stop.
109:   // Если пользователь не изменит параметр,
110:   // будет возвращено это значение
111:   //
112:   bool Stop = false;
113:   //
114:   // Если с событием связан обработчик,
115:   // вызов этого обработчика (событие инициируется здесь)
```

см. продолжение

Листинг 20.6. (продолжение)

```
116:    //
117:    if (OnLimitReached) OnLimitReached(this, Stop);
118:    //
119:    // Установка значения свойства FlashEnabled
120:    // в зависимости от значения параметра Stop
121:    //
121:    FlashEnabled = !Stop;
122:    //
123:    // Подача звукового сигнала (просто так)
124:    //
125:    MessageBeep(-1);
126: }
130: //---------------------------------------------------------------------
131: namespace Flashing
132: {
133:    void __fastcall Register()
134:    {
135:      TComponentClass classes[1] = {__classid(TFlashingLabel)};
136:      RegisterComponents("Samples", classes, 0);
137:    }
138: }
```

Листинг 20.7. FlashTst.cpp

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: #pragma hdrstop
 4:
 5: #include "FlashingLabel.h"
 6: #include "FTMain.h"
 7: //---------------------------------------------------------------------
 8: #pragma resource "*.dfm"
 9: TMainForm *MainForm;
10: //---------------------------------------------------------------------
11: __fastcall TMainForm::TMainForm(TComponent* Owner)
12:    : TForm(Owner)
13: {
14:    Flasher = new TFlashingLabel(this);
15:    Flasher->Parent = this;
16:    Flasher->SetBounds(20, 20, 200, 20);
17:    Flasher->Font->Size = 16;
18:    Flasher->Caption = "This is a test";
19:    Flasher->FlashRate = 800;
20:    Flasher->FlashLimit = 5;
21:    Flasher->OnLimitReached = OnLimitReached;
22: }
23: //---------------------------------------------------------------------
24: void __fastcall TMainForm::FlashBoxClick(TObject *Sender)
25: {
26:    Flasher->FlashEnabled = FlashBox->Checked;
27: }
28: //---------------------------------------------------------------------
29: void __fastcall TMainForm::GroupClick(TObject *Sender)
30: {
31:    switch (Group->ItemIndex) {
```

```
32:      case 0 :  Flasher->FlashRate = 1200; break;
33:      case 1 :  Flasher->FlashRate = 800; break;
34:      case 2 :  Flasher->FlashRate = 400; break;
35:      case 3 :  Flasher->FlashRate = 150; break;
36:    }
37: }
38: //---------------------------------------------------------------------
39: void __fastcall TMainForm::OnLimitReached(TObject* Sender, bool& Stop)
40: {
41:    //
42:    // Обработчик события OnLimitReached. Установка параметра
43:    // Stop в true для остановки мигания
44:    //
45:    Stop = true;
46: }
47: //---------------------------------------------------------------------
```



Этот компонент и программа FlashTst включены в состав CD-ROM, прилагаемого к книге. Однако исходный файл компонента там называется Flashing.cpp, а заголовок — Flashing.h.

Запустите тестовую программу и убедитесь в том, что компонент работает, как ожидалось. Поэкспериментируйте с программой, чтобы лучше почувствовать работу событий и их обработчиков. Обратите внимание, что динамик издает звук при щелчке на тексте (пока он видим). Это обеспечивает подставляемая функция Click() в объявлении класса компонента. Я добавил эту функцию, чтобы проиллюстрировать замещение события базового класса.



Если вы хотите инсталлировать новый компонент FlashingLabel в палитру компонентов, выберите в главном меню пункт Component | Rebuild Library, и старая версия FlashingLabel будет заменена на новую.

Достижение мастерства в написании событий требует некоторого времени. Вам придется перецеловать много лягушек, пока одна из них не превратится в принцессу (или в принца). Иными словами, когда дело дойдет до написания событий, ничто не заменит настоящий опыт. Вероятно, вы набьете немало шишек, но это будет хорошей школой и вы почувствуете, что стали лучшим программистом, чем были раньше.

Заключение

Итак, теперь вы в большой лиге. Написание компонентов — это работа не для слабых духом. Если вы поняли основные идеи, весь мир лежит у ваших ног. Если вы в чем-то похожи на меня, то потратите на изучение невизуального программирования столько же времени, сколько потратили на визуальное. Если же вы чувствуете себя слегка растерянными, не огорчайтесь. Возможно, вам нужно сделать перерыв на несколько дней и затем снова вернуться к этой главе, когда новая информация усвоится. Не исключено, что вам придется прочитать несколько разных руководств по написанию компонентов, пока не наступит полное понимание. Не сдавайтесь, и в конце концов эти новые вещи обретут для вас смысл.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Есть ли причины, по которым мне нужно знать, как пишутся компоненты?

Нет. Вы можете использовать в своих приложениях базовые компоненты, которые предоставляет C++Builder. У вас может никогда не возникнуть необходимости писать компоненты.

Могу ли я купить компоненты?

Да. Для приобретения библиотек компонентов существует большое количество независимых источников. Эти библиотеки продаются компаниями, которые специализируются на написании компонентов VCL. В дополнение, многие бесплатные или условно-бесплатные компоненты доступны из различных сетевых источников. Обязательно поищите такие источники в Интернете.

Могу ли я использовать свои компоненты Delphi в C++Builder?

Да. C++Builder имеет возможность компилировать и инсталлировать компоненты Delphi. Инсталлируйте их как обычные компоненты C++Builder.

Обязан ли я использовать методы чтения и записи для своего свойства?

Нет. Вы можете использовать прямой доступ и сохранять значение свойства непосредственно в базовом члене, соответствующем этому свойству.

Каковы преимущества использования метода записи для моих свойств?

Использование метода записи позволяет вам выполнять дополнительные операции при записи в свойство. Запись в свойство часто сопровождается определенными действиями компонента. Использование метода записи позволит вам выполнять эти действия.

По каким причинам мне следует делать свойство открытым, но не отображаемым?

Отображаемое свойство показывается в инспекторе объектов во время разработки. Некоторые свойства не имеют для этого соответствующего интерфейса. Их следует делать открытыми, чтобы пользователь мог изменить или считать их во время выполнения программы, но не во время разработки.

Как мне протестировать свой компонент, чтобы убедиться в его правильной работе до инсталляции?

Напишите тестовую программу и добавьте к ней исходный файл компонента. В конструкторе главной формы создайте экземпляр своего компонента. Установите все свойства, которые должны быть заданы перед тем, как компонент станет видимым. Затем в тестовой программе изменяйте открытые свойства компонента и смотрите, все ли работает правильно.

Должен ли я писать события для своих компонентов?

Не обязательно. Некоторые компоненты используют события, другие — нет. Не уходите в сторону от решаемой задачи ради написания событий, но и не избегайте этой работы, когда в ней возникнет необходимость.

Вопросы

1. Должно ли свойство использовать метод записи?
2. Должно ли свойство иметь базовые члены-данные?
3. Можете ли вы создать компонент путем изменения уже существующего?
4. Что произойдет, если вы не укажете спецификатор записи (ни метод записи, ни прямой доступ) в объявлении свойства?
5. Что означает прямой доступ?
6. Должны ли ваши свойства иметь значения по умолчанию?
7. Приводит ли указание значения по умолчанию к автоматической установке значения базового члена?
8. Как инсталлировать компонент в палитру компонентов?
9. Как указать значок кнопки, которую ваш компонент будет использовать в палитре компонентов?
10. Как инициировать событие, определенное пользователем?

Упражнения

1. Вернитесь к исходному тексту компонента FlashingLabel в листингах 20.5 и 20.6. Полностью разберитесь в том, что делает этот код.
2. Удалите компонент FlashingLabel из библиотеки компонентов. Инсталлируйте его заново.
3. Напишите тестовую программу, которая использует три компонента FlashingLabel с различными частотами мигания.
4. Измените значок кнопки FlashingLabel в палитре компонентов на свой собственный.
5. Напишите метод записи для свойства FlashLimit компонента FlashingLabel так, чтобы пользователь не мог ввести число, большее 100.
6. Измените событие OnLimitReached для компонента FlashingLabel на обычное уведомляющее событие (**Указание**: TNotifyEvent).
7. **Дополнительное задание**: Напишите компонент своей собственной разработки.
8. **Дополнительное задание**: Протестируйте ваш новый компонент и инсталлируйте его в палитру компонентов.

C++Builder и Delphi



Кен Хендерсон

Одной из первых вещей, которые отмечают в C++Builder, является его внешнее сходство с Delphi. Естественно, возникают вопросы, в чем состоит различие между ними и почему два различных продукта существуют на первом месте. Сегодня я попытаюсь дать ответы на эти вопросы и рассказать о том, какое отношение C++Builder имеет к Delphi и как он может использоваться совместно с Delphi.

Основные различия между C++Builder и Delphi

Самое большое различие между C++Builder и Delphi состоит в том, что Delphi основан на языке программирования Объектный Паскаль (Object Pascal), тогда как C++Builder (что очевидно) — на Си++. Объектный Паскаль является последним вариантом языка Турбо-Паскаль, впервые представленного Borland в 1980-х годах и имевшего большой успех. Хотя многие разработчики знали и любили Паскаль еще в 80-х, до появления Delphi, этот язык был близок к вымиранию. Delphi спас язык от забвения и теперь он используется большим числом разработчиков, чем когда-либо ранее. Несмотря на это, язык Си был и, вероятно, всегда будет популярнее Паскаля. Если вы были одним из тех, кто не давал Delphi шансов на успех, потому что он был основан на Объектном Паскале, вам повезло. Borland C++Builder предоставляет вам большую часть сильных сторон Delphi в продукте, основанном на Си++.

Я должен подчеркнуть, что своим успехом Delphi во многом обязан именно Объектному Паскалю, а не Си++. Фирма Borland ввела свой стандарт на язык программирования Объектный Паскаль, который не соответствует требованиям совместимости с другими поставщиками или компиляторами. Это позволило Borland расширить Объектный Паскаль для поддержки модели визуального программирования Delphi. Тогда как поставщики программных инструментов Си++ должны отвечать перед комитетом по стандартам ANSI за все изменения, которые они вводят в язык, люди, модифицирующие Delphi, не скованы этим. В результате Borland может совершенствовать Объектный Паскаль до идеального соответствия потребностям визуальной разработки, все время сохраняя возможность компилировать его в машинный код. Быть диссидентом в данном случае означает иметь преимущество.

Однако диссидентство имеет и обратные стороны. С одной стороны, дополнительные библиотеки для диалектов Borland Pascal всегда были скудны по сравнению с тем, что предлагалось для языков программирования Си и Си++. С другой стороны, если Borland вдруг решит испортить Объектный Паскаль каким-нибудь ужасным способом (я сомневаюсь, конечно, что это произойдет), никакая группа по стандартам ей не помешает. Поскольку для Объектного Паскаля отсутствует реальная конкуренция, реализация Borland является стандартом де-факто.

