О создании .NET-приложений — простым и доступным языком!

Более 100 миллионов "Чайников" продано во всем мире

Visual C++® .NET для "Чайников"™

Исходный код описанных в книге примеров находится на Web-сервере

Для сомневающихся

Майкл Хаймен
Боб Арнсон

Авторы книги
Visual C++ 6 For Dummies

Диалектика

Visual C++ .NET

для

"ЧАЙНИКОВ"™

Visual C++® .NET FOR DUMMIES®

by Michael Hyman
and Bob Arnson

Hungry Minds™
HUNGRY MINDS, INC.
Best-Selling Books • Digital Downloads • e-Books • Answer Networks • e-Newsletters • Budget Web Sites • e-Learning
New York, NY • Cleveland, OH • Indianapolis, IN

Visual C++ .NET для "Чайников"™

Майкл Хаймен
Боб Арнсон

ДИАЛЕКТИКА

Москва • Санкт-Петербург ◆ Киев
2002

ББК 32.973.26-018.2.75
X15
УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией *В.В. Александров*

Перевод с английского и редакция *И.А. Минько*

По общим вопросам обращайтесь в издательство "Диалектика"
по адресу: info@dialektika.com, http://www.dialektika.com

Хаймен, Майкл, **Арнсон,** Боб.
X15 Visual C++.NET для "чайников". : Пер. с англ. — М. : Издательский дом "Вильямс", 2002. — 288 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0326-2 (рус.)

Итак, вы решили серьезно взяться за Visual C++ .NET. Это хорошая идея, ведь вы в действительности убиваете сразу трех зайцев: в ваших руках оказывается мощный, полезный и широко распространенный инструмент. С языком C++ можно сделать очень многое. С его помощью созданы такие продукты, как Excel и Access. Этот язык также применяется при разработке управленческих информационных систем и систем целевого назначения, используемых для анализа деятельности предприятий и принятия решений в сфере управления бизнесом. И, конечно же, целые армии хакеров и не только хакеров используют C++ для создания инструментов, утилит, игр и шедевров мультимедиа. Знания, которые вы получите, изучив язык C++ .NET, позволят создавать не просто приложения, а приложения, работающие в разных операционных системах. Возможности этого языка практически не ограничены, и вы сами в этом убедитесь, прочитав эту книгу.

Книга предназначена для начинающих программистов.

ББК 32.973.26-018.2.75

ISBN 5-8459-0326-2 (рус.)
ISBN 0-7645-0868-7 (англ.)

Оглавление

Введение 14
Глава 1. Что представляет собой пакет Visual C++ .NET 19
Глава 2. Что такое программа 25
Глава 3. Приступаем к созданию программ 35
Глава 4. Принятие решений — дело серьезное 49
Глава 5. Хороший редактор — что еще нужно бывалому программисту? 55
Глава 6. Компиляция программ, или Первые трудности 65
Глава 7. Типы данных — это серьезно 71
Глава 8. Использование переменных 81
Глава 9. Структуры данных 85
Глава 10. Выразите свои желания 89
Глава 11. Ключевые слова — ключ к диалогу с компьютером 105
Глава 12. Внимание! Повышенная функциональность 117
Глава 13. Указатели 129
Глава 14. Масса информации? Используйте массивы! 155
Глава 15. Пришел, увидел, применил 165
Глава 16. Через тернии к... работающей программе 171
Глава 17. Смотрите на мир объективно 189
Глава 18. Конструкторы и деструкторы 209
Глава 19. Наследование 219
Глава 20. Исключительные ситуации 237
Глава 21. Потоки данных 249
Глава 22. Создаем пользовательский интерфейс 255
Глава 23. Десять синтаксических ошибок 265
Глава 24. Вторая десятка синтаксических ошибок 271
Глава 25. Десять функций .NET 277
Предметный указатель 280

Содержание

Введение 14

Часть I. Первое знакомство с Visual C++ .NET 17

Глава 1. Что представляет собой пакет Visual C++ .NET 19

Инструменты Visual C++ 19

Компилятор, запускающий тысячи программ 20

Отладчик программ 20

Интегрированная среда разработки программ 21

Библиотеки - хранилища электронных инструментов 21

Эти разумные утилиты 22

Помощь, которая всегда рядом 22

Не знаете с чего начать - просмотрите примеры программ 22

Управляемые и неуправляемые программы 23

Глава 2. Что такое программа 25

Введение в программирование 25

Главная функция программы 26

Стандартные подпрограммы 27

Для чего создаются программы 28

А теперь немного теории 31

Что такое объект и с чем его едят 31

Инкапсуляция 32

Наследование 32

Полиморфизм 33

Глава 3. Приступаем к созданию программ 35

Зачем нужны исходные файлы 35

Как все это выглядит на практике 36

С чего начинается выполнение программы 38

Как организовать диалог с пользователем 39

Не скупитесь на комментарии 40

Исходный код программы HelloWorld 41

Пару штрихов к программе Hello World 42

Забудем о .NET 42

Отображение информации на экране 43

Печать с новой строки 44

Обратная связь: получение ответа 45

Использование библиотечных функций 45

Итак, займемся делом 46

Глава 4. Принятие решений - дело серьезное 49

Правильное решение может сделать вас счастливым 49

Файлы проектов сделают вашу жизнь проще 50

Решения и проекты 50
Построение программ 50
Определение параметров нового проекта 50
Добавление файлов к проекту 52
Что может окно Solution Explorer 53
Глава 5. Хороший редактор — что еще нужно бывалому программисту? 55
Коды существуют для того, чтобы их редактировать 55
Приемы редактирования 57
Коды бывают разные - черные, белые, красные 59
Помощь, которая всегда рядом 60
Навигация по просторам программы 60
Сокрытие и отображение кодов 61
Поиск и автозамена 62
Глава 6. Компиляция программ, или Первые трудности 65
Начать компилировать программу очень просто 65
Синтаксические ошибки: неужели их может быть так много?! 66
Предупреждения 67
Почему компилятор не исправляет ошибки самостоятельно 68
Компилировать можно по-разному 68
Часть II. Все, что вы хотели знать о C++, но о чем боялись спросить 69
Глава 7. Типы данных - это серьезно 71
Строгие и нестрогие языки программирования 71
Объявление переменных 72
Наиболее часто используемые типы данных 72
Реже используемые типы данных 73
Обеспечение типовой безопасности 74
Функции преобразования типов 75
Константы - то, что никогда не меняется 76
Использование констант в кодах программы 77
Константы и борьба с ошибками 78
Строки как один из наиболее важных типов данных 78
Глава 8. Использование переменных 81
Именование переменных 81
Определение переменных 83
Инициализация переменных 83
Как сделать имя информативным 84
Глава 9. Структуры данных 85
Объявление структур 85
Использование этих загадочных структур 86
Использование одних структур для создания других 86
Структуры на практике 87

Глава 10. Выразите свои желания 89
Можно ли "выражаться"? 89
Простые операторы 89
Более сложные операторы 90
Оператор ++ 91
Оператор >> 91
Оператор « 92
Истина и ложь в логических выражениях 92
Оператор присвоения 94
Все об операторах 96
Работа с битами 97
Условный оператор 98
Приоритет операторов 99
Примеры работы операторов 101
Математические функции 101
Старый формат математических функций 103
Глава 11. Ключевые слова — ключ к диалогу с компьютером 105
Великолепная тройка: ключевые слова if, for и while 106
Условный оператор 106
Оператор for 109
Пример использования цикла for 109
Повторение ради повторения 110
Вычисление факториала 110
Оператор while 112
Ключевые слова switch и do 112
Оператор switch 113
Оператор do 114
Глава 12. Внимание! Повышенная функциональность 117
Некоторые вводные замечания 117
Создание функций 118
Использование аргументов 119
Функции, которые возвращают результат 120
И снова вернемся к факториалам 122
Рекурсия: спасибо мне, что есть я у меня 124
Если тип аргументов не определен . . . 127
Значения, установленные по умолчанию 128
Глава 13. Указатели 129
Почему указатели 129
Указатели и переменные 130
Что вы! Указатели - это очень сложно 131
Информация и ее адрес 131
Безымянные данные 131

Связанный список - размер не ограничен 132
Использование указателей в C++ 133
Дайте указателю адрес 134
Как получить значение, на которое ссылается указатель 134
Пример программы, использующей указатели 134
Изменение значения, на которое ссылается указатель 136
Изменение значений в структурах данных 136
Использование стрелки 136
Динамическое выделение памяти 136
Знимание - графика 137
Все цвета радуги 139
Перья для рисования 141
Кисточка для раскраски 142
Шрифты 142
Займемся рисованием 142
Связанные списки и графика 144
Как эта программа работает 144
Код программы 146
Вопросы безопасности 149
Освобождение памяти 149
Общее нарушение защиты 150
Генеральная уборка 151
Кое-что о строках 152
Подведем итог 153
Глава 14. Масса информации? Используйте массивы! 155
Массивы: познакомимся поближе 155
Это же "элементарно", Ватсон 156
Инициализация массива 157
Многомерные массивы 158
Класс ArrayList 159
Класс Stack 161
Перечислимые типы 162
Безопасность при использовании перечислимых типов 162
Одно маленькое "но" 163
Глава 15. Пришел, увидел, применил 165
Немного теории 165
Почему это так важно 166
Правила определения области видимости 169
Глава 16. Через тернии к... работающей программе 171
Синтаксические и логические ошибки 171
Процесс отладки программы 172
Отладчик плюс редактор - и все в наших руках 173

Остановись, мгновение! 173
Шаг за шагом: движение к цели 174
Посмотрим, что получилось 175
Не нравится значение - измените его 176
Торопитесь? Нет проблем! 177
Очень торопитесь? 177
Генеральная уборка 178
Итак, приступим 178
Так где же ошибка? 180
Что теперь? 183
Но в программе по-прежнему есть ошибка 184
Устранение ошибки 184
Последние штрихи 185
Часть III. Экскурс в объектно-ориентированное программирование 187
Глава 17. Смотрите на мир объективно 189
Что такое классы и с чем их едят 189
Разберемся в деталях 190
Члены данных 190
Функции-члены 190
Объявление классов 191
Ограничение доступа 191
Защищенный доступ 192
Определение функций-членов 192
Что делать с готовыми классами? 193
Доступ к элементам класса 193
Имена, используемые функциями-членами 194
Немного практики 194
Управление памятью в неуправляемых программах 198
Функции доступа 203
Общие рекомендации 207
Глава 18. Конструкторы и деструкторы 209
Работа "до" и "после" 209
Подготовительный этап 209
Много конструкторов - много возможностей 210
Открытые и закрытые конструкторы 212
Заметание следов 212
Когда объект больше не нужен 212
Не забывайте также освобождать динамически выделяемую память 214
Классы внутри классов 215
Чтение кодов объектно-ориентированных программ 216
Глава 19. Наследование 219
Что происходит при наследовании кодов 219

Наследование открытых, закрытых и защищенных элементов 221
Перегрузка функций-членов 221
Родительские связи 222
А теперь немного практики 222
Как процесс наследования отображается на конструкторах и деструкторах 224
Универсальный указатель 224
Защищенное и закрытое наследование 225
Виртуальная реальность 226
Тест на виртуальность 227
Декларация ваших виртуальных намерений 227
Когда без этого не обойтись 228
Абстрагирование от деталей 231
Искусство абстрагирования 232
Глава 20. Исключительные ситуации 237
Как это было раньше 237
Новый усовершенствованный способ обработки ошибок 238
Вот как это выглядит на практике 239
Гибкость – основное свойство механизма обработки исключений 241
Определение собственных исключительных ситуаций 241
Поговорим о синтаксисе 242
Все это хорошо, но несколько запутанно 245
Наследование классов, описывающих исключения 246
Пять правил исключительного благополучия 247
Глава 21. Потоки данных 249
.NET-классы I/O 249
Записали - прочитали 250
Повторное использование потоковых переменных 251
Записали - прочитали 252
Специальные команды, погружаемые в потоки данных 253
Формат передаваемых данных 253
Кое-что о работе с файлами 254
Глава 22. Создаем пользовательский интерфейс 255
Основной набор инструментов 255
Формы 257
Наследование форм 257
Настройка формы 258
Открытие формы 258
Обработка событий 259
Объекты, которые умеют реагировать 259
Хорошие менеджеры умеют делегировать свои обязанности 260
Объявление и определение обрабатывающей событие функции 261
Добавление делегата 261

Часть IV. Горячие десятки 263

Глава 23. Десять синтаксических ошибок 265

Подключаемый файл не может быть найден 265
Пропущена точка с запятой 266
Не подключен заголовочный файл 266
Не обновлено объявление класса 267
Использование названия класса вместо названия объекта 267
После объявления класса не поставлена точка с запятой 268
В определении класса пропущено слово public: 268
Неправильно набранные имена переменных 268
Использование точки вместо стрелки и наоборот 269
Пропущена закрывающая фигурная скобка 269

Глава 24. Вторая десятка синтаксических ошибок 271

Конструкторы, которые не требуют аргументов 271
Незакрытые комментарии 272
Несоответствие типов данных 272
То, что работало в C, может не работать в C++ 273
Использование ключевого слова void 273
Конструкторы для производных классов 274
Точка с запятой в конце строки #define 274
Отсутствие свободного места на диске 274
Так в чем же проблема 275

Глава 25. Десять функций .NET 277

Console: :WriteLine 277
Console::ReadLine 277
Int32::Parse 277
Application::Run 278
Graphics->DrawLine 278
Color: ;FromArgb 278
Graphics->DrawString 278
Image::FromFile 279
Form::OnMouseMove 279
Controls->Add 279

Предметный указатель 280

О авторах

Майкл Хаймен (Michael Hyman) — руководитель технического отдела компании DataChannel. Ранее занимался вопросами медиатехнологий, XML и Internet. Майкл написал множество книг по компьютерной тематике, среди которых *Dynamic HTML For Dummies, Visual J++ For Dummies* и *PC Roadkill*. Он имеет ученую степень по электронной инженерии и компьютерным наукам, полученную в Принстонском университете. В свободное время Майкл занимается серфингом, ходит в спортзал и стирает пеленки.

Боб Арнсон (Bob Arnson)— разработчик программного обеспечения в крупной компьютерной компании. Боб написал несколько других книг по Visual Basic, Visual C++ и Borland C++. Также его статьи публиковались в журналах *VB Tech Journal* и VC++ *Professional.*

Посвящения

Майкл Хаймен посвящает эту книгу Мириам Бэс (Miriam Beth) и Габриеле Миа (Gabrielle Mia).
Боб Арнсон посвящает эту книгу Марио (Mario).

Благодарности

Особая благодарность моей жене Саре, которая неделями терпела мои ночные писательские бдения и, как следствие, мой уставший вид и сонные глаза. Также спасибо моим дочерям, которые иногда отвлекали меня от компьютера и возвращали на землю. Спасибо читателям, просто за то, что они интересуются такими вопросами, как программирование на Visual C++. А также спасибо всем людям, которые приняли участие в создании этой книги.

Майкл Хаймен

Спасибо моим друзьям из Нью-Гемпшира и Мичигана за моральную помощь и поддержку, которая так нужна была мне в мои нелегкие писательские будни. Спасибо моим коллегам по издательству, которые внесли свою посильную лепту в создание этой книги и благодаря которым она имеет такое прекрасное оформление. Также хочу выразить благодарность компании Microsoft за новые технологические решения, появившиеся вместе со средой .NET.

Боб Арнсон;

Введение

Итак, вы решили серьезно взяться за C++ .NET. Это хорошая идея, поскольку вы убиваете сразу трех зайцев: в ваших руках оказывается мощный, полезный и широко распространенный инструмент. С языком C++ можно сделать очень многое. С его помощью созданы такие продукты, как Excel и Access. Этот язык также применяется при разработке управленческих информационных систем и систем целевого назначения, используемых для анализа деятельности предприятий и принятия решений в сфере управления бизнесом. И, конечно же. целые армии хакеров и не только хакеров используют C++ для создания инструментов, утилит, игр и шедевров мультимедиа.

Знания, которые вы получите, изучив язык C++ .NET, позволят вам создавать не просто приложения, а приложения, работающие в разных операционных системах. Возможности этого языка практически не ограничены, и вы скоро сами в этом убедитесь.

Об этой книге

Скажем сразу, книга должна вам понравиться. Здесь не нужно читать главы одну за другой от начала и до конца, Если вы уже знакомы с Visual Studio, пропустите главы, посвященные ей. Просмотрите содержание книги и. если найдете что-то для себя незнакомое, просто начните читать с этого места. Если же во время чтения вы вдруг уснете — не волнуйтесь, это тоже бывает. Здоровый сон еше никому не повредил.

Тем, кто еще ничего не знает о C++, следует начинать чтение с самого начала, чтобы узнать не только о C++, но и о Visual Studio .NET. Хотим сразу предупредить, что язык C++не так прост, как это может показаться вначале. Придется немного напрячь умственные способности, чтобы осилить его. Вы столкнетесь с множеством правил, приемов, концепций и терминов. Кроме того, в отличие от обычных языков программирования, C++ реализует концепцию объектно-ориентированного программирования, что вообще является нишей не для слабых умов.

Вдобавок к сложностям C++ вас ожидает знакомство с массой инструментов Visual Studio ЛЕТ. В целом приложение Visual C++, имеющие 120 Мбайт надстроек, библиотек и тому подобных вещей, может внушать непосвященным благоговейный ужас. Вначале даже трудно решить, с какого конца к нему подойти.

Но не все так страшно, когда у вас в руках эта книга. Она дает общее представление как о C++, так и о Visual C++. На вас не обрушится лавина технических деталей и терминов, вместо этого вы познакомитесь с наиболее важной информацией и общими концепциями. Причем это будет сделано в стиле, понятном любому нормальному человеку.

Эта книга не раскрывает всех возможностей языка C++. поскольку для этого понадобилось бы добавить в нее еше несколько сот страниц или набрать весь текст очень мелким шрифтом. Но зато вы получите достаточно знаний для того, чтобы начать писать на языке C++ свои собственные программы.

Что вы можете не читать

Можете не читать высказывания жены Михаэля. поскольку она просто злилась на то, что ее муж уделяет больше внимания книге. чем ей. Можете не читать газеты, которые пишут о политике. Не читайте электронную почту, если она приходит вместе с фотоснимком известной русской теннисистки. Если не хотите. не читайте текст. отмеченный в книге как технические подробности, поскольку эти вещи, по сути, являются второстепенными и могут вызвать интерес разве что у заядлых любителей покопаться в деталях.

Исходные предпосылки

Чтобы прочитать и понять эту книгу, не обязательно иметь опыт программирования. Однако, если раньше вы уже пытались что-то программировать, например создавали макросы для электронных таблиц или баз данных, вы будете чувствовать себя во время чтения книги намного комфортнее тех, кто ранее с программированием вообще не сталкивался.

Если вы уже освоили BASIC, COBOL, Pascal или даже C, научиться программировать на C++ для вас не составит большого труда. (Но если вы уже профессионал по C++, то, вероятно, купили не ту книгу-)

Вне зависимости от вашего программистского прошлого, мы все же предполагаем, что вы знаете, как запускаются приложения Windows и что представляют собой файлы и программы, Также мы предполагаем, что на вашем компьютере установлены .NET и Visual Studio .NET (если вы, конечно, хотите сами попробовать набрать приведенные в книге коды и проверить, как они работают).

Как организована эта книга

Книга состоит из четырех частей. В первой части дается краткий обзор Visual C++. Вы познакомитесь с основными возможностями этого языка программирования и научитесь ими пользоваться.

Во второй части дается обзор основ программирования на C++ (обратите внимание, что многие рассмотренные положения будут справедливы также и для языка C). Здесь же вы узнаете, что такое программы .NET и как они создаются.

В третьей части вы окунетесь в мир объектно-ориентированного программирования и расширите свои познания о среде .NET.

В четвертой, заключительной, части, вы найдете практические советы, позволяющие решить многие проблемы, наиболее часто возникающие у начинающих программистов.

Пиктограммы, используемые в книге

Пиктограммы — это маленькие картинки, акцентирующие ваше внимание на некоторых важных вещах. Вот, что они обозначают.

Этой пиктограммой обозначаются технические подробности, которые, впрочем, читать не обязательно.

Внимательно относитесь к информации, отмеченной такой пиктограммой! Обязательно постарайтесь ее запомнить. Это будет несложно, поскольку информация дается в максимально сжатом и лаконичном виде. Если же вы ее проигнорируете, потом об этом можете пожалеть.

Этой пиктограммой отмечены различные полезные советы, которые могут облегчить вашу программистскую жизнь и избавить от ненужных проблем.

Такой пиктограммой отмечено все, что касается неуправляемых программ, т.е. программ, написанных на старом добром C++, которым разработчики среды .NET дали такое пренебрежительное определение.

Куда двигаться дальше?

На Гавайи, Тайвань, Фиджи. Но только не туда, куда посылала Михаэля эго жена, обиженная тем, что на какое-то время он посвятил себя написанию этой книги.

Часть I

Первое знакомство с Visual C++ .NET

Прошу прощения! Есть ли здесь кто-то, кто говорит НЕ о Visual C++ .NET?

В этой части...

Здесь дается краткий обзор Visual C++, начиная с установочного пакета и заканчивая описанием его отдельных инструментов.

Вы узнаете, как создаются программы, и сможете получить общее представление об основных принципах объектно-ориентированного программирования. Также вы увидите, как можно использовать мастера Visual C++ для создания программ .NET. Здесь же вы познакомитесь с такими важными компонентами окружения Visual C++, как компилятор, редактор кодов и средство Solution Explorer.

Глава 1

Что представляет собой пакет Visual C++ .NET

В этой главе...

- Что такое Visual C++
- Обзор возможностей Visual C++
- Разница между управляемыми и неуправляемыми программами

Итак, вы решили стать программистом на языке C++. Это действительно хорошая идея, Освоив этот язык, вы дополните свое резюме, сможете создавать потрясающие приложения и находить общий язык с множеством очаровательных людей. Но главное — вы сможете сказать, что владеете самым великолепным и мощным языком программирования из всех когда-либо созданных.

Инструменты VisualC++

Не все качественные приложения занимают много места.

Увидев впервые установочный комплект Visual C++, вы, наверное, решите, что чего-то в нем не хватает,— настолько он небольшой и компактный. Казалось бы, приложение, занимающее множество мегабайт памяти, должно иметь более внушительный вид. Однако такое впечатление создается потому, что наиболее массивная часть почти каждого установочного пакета — руководство пользователя — поставляется не в распечатанном, а в электронном виде. (Таким образом не только экономится место, но и спасается несколько деревьев.)

Открыв установочный пакет Visual C++, вы найдете в нем компакт-диск с программным обеспечением. На нем содержится множество инструментов, участвующих в создании программ на языке C++:

- ✓ компиляторы;
- ✓ отладчики;
- ✓ интегрированная среда разработки;
- ✓ системы создания приложений;
- ✓ библиотеки;
- ✓ утилиты Windows;
- ✓ общие утилиты;
- ✓ библиотека оперативной документации;
- ✓ примеры программ.

Далее в этой главе дается краткое описание этих инструментов с тем, чтобы вы имели о них общее представление и знали, для чего устанавливаете.

Компилятор, запускающий тысячи программ

Компилятор преобразует коды из вида, понятного программистам *(исходные коды)*, к виду, воспринимаемому компьютером *(выполняемые коды)*. В приложении Visual C++ есть два компилятора:

- ✓ строковый компилятор;
- ✓ интегрированная среда разработки.

Когда вы будете изучать описание примеров, приведенных в книге, используйте интегрированный компилятор, поскольку его пользовательский интерфейс проще и удобнее.

Что представляет собой строковый компилятор

Строковый компилятор - это компилятор, не имеющий графического пользовательского интерфейса. Он быстр, но не очень дружелюбен по отношению к пользователю. Чтобы обратиться к нему, нужно набрать специального вида компактную инструкцию, подобную этой:

```
cl /FR /WX foo.cpp
```

В данном примере первый элемент является именем компилятора, который вы хотите использовать, второй и третий представляют собой команды, сообщающие компилятору, что нужно делать, и наконец последний – это имя файла, который должен быть откомпилирован. (Использовать строковый компилятор вместо интегрированной среды разработки могут только те программисты, которые знают, например, что команда /FR применяется для отображения окна просмотра информации, а команда /wx обозначает, что все предупреждения нужно интерпретировать как ошибки. Те, для кого приведенная выше строка является просто набором букв, использовать строковый компилятор не могут.)

В старые добрые времена известны были только строковые компиляторы. Те, кто занимается программированием довольно длительное время, используют в основном строковые компиляторы, во-первых, по привычке, а во-вторых, поскольку они предоставляют возможность дополнительного использования множества замечательных инструментов. Те же, кто только начинает программировать на Visual C++, предпочитают интегрированную среду разработки программ, поскольку она намного понятнее и проще в использовании.

Отладчик программ

Если вы написали программу, состоящую более чем из двух или трех строк, у вас наверняка возникнут проблемы при ее компиляции. Если же этого не произойдет, то вы либо великий программист, либо скопировали коды программы из этой книги.

Все возникающие при запуске программы проблемы можно разделить на две категории: синтаксические ошибки и логические ошибки. *Синтаксические ошибки* случаются тогда, когда вы что-то неправильно набираете, забываете указать информацию, которая нужна компилятору, или некорректно используете команду. Компилятор сам находит синтаксические ошибки и указывает строки, где такие ошибки встречаются. Вам придется исправить все найденные синтаксические ошибки, иначе компилятор не сможет преобразовать набранные вами коды в выполняемые.

Логические ошибки случаются тогда, когда неправильно разработана или реализована сама программа. Например, вы забыли набрать коды для обработки какой-то информации или даете команду отобразить на экране не те данные, которые необходимы. Программы, содержащие логические ошибки, процесс компиляции проходят успешно, однако выполняются не так, как вы хотите.

Рассмотрим это на примере. Допустим, вы создали программу, отслеживающую количество денег на вашем счете. Но при написании ее кодов забыли набрать команду, прибавляю-

щую суммы, вносимые на счет. Или наоборот: забыли набрать команду, вычитающую из общей суммы деньги, снимаемые со счета. Тогда сколько бы денег вы со счета не снимали, состояние счета не изменялось бы (этакий вечный источник наличных). Это пример логической ошибки. Вы забыли набрать один оператор, и в результате получилась хотя и симпатичная, но работающая неправильно программа.

Исправить логические ошибки намного сложнее, чем синтаксические. Сложность заключается в их поиске и локализации места возникновения. Помочь справиться с этой проблемой призван отладчик программы. *Отладчик* позволяет строка за строкой проследить за выполнением программы и просмотреть значения обрабатываемых при этом данных. Таким образом, обнаружив, что какая-то переменная приняла неверное значение, вы сразу же определите место возникновения ошибки.

Интегрированная среда разработки программ

Пакет Visual C++ включает в себя *интегрированную среду разработки* (devent.exe), объединяющую множество разнообразных инструментов, значительно упрощающих процесс создания программ. Если вы используете среду разработки Visual C++. нет необходимости изучать и запоминать принципы работы каждого инструмента в отдельности.

Среда разработки Visual C++ состоит из таких основных компонентов:

- ✓ редакторы, позволяющие набирать и модифицировать исходные коды программы;
- ✓ компилятор, выполняющий компиляцию кодов программы (на этом этапе отсеиваются и исправляются все синтаксические ошибки);
- ✓ отладчик, помогающий исправить логические ошибки и заставить программу работать так, как вы хотите;
- ✓ диспетчер проектов, позволяющий легко создавать выполняемые подпрограммы (файлы с расширением DLL и LIB);
- ✓ обозреватель, позволяющий отследить связи между различными объектами объектно-ориентированных программ;
- ✓ Visual-инструменты (мастера), с помощью которых можно легко создавать Windows-приложения;
- ✓ списки свойств, которые помогают контролировать отображение и поведение объектов пользовательского интерфейса создаваемой программы.

Чтобы воспользоваться возможностями любого из этих компонентов, достаточно просто выбрать команду из раскрывающегося меню и задать настройки в диалоговом окне. Это значительно упрощает процесс реализации сложных проектов, поскольку нет необходимости изучать и применять множество не совсем понятных командных строк. Читая эту книгу, вы будете иметь дело только со средой разработки Visual C++ и освоите при этом большое количество различных приемов программирования.

Библиотеки — хранилища электронных инструментов

Библиотеки — это наборы заранее созданных функций и классов, которые могут быть использованы для решения многих типичных задач. Приложение Visual C++ имеет несколько библиотек. Они могут заметно облегчить процесс создания программ, поскольку позволяют использовать готовые варианты вместо разработки собственных решений. Среда .NET, по существу, является одной большой библиотекой. Библиотеки выполнения (обозначаемые аббревиатурой RLT — runtime libraries) содержат множество полезных функций, выполняющих математические вычисления, обработку текстовых значений и многие другие задачи. Файлами библиотек RLT являются libc.lib, libcmt.lib и msvcrt.lib.

О чем говорят названия файлов библиотек

Со временем вы обратите внимание на то, что одни и те же библиотеки реализованы в нескольких вариантах. Каждый вариант соответствует разным наборам опций, поддерживаемых библиотекой и обозначаемых буквой (или буквами), которая добавляется в начале или в конце названия файла библиотеки. Например, название mfc70d.lib обозначает, что это файл библиотеки MFC, выполненной в варианте для отладчика (d – debuggable version).

Вот список букв, используемых для обозначения версий библиотек;

- ✓ D Debuggable (Отладка);
- ✓ MT Multithread (Многопоточные процессы);
- ✓ O OLE (Технология связывания и внедрения объектов);
- ✓ S Static (Статические объекты);
- ✓ U Unicode (Уникоды).

Эти разумные утилиты

Visual C++ содержит набор утилит, позволяющих проследить за выполнением .NET- и Windows-программ. Обычно только опытные программисты пользуются услугами утилит, наиболее важной среди которых является Spy++. Во время выполнения программы она отображает на экране информацию о классах, применяемых этой программой. Вы можете использовать ее для отслеживания не только своих, но и чужых программ. Другие утилиты созданы для упрощения процесса программирования. В основном они рассчитаны на опытных программистов и предназначены для использования совместно с инструментами командной строки. Наиболее часто используемой среди них является утилита NMAKE. Но вы можете также найти для себя парочку полезных утилит, не требующих умения работать с командной строкой. Например, утилита WINDIFF может сравнивать между собой содержимое двух файлов и находить в них отличия. Утилита ZOOM1N позволяет изменять размеры отображаемых объектов таким образом, чтобы они выглядели на экране именно так, как вы хотите.

Помощь, которая всегда рядом

Все, что есть в пакете Visual C++ (компиляторы, отладчики, среда разработки, библиотеки, утилиты), должно сопровождаться документацией. Однако, чтобы не делать установочный пакет слишком громоздким, компания Microsoft поставляет документацию в электронном виде. Причем эта документация организована в виде оперативной справочной системы, позволяющей быстро находить нужную информацию и читать ее, не выходя из среды разработки программ. Справочная система может быть открыта также в отдельном окне. В самом крайнем случае вы можете лаже подключить дополнительный монитор к своей системе и постоянно иметь перед глазами справку об инструментах и возможностях, используемых в данный момент.

Не знаете с чего начать — просмотрите примеры программ

Пакет Visual C++ включает в себя множество примеров готовых программ, воспользовавшись которыми вы сможете быстрее и проще создавать собственные программы. Некоторые из них демонстрируют определенные приемы программирования, другие являются готовыми вариантами решения каких-то задач, например обработки текстовой информации. Самое лучшее в этих программах то, что их коды можно копировать и вставлять в коды своих программ. Это сохранит ваше время и силы и позволит сосредоточить внимание на решении более специфических проблем.

Управляемые и неуправляемые программы

Visual C++ позволяет создавать программы как для среды Windows, так и для среды Microsoft .NET. В этой книге основное внимание уделяется вопросам разработки приложений для среды .NET, но будет также рассмотрено и создание приложений для Windows. Среда .NET, разработанная компанией Microsoft, предназначена для создания программ, работающих в Internet. Она включает в себя язык CLR (Common Language Runtime) и библиотеку классов.

CLR является основой среды .NET — все работает под его управлением. CLR следит за выполнением кодов и обеспечивает предоставление программам всех сервисов нижнего уровня, таких, например, как управление памятью. Библиотека классов является *приложением среды* .NET и содержит в себе набор готовых классов, которые вы можете использовать при создании объектно-ориентированных программ. (О том, что такое классы и для чего они нужны, вы узнаете чуть позже.)

Чтобы работать в среде .NET, программа должна быть изначально созданной для этой цели. Это подразумевает использование при ее создании классов .NET, но главное — она должна быть откомпилирована для работы в этой среде. Visual C++ может удовлетворить оба требования. Коды программы, которая может работать в среде .NET, называются *управляемыми*.

Коды программ, написанных для работы в среде Windows и не предназначенных для использования в среде .NET, принято называть *неуправляемыми*.

Если вы не используете при разработке программы систему создания приложений, снабдить программу хорошим пользовательским интерфейсом будет отнюдь не просто. Пользователям очень нравятся программы с качественным интерфейсом, однако программистам приходится потратить немало времени и усилий, чтобы создать его. Например, чтобы создать небольшую программу для Windows, содержащую несколько пунктов меню и отображающую на экране сообщение "Hello World!", потребуется набрать от двух до четырех тысяч строк с кодами. (Программисты называют между собой такие программы Hello World. По тому, насколько легко или сложно создать программу Hello World, определяют, насколько проста и успешна та или иная система программирования.) Это действительно так, если не использовать при создании программы библиотеку классов среды .NET. Только программисты, помешанные на командах нижнего уровня, занимаются подобными глупостями, все же остальные используют среду разработки .NET. И сложность даже не в том, что нужно набирать эти четыре тысячи строк, а в том. что нужно знать и помнить тысячи команд, используемых для управления средой Windows. Это, пожалуй, не та информация, которой стоит засорять голову.

Большинство программ на порядок сложнее, чем Hello World, и при их создании приходится решить множество задач, прежде чем они заработают так, как нужно. Например, к их числу относятся такие задачи:

- ✓ определение алгоритма получения сообщений Windows;
- ✓ определение комбинаций клавиш, нажатием которых будут вызываться эти сообщения;
- ✓ определение, в какой части программы будут храниться полученные сообщения;
- ✓ определение, какие другие части программы выполняются в данный момент;
- ✓ регистрация имен разных частей программы.

Обратите внимание, что ни одно из этих действий не имеет никакого отношения к выводу чего-либо на экран.

Системы создания приложений позволяют автоматически решать эти и многие другие задачи. Например, приступив к созданию программы, вы можете воспользоваться классом System.Windows.Forms.Application, содержащим в себе все коды, необходимые для начала написания программы. В частности, этот класс содержит коды, необходимые для создания окон. Эти и подобные им классы автоматически вставляют в коды вашей программы типичные фрагменты, что дает возможность сосредоточиться на решении более важных задач.

Visual C++ содержит также другие системы создания приложений, например библиотеку MFC (Microsoft Foundation Classes), включающую в себя классы C++, используемые при создании Windows-программ. Библиотека ATL (Active Template Library) содержит шаблоны C++, используемые при создании объектов для модели COM (Component Object Model). Однако системы ATL и COM настолько сложны, что их использование вызывает трудности даже у хакеров со стажем, а потому в данной книге рассматриваться не будут.

Глава 2

Что такое программа

В этой главе...

- Основы программирования
- Выражения, переменные, комментарии и библиотеки
- Чтение программ
- Основы объектно-ориентированного программирования

В этой главе дается краткий обзор основных этапов создания программ: проектирования, написания, компиляции и отладки. Кроме того, вам придется "проглотить" немного теории, касающейся объектно-ориентированного программирования. Вас также ожидает встреча с настоящей живой работающей .NET-программой.

Введение в программирование

Проектированием программ называется этап, на котором принимается решение о том, что же именно программа должна делать. Для этого нужно четко определить проблему или круг проблем, которые должны быть решены, и выработать стратегию их решения.

Написание программ — это этап, во время которого вы садитесь и набираете коды своей программы. Обычно вы набираете инструкции языков программирования высокого уровня. Например, одной строкой можно сказать компьютеру, что нужно вывести что-то на экран. В этом и следующих разделах вы познакомитесь с множеством команд языка C++, с помощью которых можно заставить компьютер делать что-нибудь полезное.

Далее программу нужно *откомпилировать*. На этом этапе Visual C++ .NET преобразует коды C++ высокого уровня, понятные вам, в коды низкого уровня, понятные компьютеру. В процессе проектирования и написания программы обычно разбиваются на несколько отдельных файлов (поскольку это удобно, а иногда и полезно). В процессе компиляции программы эти файлы объединяются в приложение. После того как программа откомпилирована, компьютер знает, как ее нужно выполнять.

Теперь можно запустить программу на *выполнение*. Если она не примитивна и состоит более чем из нескольких строк, вполне вероятно наличие в ней ошибок. Поскольку ошибки не всегда бывают явными, необходимо тщательно протестировать программу, чтобы убедиться в том. что она работает именно так, как было задумано.

Процесс поиска и исправления ошибок называется *отладкой* программы. Для поиска ошибок обычно используются услуги отладчика. Когда ошибка найдена, вы снова возвращаетесь к этапу редактирования кодов программы для ее исправления.

Все программы состоят из команд, объясняющих компьютеру, что нужно делать и как обрабатывать имеющиеся данные. Почти все действия программ сводятся к получению, обработке и отображению (или сохранению) данных. Даже, например, программы видеоигр заняты в основном только этим. Они получают данные, следя за нажатием клавиш, движениями

мыши или джойстика, обрабатывают данные, принимая решения, что нужно делать дальше, и затем отображают результат (например, на экране отображается следующая комната лабиринта и с потолка прыгают два монстра с пушками в лапах).

Языки высокого и низкого уровня

Чтобы понять разницу между языком программирования высокого уровня и языком программирования низкого уровня, сравните между собой выражение, набранное в C++ (язык высокого уровня), и эквивалентный ему код, набранный на языке ассемблера (низкий уровень).

Код C++:

```
a = 3*a - b*2 + 1;
```

Эквивалентный ему код ассемблера:

```
mov eax, DWORD PTR _a$[ebp]
lea eax, DWORD PTR[eax+eax*2]
mov ecx, DWORD PTR _b$[ebp]
add ecx, ecx
sub eax, ecx
inc eax
mov DWORD PTR _a$[ebp], eax
```

В действительности коды ассемблера также относятся к более высокому уровню, чем коды, которые компьютер может воспринимать непосредственно (без компиляции). Компьютер может понимать только машинный язык, представляющий любые инструкции и команды в числовом виде.

Вот как выглядит фрагмент кодов машинного языка:

```
8b 45 fc
8d 04 40
3b 4d f8
03 c9
2d c1
40
89 45 fc
```

Теперь вы можете сравнить коды языков высокого и низкого уровня и решить, язык какого уровня вы хотели бы использовать для написания программ.

Главная функция программы

Выполнение всех программ, написанных на языке C++, начинается с функции, именуемой main. При запуске программы прежде всего выполняется первое выражение функции main. *Выражение* — это строка кодов, представляющая собой отдельную инструкцию для компьютера. *(Функция* состоит из группы выражений, собранных вместе для решения определенной задачи. Более подробно функции будут рассмотрены в следующей главе.) Затем поочередно выполняются все остальные выражения — по одному за раз.

Инструкции некоторых выражений выполняются только в тех случаях, если справедливо какое-то условие. Такие выражения называются *условными операторами*. Так, строка кодов может содержать, например, следующую информацию для компьютера: "Если пользователь щелкнет на кнопке Печать, выведи открытый документ на печать".

Переменные используются для представления данных в программе. Например, если нужно запомнить имя пользователя, можно создать переменную Имя. Затем в любой момент, когда потребуется имя пользователя, можно просто сослаться на значение переменной Имя. В процессе выполнения программы значения переменных могут изменяться. Например, можно присвоить переменной Имя значение Мартин, а потом другим выражением присвоить этой же переменной значение Степан. Но само по себе значение переменной никогда не меняется — в любом случае вы должны написать какое-нибудь выражение, меняющее одно значение на другое.

Комментарии используются для описания того, что происходит в процессе выполнения программы. Вы можете добавлять их для расшифровки целей, с которыми пишутся те или иные фрагменты кодов, для фиксирования каких-то мыслей и идей, для описания решаемых задач. Добавляя комментарии, вы упрощаете чтение кодов вашей программы пользователями. Для вас комментарии также могут быть очень полезны. Если через некоторое время вы захотите внести изменения в уже набранные коды, вам, скорее всего, трудно будет вспомнить, для чего используется та или иная переменная и зачем нужно было создавать ту или иную функцию. В таких случаях, пожалуй, комментарии могут быть единственным средством, которое поможет вам вспомнить, что же именно вы хотели сделать, набирая эти коды. (Кроме того, в комментариях программы вы можете высказать все, что наболело. Компьютеру ведь все равно, а вы избавитесь от лишнего груза.) При преобразовании кодов C++ в машинные коды компилятор игнорирует строки, являющиеся комментариями, и просто пропускает их.

На рис. 2.1 показана небольшая программа, в которой используются функция main, выражения, переменные и комментарии.

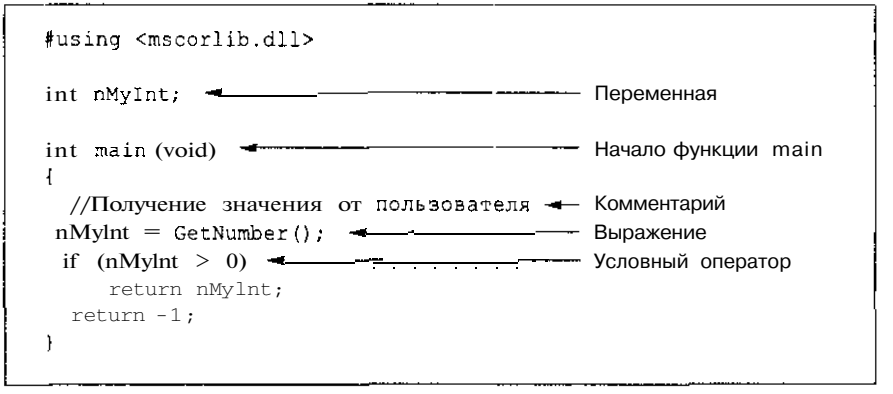


Рис. 2.1. Основные компоненты, из которых состоит программа

Стандартные подпрограммы

Коды общих для большинства программ алгоритмов хранятся в компоненте среды .NET, называемом *CLR* (Common Language Runtime). Например, почти все программы отображают какие-то значения на экране. Возможно, это покажется вам странным, но для выполнения такого простого действия компьютеру нужно сделать множество шагов. Вместо того чтобы писать целые наборы кодов для всех этих шагов каждый раз, когда нужно что-то вывести на экран, вы можете просто воспользоваться готовой подпрограммой, созданной для среды .NET.

Когда вы пишете *неуправляемые* программы (те, которые создаются с использованием старой версии C++), подпрограммы, реализующие типичные алгоритмы, содержатся в так называемых *библиотеках*. Чем они отличаются от CLR? По сути, ничем. И те и другие являются библиотеками. Разница только в названии. CLR может быть использован при создании программ на языках Visual Basic, C# и Visual C++ (при этом сами библиотеки для разных языков будут несколько отличаться). Хотя CLR, по существу, является тем же набором библиотек, его возможности значительно шире возможностей стандартных библиотек языка C++.

Есть два типа библиотек: статические и динамические. Если вы используете при создании своей программы процедуры *статической библиотеки*, коды этих процедур непосредственно копируются в текст вашей программы, увеличивая таким образом ее размер. Если же вы используете *динамические библиотеки* (называемые также DLL), сами процедуры не копируются в вашу программу, а вызываются в процессе ее выполнения.

Для чего создаются программы

Любая программа создается для решения каких-то задач. При этом, перед тем как начать писать колы, нужно разделить задачу на несколько отдельных логических частей и затем создать процедуры, каждая из которых будет решать проблемы своей части. Поначалу перспектива разделения общей задачи на несколько более мелких может показаться довольно сложной, однако по мере приобретения опыта программирования вы научитесь делать по быстро и почти автоматически. (Любые хорошие компьютерные курсы уделяют лому вопросу большое внимание.)

В качестве примера рассмотрим вариант создания программы для решения обычной задачи. Предположим, вам нужно вычислить площадь квадрата. (Зачем? Какая разница. Может, у вас вечеринка и вы хотите чем-то удивить своих друзей.) Вше со школы вы помните, что площадь квадрата вычисляется как умножение его стороны на саму себя.

В контексте создания программы эта задача может быть разбита на три логические подзадачи.

1. **Определение длины стороны квадрата.**
2. **Вычисление площади квадрата путем умножения стороны на саму себя.**
3. **Отображение полученного результата.**

Теперь, когда общая проблема разбита на отдельные подзадачи, каждую из них следует преобразовать в коды программы, чтобы компьютер знал, какие задачи ему нужно решать. Например, на языке C++ .NET эти коды могут выглядеть так:

```
//SquareArea
//Вычисление площади квадрата при условии, что известна
//длина его стороны

#include "stdafx.h"

#using <mscorlib.dll>
```

```
using namespace System;


//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
ttelse
int main(void)
#endif
{

  String *pszSize;
  int nSize;
  int nArea;


  //Запрос от пользователя значения длины стороны квадрата
  Console::WriteLine(L"Чему равна длина стороны квадрата?");


  //Получение ответа
  pszSize = Console::ReadLine();


  //Преобразование полученного значения в число
  nSize = nSize.Parse(pszSize);


  //Вычисление площади квадрата
  nArea = nSize*nSize;


  //Отображение результата
  //Обратите внимание, что для этого вначале нужно
  //преобразовать полученное значение в строку
  Console::WriteLine(L"Площадь квадрата составляет {0}
                              единиц.", nArea.ToString());


  //Ожидание, пока пользователь не остановит
  //выполнение программы
  Console: :WriteLine(L"Нажмите Enter, чтобь: завершить
                        выполнение программы");
  Console::ReadLine();
     return 0;
}
```

Итак, вы только что узнали, как выглядят коды программы. Но как разобраться, для чего нужна каждая отдельная строка? Для этого сначала познакомьтесь с общими принципами чтения кодов программы (если вы их еще не знаете).

1. **Начинайте читать с самого начала.**
2. **Читайте по одной строке.**
3. **Старайтесь понять, для чего нужна каждая строка.**
4. **Если вы чего-то не понимаете, переходите к следующей строке.**

Эти же принципы используются при работе компилятора, за исключением разве что четвертого пункта.

Далее описывается, как вы могли бы интерпретировать коды приведенной выше программы. Итак, обратимся к первым двум строкам:

```
//SquareArea
//Вычисление площади квадрата при условии, что известна
//длина его стороны
```

Увидев их, вы можете подумать: "Ну, в этих строках нет ничего сложного. В них говорится, что эта программа вычисляет площадь квадрата исходя из длины его стороны". (Две косые черты в начале строк указывают, что эти строки являются комментариями.)

```
#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;
```

"А это уже что-то совсем непонятное. Наверное, о том, для чего все это нужно, я узнаю в следующих главах". (Совершенно правильный ход мыслей.)

```
#ifdef _UNICODE
int wmain(void)
ttelse
int main(void)
#endif
```

"Опять ЧТО-ТО непонятное. Пожалуй, сейчас это можно пропустить".

```
  String *pszSize;
  int nSize;
  int nArea;
```

"Зачем нужны эти строки я точно не знаю, однако слова size (длина) и *area* (площадь) уже как-то созвучны с решаемой задачей".

```
//Запрос от пользователя значения длины стороны квадрата
  Console::WriteLine(L"Чему равна длина стороны квадрата?");


  //Получение ответа
  pszSize = Console::ReadLine();


  //Преобразование полученного значения в число
  nSize = nSize.Parse(pszSize);
```

"Все это выглядит немного странно, но в целом можно понять, что коды эти нужны для того, чтобы компьютер узнал, чему равна длина стороны квадрата".

```
  // Вычисление площади квадрата
  nArea = nSize*nSize;
```

"Да, да! Это я понимаю! В этой строке длина стороны умножается на саму себя, и мы находим площадь квадрата".

Ну и так далее.

Закончив читать эту книгу, вы будете знать, для чего нужна каждая из встретившихся вам здесь команд. Более того, вы будете знать не только как читать коды программ, но и как их создавать. Другими словами, вы научитесь программировать.

А теперь немного теории

Вы только что узнали, как выглядят коды настоящей программы на C++. Однако в этой программе не были использованы возможности объектно-ориентированного программирования. Язык C++ настолько популярен во многом благодаря именно этим возможностям. Почему? На это есть ряд причин.

- ✓ Объектно-ориентированное программирование позволяет повторно использовать одни и те же коды, что значительно экономит время и усилия, необходимые при создании программы.
- ✓ Объектно-ориентированные программы имеют хорошую структуру, что существенно облегчает чтение и понимание их кодов.
- ✓ Объектно-ориентированные программы легко тестировать. Их можно разбить на небольшие компоненты и отдельно проверять работу каждого из них.
- ✓ В объектно-ориентированные программы очень легко вносить изменения и добавлять новые возможности.

Основная идея объектно-ориентированного программирования проста. Данные и процедуры, обрабатывающие эти данные, объединяются в одно целое, называемое *классом*. Если нужно получить доступ к данным какого-то класса, используются процедуры этого же класса.

В старых процедурных языках программирования, напротив, данные и процедуры, их обрабатывающие, не могли быть объединены и воспринимались как совершенно разные компоненты.

Что такое объект и с чем его едят

В основу всего объектно-ориентированного программирования положена такая идея: разбить решаемую задачу на группу объектов. *Объекты* состоят из данных и процедур, которые эти данные обрабатывают. Например, рассмотрим программу, управляющую дисководом с автоматической сменой дисков. Она может состоять из нескольких объектов, обрабатывающих информацию о воспроизводимых с устройства музыкальных треках. Один объект, например, может использоваться для представления воспроизводимых треков, а второй для представления самого дисковода. Каждый объект должен содержать данные, которые его описывают (объект Трек может содержать информацию о продолжительности песни), и функции, которые оперируют этими данными (объект Трек может иметь функцию, отображающую название песни и ее продолжительность). Объединение данных и обрабатывающих их функций в отдельные объекты называется *инкапсуляцией*.

Уже готовые объекты можно объединять между собой для создания новых объектов. Это называется *композицией*. Например, можно создать объект Караоке, объединив объект ДИСКОБОЛ С объектом Микрофон. (Можно создать объект Сутки, объединив объект День с объектом Ночь.)

Можно создавать новые объекты на основе уже существующих объектов. Например, на основе объекта, воспроизводящего видео, можно создать объект, распространяющий видео, добавив в него функции обработки денежных переводов. Изменение существующих свойств объектов или добавление к ним новых функций для создания новых объектов называется *наследованием*.

Наследование — одно из наиболее важных свойств объектно-ориентированного программирования. Создавая новые объекты путем наследования кодов уже существующих работающих объектов, вы получаете ряд преимуществ.

- ✓ Не нужно повторно набирать те же коды: все коды, которые набраны для исходных объектов, автоматически могут быть использованы новыми объектами.
- ✓ Снижается вероятность возникновения ошибок: если вы точно знаете, что исходный объект работал правильно, значит, любые возникшие ошибки следует искать в кодах, которые были добавлены к новому объекту при его создании. С другой стороны, если вы найдете и исправите ошибку в исходном объекте, она автоматически будет исправлена для всех других объектов, созданных из данного путем наследования.
- ✓ Коды программы становятся легче для чтения и понимания: нужно понять, как работают исходные объекты. Понять, как работают объекты, созданные путем наследования, будет намного проще, поскольку вам останется только изучить, как работают добавленные к ним данные и функции.