Я мог бы не беспокоиться так об отсутствии стандартов или возможной кончине дополнительных библиотек для Объектного Паскаля или Delphi. Однако для некоторых это имеет огромное значение. Этим людям будет отрадно знать, что под всем очарованием и блеском C++Builder скрывается стандартный Си++, который они узнали и полюбили по предыдущим продуктам серии Borland C++. Вы можете подумать, что C++Builder является гибридом Delphi 2.0 и Borland C++ 5.0. Да, это инструмент RAD (rapid application development), но это одновременно и стандартный инструмент. В этом заключается еще одно различие между C++Builder и Delphi. Delphi определенно стоит особняком.

Другие различия между C++Builder и Delphi

После того, как я рассмотрел главные различия между C++Builder и Delphi, давайте обратимся к менее существенным. Это в основном будет полезно тем, кто переходит к C++Builder от Delphi.

Файлы проектов

C++Builder сохраняет опции проекта в *информационных файлах (makefiles)*, имеющих расширение .MAK. Delphi, напротив, сохраняет опции проекта в файлах с расширением .DOF.

Иформационный файл, в котором C++Builder сохраняет опции компилятора/компоновщика, является полноценным — вы можете использовать его вне C++Builder вместе с поставляемыми инструментами командной строки. Вы можете даже редактировать информационный файл C++Builder для установки расширенных опций компилятора или компоновщика, не поддерживаемых IDE C++Builder.



Все внешние изменения в информационных файлах должны выполняться в строгом соответствии с синтаксисом информационных файлов C++Builder. В противном случае их дальнейшее использование с IDE C++Builder будет невозможно.

Библиотека компонентов

Другое небольшое различие между C++Builder и Delphi заключается в способе реализации библиотеки визуальных компонентов (VCL). Основу VCL C++Builder составляют DLL объектных файлов. В Delphi 2.0, напротив, VCL реализована через файлы DCU (Delphi Component Unit — модуль компонентов Delphi).

Поддержка нескольких языков программирования

Хотя C++Builder основан на Си++, он будет компилировать также модули Delphi, написанные на Объектном Паскале. Вы можете открывать в C++Builder формы Delphi и инсталлировать компоненты Delphi. В этом C++Builder представляет собой полную противоположность Delphi, который поддерживает только Объектный Паскаль. C++Builder позволяет вам свободно смешивать модули исходного кода .CPP и .PAS в своих приложениях и DLL.

Заголовочные файлы

В противоположность Delphi, C++Builder использует отдельные заголовочные файлы для определений классов форм. Delphi хранит определения классов, соответствующие вашим формам, в разделе interface исходного файла формы. C++Builder, напротив, хранит эту информацию в отдельном заголовке (.H-файле), который затем включается в главный модуль исходного кода формы директивой #include.

Эмуляция элементов Delphi в C++Builder

Существует несколько областей, где C++Builder эмулирует функции, первоначально реализованные в Delphi. Это служит еще одним доказательством того, что две среды максимально совместимы и помогает делать в C++Builder те же вещи, которые способствовали такой популярности Delphi.

Изменения в компиляторе

В компиляторе C++Builder были сделаны несколько изменений, которые сблизили его с компилятором Delphi. Например, была введена поддержка собственных свойств и операторов, а также новых ключевых слов, относящихся к Delphi. Для совместимости и возможности взаимодействия с Delphi были добавлены ключевые слова __declspec, delphiclass, delphireturn, __automated, __published, __closure, __property, __classid(class) и __dispid(int).

Новые классы

Для совместимости с Delphi были также добавлены несколько новых классов Си++. Это классы AnsiString, Variant, ShortString, Currency, TDateTime и Set.

Поддержка компонентов

Поскольку компоновщик C++Builder поддерживает DCU Delphi, вы можете инсталлировать и использовать в C++Builder компоненты Delphi 2.0. Если у вас есть исходный код для этих компонентов, вы можете инсталлировать их в палитру компонентов C++Builder и использовать в своих приложениях или в качестве основы для новых компонентов.

Открытые массивы

К особенностям Delphi относится поддержка *открытых массивов (open arrays)* — массивов, для которых точное число и тип элементов неизвестны на этапе компиляции. C++Builder также поддерживает эту мощную возможность и предоставляет несколько вспомогательных макросов для работы с открытыми массивами. Это макросы OPENARRAY, ARRAYOFCONST, EXISTINGARRAY и SLICE.

Среда разработки

Вы можете заметить, что кроме заголовка в верхней части экрана, интегрированная среда разработки (IDE) C++Builder почти идентична среде Delphi 2.0. Инспектор объектов, редактор форм и палитра компонентов совершенно одинаковы. Если вы раньше использовали Delphi, то должны чувствовать себя в C++Builder как дома. Рис.21.1 и 21.2 иллюстрируют сходство между двумя средами.

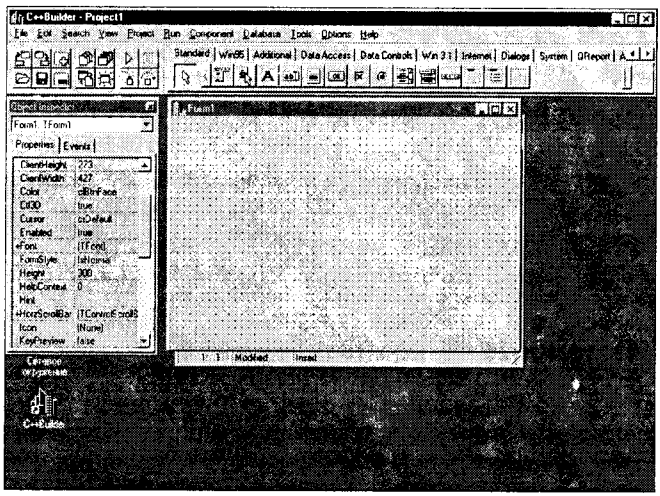


Рис. 21.1
IDE C++Builder

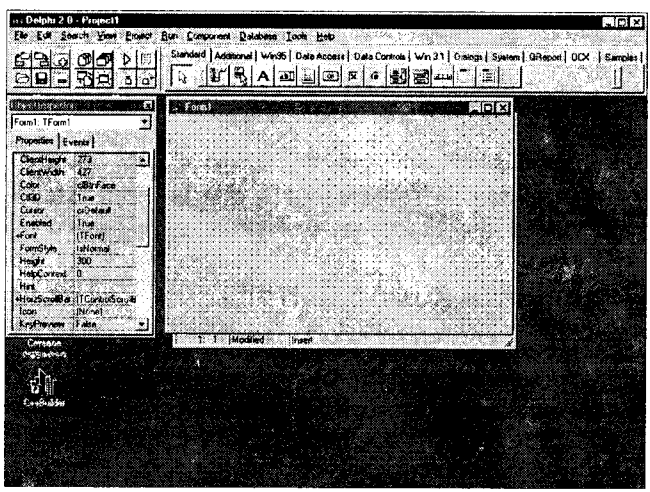


Рис. 21.2
IDE Delphi 2.0

Доступ к базам данных

В силу того, что Delphi и C++Builder используют для доступа к базам данных Borland Database Engine, разработка приложений для работы с базами данных выглядит практически одинаково. Концепции, на основе которых строится доступ к базам данных в Delphi, в равной мере применимы к C++Builder.

Заключение

Сегодняшнее краткое обсуждение было посвящено сходствам и различиям между C++Builder и Delphi, включая элементы их языков, наборы компонентов, нюансы компиляторов и дополнительные возможности. Я надеюсь, что это обсуждение поможет вам понять относительную силу и слабость каждой из двух сред достаточно хорошо, чтобы эффективно работать с ними.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Могу ли я компилировать в C++Builder программы на языке Объектный Паскаль?

Да, C++Builder способен компилировать модули, написанные на Объектном Паскале.

Могу ли я компилировать свои исходные модули C++Builder в Delphi?

Нет, Delphi не предоставляет вам такой возможности.

Могут ли компоненты, разработанные для Delphi, использоваться также и в C++Builder?

Да.

Вопросы

1. Какие новые ключевые слова, относящиеся к Delphi, были добавлены в C++Builder?
2. Поддерживает ли C++Builder открытые массивы Объектного Паскаля?
3. Какие новые классы, относящиеся к Delphi, были добавлены в C++Builder?
4. К какой версии Delphi наиболее близок C++Builder?
5. Когда вы создаете новую форму и начинаете помещать в нее компоненты, где хранится определение класса формы при ее визуальной разработке?

Упражнения

1. Инсталлируйте в C++Builder компонент, изначально разработанный для Delphi.
2. Откройте в C++Builder форму, созданную в Delphi.
3. Откройте один из .MAK-файлов C++Builder в редакторе кода и рассмотрите синтаксис, который в нем используется.
4. Разработайте простое приложение, состоящее из одной формы, в Delphi и C++Builder. Затем сравните количество строк кода, которые должны компилироваться в каждом случае для создания исполняемого файла.

Подведение итогов



Это была исключительно продуктивная неделя, не правда ли? Вы, вероятно, и не подозревали, что операции с базами данных могут быть так просты. Простота эта является относительной. Я не буду заходить слишком далеко, утверждая, что программирование баз данных вообще не представляет труда, но C++Builder несомненно делает его проще по сравнению с другими библиотеками Си++.

Я обещал, что вы узнаете о DLL и после этого сможете решать, использовать ли DLL в своих приложениях. В любом случае, вы теперь увидели преимущества DLL и можете принять обоснованное решение. Если вы планируете вызывать формы C++Builder из программ, написанных не в C++Builder, то вам не обойтись без DLL. Поначалу работа с DLL может вызывать затруднения, но, как я говорил, секрет заключается в правильном написании заголовков. После того, как это будет сделано, остальное представляет собой обычное программирование на Си++. DLL не слишком сложны, но, повторю еще раз, вам потребуется небольшой опыт.

На двадцатый день вы учились писать свои собственные компоненты. Это занятие либо понравилось вам, либо оставило вас в растерянности. В последнем случае не беспокойтесь. Возможно, вам никогда не придется этого делать. Существует масса других источников хороших, профессионально написанных компонентов. К тому же, вы всегда можете вернуться назад и снова взяться за День 20, имея некоторый опыт работы с C++Builder. Если вам понравилось изучать написание компонентов, то, вероятно, было трудно заставить себя прочитать последнюю главу! Я готов спорить, что в какой-то момент двадцатого дня можно было услышать, как вы говорите: «Класс!» Написание своих собственных компонентов действительно увлекает. Но нужно изучить гораздо больше, и многое из этого может быть изучено только на практике. К счастью, эта глава дает вам достаточно хорошую основу.

Наконец, вы узнали, как можно использовать код C++Builder (например, формы и DLL) в приложениях, написанных в других средах программирования. Возможность вызова форм C++Builder из других приложений является важным свойством этой среды. Понимание того, как использовать C++Builder совместно с другими средами программирования, делает вас более ценным программистом.

Создание приложений для Internet



Кен Хендерсон

Сегодня вы узнаете, как с помощью C++Builder создаются приложения для Internet. Поскольку C++Builder поставляется с набором Internet-ориентированных элементов управления OCX, он идеально подходит для построения таких приложений. В этой главе вы узнаете о назначении Internet-компонентов и используете их для создания своего собственного Web-браузера. Сегодняшний урок состоит из следующих частей:

- Общие сведения о построении Internet-приложений с помощью C++Builder
- Обзор вкладки Internet палитры компонентов C++Builder
- Учебное занятие, в течение которого вы пройдете через весь процесс создания своего собственного Web-браузера с помощью C++Builder.

Общие сведения

Поскольку C++Builder основан на Си++, а Си++ является очень популярным языком программирования, вы можете делать в C++Builder практически все, что угодно, в том числе и создавать Internet-приложения. Соответствующие спецификации опубликованы, необходимые интерфейсы разработаны, так что при желании можно писать эти приложения прямо «с нуля».

Но такой подход лишает смысла использование инструмента типа C++Builder. Вы используете средства RAD из-за выигрыша в скорости, который они дают. Однако вас не устроит нечто наскоро скомпилированное — вам нужен инструмент, который эффективно генерирует *эффективные* исполняемые файлы.

C++Builder замечателен своей способностью создавать с помощью техники drag-and-drop файлы для непосредственного выполнения процессором. Как и его старший родственник, Delphi, C++Builder сочетает в себе лучшее из объектно-ориентированной и компонентной моделей, предоставляя легкую в использовании и чрезвычайно мощную платформу для разработки приложений.

Что это должно означать применительно к разработке Internet-приложений? Для начинающих скажу, что благодаря набору Internet-компонентов вы можете создавать приложения для Internet так же легко, как и приложения любых других типов — помещая компоненты в форму. Вы можете также расширить возможности этих компонентов, сохраняя в любой момент доступ к низкоуровневым деталям API, поскольку C++Builder является собственным компилятором и основан на Си++.

Для непосредственной работы с низкоуровневыми API (например, ISAPI), что требуется при разработке приложений типа клиент/сервер для Internet, выбранный вами инструмент должен уметь использовать и конструировать DLL, а также использовать внешние интерфейсы к этим DLL. C++Builder отвечает этим всем этим требованиям. Кроме того, из-за использования в качестве базового языка Си++, для него непосредственно доступны интерфейсы всех основных API, относящихся к Internet. По отношению к разработке приложений Win32, C++Builder подобен швейцарскому армейскому ножу с множеством лезвий. Будь то игра, база данных или Internet-приложение, вы обнаружите, что C++Builder дает вам возможность делать все, что нужно, и даже позволяет в какой-то степени получать удовольствие от этого процесса.

Обзор вкладки Internet палитры компонентов

Прежде, чем мы приступим к созданию своего Web-браузера, давайте коротко познакомимся с компонентами вкладки Internet палитры компонентов C++Builder. Хотя в сегодняшнем проекте вы будете использовать только один из них, полезно знать, что же вам доступно. В таблице 22.1 перечислены все компоненты и описаны их функции.

Компонент	Описание
TFTP	Обеспечивает обмен файлами между удаленными компьютерами по протоколу FTP (File Transfer Protocol).
THTML	Обеспечивает анализ и компоновку HTML-данных, предоставляет возможность просмотра текущей HTML-страницы
THTTP	Обеспечивает доступ к протоколу передачи гипертекста HTTP (Hypertext Transfer Protocol)
TNNTP	Обеспечивает возможность участия в телеконференциях
TPOP	Обеспечивает доступ к протоколу POP3 электронной почты
TSMTP	Обеспечивает пересылку почтовых сообщений на серверы SMTP
TTCP	Обеспечивает доступ к протоколу TCP (Transfer Control Protocol) WinSock (со стороны как клиента, так и сервера)
TUDP	Обеспечивает доступ к протоколу пользовательских дейтаграмм (User Datagram Protocol) Winsock (со стороны как клиента, так и сервера)

Некоторые из этих компонентов обеспечивают доступ к популярным сервисам Internet, с которыми вы, вероятно, уже освоились. Например, если вы собираетесь создавать утилиту FTP, то нужно использовать компонент TFTP. Для создания программы чтения новостей потребуется компонент NNTP. Чтобы сконструировать Web-браузер (чем мы будем заниматься се-

годня), вам нужен компонент THTML. Если вы захотите посылать через Internet почту, то должны будете использовать компонент TPOP или TSMTP. Эта группа высокоуровневых элементов управления делает ненужной большую часть непосредственной работы с Winsock и позволяет моментально разрабатывать сложные Internet-приложения.

Создание собственного Web-браузера

Теперь, когда я познакомил вас с Internet-компонентами C++Builder, давайте создадим простой Web-браузер. Цель этого упражнения — продемонстрировать, насколько легко конструируются Internet-приложения, для создания которых еще пару лет назад приходилось затрачивать немало усилий. Реализованный в C++Builder подход к разработке, основанный на компонентах, позволяет вам сконцентрироваться на том, *что* вы хотите сделать, а не на том, *как* это делать.

Давайте начнем с открытия нового проекта. Выберите в системе меню C++Builder пункт File | New Application для инициализации проекта приложения. После этого перед вами окажется новая пустая форма, в которую можно помещать компоненты. Эта форма и будет вашим Web-браузером. Измените значение ее свойства Caption на Surf, а свойства Position — на poScreenCenter. Перед тем как продолжить, сохраните новое приложение как Surf.cpp, а связанную с ним форму — как Surf00.cpp. Я предлагаю вам использовать похожие, но различные имена, поскольку файлы приложения и формы не могут иметь одно и то же имя. Для приложений, состоящих из одной формы, это может показаться немного странным, но таково требование C++Builder.

Начните конструирование формы своего Web-браузера с помещения компонента TTabControl в ее верхнюю часть. Этот элемент управления будет содержать закладки для просмотренных URL, чтобы вы легко смогли вернуться на тот адрес, где уже побывали. Установите для свойства Align значение alTop. В результате при любом масштабировании формы ваш компонент всегда будет занимать ее верхнюю часть. Вы можете перетащить нижний край TTabControl для изменения его размера по вертикали.

Затем установите для свойства TabWidth компонента TTabControl значение в районе 145 пикселов. Это обеспечит каждой вкладке достаточный размер для отображения простого URL, не позволяя ей при этом занимать слишком много места. На рис. 22.1 показано, как должен выглядеть ваш TTabControl.

Теперь, когда верхняя часть формы браузера готова, давайте займемся низом. Поместите компонент TPanel в нижнюю часть формы и установите для его свойства Align значение alBottom. Как и в случае компонента TTabControl, это обеспечит постоянное размещение TPanel в нижней части формы независимо от ее размера.

Нажмите F11 для вызова инспектора объектов, затем сотрите у компонента TPanel значение свойства Caption. Установите свойство TabOrder в 0. Это приведет к тому, что при первом отображении формы по умолчанию будет выделен компонент, размещенный на TPanel, и являющийся, в свою очередь, компонентом по умолчанию для TPanel.

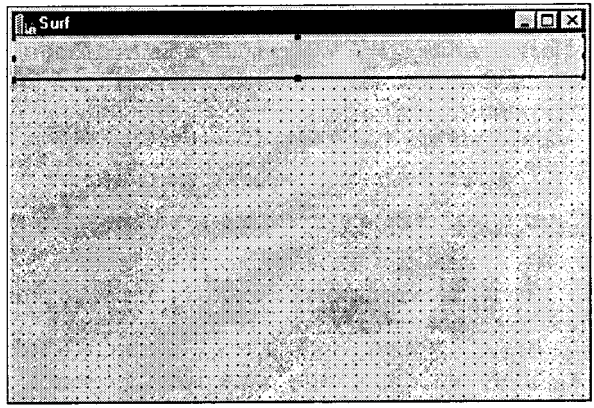


Рис. 22.1
Форма вашего Web-браузера с размещенным компонентом TTabControl

Закончив настройку TPanel, вы можете помещать на него компоненты-потомки. Начните с размещения на его правом краю другого компонента TPanel. Установите для свойства Align нового компонента значение alRight и сотрите значение свойства Caption. Это позволит вам располагать элементы управления так, чтобы они всегда были в правой части формы независимо от ее размера.

Затем поместите на новый TPanel два компонента TBitBtn. Установите для свойства Caption первой кнопки значение Go!, а второй — значение Home. С помощью свойства Glyph вы можете присвоить каждой кнопке какое-нибудь растровое изображение. Свойство Default кнопки Go! установите в true. Этот даст возможность пользователю просто ввести URL и затем нажать Enter для перехода к нему. При нажатии кнопки Go! браузер будет пытаться открыть выбранный URL, а кнопка Home позволит сразу перейти на вашу домашнюю страницу. Скоро я покажу вам, как указать домашнюю страницу в исходном коде браузера.

После того, как кнопки размещены, поместите компонент TComboBox с левого края вашего нижнего TPanel, максимально растяните его по горизонтали и сотрите значение свойства Text. Вы будете использовать этот комбинированный список для ввода адресов Web-страниц. После запуска приложения вам нужно будет ввести в TComboBox соответствующий URL и щелкнуть на кнопке Go!. На рис. 22.2 показано, как сейчас должна выглядеть ваша форма.

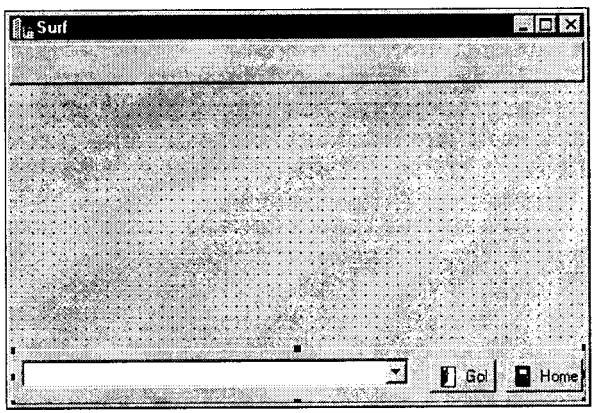
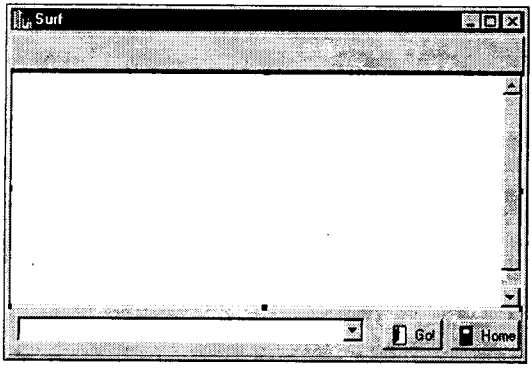


Рис. 22.2
Форма вашего Web-браузера с завершенными верхней и нижней частями

Теперь, когда верх и низ формы вашего Web-браузера завершены, давайте займемся серединой. Поместите в центр формы компонент HTML и установите для его свойства Align значение alClient. В результате компонент займет все пространство формы, не занятое верхним TTabControl и нижним TPanel. При изменении размера формы ее область, отведенная под HTML, будет также сжиматься или расширяться. Форма со всеми элементами управления показана на рис. 22.3.

Рис. 22.3
Ваша новая форма со всеми визуальными компонентами



«Оживление» новой формы

Когда форма полностью разработана визуально, вам остается связать с ее событиями код, чтобы выполнялись какие-либо действия. Самое крупное событие, которое вам предстоит настроить — это OnBeginRetrieval компонента THTML. Разумеется, вы также должны связать код с событиями OnClick обеих кнопок, а также с событием OnChange компонента TTabControl.