Еще одним свойством объектно-ориентированного программирования является тот факт, что одна и та же функция может выполняться по-разному в зависимости от того, данные какого объекта обрабатываются. Это свойство называется *полиморфизмом*. Например, если функция должна распечатать содержимое ячейки электронной таблицы, на печать выводится число. Если же функция должна распечатать диаграмму, на печать выводится изображение. В обоих случаях используется одна и та же функция, но поскольку объекты разные (ячейка и диаграмма), выполняемые действия будут совершенно различными.

Инкапсуляция

Инкапсуляцией называется выделение данных и функций, обрабатывающих эти данные, в отдельные элементы, называемые *объектами* или *классами*. Данные, в зависимости от того как они используются, иногда называют *свойствами* классов. Функции также иногда называют *методами* классов.

Секрет создания качественной объектно-ориентированной программы заключается в том, чтобы выделить классы, которые бы максимально точно описывали реальную решаемую проблему и которые можно было бы максимально часто повторно использовать. Поначалу эта задача может показаться довольно сложной, однако по мере приобретения практического опыта вы научитесь это делать почти автоматически.

Наследование

Это одно из наиболее интересных свойств объектно-ориентированного программирования. Как отмечалось ранее, с помощью этого свойства можно создавать новые объекты, используя для этого уже существующие. Новый класс, созданные на основе готового класса, называется *производным*. А тот класс, который был взят за основу, называется *базовым*. (Иногда производные классы называются *дочерними*, а базовые — *родительскими*.)

На рис. 2.2 показано семь классов. Класс Звук является самым простым. Он содержит название воспроизводимого звука и его продолжительность. Класс Трек основан на классе Звук, а потому также содержит информацию о названии и продолжительности. Кроме того, он содержит данные об исполнителе и о дате записи. Класс Rock является производным от класса Трек. Он содержит все элементы класса Трек (как и все элементы класса Звук) плюс еще некоторые. Таким образом, на основе любых классов можно создавать более сложные классы, содержащие в себе больше разных данных и функций для их обработки. При этом каждый производный класс содержат в себе все данные и функции, которые есть в базовом классе.

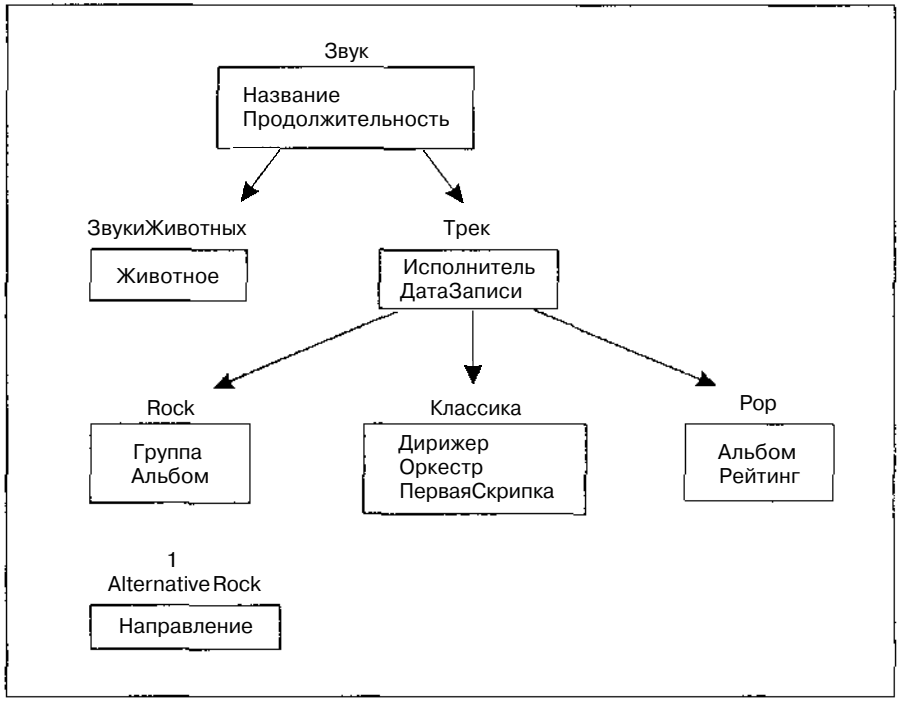


Рис. 2.2. Новые классы можно создавать путем наследования свойств и методов уже созданных классов

Полиморфизм

При создании новых классов можно использовать не только возможность наследования, но также полиморфизм для определения способа поведения новых классов. Если вы еще не запомнили все греческие слова, употребляемые в этой главе, напомним, что под *полиморфизмом* подразумевается возможность одних и тех же функций по-разному обрабатывать данные, принадлежащие разным объектам. Например, класс Звук может иметь функцию Получить. Если запустить ее для объекта ЗвукиЖивотных, она может отобразить на экране сообщение, советующее взять с собой диктофон и сходить в ближайший зоопарк. Если запустить эту же функцию для объекта Rock, может быть предпринята попытка открыть соответствующую Web-страницу с возможностью загрузки нужного файла.

Используя одновременно возможность наследования и полиморфизм, вы сможете легко создавать наборы подобных, но уникальных объектов. Благодаря наследованию объекты будут во многом похожи друг на друга. А благодаря полиморфизму каждый из них может чем-то отличаться от других. Так, если вы используете полиморфизм, то функция, присутствующая в разных объектах (такая, как функция Получить), для каждого из них будет работать по-разному.

Если для какой-то функции полиморфизм не используется, для производного класса она будет выполняться точно так же, как и для базового. Если для класса, производного от класса [illegible], не *перегрузить* функцию "случить (т.е. просто наследовать ее без внесения изменений), то ее выполнение также будет приводить к попытке открыть ту же Web-страницу. Таким образом, производные классы изначально имеют ту же функциональность, что и базовые. Если же какие-то функции отличаются, значит, к ним был применен полиморфизм.

Глава 3

Приступаем к созданию программ

В этой главе...

- Что такое исходные файлы
- Создание и выполнение управляемых программ .NET C++
- Создание и выполнение неуправляемых программ C++

Эта глава поможет вам создать свою первую программу C++. (И даже не одну.) Начнем с создания управляемой программы C++. Это дружественное определение дано разработчиками среды .NET программам, создаваемым для работы в этой среде, и обозначает, что при разработке этих программ предоставляется множество дополнительных возможностей (благодаря CLR) и обеспечивается их корректное выполнение (благодаря правильному управлению памятью и многому другому). Также вы научитесь создавать добрые старые (или стандартные) программы C++, для которых разработчики среды .NET придумали нехорошее название — неуправляемые программы. Но вначале вы научитесь создавать то, что является основой любой программы C++ — исходные файлы.

Зачем нужны исходные файлы

Исходные файлы, или файлы реализации с расширением CPP (читается как C Plus Plus) либо CXX, содержат в себе основную часть выполняемой программы. Именно здесь определяются функции, данные и ход выполнения программы. (Под *ходом выполнения программы* подразумевается последовательность, в которой будут выполняться отдельные выражения или части программы.)

Исходные файлы состоят из выражений. *Выражением* является строка программы, которая дает указание компьютеру выполнить какое-то действие.

Например, приведенная ниже строка является выражением, которое указывает компьютеру, что необходимо вычислить площадь квадрата путем умножения длины стороны на саму себя:

```
nArea = nSize * nSize;
```

Каждое выражение должно заканчиваться точкой с запятой. Точка с запятой показывает компьютеру, где заканчивается одно выражение и начинается другое (так же, как на письме одно предложение заканчивается точкой, после чего начинается другое предложение).

Можно объединить несколько выражений, заключив их в фигурные скобки ({ }). Например, допустим, что есть такое выражение:

```
if чай горячий, подожди немного;
```

(Команда `if`, которая будет более подробно рассмотрена в главе 11, указывает компьютеру на необходимость выполнить какое-то действие в случае, если справедливо некоторое условие.) Если вы хотите, чтобы в это же время компьютер сделал еще какое-то действие, можно набрать такое выражение:

```
if чай горячий {подожди немного; просмотри почту;}
```

Два выражения были объединены с помощью фигурных скобок.

Открывающую и закрывающую фигурные скобки можно также использовать для обозначения начала и окончания функции. Более подробно об этом речь идет в главе \2.

Если для выполнения функции требуется указание параметров, они заключаются в круглые скобки. Например:

```
if номер занят {немного подожди(2 минуты); позвони еще раз;}
```

Таким образом функция подожди точно указывает, сколько нужно подождать, перед тем как еще раз позвонить. Узнать об этом более подробно (не о телефонных звонках, а об использовании параметров), вы сможете, прочитав главу 12.

Итак, подытожим полученные знания об использовании выражений:

- ✓ строки заканчиваются точкой с запятой (;);
- ✓ объединение строк осуществляется с помощью фигурных скобок ({ });
- ✓ значения параметров заключаются в круглые скобки — ().

Как все это выглядит на практике

Пожалуй, теории пока достаточно. Самое время заняться делом. Начнем с создания нового проекта .NET.

1. **Выберите команду File⇨New⇨Project (Файл⇨Создать⇨Проект).**
2. **Откроется диалоговое окно New Project.**
3. **В разделе Project Types (Тип проекта) выберите пункт Visual C++ Project.**
4. **В разделе Templates (Шаблоны) выберите Managed C++ Application.**
5. **В поле Name (Имя) наберите Hello World.**
6. **В папке Visual Studio Projects (она расположена в папке My Documents) будет создана папка Hello World.**

   Обратите внимание на рис. 3.1. Если вы хотите создать папку Hello World в какой-нибудь другой папке, укажите ее название в поле Location (Размещение) или щелкните на кнопке **Browse** (Обзор), чтобы открыть окно для поиска этой папки.

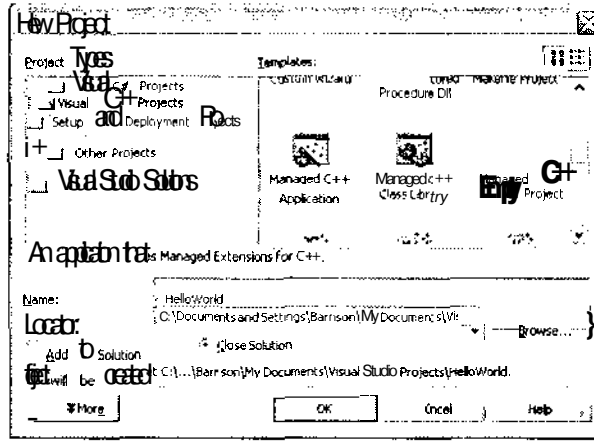


Рис. 3.1. Использование диалогового окна New Project для создания проекта Hello World

7. Щелкните на кнопке OK.

Visual C++ создаст целый набор файлов, принадлежащих одному проекту. Один или несколько создаваемых проектов составляют одну *задачу (solution)*. Более подробно о задачах речь идет в главе 4. Окно Solutin Explorer (Разработчик задач), показанное на рис. 3.2, отображает файлы, из которых состоит проект Hello World.

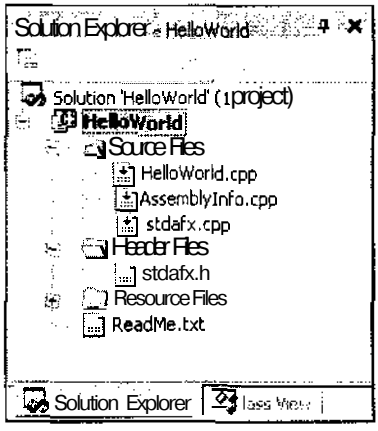


Рис. 3.2. S окне *Solution Explorer можно увидеть файлы, используемые данным проектом*

Все эти файлы были созданы для того, чтобы можно было реализовать простую программу, отображающую на экране сообщение Hello World. Чтобы увидеть исходные коды программы, дважды щелкните в окне Solution Explorer на значке HelloWorid.cpp. Этот файл расположен в папке Source Files (Исходные файлы) проекта Hello World.

К счастью, эта простая программа, отображающая на экране сообщение Hello World, может быть создана автоматически. Для этого выполните следующее.

1. Выберите команду **Debug⇨Start Without Debugging (Отладка⇨Запустить** без отладки).

   Visual C++ отобразит сообщение о том, что проект HelloWorid - Debug Win32 является устаревшей программой.

2. **Щелкните на кнопке Yes, чтобы Visual** C++ создал этот проект автоматически.

   Откроется окно Output (Выходное окно), в котором вы сможете наблюдать за процессом построения проекта.

   Как только проект будет создан, на экране мелькнет черное окно и сразу же исчезнет. Разумеется, вы не успеете заметить, что было в этом окне. Так случается всегда, когда программа просто выполняет набор каких-то действий и сразу завершает свою работу.

Но как же определить, правильно программа работает или нет? И делает ли она вообще что-то? Сделать это можно разными способами. Из материала главы 16 вы узнаете, как можно использовать отладчик для остановки в самом конце выполнения программы. А далее в этой главе описано, какие коды необходимо добавить, чтобы программа в нужном месте остановилась и ожидала, пока пользователь не даст ей сигнал продолжать работу. По мере приобретения опыта программирования вы оцените полезность и универсальность этого приема. Ну а пока выполните следующее.

1. **Откройте окно** для ввода команд.

Если у вас установлена Windows 2000 или Windows XP, выберите команду Start⇨Programs⇨Accessories⇨Command Prompt. Если у вас установлена Windows NT, выберите команду Start⇨Programs⇨Command Prompt.

2. **Используйте** команлу CD, чтобы перейти в папку с проектом **HelloWorld.**

   Если имя папки содержит пробелы, возьмите его в кавычки. Для Windows 2000 и Windows XP путь к папке может выглядеть приблизительно так: "C:\Documents and Settings\MyUserName\My Documents\Visual Studio Projects\HelloWorld". (В действительности путь будет зависеть от настроек вашего компьютера. Например, вместо слова MyUserName нужно будет указать ваше пользовательское имя.)

3. Чтобы **запустить** программу, наберите `Debug\HelloWorld.`

   Поскольку Visual Studio создала проект автоматически, т.е. со всеми установками, заданными по умолчанию, она поместила файл, с которого начинается выполнение программы. в папку Debug. Результат выполнения программы вы можете увидеть на рис. 3.3.

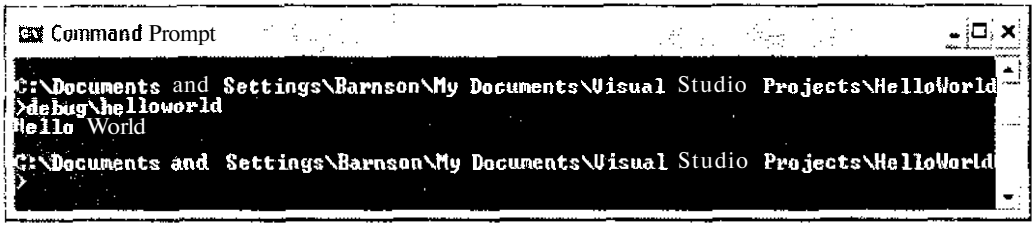


Рис. 3.3. Вокнедляввода команд отображается надпись Hello World

С чего начинается выполнение программы

Теперь, когда вы видели результат выполнения программы, давайте посмотрим, каким образом все это происходит. Для начала компьютер должен определить, с какой строки нужно начинать выполнение программы. Вместо того чтобы нумеровать все строки (как это делалось в старые добрые времена), нужно просто поместить первую строку программы в функцию, именуемую `main`. С первой строки этой функции начинается выполнение любой программы. Например, вот код простой программы, которая ничего не делает:

```
int main(void)
{
   // Здесь абсолютно ничего не происходит
}
```

Выполнение этой программы, как и всех остальных, начинается с функции `main`. Поскольку функция `main` (как и любая другая функция) может использовать в своей работе значения параметров, после ее названия следует пара круглых скобок. (В среде .NET функции обычно используются без каких-либо параметров, поэтому вместо них постоянно набирают слово `void`, обозначающее факт их отсутствия.) Далее следует открывающая фигурная скобка, которая указывает компьютеру, что за ней идут строки функции `main`. Закрывающая фигурная скобка говорит о том, что на этом выполнение функции заканчивается. Вам может показаться непонятным значение слова `int`, стоящего перед названием функции. Более подробно это рассматривается в главе 12, а пока напомним, что *функция* — это отдельная подпрограмма, которая может выполнять какие-то действия (например, выводить что-то на печать), обрабатывать данные или запрашивать какую-либо информацию. Функции могут также возвращать в качестве результата какое-то значение. В C++ `main` также является функцией, а следовательно, может возвращать какое-нибудь число (integer).

Начало функций main всех управляемых программ C++, коды которых вы встретите в этой книге, выглядит несколько сложнее:

```
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#er:dif
```

Слово _UNICODE обозначает, что будет использоваться стандарт, предусматривающий ввод с клавиатуры букв и символов других языков, включая азиатские. При компиляции программы может быть выбрана опция, указывающая на то, что программа должна быть откомпилирована с учетом этой возможности.

Как организовать диалог с пользователем

Конечно, можно создавать программы, которые ничего не будут делать. Но все же лучше, если программа будет в состоянии отобразить что-либо на экране и принять какую-нибудь информацию от пользователя. Есть миллион способов (ну хорошо, пять или чуть больше) сделать это в C++ .NET. Самый простой — использовать для этого функции Console. Они позволяют считывать и отображать сообщения в окне консоли (это то черное окно, которое вы видели при выполнении вашей первой программы). Чтобы отобразить сообщение на экране, используйте команду Console : : WriteLine. Дайте ей любой текст, и она высветит его в окне консоли. Например, показанная ниже строка выводит на экран сообщение Hello World (обратите внимание, что текст взят в кавычки):

```
Console: :WriteLine( "Hello World") ;
```

Допустим, вместо этого сообщения нужно отобразить текст Здравствуйте, я ваша тетя ! Делается это аналогичным образом:

```
Console::WriteLine("Здравствуйте, я ваша тетя!");
```

Чтобы получить информацию от пользователя, используйте команду Console:: ReadLine. Полученная информация будет воспринята как строка текста. (В главе 7 рассматривается, как преобразовать эту информацию к какому-нибудь полезному виду.) Например, приведенная ниже строка заставит компьютер ждать, пока пользователь не введет какой-нибудь текст:

```
Console::ReadLine();
```

Обратите внимание, что команда Console: :ReadLir.e ничего не отображает на экране. Обычно для получения от пользователя каких-либо данных вначале нужно использовать команду Console: :WriteLine, чтобы отобразить на экране соответствующий вопрос, а затем уже применить команду Console : : ReadLine, чтобы принять от пользователя ответ. Например:

```
Console::WriteLine(L"Нажмите Enter, чтобы завершить
                    выполнение программы");
Console::ReadLine();
```

Функции Console являются частью .NET библиотек CLR. Поэтому, перед тем как их использовать, нужно указать Visual C++ .NET, в каком месте их искать. Сделать это можно с помощью ключевого слова #using. Все функции CLR разбиты на множество динамических библиотек (файлы с расширением DLL). Строки со словом #using сообщают компьютеру, функции каких именно библиотек будут использованы в данной программе.

CLR является аОбревиатурой от *Common Language Runtime* (переводится приблизительно как *общий язык выполнения программ*), но, как вы уже знаете, представляет собой нечто иное, а именно совокупность библиотек с различными полезными функциями. Название Common Language Runtime было выбрано исходя только лишь из маркетинговых соображений. Кстати, сокращенное название CLR впервые упоминается только *в* этой книге, так что, если вы произнесете его в разговоре со своими друзями-программистами и они не поймут, о чем речь, просто посмотрите на них свысока и скажите, что они отстали от жизни.

Большинство основных функций .NET, включая функции Console, расположены в библиотеке mscorlib. Ее название происходит от слов Microsoft core library, что переводится как "корневая библиотека". Возможно, это поможет вам запомнить ее название. Если в своей программе вы захотите использовать функции библиотеки mscorlib.dll, наберите такую строку:

```
#using <mscorlib.dll>
```

Все функции CLR разбиты на отдельные группы, обозначаемые термином *manespace (пространство имен)*. Эти группы состоят из классов (более подробно классы описаны в главе 17). А классы, в свою очередь, содержат функции. Например, функции ReadLine и WriteLine принадлежат классу Console. Класс Console относится к группе System. Таким образом, полное имя функции ReadLine будет выглядеть так: System: :Console ::ReadLine.

Ключевые слова using manespace могут избавить вас от необходимости каждый раз при вызове функции набирать название ее группы. Например, в кодах программы HelloWorld вы могли видеть такую строку:

```
using namespace System;
```

Именно благодаря ей далее по тексту программы можно набирать Console: : ReadLine вместо System: : Console: : ReadLine.

Обратите внимание, что ключевые слова #using и using namespace не являются частью стандартного языка C++. Точно так же, как и все функции CLR. Поэтому все команды, с которыми вы познакомились до этого, могут быть использованы только при создании программ .NET. Далее в главе вы узнаете, как написать аналогичные программы, используя возможности стандартного языка C++.

Не скупитесь на комментарии

Только вы можете точно знать, для чего вами была набрана та или иная команда. Но если коды вашей программы должен будет смотреть еще кто-то, он этого, скорее всего, не поймет. Или, если спустя несколько лет вы захотите внести в программу некоторые изменения, вы уже вряд ли сможете вспомнить, как это все работает. Вот почему так важно использовать комментарии. *Комментарии* объясняют простым русским языком (или не русским, но обязательно тем, на котором вы привыкли разговаривать), что вы пытались сделать, набирая все эти коды.

В кодах программ, приводимых в этой книге, вы уже сталкивались с комментариями. Распознать их помогут две косые черты (//). Как вы уже могли догадаться, весь текст строки, следующий за ними, является комментарием. (Прочитайте раздел "Старый формат комментариев", чтобы знать, как еще можно создавать комментарии.)

Комментарий может быть выделен в отдельную строку:

```
//Это строка комментариев.
```

Также комментарий может завершать собой какую-то строку:

```
a = 10; //Присвоение переменной значения 10.
```

Старый формат комментариев

В языке C++ для обозначения комментариев используются две косые черты (//). Ранее в языке C комментарии выделялись несколько иначе; начало комментария обозначалось символами /*, окончание - */. Этот же способ по-прежнему можно использовать в языке C++. Например:

```
/* Это старый способ выделения комментариев */
a = 10; /*Присвоение переменной значения 10*/
/*Переменной a*/ a = 10; /*присваивается значение 10*/
```

При использовании старых комментариев самое главное - не забыть обозначить их окончание! Если вы забудете набрать в конце комментария символы */, компилятор проигнорирует все, что было набрано после открытия комментария символами /*.

Обратите внимание, что, когда для создания комментария используются две кхые черты (//), никакими специальными символами заканчивать строку не нужно. Две косые черты говорят компилятору, что далее вся строка (и только эта строка) является комментарием. Однако, в отличие от старого формата, если комментарий занимает несколько строк, каждую новую строку должны предварять две кхые черты {//).

Исходный код программы HelloWorld

Теперь, когда вы уже достаточно знаете о принципах создания программ Visual C++ .NET, посмотрите на исходный код созданной ранее в этой главе программы HelloWorld. Он должен быть вам понятен (в нем действительно нет ничего сложного).

```
//Это код исходного файла приложения VC+ +
//Программа была создана с использованием мастера


#include "stdafx.h"


#using <mscorlib.dll>


using manespace System;


//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
   Console::WriteLine("Hello World");
   Return 0;
}
```

Как видите, программа начинается с комментариев. Далее следуют строки с ключевыми словами `#using` и `using namespace`, определяющие способ использования CLR. Затем, после еше одного комментария, следуют коды функции `main`, с которых, собственно, и начинается выполнение программы. На экране будет отображена надпись `Hello World`, сразу после чего выполнение программы прекратится. Строка `return 0` говорит о том, что функция `main` возвращает нулевое значение. Вы можете пока не обращать на нее внимания.

Строки бывают разными

В компьютерной терминологии текстовое значение, такое как "Hello World", называется *строкой (string).* При использовании строк перед ними можно добавлять специальные буквы, предоставляющие компьютеру дополнительную информацию. Например, если строка должна быть интерпретирована как набор международных символов {Unicode), наберите перед ней букву L:

```
Console::WriteLine(L"Hello World");
```

Строки Unicode могут содержать в себе любые международные символы, включая символы японского алфавита. Однако в языках .NET, если вы не укажете, что строка содержит символы других алфавитов, она все равно будет преобразована к строке Unicode в процессе выполнения программы. Поэтому самостоятельно указывать, что строка является строкой Unicode, особого смысла не имеет.
Намного разумнее набрать перед текстовым значением букву S:

```
Console::WriteLine(S"Hello World");
```

Это будет сигналом компьютеру использовать встроенный класс string, что предоставит вам дополнительные возможности по обработке этих данных.

Пару штрихов к программе Hello World

Теперь добавим некоторые изменения к программе HelloWorld. Для начала обозначьте все строки как строки .NET (поставьте перед ними буквы S). Затем добавьте в конце программы несколько команд, которые позволят выполнять ее, находясь в среде разработки Visual C++ .NET (IDE), и видеть результат выполнения до того, как окно исчезнет с экрана. Итак, окончательный код программы должен выглядеть следующим образом:

```
Console::WriteLine(S"Hello World");
//Ожидание, пока пользователь не решит
//остановить выполнение программы
Console::WriteLine(S"Нажмите Enter, чтобы завершить выполнение
                    программы");
Console::ReadLine();
```

Забудем о .NET

Все программы, которые были рассмотрены ранее, являлись программами .NET. Это означает, что они не будут работать вне среды .NET. Напротив, C++ является стандартизированным языком программирования, и созданные с его помощью программы могут работать в различных компьютерных системах. Далее вы узнаете, как создать обычную программу C++ для отображения на экране надписи "Hello World". Напомним, что разработчики .NET назвали программы, созданные без использования возможностей .NET (к числу которых относятся и обычные программы C++), *неуправляемыми программами.*

Однако, перед тем как приступить к программированию, познакомьтесь еще о некоторыми тонкостями терминологии. В среде .NET доступ к встроенным функциям обеспечивает CLR. В обычном C++ функции расположены в различных библиотеках. В .NET нужно использовать ключевое слово frusing, чтобы получить доступ к функциям CLR. В C++ для использования других функций нужно подключить *заголовочный файл (header file).* Заголовочный файл сообщит компьютеру названия и характеристики подпрограмм из других файлов или библиотек, которые вы намериваетесь использовать. Например, предположим, что вы хо-

тите использовать функцию foo, хранящуюся в библиотеке. Для этого необходимо подключить заголовочный файл, который сообщит компьютеру, что foo — это подпрограмма, и укажет, данные каких типов потребуются для ее выполнения.

Если вы создаете подпрограмму в одном файле и хотите использовать ее затем в другом файле, нужно создать заголовочный файл, описывающий эту подпрограмму. Подключая этот файл к другим файлам, вы можете повторно использовать созданную подпрограмму в других программах.

Для чего нужен компоновщик

Когда вы определяете внешнюю функцию в заголовочном файле, вы указываете только ее название и параметры, но не название самой библиотеки или исходного файла, в которых она может храниться. Компилятор генерирует список всех используемых программой функций, куда входят названия как внешних функций, так и функций, созданных в самой программе. После компилятора в действие вступает компоновщик (linker). Помимо прочего, *компоновщик* проверяет список названий всех требующихся для выполнения программы функций и ищет сами функции в других файлах и библиотеках. Если функция найдена, она используется автоматически. Если компоновщик не может найти функцию, он выдает сообщение об ошибке.

Отображение информации на экране

Если вы хотите сообщить какую-то информацию пользователю, ее нужно отобразить на экране монитора. Работая с .NET, мы использовали для этого команду Console: :WriteLir.e. В неуправляемых программах вместо нее используется выражение cout (читается как see-out — "вижу-вывожу"). Если вы даете выражению cout какое-нибудь значение, оно с радостью выводит его на экран. Чтобы передать выражению cout значение, используется команда <<.

Как и в случае создания программ .NET, текстовые значения должны быть взяты в кавычки.

Ниже приведено несколько примеров. В каждом случае все, что следует за командой cout «, будет выведено на экран.

```
// Отображается Hello World
cout << "Hello World";
// Отображается Apollo 16
cout << "Apollo 16";
```

Можно также одной командой отобразить сразу несколько значений, как это сделано в следующих двух примерах. В первом примере отображается текст "Меня зовут " и "Мишель" (это две разных строки). Во втором примере отображается одновременно текст "Apollo " и число 16.

```
//Отображается текст Меня зовут Мишель

cout << "Меня зовут " << "Мишель";
//Отображается Apollo 16
cout << "Apollo " << 16;
```

Как видите, комбинировать текст и числа очень легко. Обратите также внимание, что каждое выражение заканчивается точкой с запятой.

Ну как? Не правда ли, в C++ отображать информацию на экране еще проще, чем в среде .NET?

Специальные символы

Есть целый набор специальных символов, определяющих параметры печати.

\n Продолжить со следующей строки

\t Табуляция

\b Вернуться на один знак назад

\f Продолжить со следующей страницы

\\ Отобразить символ \

\' Отобразить символ '

\" Отобразить символ "

Например, нужно отобразить на экране сообщение "И снова "Здравствуйте!"".

Для этого наберите такую команду:

```
cout « "И снова \"Здравствуйте!\"\n";
```

Печать с новой строки

Как видите, специальных символов не так уж мало. Наиболее часто используемым среди них является символ \n, который выполняет переход на новую строку. (Его иногда так и называют— *символ новой строки.)* Например, если вы хотите, чтобы выводимый текст отображался в разных строках, наберите такие команды:

```
//Отображается текст Меня зовут Мишель
cout << "Меня зовут " << "Мишель\n";
//Отображается Apollo 16
cout << "Apollo " << 16 << "\n";
```

Символ \n интерпретируется так же, как и обычные символы. Другими словами, вы можете использовать его отдельно (как во втором примере: 16 << "\n") либо в любом сочетании с другими символами (как в первом примере: Мишель\n). Можно набрать нечто вроде Миш\nель, в результате чего в одной строке будет отображено "Миш", а во второй строке "сль". В отличие от обычных символов, \n не отображает ничего на экране; он используется только для перехода на новую строку.

Поначалу использование всех этих стрелочек (<<) и всяких специальных символов может показаться крайне неудобным и слишком запутанным, но вскоре вы сами удивитесь, насколько быстро к ним привыкнете.

При использовании функции cout, специальный символ \n не является единственно возможным средством для начала новой строки. В C++ есть также специальная функция endl (ее название образовано от слов end line— конец строки), которая выполняет то же действие:

```
cout << "Меня зовут Джимми" « endl;
cout « "Мой номер " << 12 << endl;
```

В отличие от \n, функцию endl нельзя использовать в конце или в середине строки. Поэтому при необходимости нужно просто разделить строку на две части, как показано ниже:

```
Cout « "Меня зовут " << endl << "Джимми" « endl;
```

Хотя использовать функцию endl несколько сложнее, чем символ \n, код с ее участием намного проще для чтения и понимания.

Обратная связь: получение ответа

Считывать информацию, набранную пользователем, так же просто, как и отображать данные на экране. Для этого используются функция `cin` и команда >>. (название функции произносится как "sin", что означает "see-in" — "вижу-получаю"). Например, если нужно, чтобы пользователь указал свой номер телефона, наберите;

```
cin » НомерТелефона;
```

Обычно информация, поступающая от пользователя, сохраняется как значение какой-нибудь переменной. Об использовании переменных речь идет в главе 8.

Использование библиотечных функций

Полученных вами знаний уже почти достаточно, чтобы вы могли приступить к созданию своей первой неуправляемой программы. Однако функции `cout` и `cin` являются частью библиотеки, и, как отмечалось ранее в главе, чтобы использовать функции библиотек, нужно подключить к программе заголовочный файл. Делается это с помощью команды `#include`.

Все команды, которые начинаются с символа #, называются *директивами препроцессора*. Этим длинным названием обозначаются команды, дающие указания компилятору. Директивы препроцессора не преобразуются в коды программы, они лишь управляют процессом компиляции.

Например, директива `#include` указывает компилятору, какой файл нужно подключить к программе. Описания функций `cin` и `cout` хранятся в файле, именуемом `iostream.h`. (Расширение `.h` является стандартным для заголовочных файлов.) Чтобы эти описания стали доступными, наберите в начале программы такую строку:

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

Эта команда подключает файл с описаниями `cin`, `cout` и многих других функций, являющихся частью библиотеки `iostream`.

Обратите внимание, что строки с директивами препроцессора не заканчиваются точкой с запятой. Отдельная директива препроцессора может состоять только из одной строки. Поэтому компилятор сам знает, что окончание строки означает окончание директивы.

Почему настоящие хакеры вместо < > чаще набирают " "

За командой `#include` следует название заголовочного файла. Если этот файл является стандартным, т.е. тем, который поставляется вместе с компилятором, возьмите его название в угловые скобки (< >):

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

Компилятор знает, где искать свои собственные заголовочные файлы, и сразу переходит к нужной директории. Если вы хотите подключить заголовочный файл, который вы создали сами, возьмите его название в кавычки. Это будет указанием компилятору начать поиск в текущей папке, а в случае необходимости продолжить в папке со стандартными заголовочными файлами:

```
#include "foo.h"
```

Можно также указать полный путь к файлу:

```
#include "\michael\jukebox\foo.h"
```

Язык C++ имеет очень мало встроенных команд, но зато очень много библиотечных функций. Многие библиотечные функции доступны всем компиляторам C++, поэтому вы всегда можете получить к ним доступ и использовать их в своей работе. Другие функции являются надстройками (add-ons). Можно, например, приобрести дополнительные библиотеки, содержащие функции, для проведения статистических вычислений или для обработки изображений.

Итак, займемся делом

Теперь вы знаете достаточно для того, чтобы создать версию программы HelloWorld, написанную на стандартном C++. А потому приступим.

1. Выберите команду **File⇨New⇨Project (Файл⇨Создать⇨Проект).**
2. В разделе **Project Types (Тип** проекта) выберите **Visual C++ Project.** В разделе Templates (Шаблоны) выберите **Win32 Project.**
3. В поле Name (Название) наберите `HelloWorld2`.

   По умолчанию Visual C++ приведет в действие команду Close Solution (Закрыть задачу), в результате произойдет закрытие текущей задачи (открытой для создания проекта HelloWorld) и открытие повой для создания проекта Hel!o\Vorid2. Не отменяйте выполнение этой команды. Одна задача может содержать в себе несколько проектов (для этого, собственно, задачи и существуют), однако разработка многопроектных задач — дело далеко не из легких, поэтому лучше для каждого нового проекта открывать новую задачу.
4. Щелкните на **кнопе** OK.

   Будет запущен мастер Win32 Application Wizard, окно которого появится на экране (рис. 3.4).

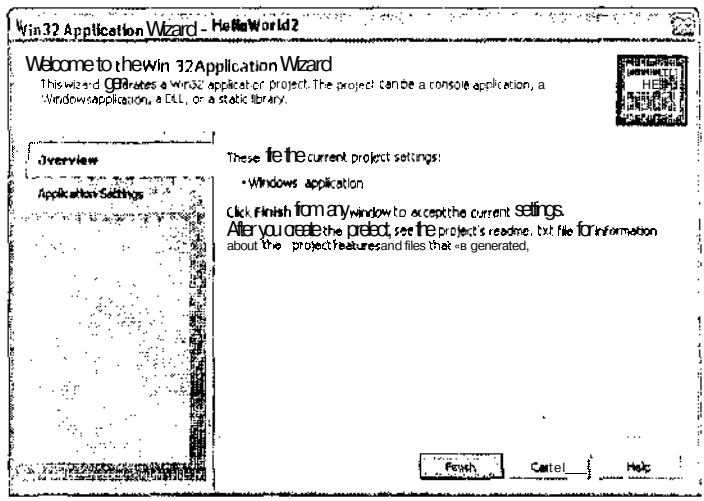


Рис. 3.4. Мастер Win32 Application Wizard поможет вам создать программу HelloWorld2

5. Щелкните на вкладке **Application Settings.**

   Мастер отобразит настройки, которые можно установить для создаваемой программы.
6. Выберите опцию **Console Application** вместо установленной по умолчанию Windows **Application.**

   (Console Application также является приложением Windows, как и Windows Application. Но почему-то многие, увидев название "Windows Application", думают, что после выбора этого шаблона будет сразу же создано приложение с набором окон, кнопок и с потрясающей графикой. В действительности все это нужно создавать самостоятельно, используя возможности Visual C++.)

7. **Щелкните на кнопке Finish (Готово).**

   Через несколько секунд будет создана новая задача, уже содержащая какие-то исходные коды.

8. **В окне Solution Explorer дважды щелкните на названии файла `HelloWorld2.cpp`.**

   Visual C++ откроет файл в окне редактора кодов. Вы увидите, что Win32 Application Wizard уже создал для вас какие-то коды. Это очень мило с его стороны, не так ли? Конечно, большое спасибо, но мы хотим набрать свои собственные коды.

9. **Нажмите комбинацию клавиш <Ctrl+A>, чтобы выделить все коды, и затем клавишу <Delete>, чтобы избавиться от них.**

10. **А теперь наберите такой код:**

```
\\ HellcWorld2
\\ Отображение на экране слов Hello World
\\ Неуправляемая


#include "stdafx.h"
#include <iostream.h>


int _tmain(int argc, _TCHAR* argv[])
{
  \\Вывод на экран
  cout -:< "Hello World\n";
  return 0;
}
```

Синтаксические ошибки

Если вы что-то неправильно наберете или неверно используете команду C++, компилятор сообщит, что вы допустили синтаксическую ошибку. Это будет означать, что компилятор либо не может распознать набранное вами слово, либо вы что-то пропустили при использовании команды. Синтаксические ошибки бывают самых разных видов. Большинство из них, а также способы их устранения описаны в главах 23 и 24.

Вы должны помнить еще по программе HelloWorld, что интегрированная среда IDE не позволила увидеть результат выполнения этой программы. С тех пор ничего не изменилось, поэтому программу HelloWorld2 также придется выполнять через окно ввода команд, предварительно построив ее. Для этого выполните ряд действий.

1. **Щелкните правой кнопкой мыши на названии `HelloWorld2` (на том, которое выделено полужирным шрифтом, а не на `HelloWorld2.cpp`) и выберите команду Build (Построить).**

   Visual C++ откроет окно Output, в котором вы сможете наблюдать процесс построения программы и видеть возможные ошибки. Когда построение программы HelloWorld2 будет закончено, в окне Output отобразится сообщение: "Build: 1 succeeded, 0 failed, 0 skipped (1 успешно, 0 неудачно, 0 пропущено)".

2. **После того как программа Hello\Yorld2 будет построена, откройте окно** для ввода **команд (Command Prompt).**

3. **Используйте команду CD, чтобы перейти из текущего каталога в каталог, где сохранен проект HelloWorld2.**

4. Запустите программу, **набрав** команду **Debug⇨HelloWorld2**.
Результат выполнения программы показан на рис. 3.5.

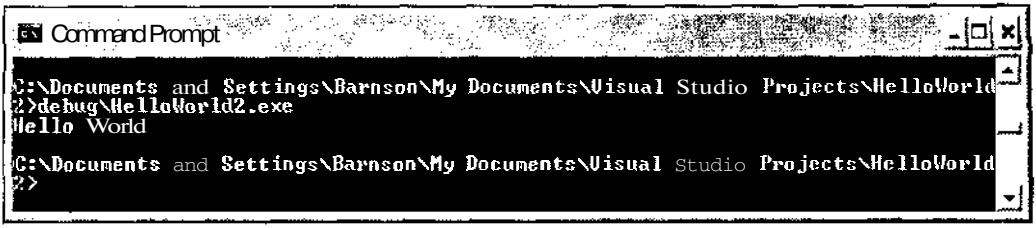


Рис. 3.5. Выполнение программы HelloWorld2 в окне для ввода команд

Вы могли обратить внимание, что в созданной стандартной программе C++ функция `main` имеет два параметра. Они называются параметрами командной строки. Это означает, что при запуске программы из окна **Command** Prompt, ей можно передать значения двух параметров. Например, в этом окне можно набрать команду `dir *.*`. Здесь символы `*.*` будут параметром командной строки. Если набрать команду `HelloWorld2 *.*`, то символы `*.*` будут восприняты как значения для параметров `argc` и `argv`. Однако вряд ли вам это когда-нибудь пригодится, поэтому не засоряйте голову подобными мелочами.

Глава 4

Принятие решений — дело серьезное

В этой главе...

- Что такое решения и проекты
- Создание новых решений и проектов
- Возможности окна Solution Explorer

Если вы читаете эту книгу по порядку, то вы уже должны были создать несколько .NET-программ. В процессе их создания использовались многие инструментальные средства разработки языка Visual C++. Закончив читать часть I книги, вы будете уметь пользоваться еще большим количеством возможностей окружения Visual C++. Основной темой этой главы являются решения и проекты.

Решения и *проекты* упрощают процесс создания программ благодаря объединению всех необходимых исходных файлов и прочих сопутствующих элементов. Файл проекта содержит информацию обо всех исходных файлах, из которых состоит программа. Файлы проектов также упрощают добавление других исходных файлов к программе и позволяют контролировать различные параметры присоединения этих файлов к программе. Решение — это просто один или несколько проектов,

Правильное решение может сделать вас счастливым

Некоторые программы состоят из одного файла. Но большинство программ, в том числе и те, создание которых было описано в главах 2 и 3, имеют несколько большие размеры. Для их выполнения требуются различные файлы с исходными кодами, а также заголовочные файлы и библиотеки. Чтобы создать выполняемую версию программы, нужно откомпилировать все эти файлы, а затем связать их воедино.

Сделать это можно двумя способами. Первый— использовать инструмент командной строки, называемый NMAKE, и построить формирующий файл (makefile). Второй (более удобный и элегантный)— использовать решения и проекты. *Формирующий файл* состоит из списка команд, с помощью которых создается приложение. Например, в этом списке могут быть команды, указывающие, что нужно откомпилировать подпрограмму foo, подпрограмму bar, затем связать их с библиотекой muck и т.д. Создание формирующих файлов сопряжено с многими трудностями. Нужно знать множество деталей о том, как файлы компилируются и связываются. Более того, нужно знать специальный язык создания формирующих файлов!

Однако большим преимуществом создания формирующих файлов является тот факт, что инструмент NMAKE определяет, какие файлы были изменены с момента последнего создания приложения. Поэтому при повторном построении приложения заново откомпилированы будут только эти измененные файлы. Иногда это очень экономит время.

Файлы проектов — это те же формирующие файлы, которые Visual C++ создает автоматически и позволяет управлять ими с помощью визуальных средств, без необходимости знания всех деталей работы компилятора.

Файлы проектов сделают вашу жизнь проще

Файлы проектов, как и формирующие файлы, предназначены для построения приложений. Однако есть несколько причин, по которым использование файлов проектов намного проще, чем формирующих файлов. Первая — при использовании файлов проектов компилятор сам просматривает исходные файлы и определяет все зависимости. *(Зависимости* — это наборы файлов, изменение которых требует повторной компиляции проекта.)

Вторая причина— управление файлами проектов осуществляется визуальными средствами, что намного удобнее и проще. Кроме того, поскольку Visual C++ сам знает, как компилируются файлы C++, как они связываются и т.п., вам самим не нужно точно регламентировать этот процесс. Все, что от вас требуется, — просто добавлять исходные файлы к файлу проекта. Всю остальную работу сделает Visual *C++*. Удобно, не правда ли?

Кстати, вы уже могли сталкиваться с файлом проекта. Если вы пробовали самостоятельно воспроизвести примеры программ из предыдущих глав, мастер AddWizard создавал для ваших программ файл проекта, содержащий список всех исходных файлов.

Можно использовать окно Solution Explorer для решения разнообразных задач программирования. Например, в нем можно просмотреть список всех исходных файлов, из которых состоит проект, и открыть любой из них для редактирования. Там же можно контролировать все параметры, в соответствии с которыми выполняется построение приложения. Можно осуществлять саму компиляцию приложения. Более детально возможности этого окна описаны в разделе "Что может окно Solution Explorer".

Решения и проекты

В справочной системе Visual C++ слова *solution {решение)* и *project {проект)* часто используются вместе. Вообще говоря, эти два слова обозначают разные веши. *Решение* состоит из одного или более проектов. Наиболее часто все же одно решение содержит один проект. Вы сможете смело называть себя хакером, если научитесь создавать решения из нескольких проектов.

Поскольку решения и проекты, по сути, являются почти одним и тем же, мы будем употреблять их как взаимозаменяемые понятия, за исключением тех случаев, когда информация будет относиться только к одному из них.

Построение программ

Построение любой программы подразумевает создание файла проекта и решения. Файл проекта сообщает компилятору, какие исходные файлы нужно откомпилировать для построения приложения. Также в нем содержится информация для компилятора о том, какие библиотеки должны быть присоединены.

Определение параметров нового проекта

Создать новый проект очень просто. Для начата выберите команду File⇨New⇨Project (Файл⇨Создать⇨Проект), чтобы открыть показанное на рис. 4.1 диалоговое окно New Project (Создание проекта). Это окно, в котором определяются параметры будущего проекта. Здесь нужно указать название проекта, а также каталог, в котором он будет сохранен. Здесь же выбирается тип создаваемого проекта и один из стандартных шаблонов.

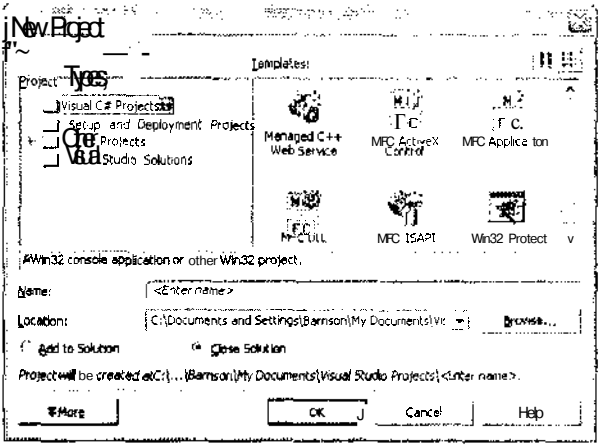


Рис. 4.1. Использование окна New Project для создания нового проекта

- ✓ **Project Types** (Тип проекта): в этом разделе определяется тип создаваемого проекта. В данной книге будет рассмотрен только тип Visual C++ Project. В зависимости от того, какой установочный пакет Visual C++ вы приобрели, список доступных типов проектов может существенно отличаться от показанного на рис. 4.1.
- ✓ **Templates** (Шаблоны): здесь можно выбрать один из подвидов проектов Visual C++, основанный на выбранном в разделе Project Types типе. Проекты Visual C++ имеют более дюжины шаблонов. Некоторые из этих шаблонов в действительности являются мастерами, которые предоставляют множество возможностей для контроля над тем, какие именно свойства будут иметь созданные вами программы. С двумя из них вы будете часто сталкиваться в процессе чтения книги. Это шаблоны Win32 Project и Managed C++ Application. (Есть много других вариантов, например шаблоны для создания различных средств управления, библиотек и других хитрых штучек.)
- ✓ **Name** (Название): в этом поле нужно указать название нового файла проекта. Для проекта обычно используется то же название, что и для создаваемого с его помощью приложения.
- ✓ **Location** (Размещение): укажите в этом поле папку, где должен быть сохранен создаваемый проект. Можно щелкнуть на кнопке Browse (*Обзор*), чтобы найти существующую папку. Если указанной папки еще нет, Visual C++ создаст ее автоматически. Хорошей практикой является хранение всех исходных файлов, имеющих отношение к определенному проекту, в отдельной папке. Предположим, например, что вы создаете приложение, именуемое Jukebox9, и хотите сохранить его в папке МояРабота\Jukebox9. Вначале щелкните на кнопке Browse, чтобы найти папку МояРабота, или сами укажите путь к ней в поле Location. Затем в поле Name в качестве названия проекта наберите Jukebox9. Visual C++ автоматически создаст папку Jukebox9 в папке МояРабота.
- ✓ **Add** to Solution/Close Solution (Добавить к решению/Закрыть решение): как вы помните, решение может состоять из одного ИЛИ нескольких проектов. Поэтому при создании нового проекта его можно *либо* добавить к открытому в данный момент решению **(Add** to Solution), либо закрыть текущее решение (Close Solution) *и* создать новое для нового проекта.

После того как вы укажете всю необходимую информацию, щелкните на кнопке OK. Затем для большинства типов проектов открывается мастер, позволяющий определить дополнительные параметры создаваемого проекта. Некоторые мастера, такие как Win32 Application Wizard, с которым вы встречались в главе 3, состоят из нескольких шагов. Другие, попроще, состоят из одного шага и позволяют лишь добавить какие-то файлы к вашему проекту.

Добавление файлов к проекту

Чтобы добавить новый файл, выполните ряд действий.

1. Выберите команду **File⇨New⇨File (Файл⇨Создать⇨Файл).**
   Откроется диалоговое окно New File (рис. 4.2).

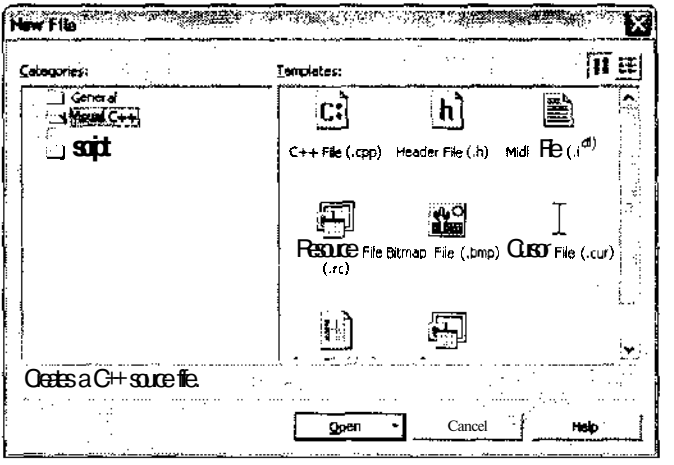


Рис. 4.2. Использование диалогового окна New File для создания нового исходного файла

2. В разделе **Categories** (Категория) выберите пункт **Visuaf C++.**
3. **В** разделе **Templates** (Шаблоны) выберите **C++ File(.cpp).**
4. Щелкните на кнопке **Open (Открыть).**

К проекту будет добавлен пустой файл C++.

Чтобы добавить уже существующий файл, выполните следующее.

1. Выберите команду **Project⇨Add Existing Item (Проект⇨Добавить существующий элемент).**
   Откроется диалоговое окно Add Existing Item. Принцип его работы тот же, что и у всех диалоговых окон Windows, предназначенных для открытия файлов.
2. Выберите файл, который хотите добавить к проекту.
   Если вы хотите добавить сразу несколько файлов, выделите их, удерживая нажатой клавишу <Ctrl>. После того как файлы будут добавлены к проекту, окно Solution Explorer будет выглядеть приблизительно так, как показано на рис. 4.3.