Для начала щелкните на TTabControl и затем дважды щелкните на событии OnChange в окне инспектора объектов. Введите следующий код в окне редактора кода C++Builder:

```
cbURL->Text = tcURL->Tabs->Strings[tcURL->TabIndex];
cbURL->SelectAll();
bbGo->Click();
```



В этом и других фрагментах кода данной главы предполагается, что вы придерживаетесь определенного соглашения об именах объектов вашего приложения. Я всегда ставлю в начале идентификатора двухсимвольный префикс, который позволяет мне определить тип соответствующего компонента. В данном случае cbURL относится к TComboBox, размещенному на нижнем TPanel, tcURL — к TTabControl в верхней части формы, а bbGo — к компоненту TBitBtn кнопки Go!.

Установка для свойства Text объекта cbURL значения текста вкладки, которая будет щелкнута при инициировании события OnChange, обеспечивает синхронизацию компонентов cbURL и tcURL. Вызов метода SelectAll() обеспечивает выделение значения свойства Text компонента TComboBox. Вызов события Click() компонента bbGo имитирует нажатие кнопки. Это позволяет сделать эквивалентными щелчок на вкладке и кнопке Go!.

Теперь, когда элемент управления вкладками настроен, щелкните на компоненте THTML и затем дважды щелкните на его событии OnBeginRetrieval в окне инспектора объектов. Введите следующий код в окне редактора кода:

```
Cursor = (TCursor)crHourGlass;
if (cbURL->Items->IndexOf(htBrowser->URL) == -1) {
    cbURL->Items->Add(htBrowser->URL);
    cbURL->Text = htBrowser->URL;
    cbURL->SelectAll();
    tcURL->Tabs->Add(htBrowser->URL);
    tcURL->TabIndex = tcURL->Tabs->Count - 1;
    bbGo->Click();
}
```

Этот код проверяет, содержится ли в списке компонента TComboBox тот URL, который должен быть открыт. Если URL там отсутствует, он добавляется одновременно к TComboBox и TTabControl.

Обратите внимание на присваивание значения свойству Cursor формы. Это показывает, что форма находится в процессе загрузки Web-страницы.

Конечно, если курсор принял вид песочных часов, его прежний вид должен быть в какой-то момент восстановлен. Подходящее место для этого — событие OnEndRetrieval компонента THTML. Введите следующий код в обработчик события OnEndRetrieval:

```
Cursor = (TCursor)crDefault;
```

Эта строка будет восстанавливать курсор после загрузки страницы.

Теперь, когда компоненты TTabControl и THTML настроены, можно заняться кнопками. Дважды щелкните на кнопке Go! в редакторе форм для вызова обработчика события OnClick. Введите следующий код:

```
htBrowser->RequestDoc(cbURL->Text);
```

Это просто указание компоненту THTML (с именем htBrowser) открыть URL из TComboBox (с именем cbURL).

Когда вы закончите с кнопкой Go!, дважды щелкните на кнопке Home. Введите следующий текст в редакторе кода:

```
htBrowser->RequestDoc(HomePage);
```

Этот код содержит константу с именем HomePage, которую вам следует предварительно определить. Перейдите в начало Surf00.cpp и объявите переменную HomePage примерно следующим образом:

```
String HomePage = "http://www.borland.com";
```

При желании вы можете поместить объявление HomePage сразу же после объявления экземпляра вашей формы — это помогает держать вместе глобальные переменные формы.

На этом ваш очень простой браузер закончен. Листинги 22.1–22.3 содержат исходный код вашего нового приложения в том виде, который он должен иметь на данный момент.

Листинг 22.1. Заголовочный файл главной формы вашего Web-браузера (Surf00.h)

```
 1: //---------------------------------------------------------------------
 2: #ifndef surf00H
 3: #define surf00H
 4: //---------------------------------------------------------------------
 5: #include <vcl\Classes.hpp>
 6: #include <vcl\Controls.hpp>
 7: #include <vcl\StdCtrls.hpp>
 8: #include <vcl\Forms.hpp>
 9: #include <vcl\ComCtrls.hpp>
10: #include <vcl\ExtCtrls.hpp>
11: #include <vcl\Buttons.hpp>
12: #include <vcl\ISP.hpp>
13: #include <vcl\OleCtrls.hpp>
14: //---------------------------------------------------------------------
15: class TForm1 : public Tform
16: {
17: __published:    // Компоненты, управляемые IDE
18:     TTabControl *tcURL;
19:     TPanel *Panel1;
20:     TPanel *Panel2;
21:     TBitBtn *bbGo;
22:     TBitBtn *bbHome;
23:     TComboBox *cbURL;
24:     THTML *htBrowser;
25:     void __fastcall tcURLChange(TObject *Sender);
26:     void __fastcall htBrowserBeginRetrieval(TObject *Sender);
27:     void __fastcall htBrowserEndRetrieval(TObject *Sender);
28:     void __fastcall bbGoClick(TObject *Sender);
29:     void __fastcall bbHomeClick(TObject *Sender);
30: private:         // Объявления пользователя
31: public:          // Объявления пользователя
32:     virtual __fastcall TForm1(TComponent* Owner);
33: };
34: //---------------------------------------------------------------------
35: extern TForm1 *Form1;
36: //---------------------------------------------------------------------
37: #endif
```

Листинг 22.2. Исходный текст главной формы вашего Web-браузера (Surf00.cpp)

```
 1: //---------------------------------------------------------------------
 2: #include <vcl\vcl.h>
 3: #pragma hdrstop
 4:
 5: #include "surf00.h"
 6: //---------------------------------------------------------------------
 7: #pragma resource "*.dfm"
 8: TForm1 *Form1;
 9: String HomePage = "http://www.borland.com";
10: //---------------------------------------------------------------------
11: __fastcall TForm1::TForm1(TComponent* Owner)
12:     : TForm(Owner)
13: {
14:     cbURL->Text = HomePage;
```

см. продолжение

Листинг 22.2. (продолжение)

```
15: }
16: //---------------------------------------------------------------------
17: void __fastcall TForm1::tcURLChange(TObject *Sender)
18: {
19:     cbURL->Text = tcURL->Tabs->Strings[tcURL->TabIndex];
20:     cbURL->SelectAll();
21:     bbGo->Click();
22: }
23: //---------------------------------------------------------------------
24: void __fastcall TForm1::htBrowserBeginRetrieval(TObject *Sender)
25: {
26:     Cursor=(TCursor)crHourGlass;
27:     if (cbURL->Items->IndexOf(htBrowser->URL) == -1) {
28:         cbURL->Items->Add(htBrowser->URL);
29:         cbURL->Text = htBrowser->URL;
30:         cbURL->SelectAll();
31:         tcURL->Tabs->Add(htBrowser->URL);
32:         tcURL->TabIndex = tcURL->Tabs->Count - 1;
33:         bbGo->Click();
34:     }
35: }
36: //---------------------------------------------------------------------
37: void __fastcall TForm1::htBrowserEndRetrieval(TObject *Sender)
38: {
39:     Cursor = (TCursor)crDefault;
40: }
41: //---------------------------------------------------------------------
42: void __fastcall TForm1::bbGoClick(TObject *Sender)
43: {
44:     htBrowser->RequestDoc(cbURL->Text);
45: }
46: //---------------------------------------------------------------------
47: void __fastcall TForm1::bbHomeClick(TObject *Sender)
48: {
49:     htBrowser->RequestDoc(HomePage);
50: }
51: //---------------------------------------------------------------------
```

Листинг 22.3. Текстовое представление главной формы вашего Web-браузера

```
 1: object Form1: TForm1
 2:   Left = 199
 3:   Top = 147
 4:   Width = 435
 5:   Height = 300
 6:   Caption = 'Surf'
 7:   Font.Charset = DEFAULT_CHARSET
 8:   Font.Color = clWindowText
 9:   Font.Height = -11
10:   Font.Name = 'MS Sans Serif'
11:   Font.Style = []
12:   Position = poScreenCenter
13:   PixelsPerInch = 96
14:   TextHeight = 13
15:   object tcURL: TTabControl
```

```
16:      Left = 0
17:      Top = 0
18:      Width = 427
19:      Height = 33
20:      Align = alTop
21:      TabOrder = 1
22:      TabWidth = 145
23:      OnChange = tcURLChange
24:    end
25:    object Panel1: TPanel
26:      Left = 0
27:      Top = 232
28:      Width = 427
29:      Height = 41
30:      Align = alBottom
31:      TabOrder = 0
32:      object Panel2: TPanel
33:        Left = 304
34:        Top = 1
35:        Width = 122
36:        Height = 39
37:        Align = alRight
38:        BevelOuter = bvNone
39:        TabOrder = 0
40:        object bbGo: TBitBtn
41:          Left = 8
42:          Top = 8
43:          Width = 49
44:          Height = 25
45:          Caption = 'Go!'
46:          Default = True
47:          TabOrder = 0
48:          OnClick = bbGoClick
49:          Glyph.Data = {
50:  78010000424D78010000000000007600000028000000200000000100000000100
51:  04000000000000000000120B0000120B0000000000000000000000000000000000
52:  80000080000000808000800000008000800080800000007F7F7F00BFBFBF000000
53:  FF0000FF000000FFFF00FF000000FF00FF00FFFF0000FFFFFF00330000000000
54:  03333377777777777F333301BBBBBBBB033333773F3333337F3333011BBBBBBB
55:  0333337F73F333337F33330111BBBBBB0333337F373F33337F333301110BBBBB
56:  0333337F337F33337F333301110BBBBB0333337F337F33337F333301110BBBBB
57:  0333337F337F33337F333301110BBBBB0333337F337F33337F333301110BBBBB
58:  0333337F337F33337F333301110BBBBB0333337F337FF3337F33330111B0BBBB
59:  0333337F337733337F333301110BBBBB0333337F337F33337F333301110BBBBB
60:  0333337F3F7F33337F333301E10BBBBB0333337F7F7F33337F333301EE0BBBBB
61:  0333337F777FFFFF7F33330000000000033333777777777773330000}
62:          NumGlyphs = 2
63:        end
64:        object bbHome: TBitBtn
65:          Left = 64
66:          Top = 8
67:          Width = 57
68:          Height = 25
69:          Caption = 'Home'
70:          TabOrder = 1
71:          OnClick = bbHomeClick
72:          Glyph.Data = {
```