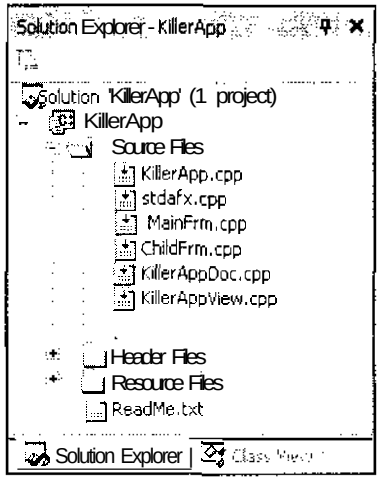


Рис. 4.3. В окне Solution Explorer отображается список файлов, из которых состоит ваш проект (решение)

Что может окно Solution Explorer

Окно Solution Explorer может быть использовано для решения множества текущих задач. Ниже приведен список некоторых из них, к которым вы будете наиболее часто возвращаться в процессе разработки приложений.

- ✓ **Просмотр и редактирование файлов, из которых состоит проект.** Дважды щелкните на названии файла, чтобы открыть его в окне редактора кодов. Если этот файл еще не имеет реального наполнения, Visual C++ спросит, хотите ли вы создать новый файл. Ответьте "Да", чтобы увидеть перед собой девственно чистое окно редактора кодов.
- ✓ **Добавление к проекту новых файлов.** Щелкните правой кнопкой мыши на названии проекта и выберите команду Add⇨Add Existing Item (рис. 4.4). Эти действия эквивалентны выбору команды Project⇨Add Existing Item.

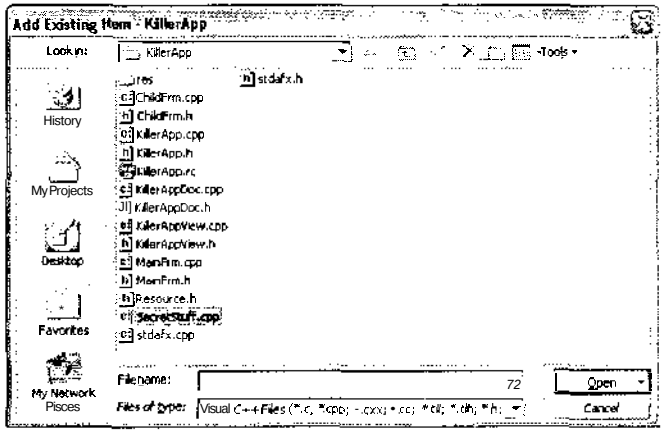


Рис. 4.4. Использование Solution Explorer для добавления к проекту уже существующих файлов

Окно Solution Explorer отображает зависимости проекта в папках Header Files (Заголовочные файлы) и Resourse Files (Файлы ресурсов), как показано на рис. 4.5. Файлы ресурсов представляют собой отдельные ресурсы программы, такие как растровые отображения графических объектов и пиктограммы. Они являются такими же зависимостями, как заголовочные файлы.

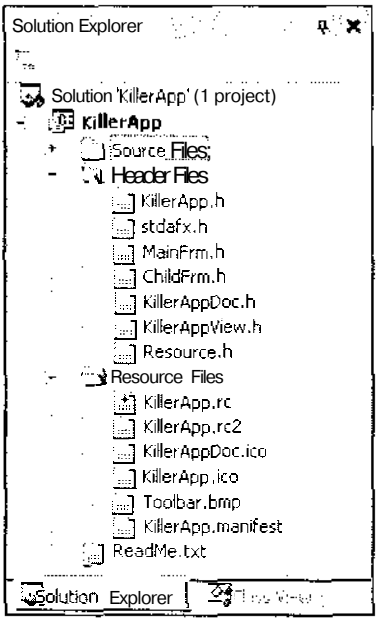


Рис. 4.5. Отображение зависимостей проекта в окне Solution Explorer

Решения и проекты являются важным аспектом Visual C++. Их использование необходимо для построения программ и для быстрого поиска и просмотра нужных исходных кодов. При разработке приложений работа в окне Solution Explorer занимает наибольшее количество времени. После написания кодов, конечно.

Глава 5

Хороший редактор — что еще нужно бывалому программисту?

В этой главе...

- Открытие и сохранение файлов в окне редактора кодов
- Обычные и специальные задачи редактирования
- Выделение цветом отдельных "элементов программы
- Получение контекстно-зависимой справки
- Использование строки перехода
- Сокрытие отдельных фрагментов кодов
- Поиск нужного текста

Большую часть процесса разработки программы занимает написание кодов в окне редактора. Как хороший текстовый редактор кардинально упрощает процесс написания книги (ну хорошо, не кардинально, но все же упрощает), так и хороший редактор кодов значительно облегчает процесс создания программ. *Редактор кодов* позволяет решать не только обычные задачи редактирования, такие как копирование, вырезание и вставка текста, но и специальные задачи, касающиеся создания программ.

Visual C++ имеет многофункциональный настраиваемый редактор кодов, позволяющий решать многие специальные задачи, например такие, как одновременный сдвиг нескольких строк или быстрое подключение заголовочного файла. В этой главе дается краткий обзор наиболее важных возможностей этого редактора.

Коды существуют для того, чтобы их редактировать

Visual C++ позволяет открывать для редактирования любое количество файлов (может и не любое, но достаточно большое). Если вы открыли сразу несколько файлов, коды каждого из них будут отображаться в отдельном окне. Visual C++ поддерживает как многооконный интерфейс MDI (Multiple Document Interface), так и отображение окон в виде вкладок. (MDI — это всего лишь интерфейс, позволяющий отображать на экране одновременно несколько окон и перемещать их независимо друг от друга.) Режим отображения окон в виде вкладок установлен по умолчанию. Как обычно выглядит окно редактора кодов, показано на рис. 5.1.

Для открытия окна редактора кодов обычно используют один из перечисленных ниже способов.

- ✓ Открытие для редактирования одного из файлов проекта. Дважды щелкните на названии файла в окне Solution Explorer. Коды этого файла будут отображены в окне редактора.

- ✓ Редактирование элемента класса. Дважды щелкните на названии элемента в списке Class. Файл будет открыт в редакторе кодов, при этом курсор будет помещен перед выбранным элементом. Можно также щелкнуть на названии элемента правой кнопкой мыши и выбрать команду Go to Definition (Перейти к определению), чтобы открыть для редактирования коды исходного файла (с расширением .cpp), или выбрать команду Go to Declaration (Перейти к объявлению), чтобы открыть для редактирования коды заголовочного файла (с расширением .h).
- ✓ Создание нового файла. Выберите команду File⇨New⇨File (Файл⇨Создать⇨Файл), чтобы открыть диалоговое окно New- File. Выберите категорию создаваемого файла, его тип и щелкните на кнопке OK.
- ✓ Открытие существующего файла. Выберите команду File⇨Open⇨File. В открывшемся диалоговом окне укажите название файла, который вы хотите открыть для редактирования.

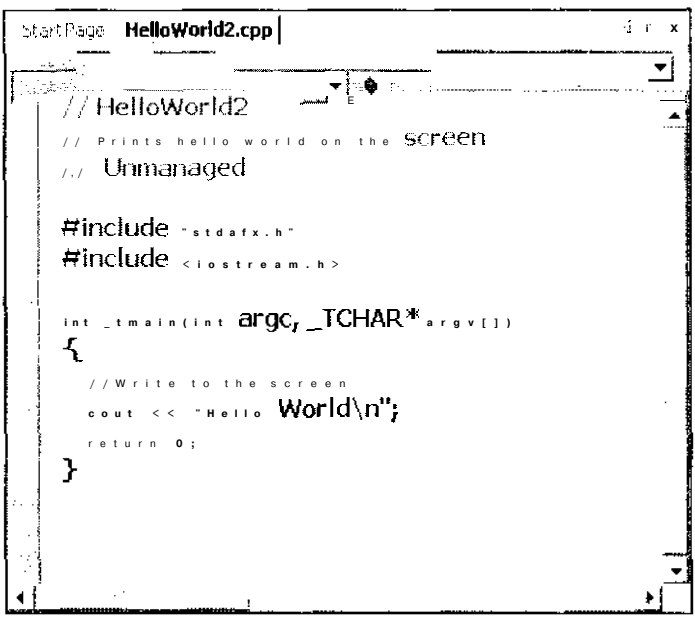


Рис. 5.1. Редактор используется для отображения, ввода и редактирования кодов программы

Сохранить отредактированный файл также очень просто. Можно даже сохранить сразу все файлы, открытые для редактирования. Вот как это делается.

- ✓ Сохранение файла. Выберите команду File\Save *ИмяФайла*.
- ✓ Сохранение файла с присвоением ему нового имени. Выберите команду File\Save *ИмяФайла* As. В диалоговом окне наберите новое имя файла и щелкните на кнопке Save (Сохранить).
- ✓ Сохранение всех файлов, в которые были внесены изменения. Выберите команду File⇨Save All. Рекомендуется это делать перед тем, как запускать на выполнение только что созданную программу. Если вы этого не сделаете и новая программа приведет к сбою в работе компьютера и к его перезагрузке, вся проделанная вами работа будет потеряна.

Приемы редактирования

В процессе написания кодов вы можете заметить, что некоторые действия приходится выполнять снова и снова. Одни из них относятся к разряду обычных (например, вырезание или копирование текста). Другие возможны только при написании программ (например, одновременное смещение группы строк или открытие заголовочного файла). В табл. 5.1 приведен список наиболее часто выполняемых задач и описание способов их решения. Более специфические задачи описаны после более общих.

Таблица 5.1. Задачи редактирования, решаемые в процессе написания кодов программ

Задача	Способ решения
Выделение текста	Щелкните в позиции, с которой должно начинаться выделение. Нажмите левую кнопку мыши, перетащите курсор к позиции, в которой выделение должно заканчиваться, и отпустите кнопку мыши. Или щелкните в позиции, с которой должно начинаться выделение, нажмите и удерживайте клавишу <Shift> и щелкните в позиции, в которой выделение должно заканчиваться. Или используйте клавиши управления курсором, чтобы перейти в позицию начала выделения, нажмите и удерживайте клавишу <Shift> и, опять используя клавиши управления курсором, перейдите в позицию окончания выделения
Вырезание текста	Выделите текст. Выберите команду Edit⇨Cut(Правка⇨Вырезать) или нажмите клавиши <Ctrl+X>. Можете затем вставить вырезанный текст в любом другом месте. (Для вырезания текста можно также щелкнуть правой кнопкой мыши на выделенном фрагменте и выбрать из открывшегося контекстного меню команду Cut.)
Копирование текста	Выделите текст. Выберите команду Edit⇨Copy(Правка⇨Копировать) или нажмите клавиши <Ctrl+C>. Можете затем вставить скопированный текст в любом другом месте. (Для копирования текста можно также щелкнуть правой кнопкой мыши на выделенном фрагменте и выбрать из открывшегося контекстного меню команду Copy.)
Вставка текста	Выберите команду Edit⇨Paste(Правка⇨Вставить) или нажмите клавиши <Ctrl+V>. В окно редактора будет вставлен текст из буфера обмена. (Можно также щелкнуть правой кнопкой мыши в позиции, где должен быть вставлен текст, и выбрать из открывшегося контекстного меню команду Paste.)
Переход в начало файла	Нажмите <Ctrl+Home>
Переход в конец файла	Нажмите <Ctrl+End>
Переход на страницу назад	Нажмите клавишу <PgUp>
Переход на страницу вперед	Нажмите клавишу <PgDn>
Переход вправо на одно слово	Нажмите <Ctrl+стрелка вправо>

Задача	Способ решения
Переход влево на одно слово	Нажмите <Ctrl+стрелка влево>
Одновременный сдвиг нескольких строк	Выделите несколько строк и нажмите клавишу <Tab>. Это может быть полезно, например, если вы добавили оператор if и хотите выделить часть кодов, которые к этому оператору относятся
Отмена сдвига нескольких строк	Выделите строки и нажмите <Shift+Tab>. Этот прием может быть полезен, если вы скопировали часть кодов в другое место программы и теперь нет необходимости специально выделять их с помощью отступов
Переход ко второй скобке из пары скобок	Нажмите <Ctri+]>, чтобы перейти от открывающей скобки (, {, < или [- курсор уже должен находиться перед ней - к соответствующей ей закрывающей скобке:),}, > или]. Тем же нажатием выполняется обратный переход. Эта возможность полезна в случае, если вы имеете много вложенных друг в друга операторов и хотите найти границы каждого из блоков. Также таким образом можно проверять, не забыли ли вы закончить вызов функции закрытием скобки
Использование закладок	Поместите курсор в том месте, где вы хотите оставить закладку. Нажмите дважды <Ctrl+K>. Таким образом вы отметите позицию в кодах программы, к которой при необходимости сможете быстро вернуться. Например, если вы хотите скопировать фрагмент кодов из одной части программы в другую, установите закладку, найдите нужный фрагмент кодов, скопируйте его, вернитесь обратно к закладке и вставьте скопированный фрагмент
Переход к закладке	Нажмите <Ctrl+K> и <Ctrl+N>, чтобы перейти к следующей закладке в файле. Нажмите <Ctrl+K> и <Ctrl+P>, чтобы перейти к предыдущей закладке
Переключение между окнами редактора	Нажмите <Ctrl+Tab> или <Ctrl+F6>, чтобы перейти к следующему окну редактора (которое было открыто после текущего). Нажмите <Shift+Ctrl+Tab> или <Shift+Ctrl+F6>, чтобы перейти к предыдущему окну редактора (которое было открыто перед текущим)
Открытие заголовочного файла, соответствующего открытому файлу	Щелкните правой кнопкой мыши на команде #include и выберите из открывшегося контекстного меню команду Open Document. При написании программ C++ редактировать заголовочные файлы приходится почти так же часто, как и исходные файлы. Иногда открывать заголовочный файл необходимо для того, чтобы просмотреть, что уже было определено
Получение справки о командах	Щелкните на том элементе программы, по которому нужно получить справку. Нажмите клавишу <F1 >. Таким образом, например, можно получать справку относительно синтаксиса вызова библиотек, Windows API или команд C++

Коды бывают разные — черные, белые, красные

В Visual C++ есть возможность выделять цветом отдельные синтаксические элементы, что значительно упрощает чтение набранных кодов. Выделив цветом отдельные элементы программы, такие как комментарии, ключевые слова, значения и переменные, вы делаете для себя их идентификацию автоматической. Этот прием также облегчает обнаружение типичных синтаксических ошибок.

Например, если все коды программы будут отображаться черным цветом, а комментарии — синим, вы сразу увидите, что забыли закрыть какой-то комментарий (поставив в его конце символы `*/`), поскольку далее за ним будет открываться бескрайнее море синего текста. Если установить, что все ключевые слова должны отображаться зеленым цветом, и если какое-то набранное ключевое слово не окрасится в зеленый цвет, вы сразу же поймете, что при его наборе допустили синтаксическую ошибку. Например, если вы наберете `claps` вместо `class`, набранное слово зеленым не станет. Точно так же, если при объявлении какая-то переменная окрасится в зеленый цвет, вы сразу поймете, что это имя для нее неприемлемо, поскольку имена переменных не должны совпадать с ключевыми словами. (Более детально правила присвоения имен переменным будут рассмотрены в главе 8.) Чтобы определить цвета для различных синтаксических элементов программы, выберите команду Tools⇨Options (Сервис⇨Параметры) и во вкладке Environment (Окружение) перейдите к опциям Fonts and Colors (Шрифты и цвета).

На рис. 5.2 показано окно редактора кодов с включенной возможностью выделения цветом. (Разумеется, поскольку вы смотрите на черно-белый рисунок, для вас это больше похоже на выделение серым цветом.)

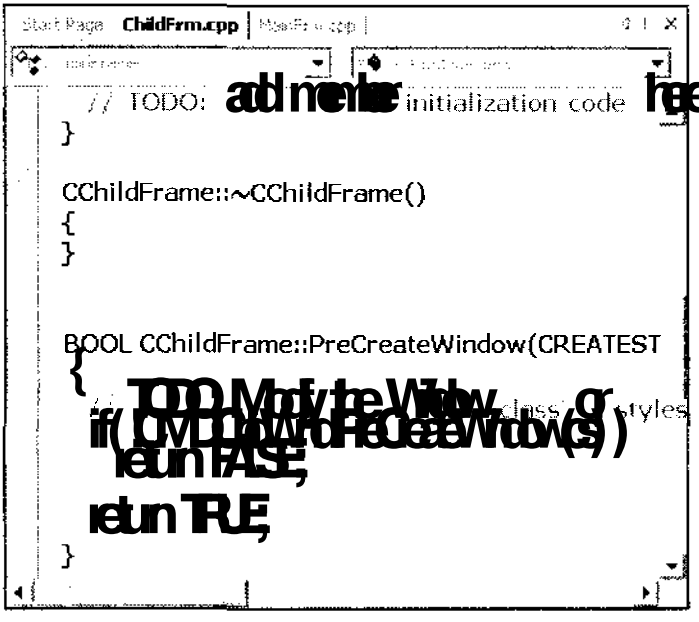


Рис. 5.2. Выделение цветом отдельных синтаксических элементов программы значительно облегчает их визуальную идентификацию

Помощь, которая всегда рядом

Если вы используете C++, окружение .NET, библиотечные функции, среду Windows и не помните или не знаете, как работает то или иное средство, либо что обозначает тот или иной элемент, нужная подсказка находится на "расстоянии" одного щелчка. Например, допустим, что вы используете библиотечную функцию `cout`, но не уверены в корректности такого выражения:

```
cout << "Bob";
```

Щелкните на слове `cout`, нажмите клавишу <F1>, и тут же откроется окно с необходимой справочной информацией.

Рассмотрим другой пример. Предположим, вы набрали такое выражение:

```
while (strlen(bar) > 10)
```

Можно щелкнуть на ключевом слове `while`, нажать клавишу <F1> и получить справку об использовании команды `while`, а можно щелкнуть на слове `strlen`, нажать клавишу <F1> и получить справку об использовании библиотечной функции `strlen`.

Навигация по просторам программы

Чтобы добиться правильного выполнения программы, в процессе редактирования ее кодов вам придется переключаться с одной функции на другую **почти** так же часто, как переходить от одной строки программы к другой в пределах одной и той же функции.

На панели перехода, расположенной в верхней части окна редактора кодов, вы можете видеть названия классов и функций, используемых в открытом в данный момент файле. Чтобы быстро перейти к нужной функции выполните следующее.

1. **Щелкните на поле со списком, расположенном слева.**

   Откроется список используемых классов.

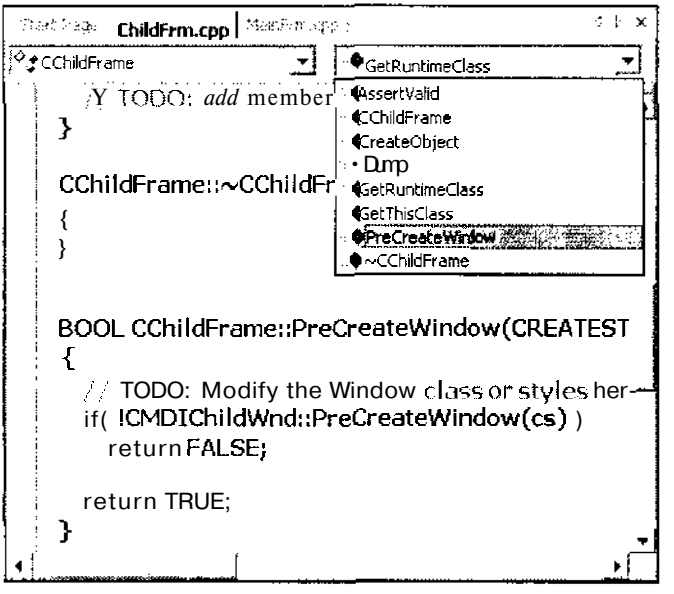


Рис. 5.3. Панель перехода автоматизирует переход к нужным элементам программы

2. **Выберите интересующий вас класс.**
3. **Щелкните на поле со списком, расположенном справа.**

   Откроется список функций, используемых в выбранном классе (рис. 5.3).
4. **Выберите функцию, коды которой хотите просмотреть или отредактировать.**

   Visual C++ переместит курсор к этой функции.

Обратите внимание, что панель перехода отображает только классы и функции, используемые в текущем файле. Если вы хотите просмотреть другие элементы классов, используйте средство Class view.

Сокрытие и отображение кодов

Иногда обилие кодов в открытых окнах нескольких проектов может создавать впечатление хаоса. Разработчики Visual C++ учли эту проблему и проедложили способ ее решения. Те фрагменты кодов, которые в данный момент вас не интересуют, могут быть *свернуты,* чтобы не занимать место на экране. А если вы снова захотите их просмотреть, их отображение можно быстро восстановить. Набор строк, который одновременно может быть свернут или вновь отображен, называется *фрагментом* кодов. Фрагменты могут быть созданы автоматически, либо вы сами можете их определить. Рядом с каждым фрагментом расположен знак "плюс" или "минус", позволяющий отображать либо сворачивать нужный фрагмент (рис. 5.4).

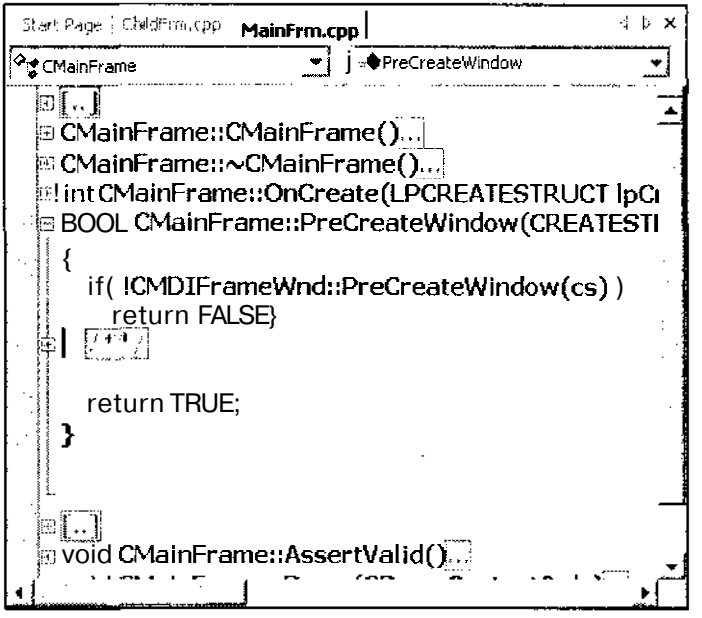


Рис. 5.4. Свернув коды, вы можете видеть "общую картину"

Существует два основных способа свертывания кодов.

- ✓ **Свертывание произвольных фрагментов.** Вы можете легко свернуть любой нужный вам набор кодов. Для этого просто выделите его, щелкните на нем правой кнопкой мыши и выберите команду Outlining⇨Hide Selection (Свернуть⇨Скрыть выделенное). Можно также выделить нужный набор кодов и в главном меню выбрать команду Edit⇨Outlining⇨Hide Selection (Правка⇨Свернуть⇨Скрыть выделенное).

- ✓ Свертывание блоков. Visual C++ автоматически выделяет структурные элементы языка C+-. Таким образом вы можете сразу сворачивать отдельные блоки программы. (Более подробно о блоках программы, к которым относятся циклы, операторы управления потоками данных и функции, речь идет в части II книги.) Чтобы пол>чить возможность сворачивать отдельные блоки, выберите команду Edit⇨Outlining⇨Collapse to Definitions (Правка⇨Свернуть⇨Сворачивать по определениям). Visual C++ выделит в отдельные фрагменты каждое определение функции, определение класса и все прочие блоки, имеющиеся в кодах открытого в данный момент окна. Также будут выделены все блоки, входящие в состав дргих блоков.

Поиск и автозамена

В процессе написания программ вы очень часто будете сталкиваться с необходимостью поиска некоторых уже набранных кодов. Вместо того чтобы самостоятельно просматривать коды всей программы в поиске нужного фрагмента, доверьте это занятие редактору кодов.

Чтобы найти какой-то фрагмент текста программы, воспользуйтесь командой Edit⇨Find and Replace⇨Find (Правка⇨Поиск и замена⇨Найти) или нажмите клавиши <Ctrl+F>. Если найденный фрагмент текста нужно сразу же заменить каким-то другим, примените команду Edit⇨Find and Replace⇨Replace (Правка⇨Поиск и замена⇨Заменить) или нажмите комбинацию клавиш <Ctrl+H>. Например, можно использовать команду Edit⇨Find and Replace ⇨Replace. чтобы найти в тексте программы все упоминания переменной `foo` и заменить их переменной `goo`.

Диалоговые окна Find (Поиск) и Replace (Замена) позволят вам определить множество дополнительных установок, контролирующих процесс поиска нужных фрагментов в тексте программы. Диалоговое окно Find показано на рис. 5.5.

Откуда берутся все эти загадочные термины

Подстановочные знаки иногда называют *regular expression matching commands* (*команды сопоставления регулярных выражений).* Этот громоздкий термин появился благодаря некоторым особенностям работы компиляторов.

При построении компиляторов создаются так называемые *лексические анализаторы* - специальные программы, просматривающие текст исходных файлов и разбивающие его на отдельные составляющие, которые компилятор сможет как-то интерпретировать. Разбивка текста файлов осуществляется на основании сопоставления его с заранее определенными образцами (или шаблонами). Эти шаблоны и называются регулярными выражениями.

Таким образом, сопоставление регулярных выражений обозначает всего лишь поиск и идентификацию текста, соответствующего заданным шаблонам.

Но вы можете называть это просто использованием подстановочных знаков.

Регулярные выражения, представленные в табл. 5.3, иногда называются *регулярными выражениями GREP*, поскольку они используются программой GREP - одной из наиболее распространенных программ поиска текста. GREP стала популярным инструментом именно в системах UNIX. А как известно, многие инструменты UNIX имеют такие своеобразные названия, как MAWK, SED, DIFF, или как GREP. В разговоре с каким-нибудь хакером, можете спокойно использовать слово *grep* вместо слова *поиск,* если вы имеете в виду поиск текста по заданному шаблону. Если это настоящий хакер, он вас поймет.

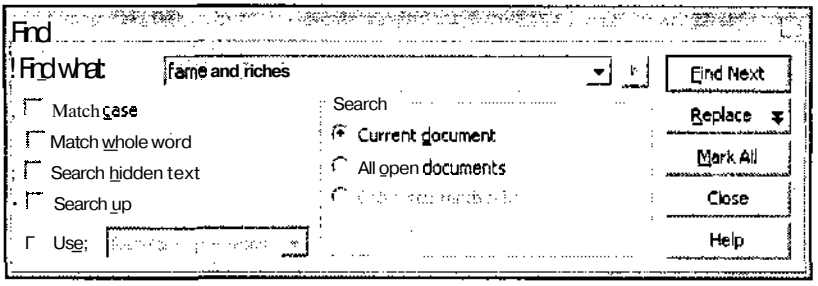


Рис. 5.5. *Укажите, что нужно найти, и вы немедленно получите результат*

Ниже приведен список основных опиий диалоговых окон Find и Replace.

Match case (С учетом регистра). В языке C++ строчные и прописные буквы считаются разными. Например, слова `Boogs` и `boogs` рассматриваются как названия двух разных переменных. Если вы активизируете опцию **Match case,** будут найдены (или заменены) только те слова, в которых очередность строчных и прописных букв точно совпадает с заданным образцом. Используйте ее только в том случае, если точно знаете, какое слово должно быть найдено (учитывая строчные и прописные быквы). Если есть какие-то сомнения, отключите эту опцию.

✓ **Match whole word only** (Только целые слова). Если эта опция активизирована, обрабатываться будут только отдельные целые слова (слова, которые отделены от других слов пробелами, запятыми или скобками). Если вы эту опцию отключите, найден будет также тот текст, который совпадает со сравниваемым образцом и является составной частью другого слова. Например, если вы отключите эту опцию и укажете для поиска текст `const`, он будет найден в таких словах, как `constant` и `deconstruct`.

✓ **Search hidden text** (Просматривать скрытый текст). Если вы активизируете эту опцию, просматриваться также будет скрытые (в результате сворачивания) фрагменты текста.

✓ **Search up** (Найти выше). Обычно поиск ведется от текущей позиции курсора вниз до конца файла. Эта опция указывает редактору, что нужно просматривать текст от текущей позиции курсора вверх до начала файла.

✓ **Use regular expression and wildcards** (Использовать регулярные выражения и подстановочные знаки). *Подстановочные знаки*— это специальные символы, которые используются для представления других символов. Например, предположим, что нужно найти какую-то переменную и вы не помните точно ее названия, но знаете, что оно начинается с буквы. В таком случае при указании образца для поиска наберите между буквами S и h подстановочный знак * (звездочка), который заменит собой произвольное количество любых других символов.

✓ Подстановочные знаки являются упрощенным вариантом более гибкого инструмента — регулярных выражений. В табл. 5.2 приведен список наиболее часто употребляемых подстановочных знаков. В табл. 5.3 вы найдете перечень основных регулярных выражений.

✓ Search: **Current window⇨Current document** (Поиск: Текущее окно⇨Текущий документ). Эта опция сообщает редактору, что нужно просматривать только тот файл, который открыт в текущем окне.

✓ **Search: All Open documents** (Просмотр всех открытых документов). Эта опция сообщает редактору, что нужно просматривать все открытые в данный момент файлы. Эта возможность может быть полезной, например, если вы хотите изменить имя класса, используемого в разных файлах проекта.

- ✓ Search: Only <current block> (Просмотр только текущего блока). Поиск может быть ограничен пределами блоков, на которые разбивается текст или коды. Если поиск в пределах блоков будет возможен, Visual C++ сделает эту опцию доступной и укажет параметры этого поиска.
- ✓ Search: Selection only (Просмотр только выделенного фрагмента). Эта опция позволяет заменить текст только в выделенном в данный момент фрагменте. Это удобно, если необходимо, например, изменить имя переменной для какой-то одной функции.
- ✓ Mark All (Пометить все). Щелчок на этой кнопке дает указание Visual C++ найти все соответствия, но не отображать их на экране, а просто пометить, с тем чтобы впоследствии вы могли к ним вернуться.

Таблица 5.2. Подстановочные знаки

Команда	Значение
?	Заменяет один произвольный символ. Например, шаблону s?ip соответствуют слова Skip, Swip и т.п.
*	Заменяет любое количество произвольных символов. Например, шаблону Sl* сооветствуют слова Sl, Slip, Sliding и др.
[]	Заменяет один из символов, указанных с скобках. Например, шаблону S[ml]ug соответствуют слова Smug и Slug
[!]	Заменяет любой символ, за исключением тех, что указаны в скобках. Например, шаблону S[!ml]ug соответствуют такие слова, как Stug, Swug, но не Smug и Slug

Таблица 5.3. Команды сопоставления регулярных выражений

Команда	Значение
.	Заменяет собой любой один символ. Например, шаблону S.ip соответствуют слова Skip, Swip и т.п,
*	Заменяет любое количество произвольных символов. Например, шаблону Sl* соответствуют слова Sl, Slip, Sliding и др.
+	Заменяет любое количество символов, предшествующих (по алфавиту) символу, после которого эта команда набрана. Например, шаблону So* соответствуют слова Soon, Son, So и др.
^	Ищет соответствие по началу строки. Например, шаблон ^// находит все комментарии, которые начинаются от начала строки
$	Ищет соответствие по концу строки. Например, шаблон foo$ находит только те слова foo, которые набраны в конце строки
['-]	Заменяет собой один из символов, набранных внутри скобок. Например, шаблону s[ml]ug соответствуют слова Smug и Slug
['^]	Заменяет собой любой символ, кроме тех, что набраны внутри скобок. Например, шаблону Б[^ml]ug соответствуют слова Saug, Skug и другие, но не Smug и Slug
[-]	Заменяет собой любую букву из диапазона, указанного в скобках (включая сами набранные буквы). Например, шаблону S[c-l]ug соответствуют только слова из набора Scug, Sdug, Seug, . . . Slug

Глава 6

Компиляция программ, или Первые трудности

В этой главе...

- Компиляция программ
- Синтаксические ошибки и предупреждения
- Различие между компиляцией, построением и повторным построением

Написание кодов программ может быть весьма увлекательным занятием. В набранные формулы и выражения можно вложить самый разнообразный смысл, затем вывести все эти коды на печать и развеселить ими своих друзей. Например, с помощью команд и операторов можно перевести на язык программирования хорошо всем известные крылатые фразы. Если бы Шекспир был программистом, он мог бы написать что-то наподобие if (_2B) {} else {}. (А если бы он был толковым хакером, он написал бы_2B ? {} : { }.)

Однако, если вы хотите, чтобы программа была действительно полезной и выполняла какие-то действия, одного лишь красивого текста недостаточно. Программу нужно откомпилировать. *Компиляция* программы — это преобразование исходных кодов программы, которые понятны программистам, в машинные коды, которые понятны компьютерам.

Процесс преобразования исходных кодов в машинные крайне сложен. Он требует точного сопоставления всех набранных вами инструкций (которые называются инструкциями высшего уровня) соответствующим инструкциям нижнего уровня (понятным компьютеру). Этот процесс завершается созданием *выполняемой* программы. Именно она может повлечь за собой какие-то реальные действия.

О процессе преобразования набранных программистами команд в машинные коды написано множество книг. К счастью для вас, разработчики, занимающиеся созданием компиляторов, все эти книги уже прочитали, поэтому вам вникать в различные тонкости этого процесса не нужно. У вас есть Visual C++, который сам откомпилирует набранные вами коды.

Начать компилировать программу очень просто

Процесс создания программы включает набор кодов, компиляцию и отладку. В большинстве случаев происходит также корректировка работы программы и добавление к ней новых возможностей, что влечет за собой многократную повторную компиляцию. К счастью, в Visual C++ компилировать программы очень просто. В действительности вы уже компилировали программу (и не один раз). если выполняли инструкции, описанные в главе 3.

Чтобы откомпилировать программу, откройте ее решение (solution). (Решения подробно описаны в главе 4.) Затем выберите команду Build⇨Build (Построить⇨Построить) или щелкните на кнопке Build одноименной панели (рис. 6.1).

Рис. 6.1. На этой панели расположены кнопки, которые запускают процесс компиляции

Visual C++ просматривает все исходные файлы открытого решения и преобразует их к машинным кодам. В результате создается выполняемая программа. На экране вы не увидите никаких изменений, но теперь у вас уже есть реальная программа, которая хранится на жестком диске. Ее можно запустить на выполнение, скопировать на дискету и подарить кому-нибудь, а можно делать с ней все, что вы считаете нужным. (Только не предпринимайте ничего такого, что может повредить вашему здоровью или здоровью окружающих.)

Если вы хотите запустить программу, выберите команду Dedug⇨Start (Отладка⇨Запустить) или щелкните на кнопке Start панели Dedug (рис. *6.2).* Если вы уверены, что в программе нет ошибок (это равносильно тому, что вы считаете себя непревзойденным хакером), выберите команду Debug⇨Start Without Debugging (Отладка⇨Запустить без отладки), чтобы запустить программу без использования отладчика Visual C++.

Рис. 6.2. На этой панели есть кнопки для запуска, остановки и пошагового выполнения программы

Синтаксические ошибки: неужели их может быть так много?/

Если при написании кодов программы вы допустили какую-то ошибку, например передали функции неверное количество аргументов, неправильно набрали команду или присвоили недопустимое имя переменной, компилятор не сможет понять ваши коды и программа не будет откомпилирована.

Если это случится (правильнее сказать: *когда* это случится, поскольку случается это практически всегда), компилятор отобразит список синтаксических ошибок в окне Output, как показано на рис. 6.3. *Синтаксическая ошибка* в окне Output сопровождается сообщением о том, что вы что-то сделали неправильно. Дважды щелкните на этом сообщении, чтобы перейти к той строке кодов программы, где эта ошибка произошла. Чтобы программа была успешно откомпилирована, необходимо исправить все синтаксические ошибки.

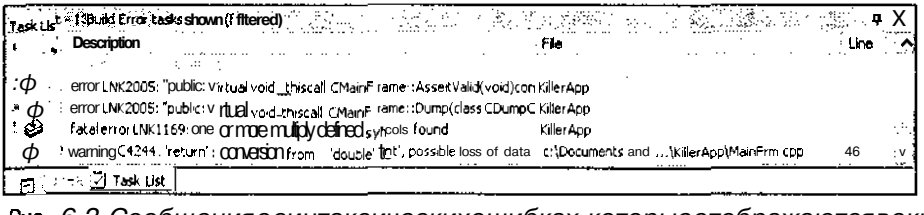


Рис. 6.3. Сообщения о синтаксических ошибках, которые отображаются в окне Output

ОДНИ ошибки очевидны и могут быть сразу же исправлены. Другие не так просты, и вам придется потратить немало времени, чтобы их обнаружить и исправить (особенно если у вас еще нет в этом большого опыта). Со временем вы научитесь быстро различать типичные ошибки и сразу же их исправлять.

Вот несколько правил, которые помогут вам в борьбе с ошибками.

- ✓ Сохраните файлы программы, перед тем как приступить к их компиляции.
- ✓ Если вы никак не можете найти ошибку в какой-то строке, проверьте предыдущую строку. Бывает, что ошибка "сидит" именно там.
- ✓ Многие ошибки случаются из-за недостающих или лишних фигурных скобок ({ }) или точек с запятой (;).
- ✓ Компилируйте программу небольшими частями. Это позволит вам сосредоточиться на отдельных фрагментах, а не рассеивать внимание по всей программе.
- ✓ Сообщение `Cannot convert from ... to ...` (Нельзя преобразовать ... к ...) обычно появляется, если вы пытаетесь присвоить переменной значение не того типа. Например, если текстовой переменной вы пытаетесь присвоить значение типа integer.
- ✓ Убедитесь, что вы правильно набрали имена переменных. Очень многие ошибки происходят именно из-за неверного указания имен переменных.
- ✓ Просмотрите главы 23 и 24. В них вы найдете перечни наиболее часто встречающихся ошибок и способы их устранения.
- ✓ Иногда одна небольшая проблема может привести к тому, что компилятор обнаружит сразу "тысячу" ошибок. Например, неправильно указанный путь к заголовочному файлу может стать причиной появления нескольких десятков сообщений об ошибках. (Обычно это случается при создании неуправляемых программ.) Достаточно устранить причину — и многие ошибки сразу же отпадут. Поэтому, если вы компилируете программу и на вас обрушивается целый поток сообщений об ошибках, не паникуйте. Скорее всего, большинство из них можно исправить одним простым действием.
- ✓ Если вы никак не можете понять, в чем заключается ошибка, попросите кого-нибудь о помощи. Иногда свежий взгляд на вещи помогает легко решить казалось бы неразрешимую проблему.

Предупреждения

Иногда в процессе компиляции помимо сообщений об ошибках вы можете сталкиваться также с предупреждениями. *Предупреждения* появляются тогда, когда компилятор может правильно интерпретировать набранные вами коды, но считает, что выполнение таких кодов может привести к возникновению проблем. Например, если вы создаете переменную, но ни разу ее не используете, компилятор предупредит вас об этом и спросит, все ли здесь правильно. Или, что значительно хуже, вы можете начать использовать переменную еще до того, как присвоите ей какое-то значение. В этом случае на экране отобразится сообщение с предупреждением, подобное показанным на рис. 6.4.

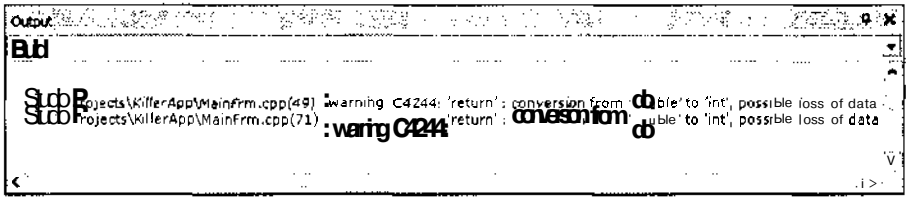


РИС 6.4. Подобные предупреждения появляются на экране, если набранные вами коды могут привести к возникновению проблем

Относитесь к предупреждениям с должным вниманием. Хотя иногда они могут появляться как следствие чрезмерной осторожности компилятора, в большинстве случаев они помогают избежать многих неприятностей. Например, если вы забудете добавить строку, задающую на-

чальное значение переменной, компилятор выдаст предупреждение. Если вы его проигнорируете, в процессе выполнения программы эта переменная будет гринимать совершенно бессмысленные значения, в результате чего программа, разумеется, будет работать неправильно.

Почему компилятор не исправляет ошибки самостоятельно

Хотя компилятору известно, где и какая ошибка произошла, он не знает, *почему* она произошла. Например, компилятор может установить, что в конце некоторой строки не хватает точки с запятой. Почему он ее просто не добавит? Потому что у этого могут быть разные причины. Вы могли просто забыть поставить точку с запятой, могли забыть набрать кроме этой точки с запятой еще какие-то коды или же могли вообще забыть выделить эту строку как комментарий.

Вместо того чтобы брать на себя ответственность и самостоятельно выбирать один из вариантов устранения ошибки, компилятор не пытается вместо вас создавать программу и только лишь указывает на ошибку, предоставляя вам право выбора дальнейших действий. Кроме того, поскольку круг вариантов исправления ошибки, а следовательно, и причин ее возникновения может быть довольно большим, компилятор иногда в состоянии только приблизительно указать на ошибку. Например, хотя компилятор может указать на некоторую строку и сказать, что здесь не хватает точки с запятой, на самом деле ошибка будет возникать из-за того, что точки с запятой не хватает в предыдущей строке.

Компилировать можно по-разному

Есть три способа компиляции файла: он может быть построен, откомпилирован либо повторно построен.

При *построении* программы (команда Build⇨Build) обновляются файлы OBJ для всех измененных исходных файлов. Когда все файлы OBJ обновлены, программа компонуется для создания нового выполняемого файла (EXE). Результатом такой компиляции будет либо набор сообщений о синтаксических ошибках, либо полностью рабочая программа.

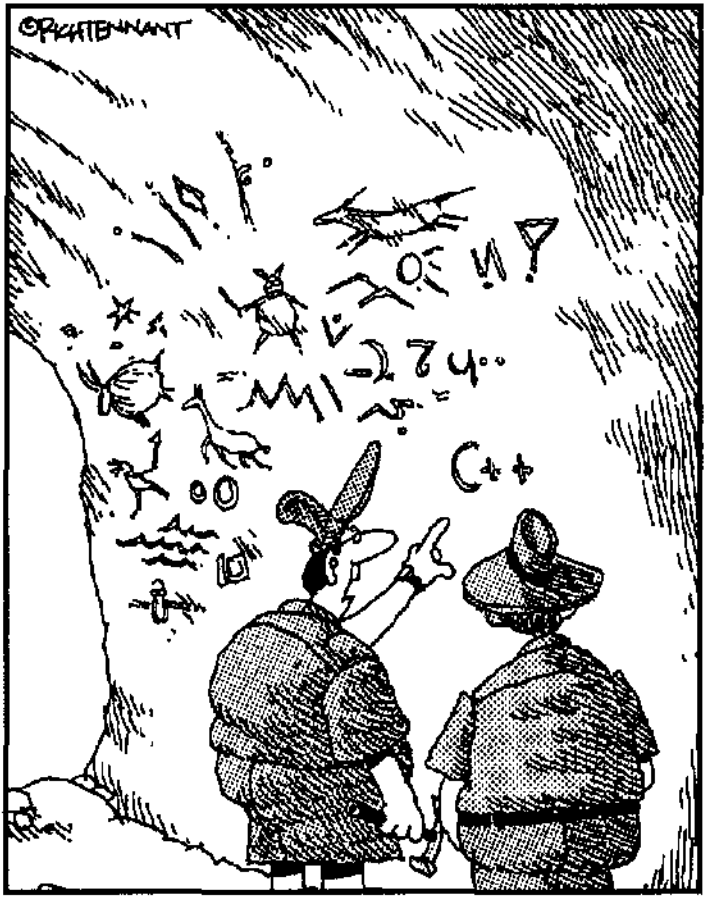
Если вы работаете над проектом и вносите измнения только в некоторые его части, используйте именно этот способ компиляции. При этом заново откомпилированы будут только измененные файлы и вам не нужно будет тратить массу времени и ждать, пока компилятор сделает бессмысленную работу и еще раз откомпилирует те файлы, которые вы не трогали. Поэтому используйте команду Build⇨Build, если вы внесли какие-то изменения в исходные файлы и вам нужен теперь новый файл EXE.

При обычной *компиляции* (команда Build⇨Compile) компилируется только тот файл, который открыт в данный момент для редактирования. Этой командой не создается рабочая программа. Используйте ее в том случае, если хотите проверить какой-либо исходный файл на наличие синтаксических ошибок. Если компиляция проходит успешно, создается объектный файл (OBJ).

При *повторном построении* (команда Build⇨Rebuild All) все файлы проекта компилируются и затем компонуются для создания выполняемой программы (EXE). Как и в случае с построением, результатом такой команды будет либо набор сообщений о синтаксических ошибках, либо полностью рабочая программа. Даже если вы только что построили программу, при повторном построении каждый отдельный файл компилируется заново. (Результатом такой операции будут обновленные объектные файлы всех исходных файлов плюс выполняемый файл программы, созданный путем компоновки всех объектных файлов.) Используйте эту команду в том случае, если хотите быть уверены, что *весь* проект был построен заново.

Часть II

Все, что вы хотели знать о C++, но о чем боялись спросить



Этим периодом можно датировать появление первых сумасшедших гениев.

В этой части...

С этой части начинается серьезное обучение. Вы получите фундаментальные знания о Visual C++, начиная с того, как укомплектовать программу переменными, операторами и указателями. Также вы познакомитесь с одним из наиболее важных инструментов создания программ — отладчиком.

Ни в коем случае не игнорируйте приводимые примеры, поскольку многое вещи познаются на практике. Пробуйте самостоятельно набрать коды программ и выполнить их на своем компьютере, чтобы быть уверенным, что вы правильно понимаете излагаемый здесь материал.

Сразу скажем, что не нужно пугаться этой части, поскольку сухость и серьезность рассматриваемых здесь вопросов компенсируется хорошим, добрым юмором.

Глава 7

Типы данных — это серьезно

В этой главе...

- Строгие и нестрогие языки программирования
- Объявление типов переменных
- Преобразование строк в числа
- Создание констант
- Работа со строками

Компьютерные программы занимаются обработкой различных данных, которые, в свою очередь, могут принадлежать различным типам. Например, в электронных таблицах данные могут быть представлены числами с плавающей запятой, в инвентаризационных системах — порядковыми записями, а в программах для рисования — автофигурами. Однако нужно понимать, что, хотя режиссеры Голливуда и создают мифы о всемогущих и всезнающих компьютерах, на самом деле это не так. Компьютер может работать только с данными строго определенных типов, причем предварительно нужно указать, данные какого именно типа будут ему передаваться. В этой главе вы познакомитесь с основными типами данных, особенно детально обсуждается один из самых важных, который обозначается словом `String`.

Строгие и нестрогие языки программирования

Строгие языки программирования, такие как C++, требуют, чтобы программисты заранее объявляли, данные каких типов они собираются использовать. Например, если вы хотите сохранить число, нужно заранее сообщить об этом компьютеру, чтобы он был готов принять от вас это число.

Строгие языки имеют ряд неоспоримых преимуществ. Например, если вы случайно попытаетесь запись о сотруднике интерпретировать как число, компилятор выдаст сообщение об ошибке. Это хорошо, поскольку без подобного контроля вы можете непреднамеренно уничтожить важную информацию. Предположим, запись о сотруднике занимает 8 байт памяти, а для числа отводится только 2 байта. Когда вы удаляете запись, освобождается 8 байт памяти. Если же вы используете эту запись как число и затем это число удалите, помимо числа будет удалено еще 6 байт памяти, в которых может содержаться очень важная информация. Возможно, потеря этой информации будет иметь для вас очень плохие последствия.

Помимо строгих языков программирования, есть также *нестрогие,* к числу которых относится, например, JavaScript. Они не требуют предварительного объявления типов данных и сами оценивают данные в тот момент, когда с ними сталкиваются. Например, если вы присвоите переменной текстовое значение `root`, языковой процессор сделает эту переменную текстовой. Нестрогие языки программирования могут быть проще в использовании, и некоторые программисты отдают им предпочтение, однако они не умеют предостерегать вас от ошибок, как это делают строгие языки.

Объявление переменных

В языке C++ необходимо заранее объявлять переменную и указывать тип используемых ею данных. Другими словами, нужно сразу указать компилятору, значения какого типа могут быть присвоены переменной. (Более подробно об использовании переменных речь идет в главе 8. Пока же рассматривайте переменную как нечто, способное сохранять информацию, например как ячейку электронной таблицы, которая, однако, имеет свое собственное имя.) Объявить тип переменной можно несколькими способами, но обычно это делается в момент ее создания.

Процесс создания переменной называется ее *определением*. Чтобы определить переменную, нужно просто указать используемый ею тип данных, а затем ее имя. Ниже приведено несколько примеров определения переменных. В следующей строке создается переменная `foo` типа `int` (integer, число):

```
int  too;
```

Далее создается переменная `bar` типа `char` (character, символ):

```
char bar;
```

И ниже сообщается о том, что функция `min` принимает два числа типа `int` в качестве значений параметров и возвращает другое число типа `int` в качестве результата. (Более подробно определение и использование функций рассматривается в главе 12.)

```
int min(int first, int second);
```

Обратите внимание, что *объявление* переменной отличается от ее *определения*. Когда вы *объявляете* тип данных, вы просто указываете, значения какого типа будет принимать эта переменная. При этом выделения памяти для объявляемой переменной не происходит. (Физически переменная не создается.) Когда вы что-то определяете, вы создаете это физически. В случае с переменными это различие не столь принципиально. Почти всегда при определении переменной указывается и ее тип. Хотя делать это можно и раздельно. Познакомившись со структурами (глава 9) и классами (глава 17), вы увидите, что для них определение и объявление — это два разных действия, каждое из которых выполняется отдельно.

Наиболее часто используемые типы данных

В языке C++ имеется набор заранее определенных типов данных, которые сразу можно использовать. Используя стандартные типы данных, можно создавать собственные, более сложные, типы данных, что описывается в главе 9. Вот три наиболее часто используемых типа данных.

- ✓ `double`. Число с плавающей запятой. (Не думайте, что это число с водой, в которой плавает запятая. Это всего лишь вещественное число с десятичной запятой, которая может "плавать" в зависимости от количества десятичных знаков в числе и не имеет точно зафиксированной позиции. Сюда относятся, например, такие числа, как 4,3444; 42,1; 3,14.) Переменные такого типа могут содержать значения в пределах от $\pm 1{,}7*10^{-308}$ до $\pm 1{,}7*10^{+308}$.
- ✓ `int`. Целое число. Значение может быть положительным, отрицательным либо нулевым. Например, к этому типу относятся значения 0, 1, 39, -42, но не 1,5.
- ✓ `String`. Любой текст, например Hello World.

Например, вам нужна переменная для хранения значения радиуса круга. Можете определить ее таким образом:

```
double dblRadius;
```

Этой строкой создается переменная dblRadius, которая может принимать значения типа double (числа с плавающей запятой). Для сохранения значения радиуса круга такая переменная вполне подойдет.

Перед созданием программы определите для себя, переменные каких типов вам потребуются для хранения всех тех данных, которые вы собираетесь использовать. Например, для сохранения числа, обозначающего количество людей, посетивших концерт, можно создать переменную типа int (целое число), поскольку, разумеется, количество людей не может быть дробным. (Правда, кто-то может прийти со своей "второй половиной", но это для него она "половина", а для всех остальных — отдельный человек.) Если же вам нужна переменная для сохранения значения радиуса круга, лучше использовать тип double, поскольку радиус вполне может обозначаться дробным числом. И если вам нужно сохранять текстовую информацию, создайте переменную типа String.