см. продолжение

Листинг 22.3. (продолжение)

```
 73:  78010000424D78010000000000007600000028000000200000001000000001000
 74:  0400000000000000000120B0000120B0000000000000000000000000000000000
 75:  8000008000000080800080000000800080008080000007F7F7F00BFBFBF000000
 76:  FF0000FF000000FFFF00FF000000FF00FF00FFFF0000FFFFFF00330000000000
 77:  0333337777777777F33330111111111033337F333333337F33330111111111
 78:  0333337F333333337F33330111111111033337F333333337F33330111111111
 79:  0333337F333333337F33330111111111033337F333333337F33330111111111
 80:  0333337F3333333F7F333301111111B10333337F333333737F33330111111111
 81:  0333337F333333337F33330111111111033337F33FFFFF37F3333011EEEEE11
 82:  0333337F377777F37F3333011EEEEE110333337F37FFF7F37F3333011EEEEE11
 83:  0333337F377777337F33330111111111033337F333333337F33330111111111
 84:  0333337FFFFFFFFF7F3333000000000003333377777777773330000}
 85:        NumGlyphs = 2
 86:      end
 87:    end
 88:    object cbURL: TComboBox
 89:      Left = 8
 90:      Top = 8
 91:      Width = 289
 92:      Height = 21
 93:      ItemHeight = 13
 94:      TabOrder = 1
 95:    end
 96:   end
 97:   object htBrowser: THTML
 98:     Left = 0
 99:     Top = 33
100:     Width = 427
101:     Height = 199
102:     Align = alClient
103:     ParentColor = False
104:     ParentFont = False
105:     TabOrder = 2
106:     OnBeginRetrieval = htBrowserBeginRetrieval
107:     OnEndRetrieval = htBrowserEndRetrieval
108:     ControlData = {
109:  21433412080000000222C0000911400000000001010100011E0000000100000000
110:  C0C0C000000000000000FF00FF00FF0001CC0000900144420100OD4D53205361
111:  6E732053657269660100000090010C7C01000B436F7572696572204E65770100
112:  0000BC0280A903000F54696D6573204E657720526F6D616E01000000BC0220BF
113:  02000F54696D6573204E657720526F6D616E01000000BC02580F02000F54696D
114:  6573204E657720526F6D616E01000000BC02C0D401000F54696D6573204E6577
115:  20526F6D616E01000000BC02DC7C01000F54696D6573204E657720526F6D616E
116:  01000000BC02F82401000F54696D6573204E657720526F6D616E0000}
117:   end
118: end
```

Теперь, когда Web-браузер полностью завершен, можно испытать его в действии. Сохраните проект, после чего нажмите F9 для запуска вашего нового приложения. На рис. 22.4 показано, как должен выглядеть ваш браузер во время выполнения.



Рис. 22.4
Ваш новый Web-браузер в действии

Заключение

Сегодня вы узнали о назначении каждого из компонентов во вкладке Internet палитры компонентов C++Builder. Вы прошли через процесс, включающий в себя выбор одного из этих компонентов, элемента управления THTML, и использование его для построения Internet-утилиты — в данном случае, Web-браузера. Если бы вам потребовался клиент электронной почты или программа для чтения новостей, этот процесс был бы во многом аналогичным. Архитектура C++Builder, основанная на компонентах, позволяет вам сосредоточиться на том, что вы хотите сделать, а не задумываться о том, как это реализовать.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Кто автор Internet-компонентов, которые поставляются с C++Builder?

Фирма Borland использует в C++Builder лицензионные Internet-компоненты от NetManage.

Если у меня уже инсталлирован Microsoft Internet Explorer 3.0, могу ли я как-нибудь использовать его в своих Web-приложениях вместо компонента HTML NetManage?

Да. Для этого нужно сделать следующее:

1. Выберите в главном меню C++Builder пункт Component | Install.
2. Щелкните на кнопке ActiveX.
3. После появления диалогового окна Import OLE Control (Импортировать элемент управления OLE), выберите Microsoft Internet Controls из списка Registered Controls (Зарегистрированные элементы).

4. Проверьте, чтобы в поле Palette page стояло Internet — при этом новый компонент будет инсталлирован во вкладку Internet палитры компонентов C++Builder.
5. Щелкните на кнопке OK и еще раз на кнопке OK. Вы должны увидеть новый компонент с названием TWebBrowser во вкладке Internet палитры компонентов. Это браузер Microsoft Internet Explorer 3.0, реализованный как элемент управления OLE. Web-приложения, в которых вы используете этот компонент, будут иметь те же основные возможности, что и сам Internet Explorer 3.0.

Вопросы

1. Какой компонент C++Builder вы бы использовали при создании своего собственного приложения для чтения новостей?
2. Какое событие инициируется компонентом THTML перед открытием документа?
3. Какое свойство компонента THTML содержит адрес текущей Web-страницы?
4. Какой компонент вы будете использовать при разработке утилиты FTP?

Упражнения

1. Измените браузер, созданный сегодня, таким образом, чтобы в нем использовался элемент OCX Microsoft Internet Explorer.
2. Создайте приложение C++Builder, которое позволило бы вам участвовать в телеконференциях.
3. Напишите обработчик исключительной ситуации C++Builder, который посылает e-mail при генерации этого исключения в приложении.
4. Создайте приложения сервер и клиент WinSock, которые связываются между собой с использованием компонента TCP.
5. Создайте приложение, которое автоматически связывается с заданным FTP-сервером и загружает все файлы, которых у вас нет.

Расширения Си++ в C++Builder



Кент Рейсдорф

C++Builder — это мощный инструмент RAD. Своей мощью он в значительной степени обязан модели свойств, методов и событий (PME — properties, methods, and events). Однако в языке Си++ отсутствуют некоторые возможности, необходимые для полной реализации модели PME. Сейчас мы рассмотрим расширения Си++, содержащиеся в C++Builder, но прежде всего мне хотелось бы сказать о том, что привело нас к этому обсуждению.

Немного истории

В течение последних двух лет пользователи Borland C++ (и вообще программисты на Си++) испытывали острую потребность в инструменте RAD, основанном на Си++. Borland Delphi, несмотря на всю свою мощь, основан на Объектном Паскале. Объектный Паскаль для среднего разработчика обладает не меньшими возможностями, чем Си++, но, как говорится, имидж решает все. Существует множество организаций (общественных, частных, правительственных), которые наотрез отказываются использовать продукты, основанные на Паскале. Очевидно, что инструмент RAD на Си++ был необходим.

Фирма Borland начала работать над решением стоящей задачи. При этом имело смысл использовать популярность Delphi, а также существующую IDE Delphi. После того успеха, который имела эта среда программирования, люди требовали «Delphi для Си++», так почему бы не дать им то, что они хотели? Было очевидно, что использование IDE Delphi не представляет сложностей и является самым быстрым решением. Не имело никакого смысла модифицировать IDE Borland C++ (или скрещивать среды Borland C++ и Delphi), когда IDE Delphi содержала большую часть того, что пользователи хотели бы иметь, не говоря уже о том, что это было готовым решением. Таким образом, эта часть задачи решалась легко.

Трудности начались, когда в Borland стали рассматривать возможность реализации модели PME на Си++ при сохранении стопроцентного соответствия стандарту ANSI. Ситуация осложнялась тем, что Borland C++ был одним из немногих компиляторов Си++ для PC, который все время обеспе-

чивал соответствие стандарту ANSI. (Microsoft ввела в Visual C++ много расширений, которые не давали ей права заявлять о полной совместимости.) Тем не менее, реализовать модель PME в том виде, какой она имела в Delphi, и остаться в рамках существующих стандартов было невозможно. Было решено расширить Си++, чтобы использовать всю мощь модели PME и существующей библиотеки визуальных компонентов. Теперь вопрос состоял не в том, как это сделать, а скорее в том, решаться на такой шаг или нет?

В сомнительной ситуации лучше узнать мнение общественности. Borland провела маркетинговые исследования, чтобы узнать точку зрения разработчиков программного обеспечения. Эти исследования были достаточно обширными. Borland хотела получить ответ на один очень важный вопрос: «Что более важно для вас: быстрая разработка приложений или совместимость со стандартом ANSI?» Результаты опросов показали, что строгое соответствие стандартам Си++ для большинства программистов уже не так важно, как когда-то. Создалось общее ощущение, что RAD имеет большую ценность, чем приверженность стандартам. Вооружившись этим, Borland продолжила добавлять к Си++, реализованном в C++Builder, те расширения, которые требовались для модели PME.



Необходимо заметить, что ничто в C++Builder не препятствует компиляции Си++ кода, на 100 процентов соответствующего стандарту ANSI. Расширения Си++ работают, пока вы в них нуждаетесь, но это никак не влияет на способность C++Builder компилировать программы на стандартном Си++. Если вы хотите строго соблюдать требования стандартов, то по-прежнему можете использовать C++Builder в качестве компилятора. Если, однако, вы захотите использовать возможности RAD, заложенные в C++Builder, вам придется принять соответствующие расширения Си++. Я уверен, что начав работать, вы легко к ним привыкнете.

Расширения языка: великие дебаты

Примерно в 1994 году фирма Borland начала работу над инструментом RAD, получившим кодовое название Delphi. Уже было решено, что лучшим способом реализации RAD является архитектура, основанная на компонентах, и теперь стояла задача выбора языка программирования, который должен был стать сердцем системы. В то время Borland была единственным серьезным поставщиком компиляторов с Паскаля. Она имела репутацию компании, производящей лучшие инструментальные средства для этого языка. Если вы программировали на Паскале, то с большой вероятностью использовали одну из версий Турбо-Паскаля. Borland в той или иной степени «присвоила» себе этот язык. Хотя Borland и не являлась владельцем Паскаля в юридическом смысле, из-за своих позиций на рынке она имела значительную свободу в реализации новых возможностей и расширений языка. Кроме того, не существовало комитета по стандартам Паскаля, с которым могли возникнуть трения. В результате Borland создала Delphi.

Еще до появления Delphi, Borland уже изменила язык Паскаль в лучшую сторону. Он был расширен путем создания нового языка, получившего название Объектный Паскаль (Object Pascal). Можно сказать, что Объектный Паскаль и обычный Паскаль соотносятся так же, как Си++ и Си. В Объ-

ектный Паскаль были добавлены классы, что ввело Паскаль в мир объектно-ориентированных языков. Когда наступило время Delphi, было достаточно легко добавить новые ключевые слова и новые правила, необходимые для реализации модели компонентов. Наряду со многими другими, появились ключевые слова published и property. Это позволило Borland полностью реализовать те немалые возможности, которые давала модель компонентов. Модифицировав таким образом Паскаль, Borland получила возможность сделать разработку приложений действительно быстрой. По существу, Объектный Паскаль модифицировался по мере необходимости, возникавшей в процессе разработки неизвестного тогда продукта с названием Delphi. В результате появился язык, который естественным образом работает с моделью PME (моделью компонентов).

Несмотря на то, что модификация Паскаля стала достаточно смелым шагом со стороны Borland, подобные прецеденты уже были. Еще раньше Microsoft взяла язык Бейсик и изменила его, создав новый язык с названием Visual Basic. В этом новом языке было трудно узнать тот Бейсик, который послужил в качестве основы. Можно сказать, что Microsoft пошла на риск, создавая Visual Basic, но Microsoft — это все же Microsoft, и она могла позволить себе рисковать. Язык Бейсик не слишком широко использовался, и поэтому почти не было слышно возмущенных криков, когда Microsoft выступила со своим Visual Basic (крики были, но они не носили негативного характера). Borland пошла на больший риск, модифицируя Паскаль. У этой фирмы было достаточно много приверженцев, и они могли отнестись неодобрительно к расширению языка, который изучили и полюбили. Тем не менее, Borland занимала твердые позиции на рынке Паскаля, и имела далеко идущие планы. Delphi стал точным попаданием (внутреннее кодовое имя прижилось и стало официальным названием продукта).