Реже используемые типы данных

Ниже дается описание некоторых других типов данных, которые тоже используются при создании программ, но не так часто, как приведенные выше.

- ✓ float. Вещественное число с плавающей запятой; однако переменные этого типа данных могут принимать значения из диапазона гораздо меньшего, чем переменные типа double. Для этого типа допустимые значения лежат в интервале от $\pm 3{,}4*10^{-38}$ до $\pm 3{,}4*10^{38}$.
- ✓ long. Это слово может быть добавлено перед словом int, чтобы обозначить, что для хранения одного значения должно быть выделено 32 бит памяти. В Visual C++ .NET такой объем памяти выделяется по умолчанию, поэтому в использовании этого слова нет необходимости.
- ✓ short. Это слово может быть добавлено перед словом int, чтобы обозначить, что для хранения одного значения должно быть выделено 16 бит памяти.
- ✓ singed. Это слово может быть добавлено перед словом int, чтобы обозначить, что принимаемые значения могут быть как позитивными, так и негативными. (В Visual C++ эта возможность предусмотрена по умолчанию.)
- ✓ unsigned. Это слово может быть добавлено перед словом int, чтобы обозначить, что все принимаемые значения будут только позитивными. Благодаря этому освобождается один дополнительный бит памяти для обозначения величины принимаемого значения. Так, если переменная типа int может принимать значения в диапазоне от -2 147 483 648 до +2 147 483 647, то слово unsigned определяет диапазон от 0 до 4 294 967 295.
- ✓ void. Тип не определяется. Используется для обозначения того, что функция не возвращает никакого результата. (Более подробно об этом речь идет в главе 12.)
- ✓ char. Символ. A, B или & — это все символы. Этот тип данных чаще всего используется в неуправляемых программах, в то время как в управляемых в основном применяется более гибкий тип String.

Обеспечение типовой безопасности

Языки машинного уровня не заботятся о корректном использовании типов данных. Для них данные — это всего лишь нечто, занимающее часть памяти. Вот почему программы, написанные на языках машинного уровня, могут так просто сохранять числа поверх записей о сотрудниках, символы поверх чисел, и вообще делать все, что вы им прикажете. (Многие компьютерные вирусы уничтожают данные именно таким способом.)

Тип, который отличается от всех остальных

Чтобы убедиться, что все объявленные типы используются корректно, Visual C++ проделывает большую работу. Во время компиляции исходного файла формируется так называемая *таблица имен.* Она содержит подробную информацию обо всех переменных и функциях, используемых в исходном файле. Каждый раз, когда переменная (или функция) как-то используется, компилятор находит ее описание в таблице имен, проверяет объявленный для нее тип данных и определяет, возможно ли в отношении переменной (или функции) предпринимать такие действия.

Компилятор имеет также набор строгих правил, определяющих, как правильно преобразовать значения одного типа к значениям другого. Например, если функции требуется для работы значение параметра типа `double` {более детально передача функции значений параметров описана в главе 12), а вы передаете ей значение типа `int`, компилятор сам сможет преобразовать целое число к числу с плавающей запятой. Если же он обнаружит несоответствие типов, для которых не определено правило взаимного преобразования, будет возвращено сообщение об ошибке.

При компиляции программы, состоящей из нескольких исходных файлов, ситуация усложняется. В одном исходном файле могут использоваться переменные или функции, объявленные в других файлах. Чтобы убедиться в правильности использования типов данных, компилятор теперь должен сравнивать переменные и функции из разных файлов. Строго говоря, этот шаг выполняется уже на этапе компоновки программы.

Проверка правильности использования типов данных (так называемый *внешний анализ)* осуществляется с использованием техники, которая называется *корректировкой имен.* Когда вы объявляете переменную, функцию или любой другой элемент, компилятор несколько изменяет присваиваемое вами имя (это имя также называют *внешним).* Внешнее имя содержит информацию о типе элемента. Можно сказать, например, что компилятор добавляет к исходному имени букву i, чтобы обозначить, что элемент имеет тип `int`, и буквы `cp`, чтобы обозначить, что элемент имеет тип `char`. Далее предположим, что а одном из исходных файлов вы создали такую функцию:

```
int NumberOfSongs();
```

В другом исходном файле вы набрали такой код:

```
char *MyText;
MyText = NumberOfSongs();
```

Компилятор преобразует имя `NumberOfSongs` к имени `iNumberOfSongs`, а имя `MyText` - к имени `cpMyText`. Сами вы никогда не увидите внутренних имен (разве что посмотрите на листинг кодов языка ассемблера), но компоновщик видит их очень хорошо.

Для него, например, последняя строка кодов выглядит так;

```
cpMyText = iNumberOfSongs();
```

Сравнивая скорректированные имена, компоновщик фиксирует несовпадение типов (поскольку символ и число - это разные данные) и выдает сообщение об ошибке.

Однако C++ заботится о правильном использовании типов данных. Если вы объявите для переменной один тип данных, а затем попробуете использовать ее как переменную другого типа, компилятор выдаст сообщение об ошибке. Эта возможность называется *обеспечением типовой безопасности* и является преимуществом языка C++, поскольку таким образом автоматически выявляются наиболее распространенные ошибки. (Это действительно лучше, чем игнорирование компилятором неправильного использования типов данных, за исключением разве что тех случаев, когда вы создасте самоуничтожающуюся программу.)

Приведенный ниже код будет работать корректно.

```
//Создание переменной типа int
int nSize;
//Присвоение переменной значения 7
nSize = 7;
```

Но, если вы наберете следующий код, компилятор выдаст сообщение об ошибке, поскольку переменная типа `int` не может принимать текстовые значения:

```
//Попытка присвоить переменной текстовое значение
nSize = "Hello World";
```

Иногда компилятор автоматически преобразует значения одного типа данных к значениям другого типа. В приведенном ниже примере число с плавающей запятой автоматически преобразуется к целому числу:

```
int nSize;
nSize = 6.3;
```

После выполнения этого кода переменной `nSize` присваивается значение 6. (Компилятор автоматически округляет число. Все цифры, расположенные справа от десятичной запятой, просто отбрасываются.) Однако при этом компилятор отображает на экране предупреждение о том, что будет выполнено преобразование чисел. Хорошие и по-настоящему качественные программы должны компилироваться без предупреждений и ошибок. Чтобы избежать подобных предупреждений, можно явно указать компилятору на необходимость преобразования одного типа данных к другому. Для этого перед переменной, значение которой должно быть преобразовано, в круглых скобках укажите требующийся тип данных. Например:

```
int nSize;
double BigSize;
nSize = (int) BigZise;
```

В этом случае компилятор без лишних предупреждений преобразует значение типа `double` к значению типа `int`. Это означает, что дробная часть значения переменной `BigZise` будет просто усечена. Например, если эта переменная имеет значение, равное числу 3,141592653, переменной `nSize` будет присвоено значение, равное числу 3.

Функции преобразования типов

В стандартном C++ типы данных являются просто структурами, которые требуют для себя некоторого количества памяти. С другой стороны, в среде .NET типы данных обладают еще и встроенной функциональностью. Технически они являются объектами. (Более подробно объекты описаны в главе 17.) Как и другие объекты, типы данных имеют встроенные функции, наиболее полезными из которых являются функции преобразования одних типов в другие. В табл. 7.1 приведен список функций преобразования типов для основных типов данных.

Таблица 7.1. Функции преобразования типов данных

Тип	Функция	Выполняемое действие
String	ToDouble()	Преобразует тип string к типу double
String	ToInt32()	Преобразует тип String к типу int
double	ToString()	Преобразует тип double к типу String
double	Parse()	Преобразует тип String к типу double
int	ToString()	Преобразует тип int к типу String
int	Parse()	Преобразует тип String к типу int

Предположим, например, что вам нужно отобразить на экране число. Для этого предварительно необходимо преобразовать его в строку:

```
int nNumber = 3;
Console::WriteLine(nNumber.ToString());
```

Теперь предположим, что у вас есть текстовое значение (String) и из него нужно сделать число (double). Делается это так:

```
String *szNumber = S"3.14"
double dNumber;
dNumber = Double::Parse(szNumber);
```

Вы, наверное, обратили внимание на использование звездочки (*) перед именем текстовой переменной. Она там нужна, поскольку при работе со строками приходится использовать указатели. Более подробно указатели описаны в главе 13. Пока же просто примите к сведению, что при объявлении текстовых переменных перед их именем нужно добавлять звездочку (*). И каждый раз при вызове встроенных для типа String функций вместо точки (.) используйте символы ->.

Если вас интересуют другие подробности приведенного выше кода, прочитайте этот абзац. Буква S перед значением "3.14" указывает компилятору, что это должна быть строка, предусмотренная для использования в среде .NET. На самом деле можно создавать текстовые значения самых разных видов, но в данном случае создается именно такое значение, которое для среды .NET подходит наилучшим образом. Использование двух двоеточий (: :) в фрагменте Double : : Parse является специальным синтаксическим приемом, позволяющим применять методы (такие, как Parse) объектных типов (таких, как double) без предварительного создания самих объектов. В будущем вы сможете по достоинству оценить эту возможность языка C++.

Константы — rііо, что никогда не меняется

Иногда при написании программы требуется многократно использовать одно и то же число или одну и ту же строку. Например, если вы знакомы с математикой, то знаете, что число *п* всегда равно 3,141592.... Если вы составляете программу, вычисляющую ваш гороскоп, то используемая в вычислениях дата вашего рождения также будет оставаться неизменной.

В подобных случаях становится возможным создание констант. Сами *константы* — это просто названия элементов, которые в процессе выполнения программы никогда не изменяются. Чтобы сделать элемент константой, начните его объявление со слова `const`. Например, следующим кодом создастся константа PI, значение которой постоянно будет равно числу 3,141592:

```
const double PI = 3.141592;
```

Использование констант значительно облегчает чтение кодов ваших программ, поскольку обычные числа можно заменить объясняющими их именами. Кроме того, это удобно, если одно и то же значение должно быть изменено во всей программе. Для этого нужно просто определять новое значение для соответствующей константы. Например, если вы вдруг решите приспособить написанную вами программу к некоторой виртуальной реальности, где пространство и время деформированы и число *п* равно, скажем, числу 7, вам нужно будет всего лишь изменить значение соответствующей константы. При этом все вычисления, выполняемые вашей программой с участием числа *п,* автоматически будут использовать новое указанное значение, что избавляет вас от необходимости самостоятельно просматривать коды всей программы и вносить нужные коррективы.

В кодах программы использовать константы можно везде, где может быть использован элемент того же типа, что и сама константа. Например, если у вас есть константа, представляющая число с плавающей запятой, вы можете применять ее везде, где будет уместно использование числа с плавающей запятой. Так. константа PI может быть умножена на диаметр круга, в результате чего будет вычислена длина окружности.

Использование констант в кодах программы

Ниже приведен код программы, в которой используются константы и выполняется преобразование типов данных. В процессе ее выполнения пользователю предлагается указать радиус, в соответствии с которым вычисляется площадь круга. Полученный результат отображается на экране. Не поленитесь и создайте такую же программу. Для этого создайте новую программу .NET, наберите приведенный здесь код и запустите программу на выполнение.

```
//CircleArea

//Вычисление площади по значению радиуса

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  double fltRadius;
  double fltArea;
```

```
  const double PI = 3.141592;

  //Отображение вопроса о значении радиуса
  Console::WriteLine(S"Укажите радиус круга");

  //Получение ответа и преобразование его к числу с
  //плавающей запятой
  fltRadius = Double::Parse(Console::ReadLine());
  //Вычисление площади круга
  fltArea = PI*fltRadius*fltRadius;

  //Отображение результата на экране
  //Обратите внимание, что для этого число должно быть
  //предварительно преобразовано в строку
  Console::WriteLine(S"Площадь круга составляет {0}
                          единиц.",Area.ToString());

  //Ожидание, пока пользователь не остановит
  //выполнение программы
  Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                       остановить выполнение программы");
     Console::ReadLine();

  return 0;
}
```

Константы и борьба с ошибками

Еще одно преимущество использования констант: если вы случайно попытаетесь изменить их значения в процессе выполнения программы, компилятор заметит это и выдаст сообщение об ошибке. Предположим, например, что где-то в кодах программы вы набрали такую строку:

```
PI = 15;
```

Как только компилятор увидит этот код, он сразу же выдаст сообщение об ошибке, в котором будет сказано, что изменять значение константы нельзя. Своевременное обнаружение и устранение таких, казалось бы, мелких ошибок позволяет избежать возникновения в дальнейшем более серьезных проблем.

Строки как одни из наиболее важных типов данных

Строки (тип `String`) являются основными данными почти для всех создаваемых программ. По крайней мере почти ни одна программа не может без них обойтись. Чтобы эффективно их использовать, нужно владеть некоторым количеством не очень сложных приемов. Основные их них описаны в табл. 7.2.

Таблица 7.2. Способы обработки текстовой информации

Задача	Решение
Комбинирование двух строк	`sResult = String::Concat (st1, st2);`
Отображение на экране текста со значениями	`Console::WriteLine(S"Значение 1: {0} Значение 2 {1}", зн1, зн2);`
Сравнивание двух строк	`cResult = st1->Equals(st2);`
Определение позиции, начиная с которой одна строка входит во вторую	`nOffset = st1->IndexOf(st2);`
Получение n символов начиная с m-й позиции	`mResult = st1->Substring(m,n);`
Определение длины строки	`nLength = st->Length;`
Удаление лишних пробелов в начале и в конце строки	`Result = st1->Trim();`

В кодах программ, которые приведены на страницах этой книги, вы будете часто сталкиваться со строками. Ниже дан код небольшой программы, на примере которого вы можете увидеть в действии различные способы обработки текстовой информации,

```
//Strings101
//Некоторые приемы работы со строками

#include "stdiafx.h"

#using <mscorlib.dll>

using namespace System;

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  String *pszString;
  int nNumber;
  double fltNumber;

  //Комбинирование двух строк
  pszString = String::Concat(S"Майкл", S" Джексон");
  Console::WriteLine(pszString);
```

```
    //Преобразование типов и комбинирование строк
    nNumber = 3;
    pszString = String::Concat(S"Преобразованное ",
                nNumber.ToString());
    Console::WriteLine(pszString);


    //Преобразование строки к типу double
    Console::WriteLine(S"Введите число");
    fltNumber = Double::Parse(Console::ReadLine());
    //Добавление к этому значению числа
    fltNumber = fltNumber + 30.5;
    //Отображение полученного результата на экране
    Console::WriteLine(String::Concat(S"Результат: ",
                fltNumber.ToString()));


    //Объединение строк и их отображение
    Console::WriteLine(String::Concat(S"Один ",
                S"Два ", S"Три"));


    //Отображение с использованием символов форматирования
    Console::WriteLine(S"Строка: {0}\nЧисло: {1}",
                S"моя строка", nNumber.ToString());


    //Удаление лишних пробелов в начале и в конце строки
    pszString = "   Hello World     ";
    Console::WriteLine(S"Строка с пробелами: {0}\n
                и без пробелов: {1}", pszString,
                pszString->Trim());


    //Определение символа середины строки
    Console::WriteLine(S"Посередине строки {0}
         расположен символ \"{1}\"", pszString,
         pszString->Substring(pszString->Length/2, 1));


    //Ожидание, пока пользователь не остановит
    //выполнение программы
    Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                           остановить выполнение программы");
    Console::ReadLine();
    return 0;
}
```

Глава 8

Использование переменных

В этой главе...

- Имена переменных
- Определение и инициализация переменных
- Соглашения о присвоении имен переменным

Программы получают, обрабатывают и возвращают данные. Если нужно сохранить какое-то значение или результат каких-то вычислений, используются переменные. *Переменная* — это имя, которым обозначается какой-либо фрагмент информации. Переменные могут быть использованы для хранения самых разнообразных видов данных, начиная от баллов, набранных в компьютерной игре, и заканчивая биржевыми показателями.

Имена переменных часто несут в себе какую-то информацию о принимаемых значениях. Например, по имени `nRadius` можно догадаться, что переменная используется для хранения (и представления) радиуса круга. С другой стороны, переменным иногда присваиваются не очень удачные имена, по которым трудно сделать вывод об их назначении. Например, имя C4P0 может ассоциироваться с серийным номером, паролем и вообще с чем угодно.

Каждый раз, когда необходимо получить доступ к хранимой в переменной информации, ссылаются на имя переменной. Поскольку C++ является строгим языком программирования, перед тем как приступить к использованию переменной, нужно объявить ее тип данных.

В этой главе рассматриваются вопросы, связанные с именованием переменных, их определением и инициализацией.

Именование переменных

При выборе имени для ребенка родители сталкиваются с множеством ограничений. Допустим, например, что они сами помешаны на программировании и надеются, что, когда их чадо вырастет, непременно станет великим программистом. Тогда вместо имени Джон они могли бы дать ему имя wwwДжон. ИЛИ вместо имени Джессика назвали бы девочку double. Но к сожалению, даже если соответствующее административное учреждение (и то не каждое) зарегистрирует такое имя, у ребенка будут проблемы со сверстниками, тяжелые школьные годы, и он никогда не простит своих родителей за такой "подарочек".

Выбрать имя для переменной намного проще. Ей безразлично, как вы ее назовете, поэтому смело давайте ей любое понравившееся имя. Правда, и здесь существуют некоторые ограничения. Впрочем, они вполне разумны и их легко запомнить. Итак, имя переменной:

- ✓ не должно начинаться с цифры;
- ✓ не должно содержать пробелов;
- ✓ может состоять только из букв, цифр и символов подчеркивания (_). Нельзя использовать специальные символы, такие как точка, запятая, точка с запятой, кавычки и т.п.;
- ✓ не должно совпадать с названиями библиотечных функций;
- ✓ не должно совпадать с зарезервированными ключевыми словами.

В табл. 8.1 приведен перечень ключевых слов Visual C++, которые являются командами языка C++. Дочитав книгу до конца, вы будете знать, как пользоваться большинством из этих ключевых слов. Слова, начинающиеся с двух символов подчеркивания (___), являются специальным расширением Visual C++, которое призвано упростить процесс создания программ для персональных компьютеров. Просмотрите таблицу и не присваивайте переменным таких же имен.

Таблица 8.1. Ключевые слова C++

__abstract	__hook	__multiple_inheritance
__alignof	__identifier	__nogc
__asm	__if_exists	__noop
__assume	__if_not_exists	__pin
__based	__inline	__property
__box	__int8	__raise
__cdecl	__int16	__sealed
__declspec	__int32	__single_inheritance
__delegate	__int64	__stdcall
__event	__interfase	__super
__except	__leave	__try_cast
__fastcall	__m64	__try/__except
__finally	__m128	__try/__finally
__forceinline	__m128d	__unhook
__gc	__m128i	__uuidof
__value	float	signed
__virtual_inheritance	for	sizeof
__w64	friend	static
__wchar_t	goto	static_cast
bool	if	struct
break	inline	switch
case	int	template
catch	long	this
char	mutable	thread
class	naked	throw
const	namespase	true
const_cast	new	try
continue	noinline	typedef
default	noreturn	typeid

delete	nothrow	typename
deprecated	novtable	union
dllexport	operator	unsigned
dllimport	private	using
do	property	uuid
double	protected	virtual
dynamic_cast	public	void
else	register	voletile
enum	reinterpret_cast	wchar_t
explicit	return	while
extern	selectany	
false	short	

Вот примеры допустимых имен переменных: `way_cool`, `RigthOn`, `Bits32`. А такие имена присваивать переменным нельзя: `case` (совпадает с ключевым словом), `52PickUp` (начинается с цифры), A Louse (содержит пробел), `+-v` (включает недопустимые символы).

В именах переменных строчные и прописные буквы воспринимаются как разные. Например, имена `bars`, `Ears`, `bArs` и `BARS` обозначают разные переменные.

Определение переменных

Перед тем как использовать переменную, ее нужно определить. Для этого просто укажите тип принимаемых значений и ее имя. Вот, например, определение нескольких переменных:

```
int Counter;
double OrNothing;
long Johns;
```

Если объявляемые переменные имеют один и тот же тип данных, объявить их можно в одной строке. Например, если нужно объявить переменные `First`, `Second` и `Third`, каждая из которых должна принимать значения типа `float`, можете сделать это так:

```
float First;
float Second;
float Third;
```

Или так:

```
float First, Second, Third;
```

Инициализация переменных

Одновременно с объявлением переменные можно *инициализировать*. Этот громоздкий термин обозначает всего лишь присвоение переменным исходных значений. Чтобы сделать это, наберите после имени переменной знак равенства (=) и затем укажите нужное значение.

Например:

```
int Counter = 3;
double OrNothing = 3.5;
long Johns = 32700;
```

Вот и все. Ничего сложного, не так ли?

Как сделать имя информативным

Вы, конечно, можете присваивать переменным самые разнообразные имена (за небольшими исключениями, они упоминались выше), однако обычно программисты стараются придерживаться некоторых общих соглашений, цель которых — упростить чтение и понимание кодов программ. Основным из них является так называемое *венгерское обозначение*, придуманное, как говорят, неким венгром, работающим на компанию Microsoft. Его идея состоит в том, чтобы начинать имена со специальных префиксов из нескольких букв, обозначающих тип переменной. В этой книге при назначении имен переменным используется одна из версий венгерского обозначения, которая, на наш взгляд, является самой простой и удобной.

Например, чтобы обозначить, что переменная имеет тип integer (`int`), начинаем ее имя с буквы n:

```
int nRadius;
int nCount = 0;
```

В табл. 8.2 приведен список префиксов, используемых в книге. Напомним, что, хотя использование таких префиксов является хорошей практикой, позволяющей помимо прочего допускать меньше ошибок, вы всегда можете отказаться от них и присваивать переменным такие имена, которые вам больше нравятся.

Таблица 8.2. Использование префиксов

Тип данных	Префикс	Пример
Integer	n	nCount
Double	dbl	dblRadius
String	psz	pszName
Boolean	f	fFinished
object	o	oLine
object pointer	po	poCircle
array	a	aShapes
member	m_	m_nShapes

Некоторые типы данных, упомянутые в этой таблице, вы встречаете, наверное, впервые. Например, массивы (array) будут описаны в главе 14, а объекты (object) — в главе 17.

Глава 9

Структуры данных

В этой главе...

- Объявление и использование структур
- Комбинирование структур
- Добавление структур к приложению

В предыдущей главе рассматривались простые типы данных и способы использования переменных для хранения информации. Но предположим, что необходимо сохранить более сложную информацию, которая не может быть представлена только лишь одной переменной какого-то определенного типа данных. Например, информацию о сотруднике, которая включает его имя, адрес и телефонный номер. Если рассматривать только простые типы данных, вы *должны* использовать для этой цели три различные переменные: одну для представления имени, вторую для адреса и третью для телефонного номера.

Такой подход нельзя назвать самым удачным, поскольку в реальном мире почти каждое явление описывается целым набором разных значений. Представим, например, что вам необходимо отобразить информацию о сотруднике. Несмотря на то что эта информация состоит из разных частей (имя, адрес, телефон и т.п.), вы все равно представляете ее как нечто целое. По крайней мере намного предпочтительнее и удобнее сказать: "отобрази информацию о сотруднике", чем: "отобрази имя сотрудника, его адрес, телефон и т.д.".

В этой главе описано, как объединить несколько отдельных переменных в нечто целое, называемое *структурой*. Группируя связанные по смыслу элементы в общую структуру, вы значительно упрощаете чтение и понимание кодов программ. Возможно, вы будете использовать структуры во всех своих будущих программах, по крайней мере в этой книге вы будете часто с ними встречаться. В главе 17 рассматривается, как обычные структуры могут быть преобразованы в классы — самые главные компоненты, из которых строятся объектно-ориентированные программы.

Объявление структур

Объявление структур во многом похоже на объявление обычных типов данных (см. главу 7). Чтобы объявить структуру, наберите слово `class`, затем укажите название структуры. Далее в фигурных скобках наберите `public:` и объявите переменные, из которых будет состоять структура.

Вот пример объявления структуры:

```
class CircleInfo
{
public:
   double dblRadius;
   double dblArea;
};
```

В данном случае структура названа именем `CircleInfo` (Информация о круге). Она состоит из переменной `dblRadius` типа double, в которой хранится значение радиуса круга, и переменной `dblArea` (также типа double), в которой хранится значение площади круга.

Чтобы определить переменную, которая будет представлять структуру, просто наберите название структуры и затем название переменной.

```
//Создание переменной, представляющей структуру
CircleInfo oCircle;
```

Что в действительности означает определение переменной, представляющей структуру? Приведенной выше строкой была создана переменная oCircle типа CircleInfo. Это означает, что переменная oCircle содержит две другие переменные — dblRadius и dblArea, поскольку эти переменные определены для структуры CircleInfo. Можно сказать, что одна переменная (oCircle) типа CircleInfo содержит несколько значений (dblRadius и dblArea).

Объединяя различные переменные в отдельные структуры, можно логически правильно организовать используемую в программе информацию. Далее вы узнаете, как используются переменные, представляющие структуру.

Разница между объявлением и определением

В C++ *объявление* и *определение* являются техническими терминами. Их смысл несколько различается, но часто их используют как взаимозаменяемые (хотя по сути это неправильно).

Обьявляя структуру, вы сообщаете компьютеру, из чего эта структура будет состоять. Например:

```
class CircleInfo
{
public:
    double dblRadius;
    double dblArea;
};
```

При этом память для структуры не выделяется.

Определяя переменную, вы даете компьютеру указание создать эту переменную. В этом случае для создаваемой переменной выделяется память:

```
CircleInfo oCircle;
```

Использование этих загадочных структур

После того как структура создана, чтобы получить доступ к хранимой в ней информации (к значениям переменных, из которых состоит структура), наберите имя переменной, представляющей структуру, затем точку (.) и затем имя переменной, значение которой нужно получить.

Например, чтобы отобразить значение радиуса круга, наберите такой код:

```
//Создание переменней, представляющей структуру
CircleInfo oCircle;
//Отображение части информации, хранимой в структуре,
//которая обозначается именем dblRadius
Console::WriteLine(oCircle.dblRadius.ToString());
```

Как видите, выражение oCircle.dblRadius выступает здесь в роли обычной переменной. Использование элементов, входящих в структуру, по существу, ничем не отличается от использования стандартных переменных, представляющих только одно значение.

использование одних структур для создания других

Можно объединить несколько структур для создания одной большой структуры. Структуры, входящие в состав других структур, называются *вложенными*.

Прежде чем использовать структуру для создания другой структуры, ее нужно объявить. Например, перед тем как создать структуру MegaCircle, состоящую из структур CircleInfo,нужно объявить структуру CircleInfo.Впрочем, это вполне логично. Нужно просто не запутаться, что и в какой последовательности должно быть объявлено. Если же вы нарушите естественный порядок. Visual C++ .NET выдаст сообщение об ошибке.

Вот пример объявления структуры, состоящей из двух структур C i r c l e l n f o :

```
class MegaCircle
{
public:
   CircleInfo oTopCircle;
   CircleInfo oBottomCircle;
};
```

Структуры на практике

Ниже приведен код программы CircleArea, который был немного изменен в связи с использованием структуры для представления данных.

```
//CircleArea3
//Вычисление площади по значению радиуса

//Использование структуры для представления данных

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;
//Объявление структуры для представления информации
//о параметрах круга

class CircleInfo
{
public:
   double dblRadius;
   double dblArea;
};

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  const double PI = 3.141592;
  CircleInfo oCircle;

  //Отображение вопроса о значении радиуса
  Console::WriteLine(S"Укажите радиус круга");

  //Получение ответа и преобразование его к числу с
  //плавающей запятой
```

```
    oCircle.dblRadius = Double::Parse(Console::ReadLine());

    //Вычисление площади круга
    oCircle.dblArea = PI*oCircle.dblRadius*oCircle.dblRadius;

    //Отображение результата на экране
    //Обратите внимание, что для этого число должно
    //быть предварительно прообразовано в строку
    Console::WriteLine(S"Площадь круга составляет {0}
                        единиц,", oCircle.dblArea.ToString());

    //Ожидание, пока пользователь не остановит
    //выполнение программы
    Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                              остановить выполнение программы");
    Console::ReadLine();

    return 0;
}
```

Вы могли обратить внимание на несколько интересных моментов в этой программе. Так, структура `CircleInfo` была объявлена перед функцией `main` (это функция, с которой начинается выполнение программы). Именно так и нужно делать, поскольку структура должна быть объявлена до того, как вы начнете ее использовать. В данном случае функция `main` использует структуру `CircleInfo`. поэтому она и должна быть объявлена перед кодами функции `main`. Это общий принцип для всех программ на C++. Сначала объявляется класс. Определяются все его функции. И только после этого класс можно использовать.

Поначалу вам может показаться это несколько странным, поскольку более простые и, по сути, служебные элементы описываются раньше, чем те элементы, в которых они используются. Но по мере приобретения опыта программирования на языке C++ вы к этому привыкнете. Пока же для удобства, если хотите, можете приступить к чтению кодов с того места, где начинается выполнение программы.

Кое-что о структурах и классах

В этой книге при создании структур будет использоваться ключевое слово `class`. Это же слово используется и при создании классов. (О них речь идет в главе 17.)

В действительности в языке C++ есть два слова, которые могут быть использованы для объявления структур (или классов): `class` и `struct`. Использование слова `struct` почти эквивалентно использованию слова `class`, за исключением лишь того, что в этом случае вам не обязательно добавлять слово `public:` перед описанием переменных, входящих в структуру. Некоторые предпочитают использовать слово `struct` при объявлении структур и слово `class` при объявлении классов (которые представляют собой те же структуры, содержащие в себе не только переменные, но и функции). Обратите внимание, что для объявления классов также можно использовать слово `struct`.

Когда вы поближе познакомитесь с классами (начиная с главы 17), вы узнаете, что означает слово `public`, а также слова `private` и `protected`. Если вы применяете слово `struct`, по умолчанию используется слово `public`, а если `class` - слово `private`. Наш совет: чтобы не путаться, постоянно используйте слово `class` при объявлении классов и структур.

Почему два ключевых слова выполняют практически одни и те же функции? Слово `struct` использовалось еще в языке C и было оставлено для того, чтобы программы, написанные на языке C, можно было компилировать с помощью компилятора C++. В языке C++ этому слову добавили немного функциональности, и теперь оно почти идентично слову `class`. Но, поскольку вы являетесь программистами на C++, используйте при написании своих программ только слово `class`,

Глава 10

Выразите свои желания

В этой главе...

- Чем являются выражения
- Знакомство с основными операторами
- Логические выражения
- Оценка значений
- Сравнение значений
- Использование математических функций

Программы обрабатывают данные, и одним из видов такой обработки является выполнение различных вычислений. Указание провести вычисления (которое записывается в виде формулы), в языке Visual C++ .NET называется *выражением*. Если вы знакомы с электронными таблицами (например, Excel), то наверняка уже имеете опыт использования выражений: каждый раз, набирая формулы в ячейках, вы набирали выражения.

Выражения применяются для получения новых данных исходя из уже имеющихся. Например, выражения можно использовать для вычисления площади круга, объема воздушного шара, веса танка или проведения более сложных вычислений, скажем, подведения итогов голосования.

Можно ли "выражаться"?

Использование выражений просто необходимо при создании серьезных программ. Если вы читаете главы по порядку, то уже сталкивались с выражениями, вычисляющими площадь квадрата и площадь круга. Например, в следующей строке, взятой из программы CircleArea3 (глава 9), вычисляется площадь круга исходя из информации о значении его радиуса:

```
oCircle.dblArea = PI * oCircle.dblRadius * oCircle.dblRadius;
```

Выражения также используются для того, чтобы определить, выполняется ли определенное условие. Например, если нужно определить, достигнут ли лимит по кредитной карточке, следует написать выражение, которое будет сравнивать значение лимита с остатком денег на счете.

Если вы хотите создать программу, способную на какие-то реальные действия, вам обязательно придется прибегнуть к помощи выражений. Посмотрите на выражения, приведенные ниже:

```
2 + 2
3.14159 * Radius * Radius
ОстатокНаСчете < Лимит
```

Как видите, без помощи выражений вы не сможете даже сложить двух чисел!

Простые операторы

В табл. 10.1 приведен список из пяти операторов, наиболее часто используемых при построении выражений. *Оператором* называется математический символ, указывающий, какое действие необходимо произвести для получения результата. Например, в выражении 4+5 оператором будет знак +.

Таблица 10.1. Математические операторы

Оператор	Пример	Выполняемое действие
*	foo * bar	Умножает одно число на другое. Например, 6*3 будет 18
/	foo / bar	Делит одно число на другое. Например, 6/3 будет 2
+	foo + bar	Суммирует два числа. Например, 6+3 будет 9
-	foo - bar	Отнимает от одного числа другое. Например, 6–3 будет 3
%	foo % bar	Возвращает остаток от деления двух чисел. Например, 10%3 даст число 1, поскольку 10/3 будет 3 и остаток 1

Вычисление остатка от деления часто используется для того, чтобы не позволить значениям выходить из заданного диапазона. Например предположим, что вы создаете программу, которая перемещает фигурку космического корабля вдоль нижней части экрана (как в игре Space Invaders, если вы ее видели). Для того чтобы корабль, залетая за правую часть экрана, появлялся с левой стороны, используйте оператор вычисления остатка от деления. Допустим, экран имеет в ширину 10 единиц и pos — это позиция фигурки корабля, тогда результат операции pos % 10 всегда будет принадлежать диапазону от 0 до 9, независимо от того, какое значение имеет переменная pos. Теперь, когда корабль достигает позиции 9 (крайняя справа), к ее значению прибавляется число 1, чтобы перейти в следующую позицию (корабль пропадает с экрана), далее позиция вычисляется как результат выражения pos % 10, получаем число 0, и корабль отображается в позиции 0 (крайняя слева).

Более сложные операторы

Операторы, показанные в табл. 10.2, несколько сложнее, чем операторы, приведенные в табл. 10.1. На данном этапе вы, наверное, могли бы обойтись и без них. Но по мере приобретения опыта создания программ вы оцените эти операторы по достоинству и с удовольствием будете их использовать. Некоторые из этих "сложных" операторов рассмотрены ниже в отдельных разделах этой главы. (Вы, наверное, подумаете: "Неужели они настолько сложные, что для их описания нужно выделять отдельные разделы?!".)

Таблица 10.2. Операторы увеличения, уменьшения и сдвига

Оператор	Пример	Выполняемое действие
++	foo++, ++foo	Увеличение. Добавляет число 1 к значению элемента. Например, если переменная nAge имеет значение 2, то пхле выполнения операции nAge++ она будет иметь значение 3. (Кстати, свое название язык C++ получил благодаря этому оператору.)
--	foo--, --foo	Уменьшение. Его действие противоположно действию оператора увеличения. Например, если переменная a имеет значение 2, после выполнения операции a-- она будет иметь значение 1
>>	foo >> bar	Сдвиг разряда вправо. Выражение foo >> bar возвращает тот же результат, что и выражение foo/2bar. Более подробно этот оператор описан ниже в главе
<<	foo << bar	Сдвиг разряда влево. Его действие противоположно действию оператора >>. Более подробную информацию вы найдете далее в главе

Оператор ++

Этот оператор не так прост, как может вначале показаться, поскольку степень увеличения зависит от типа переменной, значение которой увеличивается. Например, если у вас есть указатель на переменную foo и значение переменной foo занимает четыре байта, то оператор увеличения, примененный к указателю, увеличит его значение на число 4, поскольку именно в этом случае указатель будет ссылаться на следующее значение после значения переменной foo. Сложновато, не так ли? Более детальную информацию об указателях вы можете найти в главе 13.

Есть два способа использования оператора ++. Его можно набрать или перед именем переменной: ++bar, или после: bar++.

Набрав ++bar, вы даете указание вначале увеличить значение переменной bar, а затем его использовать. Рассмотрим это на таком примере:

```
ir.t bar = 1;
Console::WriteLine(++bar.ToString());
```

В этом случае значение переменной bar будет увеличено на число 1, и на экране будет отображено число 2 (новое значение переменной bar).

Напротив, если вы наберете bar++, значение этой переменной вначале будет обработано, а затем увеличено. Например:

```
int bar = 1;
cout << bar++;
```

В этом случае значение переменной bar будет увеличено на единицу, но на экране будет отображено се старое значение (число !), поскольку увеличение происходит после отображения значения переменной. (В первом примере для отображения значения использован управляемый коп, а во втором — старый неуправляемый код C++.)

Оператор ++ часто используется в циклах для отсчета количества итераций. (Вам это непонятно? Не переживайте. Эти вопросы подробно обсуждаются в главе 11.)

Оператор -- работает по тому же принципу, что и ++, с той лишь разницей, что он уменьшает значения переменных на единицу. (Ну хорошо, не всегда на единицу. В некоторых случаях, например при работе с указателями, значение может уменьшаться более чем на единицу. Но в основном значение уменьшается именно на число 1. Уф!)

Оператор »

Оператор сдвига очень полезен при работе с двоичными числами. Вот некоторые примеры его использования:

16 >> 1 возвращает число 8;
16 >> 2 возвращает число 4;
16 >> 3 возвращает число 2;
15 >> 1 возвращает число 7;
15 >> 2 возвращает число 3.

Для получения результата первое число представляется в двоичном виде, а затем все биты сдвигаются вправо на количество позиций, указанное вторым числом. Обратите внимание, что при сдвиге битов вправо число уменьшается.

Для внесения ясности рассмотрим действие оператора сдвига подробнее. Число 16, например, в двоичном виде выглядит так:

1 0 0 0 0

Если сдвинуть все биты вправо на одну позицию, получим число

0 1 0 0 0

В десятичной системе это число соответствует числу 8, поэтому выражение 16 >> 1 даст число 8.

Рассмотрим другой пример. Число 15 в двоичной системе счисления выглядит так:

0 1 1 1 1

Поэтому результатом операции 15 >> 2 будет число

0 0 0 1 1

В десятичной системе ему соответствует число 3.

Оператор «

Ниже приведены примеры использования оператора <<.

16 << 1 возвращает число 32;
16 << 2 возвращает число 60.

Если получаемое число превышает максимально допустимое значение переменной, сдвигаемые влево биты обрезаются. Например, если для значения переменной зарезервировано только 8 бит памяти и вы сдвигаете исходное значение на 8 бит влево, то в результате получите число 0 (нуль). Это произойдет потому, что все биты исходного значения были сдвинуты за пределы переменной и обрезаны.

Обратите внимание, что оператор << выглядит точно так же, как символы <<, используемые вместе с командой cout. (Напомним, что cout является неуправляемым оператором языка C++, который используется для отображения данных на экране.) Если вы попытаетесь использовать оператор cout для отображения результата выполнения операции сдвига влево, его символы << будут иметь приоритет перед такими же символами оператора сдвига, из-за чего будет отображен не тот результат, который вам нужен (дело в том, что выражения обрабатываются слева направо и команда cout << будет обработана первой).

Поэтому, если вам нужно отобразить результат выполнения операции сдвига влево, возьмите это выражение в скобки:

```
cout « (16 << 2) << endl;
```

Истина и ложь в логических выражениях

Все рассмотренные ранее операторы используются в выражениях для вычисления какого-то определенного результата. Например вы видели, как используется оператор умножения для вычисления площади круга, если известно значение его радиуса.

Теперь же вы познакомитесь с логическими выражениями. *Логические выражения* используются не для вычисления какого-то конкретного результата, а для определения ложности или истинности проверяемого условия.

[!например. вас могут интересовать такие вопросы: "Любит ли она меня?", “Есть ли деньги па моем счете?" или "Нажал ли пользователь клавишу Enter?". Почти всегда, когда нужно получить ответ на вопрос о выполнении какого-то условия, используются логические выражения. Обычно в программах истинность условий проверяется в выражениях, подобных этому: "Если проверяемое условие истинно, выполните вот этот набор инструкций".

Если логическое выражение возвращает нулевое значение, проверяемое условие считается ложным (т.е. оно не выполняется). Если возвращаемое значение отлично от нулевого, условие считается истинным (оно выполняется).

В языке C++ логические выражения используются настолько часто, что для представления возвращаемых ими результатов предусмотрен даже отдельный тип данных: `bool`. У этого типа данных есть только два значения, для представления которых специально зарезервированы два ключевых слова: `true` (Истина) и `false` (Ложь).

В табл. 10.3 рассмотрены операторы, которые используются в логических выражениях. Отсюда происходит их название: *логические операторы*. В следующей Главе описано, как логические операторы могут использоваться для определения условий, проверяемых условными операторами (такими, например, как оператор `if`).

Таблица 10.3. Операторы сравнения (или логические операторы)

Оператор	Пример использования	Выполняемое действие
>	foo > bar	Больше чем. Возвращает логическое значение `true`, если значение слева больше значения справа. Например: 3 > 5 возвращает значение `false`; 3 > 1 возвращает значение `true`; 3 > 3 возвращает значение `false`, поскольку три равно трем, но не больше
>=	foo >= bar	Больше или равно. Возвращает логическое значение `true`, если значение слева больше значения справа или равно ему. Например: 3 >= 5 возвращает значение `false`; 3 >= 1 возвращает значение `true`; 3 >= 3 возвращает значение `true`, поскольку три равно трем
<	foo < bar	Меньше чем. Возвращает логическое значение `true`, если значение слева меньше значения справа. Например: 3 < 5 возвращает значение `true`; 3 < 1 возвращает значение `false`; 3 < 3 возвращает значение `false`
<=	foo <= bar	Меньше или равно. Возвращает логическое значение `true`, если значение слева меньше значения справа или равно ему. Например: 3 <= 5 возвращает значение `true`; 3 <= 1 возвращает значение `false`; 3 <= 3 возвращает значение `true`
==	foo == bar	Равенство. Возвращает логическое значение `true`, если значение слева равно значению справа. Например: 1 == 2 возвращает значение `false`; 1 == 1 возвращает значение `true`
!=	foo != bar	Не равно. Возвращает логическое значение `true`, если значение слева не равно значению справа. Например:

Оператор	Пример использования	Выполняемое действие
		1 != 2 возвращает значение true;
		1 != 1 возвращает значение false
!	!foo	Не. Требует значение только одного аргумента. Если аргумент имеет значение true, оператор возвращает значение false. Если аргумент имеет значение false, оператор возвращает значение true. Например:
		!1 возвращает значение false;
		!0 возвращает значение true
&&	foo && bar	Логическое И. Возвращает значение true только в том случае, если значению справа и значению слева соответствуют значения true. Например:
		1 && 1 возвращает значение true;
		0 && 1 возвращает значение false;
		используется в выражениях наподобие: "Если есть возможность && есть желание, тогда ..."
||	foo || bar	Логическое ИЛИ. Возвращает значение true в случае, если хотя бы одному из проверяемых значений соответствует значение true. Например:
		1 || 0 возвращает значение true;
		1 || 1 возвращает значение true;
		0 || 0 возвращает значение false

Оператор присвоения

Оператор присвоения (=) используется в тех случаях, когда необходимо присвоить значение какой-нибудь переменной. Например, если нужно сохранить входящую информацию или вычисленный результат. С оператором присвоения вы уже встречались ранее и видели, например, в такой строке:

```
fltArea = PI * fltRadius * fltRadius;
```

Когда происходит присвоение значения, то значение, вычисляемое расположенным справа от знака равенства выражением, присваивается переменной, указанной слева от знака равенства (=).

Оператор присвоения можно использовать несколько раз в одном и том же выражении. Например, в приведенном ниже коде значение 0 (нуль) присваивается сразу нескольким переменным:

```
a = b = c = 0;
```

Переменной можно присвоить только значение того же типа, что и сама переменная, или типа, который может быть автоматически преобразован к типу переменной. Продемонстрируем это на таком примере:

```
int a = 10;
```

Ошибки здесь нет, поскольку присваиваемое значение имеет тот же тип, который объявлен для переменной.

А вот в следующей строке содержится ошибка:

```
int a = "Sasha";
```

Переменная a может принимать только числовые значения типа integer (int). в то время как предпринимается попытка присвоить ей текстовое значение Sasha. Компилятор C++ будет с этим не согласен,

Не путайте оператор присвоения с оператором равенства

Обратите внимание, что логический оператор равенства == отличается от оператора присвоения =. Оператор присвоения (=) присваивает переменной, указанной слева, значение, расположенное справа. Оператор равенства {==) проверяет, совпадает ли значение, расположенное слева, со значением, расположенным справа, но при этом ни одно из значений не изменяется. Использование оператора = в тех местах, где подразумевается использование оператора ==, является довольно распространенной ошибкой, способной повлечь за собой серьезные проблемы.

Например, в результате выполнения приведенного ниже кода переменной a всегда будет присваиваться значение 2. Это будет происходить потому, что при проверке условия оператора if переменной a присваивается значение 1, а поскольку числу 1 соответствует логическое значение true, инструкция условного оператора a = a + 1 будет выполняться:

```
if (a = 1)
  a = a + 1;
```

Совершенно иначе выполняется код, показанный ниже, поскольку в этом случае значение переменной a изменяется только при условии, что оно равно единице:

```
if (a == 1)
  a = a + 1;
```

Чтобы не ошибиться и быть точно уверенным, что значения будут сравниваться, а не присваиваться, можно поступить следующим образом. Если нужно сравнить значение переменной с каким-то числом или каким-то постоянным значением, укажите его первым, а затем наберите знаки равенства и название переменной. Так, вместо кода

```
if (a == 1)
  a = a + 1;
```

наберите код

```
if (1 == a)
  a = a + 1;
```

Оба фрагмента будут работать одинаково. Однако вы можете случайно пропустить один знак равенства и набрать код

```
if (1 = a)
  a = a + 1;
```

В этом случае компилятор выдаст сообщение об ошибке. Таким образом вы рискуете получить только синтаксическую ошибку (которая сразу же будет обнаружена), вместо логической ошибки, найти которую не так просто и которая может оказаться не такой уж безобидной. Кстати, ошибочное использование оператора присвоения вместо оператора равенства настолько распространено, что во всех спорных случаях предусмотрено генерирование предупреждения компилятором. Поэтому старайтесь не оставлять без внимания появляющиеся предупреждения - это поможет вам сэкономить время и усилия при отладке программы.

Все об операторах

Довольно часто возникает необходимость совершить какое-то действие в отношение только одной переменной. Например, нужно прибавить какое-то число к общей сумме или умножить значение на какую-то константу.

Конечно, это можно сделать, набрав выражение наподобие такого:

```
foo = foo * 3;
bar = bar + 2;
```

Однако C++ предлагает для таких случаев набор специальных операторов присвоения, которые одновременно обрабатывают значение переменной и присваивают ей полученный результат. Выражения с такими операторами более лаконичны и наглядны. Работают они по следующему принципу. Выражение, наподобие

```
foo = foo operator bar;
```

можно заменить эквивалентным ему выражением

```
foo operator bar;
```

Например, вместо выражения

```
b = b + 1;
```

можно набрать

```
c += 1;
```

В табл. 10.4 приведен список всех специальных операторов присвоения и описаны выполняемые ими действия.

Таблица 10.4. Специальные операторы присвоения

Оператор	Пример	Выполняемое действие
+=	`foo += bar`	Добавляет значение, указанное справа, к значению переменной, указанной слева. Например, чтобы добавить к значению переменной foo число 3, наберите `foo += 3;`
-=	`foo -= bar`	Отнимает от значения переменной, указанной слева, значение, указанное справа. Например, чтобы вычесть из значения переменной foo число 3, наберите `foo -= 3;`
*=	`foo *= bar`	Умножает значение переменной, указанной слева, на значение, указанное справа. Например, чтобы умножить значение переменной foo на число 3, наберите `foo *= 3;`
/=	`foo /= bar`	Делит значение переменной, указанной слева, на значение, указанное справа. Например, чтобы разделить значение переменной foo на число 3, наберите `foo /= 3;`

Оператор	Пример	Выполняемое действие
%=	foo %= bar	Присваивает переменной, указанной слева, остаток, получаемый в результате деления исходного значения этой переменной на значение, указанное справа. Например, чтобы присвоить переменой foo остаток, получаемый при делении ее значения на число 10, наберите foo %= 10;
<<=	foo <<= bar	Выполняет сдвиг влево значения переменной, указанной слева, на число битов, равное значению, указанному справа. Например, чтобы сдвинуть значение переменной foo на два бита влево, наберите foo <<= 2;
>>=	foo >>= bar	Выполняет сдвиг вправо значения переменной, указанной слева, на число битов, равное значению, указанному справа. Например, чтобы сдвинуть значение переменной foo на два бита вправо, наберите foo >>= 2;
&=	foo &= bar	Выполнение поразрядной операции И с присвоением полученного результата переменной, указанной слева. Например, если переменная foo имеет значение 10, то в результате выполнения следующего выражения ей будет присвоено значение 2: foo &= 2;
|=	foo |= bar	Выполнение поразрядной операции ИЛИ с присвоением полученного результата переменной, указанной слева. Например, если переменная foo имеет значение 10, то в результате выполнения следующего выражения ей будет присвоено значение 11: fOO |= 1;
^=	foo ^= bar	Выполнение поразрядной операции ИСКЛЮЧИТЕЛЬНОЕ ИЛИ с присвоением полученного результата переменной, указанной слева. Например, если переменная foo имеет значение 10, то в результате выполнения следующего выражения ей будет присвоено значение 8: foo ^= 2;

Работа с битами

Для сохранения одного числа используется некоторое количество битов. Например, для числа типа integer (int) выделяется 32 бита памяти. Количество битов, выделяемых для переменной, определяет максимальное значение, которое она может принять.

Для логических переменных, несмотря на то что они могут принимать только два значения — true и false,обычно выделяется столько же памяти, сколько и для числовых переменных. Допустим, вам нужно сохранить и обработать большое количество логических значений. В этом случае вы можете в значительной степени оптимизировать использование памяти, если для представления одного логического значения будете использовать только один бит памяти (вместо обычных 32 битов).

Например представим, что создаваемая программа должна сохранить и обработать результаты опроса 10 000 респондентов, каждому из которых будет задано 32 вопроса, требующих ответа "нет" или "да". Если вы используете для этого обычные логические значения, для которых памяти нужно столько же, сколько для представления обычных чисел, программа потребует для

своего выполнения столько памяти, сколько нужно для представления 10 000 x 32 (320 000) чисел. Это даже больший объем памяти, чем имели некоторые первые компьютеры!

Если же для сохранения одного ответа вы будете использовать только один бит, ситуация меняется кардинальным образом. Действительно, если ответа может быть только два, 0 можно использовать для представления отрицательного ответа, а 1 — для положительного. Тогда нулевой бит числа можно использовать для хранения информации об ответе на первый вопрос, первый бит — для хранения информации об ответе на второй вопрос и т.д. (нумерация битов начинается с нулевого). Таким образом, каждая числовая переменная может хранить данные об ответе на 32 вопроса (так как в ее распоряжении есть 32 бита памяти) и ваша программа уже будет требовать для своего выполнения память в объеме, необходимом для хранения 10 000 чисел. Экономия существенная, не так ли?

Если вы используете числа именно таким образом, это называется созданием *битовых полей*. В табл. 10.5 описаны операторы, которые позволяют обрабатывать значения переменных на уровне битов (в таблице обрабатываемые значения и получаемые результаты представлены в двоичном виде).

Таблица 10.5. Побитовые операторы

Оператор	Пример	Выполняемое действие
~	~foo	Побитовое НЕ. Нули меняются на единицы и наоборот. Например: ~1011 даст 0100
<<	foo << bar	Сдвиг влево на указанное количество битов. Например: 1011 « 2 даст 1100 (см. также табл. 10.2)
»	foo >> bar	Сдвиг вправо на указанное количество битов. Например: 1011 » 2 даст 0010 (см. также табл. 10.2)
&	foo & bar	Побитовое И. Если биту в значении слева и биту под тем же номером в значении справа соответствует единица, возвращается единица. Во всех остальных случаях возвращается нуль. Например: 1011 & 1010 даст 1010
|	foo | bar	Побитовое ИЛИ. Если биту в значении слева или биту под тем же номером в значении справа соответствует единица, возвращается единица. В противном случае возвращается нуль. Например: 1011 I 1010 даст 1011
^	foo ^ bar	Побитовое ИСКЛЮЧИТЕЛЬНОЕ ИЛИ. Только если одному из сравниваемых битов соответствует единица, а второму- нуль, возвращается единица. В остальных случаях возвращается нуль. Например: 1011 ^ 1010 даст 0001

Условный оператор

Условным оператором называется оператор if. Он работает по тому же принципу, что и оператор IF, который вы могли видеть в электронных таблицах. Оператору if требуются три выражения. Вначале оценивается первое выражение. Если ему соответствует значение true,

возвращается результат, получаемый при обработке второго выражения, если false — результат, получаемый при обработке третьего выражения.

В электронных таблицах синтаксис оператора IF выглядел так:

```
IF(expr1, expr2, expr3)
```

Это означало следующее: если выражение expr1 истинно, вернуть значение expr2, в противном случае вернуть значение expr3.

В языке C++ то же самое записывается несколько иначе:

```
expr1 ? expr2 : expr3
```

Таким образом, для карточной игры можно было бы набрать нечто подобное:

```
UserMessage = (Очки > 21) ? "Перебор!" : "Еще карту?";
```

Приоритет операторов

Если вы еще не забыли того, что учили в школе на уроках математики, для вас не будет неожиданностью выполнение в формулах одних математических операций раньше других (даже если они указаны в конце формулы).

При обработке математических выражений компьютер придерживается тех же правил, которые вы учили в школе (и, возможно, уже забыли). Выражения обрабатываются слева направо, но все же некоторые операторы обрабатываются в первую очередь. Например, результатом выражения 3 + 2 x 3 будет число 9. Почему? Потому что оператор умножения имеет больший приоритет, чем оператор сложения, следовательно, вначале два умножается на три и только потом к полученному числу прибавляется три. Если бы приоритет операторов не учитывался и выражение просто вычислялось бы слева направо, мы получили бы число 15.

Чтобы изменить порядок обработки операторов, определяемых их приоритетом, используйте круглые скобки. Например, можете набрать выражение (3 + 2) x 3. В этом случае вначале будет обработан тот оператор, который заключен в скобки, т.е. оператор сложения, и только потом полученное число будет умножено на три.

В табл. 10.6 представлен список операторов с учетом их приоритета, Те операторы, которые указаны выше, обрабатываются раньше, чем расположенные ниже (другими словами, они *имеют больший приоритет).* Например, оператор + имеет больший приоритет, чем оператор >. Поэтому, выражение 1 + 0 > 1 равнозначно выражению (1 + 0) > 1. Но сами по себе, как вы понимаете, эти выражения ложны.

Те операторы, которые расположены в одной и той же строке таблицы, имеют одинаковый приоритет, поэтому если в выражении они стоят рядом, то вычисляются по очереди слева направо. Например, 3 x 4 / 2 равнозначно (3 x 4) /2.

Таблица 10.6. Приоритет операторов

Приоритет	Операторы
Высший приоритет	()
	++ -- ~ !
	* / %
	+ -
	>> <<
	< <= > >=
	== !=
	&

Окончание табл. *10.6*

Приоритет	Операторы
	^
	|
	&&
	||
Низший приоритет	? :

Если вы не уверены, в каком именно порядке будут обрабатываться операторы в набранном выражении, всегда добавляйте круглые скобки, чтобы убедиться, что вычислен будет именно тот результат, который вам нужен.

Двоичная и десятичная системы счисления

Если вы не занимались раньше программированием, упоминания о двоичных числах и **побитовых** операциях могут поставить вас в затруднительное положение. Система счисления, которой мы пользуемся в повседневной жизни и к которой привыкли, называется *десятичной.* Каждая цифра десятичного числа, в зависимости от своей позиции, означает единицы, десятки, сотни, тысячи и т.д. Например, число 125 представляется как $100 + 20 + 5$ или, что то же самое, как $1 \times 10^2 + 2 \times 101 + 5 \times 100$. (Со школы вы наверное еще помните, что любое число в нулевой степени равно единице.)

Компьютер не может представлять числа с использованием десяти различных цифр. Так как он может различать информацию только в формате **"да–нет"**, каждая позиция в числе может быть представлена только одной из двух цифр: нулем и единицей. Числа, представленные с помощью нулей и единиц, называются ***двоичными.*** Каждая цифра такого числа называется *битом* (или *двоичной **цифрой***). Например, двоичное число 1101 расшифровывается как $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. Если перевести это в десятичную систему счисления, то получим $8 + 4 + 0 + 1$, или просто число 13.

Для хранения одного числа компьютер использует набор из восьми битов, который обозначается термином байт. Один байт может представить 256 (или 2^8) уникальных значений. Два байта, объединенные вместе для представления одного значения, называются *словом.* Слово имеет в своем распоряжении 16 битов и может уже представить 65 536 (или 2^{16}) уникальных значений. Четыре байта (32 бита) образуют *двойное* слово.

Для тех, кто имеет отношение к компьютерам, число 2^{10} является ключевым. Это 1024 байта, или *килобайт* памяти (K). Хотя приставка *кило* обозначает тысячу, в мире компьютерных технологий она обозначает число 1024. Таким образом, например, 64K означает 64×1024, или 65 536 байт памяти.

Точно так же, хотя приставка *мега* (M) в обычном мире означает миллион, для компьютерщиков это число 1024×1024, или 1 048 576.

Поскольку двоичные числа не очень удобны (они слишком длинные), иногда используется *шестнадцатеричная* система *счисления.* В этой системе каждая позиция в числе может быть представлена 16 цифрами. Четыре бита объединяются для представления одной шестнадцатеричной цифры, которая называется ***гексит.*** Поскольку гексит может принимать значения от 0 до 15, для представления цифр, обозначаемых десятичными числами от 10 до 15 используются первые шесть букв латинского алфавита (от A до F). Другими словами, буква A обозначает 10, B- 11 и т.д. Если вам нужно набрать шестнадцатеричное число в кодах C++, начните его с приставки 0x. Так, число 0x0A соответствует десятичному числу 10, а число 0xFF - десятичному числу 255. Если среди ваших знакомых есть хакер, помешанный на языках программирования, спросите, сколько ему лет в шестнадцатеричной системе, и он не задумываясь ответит.

Почему компьютеры используют двоичную систему счисления? Это связано прежде всего с аппаратной частью, в частности с принципами работы транзисторов. Но это уже отдельная тема, и, если вас эти вопросы действительно интересуют, обратитесь к специальной литературе.

Примеры работы операторов

Чтобы получить лучшее представление о работе операторов, просмотрите приведенные ниже примеры.

Пример 1. Вычисление площади круга.

```
dblArea = PI * dblRadius * dblRadius;
```

Пример 2. Вычисление объема налога, которым облагаются продаваемые товары, если известны налоговая ставка и сумма, вырученная от продажи.

```
dblTax = dblPurchase * dblTaxRate;
```

Пример 3. Вычисление розничной цены товара при условии, что она должна включать в себя налог на продаваемые товары, объем которого был вычислен во втором примере.

```
dblPrice = (1 + dblTaxRate) * dblPurchase;
```

Пример 4. Сравнивается значение кредитного лимита с ценой товара, и, если лимит меньше цены, его значение увеличивается на 500.

```
DblCreditLimit = (dblPrice > dblCreditLimit) ?
            blCreditLimit + 500 : dblCreditLimit;
```

Математические функции

Во всех выражениях, с которыми вы сталкивались до этого, использовались только самые простые операторы. Но предположим, вам нужно вычислить что-то более серьезное и специфическое, связанное, например, с тригонометрическими операциями. Средство языка Visual C++, обозначаемое термином CLR (Common Language Runtime), предоставляет вам возможность использования целого набора математических функций для проведения специальных вычислений. Все они принадлежат классу Math. Наиболее часто используемые из них представлены в табл. 10.7.

Таблица 10.7. Математические функции

Функция	Пример использования	Выполняемое действие
Abs	`Math::Abs(-4)`	Возвращает абсолютное значение числа (модуль)
Ceil	`Math::Ceil(4.2)`	Округление до ближайшего большего целого. Например, применив эту функцию к числу 4,2, получим число 5
Cos	`Math::Cos(.03)`	Возвращает косинус числа. Число выражается в радианах
E	`Math::E`	Возвращает значение числа e, которое основано на вычислении натурального логарифма и равно приблизительно 2,718
Exp	`Math::Exp(4)`	Возвращает результат вычисления e^n, где вместо n используется аргумент функции
Floor	`Math::Floor(4.2)`	Округляет число до ближайшего меньшего целого. Например, если аргументом является число 4,2, возвращается число 4

Функция	Пример использования	Выполняемое действие
Log	Math::Log(4)	Возвращает значение натурального логарифма для указанного числа
Max	Math::Max(4.5, 8(3.2)	Возвращает большее из двух чисел
Min	Math::Min(4.5, Œ3.2)	Возвращает меньшее из двух чисел
PI	Math::PI	Возвращает значение числа π, которое равно приблизительно 3,141592
Pow	Math::Pow(4, 2)	Возвращает число x^n, где x - это первый аргумент, n - второй. В данном случае 4^2 даст число 16
Round	Math::Round(4.2)	Округляет число до ближайшего целого. Например, число 4,2 будет округлено до числа 4, а число 4,8 - до числа 5
Sin	Math::Sin(.03)	Возвращает синус числа. Число представляется в радианах
Sqrt	Math::Sqrt(4)	Возвращает квадратный корень числа
Tan	Math::Tan(.03)	Возвращает тангенс числа. Число представляется в радианах

Увидеть некоторые из этих функций в действии вы можете в приведенном ниже листинге программы, которая, как и прежде, вычисляет площадь круга.

```
//CircleArea4
//Вычисление площади по значению радиуса

//Использование математических функций

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;

//Объявление структуры для представления информации
//о параметрах круга
class CircleInfo
{
public:
   double dblRadius;
}; double dblArea;

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
```

```
#endif
{
  CircleInfo oCircle;

  //Отображение вопроса о значении радиуса
  Console::WriteLine(S"Укажите радиус круга");

  //Получение ответа и преобразование его к числу с
  //плавающей запятой
  oCircle.dblRadius = Double::Parse(Console::ReadLine());

  //Вычисление площади круга
  oCircle.dblArea= Math::PI*oCircle.dblRadius*
                              oCircle.dblRadius;

  //Отображение результата на экране
  //Обратите внимание, что для этого число должно
  //быть предварительна преобразовано в строку
 Console::WriteLine(S"Площадь круга составляет {0}
                     единиц.", oCircle.dblArea.ToString());

  //И еще одна (может и ненужная) математическая операция
  Console::WriteLine(S"Если вам это интересно, то
                корень квадратный от радиуса равен {0}",
                Math::Sqrt(oCircle.dblRadius).ToString());


  //Ожидание, пока пользователь не остановит
  //выполнение программы
  Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                       остановить выполнение программы");
  Console::ReadLine();

  return 0;
}
```

Старый формат математических функций

В действительности все основные математические функции, которые доступны при написании программ .NET, используются также и при написании стандартных программ C++. Эти функции являются составной частью библиотеки `math`, которая подключается таким кодом:

```
#include <math.h>
```

В табл. 10.8 представлены наиболее часто используемые функции библиотеки math. Помните, что их использование уместно только при написании неуправляемых программ.

Таблица 10.8. Математические функции языка C++

Функция	Пример использования	Выполняемое действие
abs	abs(-4)	Возвращает абсолютное значение числа (модуль)
ceil	ceil (4.2)	Округление до ближайшего большего целого
cos	cos (.03)	Возвращает косинус числа. Число выражается в радианах
exp	exp (4)	Возвращает результат вычисления e^n
floor	floor (4.2)	Округляег число до ближайшего меньшего целого
log	log (4)	Возвращает значение натурального логарифма для указанного числа
pow	pow(4, 2)	Возвращает число x^n, где x- это первый аргумент, п-второй
sin	sin(.03)	Возвращает синус числа. Число представляется в радианах
sqrt	sqrt(4)	Возвращает квадратный корень числа
tan	tan(.03)	Возвращает тангенс числа. Число представляется в радианах

Глава 11

Ключевые слова — ключ к диалогу с компьютером

В этой главе...

- Ключевые слова операторов управления
- Использование if для проверки условий
- Создание циклов с использованием слов for и while
- Использование слов switch, case и break

Если вы читаете книгу по порядку, вы уже получили представление об основных аспектах написания программ. Но до сих пор все рассматриваемые программы выполнялись только последовательно. Другими словами, выполнение программы всегда начиналось с первой строки функции main, а затем последовательно осуществлялся переход от одного выражения к другому, и порядок этот никогда не нарушался.

Однако, как вы уже могли убедиться на собственном опыте, течение жизни не всегда прямолинейно и порой случаются отклонения от общего курса. Хорошие программы обладают подобной гибкостью. Существуют, конечно, задачи, которые могут быть решены с помощью последовательно выполняющихся программ. Но в большинстве случаев программы должны уметь реагировать на некие внешние воздействия или команды и, в зависимости от этого, както изменять свое поведение. Для моделирования таких ситуаций C++ предлагает целый набор операторов, позволяющих управлять последовательностью выполняемых действий. Эти операторы разрешают выполнение определенной операции только в случае выполнения указанного условия. Кроме того, они способны выполнять один и тот же набор инструкций до тех пор, пока заданное условие не перестанет выполняться.

Существует множество ситуаций, которые невозможно смоделировать без использования таких операторов. Ниже приведено несколько примеров типичных задач, требующих для своего решения многократного повторения одних и тех же действий.

- ✓ Планомерное повышение цен до тех пор, пока товары успешно продаются (ценовое регулирование спроса).
- ✓ Продолжение стрельбы до тех пор, пока цель не будет поражена.
- ✓ Определение среднего возраста для группы из 32 студентов путем суммирования возраста всех студентов и деления полученного числа на 32.

В других ситуациях вам необходима возможность выбора. Обычно эта возможность формулируется так: если условие выполняется, сделайте это. Приведем примеры нескольких подобных ситуаций.

- ✓ Если профессор настаивает на выполнении домашнего задания, сделайте его. (В противном случае не делайте.)
- ✓ Если покупатель сделал более 3 000 покупок, предоставьте ему скидку.
- ✓ Если цвет желтый — жмите на газ. Если красный — тормозите.

Как видите, существует очень широкий спектр самых типичных задач, требующих для своего выполнения либо многократного повторения одних и тех же действий, либо осуществления выбора. Справиться с такими задачами призваны операторы, управляющие последовательностью выполняемых инструкций. И эта глава посвящена именно им.

Великолепная тройка: ключевые слова if, for и while

Три оператора управления используются почти во всех программах: if, for и while. Оператор if (называемый также *условным оператором)* выполняет определенный набор инструкций в том и только том случае, если заданное условие выполняется. Операторы for и while (называемые иногда *цикл* for и *цикл* while) повторяют нужное количество раз один и тот же заданный набор инструкций.

Условный оператор

Синтаксис условного оператора ifочень прост:

```
if (exprl)
     stmt;
```

(Обратите внимание, что здесь и далее в книге словом expr будет обозначаться любое выражение, например такое, как i < 1, а словом stmt — инструкция, например cost = cost + 1.)

Вместо exprl можно подставить любое выражение, которому может быть сопоставлено логическое значение. Если выражение истинно, выполняется инструкция stmt. Можно использовать фигурные скобки, для того чтобы заключить в них набор инструкций, которые должны будут выполняться в случае, если проверяемое выражение истинно. Например, следующим кодом проверяется значение логической переменной fDo Do, и, если оно совпадает со значением true, выполняется набор из трех инструкций:

```
if (fDcDo)
{
  nDeedle = 0;
  nDidle = 1;
  nDum = 0;
}
```

А этот код моделирует ситуацию, в которой покупателю предоставляется скидка, если он сделал покупки на сумму, превышающую $3 000:

```
if (nCount > 3000)
{
  dblDiscount = .2;
}
```

Можно придать оператору ifбольшую гибкость, если использовать вместе с ним ключевое слово else:

```
if  (exprl)
   stmt1;
else
   stmt2;
```

В этом случае, если проверяемое выражение (exprl)ложно, выполняется инструкция stmt2.

Приведенный ниже код моделирует эпизод карточной игры, когда проверяются набранные игроком очки, и, если их сумма превышает число 21, фиксируется перебор (`fBusted`), в противном случае предлагается взять еще одну карту:

```
if (nHandValue > 21)
{
  //В картах перебор
  nUserScore -= nBet;
  fBusted = true;
}
else
{
  //Игроку предлагается взять еще одну карту
  cout << "Еще?\n";
  cin >> fHitMe;
}
```

Введение в форматирование кодов программ

Коды программ можно форматировать множеством способов. И хотя для внешнего оформления кодов не существует никаких стандартов и ограничений, вы можете применить пару приемов, позволяющих значительно упростить их чтение и понимание.

Хорошей практикой является выделение с помощью отступов всех строк, заключенных в пару фигурных скобок. Таким образом вы сразу можете видеть набор инструкций, выполняемых вместе. Например, следующий код легко читается, поскольку к нужным строкам добавлены отступы:

```
if (nHandValue > 21)
{
   //В картах перебор
   nUserScore -= nBet;
   fBusted = true;
}
else
<
   //Игроку предлагается взять еще одну карту
   cout « "Еще?\п";
   cin » fHitMe;
>
```

А вот тот же код, набранный без отступов:

```
if (nHandValue > 21)(
//В картах перебор
nUserScore -= nBet; fBusted = true;} else {"
//Игроку предлагается взять еще одну карту
cout « "Еще?\п"; cin >> fHitMe;}
```

Первый код проще, поскольку из него ясно видно, какие инструкции будут выполнены в случае выполнения условия, а какие в случае невыполнения.

Если вы читаете главы книги по порядку, то уже видели использование этого приема на примере функции `main`, все строки которой выделяются отступами. А также видели нечто подобное при создании структур.

При использовании вложенных операторов выделяйте коды каждого из них дополнительными отступами. Например:

```
if(too)
{
  bar++
  if(bar>3)
  {
    baz=2;
  }
  if (goober < 7)
  {
  } flibber = 3;
}
```

Другим приемом, облегчающим чтение кодов программы (который также использовался во всех приведенных выше примерах), является размещение закрывающей фигурной скобки (}) на одном уровне с соответствующей ей открывающей скобкой:

```
if (fFoo)
{
}
```

Благодаря этому легко увидеть, где начатый блок кодов заканчивается. Кроме того, хотя для оператора if использовать фигурные скобки обязательно только в том случае, если он содержит более одной инструкции для выполнения, постоянное их применение может предотвратить возникновение некоторых тривиальных ошибок. Например, в следующем коде фигурные скобки не используются:

```
if (fFoo)
  nVal++;
nCheck++;
```

Приведенный ниже код выполняет точно те же действия, но читается намного проще и позволяет в случае необходимости без особых усилий добавить к оператору if дополнительные инструкции для выполнения:

```
if (fFoo)
{
  nVal++;
}
nCheck++;
```

Visual C++ .NET форматирует коды вашей программы автоматически. Например, если вы выделите с помощью отступа какую-то строку, набираемые после нее строки будут автоматически выравниваться по ней. Если вы наберете скобку }, она автоматически будет выровнена по соответствующей ей скобке {.

Следующий код определяет размер предоставляемой скидки в зависимости от суммы совершенных покупок:

```
//Если сумма превышает $5000, скидка равна 30%
if (nSumm > 5000)
{
  dblDiscount = 0.3;
}
else
{
  //Если сумма превышает $3000, скидка равна 20%
  if (nSumm > 3000)
```

```
  {
    dblDiscount = 0.2;
  }
  //В противном случае скидка не предоставляется
  else
  {
    dblDiscount = 0.0;
  }
}
```

В некоторых случаях, как в приведенном выше примере, один оператор if используется внутри другого оператора if. Это называется *вложением* операторов. Чтобы коды вложенных операторов были проще для чтения и понимания, добавляйте отступы, выделяющие границы каждого из них.

Оператор for

Ключевое слово `for` используется в тех случаях, когда какие-то инструкции должны выполняться подряд определенное количество раз. Синтаксис оператора `for` выглядит следующим образом:

```
for (expr1; expr2; expr3)

  stmt1;
```

Такой код еще называют *циклом* `for`.

При выполнении цикла `for` в первую очередь обрабатывается выражение `expr1`, которым определяется начальное значение переменной, используемой для контроля за количеством итераций. (*Итерация* — это одноразовое выполнение инструкций цикла.) Затем оценивается выражение `expr2`. (Оно оценивается перед каждой итерацией, для того чтобы определить, следует ли продолжать выполнение цикла.) Если выражение `expr2` истинно, выполняется инструкция `strati`, после чего обрабатывается выражение `exprЗ`. Выражение `exprЗ` используется обычно для изменения значения переменной, контролирующей количество итераций. Если выражение `expr2` ложно, выполнение цикла прекращается и программа переходит к обработке следующих за этим циклом кодов.

Примериспользованияциклаfor

Приведенное выше объяснение может показаться вам не очень понятным, поэтому продемонстрируем работу цикла `for` на практическом примере.

```
int i;
for (i = 0; i < 2; i++)
{
  Console::WriteLine(i.ToString());
}
```

Опишем, как этот код выполняется.

1. Вначале обрабатывается выражение `expr1`. Переменной i присваивается значение 0.
2. Затем обрабатывается выражение `expr2`. Проверяется, меньше ли значение переменной i числа 2. Поскольку переменной i только что было присвоено значение 0, выражение `expr2` будет истинным, что дает разрешение на выполнение инструкции `stmt1`. В данном случае эта инструкция выглядит так:

   ```
   Console::WriteLine(i.ToString());
   ```

   Ее выполнение приводит к отображению на экране значения переменной i.

3. Далее обрабатывается выражение `expr3`. В данном случае оно выглядит как `i++`, таким образом, значение переменной `i` увеличивается на единицу.
4. Поскольку выражение `expr2` должно оцениваться перед выполнением каждой итерации, программа снова возвращается к нему. Переменная i имеет уже значение 1, но это все равно меньше числа 2. Поэтому инструкция `strati` выполняется еще раз, и на экране отображается следующее значение переменной i.
5. Снова обрабатывается выражение `expr3`. Значение переменной i увеличивается на единицу и становиться равным числу 2.
6. Программа переходит к проверке выражения `expr2`. Теперь оно ложно, поскольку значение переменной i (2) не меньше числа 2. На этом выполнение цикла заканчивается.

Изменяя и усложняя выражение `expr2`, можно задавать самые разные условия, определяющие количество выполняемых итераций и момент прекращения выполнения цикла. Далее в примерах этой книги вы еще неоднократно встретитесь с ним.

Повторение ради повторения

Если вы хотите, чтобы инструкция была выполнена несколько раз подряд, наберите такой код:

```
for (i = 0; i < n; i++)
{
  //Инструкция или набор инструкций для повторения
}
```

Значение n определяет количество выполняемых итераций. Например, если вам нужно, чтобы строка "Терпение и еще раз терпение" была выведена на экран 50 раз подряд, наберите такой код;

```
for (i = 0; i < 50; i++)
{
  Console: :WriteLine(S"Терпение и еще раз терпение");
)
```

Чтобы не показаться навязчивым, можете позволить пользователю самому определить, сколько раз строка должна отобразиться на экране. Делается это так:

```
int nCount;
Console::WriteLine(S"Сколько строк отобразить?");
nCount = Console: :ReadLine() ->ToInt32();

for (i = 0; i < nCount; i++)
{
  Console::WriteLine(S"Терпение и  еще раз терпение");
}
```

Вычисление факториала

Наверное, каждый программист когда-нибудь сталкивался с такой проблемой: как написать код, вычисляющий n-факториал?

Это выражение (п-факториал) вычисляется по такой формуле: пх (п - 1) х (п - 2) х ... х 1. Так, 2-факториал (пишется как 2! в математических книгах, но не в компьютерных кодах) вычисляется как 2 х 1, а 3! ~ как 3 х 2 х 1.

Одним из вариантов решения такой проблемы может быть приведенный ниже код.

```
//Вычисление  г.!
p. = Int32: : Parse (Console: : ReadLine () ) ;
```

```
if (n == 1)
{
  Console::WriteLine(S"1");
}
else if (n == 2)
{
  Console::WriteLine(S"2");
}
else if (n == 3)
{
  Console::WriteLine(S"6");
}
```

Этот способ, несмотря на свою простоту, крайне неэффективен, поскольку, даже если вы будете продолжать аналогичным образом набирать коды для чисел 4, 5, ... 35 и т.д., все равно вы не сможете перебрать даже малую часть всех возможных вариантов. Реальным решением этой проблемы может быть использование цикла for.

```
//Factorial
//Вычисление n!

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#eise
int main(void)
#endif
{
  int nNumber;  //Число, определяемое пользователем
  int nResult = 1;

  int i;  //Переменная, используемая для отсчета
          //количества итераций
  //Получение числа, для которого нужно вычислить
  //факториал
  Console::WrileLine("Для какого числа?");
  nNumber = Int32::Parse(Console::ReadLine());

  //Далее вычисляется факториал путем умножения на
  //каждой итерации значения переменной i на
  //общий результат. Таким образом мы получим число,
  //равное 1*2*3*...*n
  for (i=1; i<=nNumber; i++)
  {
    nResult *= i;
  }

  //Отображение полученного значения
  Console::WriteLine(S"n! равен {0}", nResult.ToString());
  //Ожидание, пока пользователь не остановит
```

```
  //выполнение программы
  Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                      остановить выполнение программы");
  Console::ReadLine();

  return 0;
}
```

Оператор while

Как и цикл for, цикл while используется в тех случаях, когда некоторая инструкция (или набор инструкций) должна быть выполнена несколько раз подряд. Его синтаксис даже проще, чем синтаксис оператора for, и выглядит следующим образом:

```
while (expr1)
  stmt;
```

Вначале оценивается выражение expr1. Если оно истинно, выполняется инструкция stmt. Затем опять проверяется выражение expr1. До тех пор пока выражение expr1 будет оставаться истинным, инструкция stmt будет выполняться снова и снова. Как только выражение expr1 становится ложным, выполнение цикла прекращается.

Например, если нужно создать цикл, выполняющий десять итераций, наберите такой код:

```
int i = 0;
while (i < 10)
{
  i++;
}
```

Среди инструкций stmt (те, которые выполняются на каждой итерации) обязательно должна быть та, которая как-то изменяет значение переменной, входящей в состав выражения expr1. В противном случае выполнение цикла никогда не прекратится.

Бесконечный цикл

Ниже приведен пример небольшой программы, выполнение шторой неминуемо приведет к зависанию компьютера.

```
//Компьютер завис
int main(void)

  while(1);
}
```

Произойдет это по следующей причине. Выполнение цикла while продолжается до тех пор, пока выражение expr1 не станет ложным. В данном случае выражению expr1 постоянно соответствует число 1, которое всегда воспринимается как истинное. Поэтому выполнение цикла while без внешнего вмешательства никогда не прекратится. Подобные циклы обычно называют *бесконечными*.

Ключевые слова switch и do

Visual C++ имеет в своем запасе еще пару операторов, управляющих последовательностью выполняемых инструкций. Познакомимся с ними поближе.

Оператор switch

Оператор switch подобен оператору if, но позволяет учесть сразу множество вариантов и выбрать только один из них. (Описание каждого варианта начинается с ключевого слова case.) Таким образом, если необходимо создать код, решающий задачу наподобие; "если это Вася, тогда ..., если это Маша, тогда ..., если это Дима, тогда ...", используйте оператор switch, который может заменить собой сразу несколько операторов if. Синтаксис оператора switch выглядит так:

```
switch (expr)
{
  case val1:
     stmt1;
  case val2:
     stmt2;
  ...
  default:
     dflt stint;
}
```

Вначале оценивается выражение expr и его значение сравнивается со значением val1. (Значение val1 должно быть каким-нибудь числом, например 1 или 23,5.) Если значение выражения expr совпадает со значением val1, выполняется инструкция stmt1 и все последующие за ней инструкции цикла switch. Если значение выражения expr не совпадает со значением val1, оно сравнивается со значением val2 и т.д. Если вы хотите быть уверены, что по крайней мере какая-то инструкция будет обязательно выполнена (в том случае, если ни одно из заданных значений не совпадет со значением выражения expr), добавьте ключевое слово default (как показано выше) и укажите после него нужную инструкцию. Если после выполнения какой-то инструкции вы не хотите, чтобы все последующие инструкции также выполнялись, наберите после нее слово break, которое сразу же остановит выполнение оператора switch.

Если вы знакомы с языком Visual Basic, вы, наверное, ожидаете, что, как и в Visual Basic, в качестве проверяемых значений val могут быть использованы строки. К сожалению, это не так. Сравниваться могут только числовые значения, но не текстовые.

Продемонстрируем работу оператора switch на примере, отображающем названия некоторых чисел:

```
//Названия чисел 1, 2, 3 и 4
switch (n)
{
  case 1:
     Console::WriteLine(S"один");
     break;
  case 2:
     Console::WriteLine(S"два");
     break;
  case 3:
     Console::WriteLine(S"три");
     break;
  case 4:
     Console::WriteLine(S"четыре");
```

```
      break;
  default:
      Console::WriteLine(S"неизвестное число");
}
```

Обратите внимание на наличие ключевых слов `break` после каждой инструкции, следующей за словом `case`. Если бы их там не было, возвращаемый программой результат выглядел бы так:

n	Результат
1	одиндватричетыренеизвестное число
2	дватричетыренеизвестное число
и т.д.	

Не забывайте добавлять после инструкций, следующих за каждым словом `case` ключевое слово `break`. Если этого не делать, выполняться будут все инструкции оператора `switch`, набранные после слова `case` со значением, которое совпало с проверяемым.

Оператор do

Выполнение цикла do во многом подобно выполнению цикла `while`. Отличие состоит в том, что цикл `while` проверяет истинность выражения до того, как будет выполнена какая-либо инструкция. Поэтому возможен вариант (если выражение сразу окажется ложным), при котором ни одна из инструкций не будет выполнена. Цикл do, наоборот, вначале выполняет первую итерацию и только затем проверяет истинность выражения. Если выражение истинно, выполняется следующая итерация и т.д. Когда выражение становится ложным, выполнение цикла прекращается.

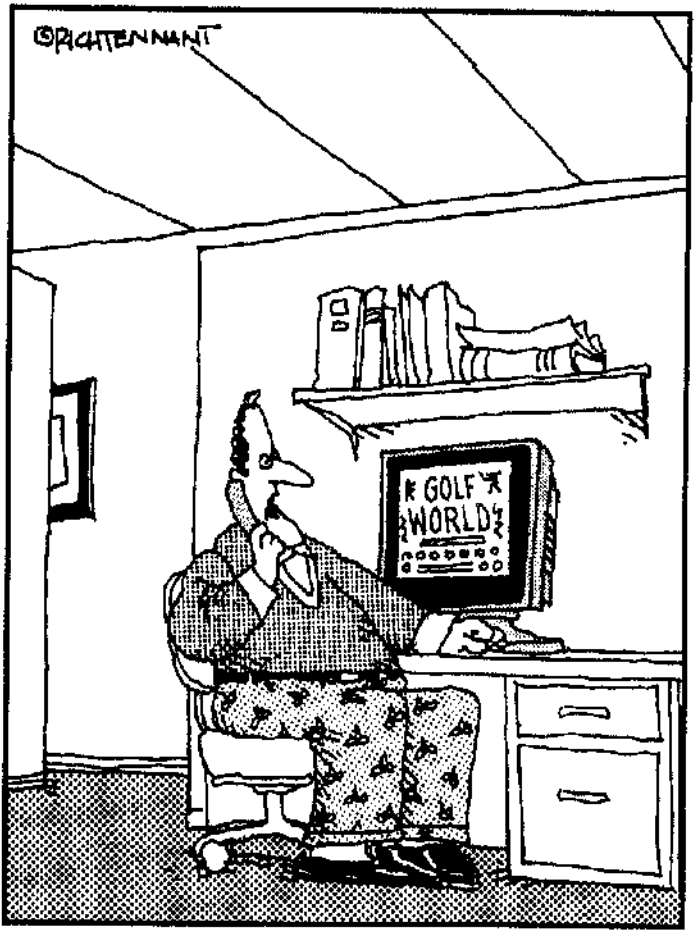
```
Do
  stmt;
while (expr);
```

Вот пример использование цикла do, который выполняется до тех пор, пока значение переменной i не сравняется со значением переменной n:

```
int i = 0;
do
{
  Console::WriteLine(i.ToString());
  i++;
}
while (i < n);
```

Если даже переменная n имеет значение 0, это число все равно будет отображено на экране, поскольку проверяемое выражение (в данном случае i<n) оценивается только после выполнения первой итерации.

Выражение i=i+1 эквивалентно выражению i++.



Я не могу понять, КАК это происходит, но когда GOLF мой игрок бьет клюшкой по мячику, на мою Web-страницу летят маленькие комки кодов.

Глава 12

Внимание! Повышенная функциональность

В этой главе...

- Что такое функция
- Создание функций
- Использование аргументов
- Результат, возвращаемый функцией
- Рекурсия и значения аргументов, установленные по умолчанию

Большинство создаваемых программ сложны и объемны. Некоторые из них состоят из тысяч и даже миллионов инструкций. Чтобы упростить процесс создания, отладки, модификации и вообще понимания огромных (и не очень) программ, их разбивают на некоторое количество более простых подпрограмм.

Visual C++ позволяет разбивать программу на более мелкие составляющие путем объединения логически связанных между собой выражений и присвоения им какого-то имени. Такие выделенные в отдельную группу выражения называются *функциями*. (Иногда функции называют *подпрограммами* или *процедурами*. В данной книге в основном будет использоваться термин *функция,* однако другие два термина также корректны.)

Функции могут иметь различную область видимости. *Глобальные функции* могут быть вызваны из любой части программы. *Библиотечные функции* могут вызываться самыми разными программами. Однако большинство создаваемых вами функций будут, скорее всего, использоваться в пределах только одного объекта. Такие функции, называемые *функциями-членами,* подробно рассматриваются в главе 17.

Можно также использовать ранее созданные функции для построения новой функции. Этот прием позволяет значительно упростить процесс написания, чтения и редактирования кодов программ. В этой главе описывается, как использовать функции для разбивки большой программы на небольшие и понятные составляющие.

Некоторые вводные замечания

Если вы просматривали предыдущие главы, то наверняка обратили внимание на приводимые в качестве примеров коды программ, В главе 3 упоминалось правило, согласно которому все выражения должны заканчиваться точкой с запятой (;). Но как известно, из любого правила есть исключения, и подтверждением тому могут быть некоторые выражения в приводимых ранее примерах, после которых точка с запятой не ставилась.

Итак, общее правило звучит так:

Почти все выражения должны заканчиваться точкой с запятой.

А вот исключения из этого правила.

- ✓ Если выражение начинается с символа # (знак "решетки"), в конце него точка с запятой не ставится.

- ✓ Если выражение начинается с символов //, точкой с запятой заканчивать его не обязательно, поскольку это комментарий и компилятор все равно его проигнорирует.
- ✓ Точка с запятой не ставится после закрывающей фигурной скобки (}). за исключением тех случаев, когда этим символом заканчивается объявление класса (class), структуры (struct) или перечня (enum).

Создание функций

Определение функции начинается с набора ее имени и пары круглых скобок (). (Позже, когда вы познакомитесь с *аргументами,* в упомянутых скобках можно будет определить список этих самых аргументов.) Далее следуют выражения, из которых собственно и состоит функция. Правила присвоения функциям имен в точности совпадают с правилами присвоения имен переменным, которые были рассмотрены в главе 8. Итак, вот код, которым определяется функция:

```
void function_name()
{
  stmt;
}
```

Например, следующим кодом создается функция, отображающая на экране текст "Hello World".

```
void PrintHW()
{
  Console::WriteLine(S"Hello World");
}
```

Теперь каждый раз, когда вы хотите, чтобы функция была выполнена, наберите ее имя и сразу за ним пару пустых скобок. Это действие называется *вызовом* функции. Функция может быть вызвана неограниченное количество раз.

Как и в случае со структурами, функции должны быть определены до того, как будут использованы, что показано в приведенном ниже примере. Это программа HelloWorld, которую вы уже видели ранее, но здесь код, заставляющий программу ожидать, пока пользователь не нажмет на клавишу <Enter> и не закончит ее выполнение, выделен в отдельную функцию, названную именем HangOut. Функция HangOut определена в начале программы и затем вызывается функцией main.

```
//HelloWorld4
//Использование функции

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;

//Ожидание, пока пользователь не остановит
//выполнение программы
void HangOut()
{
  Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                      остановить выполнение программы");
  Console::ReadLine();
}
```

```
//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  //Отображение текста на экране
  Console::WriteLine("Hello World");

  //Ожидание, пока пользователь не остановит
  //выполнение программы
  HangOut();
  return 0;
}
```

Использование аргументов

Функция может требовать для своего выполнения передачи ей значений аргументов. Для каждого *аргумента* (их называют также *параметрами*)должно быть определено имя и тип данных. Функции, работа которых зависит от передаваемых значений аргументов, более универсальны и могут многократно использоваться в процессе выполнения программы. Для каждой функции можно определить любое количество аргументов любого типа данных.

Вот как определяются аргументы:

```
void function_name(data_type1 arg1, data_type2 arg2,  ...)
{
}
```

Например, приведенная ниже функция вычисляет факториал числа. Число, представляемое переменной n, передается функции в качестве значения аргумента. Далее это значение используется функцией при проведении вычислений.

```
//Вычисление и отображение факториала числа
void Factorial(int nNumber)
{
  int nResult = 1;
  int i;  //Используется для отсчета итераций

  //Цикл, вычисляющий факториал. На каждой итерации
  //общий результат умножается на i
  for (i=1; i<=nNumber; i++)
    {
      nResult *= i;
    }

  //Отображение полученного результата
  Console::WriteLine(nResult.ToString());
}
```

Каждый раз, когда потребуется отобразить на экране значение факториала какого-нибудь числа, просто вызовите эту функцию. Например, ниже приведен код цикла, выполняющего три итерации. На каждой итерации программа запрашивает число, для которого нужно вычислить факториал, и затем вызывает функцию `Factorial`, которая отображает нужное значение на экране.

```
int nNumber;
int i;
```

```
//Цикл, выполняющий три итерации
for (i=0; i<3; i++)
{
  //Получение числа от пользователя
  nNumber = Int32::Parse(Console::ReadLine());
  //Вызов функции с передачей ей значения аргумента
  Factorial(nNumber);
}
```

Так же просто создается функция, требующая значений нескольких аргументов. Например, следующая функция отображает результат, вычисленный по формуле `foo*n!`, где значения переменных foo и n являются аргументами этой функции:

```
//Функция, использующая значения двух аргументов
viod Fooctorial(int r.Foo, int nNumber)
{
  int nResuIt = 1;
  int i;

  //Цикл, вычисляющий факториал. На каждой итерации
  //общий результат умножается на i
  for (i=1; i<=nNumber; i++)
    {
      nResult *= i;
    }

  //Умножение факториала на число foo
  nResult *= nFoo;

  //Отображение полученного результата
  Console::WriteLine(nResult.ToString() ;
}
```

Функции, которые возвращают результат

Все функции, рассмотренные ранее, выполняли какие-то действия (например, вычисляли факториал и отображали полученное значение на экране). Однако функции могут также возвращать некоторое значение в качестве собственного результата. Эта возможность очень полезна, поскольку позволяет использовать функции при построении выражений. Например, можно использовать математические библиотечные функции, такие как `Math::Cos ()`, внутри выражений, подобных этому: `3*Math: :Cos (angle)`.

Вы можете создавать собственные функции, возвращающие результат. Например, можете создать функцию, просматривающую базу данных и возвращающую имя клиента, совершившего наибольшее количество покупок. Или можете создать функцию, возвращающую в качестве результата общую сумму совершенных за последний месяц сделок.

Чтобы функция возвращала какой-то результат, нужно выполнить два условия:

- ✓ при определении функции перед ее именем вместо слова `void` указать тип данных, который будет иметь возвращаемое функцией значение;
- ✓ использовать ключевое слово `return` в кодах функции до того, как выполнение функции будет остановлено.

Словом `return` возвращается полеченный результат, а выполнение самой функции немедленно прекращается. Если вы наберете слово `return` посреди кодов функции и оно будет

обработано, все остальные коды функции, набранные после слова return, выполняться не будут. (Не все слова return обрабатываются в обязательном порядке. Например, если оно будет набрано среди инструкций оператора if, обрабатываться оно будет только в случае выполнения проверяемого условия.)

Ниже приведен пример функции, вычисляющей факториал и возвращающей его в качестве своего значения. Эта функция работает так же, как и предыдущая, за исключением того, что полученное значение не отображается на экране, а возвращается как результат функции.

```
//Вычисление факториала числа
void Factorial(int nNumber)
{
  int nResult = 1;
  int i;  //Используется для отсчета итераций

  //Цикл, вычисляющий факториал. На каждой итерации
  //общий результат умножается на i
  for (i=1; i<=nNumber; i++)
    {
      nResult *= i;
    }

  //Полученное значение возвращается как результат
  return nResult;
}
```

Поскольку теперь функция Factorial возвращает результат, вы можете использовать ее при построении выражений. В некоторых случаях это может быть очень полезной возможностью. Вот, например, код, где функция Factorial вызывается функцией WriteLine, которая сразу же отображает полученное значение на экране:

```
nNumber = Int32::Parse(Console::ReadLine());
Console::WriteLine(S"Факториал числа {0} равен {1}",
           nNumber.ToString(),
           Factorial(nNumber).ToString());
```

Функции, возвращающие значения, могут быть использованы в любом месте, где могут быть использованы значения того же типа, что и возвращаемые функциями результаты. Так, если функция возвращает числовое значение типа integer, ее можно использовать в любом месте, где можно подставить число типа integer. Например, в таком выражении:

```
nMyNumber = 3*Factorial(nNumber);
```

Возвращаемые функциями результаты могут также быть использованы как значения аргументов, передаваемые другим функциям. Например, чтобы вызвать функцию Factorial, ей нужно передать в качестве аргумента числовое значение типа integer. Поскольку сама функция также возвращает значение типа integer, оно может быть использовано и как аргумент этой функции. Ниже показан код, в результате выполнения которого на экране отображается значение факториала от факториала числа.

```
nNumber = Int32::Parse(Console::ReadLine());
Console::WriteLine(Factorial(Factorial(nNumber)).ToString());
```

А этим кодом определяется, будет ли факториал заданного числа больше числа 72:

```
if ((Factorial(nNumber) > 72)
{
  Console::WriteLine(S"Факториал больше чем 72");
}
```

Плюсы и минусы глобальных переменных

После того как функция завершает свою работу, все значения всех переменных, объявленных внутри этой функции, теряются. После этого восстановить информацию, которая хранилась как значения этих переменных, уже невозможно. Если же она вам еще нужна, ее необходимо вернуть с использованием слова return как результат этой функции.

Предположим, например, что вам нужно узнать имя самого высокооплачиваемого сотрудника вышей организации. Если вы хотите только отобразить это имя на экране и больше никогда к нему не возвращаться, наберите команду вывода на экран внутри функции и не возвращайте никакого результата. Но, если это имя должно быть как-то использовано за пределами функции, например при создании какого-нибудь отчета, верните его как результат функции.

Сохранить информацию можно и без использования ключевого слова return. Делается это с помощью так называемых *глобальных переменных,* которые объявляются до начала работы функции main. Свое название эти переменные получили благодаря возможности их использования в любом месте программы. Если вернуться к нашему примеру, то имя самого высокооплачиваемого сотрудника может быть сохранено как значение одной из глобальных переменных. Значения глобальных переменных не пропадают после завершения работы какой-либо отдельной части программы, поэтому они могут использоваться для сохранения информации, полученной во время выполнения функции.

Однако, к сожалению, использование глобальных переменных может сделать коды программы похожими на спагетти (так называют коды, которые трудно прочитать и понять, как они работают). Если вы будете использовать глобальные переменные внутри функций, понять, что произойдет в результате выполнения кодов программы, будет крайне сложно, так как вам придется просматривать подряд все коды программы. Это нельзя считать хорошей практикой, так как в идеале вы должны без просмотра каждой отдельной строки функции суметь точно определить, значения каких аргументов ей нужно передать и что произойдет в результате ее выполнения. Это важно, поскольку в этом случае вы можете понимать назначение и использовать функции без необходимости вникать во все нюансы выполнения отдельных кодов,

Большинство сложных для обнаружения логических ошибок возникают из-за неаккуратного использования глобальных переменных. Поэтому намного предпочтительнее возвращать результат функции, чем использовать внутри нее глобальные переменные. Если необходимо вернуть много значений, воспользуйтесь структурой или ссылкой на структуру.

Основа вернемся к факториалам

В этом разделе мы вновь вернемся к программе, вычисляющей факториал заданного числа. Но теперь программа будет состоять из нескольких функций. Когда будете просматривать коды программы, обратите внимание, что, хотя теперь она выглядит несколько сложнее, колы функции main стали намного проще. Ведь теперь они состоят всего лишь из четырех выражений (если не считать комментариев).

В программе кроме функции main используются еще две функции: Factorial и GetNumber. Функция Factorial вычисляет факториал заданного числа. Она принимает в качестве аргумента число, для которого нужно вычислить факториал, и возвращает в качестве результата факториал этого числа. Функция GetNumber используется для получения от пользователя числа, для которого нужно вычислить факториал.

Функция main снова и снова использует функцию GetNumber для получения от пользователя новых чисел. Получив очередное число, функция main вызывает функцию Factorial для вычисления факториала и отображения его на экране. Это продолжается до тех пор, пока пользователь не наберет число 0 (нуль).

Обратите внимание функция main определяет, когда нужно остановиться:

```
while (nNumber != GetNumber())
```

Помните, что цикл while использует выражение как условие для выполнения или невыполнения следующей итерации. Итерация выполняется, если проверяемое выражение истинно. В нашем случае выражение вызывает вначале функцию GetNumber для получения числа от пользователя. Затем полученное число присваивается переменной nNumber. Далее возможны два варианта. Если пользователь набирает число 0, выполнение цикла прекращается, поскольку этому числу соответствует логическое значение false и выражение воспринимается как ложное. Если же пользователь набирает любое положительное число, оно присваивается переменной nNumber и выполняется итерация цикла. Этот способ использования цикла while довольно часто можно видеть в программах C++. (Если вы тестируете программу или просто любите создавать проблемы, можете ввести отрицательное число, тогда программа выдаст неправильный результат. Чтобы избежать этого, можете добавить код, который будет проверять вводимые пользователем числа и просить ввести число заново, если оно окажется отрицательным.)

Итак, вот новый код программы, вычисляющей факториал:

```
//Вычисление факториала до тех пор, пока
//пользователь не наберет число О

#include "stdafx.h"

#using <mscorlib.dll>

using manespace System;

//Функция вычисляет и возвращает значение факториала
int Factorial(int nNumber)
{
  int nResult = 1;
  int i;  //Используется для отсчета итераций

  //Цикл, вычисляющий факториал. На каждой итерации
  //общий результат умножается на i
  for (i=1; i<=nNumber; i++)
    {
      nResult *= i;
    }
  //Возвращение результата
  return nResult;
}

//Функция просит пользователя набрать число,
//а затем возвращает это число как результат
int GetNumber()
{
  int nNumber;

  Console::WriteLine(S"Введите число");
  nNumber = Int32::Parse(Console::ReadLine());
  return nNumber;
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
```

```
ir:t wmain (void)
#else
int main (void)
#endif
{
  int nNumber;

  //Получение чисел от пользователя до тех пор,
  //пока он не наберет число О
  while (nNumber = GetNumber())
  {
    //Отображение значение факториала для указанного
    //числа. При этом используется значение,
    //возвращаемое функцией Factorial
    Console::WriteLine(S"Факториал числа {0} равен {1}",
          nNumber.ToString(),
          Factorial(nNumber).ToString());
  }

  //Окончание выполнения программы
  Console::WriteLine(S"Bye");
  return О;
}
```

Чтение программ, содержащих функции

Если программа содержит функции, они обычно определяются до того, как будут использованы. Поэтому, если вы начнете читать коды программы от начала и до конца строка за строкой, вам вначале придется проштудировать множество разрозненных деталей и только в конце увидеть, для чего они используются и в какой последовательности выполняются.

Приведем несколько советов, которые могут заметно упростить этот процесс.

- ✓ Если файл содержит функцию main, пропустите остальные коды и начинайте с просмотра именно этой функции. Если функция main вызывает какую-то другую функцию, вернитесь к определению этой функции и просмотрите ее коды.
- ✓ Если файл состоит только из набора различных функций и не включает в себя функцию main, просмотрите вначале названия всех функций. Прочитайте комментарии, объясняющие назначение каждой функции. После этого определите для себя, детали выполнения каких функций вас интересуют, а какие функции, наоборот, можно проигнорировать.
- ✓ Обычно коды наиболее важных функций набраны в конце файла.

Если вас удивляет тот факт, что переменным внутри функций присваиваются те же имена, что и переменным, используемым за пределами этих функций, не переживайте – в главе 15 этот вопрос рассмотрен подробно.

Рекурсия спасибо мне, что есть я и меня

Если функция вызывает саму себя, это называется *рекурсией*. Рекурсия очень часто применяется в тех случаях, если одну и ту же задачу намного проще выполнить для меньшего числа исходных элементов.

Предположим, например, что необходимо отсортировать большой набор чисел. (Это классическая задача, часто рассматриваемая в книгах по программированию. Возможно, предлагаемое здесь решение с помощью использования рекурсии вам покажется не вполне очевидным, но, в отличие от реальной жизни, в контексте программирования именно этот подход считается одним из самых оптимальных.) Отсортировать большой набор чисел крайне сложно. Самый простой способ сделать это — просмотреть весь список и найти наименьшее число, перенести его в новый список, который будет итоговым, и затем продолжать этот процесс до тех пор, пока все числа не будут перенесены в этот новый список. Недостаток метода состоит в том, что один и тот же список приходится просматривать снова и снова, а это занимает много времени. Так что проблема сортировки чисел не так проста, как это может вначале показаться, и именно поэтому ей посвящены целые книги.

Чтобы ускорить процесс сортировки, используют рекурсию, разбивая при этом исходный большой список на более мелкие. Предположим, например, что вместо сортировки одного большого списка вам нужно правильно объединить вместе два небольших уже отсортированных списка. В действительности сделать это довольно просто.

Вы спросите, как это делается? Назовем список, который начинается с меньшего значения, списком А. Второй список назовем списком Б. Итоговый список назовем В. Объединить списки А и Б так, чтобы в итоговом списке все значения также были отсортированы, можно следующим образом. Возьмите первый (наименьший) элемент списка А и перенесите его в список В. Затем сравните второй элемент списка А с первым элементом списка Б. Если он также будет меньше, чем наименьший элемент списка Б, перенесите его в список В. Продолжайте это до тех пор, пока элемент списка Б не окажется меньше, чем элемент списка А. Тогда перенесите в список В элемент списка Б. Теперь сравните следующий элемент списка Б с наименьшим (он будет наименьшим из оставшихся) элементом списка А и снова меньший из этих двух элементов перенесите в список В. Продолжайте этот процесс до тех пор, пока все элементы из списков А и Б не будут перенесены в список В. Возможно, на бумаге этот путь кажется несколько запутанным, но поверьте, что это намного быстрее и эффективнее, чем снова и снова просматривать один и тот же список. Попробуйте реализовать этот процесс на компьютере, и вы в этом убедитесь.