Но модификация Си++ — это совсем другое дело. Как я сказал в начале главы, мир программирования долгое время испытывал потребность в продукте RAD, основанном на Си++, поэтому сомнений в его необходимости не было. Вопрос заключался в том, как это сделать. Проще говоря, модель компонентов, столь блестяще реализованная в Delphi, не могла быть эмулирована существующими средствами Си++. Необходим был какой-то компромисс.

С модификацией Си++ связано много проблем. Прежде всего, существует комитет по стандарту Си++. Именно к его обязанностям относится решение вопросов о том, что должно быть включено в язык и как это следует реализовать (для Паскаля или Бейсика такого комитета нет). Можно написать целую главу о том, как работает эта система, но я опущу детали. Достаточно сказать, что уже само существование комитета по стандарту Си++ означает, что любое вмешательство в язык будет воспринято критично. С другой стороны, все производители компиляторов в той или иной форме реализуют расширения Си++. Теоретически вы должны иметь возможность взять, к примеру, программу MFC и откомпилировать ее на любом компиляторе Си++. В конце концов, ведь существует стандарт, не так ли? К сожалению, все не так просто. MFC содержит определенное количество нестандартных расширений Си++, поэтому вам не удастся, скажем, беспрепятственно откомпилировать MFC с помощью компилятора фирмы Borland (Borland C++ 5.0 способен компилировать программы MFC, хотя для этого пришлось обойти много правил Си++). Как видите, мир Си++ более чем несовершенен.

Одной из других проблем с расширением Си++ является отношение к этому общественности. Поскольку считается, что Си++ должен иметь стандарт, любое изменение языка может нервировать некоторых консервативных программистов. Это, вероятно, основная причина слегка задуматься перед введением в Си++ расширений, противоречащих стандарту. В конце концов, эти люди тоже ваши покупатели, и лучше узнать их мнение до того, как вы начнете расширять выбранный ими язык. Как я уже говорил, Borland провела большие исследования, чтобы определить общественное мнение по этому вопросу. В результате выяснилось, что многие программисты были готовы согласиться с расширением языка, чтобы иметь продукт для быстрой разработки приложений на Си++. Для них споры вокруг языка оказались менее существенны, чем перспектива повысить эффективность работы.

Еще одна проблема с расширением Си++ заключается непосредственно в реализации. Хотя эта задача, конечно, является грандиозной, у меня не было сомнений, что разработчики из Borland справятся с ней относительно легко. В этой фирме работают одни из лучших умов в индустрии программного обеспечения. Короче говоря, Borland шла на вполне оправданный риск, расширяя Си++ для полной поддержки модели PME.

Следует еще раз подчеркнуть, что расширения Си++, реализованные в C++Builder — это всего лишь расширения. Они предназначены для того, чтобы вы могли использовать преимущества быстрой разработки приложений, программируя на Си++. Основой C++Builder является стандартный компилятор Си++, реализующий некоторые из последних возможностей, объявленных комитетом по стандартам. Если вы не хотите использовать расширения, вас никто не заставляет это делать. В этом случае, однако, вы вряд ли будете слишком интенсивно использовать C++Builder (и скорее всего, не стали бы читать эту книгу!).

Новые ключевые слова Си++ в C++Builder

Итак, фактически C++Builder расширяет язык Си++, чтобы получить максимум от модели компонентов. Для реализации новых расширений языка в C++Builder введено несколько дополнительных ключевых слов:

```
__automated
__classid
__closure
__declspec (modified)
__int8
__int16
__int32
__int64
__property
__published
```

Самые очевидные из них, и, вероятно, самые важные — это __closure, __property и __published. Давайте рассмотрим каждое из новых ключевых слов отдельно.

__automated

Ключевое слово __automated похоже на ключевое слово public, за исключением того, что компилятор генерирует информацию автоматического типа для функций и членов-данных, объявленных в разделе __automated объявления класса. Это необходимо при использовании класса в качестве автоматизированного OLE-сервера. Это тема достаточно сложна и здесь детально не рассматривается.

__classid

Ключевое слово __classid генерирует указатель на таблицу виртуальных функций класса (vtable). __classid предназначено в C++Builder для внутреннего использования, поэтому не рекомендуется, чтобы вы применяли его в своих приложениях.

__closure

Клауза является специальным типом указателя на функцию. Обычный указатель представляет собой 4-байтную переменную, определяющую расположение функции внутри сегмента кода программы. Клауза имеет длину в 8 байт и содержит, кроме адреса самой функции (в первых 4 байтах), адрес объекта, в котором находится функция (последние 4 байта). Этот тип указателя на функцию жизненно важен для работы программ C++Builder. В Паскале он называется *указателем метода (method pointer)*. Клауза — это механизм, который позволяет вам во время разработки или выполнения программы связывать обработчик события, содержащийся в одном классе (например, в классе вашей главной формы), с событием, которое относится к другому классу (скажем, классу меню или кнопки). Это обычная практика при программировании в C++Builder, и может показаться неожиданным, что стандартный Си++ не имеет такой возможности. Ключевое слово __closure дополняет Си++ так, чтобы события и их обработчики могли работать требуемым образом.

Итак, клауза используется с событиями. Событие представляет собой свойство, которое в качестве типа данных использует указатель на функцию. Вы можете объявить клаузу так же, как и обычный указатель на функцию, добавив ключевое слово __closure:

```
void __fastcall (__closure *TMyEvent)(System::TObject *Sender);
```

Обычно вы будете применять ключевое слово __closure только при написании событий для компонентов. Прямое использование __closure в обычной практике требуется редко, но вы, тем не менее, будете интенсивно использовать клаузы, возможно, сами того не осознавая. Я считаю, что клаузы обязательно должны присутствовать в стандарте Си++.

__declspec

Ключевое слово __declspec использовалось и раньше в компиляторах Borland C++, но в C++Builder оно было расширено. Появились новые аргументы delphiclass и delphireturn.

Вы должны использовать __declspec(delphiclass) при предварительном объявлении класса, произведенного от TObject (независимо от того, прямо

или косвенно). __declspec(delphireturn) используется при предварительном объявлении класса, произведенного от Currency, AnsiString, Variant, TDateTime или Set.

__int8, __int16, __int32 и __int64

Эти новые ключевые слова позволяют вам указать размер целой переменной: 8, 16, 32 или 64 бита. В 32-разрядной среде Windows по умолчанию int имеет размер 32 бита (так же, как и long). Используя одно из этих ключевых слов, вы можете задать любой размер. Если, например, для целого числа вам достаточно диапазона от 0 до 255, используйте ключевое слово __int8 следующим образом:

```
__int8 x = GetResult();
```

Как и для стандартного ключевого слова int, все целые переменные, объявленные с помощью новых ключевых слов, будут знаковыми. Чтобы сделать переменную беззнаковой, используйте модификатор unsigned. Например, создать 16-разрядную беззнаковую целую переменную можно следующим образом:

```
unsigned __int16 left = MainForm->Left;
```

Используйте эти новые ключевые слова, когда вам нужна целая переменная определенного размера.

__property

Ключевое слово __property используется для объявления свойства в объявлении класса. Свойства легко спутать с обычными членами-данными класса, однако свойства имеет несколько характерных особенностей, которые отличают их от членов-данных:

- ☐ Со свойством можно связать метод чтения
- ☐ Со свойством можно связать метод записи
- ☐ Для свойства можно установить значение по умолчанию
- ☐ Свойство может сохраняться в файле формы
- ☐ Свойство, объявленное в базовом классе, может быть расширено

Методы чтения и записи

Запись в свойство часто вызывает некоторые действия внутри того компонента, к которому это свойство относится. Например, изменение значения свойства Color приводит к немедленному изменению цвета фона компонента без каких-либо дополнительных действий со стороны пользователя. Вторичные эффекты, наподобие этого, реализуются с помощью методов чтения и записи, связанных со свойством. Продолжим в качестве примера рассмотрение свойства Color. Оно может быть объявлено следующим образом:

```
__property int Color {read=GetColor, write=SetColor};
```

Эта строка содержит указание компилятору вызвать метод GetColor() класса (метод чтения) при чтении свойства Color, а при записи в свойство вызвать метод SetColor() класса (метод записи). Конкретная реализация методов GetColor() и SetColor() определяется автором компонента.

Вместо методов чтения и записи может использоваться прямой доступ. Свойство почти всегда имеет закрытый член класса, в котором хранится его значение. Обычно для изменений в компоненте при записи в свойство используется метод записи, а для чтения свойства метод отсутствует. В подобном случае вы должны указать для свойства метод записи, а чтение производить с помощью прямого доступа к члену класса, хранящему значение свойства. Объявление такого свойства выглядит следующим образом:

```
__property int Color {read=FColor, write=SetColor};
```

Теперь при записи в свойство метод SetColor() будет вызван, как и прежде, но при чтении вместо вызова метода чтения будет возвращено значение переменной FColor. (Переменная FColor должна быть объявлена в разделе private объявления класса компонента.)

В объявлении свойства не обязательно указывать спецификаторы read и write. Если спецификатор write опущен, свойство будет доступно только для чтения (read-only). При отсутствии спецификатора read свойство станет доступно только для записи (write-only) и не будет иметь определенного значения. Если не указаны оба спецификатора, свойство недоступно ни для чтения, ни для записи, что делает его бесполезным.

Значения по умолчанию

Для свойства можно указать значение по умолчанию, которое будет отображаться в инспекторе объектов. Пусть, например, ваше свойство Color должно по умолчанию соответствовать белому цвету (clWhite). Его объявление будет выглядеть следующим образом:

```
__property int Color {read=FColor, write=SetColor, default=clWhite};
```

Это указывает редактору форм отобразить в инспекторе объектов рядом со свойством значение clWhite. Заметьте, что это не означает автоматического присваивания значения clWhite соответствующему члену класса. Вы по-прежнему должны сделать это в конструкторе компонента. Использование спецификатора default не является обязательным. Если вы не указали значение по умолчанию, вместо него будет использован 0 (или false для свойства типа bool).

Хранение свойства

Спецификатор stored определяет способ сохранения свойства в файле формы при записи на диск. Значение свойства будет прочитано из файла при загрузке формы в редактор или при отображении формы приложением. По умолчанию все свойства из раздела __published объявления класса будут автоматически сохраняться. Свойства, объявленные в разделе public, не сохраняются автоматически. Для их сохранения вы должны установить stored равным true:

```
__property int Value {read=FValue, write=SetValue, stored=true};
```

Как я сказал, все свойства из раздела __published сохраняются автоматически. Именно это нужно вам в большинстве случаев, поэтому вы можете вообще никогда не использовать спецификатор stored для отображаемых свойств. Но если по некоторой причине вам потребуется предотвратить сохранение свойства, вы можете указать для спецификатора stored значение false:

```
__property int Value {read=FValue, write=SetValue, stored=false};
```

В большинстве случаев вы будете оставлять для сохранения свойств действия по умолчанию.

Расширение свойств базового класса

Расширение свойства, содержащегося в базовом классе создаваемого вами компонента, является обычной практикой. Одной из наиболее распространенных причин для этого является необходимость изменить значение по умолчанию. Пусть, например, вы создали новый компонент на основе компонента TLabel и этот новый компонент должен содержать текст, выровненный по центру. В этом случае вы можете переобъявить свойство Alignment следующим образом:

```
__property Alignment = {default=taCenter};
```

При помещении вашего нового компонента в форму, для свойства Alignment будет по умолчанию установлено значение taCenter. Обратите внимание, что вам нужно указать только те спецификаторы, которые необходимы для изменения свойства. В этом примере меняется спецификатор default, поэтому в списке присутствует только он один.

Также часто возникает необходимость изменить поведение компонента по умолчанию при записи в одно из свойств. В этом случае вам нужно переобъявить свойство, изменив только спецификатор write:

```
__property Alignment = {write=MyWriteAlignment};
```

При переобъявлении отображаемого свойства убедитесь, что вы поместили его в раздел __published объявления класса.

__published

Ключевое слово __published необходимо в тех случаях, когда вы хотите отображать свойство в инспекторе объектов на этапе разработки. Оно используется в объявлении класса аналогично ключевым словам public, private и protected. Типичное объявление класса для компонента C++Builder может выглядеть следующим образом:

```
class TFlashingLabel : public TCustomLabel
{
private:
  //
  // Закрытые члены-данные и функции
  //
protected:
  //
  // Защищенные члены-данные и функции
  //
public:
  //
  // Открытые члены-данные и функции
  //
__published:
  //
  // Отображаемые свойства компонента
  //
__property int FlashLimit = {read=FFlashLimit, write=FFlashLimit, de-
fault=0};
  // Объявления других свойств
};
```

Все свойства из раздела __published будут отображаться в инспекторе объектов на этапе разработки. Хотя реализация __published тривиальна, это одно из главных ключевых слов, если рассматривать расширения Си++ в C++Builder.

Заключение

Эволюция языков программирования определяется многими факторами. Один из них, наиболее стимулирующий развитие — это требование программистами тех возможностей, которые отсутствуют в текущем воплощении данного языка. Языки программирования редко бывают статичны. Они почти всегда изменяются в той или иной степени. Изменения в языке зачастую лишают комфорта многих его пользователей. Хотя я не сторонник изменений ради самих изменений, я безусловно принимаю расширения языка, когда в этом есть смысл.

Фирма Borland решила расширить язык Си++, и это является фактом. Этим расширениям C++Builder обязан большей частью своей мощи. Не имеет смысла дискутировать по этому поводу. Вы можете либо использовать C++Builder со всеми его возможностями и согласиться с расширениями Си++, либо остаться приверженцем стандартного Си++ и писать программы для Windows более сложным способом. Выбирать вам. Я уверен, что если вы дочитали книгу до этого места, то возможность быстрой разработки приложений, которую предоставляет C++Builder, перевесила для вас проблему расширений Си++. Если вы уже программировали на Си++, то вам, возможно, придется разобраться в сплетении новых расширений языка и решить, подходят ли они для вас. Если вы новичок в Си++, то можете и не понять, из-за чего, собственно, столько шума. Из какого бы лагеря вы не пришли, ваша работа наверняка станет более продуктивной при использовании той мощи, которую предлагает C++Builder. Эта мощь имеет определенную цену, но я думаю, что эта цена незначительна по сравнению с тем, что вы за нее получаете.

Вопросы и задания

Этот раздел содержит вопросы, которые помогут вам закрепить пройденный материал, и упражнения, которые дадут опыт практического применения изученного. Ответы на вопросы приведены в приложении А.

Вопросы и ответы

Мне нравится C++Builder, но я не знаю, как относиться к расширениям языка. Какие у меня есть варианты?

Ваш выбор очень прост: либо используйте C++Builder и научитесь принимать расширения Си++, либо не используйте C++Builder и откажитесь от RAD.

Является ли практика расширения Си++ широко распространенной?

В известной степени да, но обычно не в таких масштабах, как это сделано в C++Builder. Расширения языка в C++Builder достаточно существенны.

Верно ли, что расширения Си++ в C++Builder сделали его нестандартным компилятором?

Это не так. Наличие расширений не мешает C++Builder оставаться стандартным компилятором Си++. C++Builder — это стандартный компилятор *плюс* инструмент RAD, реализованный с помощью расширений языка.

Вопросы

1. Какие ключевые слова используются для объявления свойств?
2. Является ли событие клаузой?
3. Для чего нужно ключевое слово __published?
4. Является ли ключевое слово __declspec новым для C++Builder?

Упражнения

1. Изложите письменно свое мнение относительно того, допустимо ли расширять Си++, чтобы обеспечить RAD в программировании для Windows.
2. Поищите в различных сетевых источниках проект стандарта Си++. Прочитайте выдержки из него, чтобы получить лучшее представление о том, какие силы вовлечены в работу над языком типа Си++.
3. **Дополнительное задание:** напишите компонент, который использует отображаемое свойство.

Приложение
Ответы на вопросы



День 1

1. Отсутствует функция main(). Каждая программа должна содержать функцию main().
2. Одно. Функция может иметь много параметров, но возвращаемое значение всегда одно.
3. Функция strcpy() копирует содержимое одной строки в другую.
4. Неопределенное. До инициализации переменная будет содержать случайные данные.
5. На количество функций в программе ограничений нет.
6. Да.
7. Отсутствует объявление для функции doSomething().
8. Одну.
9. 19 символов плюс завершающий NULL.
10. Нуль.

День 2

1. Оператор, следующий непосредственно за оператором if. Если за оператором if следует блок кода, будет выполнен весь этот блок.
2. Первый параметр служит начальным значением, второй параметр — проверочным выражением, а последний — параметром инкремента.
3. В цикле while условное выражение вычисляется в начале цикла. В цикле do-while условное выражение вычисляется в конце цикла.
4. Оператор break используется для выхода из цикла. После него будет выполнен оператор, следующий за телом цикла. Оператор continue передает управление обратно на начало цикла.
5. Глобальная переменная — это переменная, которая видима повсюду в программе. К ней может обращаться любая функция программы.
6. Да. Структура может содержать любое количество членов-данных произвольного типа.
7. С помощью оператора прямого доступа (.). Вот пример:

```
record.LastName = "Noble";
```

8. Да.

День 3

1. Указатель — это переменная, содержащая адрес другой переменной или объекта в памяти.
2. Разыменование указателя означает взятие значения той переменной, на которую ссылается указатель, а не значения самого указателя (которое представляет собой лишь адрес в памяти).
3. Адрес в памяти, по которому располагается вновь созданный объект.
4. Обычно классы и структуры следует передавать по ссылке для исключения ненужных затрат.
5. Использование ключевого слова const предотвращает изменение переменной.
6. Нет. Перегруженные функции должны различаться типом и количеством параметров. Эти две функции различаются только возвращаемыми значениями.
7. Это зависит от конкретной ситуации. Однозначного ответа не существует.
8. Функция-член класса — это функция, принадлежащая классу.
9. Компилятор будет помещать в объектный файл полный код подставляемой функции каждый раз, когда эта функция встречается в исходном тексте программы. Обычная функция включается в объектный файл только один раз, а при появлении ее в исходном тексте компилятор будет вставлять вызов этой функции.
10. В функции должен использоваться оператор delete в форме delete[]:

```
delete[] buff;
```

День 4

1. Структура в Си++ — это просто класс, в котором все члены-данные и функции по умолчанию имеют открытый доступ. Помимо этого, других различий нет.
2. Использование закрытых членов-данных предотвращают прямое изменение данных пользователем класса. Их изменение возможно только через открытые функции-члены, но не напрямую.
3. С помощью открытых функций-членов, вызываемых для изменения закрытых членов-данных.
4. Деструктор вызывается при уничтожении объекта. Для локальных объектов это происходит, когда объект выходит из области видимости. Для динамически размещаемых объектов это происходит при выполненении оператора delete.
5. Замещение функции означает замену функции базового класса функцией вашего производного класса. Для этого новая функция должна иметь точно такое же имя, набор параметров и тип возвращаемого значения.
6. Вызвать функцию базового класса из замещающей функции:

```
void MyClass::DoIt()
{
  BaseClass::DoIt();
  // другой код
}
```

7. Список инициализаторов используется для инициализации членов-данных класса и вызова любых конструкторов базового класса до входа в тело конструктора.
8. Да. Это бывает очень часто.
9. С помощью множественного наследования. Создайте класс на основе двух различных базовых классов.

День 5

1. Нет. Только визуальные компоненты могут быть видимы во время разработки.
2. Ни одна из них. Каждая библиотека имеет свои достоинства и недостатки.
3. Нет. Объекты VCL могут размещаться только динамически (с помощью оператора new).
4. И да, и нет. В основном они эквивалентны. Однако VCL написана на Объектном Паскале, и поэтому перегруженных методов не существует.
5. Да.
6. TOpenDialog, TSaveDialog, TRegistry, TColorDialog, TTimer, TImageList, TFontDialog и многие другие.
7. Да. Все компоненты в конечном итоге произведены от TComponent, поэтому все они имеют свойства, принадлежащие TComponent, например, Name и Owner.
8. Top, Left, Owner, Parent, Width, Height и т. д.
9. Да.
10. Холст (canvas). VCL инкапсулирует контекст устройства в классе TCanvas.

День 6

1. Щелкните правой кнопкой мыши на панели инструментов и выберите пункт Properties в контекстном меню ToolBar.
2. Перетащите их на панель инструментов, используя технику drag-and-drop.
3. Перетащите лишние кнопки за пределы панели инструментов.
4. Удерживайте нажатой клавишу Shift во время щелчка на компоненте в палитре компонентов. После этого каждый щелчок на форме будет приводить к помещению в нее нового компонента.
5. Дважды щелкните на кнопке компонента в палитре компонентов.
6. .mak, .cpp, .h, .dfm и .res.
7. Show().
8. ShowModal().
9. В инспекторе объектов перейдите на вкладку Events. Щелкните на кнопке со стрелкой в столбце Value рядом с событием. Вы увидите список обработчиков, совместимых с данным событием. Выберите один из них.

10. Щелкните на компоненте в редакторе форм и нажмите клавишу Tab на клавиатуре. При каждом нажатии Tab будет выделяться и отображаться в инспекторе объектов следующий компонент формы.

День 7

1. При выделении компонентов, являющихся потомками других компонентов (например, компонентов на панели).
2. Он служит опорным компонентом, оставаясь на своем месте. Все остальные компоненты выравниваются по нему.
3. Заключите компоненты в ограничивающую рамку (или коснитесь их рамкой).
4. Выделите все компоненты, которые вы хотите изменить. Затем выберите в главном меню пункт Edit | Size и нажмите радиокнопку Grow to Largest.
5. В редакторе кода отображается обработчик события по умолчанию для данного компонента. Для многих компонентов это будет обработчик события OnClick. В отдельных случаях (например, для компонента Image) вы увидите диалоговое окно.
6. Установка этой опции приводит к заполнению компонентом всей рабочей области своего предка (как правило, формы), независимо от того, какие размеры принимает предок.
7. По традиции оно означает, что выбор данного пункта меню приводит к отображению диалогового окна.
8. Вы можете перетащить пункт на новое место в редакторе меню или использовать буфер обмена.
9. При вводе подписи к пункту меню добавьте амперсанд (&) перед той буквой, которая будет использоваться для быстрого вызова данного пункта. Например, Caption для пункта меню File | Exit будет выглядеть как E&xit.
10. Установите для его свойства Enabled значение false.