Теперь возникает вопрос, как из одного большого неотсортированного списка сделать два меньших, но отсортированных (чтобы потом получить один общий отсортированный список). Можно разбить большой список пополам и отсортировать каждую половину. Но как отсортировать половину списка? Ее можно также разбить пополам и отсортировать половину от половины. Если продолжать делить списки пополам, рано или поздно они будут разбиты на списки, состоящие из одного или двух элементов. Понятно, что такие списки отсортировать проще простого.

Далее, имея отсортированные списки из одного-двух элементов, двигайтесь в обратном направлении, объединяя списки так, чтобы порядок сортировки не нарушался. На последнем шаге у вас будут два больших отсортированных списка, объединив которые вы получите необходимый результат. Таким образом, используя рекурсию, вы упрощаете задачу, разбивая ее на небольшие подобные подзадачи.

Продемонстрируем сказанное на маленьком примере.

1. **Вот список, состоящий из чисел, которые нужно отсортировать:**
   1 3 7 5 9 2 7
2. **Разобьем его на два списка поменьше:**
   1 3 7 5 9 2 7
3. **ЭТИ СПИСКИ по-прежнему большие, поэтому разобьем их на еще более мелкие:**
   1 3 7 5 9 2 7
4. **Вот такие списки отсортировать очень просто:**
   1 3 5 7 2 9 7

5. **Теперь пойдем в обратном направлении и объединим ранее разделенные списки:**
   1 3 5 7 2 7 9
6. **И наконец, получим окончательный результат:**
   1 2 3 5 7 7 9

Ура! Получилось!

Код, реализующий этот процесс, может выглядеть приблизительно так:

```
СписокЧисел
Sort (СписокЧисел)
{
   if (КоличествоЭлементов(СписокЧисел) == 1)
   {
      return СписокЧисел;
   }
   if (КоличествоЭлементов(СписокЧисел) == 2)
   {
      код для сортировки двух чисел //Путем их сравнения
      return СортированныйСписок;
   }
   //Если обрабатывается этот код, значит, список
   //состоит из более чем двух элементов. Он разбивается
   //пополам, и функция Sort вызывается снова
   Объединить(Sort(первая половина списка чисел),
                  Sort(первая половина списка чисел));
)
```

Задача вычисления факториала числа, которая рассматривалась ранее в главе, также иногда решается путем использования рекурсии. Приведенная ниже функция Factorial во многом похожа на свою предшественницу, но здесь для вычисления факториала вместо цикла for функция вызывает саму себя со значением аргумента, равным числу n-1. Другими словами, используется тот факт, что формулу вычисления факториала n!=n*(n-1)*(n-2)... можно представить в виде n!=n*((n-1)!).

Разумеется, (n-1)! равен (n-1)*((n-2)!). Таким образом, функция, вычисляющая факториал для какого-то числа, может делать это путем умножения этого числа на результат, возвращаемый ею же для числа, на единицу меньшего.

```
//Использование рекурсии для вычисления факториала

#include "stdafx.h"

#using <mscorlib.dll>

using manespace System;

//Функция вычисляет и возвращает значение факториала,
//используя при этом метод рекурсии.
//Факториал числа 1 равен 1. Это просто.
//Если число не равно 1, вызывается функция Factorial
//для числа, меньшего на единицу
int Factorial(int nNumber)
{
  for (i=1; i<=nNumber; i++)
    {
```

```
      nResult *= i;
    }
  //Возвращение результата
  return nResult;
}

//Функция просит пользователя набрать число,
//а затем возвращает это число как результат
int GetNumber()
{
  int nNumber;

  Console::WriteLine(S"Введите число");
  nNumber = Int32::Parse(Console::ReadLine());
  return nNumber;
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  int nNumber;

  //Получение чисел от пользователя до тех пор,
  //пока он не наберет число 0
  while (nNumber = GetNumber())
  {
    //Отображение значения факториала для указанного
    //числа. При этом используется значение,
    //возвращаемое функцией Factorial
    Console::WriteLine(S"Факториал числа {0} равен {1}",
           nNumber.ToString(),
           Factorial(nNumber).ToString());
  }

  //Окончание выполнения программы
  Console::WriteLine(S"Bye");
  return 0;
}
```

Если типаргументов не определен . . .

Чтобы определить, что функция может принимать любое количество параметров, наберите в списке аргументов три точки (...) • Например, приведенный ниже код сообщает компьютеру, что функции может быть передано произвольное количество значений аргументов, и уже ее заботой будет определить их тип и разобраться, что делать с ними дальше.

```
int factorial (...)
{
}
```

Как правило, использовать при определении списка аргументов три точки (...) — плохая идея, поскольку в последующем вы случайно можете передать этой функции совершенно не те данные, которые ей нужны. Это может повлечь за собой самые непредсказуемые последствия. И хотя вы могли видеть, что в некоторых библиотечных функциях, используемых неуправляемыми программами, вместо списка аргументов фигурируют именно эти три точки, старайтесь не пользоваться этим приемом при создании собственных функций.

Значения, установленные по умолчанию

Всем или некоторым аргументам функции можно присвоить значения, используемые по умолчанию. Предположим, например, что есть функция, названная именем foo, для работы которой требуются значения трех аргументов типа integer: a, b и c. Предположим также, что почти при каждом вызове функции в использовании аргумента с не будет необходимости. В этом случае уместно присвоить аргументу с значение, используемое по умолчанию. Это значение будет использоваться каждый раз, когда при вызове функции аргументу с не будет передаваться никакое другое значение. Другими словами, вызывая функцию, вы можете набрать `foo(1,2);` это будет означать, что аргументу a присваивается значение 1, аргументу b — значение 2, а аргумент с будет использовать значение, присвоенное по умолчанию. Если же вы наберете `foo(1,2,3)`, это будет означать, что в данном случае a = 1, b = 2, а c = 3.

Определение значений, используемых по умолчанию, полезно тогда, когда функции используют аргументы, значения которых имеет смысл специально указывать только в особых случаях. Тот, кто будет впоследствии вызывать эти функции, может спокойно проигнорировать аргументы с установленными по умолчанию значениями, что не приведет к сбою в работе этих функций. Однако, если в этом возникнет необходимость, таким аргументам можно передать нужные в данный момент значения.

Чтобы присвоить значения, используемые по умолчанию, укажите их при определении функции в списке аргументов:

```
int foo(int a, int b, int c = 3)
{
}
```

Глава 13

Указатели

В этой главе...

- Причины, по которым имеет смысл использовать указатели
- Как использовать указатели
- Рисование графических элементов с среде .NET
- Создание связанного списка
- Освобождение памяти
- Сборка мусора
- Строки в неуправляемых кодах

Наверняка некоторые из вас думают, что понять принципы работы указателей очень и очень сложно. Возможно, это действительно было так в те времена, когда при изучении процесса программирования больший акцент делался на теорию, чем не практику. Однако сейчас положение изменилось. После того как у вас пройдет первый шок от соприкосновения с таким предметом, как указатели, вы увидите, что они весьма полезны и не так сложны, как казалось раньше.

В этой главе рассматривается, как и зачем использовать указатели. Но, чтобы вам не показалось, что все настолько уж просто (чуть ли не тривиально), в конце главы вы найдете несколько отступлений, посвященных техническим деталям. Кроме того, вы сможете приступить к созданию более интересных приложений .NET, снабженных некоторыми элементами графики.

Почему указатели

В предыдущих главах вы познакомились с множеством технических приемов, позволяющих сохранять и обрабатывать данные. Но поскольку сами данные становятся все более и более сложными, представлять их с помощью именованных переменных становится все труднее и труднее. Что, если вам необходимо, например, сохранить список, состоящий из произвольных по объему фрагментов информации? Предположим, вам нужно отсканировать фотографию, но пока что вы не знаете, каких она будет размеров. Если использовать обычные переменные, необходимо заранее указать, каких размеров фотография будет отсканирована. Если же использовать указатели, программа приобретет большую гибкость и универсальность.

Другой пример: нужно создать программу, позволяющую пользователю рисовать графические элементы. С обычными переменными нужно заранее точно знать, сколько именно фигур нарисует пользователь (или наложить ограничения на количество фигур, которые пользователь может нарисовать). Если в программе для хранения информации о каждой нарисованной фигуре будут использоваться указатели, пользователь сможет нарисовать столько фигур, сколько захочет.

Еще одна причина, по которой указатели настолько полезны, заключается в том, что, хотя сами по себе они невелики, ссылаться они могут на большие объемы информации. Предположим, например, что есть большая база данных, содержащая информацию о пациентах некоторой клиники, и каждая запись в этой базе данных может занимать объемы памяти, исчисляемые тысячами байтов. Если возникнет необходимость изменить порядок расположения записей таким образом, чтобы они были отсортированы, например, по городам, и это можно сделать только путем копирования каждой записи в новую позицию, эта процедура окажется

слишком долгой. Но, если у вас есть указатели, ссылающиеся на каждую запись, отсортировать можно именно их, и сделано это будет намного быстрее. Таким образом, хотя сами записи так и остались на своих местах, изменившийся порядок указателей позволяет увидеть записи в совершенно другом ракурсе.

Указатели и переменные

Чтобы понять, в чем заключается преимущество указателей, нужно разобраться в том, как работают переменные. Все данные, которые обрабатывает компьютер, хранятся в его памяти. Когда переменной присваивается какое-то значение, оно заполняет собой некоторую часть памяти. Когда значение переменной должно быть использовано, оно считывается из памяти. Таким образом, каждая переменная — это всего лишь имя для определенного участка памяти.

Указатели — это те же обозначения каких-то отдельных фрагментов информации, записанных в памяти. Указатель точно так же, как и переменная, указывает на участок памяти. Поэтому каждый раз, когда вы используете переменную, по сути, вы используете указатель.

Различие между переменными и указателями заключается в том, что переменные всегда указывают на один и тот же участок памяти, в то время как один и тот же указатель в разные моменты может указывать на разные участки памяти.

На рис. 13.1 показаны три переменные: `foo`, `bar` и `dribble`, адреса в памяти, на которые они указывают, и принятые ими значения. Переменная **bar,** например, ссылается на адрес 4, в котором сохранено значение 17. Также на рисунке показаны два указателя: `baz` и **goo.** Первый содержит значение 4, которое представляет собой указатель на адрес 4. Таким образом, указатель `baz` может быть использован для получения значения, которое хранится в переменной `bar`. Если поменять значение `baz` с 4 на 8, его можно будет использовать для получения значения, которое хранится в переменной `dribble`.

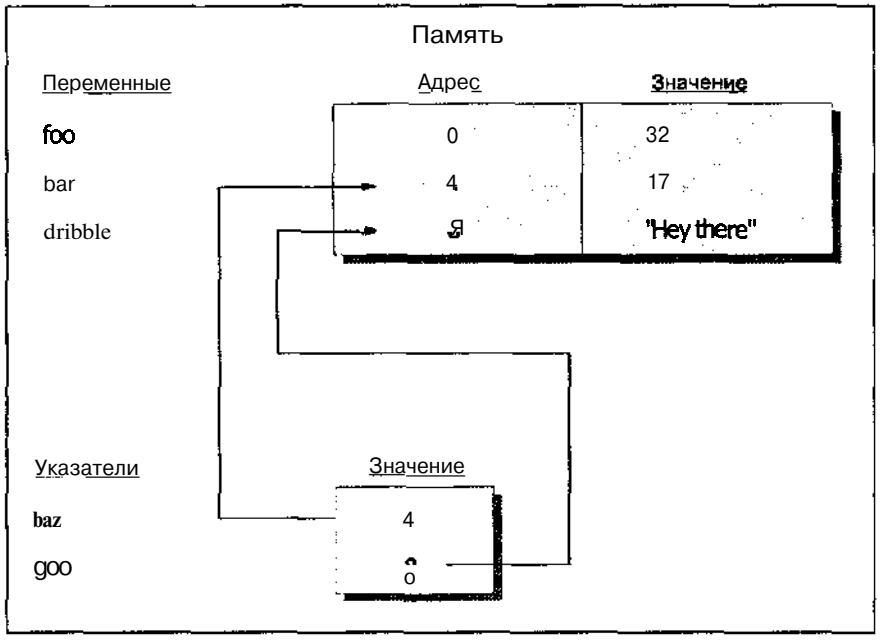


Рис. 13.1. Переменные являются обозначением определенных участков памяти. Указатели ссылаются на различные участки памяти и могут быть использованы для получения доступа к хранящимся там данным

В действительности **указатели** — это один из самых полезных и мощных инструментов, используемых при создании программ, поскольку они обладают большой гибкостью и предоставляют разработчикам большую свободу действий.

Что вы! Указатели — это очень сложно

У неопытных программистов использование указателей вызывает сложности по двум причинам. Во-первых, указатели предоставляют доступ к *двум* различным фрагментам информации. Во-вторых, указатели могут ссылаться на данные, которые не имеют имени.

Информация и ее адрес

Как только что было отмечено, указатели предоставляют доступ к двум различным фрагментам информации.

- ✓ **Значение, сохраненное самим указателем.** Этим значением всегда будет адрес в памяти, где хранится другая информация. Например, если указатель содержит значение 4, это значит, что он указывает на адрес 4.
- ✓ **Значение, на которое** указывает **указатель.** Например, если в памяти под адресом 4 хранится значение 17, указатель, содержащий значение 4, указывает на значение 17.

Значение, которое содержится в указателе, является просто адресом в памяти компьютера. Если вывести на экран значение указателя, вы увидите только число, являющееся адресом. (Поскольку само по себе это число не несет никакой смысловой нагрузки, на экран оно почти никогда не выводится.) Но указатель может также предоставить доступ и к более полезной информации — к значению, которое хранится под тем адресом, на который он указывает. Определение значения, на которое ссылается указатель, называется его *разыменовыванием*.

Например, указатель `baz` содержит значение 4 (см. рис. 13.1). Если вы разыменуете его, то получите значение 17, поскольку именно оно хранится в адресе 4.

Звучит слишком абстрактно? На самом деле в этом нет ничего сложного, ведь даже в повседневной жизни мы сталкиваемся с разыменовыванием. Например, когда вы набираете номер телефона, автоматически вызов направляется именно к тому абоненту, с которым вы хотите поговорить. Другими словами, номера телефонов в вашей записной книге являются указателями на самых разных людей. И когда вы разыменовываете номер телефона (т.е. просто набираете его), вы получаете доступ к нужному человеку.

Безымянные данные

Вторая причина, вызывающая трудности при работе с указателями, заключается в том, что они могут указывать на данные, которые никак не обозначаются (т.е. не имеют никаких имен). На рис. 13.1 вы видели, как используются указатели для получения доступа к значениям различных переменных. Так, например, указатель `baz` ссылался на значение переменной `bar`. Следовательно, вы могли разыменовать указатель `baz`, чтобы получить значение переменной `bar`. В данном случае указатель ссылается на участок памяти, который обозначен определенным именем (этим именем является имя переменной, т.е. `bar`).

Вы можете также попросить компьютер сохранить какую-то часть информации в памяти без присвоения этой информации имени. К этой возможности очень часто обращаются в случае, если возникает необходимость в динамическом использовании памяти. В указателе можно сохранить адрес информации, не обозначенной именем, и затем использовать его для доступа к этим данным.

Не засоряйте свой компьютер

В компьютерах определенная часть памяти отведена для хранения специальной информации. Например, первые 1000 байт используются для хранения информации, объясняющей компьютеру, как реагировать на нажатия клавиш, как отсчитывать время и т.п. Далее какая-то память отведена под файлы операционной системы. Еще какая-то память используется видеокартой.

Если вы знаете адреса этой памяти, то можете получить к ней доступ с помощью указателей и как-то изменить хранящиеся там данные. Иногда это может быть очень полезной возможностью. Например, большинство программ с видеоиграми знают точные адреса памяти, где хранится информация, отображаемая на экране. Благодаря этому они могут в режиме реального времени показывать видеоизображения и сопровождать их специальными эффектами.

Но, с другой стороны, если вы заполните эту память, отведенную для специальных данных, разным информационным мусором, это может привести к возникновению всяких неожиданных проблем. Это состояние называют *трешингом* компьютера (от слова trash - засорять), поскольку в действительности специальная память засоряется ненужными данными. Последствия могут быть самыми разными, но в любом случае нежелательными.

Windows пытается отслеживать корректность выполняемых программами действий. Она обычно может отличить "хорошие" указатели от "плохих" и останавливает выполнение программы до того, как она успеет сделать какую-нибудь глупость. Более подробно эти вопросы рассматриваются ниже в главе.

Испугались? То-то. Будьте аккуратны при использовании указателей.

Связанный список — размер не ограничен

В этом разделе рассматривается пример, в котором доступ к фрагменту памяти будет осуществляться с помощью указателя. Предположим, нужно создать программу, сохраняющую информацию о линиях, которые должны быть нарисованы на экране. Каждый раз, когда пользователь вводит новые координаты, динамически выделяется новая память для их хранения. И затем остается только сохранить адрес этой памяти с помощью указателя.

При этом применяются некоторые маленькие хитрости. В момент создания программы точно не известно, сколько линий захочет нарисовать пользователь. Но сохранить нужно будет информацию обо всех линиях, сколько бы их ни было. Можно, конечно, объявить целое множество указателей (Линия!, Линия2, Линия3 и т.д.) и затем использовать их один за другим. Это не самая удачная идея, поскольку количество объявленных переменных априори должно быть больше количества линий, которые может нарисовать пользователь (что, если он захочет нарисовать, например, 20 000 линий). А кроме того, придется использовать гигантский оператор `switch`, чтобы определить, какой указатель должен быть использован следующим.

Намного разумнее и удобнее использовать нечто, называемое *связанным списком*. Это список элементов, в котором первый элемент указывает на втррой, второй на третий и т.д. Это похоже на поезд: первый выгон тянет за собой второй и т.д.

Чтобы сохранить информацию о целом множестве линий также можно использовать связанный список. Каждый элемент этого списка будет хранить информацию о координатах линии, а также указатель на следующую линию. На рис. 13.2 показан связанный список, состоящий из трех элементов. Каждый элемент содержит данные о своей линии, а также указатель на следующий элемент.

Связанные списки очень удобны и часто используются для сохранения данных об элементах, количество или размеры которых заранее не известны.

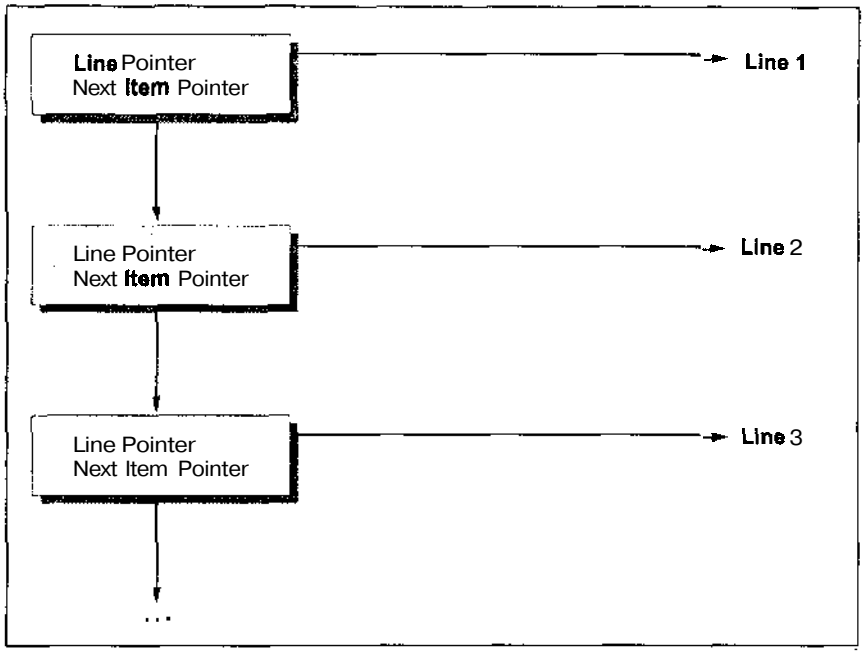


Рис. 13.2. Указатели могут указывать на большие и сложные структуры данных. Элементы этого связанного списка состоят из двух указателей; один указывает на информацию о координатах линии, а второй - на следующий элемент списка

Использование указателей в C++

Указатель, ссылающийся на значения определенного типа данных, создается точно так же, как и переменная того же типа, за исключением того, что перед именем указателя нужно набрать звездочку (*).

Например, чтобы создать указатель, ссылающийся на значения типа integer, наберите такой код:

```
int *pnFoo;
```

Если вы хотите придерживаться соглашений о присвоении имен переменным и указателям, принятым в данной книге, начинайте имена указателей с буквы p (pointer — указатель), затем набирайте префикс, который соответствует тому типу данных, на значения которого указатель будет ссылаться, и далее указывайте само имя. Например, указатель, ссылающийся на значения типа integer, должен начинаться с префикса pn, на значения типа double — с префикса `pdbl` и т.д.

Объявлять тип данных значений, на которые ссылается указатель, необходимо. Этим обеспечивается надежность и безопасность создаваемой программы, поскольку в этом случае компилятор может проверить, не будет ли указатель по ошибке ссылаться на данные не того типа. Ведь указатель — это всего лишь адрес в памяти компьютера. Если вы определяете для него тип данных (скажем, integer), а затем по какой-то причине пытаетесь, например, в память, на которую ссылается этот указатель, записать текстовое значение, компилятор выдаст значение об ошибке. (Это очень хорошая услуга, поскольку именно такие ошибки могут повлечь за собой крайне нежелательные последствия.)

Если вы набрали `int *pnFoo`, то это не значит, что название указателя — `*pnFoo`. На самом деле он называется `pnFoo`, а звездочка не является частью его имени и лишь сообщает компилятору, что это указатель.

Дайте указателю адрес

Если вы только что объявили указатель, он еще ни на что не ссылается. Пока это указатель в никуда. И в таком состоянии его лучше не оставлять. Прежде чем использовать указатель, ему нужно присвоить значение (это значение будет адресом в памяти компьютера, и таким образом указатель начнет на что-нибудь ссылаться).

Довольно часто указатели используются для получения доступа к данным, которые являются значениями переменных. Другими словами, указатели содержат в себе адреса переменных, используемых в той же программе. Когда вы разыменовываете такой указатель, вы получаете значение какой-то переменной.

Чтобы получить адрес переменной, наберите амперсант (`&`) и ее имя. Например, можете набрать такой код:

```
//Создание указателя на значение типа integer
int *pnPosition;
//Создание переменной типа integer
int nXPosition = 3;
//Присвоение указателю адреса переменной nXPosition
pnPosition = &nXPosition;
```

Здесь создается указатель `pnPosition`, который может ссылаться на значения типа integer, и ему присваивается адрес переменной `nXPosition`. Если вы разыменуете этот указатель, то получите значение переменной `nXPosition`.

Указатель может указывать на значения только того типа, которые для него объявлены. Так, если для указателя был объявлен тип integer, он может ссылаться только на переменные типа integer.

Как получить значение, на которое ссылается указатель

Разыменовывание указателя (т.е. получение значения, на которое он ссылается) выполняется очень просто: нужно только перед его именем набрать звездочку (`*`). Например, чтобы получить значение, на которое ссылается указатель `pnPosition`, наберите такой код:

```
//Отображение значения на экране
Console::WriteLine((*pnPosition).ToString());
```

В результате выполнения этого кода на экране отображается значение, полученное при разыменовывании указателя `pnPosition`. Этот указатель содержит адрес переменной `nXPosition`, ее значение равно числу 3, которое и будет отображено на экране.

Пример программы, использующей указатели

А теперь рассмотрим простенькую программу, использующую указатели. (Не огорчайтесь: скоро мы перейдем к более сложным.) В программе объявлена переменная типа integer и указатель, ссылающийся на значение этой переменной. Вот как указателю присваивается адрес этой переменной:

```
pnNumber = &nNumber;
```

По ходу выполнения программы пользователь набирает число, которое сохраняется как значение переменной. Затем это значение отображается на экране с помощью кода

```
nNumber.ToString()
```

Далее это же значение отображается еще раз, но теперь доступ к нему осуществляется посредством разыменовывания указателя pnNumber:

```
(*pnNumber).ToString()
```

Каждый раз, когда вы вызываете функцию (например, `ToString`) для значения, получаемого в результате разыменовывания указателя, всегда берите этот указатель в круглые скобки.

```
//Point

//Объявление и разыменовывание указателя

#include "stdafx.h"

#using <mscorlib.dll>

using manespace System;
//Ожидание, пока пользователь не остановит
//выполнение программы
void HangOut ()
{
  Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                      остановить выполнение программы");
  Console::ReadLine();
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain (void)
#else
int main (void)
#endif
{
  int *pnNumber;
  int nNumber;

  //Присвоение указателю адреса переменной nNumber
  pnNumber = SnNumber;

  //Получение числа от пользователя
  Console::WriteLine(S"Укажите число");
  nNumber = Int32::Parse(Console::ReadLine());

  //Отображение полученного числа

  Console::WriteLine(S"Введенное число {0}",
                             nNumber.ToString());

  //Отображение числа с помощью указателя
  Console::WriteLine(S"Введенное число {0}",
                              (*pnNumber).ToString());

  HangOut();
  return 0;
}
```

Изменение значения, на которое ссылается указатель

Можно не только просматривать значения, на которые ссылается указатель, но и изменять их. Другими словами, можно не только считывать данные из памяти, но и записывать в память новые данные.

Для этого, как и прежде, нужно лишь использовать звездочку (*). Допустим, например, что указатель pnNumber ссылается на значение типа integer (как в предыдущем примере). Чтобы изменить это значение, наберите такой код:

```
*pnNumber = 5;
```

Изменение значений в структурах данных

Если указатель ссылается на структуру, можно изменить отдельный элемент этой структуры. В приведенном ниже примере MyStruct — структура, а poFoo — указатель, который на эту структуру ссылается. Это значит, что для получения доступа к данным, которые в этой структуре хранятся, можно использовать код *poFoo. Например, можно набрать код наподобие

```
(*poFoo).value = 7.6;
```

Продемонстрируем это на таком примере:

```
class  MyStruct
<
public:
  int nData;
  double dblValue;
}
//Создание указателя, ссылающегося на такие структуры
MyStruct *poFoo;
//Создание самой структуры
MyStruct oRecord1;
//Присвоение указателю адреса этой структуры
poFoo = &oRecord1;
//Присвоение значения элементу структуры
(*poFoo).dblValue = 7.6;
```

Использование стрелки

Поскольку набирать код (*указатель).элемент не очень удобно, C++ предлагает упрощенный вариант:

```
//Изменение значения элемента структуры
poFoo->dblValue = 7.6;
```

Код указатель->элемент можно встретить почти во всех программах C++, использующих указатели.

Динамическое выделение памяти

Каждый раз, когда требуется обработать элементы, количество или размеры которых заранее не известны, вы сталкиваетесь с необходимостью *динамического* распределения памяти.

Об этом уже упоминалось, когда рассматривался пример программы, обрабатывающей задаваемые пользователем координаты линий. Количество линий, которые захочет отобразить пользователь, заранее не известно, поэтому при указании координат очередной линии сразу же выделялась память для объекта, представляющего следующую линию.

Динамическое выделение памяти происходит благодаря использованию ключевого слова new. Необходимо только сообщить, для данных какого типа нужно выделить память, и оператор new вернет ссылку на участок памяти этого типа. Если в процессе выполнения программы необходимо выделить память для значения типа integer, наберите такой код:

```
//Создание указателя на значение типа integer
int *pnPosition;
//Выделение памяти для значения типа integer и
//присвоение этого адреса указателю pnPosition
pnPosition = new int;
```

Если нужно выделить память для структуры PointList, наберите

```
//Создание указателя на структуру PointList
PointList *poPoint
//Выделение памяти для структуры PointList и
//присвоение этого адреса указателю poPoint
poPoint = new PointList;
```

Когда вы используете оператор new для создания элемента, указатель запоминает только адрес, указывающий на выделенный фрагмент памяти. При этом следует быть очень внимательным, чтобы по ошибке не потерять необходимый адрес, на который ссылается указатель, так как если вы сохраните новый адрес, прежний восстановить будет уже невозможно. Продемонстрируем это на таком примере:

```
//Забывчивая программа
//Создание указателя типа integer
int *pnPosition;
//Выделение памяти для значения типа integer
pnPosition = new int;
//Сохранение значения по этому адресу
*pnPosition = 3;
//Выделение памяти для нового значения
pnPosition = new int;
```

Последней строкой этого кода выделяется память для нового значения типа integer. Адрес этого нового участка памяти будет сохранен в указателе pnPosition. Но что случится со значением 3, сохраненным по адресу, на который до этого ссылался указатель pnPosition? Это значение по-прежнему будет храниться в памяти, но, поскольку его адрес потерян (указатель уже ссылается на совсем другой участок памяти), получить к нему доступ не будет никакой возможности.

Если вы по какой-либо причине потеряете адрес, который для каких-то данных был выделен динамически, эти данные останутся на своем месте, но найти их уже будет невозможно. Это явление называется *утечкой памяти*. Потерянные данные так и будут занимать память, и вы не сможете их удалить до тех пор, пока программа не завершит работу. (В среде .NET есть множество возможностей, позволяющих справиться с этой проблемой, о чем речь пойдет несколько ниже.)

Перед тем как использовать оператор new, убедитесь, что данные, адрес которых пока еще хранится в указателе, более не нужны. Кроме того, если вы хотите создать действительно качественную программу, перед использованием оператора new освободите память от тех данных, которые вам больше не нужны, с помощью оператора delete (более подробно об операторе delete речь идет ниже в главе).

Внимание — графика

Теперь, когда у вас есть общее представление об указателях, можно приступить к созданию более интересных программ .NET. В этом разделе коснемся графических возможностей среды .NET. Так же как до этого вы использовали функции Console для считывания и отображения текста, теперь вы будете использовать функции Graphics для отображения на экране графики.

Но, прежде чем отобразить что-то на экране, нужно создать графический объект. Делается это с помощью кода, подобного приведенному ниже,

```
Form *poForm = new FormO;
Graphics *poGraphics = poForm->CreateGraphics();
```

Сплошные указатели! Когда вы хотите что-то нарисовать, сделать это можно в окне (не в том, из которого вы смотрите на улицу, а в окне, похожем на те, что используются в среде Windows). Один из самых простых способов сделать это — использовать класс Form. Затем можно вызвать процедуру CreateGraphics, для того чтобы создать графический элемент, называемый объектом Graphics. Обратите внимание, что для доступа к объектам Form и Graphics используются указатели, а для вызова методов этих объектов — символы ->.

Перед тем как что-то нарисовать, нужно сделать форму видимой с помощью кода, подобного этому:

```
poForm->Show();
```

Затем можно вызывать графические функции .NET, часть из которых показана в табл. 13,1. Нарисовав все, что нужно, вызовите метод Dispose:

```
poGraphics->Dispose();
```

Таблица 13.1. Некоторые графические команды

Метод	Пример	Описание
Clear	poG->Clear(Color::Red)	Заполняет весь графический элемент определенным цветом (см, также табл. 13.2)
Dispose	poG->Dispose()	Освобождает память, занятую информацией о графических объектах, Метод вызывается после завершения этапа рисования
DrawEllipse	poG->DrawEllipse (poPen, 10, 20, 150, 200)	Рисует эллипс, используя указанное перо. (Перья описываются несколько ниже.) Числовые параметры - это координаты x и y, которые определяют верхний левый угол эллипса, а также значения ширина и высота. В действительности x, y, ширина и высота определяют положение и размеры прямоугольника, внутри которого будет нарисован эллипс. Если высоту и ширину сделать равными, получится круг
DrawLine	poG->DrawLine(poPen, 10, 20, 160, 220)	Рисует прямую линию, соединяющую две точки. В данном случае линия будет соединять точку с координатами (10, 20) и точку с координатами (160,220)
DrawRectangle	poG->DrawRectangle (poPen, 10, 20, 150, 200)	Рисует прямоугольник. Два первых числовых параметра определяют верхний левый угол прямоугольника, два следующих - его ширину и высоту
DrawString	poG->DrawString fS"Test", poFont, poBrush, 10, 20)	Рисует текст указанными шрифтом, кистью и начиная с указанной позиции, Шрифты и кисти будут описаны ниже

Метод	Пример	Описание
FillEllipse	poG->FillEllipse (poBrush, 10, 20, 150, 200)	Рисует эллипс, закрашенный указанной кистью
FillRectangle	poG->FillRectangle (poBrush, 10, 20, 150, 200)	Рисует прямоугольник, закрашенный указанной кистью

Все цвета радуги

Для каждого графического элемента необходимо указывать цвет, которым он будет нарисован. Осуществляется это с помощью структуры Color. Всего возможны два варианта: либо использовать один из заранее заданных цветов, либо создать свой цвет, вызвав метод FromArgb.

В табл. 13.2 показан список предопределенных цветов. Единственное предостережение: не принимайте пищу во время отображения объектов с такими цветами, как PapayaWhip или LemonChiffon. Иначе вы рискуете испортить себе аппетит и вызвать расстройство желудка.

Таблица 13.2. Предопределенные цвета

AliceBlue	DarkOrange	Khaki
AntiqueWhite	DarkOrchid	Lavender
Aqua	DarkRed	LavenderBlush
Aquamarine	DarkSalmon	LawnGreen
Azure	DarkSeaGreen	LemonChiffon
Beige	DarkSlyteBlue	LightBlue
Bisque	DarkSlyteGray	LightCoral
Black	DarkTurquoise	LightCyan
BlanchedAlmond	DarkViolet	LightGoldenrodYellow
Blue	DeepPink	LightGray
BlueViolet	DeepSkyBlue	LightGreen
Brown	DimGray	LightPink
BurlyWood	DodgerBlue	LightSalmon
CadetBlue	Firebrick	Light5eaGreen
Chartreuse	FloralWhite	LightSlateGray
Chocolate	ForestGreen	LightSteelBlue
Coral	Fuchsia	LightYellow
Cornflower	Gainsboro	Lime

Cornsilk	GhostWhite	LimeGreen
Crimson	Gold	Linen
Cyan	Goldenrod	Magenta
DarkBlue	Gray	Maroon
DarkCyan	Green	MediumAquamarine
DarkGoldenrod	GreenYellow	MediumBlue
DarkGray	Honeydew	MediumOrchid
DarkGren	Hotpink	MediumPurple
DarkKhaki	IndianRed	MediumSeaGreen
DarkMagenta	Indigo	MediumSlateBlue
DarkOliveGreen	Ivory	MediumSpringGreen
MediumTurquoise	PaleVioletRed	SlateBlue
MediumVioletRed	PapayaWhip	SlateGray
MidnightBlue	PeachPuff	Snow
MintCream	Peru	SpringGreen
MistyRose	Pink	SteelBlue
Moccasin	Plum	Tan
NavajoWhite	PowderBlue	Teal
Navy	Purple	Thistle
OldLace	Red	Tomato
Olive	RosyBrown	Transparent
OliveDrab	RoyalBlue	Turquoise
Orange	SaddleBrown	Violet
OrangeRed	Salmon	Wheat
Orchid	SandyBrownSeaShell	White
PaleGoldenrod	Sienna	WhiteSmoke
PaleGreen	Silver	Yellow
PaleTurquoise	SkyBlue	YellowGreen

Метод FromArgb получил свое название от слов Alpha-Red-Green-Blue (альфа-красный-зеленый-голубой). Любой другой цвет из диапазона, воспринимаемого человеческим глазом, может быть получен путем смешивания в разных пропорциях этих трех основных цветов.

Количественное вхождение основного цвета в состав производного измеряется в пределах от О (означает полное отсутствие) до 255 (означает максимальную интенсивность). Например, если взять красного 255 единиц, зеленого 0 и голубого 0, то получим ярко-красный цвет. Если красного взять 128, зеленого 128 и голубого 128, получим светло-серый цвет. Я бы с удовольствием рассказал о всех возможных цветах и оттенках, которые можно получить, комбинируя основные цвета, но тогда ни на что другое в книге не хватило бы места. (Ради интереса попробуйте посчитать, сколько разных цветов можно получить с помощью этой системы.)

Если вы хотите точно узнать, в каких пропорциях входят основные цвета в состав нужного вам цвета, откройте какую-нибудь графическую программу, например Adobe Photoshop. В окне цветоподборщика (color picker) найдите нужный вам цвет. В этом же окне будут показаны параметры выбранного цвета.

Чтобы использовать голубой цвет, наберите

```
Color::Blue
```

Если же вам нужен какой-то специфический цвет, наберите код наподобие этого:

```
Color::FormArgb(255, 128, 0);
```

Пример использования структуры Color будет показан несколько ниже в этой главе.

В приведенном выше примере указаны значения для красного, зеленого и голубого цветов. Но метод FromArgb может принимать значение еще одного параметра — альфа, которым обозначается прозрачность графического элемента. Поэкспериментируйте с этой возможностью — быть может, она вам пригодится.

При использовании объекта Color не нужно набирать оператор new для создания нового объекта (как это делается в случае с объектом Graphics). Вместо этого наберите символы : :, чтобы указать один из предустановленных цветов или вызвать метод FromArgb.

Перья для рисования

Рисуя линию (метод DrawLine) или фигуру (методы DrawEllipse и DrawRectangle), вы используете объект Pen (перо). Этот объект переносит в мир компьютерной графики свойства обычного пера или карандаша. Он рисует линии определенного цвета и определенной толщины. Чтобы создать красный карандаш, который будет рисовать линию толщиной в один пиксель, наберите такой код:

```
poPen = new Pen(Color::Red);
```

А этим кодом создается синий карандаш с толщиной следа в пять пикселей:

```
poPen = new Pen(Color::Blue, 5);
```

Цвет и толщину объекта Pen можно изменять в процессе выполнения программы. Это делается путем изменения значений свойств Color и Width этого объекта. Например:

```
poPen->Wigth = 12;

poPen->Color = Color::SaddleBrown;
```

Графические функции среды .NET иногда обозначаются как GDI+.

Кисточка для раскраски

Если вы хотите нарисовать заполненную чем-то фигуру (используя, например, методы FillEllipse или FillRectangle), вместо объекта Pen используется объект Brush (кисточка). Эффект применения этого объекта подобен эффекту использования обычной кисточки для рисования. В электронном варианте есть два вида кисточек: SolidBrush и TextureBrush. Кисточка SolidBrush заполняет фигуру сплошным цветом, в то время как TextureBrush может заполнить фигуру каким-нибудь рисунком или изображением.

Чтобы создать новую кисточку, наберите такой код:

```
poBrush = new SolidBrush(Color::Red);
```

Для изменения цвета кисточки измените свойство Color этого объекта:

```
poBrush->Color = Color::Thistle;
```

Шрифты

Когда вы применяете метод DrawString для отображения текста на экране, необходимо выбрать используемый шрифт. Для этого наберите код наподобие

```
poFont = new Font("Arial", 12);
```

Значением первого параметра определяется сам шрифт, а второго — его размер. Вообще сушествует множество различных шрифтов. Примеры некоторых из них показаны на рис. 13.3.

Шрифт	*Пример*
Arial	Hello 123
Courier New	Hello 123
Times New Roman	Hello 123
Symbol	Ηελλο 123
Verdana	Hello 123
WingDings	

Рис. 13.3. Наиболее часто используемые шрифты

Можно использовать вспомогательную программу Character Map, чтобы увидеть, как одни и те же буквы отображаются в разных шрифтах. Это очень полезная возможность, особенно когда речь идет о шрифтах Symbol и WingDings. Если вам скучно и вы не знаете, чем заняться, можете составить с помощью этих шрифтов многозначительное послание, состоящее из разных значков и символов, отправить его своим друзьям и посмотреть, как они на него отреагируют.

Займемся рисованием

Теперь, когда вы познакомились с основами создания графических элементов в среде .NET, можно приступать к использованию этих возможностей на практике. Ниже приведен код небольшой программы, которая рисует на экране линии и некоторые геометрические фигуры. При этом используются объекты, представляющие перья, кисти и шрифты.

```
//Graphics

//Демонстрация графических возможностей .NET

#include "stdafx.h"

#using <mscorlib.dll>
#using <System.Windows.Forms.dll>
#using <System.dll>
#using <System.Drawing.dll>
using manespace System;
using manespace System::Drawing;
using manespace System::Windows::Forms;

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  //Создание структур для отображения графики
  Form *poForm = new Form();
  Graphics *poGraphics = poForm->CreateGraphics();

  //Создание красного пера
  Pen *poPen = new Pen(Color::Red);

  //Отображение окна для рисования
  poForm->Show();

  //Отображение линии
  poGraphics->DrawLine(poPen, 10, 10, 120, 150);

  //Изменение свойств пера и
  //отображение новой линии
  poPen->Color = Color::FormArgb(100, 255, 200);
  poPen->Wigth = 5;
  poGraphics->DrawLine(poPen, 120, 150, 10, 150);

  //Отображение эллипса с использованием того же пера
  poGraphics->DrawEllipse(poPen, 10, 150, 40, 60) ;


  //Создание новой кисти
  SolidErush *poBrush = new SolidBrush(Color::Tomato);
  //Отображение прямоугольника, закрашенного этой кистью
  poGraphics->FillRectangle(poBrush, 50, 150, 40, 60);

  //Создание шрифта
  Font *poFont = new Font("Helvetica", 22);
  //Отображение текста этим шрифтом
  poGraphics->DrawString(S"Hello", poFont, poBrush, 200, 200);

  //Освобождение памяти
```

```
  poGraphics->Dispose();

  //Ожидание, пока пользователь не закроет окно с графикой
  Application::Run(poForm);
}
```

В этой программе необходимо обратить внимание на следующее. Во-первых, на наличие нескольких новых строк с ключевыми словами `#using` и `using namespace` в начале программы. Они нужны потому, что для использования графики и форм в среде .NET требуется подключение некоторых дополнительных файлов.

Во-вторых, на использование кода `Application::Run` в конце программы. Каждый раз, когда вы используете форму, необходимо набирать код, указывающий системе, что не нужно прекращать выполнение программы до тех пор. пока пользователь не закроет окно этой формы. Это намного удобнее применения функции HangOut, которая постоянно использовалась в предыдущих примерах. Форма отображается на экране, пользователь просматривает все, что на ней нарисовано, а когда щелкает на кнопке закрытия формы, программа прекращает работу.

Связанные списки и графика

После того как вы немного поупражнялись с указателями (вся предыдущая программа была построена с их использованием), перейдем к рассмотрению более типичных способов их применения. В этом разделе внимание будет уделено вопросу создания с помощью указателей связанных списков. Рассмотрено это будет на таком примере: пользователь набирает координаты произвольного количества точек, для хранения которых используется связанный список. Когда он закончит, эти точки соединяются между собой с помощью графических команд среды .NET.

Как эта программа работает

В программе создается связанный список. Каждый элемент этого списка представляет собой структуру, состоящую из координат точки, а также из указателя, который ссылается на следующий элемент этого списка. Указатель последнего элемента списка имеет значение, равное нулю. Указатели с таким значениями называются *пустыми* (или *нулевыми)*.

Итак, элемент связанного списка должен содержать в себе значения координат точки и указатель на следующий элемент. Ниже показан код, которым объявляется структура `PointList`, содержащая в себе два значения типа integer и указатель на такую же структуру.

```
class PointList
{
public:
  int nX;
  ir.t nY;
  PointList *poNext;
};
```

Программа должна содержать в себе коды, необходимые для добавления в список новых элементов, и коды, позволяющие находить нужную информацию в этом списке. Для этого нужны три дополнительных указателя. Первый будет ссылаться на первый элемент списка; таким образом, вы всегда будете знать, где список начинается. Это будет как бы точкой отсчета, дающей ключ ко всем остальным элементам списка:

```
PointList *poPoints = C;
```

Второй указатель будет каждый раз ссылаться на новый элемент, который добавляется к списку:

```
PointList *poNew;
```

Третий указатель будет содержать ссылку на последний элемент списка:

```
PointList *poLast;
```

Он также потребуется при добавлении в список нового элемента, так как последний элемент списка (теперь уже бывший последний) должен ссылаться на вновь добавляемый элемент.

Теперь рассмотрим, как связанный список будет использоваться для хранения вводимой пользователем информации. Вначале выделяется память для нового элемента списка:

```
poNew = new PointList;
```

Если это будет первый элемент списка, указатель poPoints должен ссылаться именно на него. (Помните, что poPoints всегда указывает только на первый элемент?)

```
if (!poPoints)
{
  //Если poPoints пока еще не ссылается ни на какой
  //элемент, пускай он ссылается именно на него
  poPoints = poNew;
}
```

Оператор if проверяет, не является ли указатель poPoints нулевым. Если он нулевой, выражение !poPoints истинно и инструкция poPoints = poNew выполняется. Если же указатель poPoints уже ссылается на какой-то элемент, новый элемент добавляется в конец списка:

```
poLast->poNext = poNew;
```

Смысл выполняемых этим кодом действий следующий: определяется элемент, на который ссылается указатель poLast, и составляющей poNext этого элемента присваивается адрес создаваемого элемента, на который в данный момент ссылается указатель poNew.

И наконец, обновляется значение указателя poLast так, чтобы теперь он ссылался на только что созданный объект, и вызывается функция NewPoint, которая принимает значения координат новой точки:

```
poLast = poNew;
NewPoint(poNew);
```

После того как пользователь укажет координаты всех точек, программа отобразит их на экране и соединит между собой прямыми линиями. Этот процесс начнется с отображения первой точки, после чего будет проведена линия к следующей точке, затем к следующей и т.д. Процесс рисования завершится, как только программа дойдет до нулевого указателя, поскольку нулевой указатель принадлежит последнему элементу списка.

```
PoLast = poPoints;
while (poLast)
{
  //Отображение линий
  DrawPoint(poGraphics, poLast);
  //Переход к следующему элементу
  poLast = poLast->poNext;
}
```

После того как все линии будут нарисованы, необходимо очистить память от ненужной более информации (о том, для чего это нужно делать, речь идет ниже в этой главе):

```
PoLast = poPoints;
while (poLast)
```

```
{
  PointList *poNext;
  poNext = poLast->poNext;

  //Освобождение памяти
  delete poLast;

  //Переход к следующему элементу
  poLast = poNext;
}
```

Код программы

Ниже приведен код всей программы, работа которой обсуждалась в предыдущем разделе.

```
//Draw
//Применение связанного списка для сохранения
//координат произвольного количества точек
//и их отображение с использованием GDI+

#include "stdafx.h"

#using <mscorlib.dll>
#using <System.Windows.Forms.dll>
#using <System.dll>
#using <System.Drawing.dll>

using manespace System;
using manespace System::Drawing;
using manespace System::Windows::Forms;

//Создание структуры, представляющей элемент связанного списка
class PointList
{
public:
  int nX;
  int nY;
  PointList *poNext;
};

//Получение координат новой точки
void NewPoint(PointList *poNew)
{
  //Чтение координат X и Y
  Console::WriteLine(S"Введите координату X");
  poNew->nX = Int32::Parse(Console::ReadLine());
  Console::WriteLine(S"Введите координату Y");
  poNew->nY = Int32::Parse(Console::ReadLine());
  //Присвоение указателю нулевого значения
  poNew->poNext = C;
}

//Отображение точек на экране
void DrawPoint(Graphics *poGraphics, PointList *poPoint)
{
  //Есть ли точки для отображения?
  If (poPoint->poNext)
```

```
  {
  //Создание пера для рисования
  Pen *poPen = Pen(Color::Red);
  //Рисование линии от текущей точки к следующей
  poGraphics->DrawLine(poPen, poPoint->nX, poPoint->nY,
                  poPoint->poNext->nX, poPoint->poNext->nY);
  }
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  //Указатель для первого элемента списка
  PointList *poPoints = 0;
  //Указатель для последнего созданного элемента
  PointList *poLast;
  //Указатель для нового элемента списка
  PointList *poNew;
  //Указатель для принятия ответа от пользователя
  String *pszMore;

  //Структуры для отображения графики
  Form *poForm = new Form();
  Graphics *poGraphics = poForm->CreateGraphics();

  while (!fFinished)
  <
    //Выделение памяти для нового элемента
    poNew = new PointList;
    if (!poPoints)
    {
      //Если это первый элемент, присвоение его
      //адреса указателю poPoints
      poPoints = poNew;
    )
    else
    {
      //Присоединение элемента в конец списка
      poLast->poNext = poNew;
    }
    //Обновление значения указателя poLast
    poLast = poNew;
    //Принятие от пользователя координат новой точки
    NewPoint(poNew);
    //Хочет ли пользователь ввести координаты для
    //следующей точки?
    Console::WriteLine("Нажмите y для определения
                                  следующей точки");
    pszMore = Console::ReadLine();
    if (!pszMore->Equals(S"y"))
    {
```

```
        fFinished = true;
      }
    }

    //Отображение окна для рисования
    poForm->Show();

    //Отображение точек на экране
    PoLast = poPoints;
    while {poLast}
    {
      //Отображение линий
      DrawPoint(poGraphics, poLast);
      //Переход к следующему элементу
      poLast = poLast->poNext;
    }

    //Освобождение памяти, которая использовалась для
    //отображения графики на экране
    poGraphics->Dispose();

    //Освобождение памяти, занятой элементами списка
    PoLast = poPoints;
    while (poLast)
    {
      PointList *poNext;
      poNext = poLast->poNext;

      //Освобождение памяти
      delete poLast;

      //Переход к следующему элементу
      poLast = poNext;
    }

    //Ожидание, пока пользователь не закроет окно формы
    Application::Run(poForm);

}
```

Использование ссылочных аргументов

Как уже отмечалось, изменять значения глобальных переменных в процессе выполнения функций - плохая практика. Намного разумнее использовать функции для возвращения значений и затем уже как-то изменять эти значения вне функций. {В будущем это поможет вам избежать многих ошибок и без труда понимать смысл выполняемых функциями действий.)

Если же функция должна изменить значения нескольких элементов или элементов, принадлежащих какой-то структуре, передайте ей в качестве параметра указатель, ссылающийся на нужный элемент. В процессе своего выполнения функция может разыменовать указатель и изменить значение элемента. Это намного лучше, чем изменять значения глобальных переменных {или переменных, область видимости которых распространяется за пределы этой функции), поскольку при передаче функциям значений указателей вы точно определяете, какие элементы будут изменены. Таким образом, не вникая в подробности выполнения функции, вы или другие программисты сможете сразу же определить, значения каких элементов могут быть изменены этой функций.

Другой возможностью является использование *ссылочных аргументов*. При этом функции передаются указатели на аргументы (т.е. их адреса в памяти), но не сами аргументы.

Поскольку в распоряжение функции попадают адреса элементов, она может изменять их значения непосредственно.

Использовать ссылочные аргументы удобнее, чем передавать в качестве аргументов значения указателей, поскольку в этом случае не нужно разыменовывать указатели внутри функции. Чтобы определить, что функция будет использовать ссылочный аргумент, наберите в списке аргументов перед названием этого аргумента символ &:

```
int Factorial(int &Number)
{
}
```

Теперь все изменения, выполненные функцией в отношении переменной `Number`, будут иметь постоянный характер, и это будет выглядеть так, как будто значение переменной изменяется за пределами функции.