День 8

1. Да, но это стоит делать только в крайнем случае.
2. Свойство Enabled.
3. Ее текст становится серым.
4. Текст длинной подсказки выводится в строке состояния, а текст короткой — во всплывающем окне подсказки.
5. Invalidate(), Repaint(), Refresh() и Update().
6. Три: простой, выпадающий и выпадающий список.
7. Когда нажимается кнопка, свойство ModalResult которой имеет целое значение, форма закрывается. Значение этого свойства будет использовано в качестве возвращаемого значения метода ShowModal().
8. Компонент Panel. Могут использоваться и некоторые другие компоненты.

9. true.
10. Достаточно изменить значение его свойства Title на Save As.

День 9

1. Если вам нужно, чтобы производный объект имел все свойства базового объекта и изменялся каждый раз, когда будет изменен базовый объект.
2. Выберите в главном меню пункт Project | Add to Repository или Add Project to Repository в контекстном меню менеджера проектов.
3. Все производные формы изменятся в соответствии с изменениями в базовой форме.
4. В разделе public или private. Ни в коем случае не размещайте их в разделе __published, если не знаете точно, что вы делаете.
5. В любом исходном модуле, но обычно в том же файле, где находится остальная часть кода данной формы.
6. Автор объекта будет показан, если переключиться на вид Details.
7. В диалоговом окне конфигурации репозитория объектов (для этого нужно выбрать пункт Options | Repository в главном меню).
8. Практически во всех случаях проще использовать мастер приложений.
9. Добавьте к проекту файл .res или .rc через менеджер проектов, или используйте директиву препроцессора #pragma resource.
10. Да, это делается очень просто.

День 10

1. F12.
2. Нет. Он удаляется только из проекта.
3. На вкладке Forms диалогового окна Project Options.
4. Вы должны сами создавать формы перед их использованием.
5. Трудно сказать, но по-видимому 32 Мб для Windows 95 и 40 Мб для Windows NT.
6. Если отладочная информация сгенерирована, вы можете выполнять код в пошаговом режиме во время сеанса отладки.
7. В том каталоге, где находится файл проекта.
8. Ctrl+S.
9. Закладка устанавливается нажатием клавиш от Ctrl+K+0 до Ctrl+K+9. Всего доступны 10 закладок.
10. Выберите пункт Read Only в контекстном меню редактора кода.

День 11

1. Щелкните в поле редактора слева от этой строки. Вы можете также нажать клавишу F5 или выбрать пункт Toggle Breakpoint в контекстном меню редактора кода.

2. Контрольная точка, установленная в строке, для которой компилятор не генерирует код.
3. Установите обычную контрольную точку, выберите в главном меню пункт View | Breakpoints, щелкните на этой точке в окне Breakpoint List и выберите в контекстном меню списка контрольных точек пункт Properties. Введите требуемое условие в поле Condition диалогового окна Edit Breakpoint.
4. Дважды щелкните на этом объекте в окне Watch List. На экране появится диалоговое окно Watch Properties. Измените свойства требуемым образом.
5. Щелкните на переменной и нажмите Ctrl+F5 (или выберите пункт Add Watch at Cursor в контекстном меню редактора кода).
6. Инспектор отладки.
7. С помощью клавиши F7.
8. Щелкните на этой переменной и затем выберите пункт Evaluate/Modify в контекстном меню редактора кода. Измените значение в диалоговом окне Evaluate/Modify.
9. TRACE выводит сообщение при любых условиях. WARN выводит сообщение только в том случае, если условное выражение принимает значение true.
10. Файл OutDbg1.txt в редакторе кода.

День 12

1. Через прозрачный цвет значка или курсора виден фон.
2. Щелкните на соответствующем цвете в палитре.
3. Выберите инструмент Прямоугольная область (Marquee) и растяните прямоугольник с помощью мыши. Вы можете также выбрать в меню пункт Edit | Select All для выделения всего изображения, или использовать инструмент Лассо.
4. 256.
5. Пиксел в изображении курсора, координаты которого сообщаются Windows при щелчке мышью.
6. Да.
7. Выберите в главном меню пункт Window | Follow to Focus и затем щелкните на том окне, за которым вы хотите наблюдать.
8. Для поиска текста в исходных файлах.
9. Включите опцию Editor Files во вкладке Preferences (раздел Autosave Options) диалогового окна Environment Options.
10. Щелкните на кнопке Reset Defaults во вкладке Palette диалогового окна Environment Options.

День 13

1. В окне инспектора объектов щелкните на вкладке Events.
2. Да.
3. Для этого идеально подходит компонент Notebook.

4. Чтобы установить для строки состояния только одну панель.
5. Для простой строки состояния:
```
StatusBar->SimpleText = "Text";
```
6. Установите для свойства Enabled компонента значение true, чтобы разрешить доступ к компоненту, и false, чтобы запретить доступ.
7. Через функцию Printer().
8. BeginPrint().
9. NewPage().
10. Измените свойство Cursor данного компонента.

День 14

1. Через диалоговое окно Project Options (вкладка Application) или установкой свойства HelpFile объекта Application во время выполнения программы.
2. Просто установите ненулевое значение для свойства HelpContext. Проверьте, что в справочном файле существует идентификатор контекста с таким же значением и этот справочный файл установлен для приложения.
3. HelpCommand().
4. Любые: встроенные типы, классы или структуры.
5. Да. Вы можете иметь их столько, сколько необходимо.
6. С помощью ключевого слова throw.
7. \HKEY_CURRENT_USER.
8. Нет. Деструктор TRegistry закроет ключ за вас. Вы не сможете держать ключ открытым бесконечно долго.
9. PostMessage() отправляет сообщение в очередь сообщений Windows и сразу же возвращает управление. SendMessage() отправляет сообщение и не возвращает управление до тех пор, пока это сообщение не будет доставлено получателю.
10. Perform().

День 15

1. Генерируется сообщение об ошибке, которое указывает на отсутствие данного метода в автоматизированном объекте.
2. Выберите в главном меню пункт Component | Install, затем щелкните на ActiveX.

День 16

1. Для использования в многопоточных приложениях, работающих с базами данных.
2. Дополнительные возможности по управлению (например, управление уровнем изоляции транзакции, управление процессом транзакции) и большую гибкость при изменениях внутри приложения.

3. Для локальных таблиц SQL должен:

- ☐ Использовать только локальный синтаксис SQL.
- ☐ Работать только с одной таблицей.
- ☐ Не иметь оператора ORDER BY.
- ☐ Не содержать составных функций.
- ☐ Не содержать вычисляемых полей.
- ☐ Использовать в операторе WHERE сравнения имен столбцов только со скалярными константами. К поддерживаемым операторам относятся LIKE, >, <, >= и <=. Отдельные элементы оператора могут быть объединены также с помощью AND или OR.

Для таблиц, расположенных на сервере, SQL должен:

- ☐ Использовать только локальный синтаксис SQL.
- ☐ Работать только с одной таблицей.
- ☐ Не содержать составных функций.

4. TBatchMove.

День 17

1. Выбором опции Form Wizard в меню Database, или пункта меню File | New и опции Database Form во вкладке Forms соответствующего диалогового окна.
2. Свойства DatabaseName и TableName.
3. Свойства DataSet и AutoEdit.
4. TDataSource.

День 18

1. DatabaseName.
2. Пункт Include Unit Hdr.
3. Во вкладке Forms диалогового окна New Items (вы можете получить доступ к этому окну через пункт меню File | New).
4. Одним из преимуществ использования модулей данных является возможность их сохранения в репозитории объектов C++Builder, откуда вы (и не только вы) сможете брать их для использования в других приложениях. Другим преимуществом является единообразие в доступе к базам данных для всех ваших приложений. Последнее преимущество состоит в том, что вы можете управлять доступом к базам данных в C++Builder из одной ключевой точки. Вместо того, чтобы распределять компоненты доступа к данным по всему приложению, вы собираете эти компоненты в одном месте, где их можно легко конфигурировать.

День 19

1. Добавьте к проекту библиотеку импорта (.lib-файл). DLL будет загружена при запуске приложения.

2. Используйте функцию LoadLibrary() из API Windows.
3. Достаточно вызвать функцию как обычную функцию.
4. Функция должна быть экспортирована из DLL и объявлена с модификатором extern "C".
5. Вы можете выгрузить DLL в любое время с помощью функции FreeLibrary().
6. Нужно создать экспортируемую функцию, к которой сможет обращаться вызывающее приложение. Эта функция должна содержать код, который создает и отображает форму.
7. Вы можете использовать один заголовок при построении как DLL, так и вызывающего приложения.
8. Добавьте файл .res или .rc к проекту DLL через менеджер проектов.
9. Нет. Если DLL загружается статически, она может вообще не содержать код.
10. Да. Создайте для DLL файл библиотеки импорта, как и для DLL, содержащей код, а затем добавьте этот файл к проекту вызывающего приложения.

День 20

1. Нет. Вы можете использовать прямой доступ.
2. Нет, но обычно это имеет место. Свойства, которые не хранят значений, встречаются крайне редко.
3. Разумеется. Это простейший способ создания нового компонента.
4. В этом случае свойство будет доступно только для чтения.
5. Базовый член свойства изменяется и/или считывается непосредственно (отсутствуют методы записи и чтения).
6. Нет. Значения по умолчанию устанавливать не обязательно, но это следует делать для всех отображаемых свойств.
7. Нет. Значение по умолчанию только отображается в инспекторе объектов во время разработки. Вы должны присваивать значения соответствующим членам-данным в конструкторе компонента.
8. Выберите в главном меню пункт Component | Install.
9. Создайте файл .dcr с тем же именем, что и у исходного модуля компонента. Этот файл должен содержать растровое изображение размером 24 на 24 пиксела, имеющее то же имя, что и класс компонента. Файл .dcr нужно поместить в один каталог с файлом .cpp компонента.
10. Просто вызовите событие после проверки существования его обработчика:

```
if (OnMyEvent) OnMyEvent(this);
```

День 21

1. __declspec, delphiclass, delphireturn, __automated, __published, __closure, __property, __classid(class) и __dispid(int).
2. Да, через вспомогательный макрос.

3. Классы AnsiString, Variant, ShortString, Currency, TDateTime и Set.
4. К версии 2.0.
5. В заголовочном файле, соответствующем главному исходному модулю формы.

День 22

1. Компонент NNTP.
2. OnBeginRetrieval.
3. Свойство URL.
4. Компонент FTP.

День 23

1. __property.
2. Да, событие, как и свойство, является клаузой.
3. Ключевое слово __published используется для отображения свойства в инспекторе объектов во время разработки приложения.
4. Нет. Оно существовало в предыдущих компиляторах Borland. Однако в C++Builder его использование было расширено.