Вопросы безопасности

Как известно, большие полномочия сопровождаются большей ответственностью, что справедливо и в отношении указателей. Язык C++ обладает большими возможностями, и с помощью указателей программист получает прямой доступ к памяти компьютера. Это означает, что использовать их нужно очень осторожно. В этом разделе затрагиваются вопросы, о которых нужно помнить при работе с указателями, а также описываются возможности среды .NET, которые помогут вам избежать ошибок.

Освобождение памяти

Если вы видите, что в выделенной ранее памяти больше нет необходимости, сразу же освобождайте ее. Таким образом, ваши программы не будут занимать больше памяти, чем им требуется.

Для освобождения памяти используется команда `delete`, действие которой в точности противоположно действию оператора new:

```
//Выделение памяти для элемента PointList
PointList *pnPoint = new PointList;
//Освобождение только что выделенной памяти
delete pnPoint;
//Обнуление указателя
pnPoint = 0;
```

Обратите внимание, что при обнулении указателя не набирается звездочка (`*`), поскольку обнуляется значение самого указателя, а не участок памяти, на который он ссылается.

Когда вы удаляете какой-то элемент, значение указателя не изменяется. Однако тех данных, на которые ссылается указатель, в памяти больше нет. Таким образом, указатель по-прежнему ссылается на какой-то адрес, но этот адрес уже пуст. Если вы его сейчас разыменуете, то получите совершенно бессмысленное значение. Поэтому каждый раз, удаляя какой-то элемент, обнуляйте указатель, который на него ссылался. Благодаря этому вы будете точно знать, что все указатели вашей программы ссылаются на реальные данные.

Общее нарушение защиты

Запутавшись с указателями, вы можете столкнуться с серьезными проблемами. Такого рода ошибки обозначают термином GPF (General Protection Fault — *общее нарушение защиты).* Иногда это означает, что придется закрывать некоторые приложения или даже перезагружать компьютер. Windows отслеживает возникновение ошибок GPF и, как только их находит, останавливает выполнение программы. На рис. 13.4 показано, как может выглядеть диалоговое окно с сообщением о нарушении общей защиты.

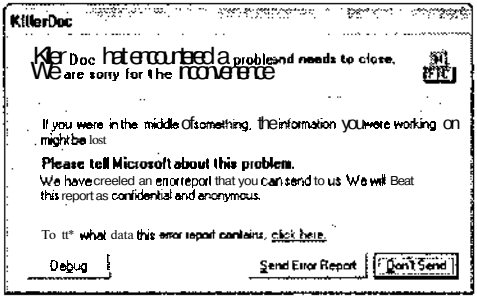


Рис. 13.4. Если Windows столкнется с ошибками GPF, она закроет вашу программу и отобразит окно с предупреждением

Неизменяемые ссылочные аргументы

Если вы передаете функциям в качестве аргументов большие структуры данных, это может занять слишком много времени, поскольку при этом компьютер должен создать новые копии всех элементов, входящих в эти структуры. Чтобы ускорить этот процесс, передавайте структуру как ссылочный аргумент. В этом случае передается только информация об адресе структуры, которая сама по себе занимает очень мало места.

Но теперь функция получает возможность изменять данные, хранящиеся в структуре. Предположим, вам этого не нужно. Чтобы решить этот вопрос, можно обозначить структуру как *неизменяемый ссылочный аргумент.* При этом вы как бы сообщаете компилятору: "Я делаю это только для ускорения работы программы. Не изменяй данные, доступ к которым получает функция".

Вот пример объявления неизменяемого ссылочного аргумента:

```
int DrawIt(const PointList &MyPoint)
{
}
```

Ниже перечислен ряд причин, по которым работа **с** указателями может обернуться маленькой трагедией.

- ✓ **Запись информации на** нулевой **адрес.** Например, если указатель `foo` обнулен и вы набираете код `*foo = x`, то это означает, что вы отправляете значение x "туда, не знаю куда". Windows это не понравится. Это вообще никакой системе не понравится. Результат таких действий будет плачевным.
- ✓ **Запись информации не на тот адрес.** Предположим, например, что указатель `foo` ссылается на первый элемент списка, а указатель `bar` — на последний. Если вы

используете bar в том месте, где должны были использовать foo, можете столкнуться с совершенно неожиданными последствиями.

- ✓ Освобождение памяти без обнуления указателей. Например, после освобождения памяти вы преспокойно записываете по тому же адресу целую кипу новых данных (при том, что теперь эта память отведена для совершенно другой информации). Результат будет ужасен.

Старайтесь не делать таких глупостей, иначе проблем не оберешься.

Генеральная уборка

В программе Draw было написано немало кодов для того, чтобы очистить память, которая использовалась в процессе ее выполнения. Если хотите, можете воспользоваться возможностью среды .NET, называемой *сборкой мусора (garbage collection),* которая автоматически выполнит всю "грязную" работу. Если при объявлении объекта вы укажете, что память после него нужно освобождать автоматически, среда .NET будет отслеживать его в процессе выполнения программы. Как только .NET увидит, что объект больше не используется, она освободит занимаемую им память. (Поскольку сборка мусора — дело не только грязное, но и отнимающее много времени, .NET занимается этим тогда, когда для этого появляется удобный момент.)

Чтобы активизировать для объекта эту возможность, при его объявлении наберите в начале строки __gc, например:

```
__gc class PointList
{
public:
  int nX;
  int nY;
  PointList *poNext;
}
```

Универсальные указатели

На первый взгляд может показаться, что использовать указатели, которые могут ссылаться на все, что угодно, - вещь удобная. При объявлении таких указателей вместо слова, обозначающего тип данных, набирается слово void (что означает "пустой тип данных"), но хотя эти указатели очень удобны (поскольку вы можете использовать их для получения доступа к значениям любых типов), они также и очень опасны (поскольку компилятор не может отследить правильность их использования).

Вот как можно создать такой указатель (но потом не говорите, что вас не предупреждали):

```
//Пусть pOff ссылается на что угодно
void *pOff;
```

Указатели void опасны (и одновременно удобны) именно тем, что для них не определен тип данных и поэтому возникает реальная возможность неправильного использования памяти.

Предположим, что в рассмотренном ранее примере вы используете для создания связанного списка указатели void и случайным образом вместо структур с информацией о координатах точек добавляете в список числа, строки, записи о сотрудниках и еще неизвестно что. Компилятор никак не сможет определить, что здесь что-то не так. Но пользователи вашей программы непременно это заметят.

Подведем итог: используйте указатели void только тогда, когда без этого никак не обойтись.

Кое-что о строках

Если вы занимались программированием раньше, то должны знать, что термином *строки* обозначается текстовая информация, точнее, наборы следующих друг за другом символов. В .NET есть встроенный класс `String`, который предназначен для работы со строками. Если же вы пишете неуправляемый код, вам придется делать все самому, используя указатели типа `char`. Но не расстраивайтесь: есть множество библиотечных функций, которые специально созданы для обработки текстовой информации.

Строки хранятся в памяти компьютера как массивы расположенных друг за другом символов, заканчивающиеся нулевым значением. (Из-за этого нулевого байта в конце строки они называются также *строками с завершающим нулем.)* Доступ к текстовой информации можно получить с помощью указателей, которые ссылаются на первый символ строки. Чтобы создать строку, наберите такой код:

```
char *szMyString = "романтическое путешествие";
```

Этим кодом создается строка, содержащая текст "романтическое путешествие". Указатель `szMyString` ссылается на эту строку. Чтобы отобразить эту строку на экране, наберите

```
cout << szMyString;
```

Указания для указателей

Здесь вы найдете несколько советов и напоминаний, которые помогут вам избежать проблем при работе с указателями,

- ✓ Указатель содержит в себе лишь адрес, указывающий на некоторые данные, сохраненные в памяти. Если вы как-то изменяете значения указателей (добавляете, вычитаете и т.п.), вы изменяете только адреса, но не то, что находится по этим адресам. Обычно, это не является вашей целью: вы хотите обрабатывать именно те данные, на которые ссылаются указатели. Для этого указатели нужно разыменовывать.
- ✓ `*pFoo` — это не имя указателя. Его имя — `pFoo`. А кодом `*pFoo` указатель разыменовывается.
- ✓ Если указатель `pnFoo` имеет тип integer, можете набирать `*pnFoo` в любом месте программы, где может быть использовано значение типа integer. Если указатель `poFoo` ссылается на данные какого-нибудь типа *X*, можете набирать `*poFoo` везде, где значения типа *x* могут быть использованы. Это значит, что вы можете набирать `*poFoo = jupiter;` или `jupiter = *poFoo;` и т.п.
- ✓ Если вы динамически выделяете какую-то память (т.е. в процессе выполнения программы), убедитесь, что адрес этой памяти сохранен как значение указателя. В противном случае вы никак не сможете получить доступ к этой памяти.
- ✓ После того как вы освобождаете память, выделенную динамически, значение указателя, ссылающегося на эту память, никак не изменяется и он по-прежнему на нее указывает. Чтобы избежать ненужных проблем, всегда обнуляйте соответствующие указатели после использования команды `delete`.
- ✓ Если вы в больших объемах используете динамически выделяемую память, подключите возможность среды .NET, называемую сборкой мусора.
- ✓ Если вы устали и "зависать" начинает уже ваша черепная коробка, сделайте перерыв, покурите сигару или позвоните любимой.

Вам наверняка будет интересно, какие библиотечные функции предназначены для работы со строками и что они могут делать. Их будет легче найти, зная о том, что названия почти всех из них начинаются с букв `str`. Например, функция `strlen` возвращает количество символов, из которых состоит строка.

В C++ есть также такой объект, как текстовый класс ANSI, который может помочь вам в работе со строками.

Выделяя память для строк, не забывайте о том, что они должны заканчиваться нулевым значением, которое занимает один байт памяти. Убедитесь, что вы включили этот байт в общее количество байтов, выделяемых для размещения текстового значения. В противном случае возникнут проблемы с использованием памяти.

Поскольку можно набирать код `char *` для создания ссылок на текстовые данные, вы можете в полной мере использовать все возможности указателей для обработки текстовой информации. Напомним, что в C++ строки должны заканчиваться нулевым байтом. Именно по нему библиотечные функции определяют, где находится конец строки.

Если необходимо, например, отобразить все символы строки поочередно (по одному за раз), можно использовать значение указателя этой строки, увеличивая его каждый раз на единицу. Если указатель ссылается на какой-то символ строки, добавив к его значению число 1, вы получите адрес следующего символа строки. Например:

```
//Строка
char *szMyString = "hello world";
//Еще один указатель
char *szCharPtr;
//Изменение первого символа строки
szCharPtr = szMyString;
*szCharPtr = "j";
//Теперь переходим к следующему символу
//Для этого значение указателя увеличиваем на 1
szCharPtr++;
//Изменяем второй символ
*szCharPtr = "o";
//Теперь строка выглядит как "jollo world"
//Отображение строки на экране
cout << szMyString;
```

Подведем итог

Вы уже знаете об указателях достаточно много. Это просто переменные, значениями которых являются адреса различных данных, сохраненных в памяти компьютера. Указатели используются для получения доступа к памяти, которая выделяется динамически (это нужно в тех случаях, когда вы заранее не знаете, какой объем памяти должен быть выделен). Также указатели используются для создания связанных списков (они нужны тогда, когда заранее не известно, сколько элементов будет использовано). Есть еще много задач, которые решаются именно с использованием указателей. Вы будете встречаться с ними каждый раз, когда будут использоваться такие возможности .NET, как команды работы с графикой или методы обработки текстовой информации.

Но, даже если вам кажется, что вы уже стали экспертом по указателям, не обольщайтесь: вам предстоит еще многому научиться.

Глава 14

Масса информации? Используйте массивы!

В этой главе...

- Что такое массив
- Сортировка и извлечение информации из массива
- Многомерные массивы
- Использование ArrayList
- Использование Stack
- Использование перечислимых типов вместо констант

Массив — это определенный способ представления данных, используемый во многих программах. Строение массива похоже на строку (или столбец) ячеек электронной таблицы: в вашем распоряжении есть целое множество отдельных ячеек, в которые можно записывать информацию. Такая структура очень удобна для сохранения данных и их последующей обработки.

В этой главе описывается использование массива для сохранения списка элементов, содержащих данные о графических объектах. Также вы познакомитесь со специальными классами .NET, которые предназначены для оптимизации процесса сохранения данных в программе.

Массивы: познакомимся поближе

Особенность массивов состоит в том, что каждый их элемент имеет порядковый номер, называемый *индексом*. Индекс используется для получения доступа к данным, сохраненным в каждом отдельном элементе. Индекс позволяет также использовать циклы для просмотра значений всех элементов массива или значений только некоторого диапазона элементов. С помощью индекса можно сразу же получить непосредственный доступ к данным любого элемента массива. Такой способ доступа к данным называется *произвольным*; он намного быстрее использования связанных списков.

Предположим, вы используете массив для хранения информации о продолжительности музыкальных записей. Тогда, если нужно просмотреть, например, продолжительность первой записи, вы просто смотрите значение элемента под номером 1. Точно так же можно сохранять информацию о котировках акций на биржах, данные о сотрудниках, результаты футбольных матчей и т.д.

Прежде чем создавать массив, вы должны определить, из какого количества элементов он будет состоять. В этом отличие массивов от списков, где количество элементов заранее не известно. (Хотим вас обрадовать: далее в главе рассматривается специальный класс .NET ArrayList, который объединяет в себе все лучшие свойства массивов и списков.)

Предположим, например, что `anFoo` — это массив чисел типа integer. Тогда он может выглядеть приблизительно так:

Индекс	Значение
0	32
1	10
2	17
3	-5
4	**10**

Как видите, первый элемент массива имеет нулевой индекс, второй — индекс 1 и т.д. И в данном случае элемент с индексом 0 имеет значение 32, а элемент с индексом 4 — значение 10.

Если вы придерживаетесь соглашений о присвоении имен, принятых в этой книге, начинайте имена массивов с буквы a, затем набирайте префикс, соответствующий типу данных массива, и далее само имя. Например, имя массива типа integer начинайте с букв an, типа double — с букв adbl и т.д.

Чтобы создать массив, нужно просто указать тип данных, имя массива и в квадратных скобках ([ ]) количество элементов.

Например, чтобы создать массив типа integer, наберите такой код:

```
//Создание массиза типа integer, состоящего из 20
//элементов (индексы от 0 до 19)
;nt anFoo[20];
```

Помните, что первый элемент массива имеет нулевой индекс. Поэтому, если массив имеет *n* элементов, последний элемент будет иметь индекс *(n* - 1). Например, для массива, показанного несколькими абзацами выше, *n* равнялось числу 5 (поскольку массив состоит из пяти элементов). Первый элемент имеет нулевой индекс, а последний — индекс 4 (или 5-1).

Те, кто только начинает использовать массивы, очень часто забывают об этом и думают, что первый элемент имеет индекс 1, а потом долго не могут понять, почему они получают из массива не те значения, которые им нужны.

Аналогичную ошибку совершают и в отношение последнего элемента массива: при обращении к нему вместо индекса *(n* -\) используют индекс *n*. В результате этого либо будут получены совершенно нечитаемые данные, либо возникнет ошибка GPF (общее нарушение защиты). Последнее случается из-за того, что C++ не может предотвратить обращение к памяти, расположенной за пределами массива. Если вернуться к приведенному выше коду и предположить, что вы ошибочно набрали anFoo [20], это будет обращение к участку памяти, расположенному сразу после области памяти, выделенной для элементов данного массива. Последствия могут быть самыми непредсказуемыми.

Это же "элементарно", Ватсон

Чтобы получить доступ к значению какого-либо элемента массива, наберите имя массива и укажите в квадратных скобках индекс этого элемента. Продемонстрируем это на примере.

```
//Массив типа integer из 20 элементов
int anFoo[20];
//Первому элементу присваивается значение 20,
```

```
//а второму значение 3
anFoo[0] = 20;
anFoo[1] = 3;
//Отображение значения второго элемента
Console::WriteLine(anFoo[1]);
//Отображение значения третьего элемента, умноженного на 5
Console::WriteLine(5*anFoo[2]);
```

Обратите внимание, что при отображении только элементов массива не нужно вызывать функцию ToString для преобразования числового значения в текстовое. Среда .NET может это сделать автоматически. Если же вы используете более сложные варианты функции Console::WriteLine, например форматируете строки, вызвать функцию ToString все же придется.

Инициализация массива

Инициализация — это присвоение начальных значений. Инициализировать массив можно несколькими способами. Можно, например, вручную присвоить значения каждому элементу:

```
anFoo[0] = 1;
anFoo[1] = 3;
.
.
.
```

Другой способ — использовать цикл. Этот путь очень удобен, если можно использовать какой-то шаблон для присвоения значений или если эти значения могут быть получены из файла данных. Например, если необходимо, чтобы массив содержал значения от 1 до 20, наберите такой код:

```
//Массив из 20 элементов
int anIntegers[20];
//Цикл, на каждой итерации которого очередному
//элементу присваивается значение,
//большее значения предыдущего элемента на 1
for (int i = 0; i < 20; i++)
  anIntegers[i] = i + 1;
```

Еще один способ инициализировать массив — указать значения элементов при объявлении этого массива. При этом можно указывать значение не всех элементов, а только некоторых (остальным элементам, для которых вы значения не укажете, автоматически будут присвоены нули). Например, массив может быть инициализирован таким образом:

```
int anMyInts[10] = {1, 3, 5, 9, 4, 8};
```

В данном случае первым шести элементам будут присвоены перечисленные значения (первому элементу — 1, второму— 3 и т.д.), а последние четыре элемента получат значения, равные нулю.

В своей программе вы можете также создать и инициализировать массив, состоящий из текстовых значений. В этом случае при объявлении массива его размер указывать не нужно. Например:

```
//Создание массива, состоящего из строк
String *aszFoo[] = {S"hello", S"goodbye"};
Console::WriteLine(S"В массиве есть слово {0}", aszFoo[1]);
```

Связь между массивами и указателями

Массив элементов со значениями типа *x,* по сути, является указателем на значение типа x. Поэтому, набранное вами

```
int foo[8];
```

будет почти аналогично набранному

```
int *foo;
```

Единственная разница в том, что когда вы объявляете массив, выделяется память сразу для множества значений указанного типа, в то время как при объявлении указателя память выделяется только для одного элемента. Переменная массива ссылается на первый элемент этого же **массива.**

Указатели иногда используются для получения доступа к элементам массива. Рассмотрим это на таком примере:

```
//Массив из 8 элементов с числовыми значениями
int foo[8];
//Указатель на числовое значение
int *bar;
//Указателю присваивается адрес первого элемента массива
bar = foo;
//Отображение значения первого элемента массива
cout << *bar;
//Отображение значения второго элемента массива
//Обратите внимание, что это равнозначно foo[1]
cout << *(bar + 1);
```

Если вы используете указатель для ссылки на значение элемента массива, при добавлении единицы в действительности добавляется размер памяти, выделенный для одного элемента этого массива. Таким образом вы получаете адрес следующего элемента.

В неуправляемых программах массивы, состоящие из строк, создаются несколько иначе. Если вы помните, сама строка — это массив символов (объявляется как `char *`). Ниже приведен код неуправляемой программы, при выполнении которого создается и инициализируется массив из строк, а затем все значения этого массива отображаются на экране.

```
//Создание массива из трех строк и присвоение его
//элементам начальных значений
char *aszFoo[3] = {"hello", "goodbye", "bye-bye"};
//Отображение строк на экране
cout << aszFoo[0] << aszFoo[1] << aszFoo[2] << endl;
```

Многомерные массивы

Массивы, которые рассматривались до сих пор, были *одномерными.* Кроме них, существуют также *многомерные* массивы, без использования которых вряд ли можно обойтись при решении многих типичных и не очень типичных задач. Представим, например, что вам нужно сохранить информацию о том, сколько домов обозначено в каждом квадрате на карте города. Поскольку карта двухмерна, для решения этой проблемы удобнее и логичнее использовать двухмерный массив (рис. 14.1).

Или возьмем другой пример: нужно смоделировать перемещение какого-нибудь летающего насекомого в пределах вашей комнаты. Для этого можно условно разбить пространство комнаты на кубики, соизмеримые с размерами животного, и отмечать его перемещение из кубика в кубик. Множество кубиков будет представлять собой трехмерный массив. Много-

мерные массивы могут быть использованы при решении не только "пространственных" задач. Например матрицы, которые применяются для обработки графики, являются двухмерными массивами. Управление большими объемами разнообразных данных может быть сведено к работе с многомерными массивами.

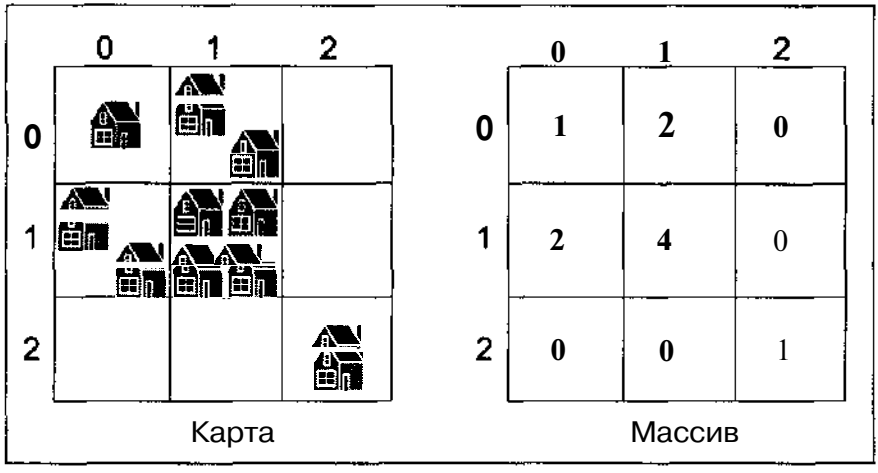


Рис. 14.1. Квадраты обычной двухмерной карты представлены ячейками двухмерного массива, значения которых соответствуют количеству обозначенных на карте домов

Чтобы определить многомерный массив нужно использовать столько квадратных скобок ([ ]), сколько размерностей будет иметь этот массив. Следующим кодом, например, создается двухмерный массив, представляющий шахматную доску (восемь клеток в ширину и восемь в длину):

```
int anChessBoard[8][8];
```

Чтобы обратиться к нужной ячейке, опять-таки необходимо использовать столько квадратных скобок, сколько размерностей имеет массив:

```
//Получение значения из ячейки 3,4
anFull = anChessBoard[3][4];
```

Класс ArrayList

Самостоятельно создавать массивы — это здорово. Не менее увлекательное занятие — создавать собственные связанные списки (что было продемонстрировано в главе 13). Но если вы не в духе или просто не хочется напрягаться, среда .NET предлагает вам воспользоваться специально разработанными классами для представления структур данных. В действительности это даже лучше, чем создавать массивы и списки самостоятельно. Пускай Microsoft выполнит за вас всю рутинную работу, а также найдет и исправит все возможные стандартные ошибки. Вы же в это время можете заняться более важными делами и создать ошибки посерьезнее.

Одна из лучших структур .NET носит название `ArrayList`. Она похожа на массив в том смысле, что к ее элементам можно обращаться непосредственно (т.е. они имеют порядковые номера), и в то же время подобна списку, поскольку на количество ее элементов нет ограничений (т.е. в любой момент в конец этой структуры может быть добавлен новый элемент). Долее того, элементы этой структуры могут содержать значения любых типов, для которых активизирована возможность сборки мусора (garbage collection).

"Многомерные" советы

Ниже приведено несколько советов, которые могут вам пригодиться при создании и использовании многомерных массивов.

- ✓ **При создании многомерного массива точно укажите объем каждой размерности.** (Можно указать объем *n* - 1 размерности л-мерного массива, но, чтобы уменьшить вероятность возникновения ошибок, укажите лучше объем для всех л размерностей.)
- ✓ **Компьютеру совершенно все равно, какая именно размерность будет использована для представления того или иного свойства.** Например, если вы используете двухмерный массив для представления объектов с координатами X и Y, можете использовать первую размерность как координату X, а вторую как координату *Y*. А можете наоборот: первую — как У, а вторую — как *X*. Главное, сами не запутайтесь,
- ✓ **Индексы каждой размерности должны указываться отдельно, каждый в своей паре квадратных скобок.** Например, [1][7] — это не то же самое, что [1,7]. Индекс [1][7] предоставляет доступ к элементу двухмерного массива, а [1,7] — это то же самое, что и [7].
- ✓ **Компилятор не проверяет, выходит ли указанный вами индекс за пределы массива.** Так, если вы объявили массив из 20 элементов и затем обращаетесь к 50-му элементу, компилятор без проблем вычислит "его" адрес и вернет то, что там записано. Поэтому сами проверяйте, принадлежат ли используемые вами индексы к объявленному диапазону.
- ✓ **Компилятор также не будет "возражать", если вы обратитесь к двухмерному массиву как к одномерному.** Компилятору индексы нужны только для того, чтобы вычислить нужный адрес. Вы можете использовать этот факт обхода некоторых условностей и для написания более быстродействующих кодов.

Вот как создается структура ArrayList:

```
ArrayList *poArray = new ArrayList();
```

А вот как созданная структура заполняется информацией:

```
//Добавление строки
poArray->Add(S"Строка");
//Добавление указателя
poArray->Add(poFirst);
```

Можно также присвоить элементу индекс, а потом получать доступ к его значению, ссылаясь на этот индекс:

```
poArray->set_Item(0, S"Hello");
Console::WriteLine(poArray->get_Item(0));
```

При назначении индекса или при попытке использовать индекс для получения значения элемента, убедитесь, что этот индекс не ссылается за пределы созданной структуры.

Структура ArrayList обладает некоторыми весьма полезными возможностями. Например, можно определить, какой из элементов структуры имеет то или иное конкретное значение:

```
Console::WriteLine(poArray->IndexOf(S"Значение"));
```

Единственной сложностью, которая имеет место при использовании структуры ArrayList (а также при использовании структуры Stack, с которой вы вскоре познакомитесь), является переход от одного элемента к другому. Предположим, например, что структура ArrayList состоит из пяти элементов и вам нужно просмотреть значения ее элементов, начиная с первого и далее. Можно создать цикл, который будет использовать для доступа к

значениям элементов метод get_Item. Или можете использовать более сложный прием, который, однако, отлично работает со всеми классами .NET, представляющими структуры данных, включая те, которые не упоминаются на страницах этой книги. Этот более сложный прием заключается в создании нумератора (его еще называют перечислителем), название которого расшифровывается приблизительно как "специальная "штуковина", позволяющая по порядку, один за другим, просматривать значения элементов структуры".

Вот как создается нумератор:

```
IEnumerator *poEnumerator = poArray->GetEnumerator();
```

После того как нумератор создан, можете использовать любой из методов, приведенных в табл. 14.1. Если, например, вам нужно поочередно отобразить значения всех элементов структуры ArrayList, наберите такой код:

```
IEnumerator *poEnumerator = poArray->GetEnumerator();
while ( poEnumerator->MoveNext() )
Console::WriteLine( poEnumerator->Current );
```

Этим кодом вначале создается нумератор, а затем компилятор, переходя от одного элемента к другому, пока не будет достигнут конец структуры, отображает значения этих элементов. Обратите внимание, что метод MoveNext, выполняющий переход к следующему элементу, должен быть вызван до отображения значения первого элемента.

Таблица 14.1. Методы, используемые при работе с нумератором

Метод	Пример	Описание
Current	poE->Current	Возвращает значение элемента, на который указывает нумератор
MoveNext	poE->MoveNext()	Изменяет значение нумератора с тем, чтобы он указывал на следующий элемент структуры. Если элементов больше нет, возвращает логическое значение false
Reset	poE->Reset()	Нумератор возвращается к первому элементу структуры

Класс Stack

Другая замечательная структура .NET, используемая для представления данных, называется Stack. Принцип ее работы очень простой. Представьте, что вы собираетесь в поход и вам нужно сложить вещи в рюкзак. Вначале разумнее положить то, что нужно будет доставать в последнюю очередь, затем можно положить какие-то другие вещи, а на самый верх то, что придется доставать в первую очередь. Когда вы будете вынимать вещи из рюкзака, вы вначале достанете то, что положили туда последним, потом другие вещи и наконец ту вещь, которую клали самой первой.

Итак, последний элемент, записанный в Stack, извлекается из него первым. Как и в случае со структурой ArrayList, в Stack можно записывать данные любых типов, для которых активизирована возможность сборки мусора.

Чтобы создать Stack, наберите такой код:

```
Stack *poStack = new Stack();
```

Следующим кодом элемент помещается в Stack:

```
poStack->Push(S"Какая-нибудь строка");
```

Чтобы извлечь элемент из структуры Stack, наберите:

```
poStack->Pop();
```

Переход от одного элемента к другому осуществляется тем же способом, который использовался для просмотра элементов структуры ArrayList:

```
poEnumerator = poStack->GetEnumerator();
while ( poEnumerator->MoveNext() )
Console::WriteLine( poEnumerator->Current );
```

Перечислимые типы

Вы только что узнали о том, как можно использовать нумератор для просмотра элементов, содержащихся в структурах ArrayList и Stack. Но есть еще одна разновидность нумераторов (или перечислителей), которая обозначается термином *перечислимый тип*. Перечислимые типы удобны в тех случаях, когда есть набор элементов, но, вместо того чтобы идентифицировать их по числовым константам, вы хотите поставить им в соответствие какие-нибудь легкочитаемые понятные обозначения. Например, вещи в рюкзаке можно обозначить словами Верхняя, Средняя и Нижняя. Первый вариант: сказать, что 0— это вещь, положенная сверху, 1 — вещь, положенная посередине, и 2 — вещь, положенная на самый низ. Второй вариант: создать перечислимый тип.

Так, например, вместо кода

```
const int Верхняя = 0;
const int Средняя = 1;
const int Нижняя = 2;
вы могли бы набрать:
enum {Верхняя, Средняя, Нижняя};
```

Любое слово, указанное в списке enum (Верхняя, Средняя или Нижняя), может быть использовано в любом месте программы в качестве константы.

Безопасность при использовании перечислимых типов

Если хотите снизить вероятность возникновения ошибок при написании программы, укажите, что набор элементов, перечисленных в списке enum, является отдельным типом данных. Таким образом будет исключена возможность случайного использования констант из одного набора enum (созданного, скажем, для представления названий музыкальных записей) в тех выражениях, где должны использоваться константы enum из другого набора (представляющего, например, типы моторных масел).

Можно, например, указать, что константы, обозначающие способ размещение вещей в рюкзаке, принадлежат типу Размещение:

```
enum Размещение {Верхняя, Средняя, Нижняя};
```

Если вы поступите таким образом, компилятор сможет проверять, действительно ли переменная, для которой определен тип Размещение, принимает только те значения, которые для этого типа предусмотрены. Продемонстрируем это на небольшом примере:

```
enum Shapes
{
  Circle,
  Square
};
```

```
//Использование списка констант enum в качестве
//типа данных
Shapes oMyShape;

//Присвоение переменной значения из списка enum
//Обратите внимание, что код oMyShape = Oval будет
//воспринят как синтаксическая ошибка
oMyShape = Circle;
```

По возможности старайтесь обозначать списки констант enum как перечислимые типы данных. Тогда, если вы по ошибке попытаетесь использовать константу перечислимого типа для обозначения информации не того вида, компилятор выдаст предупреждение. Программа из-за этого работать не перестанет, но предупреждающее сообщение поможет вам определить источник возникновения этой ошибки и вовремя ее устранить.

Одно маленькое "но"

Перед тем как создать целый букет перечислимых типов, обратите внимание на тот факт, что команда `cin` может считывать значения только стандартных (предустановленных) типов данных. Если вы попытаетесь использовать команду `cin` для получения от пользователя значения, указанного в списке enum, компилятор выдаст сообщение об ошибке.

Например, приведенный ниже код вызовет сообщение об ошибке, суть которого будет приблизительно в том, что оператор не может определить, как правильно обрабатывать значения типа enum Размещение.

```
//Создание перечислимого типа
enum Размещение {Верхняя, Средняя, Нижняя};
//Переменная foo типа Размещение
Размещение foo;
//Пользователь должен указать значение для foo
cin » foo;
```

Глава 15

Пришел, увидел, применил

В этой главе...

- Что такое область видимости
- Глобальные и локальные переменные
- Почему в программе разные переменные могут иметь одинаковые имена

Данные в программе сохраняются как значения переменных. Разные переменные используются в разных местах программы. В этой главе вы познакомитесь с таким понятием, как *область видимости переменных,* которая определяется набором правил, устанавливающим, к каким переменным имеет доступ каждая отдельная часть программы.

Немного теории

Программы становятся все больше и больше, и вместе с этим возрастает количество используемых ими функций и переменных. Большинство из этих переменных используются только при решении какой-то одной небольшой задачи. Например, многие переменные используются только для отсчета количества итераций при выполнении циклов или для временного хранения значения, которое ввел пользователь. К счастью, не нужно каждой из таких переменных присваивать какое-то уникальное имя. Ведь в противном случае вам пришлось бы придумывать миллионы отличающихся друг от друга имен.

Итак, разные переменные могут быть названы одинаковыми именами. До тех пор пока такие переменные используются разными функциями, они между собой не конфликтуют. Вы можете, например, определить переменную к для функции `foo`. А потом определить переменную к для функции `baz`. Хотя названия одинаковые, это разные переменные: одна используется только функцией `foo`, а вторая — только функцией `baz`.

Эти переменные разные, потому что имеют разную область видимости. *Область видимости переменных* — это место в программе, где эти переменные могут быть использованы. Например, областью видимости переменной к, определенной внутри функции `foo`, является функция `foo`. Это значит, что данная переменная может быть использована только в пределах функции `foo`, а вне ее эта переменная не определена.

Все переменные могут быть разделены на две категории: глобальные и локальные. *Глобальные переменные* — это те, доступ к которым имеет любая функция программы (включая `main`). Они определяются за пределами какой-либо функции. Такие переменные нужны для тех случаев, когда некоторое значение должно быть доступным независимо от того, какая из функций в данный момент выполняется. Имена всех глобальных переменных должны быть разными.

Локальные переменные — это временные переменные, которые используются какой-то одной функцией. Локальная переменная создается тогда, когда функция начинает работать, используется с процессе выполнения функции, и уничтожается, как только

функция прекращает работу. Можно использовать только одну переменную с тем же именем внутри одной функции, однако этим же именем можно обозначать переменную, используемую в другой функции. (Изменение значения одной из этих переменных никак не отразится на значении другой переменной.)

На самом деле все несколько сложнее. Переменная видима в пределах пары фигурных скобок ({}). внутри которых она была определена. Так, если у вас есть переменная nBob, объявленная внутри функции First, и функция First вызывает функцию Second, переменная nBob не будет доступна для функции Second (если только вы не передадите этой функции переменную nBob в качестве аргумента). Точно так же, если вы определите переменную fYou внутри блока оператора if, например:

```
if (nBob > 10)  {
  bool fYou = true;
}
```

В этом случае переменная `fYou` будет видима только внутри данного блока.

Это очень удобно, и вот почему. Предположим, например, что у вас есть функция, отображающая числа от одного до десяти (пусть она называется CountUp). Чтобы сделать это, в ней используется цикл. В самом цикле используется переменная i для отсчета количества итераций. Если у вас есть другая функция, отображающая числа в обратном порядке — от десяти до одного (назовем ее `CountDown`), она также может использовать какую-то переменную для отсчета количества итераций. Поскольку обе переменные используются только в пределах своего цикла, вторую также можно назвать i. Это означает, что вам не нужно придумывать уникальные имена для переменных при написании каждого нового цикла. Если вы создаете большую программу, состоящую из огромного количества функций, часть из которых написана другими программистами, вам не придется кричать на всю комнату: "Эй! Кто-нибудь уже использовал для переменной цикла название `sdbsdbsdbsdb3`?"

Каждый раз, когда вы определяете переменную внутри какой-то функции, область ее видимости будет ограничена пределами этой функции (т.е. она будет локальна). Вы можете использовать ее внутри этой функции, а также передавать в качестве аргумента другим подпрограммам, которые этой функцией вызываются. Как только функция заканчивает работу, локальная переменная прекращает свое существование. Использование переменных, которые передаются функциям в качестве аргументов, также ограничено пределами этих функций.

Почему это так важно

На первый взгляд область видимости переменных может показаться вопросом второстепенным. Действительно, при написании программ многие программисты об этом почти не задумываются. Однако для тех, кто только начинает программировать, вопросы, связанные с областью видимости, могут стать источником многих проблем. Поняв принципы определения области видимости, вы сможете избежать некоторых трудных для обнаружения логических ошибок,

Рассмотрим небольшой пример.

```
int x;

void ZeroIt(int x)
{
   x = 0;
}

int main(void)
{
   x = 1;
   ZeroIt(x);
   Console::WriteLine(x);
}
```

Что произойдет при выполнении этой программы? Вначале глобальной переменной x будет присвоено значение 1. Затем будет вызвана функция Zerolt, которая присвоит этой переменной значение 0 (нуль). Как вы думаете, какое число будет отображено при выполнении инструкции Console : :WriteLine? Отображено будет число 1.

Почему именно 1? Переменная x внутри функции Zerolt является локальной по отношению к ней (ее область видимости ограничена пределами этой функции). Поэтому она создается, когда какая-то другая функция вызывает функцию Zerolt. Этой переменной присваивается значение, которое передается функции в качестве аргумента (в данном случае это значение 1). Затем функция Zerolt присваивает переменной x значение 0. Когда функция заканчивает работу, переменная x (имеющая значение 0 и являющаяся локальной по отношению к этой функции) уничтожается. Безвозвратно. Теперь компилятор возвращается к функции main и переходит к инструкции Console: : WriteLine, которая отображает значение переменной x на экране. Но это уже другая переменная x — та, которая имеет глобальную область видимости. (Как вы помните, переменные, объявленные за пределами любой функции, являются глобальными.)

Точно такой же результат будет получен при выполнении кода, который приведен ниже. Как и в предыдущем примере, изменение значения переменной x в процессе выполнения функции Zerolt никак не отобразится на значении глобальной переменной x, которая используется функцией main. Причиной тому является локальная область видимости переменной x, которая используется внутри функции Zerolt. Следовательно, эта переменная вместе со своим значением исчезает, как только Zerolt заканчивает работу.

```
int x;

void ZeroIt()
{
  int X;
  x = 0;
}

int main(void)
{
  x = 1;
  ZeroIt(x);
  Console::WriteLine(x);
}
```

Совершенно другую картину мы наблюдаем в следующем примере:

```
int x;

void ZeroIt(int y)
{
  y = 7;
  x = 0;
}
int main(void)
{
  x = 1;
  ZeroIt(x);
  Console::WriteLine(x);
}
```

Если будет выполнен этот код, на экране отобразится число 0 (нуль).

Почему теперь будет получен О? В данном случае аргумент функции `ZeroIt` обозначен именем y. Если вы передаете этой функции в качестве аргумента значение переменной x (глобальной переменной x, которая в данном примере своей локальной тезки не имеет), оно присваивается переменной y. Затем переменной y присваивается значение 7, а переменной x — значение 0. Поскольку для функции Zerolt локальной переменной x не создавалось, это значение присваивается глобальной переменной x. Когда функция Zerolt заканчивает работу, переменная y уничтожается. Поскольку x не является локальной по отношению к функции Zerolt, она никуда не пропадает и сохраняет свое измененное значение.

Вообще-то последний приведенный фрагмент кода служит примером плохого стиля программирования. Поскольку функция Zerolt изменяет значение переменной, которое не было передано ей как аргумент, вы не узнаете, что функция изменила это значение, если не будете просматривать все ее коды. Как следствие, вполне вероятно, что написанная вами программа не будет работать так, как вы ожидаете. В общем случае не стоит изменять значения тех переменных, указатели или ссылки на которые не были переданы функции в качестве аргументов. Функция свободно может изменять значения только тех переменных, которые являются для нее локальными.

Хорошие привычки

Вот несколько советов, которые сделают ваши программы удобными для чтения и понимания и избавят вас от ошибок, связанных с неправильным пониманием области видимости.

- ✓ Создавая функцию, передавайте ей в качестве аргументов асе данные, необходимые для ее выполнения.
- ✓ Избегайте использования глобальных переменных. Создавайте их только для представления постоянных значений.
- ✓ Не изменяйте какие-либо значения за пределами видимости. Другими словами, не изменяйте значения переменных там, где это не будет для вас очевидно. Если функции будут изменять значения только локальных переменных и тех переменных, ссылки на которые им переданы в качестве аргументов, понять, что к чему в программе, будет не так уж сложно.

Правила определения области видимости

К счастью, в C++ правила определения области видимости переменных довольно просты и могут быть легко перечислены,

- ✓ **Область видимости переменных, определенных внутри функции, ограничена пределами этой функции.** Если вы определяете переменную внутри функции, она создается, когда эта функция вызывается, используется в процессе выполнения функции и уничтожается, когда функция заканчивает работу.
- ✓ **Область видимости переменных, объявленных внутри блока, ограничена пределами этого блока.** Переменные, объявленные внутри какого-то блока (например, блока оператора if, выделенного парой фигурных скобок), видимы только в пределах этого блока.
- ✓ **Любые аргументы, передаваемые функции, видимы только в пределах этой функции.** Например, если вы определяете, что функция использует параметр `ARose`, область его видимости будет ограничена пределами этой функции (точно так же, если бы вы создали переменную ARose внутри этой функции). Имя того элемента, значение которого вы передаете функции в качестве аргумента, роли не играет. Значение имеет только имя самого аргумента.
- ✓ **Когда выполняется функция, переменная, локальная для этой функции, используется вместо глобальной переменной с тем же именем.** Предположим, например, что внутри функции foo вы объявили переменную ARose типа integer, но глобальная переменная типа double также имеет имя ARose. При выполнении функции foo использоваться будет именно та переменная ARose, которая имеет тип integer и является локальной по отношению к этой функции. Чтобы получить доступ к глобальной переменной ARose, перед ее именем нужно будет набрать два двоеточия (: :). Таким образом, локальная переменная ARose — это совсем не то же самое, что глобальная переменная ARose. Это две разные переменные, которые имеют одинаковые имена, но используются в разных частях программы.
- ✓ **Изменение значения локальной переменной никак не влияет на значения переменных с тем же именем, но с другой областью видимости.** Если вы, например, измените значение локальной переменной ARose, то это никак не отразится на значении глобальной переменной с тем же именем.
- ✓ **Если внутри функции используется переменная, которая не является локальной для этой функции, компилятор пытается найти глобальную переменную** с **тем же именем.** Предположим, например, что вы используете переменную ARose при написании кодов функции **foo.** Однако ни среди аргументов функции **foo,** ни среди локальных переменных, объявленных внутри этой функции, имя ARose не встречается. Что в этом случае будет делать компилятор? Он попытается найти глобальную переменную **с** таким именем. Если среди глобальных переменных также не окажется переменной с именем ARose, компилятор выдаст сообщение об ошибке.

- **Если необходимо получить доступ к глобальной переменной во время выполнения функции, в которой используется локальная переменная с тем же именем, наберите перед именем этой переменной два двоеточия** (: :). Если, например, внутри функции объявлена локальная переменная ARose и в этой же программе используется глобальная переменная, также названная именем `ARose`, код : : ARose будет указывать на глобальную переменную, а ARose — на локальную.

Глава 16

Через тернии к... работающей программе

В этой главе...

- Отладчик программ и его возможности
- Процесс отладки программ
- Точки останова
- Пошаговое выполнение программ
- Просмотр текущих значений
- Средство QuickWatching
- Пример отладки и корректирования программы

Представьте, что вы блуждаете по лабиринту и не можете найти выхода. Заряд батареи заканчивается, и ваш фонарик вот-вот погаснет. Повсюду слышится непонятное "клик, клик, клик...". Вы опускаете глаза и видите, что по колено стоите в кишащей массе огромных тараканов!

Это может выглядеть как кошмарный сон, но примерно так чувствует себя программист, который всю ночь не отрывался от компьютера и чем больше писал кодов, тем больше появлялось ошибок — этих мерзких и противных программных тараканов. И в результате, когда на следующее утро он попытался запустить написанную программу... кошмар! Ничего не работает.

Есть четыре варианта дальнейших действий.

- ✓ Можно вообще отказаться от идеи написания программы. (Это путь неудачников.)
- ✓ Можно попробовать начать все сначала. (Это плохая идея.)
- ✓ Можно к каждой строке кодов добавить команду `print`, чтобы получать от программы отчет о каждом выполняемом ею шаге и таким образом проследить, что именно работает не так, как нужно. (Ход мыслей правильный, но выполнить это на практике будет очень сложно.)
- ✓ Можно воспользоваться отладчиком программ. (Это самый лучший вариант.)

Отладчик — это специальный инструмент, который позволяет выполнять программы пошагово, строка за строкой. Благодаря этому вы сможете проследить, как в действительности будет выполняться написанная вами программа, определить, что именно работает не так, как было задумано, и затем исправить допущенные ошибки.

В этой главе речь пойдет об использовании для корректировки программ отладчика Visual C++ .NET.

Синтаксические и логические ошибки

Как отмечалось в главе 6, синтаксические ошибки случаются тогда, когда компилятор не может "понять" написанный вами код. Например, если вы в слове вместо одной буквы набрали другую, пропустили часть команды либо неправильно использовали какую-то переменную или класс, все это будет воспринято как синтаксические ошибки. До тех пор пока в програм-

ме есть хоть одна синтаксическая ошибка, она не может быть выполнена, поскольку компилятор не сможет ее откомпилировать.

Кроме синтаксических, есть еще логические ошибки (программисты называют их *bugs* — жучки). Программа может быть написана на безупречном C++, но выполнять совершенно бессмысленные действия. Или она может работать, но несколько не так либо совсем не так, как было задумано. В таком случае в программе есть логические ошибки.

Рассмотрим, например, инструкцию по приготовлению печеной картошки.

1. Взять картошку.
2. Завернуть ее в фольгу.
3. Положить в микроволновую печь на три часа.

Инструкции просты и без труда могут быть выполнены кем угодно. Синтаксических ошибок здесь нет. Но с другой стороны, если эти инструкции выполнить, микроволновая печь просто сгорит (черт с ней, с картошкой). Наверное, это не тот результат, который нужно было получить.

Чтобы решить такого рода проблему, нужно проанализировать все предпринимаемые шаги и определить, какой из них приводит к плачевному результату. В этом и заключается процесс *отладки:* строка за строкой просмотреть проблемный фрагмент кода и определить, что именно работает неправильно.

Если программа довольно большая, устранить все логические ошибки очень сложно. Причина в том, что некоторые ошибки могут проявляться только при стечении целого ряда обстоятельств. Множество нестандартных ситуаций возникает тогда, когда программа попадает в руки пользователей. Если с программой будет работать много людей, все логические ошибки рано или поздно будут обнаружены. Пользователи, конечно же, сообщат вам об этом, и вы сможете эти ошибки исправить.

Процесс отладки программы

В основном процесс отладки сводится к определению в коде программы того места, где кроется ошибка. Цель состоит в том, чтобы определить, в каком фрагменте кода может содержаться та или иная ошибка, и затем проверить, как выполняется именно этот фрагмент. Если, например, вы точно знаете, что ошибка скрывается где-то в кодах определенной процедуры, вы можете дать указание, чтобы выполнение программы останавливалось каждый раз. когда будет вызываться эта процедура. Затем можно пошагово проследить за выполнением кодов этой процедуры, чтобы точно знать, что при этом происходит. Если вы увидите, что выполнение какой-то инструкции не приводит к ожидаемому результату, измените эту инструкцию, заново откомпилируйте программу и попробуйте запустить ее снова.

В процессе отладки программы можно использовать несколько полезных инструментов.

- ✓ **Breakpoints (Точки останова).** Дает указание отладчику остановить выполнение программы тогда, когда будет достигнута определенная строка кодов. Этот инструмент удобен в том случае, если у вас большая программа, а вы хотите проследить за выполнением только какого-то отдельного фрагмента кода. Поставьте точку останова перед этим фрагментом и запустите программу. Отладчик выполнит все инструкции, набранные до точки останова, и остановится, чтобы дать вам возможность далее выполнять программу пошагово.
- ✓ **Step Into (Пошаговое выполнение) и Step Over (Пошаговое выполнение с перешагиванием через процедуры).** Выполнение кодов программы по одной строке за раз. Используйте эти средства для контроля за выполнением каждой отдельной инструкции. Если вы будете внимательны, то обязательно обнаружите искомую ошибку (если она там есть).

- Watches (Просмотр). Отображение текущих значений переменных в процессе выполнения программы. Это средство поможет вам проследить за жизненным циклом интересующих вас переменных и увидеть, как изменяются их значения после выполнения тех или иных инструкций. Это же средство позволяет просматривать значения выражений, благодаря чему вы можете видеть, как они изменяются в результате изменения значений переменных, входящих в эти выражения.

Отладчик плюс редактор — и все в наших руках

Окно редактора кодов является также и окном отладчика. Другими словами, если вам нужно совершить такие действия по отладке программы, как, например, определение точки останова или просмотр значений переменных, сделать вы это можете прямо в окне редактора.

Процессы редактирования и отладки программы тесно связаны между собой. Иначе и быть не может, поскольку при отладке программы необходимо просматривать исходные коды, чтобы видеть, выполнение каких из них приводит к тем или иным результатам. Так как Visual C++ объединяет процессы редактирования и отладки, при отладке можно использовать любые возможности редактора кодов (такие, как прокрутка окна, разделение окна или поиск нужного слова). Вы можете даже при отладке сразу же вносить изменения в исходные коды.

Но не вся информация отображается непосредственно в окне редактора. Некоторые действия, связанные с отладкой программ, сопровождаются открытием специальных окон. Например, если вы захотите просмотреть значения переменных, для этого откроется отдельное окно Watch. (Более подробно это описывается несколько ниже в главе.)

Остановись, мгновение!

Точки останова позволяют остановить выполнение программы в нужном месте. Предположим, например, что у вас есть подпрограмма, вычисляющая факториал числа. По какой-то причине возвращаемый ею результат всегда равен нулю. Чтобы определить, в чем заключается ошибка, можно установить точку останова в начале этой подпрограммы. Затем, когда вы запустите программу, ее выполнение будет остановлено в момент вызова процедуры, вычисляющей факториал, и вы сможете использовать все средства отладчика для определения, что именно в этой процедуре работает не так. Забегая вперед, скажем, что весь материал, изложенный далее в главе, посвящен решению подобных проблем.

Установить точку останова довольно просто. В окне редактора кодов щелкните правой кнопкой мыши в том месте, где нужно установить точку останова, и в открывшемся меню выберите команду Insert Breakpoint (Установить точку останова). Слева от выбранной строки появится значок точки останова. Ну хорошо, этим значком будет красный кружок, а не ромбик или что-то еще, но главное, чтобы вы поняли общую идею,

Если вы установили точку останова и не видите слева от строки соответствующий значок, значит, у вас, скорее всего, отключена возможность отображения дополнительной информации на границе строк. Выберите команду Tools⇨Options (Сервис⇨Параметры), чтобы открыть диалоговое окно Options (Параметры). В категории Text Editor/General (Редактор/Общие) выберите General (Общие) и затем опцию Selection Margin (Выделение на границе). Щелкните на кнопке OK, и знак точки останова появится напротив выбранной строки.

Удаляется точка останова так же просто, как и устанавливается. В окне редактора кодов щелкните правой кнопкой мыши на строке, для которой установлена точка останова, и в открывшемся меню выберите команду Remove Breakpoint (Удалить точку останова). Расположенный напротив этой строки значок точки останова исчезнет с экрана.

Точки останова не исчезают после перезагрузки системы. Другими словами, если вы установили точку останова и не удаляли ее, а позже вновь открыли этот же проект, созданная точка останова по-прежнему будет на своем месте. Это очень удобно, поскольку процесс отладки программы может затянуться на какое-то время и вы можете прерывать его на то, чтобы, например, поспать, перекусить или побродить по Internet. Затем, когда вы снова вернетесь к отладке программы, вам не придется заново восстанавливать всю картину и расставлять прежние точки останова. Однако с другой стороны, если программа уже будет работать так, как нужно, но вы забудете убрать какие-то точки останова, то она потом непременно выдаст вам парочку "сюрпризов". (Выполнение программы будет прерываться каждый раз, когда в кодах будут встречаться точки останова.) Если вы не уверены, все ли точки останова удалены, выберите команду Debug⇨Windows⇨Breakpoints (Отладка⇨Окна⇨Точки останова). В результате откроется окно, в котором будут показаны все существующие на данный момент точки останова.

Шаг за шагом: движение к цели

После того как точка останова достигнута, самое время проследить, как будет выполняться каждая последующая инструкция. Сделать это можно, выполняя по одной строке кода за раз. Такие действия называются пошаговым выполнением программы.

Есть два варианта: использовать команду Step Into (Пошаговое выполнение) либо команду Step Over (Пошаговое выполнение с перешагиванием через процедуры). Если в процессе использования команды **Step Into** программа вызывает какую-то функцию, все ее коды также выполняются пошагово. Если программа вызывает функцию в процессе использования команды **Step** Over, все коды функции выполняются сразу же, без остановки на каждом шаге.

Команду Step Into следует использовать в том случае, если какая-то часть программы работает не так, как нужно, но вы не уверены, содержится ли ошибка в этом фрагменте кода или в той функции, которая при этом вызывается. Команда **Step Over** используется в том случае, если проверяемые коды должны выполняться пошагово, а вызываемые при этом функции — как единое целое.

Рассмотрим это на таком примере:

```
foo = MyRoot(x);

foo = foo + 1;
```

Если вы точно знаете, что функция MyRoot работает правильно, и хотите только проверить, как будет выполнена строка `foo+1`, можете использовать команду Step Over. Первая строка будет выполнена за один шаг (включая все инструкции, из которых состоит функция MyRoot), и затем на втором шаге будет выполнена строка `foo=foo+1`.

Если вы сомневаетесь в правильности выполнения функции MyRoot, используйте команду Step Into. При этом, выполняя первую строку, вы перейдете к первой строке функции MyRoot. Затем пошагово, одна за другой, будут выполняться все строки этой функции, и если вам повезет, вы найдете там свою ошибку.

В процессе пошагового выполнения программы слева, напротив строки, которая должна выполниться на следующем шаге, будет отображаться желтая стрелка. Если для этой строки определена точка останова, стрелка будет отображаться сверху над красным кружком.

Если при пошаговом выполнении программы вы не видите желтой стрелки, значит, у вас, скорее всего, отключена возможность отображения дополнительной информации на границе строк. Выберите команду Tools⇨Options, чтобы открыть диалоговое окно Options. В категории Text Editor/General выберите General и затем опцию Selection Margin. Щелкните на кнопке OK, и желтая стрелка появится на экране.

Для пошагового выполнения с перешагиванием через процедуры нажимайте клавишу <F10> или щелкайте на кнопке Step Over панели Debug (Отладка), показанной на рис. 16.1. Если вызываемые функции также должны выполняться пошагово, нажимайте клавишу <F11> или щелкайте на кнопке Step Into.

Рис. 16.1. На панели Debug расположены кнопки, управляющие процессом выполнения программы

Обратите внимание, что команды Step Into и Step Over могут быть использованы без предварительного определения точек останова. Если вы сделаете это, пошаговое выполнение будет начато с самой первой строки программы.

Посмотрим, что получилось

Занимаясь отладкой программы, очень важно знать, правильный ли результат возвращает формула, какое значение имеет переменная или что сохранено в качестве элемента массива. Visual C++ предлагает несколько способов просмотра значений переменных и получаемых при обработке выражений результатов.

Окно Autos (Автоматическое) открывается автоматически, когда вы приступаете к отладке программы (рис. 16.2). В нем отображаются названия и значения переменных, которые используются в текущем и предыдущем выражениях. Лично вам не нужно указывать имена переменных — они сами и их значения отображаются автоматически, по мере того как вы переходите от одной строки программы к другой.

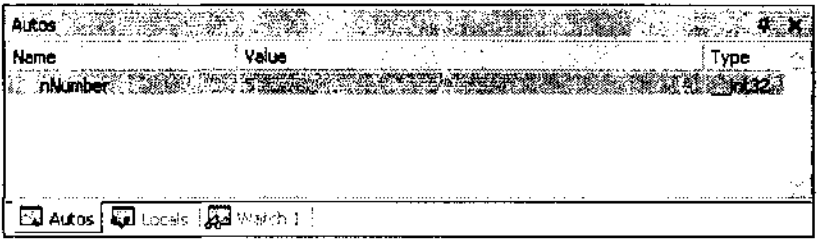


Рис. 16.2. В окне Autos отображаются имена и значения переменных, используемых в предыдущем и текущем выражениях

Также при отладке программы автоматически отображается окно Locals (Локальные), показанное на рис. 16.3. Как и в окне Autos, в нем автоматически отображаются названия переменных и их значения. Но здесь уже отображаются все локальные переменные той функции, коды которой выполняются в данный момент (т.е. переменные, объявленные внутри этой функции).

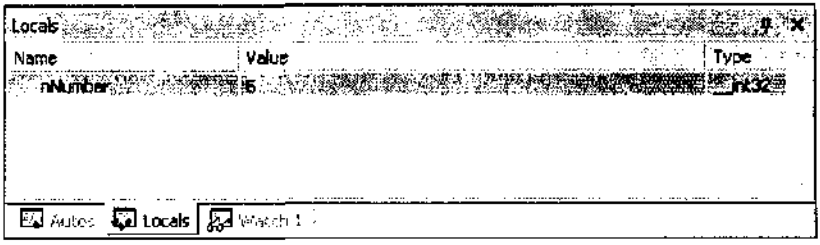


Рис. 16.3. В окне Locals отображаются имена и значения всех локальных переменных

Окна Autos и **Locals** предоставляют полезную информацию, но, поскольку они сами определяют, значения каких переменных должны отображаться, иногда они не очень удобны. Но в этом нет ничего страшного, так как в Visual C++ есть другие средства для контроля за значениями переменных. В окне **Watch** (Просмотр) вы можете сами указать, какие именно значения вас интересуют. Всего есть четыре окна **Watch, пронумерованных от** Watch1 **до Watch4,** что позволяет выделить несколько наборов переменных и выражений, значения которых вы хотите проконтролировать.

Чтобы просмотреть значение какого-либо объекта, используя окно **Watch,** выполните следующее.

1. **В окне Watch щелкните на пустой строке в колонке Name (Имя).**
2. Наберите имя объекта, значение которого вы хотите видеть.
3. **Нажмите клавишу <Enter>.**

Каждый раз, когда значение объекта изменяется, информация в окне **Watch** обновляется. Например, на рис. 16.4 показано окно **Watch** со значениями трех числовых переменных. Как только значение переменной `nNumber`, `nResult` или `i` изменится, окно **Watch** будет обновлено.

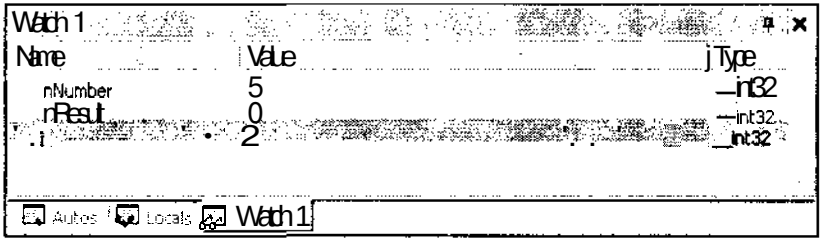


Рис. 16.4. Окно Watch позволяет видеть, как изменяются значения интересующих вас переменных в процессе выполнения программы

Не нравится значение — измените его

Если это необходимо, вы можете даже сами изменять значения просматриваемых переменных. Для этого выполните следующее.

1. **Выделите переменную, значение которой вы хотите изменить.**
2. **Выделите значение этой переменной.**
3. **Наберите новое значение.**
4. **Нажмите клавишу <Enter>.**

Эта возможность может вам пригодиться для того, чтобы проверить, правильно ли будет работать вся остальная программа, если переменная будет иметь корректное значение. Когда вы это проверите, вам нужно будет вернуться назад, чтобы определить, почему эта переменная принимает неверное значение.

Торопитесь? Нет проблем!

Visual C++ позволяет просматривать значения переменных и выражений без необходимости добавления их в окно Watch, используя для этого средство QuickWatch.

Чтобы использовать QuickWatch для просмотра значения какого-либо объекта, выполните в окне редактора следующие действия: щелкните правой кнопкой мыши на названии интересующего вас элемента и выберите в открывшемся меню команду QuickWatch. Откроется диалоговое окно **QuickWatch** (рис. 16.5). Затем, если вы захотите перенести выбранный объект из окна **QuickWatch** в окно **Watch,** просто щелкните на кнопке **Add** Watch (Добавить в окно Watch).

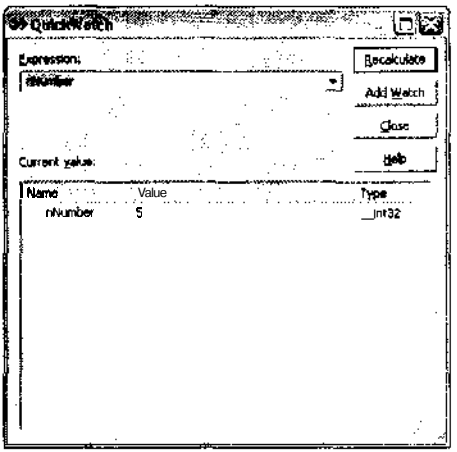


Рис. 16.5. Значения переменных можно просматривать, не добавляя их в окно Watch

Очень торопитесь?

Средства **Watch** и **QuickWatch** удобны, но все же, чтобы их использовать, вам придется несколько раз щелкнуть кнопкой мыши. Представим, что вы этого делать не хотите. Тогда Visual C++ может предложить другое средство, которое называется DataTips (Подсказка). Оно позволяет увидеть значение переменной без единого щелчка кнопкой мыши.

Чтобы просмотреть значение переменной, используя средство DataTips, выполните следующее.

1. **В окне редактора поместите курсор над той переменной, значение которой вы хотите увидеть.**
2. Подождите полсекунды.

Рядом с курсором появится подсказка, в которой будет показано текущее значение этой переменной **(рис.** 16.6).

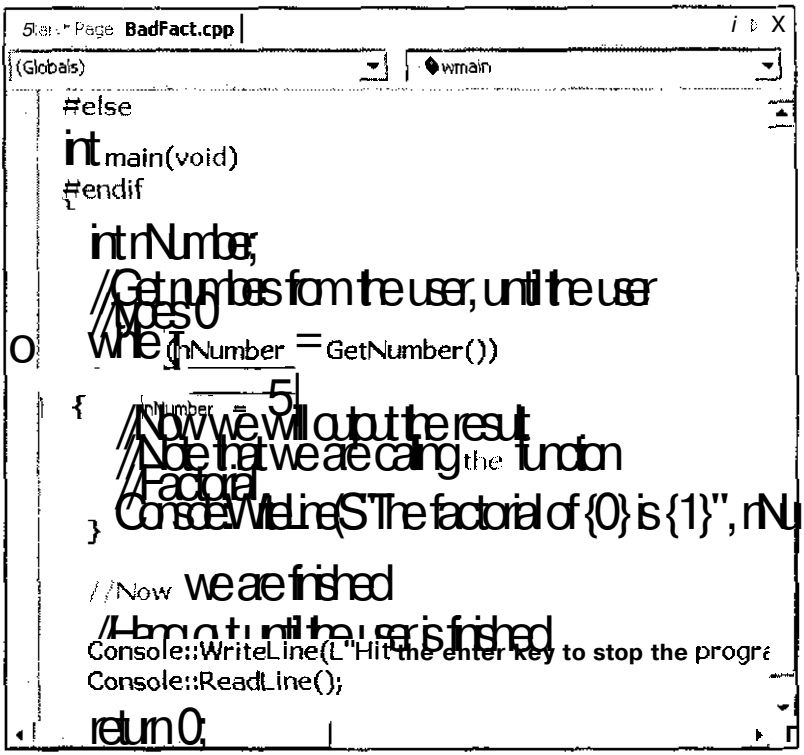


Рис. 16.6. DataTips показываетзначенияпеременныхбезкаких-либоусилий свашейстороны

Генеральнаяуборка

Хотите посмотреть, что такое отладка программы на практике? Далее в этой главе будут продемонстрированы различные приемы, позволяющие проверить правильность выполнения программы, установить наличие ошибок и затем устранить их.

Итак, приступим

Начнем с того, что создадим новую управляемую программу C++. Код этой программы показан ниже. Как и одна из рассмотренных ранее программ, эта программа отображает на экране факториал указанного пользователем числа.

```
#include "stdafx.h"
#using <mscorlib.dll>
using manespace System;

//Функция возвращает факториал числа
int Factorial(int nNumber)
{
  int nResult = nNumber;
  int i;    //Переменная для отсчета итераций

  //Цикл, на каждой итерации которого общий результат
```

```
  //умножается на значение переменной i
  for (i=0; i<=nNumber; i++)
    {
      nResult *= i;
    }
  //Возвращение результата
  return nResult;
}

//Функция просит пользователя набрать число,
//а затем возвращает это число как результат
int GetNumber()
{
  int nNumber;

  Console::WriteLine(S"Введите число");
  nNumber = Int32::Parse(Console::ReadLine());
  return nNumber;
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  int nNumber;

  //Получение чисел от пользователя до тех пор,
  //пока он не наберет число 0
  while (nNumber = GetNumber())
  {
    //Отображение значения факториала для указанного
    //числа. При этом используется значение,
    //возвращаемое функцией Factorial
    Console::WriteLine(S"Факториал числа {0} равен {1}",
                       nNumber.ToString(),
                       Factorial(nNumber).ToString());
  }

  //Ожидание, пока пользователь не остановит
  //выполнение программы
  Console::WriteLine(S"Нажмите клавишу Enter, чтобы
                      остановить выполнение программы");
  Console::ReadLine();
  return 0;
}
```

Если процесс создания управляемых проектов на C++ вызывает у вас затруднения, вернитесь к главе 3.

А теперь попытайтесь несколько раз подряд выполнить эту программу.

1. **Щелкните на кнопке Start или выберите команду Debug⇨Start (Отладка⇨Начать).**
2. **Наберите число и нажмите клавишу <Enter>.**
3. **Посмотрите на полученный результат.**

4. **Нажмите любую клавишу, чтобы закрыть программу.**
5. **Повторите этот процесс, начиная с шага 1.**

Вы увидите, что какое бы число вы не набирали, программа всегда будет возвращать нулевой результат. Но ведь это же неправильно!

Так где же ошибка?

Попытаемся определить, из-за чего программа не работает (точнее, работает, но не так, как нужно). В кодах для получения числа от пользователя вроде бы нет ничего подозрительного;

```
Console::WriteLine(S"Введите число");
nNumber = Int32::Parse(Console::ReadLine());
return nNumber;
```

Выполнять эти инструкции пошагово, по-видимому, нет необходимости. Вместо этого сосредоточим внимание на работе функции `Factorial`.

Почему именно эта функция? Потому что это единственная функция в программе, выполняющая действия, понимание которых может вызвать у вас затруднения. Отладку более сложных программ обычно производят небольшими частями — по одной функции или объекту за раз. Если вы убедитесь, что отдельные небольшие части программы работают правильно, значит, коды, в которых эти части объединяются, также будут работать правильно. Таким образом, вы всегда можете сосредоточиться на небольших, легких для понимания фрагментах программы. Иногда этот метод называют *структурной проверкой (structured testing)*.

Сначала можно установить точку останова перед той строкой, где вызывается эта функция. Для этого выполните следующее.

1. **Прокрутите окно редактора, для того чтобы увидеть коды функции** `Factorial`.

   Коды этой функции набраны в самом начале программы.

2. **Щелкните правой кнопкой мыши на строке** `int Factorial(int nNumber)` **и выберите команду Insert Breakpoint.**

   Напротив этой строки отобразится символ точки останова подобно тому, как показано на рис. 16.7.

Теперь запустите программу.

1. **Щелкните на кнопке Start.**

   Программа начнет работать, и на экране появится надпись, предлагающая ввести число.

2. **Наберите число.**

   Далее программа должна вызвать функцию `Factorial`. Это значит, что точка останова будет достигнута, в результате чего откроется окно отладчика. Напротив строки, которая должна будет выполнятся следующей, вы увидите желтую стрелку (рис. 16.8).

Теперь можно приступить к пошаговому выполнению функции `Factorial` и постараться определить, где именно прячется ошибка. Поскольку точно известно, что функция возвращает неправильный результат, а сам вычисляемый результат сохраняется как значение переменной `nResult`, вам нужно проследить, как значения этой переменной изменяются в процессе выполнения функции. Таким образом, вы рано или поздно поймете, почему функция каждый раз возвращает нулевое значение. Для просмотра значения переменной можно использовать окно Locals или подсказку DataTip. Либо можно воспользоваться окном Watch, для чего выполните следующее.

1. **В** окне **Watch** щелкните на пустой строке в колонке Name.
2. Наберите nResult.
3. Нажмите клавишу <Enter>.

Можно также воспользоваться средством QuickWatch.

1. В окне редактора щелкните правой кнопкой мыши на переменной nResult и в открывшемся меню выберите команду **QuickWatch.**
2. **В** окне **QuickWatch** щелкните на кнопке **Add Watch.**

   После этого нижняя часть экрана будет выглядеть приблизительно так, как показано на рис. 16.9.
3. Проследите за выполнением этой функции, щелкнув несколько раз на кнопке **Step Over.**

   Информация о значении переменной nResult, отображаемая в окне Watch, будет обновляться каждый раз при изменении этого значения.

В результате вы увидите, что сразу же после выполнения инструкции nResult *= i значение переменной nResult станет нулевым.

Более того, по мере выполнения программы это значение никогда не будет изменяться. Как вы узнаете, что значение переменной nResult равно нулю? Очень просто: для этого существует окно Watch (или Locals, или QuickWatch, или подсказка DataTip). Если вы используете окно Watch, то оно постоянно будет выглядеть так, как показано на рис. 16.10.

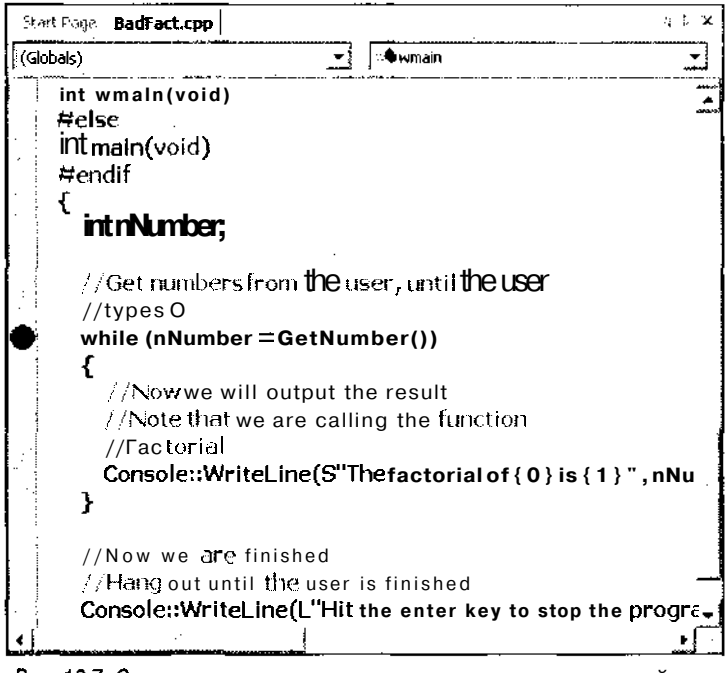


Рис. 16.7. Оналичииточкиостановасвидетельствуеткрасныйкружок,отображаемыйслеваотстроки

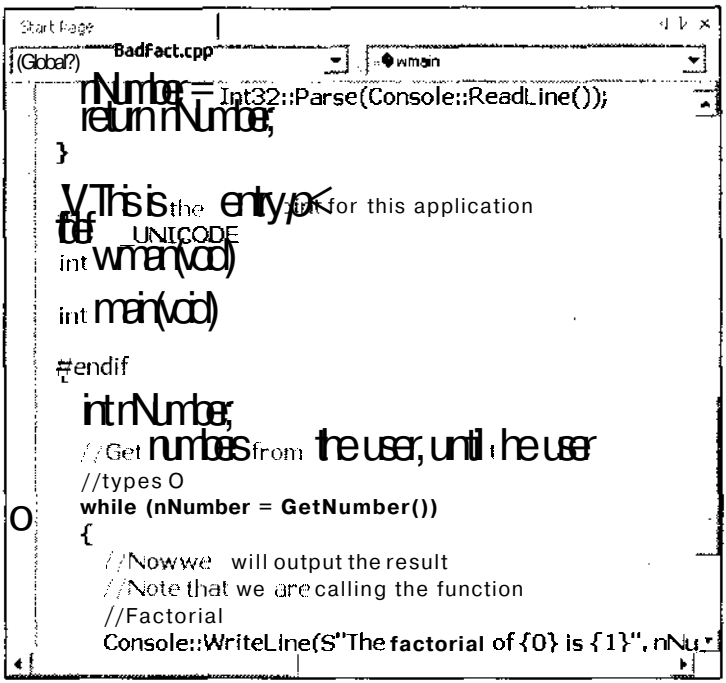


Рис. 16.8. В процессе пошагового выполнения программы значок в виде желтой стрелки отображается напротив строки, которая должна будет выполняться следующей

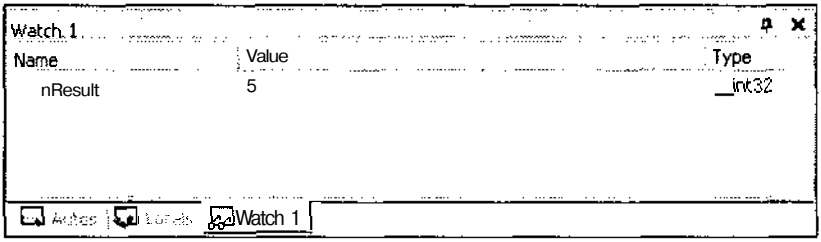


Рис. 16.9. В процессе отладки программы в окне Watch постоянно будет отображаться текущее значение переменной nResult

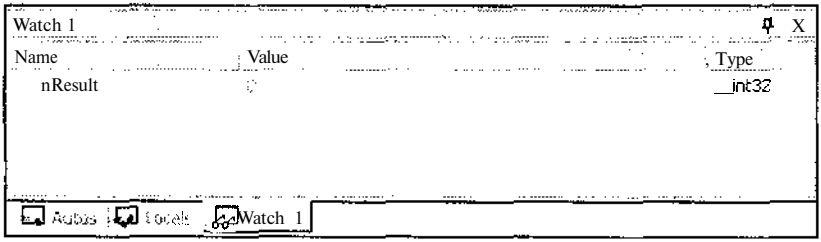


Рис. 16.10. Значение переменной nResult постоянно будет оставаться нулевым

Поскольку переменная nResult принимает нулевое значение именно в результате выполнения цикла for, вы точно можете сказать, что ошибку нужно искать именно там. Поэтому нужно более тщательно проанализировать коды этого цикла. Так как первая (и единственная) строка цикла умножает значение переменной nResult на значение переменной i, возможно, что-то не в порядке с переменной i. Если вы посмотрите на начало цикла for, то увидите, что значения переменной i изменяются от нуля до числа, введенного пользователем:

```
for (i=0; i<=nNumber; i++)
```

Это значит, что на первой итерации значение переменной i равно нулю. А поскольку умножение на ноль всегда дает нулевой результат, после выполнение инструкции nResult *= i значение переменной nResult становится нулевым и более не изменяется. Вот в чем проблема!

Что теперь?

Ошибка содержится в цикле for. Чтобы определить это, вам пришлось пошагово выполнять часть кодов программы, следить за изменением значения переменной nResult, анализировать работу кодов, после выполнения которых переменная nResult получает нулевое значение. В результате вы обнаружили, что проблему можно устранить, если переменная цикла будет изменяться не от нуля до n, а от единицы до n.

Теперь исправьте код прямо в окне редактора.

1. Щелкните в окне редактора.
   Это окно станет активным.
2. Найдите следующую строку:
   ```
   for (i=0; i<=nNumber; i++)
   ```
3. Измените ее так, как показано ниже.
   ```
   for (i=1; i<=nNumber; i++)
   ```
   После этого нужно опять запустить программу, чтобы убедиться, что теперь она работает правильно.
4. Выберите команду **Debug⇨Stop Debugging** (Отладка⇨Остановить отладку).
   Выполнение программы будет остановлено. (До этого ее выполнение было остановлено на промежуточной стадии, так как ранее вы установили точку останова.)
5. Щелкните на кнопке **Build** (Построить).
   Программа будет откомпилирована.
6. Щелкните на кнопке **Start** (Запустить).
   Программа снова будет запущена на выполнение.

Обратите внимание, что для завершения выполнения программы шаг 4 не обязателен — можно просто щелкнуть на кнопке Build. Отладчик знает, что в коды программы были внесены изменения, поэтому он спросит, хотите ли вы остановить отладку программы. Чтобы внесенные изменения вступили в силу, программа должна быть заново откомпилирована, поэтому щелкните на кнопке Yes. Отладка будет прекращена, а программа заново откомпилирована. После этого можете щелкнуть на кнопке Start, чтобы запустить программу снова.

Когда появится сообщение с просьбой ввести число, наберите, например, число 5 и посмотрите, как поведет себя программа. Переменная nResult больше не будет принимать нулевое значение, благодаря чему можно решить, что теперь программа работает правильно.

Но в программе по-прежнему есть ошибка

Минуточку! Если вы продолжите пошаговое выполнение программы, то увидите, что не все так хорошо, как хотелось бы. Что-то уж слишком быстро увеличивается значение переменной nResult.

Как и прежде, проблемы начинаются уже с первой итерации. В момент, когда вы переходите к выполнению цикла **for,** значение переменной nResult уже равно числу 5. На следующей итерации оно становится равным числу 10, затем числу 20 и т.д. Следя за этими изменениями в окне Watch, неизбежно приходим к выводу, что нужно заново разобраться в том, какое начальное значение присваивается этой переменной и каким образом оно потом изменяется.

Почему уже на первой итерации переменная nResult имеет значение 5? Еще раз подумайте о том, как программа должна работать, и сравните это с тем, что она делает на самом деле. (Точнее, нужно сравнить, что вы хотите от программы и какие инструкции вы оставили ей в действительности.) Чтобы вычислить факториал, нужно начать со значения переменной nResult, равного числу 1. затем умножить его на 2, затем на 3 и т.д. Проверим, так ли это было указано программе.

Если вы посмотрите на самые первые строки функции Factorial, то увидите, что переменная nResult объявляется и инициализируется следующим образом:

```
int nResult = nNumber;
```

Это неверно, и вот почему. Допустим, вам нужно вычислить факториал числа 5. Выполнение функции начинается с того, что переменной nResult присваивается значение 5. Затем оно умножается на число 1, потом на 2 и т.д. Таким образом получается результат, вычисленный как $5 \times 1 \times 2 3 x 4 x 5$ вместо $1 \times 2 \times 3 \times 4 \times 5$.Теперь становится понятно, что начальное значение переменной nResult должно быть равным числу 1.

Устранение ошибки

В программу нужно внести небольшое изменение так, чтобы начальное значение переменной nResult было равным числу 1. Для этого выполните ряд действий.

1. **Щелкните в окне редактора.**
   Окно редактора станет активным.
2. **Прокрутите окно редактора, чтобы найти строку**
   `int nResult = nNumber;`
3. **Внесите в эту строку следующую коррективу:**
   `int nResult = 1;`
4. **Щелкните на кнопке Build.**
5. **В ответ на вопрос, действительно ли вы хотите остановить процесс отладки программы, щелкните на кнопке Yes.**
   Программа будет откомпилирована.
6. **Щелкните на кнопке Start.**
   Программа вновь будет запущена на выполнение.

Вы можете быть довольны проделанной работой, поскольку теперь программа будет работать правильно. Например, если на запрос программы набрать число 6, то возвращен будет именно тот результат, который действительно соответствует факториалу числа 6 (рис. 16.11).

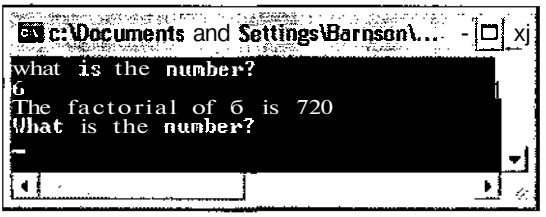


Рис. 16.11. Программа работает правильно

Последние штрихи

Теперь нужно удалить из программы точку останова и отменить отображение значения переменной в окне Watch (если вы использовали для просмотра значения именно это окно). Чтобы удалить точку останова, выполните ряд действий.

1. Выберите команду **Debug⇨Windows⇨BreakPoints** (**Отладка⇨Окна⇨Точки** останова).

   Откроется окно, в котором будет показан список всех установленных для программы точек останова (рис. 16.12).

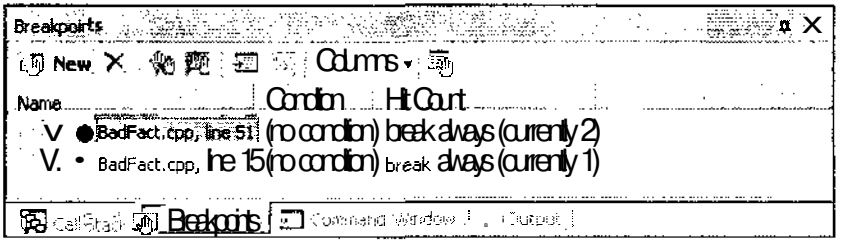


Рис. 16.12. В этом окне можно увидеть список всех точек останова, которые есть в программе

1. Выделите точку останова, которую хотите удалить.

3. Щелкните на кнопке **Delete** (Удалить).

   Выделенная на втором шаге точка останова прекратит свое существование. (Щелкните на кнопке Clear All Breakpoints (Удалить все точки останова), если хотите удалить сразу все точки останова, показанные в окне Breakpoints.)

Отменить отображение значений переменных в окне Watch несколько сложнее, так как получить доступ к этому окну можно только во время отладки программы.

1. Щелкните на кнопке **Step Into** или нажмите клавишу **<F11>.**

   Будет запущен процесс пошагового выполнения программы, и на экране отобразится окно Watch.

2. В окне **Watch** щелкните правой кнопкой мыши на названии переменной, отображение значения которой нужно отменить, и в открывшемся меню выберите команду **Delete Watch** (Удалить из окна).

3. Выберите команду **Debug⇨Stop Debugging** (**Отладка⇨**Остановить отладку), чтобы завершить выполнение программы.

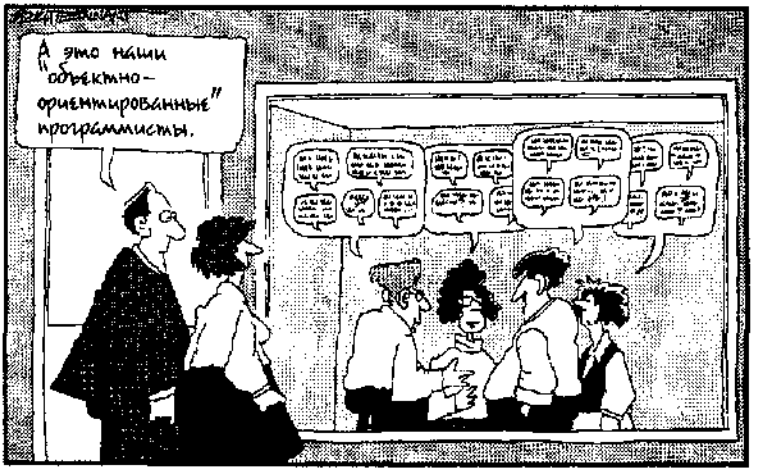
Теперь, когда вы снова запустите программу, ее выполнение не будет прерываться никакими точками останова. (Если не верите, можете проверить сами.)

Как видите, процесс отладки программы вовсе не сложен. Правда, нужно приобрести некоторый опыт, чтобы научиться быстро определять, в чем именно кроется ошибка. Чем больше вы будете программировать, тем лучше у вас это будет получаться.

Вообще говоря, отладчик может предоставить вам множество дополнительных возможностей. С его помощью можно, например, просмотреть значения регистра CPU, увидеть коды ассемблера, генерируемые компилятором, установить условные точки останова. Чтобы узнать больше о возможностях отладчика, поэкспериментируйте с командами меню Debug или просмотрите статьи об использовании отладчика, которые можно найти в справочной системе Visual C++.

Часть III

Экскурс в объектно-ориентированное программирование



В этой части...

О языке C++ вы уже знаете достаточно много. Но вот об объектно-ориентированном профаммировании (ООП) вы пока получили только самое общее представление. Третья часть устранит этот пробел в вашем образовании и познакомит вас с такими вещами, как **представление с** помощью объектов самых разнообразных явлений реального мира, использование принципа наследования для передачи функциональных возможностей одних **объектов** другим и т.п. Но главное — не забывайте о том, что **объектно-ориентированное** программирование может значительно упростить процесс создания, отладки и модификации программ, а также облегчить их чтение и понимание.

Глава 17

Смотрите на мир объективно

В этой главе...

- Общее представление о классах, членах данных и функциях-членах
- Создание объектов
- Права доступа для классов
- Статическое соответствие
- Создание объектно-ориентированной программы
- Освобождение памяти
- Создание функций доступа

Позвольте огласить две новости: одну хорошую, другую не очень. Хорошая состоит в том, что все основные идеи, изложенные в этой главе, довольно просты: вы научитесь создавать классы настолько быстро, насколько это возможно. Более того, каждый раз, когда вы использовали при написании программы встроенные функции .NET, в действительности вы использовали классы (так что некоторый опыт работы с классами у вас уже есть). Новость похуже заключается в том, что для того, чтобы научиться создавать правильно организованные классы, нужно потратить немало времени и усилий. Именно поэтому большинству программистов, которые только начинают осваивать азы объектно-ориентированного программирования, приходится по нескольку раз возвращаться к уже созданным классам, чтобы внести в них дополнительные изменения или вообще переписать их заново.

Что такое классы и с чем их едят

Классы являются фундаментальной составляющей объектно-ориентированного программирования (ООП). *Класс* — это структура, содержащая в себе некоторые данные и функции, предназначенные для обработки этих данных. Когда кто-то говорит о создании объектов, на самом деле он имеет в виду создание классов. Классы помогают моделировать состояние и поведение различных объектов реального мира. Они же значительно упрощают процесс написания и тестирования больших сложных программ. Кроме того, они предоставляют возможность наследования, что позволяет повторно использовать ранее написанные коды (это не просто удобно — это экономит массу времени и усилий).

Поскольку один класс объединяет в себе данные и функции, которые эти данные обрабатывают, понять, как работает тот или иной объект, довольно просто. По крайней мере проще, чем разобраться в кодах программы, где данные и функции разбросаны в произвольном порядке. Используя классы, вы можете не беспокоиться о том, какую библиотеку нужно подключить, чтобы передвигать картинку по экрану, или как найти домашний телефон сотрудника, занесенного в базу данных. Все это содержится в объекте. В идеале объект должен включать в себя все функции, необходимые для обработки содержащихся в нем данных.

Предположим, например, что у вас есть класс `LineObject` и вам нужно определить координаты линии или отобразить эту линию на экране. Обе функции должны содержаться

в этом же классе, поскольку они оперируют с данными этого класса. Следовательно, вам не придется просматривать коды всей остальной программы в поисках нужных данных или кодов, которые эти данные могут как-то изменить или обработать.

Изменить данные, содержащиеся в классе, или способы их обработки возможно только путем изменения элементов самого класса. Благодаря этому исключаются такие неприятные ситуации, при которых изменение одной глобальной переменной может нарушить работу всей программы. Используя классы, вы всегда сможете контролировать возможные последствия изменения отдельных переменных или их значений.

И в заключение отметим, что правильно организованные классы избавляют пользователя от необходимости вникать во все технические детали. Программист, который использует готовый класс, не должен знать, как именно сохраняются в нем данные и какие алгоритмы делают этот класс рабочим. Нужно только знать, как таким объектом можно манипулировать. Например, чтобы определить координаты линии, для хранения которых используется класс `LineObject`, можно просто вызвать функцию `Initialize`. Таким образом программисту, который использует этот класс, требуется всего лишь вызвать эту функцию и совершенно не нужно разбираться, какие при этом будут задействованы переменные или как именно будет использована функция `Console`.

Разберемся в деталях

Сюрприз! Если вы читали предыдущие главы и повторяли приведенные там примеры, значит, вы уже создали целую обойму классов. Это именно так, поскольку создание классов и создание структур — по сути, одно и то же (а структуры вы создавали уже не раз). Единственное отличие между классами и структурами в том, что классы включают в себя функции, а структуры — нет.

В этом разделе вы найдете все, что нужно знать для создания полноценных классов. Вы узнаете, что такое члены данных и функции-члены, как объявить класс и как ограничить доступ к отдельным его элементам, а также каким образом объявляются функции-члены.

Члены данных

Переменные, которые принадлежат какому-то классу, официально называются *членами данных*. (В терминологии Visual C++ они также обозначаются как *переменные-члены*, что, по сути, одно и то же.) Когда вы анализируете реальную проблему и моделируете ее в виде объекта, все то, что как-то описывает ее, представляется как члены данных.

Например, цвет, размер, цена, форма, вес, имя, исполнитель, продолжительность — это характеристики, которые могут описывать объект. Всю эту информацию можно сохранить как значения переменных и использовать в качестве членов данных различных классов.

Функции-члены

Функции-члены являются составной частью классов и используются для обработки хранящихся там данных. При анализе реальной проблемы все действия, которые можно совершать над объектом, моделируются в виде функций-членов.

Например, в отношении объектов могут быть предприняты такие действия, как настройка цвета, вычисление размеров, определение общей цены, отображение названия трека или исполнителя. Все это может быть смоделировано с помощью функций-членов.

Объявление классов

Для объявления классов используется ключевое слово class:

```
class ClassName
{
public:
  здесь описываются элементы данных и функции-члены, доступные
  пользователям этого класса
};
```

Предположим, вам нужно создать класс LineObject. За основу можете взять структуру PointList (она использовалась в качестве примера в предыдущих главах), добавив к ней функции и некоторые новые переменные:

```
__gc class LineObject
{
public:
   void  Initialize();
   void  Draw(Graphics *poG);
private:
   int m_nXFrom;
   int m_nYFrom;
   int n_nXTo;
   int m_nYTo;
};
```

Позже вы сможете определить, какие действия выполняют функции Initialize и Draw.

Обратите внимание, что класс LineObject был определен как класс, для которого активирована возможность сборки мусора (эта возможность рассматривалась в главе 13). В Visual C++ .NET обязательно объявлять классы именно таким образом. Если вы этого не сделаете, компилятор выдаст сообщение об ошибке.

Ограничение доступа

Вы можете защитить элементы данных отдельных классов от внешнего воздействия, сделав их закрытыми для всей остальной программы. Доступ к таким элементам данных имеют только функции-члены того же класса.

О том, что в классе LineObject есть такие переменные, как m_nXFrom, m_nYFrom, m_nXTo и m_nYTo, известно только функциям-членам Initialize и Draw. Для тех, кто будет просто использовать класс LineObject, эти переменные не существуют. Только функции-члены класса LineObject имеют к ним доступ и могут изменять их значения.

Как вы смогли убедиться на примере объявления класса LineObject, ограничить доступ к членам данных очень просто. После того как вы объявили открытые члены данных и функции-члены (использовав для этого ключевое слово public), наберите ключевое слово private. Затем перечислите закрытые члены данных, которые предназначены для использования исключительно внутри данного класса. (Можно создать также закрытые функции-члены. Они могут быть вызваны только другими функциями-членами этого же класса. По сути, это будут вспомогательные функции для открытых функций-членов, действия и возвращаемые результаты которых тем не менее не предназначаются для непосредственного использования за пределами этого класса.)

Если вы придерживаетесь соглашений о присвоении имен, принятых в данной книге, начинайте имена закрытых членов данных символами m_.

Защищенный доступ

До настоящего времени вы сталкивались только с двумя ключевыми словами, регулирующими права доступа: public и private. Члены данных и функции-члены, объявленные после слова public, являются открытыми (т.е. они могут быть использованы за пределами класса, в котором объявляются). Открытые члены данных и функции-члены создают внешний интерфейс класса. В свою очередь, внешний интерфейс — это то, к чему обращаются при использовании класса.

Закрытые члены данных и функции-члены (они объявляются после слова public) предназначены для внутреннего использования. Такие функции-члены не могут быть вызваны кодами, набранными вне данного класса. Точно так же значения членов данных с ограниченным доступом не могут быть прочитаны или изменены за пределами своего класса.

Использование закрытых членов данных и функций-членов позволяет создавать сложные классы с простым внешним интерфейсом. Это достигается благодаря тому, что доступ ко всем сложным вычислительным процессам, происходящим внутри класса, ограничивается и они становятся невидимыми для пользователей.

Права доступа регулируются еще одним ключевым словом — protected. Члены данных и функции-члены, объявленные после этого слова, могут быть использованы только функциями-членами этого же класса либо функциями-членами классов, *производных* от данного класса.

Определение функций-членов

После того как вы объявите, из чего состоит класс, можно приступить к определению того, какие именно действия будут выполнять функции-члены. Функции-члены определяются почти точно так же как и обычные функции (что было описано в главе 12). Но, поскольку функция-член является составной частью какого-то определенного класса, указывать нужно как имя функции, так и имя этого класса. (В конце концов, ведь несколько разных классов могут иметь, например, функцию Initialize.)

Чтобы определить функцию-член, наберите имя ее класса, далее два двоеточия (::) и затем имя самой функции. Эти два двоеточия называются *квалифицирующим оператором*. Этот оператор означает, что указанная функция-член (или член данных) является частью указанного класса. Например, код LineObject:: Draw указывает на функцию-член Draw класса LineObject, а код LineObject: :m_nXFrom ссылается на член данных m_nXFrom этого же класса.

Ниже приведен код, которым определяется функция-член Draw класса LineObject. Эта функция отображает линию на экране.

```
viod LineObject::Draw(Graphics *poG)
{
   //Создание пера для рисования
   Pen *poPen = new Pen(Color::Red);
   //Отображение линии
   poG->DrawLine(poPen, m_nXFrom, m_nYFrom, m_nXTo, m_nYTo);
}
```

Обратите внимание, что при вызове функции DrawLine вам не нужно набирать LineObject::m_nXFrom, поскольку внутри функции-члена не требуется использовать

квалифицирующий оператор (: :) при указании на члены данных. Внутри класса принадлежность члена данных к этому классу определяется автоматически. Таким образом, использование названия ra_nXFrom внутри функции-члена LineObject: : Draw равнозначно использованию кода SongList::m_nXFrom (SongList — название экземпляра класса LineObject). Также обратите внимание на то, что, поскольку Draw является функцией-членом класса LineObject, она имеет доступ к закрытым членам данных этого класса. (В то же время функции-члены других классов, например Circle: : Draw, не могут получить непосредственный доступ к закрытым членам данных класса LineObject.)

Что делать с готовыми классами?

После того как вы объявили класс и определили его функции, можно приступить к его использованию. Как и в случае со структурами, классы могут создаваться либо статически, либо динамически.

```
//Статически создаваемый класс
LineObject oFoo;
//Динамически создаваемый класс
LineObject *poBar = new SongList;
```

Обычно классы создаются динамически. (Классы .NET CLR всегда создаются динамически.)

Те же правила и принципы, которые относятся к статическим и динамическим переменным, распространяются и на статические и динамические классы. Создание классов обычно называют *созданием экземпляров классов* или *созданием объектов*.

Доступ к элементам класса

Классы — это те же структуры данных. Если нужно получить доступ к значению, сохраненному как член данных класса, используйте точку (.). Точно так же можно получить доступ и к функции-члену:

```
//Вызов функции Draw класса LineObject
LineObject oFirst;
oFirst.Draw();
```

Если вы используете указатель на экземпляр класса, функция вызывается несколько иначе:

```
LineObject *poFirst = new LineObject();
poFirst->Draw();
```

Подобно переменным, экземпляры классов (или объекты) имеют свое имя и тип.

Статическое соответствие

В главе 14 отмечалось, что можно использовать классы .NET ArrayList и Stack для хранения любых объектов, для которых включена возможность сборки мусора. Следовательно, если вы создадите класс и активизируете для него возможность сборки мусора, в последующем можно будет использовать ArrayList и Stack для хранения экземпляров этого класса. Это очень удобно, если необходимо работать с произвольными наборами элементов (вспомните набор линий, отображаемых на экране). Ниже показано, как можно использовать ArrayList для хранения объектов LineObject.

```
LineObject *poLine = new LineObject();
ArrayList *paLines = new ArrayList();
//Сохранение объекта LineObject в структуре ArrayList
paLines->Add(poLine);
```

Однако извлечь объект из структуры ArrayList будет немного сложнее. Поскольку в структуру ArrayList может быть записана самая разная информация, ей изначально не известно, как обращаться с каждым отдельным элементом. {Более подробно об этом речь идет в главе 19. Пока же мы просто даем краткий обзор такого явления, как *полиморфизм.)* Если вы хотите использовать объект, сохраненный с ArrayList, нужно будет сообщить, что собой представляет этот объект.

Предположим, например, что необходимо вызвать функцию-член Draw объекта LineObject, только что сохраненного в структуре ArrayList. Если вы непосредственно обратитесь к этому объекту и попробуете вызвать функцию Draw (как показано ниже), то получите сообщение об ошибке.

```
(poEnumerator->Current)->Draw(poG);
```

Причина в том, что компьютер не знает, что собой представляет элемент структуры ArrayList, к которому вы обращаетесь, и тем более не знает, что у этого элемента есть функция-член Draw. Чтобы объяснить ему, чем является этот элемент, нужно использовать ключевое слово static_cast (в переводе это означает установление статического соответствия}. Перед тем как указать на элемент, наберите это слово и за ним в угловых скобках укажите тип элемента. Если вернуться, например, к сохраненному в структуре ArrayList объекту LineObject, то, чтобы вызвать функцию Draw, нужно набрать такой код:

```
static_cast<LineObject*>(poEnumerator->Current)->Draw(poG);
```

Этот код как бы говорит компьютеру: "Возьми текущий элемент структуры ArrayList, рассматривай его как указатель на объект LineObject и, исходя из этого, вызови функцию Draw, которая является функцией-членом этого объекта".

Начинающие программисты часто путают названия классов и названия объектов. Если нужно сослаться на функцию-член Draw объекта oFirst, который является экземпляром класса LineObject, наберите код

```
oFirst.Draw()
```

Но ни в коем случае не набирайте

```
LineObject.Draw()
```

Другими словами, используйте название объекта, но не самого класса.

Имена, используемые функциями-членами

При написании кодов функций-членов не нужно набирать точку или символы ->, чтобы получить доступ к членам данных или к другим функциям-членам этого же объекта. Сама функция принадлежит объекту, поэтому на нее распространяется область видимости других элементов этого объекта. Следовательно, если внутри данного объекта объявлено имя x, любая функция-член может просто ссылаться на это имя.

Немкою практики

Теперь, когда вы имеете общее представление о классах, можно вернуться к программе, отображающей линии на экране, и сделать ее объектно-ориентированной. Всего нужно будет определить два класса. Объекты класса LineObject предназначены для хранения информации о каждой отдельной линии.

```
__gc class LineObject
{
public:
    void Initialize();
    void Draw(Graphics *poG);
private:
    int m_nXFrom;
    int m_nYFrom;
    int m_nXTo;
    int m_nYTo;
};
```

С этим классом вы уже встречались ранее. Второй класс, DisplayObject, сохраняет информацию обо всех отображаемых объектах. В частности, он содержит в себе структуру ArrayList, состоящую из объектов LineObject. Также у него есть функции-члены, предназначенные для добавления новых объектов LineObject и для отображения линий на экране.

```
__gc class DisplayObject
{
public:
    void Add();
    void Draw(Graphics *poG);
    void Initialize();
private:
    ArrayList *m_paLines;
};
```

Одной из наиболее интересных функций этого объекта является функция Draw. Она по порядку извлекает из списка ArrayList все объекты LineObject и для каждого из них вызывает функцию Draw (ту, которая принадлежит классу LineObject). Таким образом она использует класс LineObjectдля выполнения всей тяжелой работы.

```
//Отображение всех линий на экране
void DisplayObject::Draw(Graphics *poG)
{
    //Прссмотр всех объектов, представляющих линии
    IEnumerator *poEnumerator = m_paLines->GetEnumerator();
    while (poEnumerator->MoveNext())
        {
        //Вызов функции Draw для объекта
        static_cast<LineObject*>(poEnumerator->Current)->Draw(poG);

        }
}
```

Сама функция main этой программы очень проста. Она создает новый класс DisplayObject, инициализирует его, и вызывает метод Draw. Итак, вся программа выглядит следующим образом:

```
//Draw4

//Первая объектно-ориентированная программа

#include "stdafx.h"

# using <mscorlib.dll>
#using <System.Windows.Forms.dll>
#using <System.dll>
```

```
#using <System.Drawing.dll>

using manespace System;
using manespace System::Drawing;
using manespace System::Windows::Forms;
using manespace System::Collections;

//Сохранение информации о линиях
__gc class LineObject
{
public:
   void Initialize();
   void Draw(Graphics *poG);
private:
   int m_nXFrom;
   int m_nYFrom;
   int m_nXTo;
   int mnYTo;
};

//Инициализация объекта, представляющего линию
void LineObject::Initialize()
{
   //Чтение координат x и y для начальной и конечной
   //точек линии
   Console::WriteLine(S"Координата x начальной точки:");
   m_nXFrom = Int32::Parse(Console::ReadLine());
   Console::WriteLine(S"Координата y начальной точки:");
   m_nYFrom = Int32::Parse(Console::ReadLine());
   Console::WriteLine(S"Координата x конечной точки:");
   m_nXTo = Int32::Parse(Console::ReadLine());
   Console::WriteLine(S"Координата y конечной точки:");
   m_nYTo = Int32::Parse(Console::ReadLine());
}

//Рисование линии
void LineObject::Draw(Graphics *poG)
{
   //Создание пера для рисования
   Pen *poPen = new Pen(Color::Red);
   //Рисование линии
   poG->DrawLine(poPen, m_nXFrom, m_nYFrom, m_nXTo, m_nYTo);
}

//Сохранение координат всех линий, которые должны быть нарисованы
__gc class D.isplayObject
{
public:
   void Add();
   void Draw(Graphics *poG);
   void Initialize();
private:
   ArrayList *m_paLines;
};
```

```
//Инициализация структуры, состоящей из объектов DisplayObject
void DisplayObject::Initialize()
{
   m_paLines = new ArrayList();
}

//Добавление нового объекта
void DisplayObject::Add()
{
   bool fFinished = false;
   String *pszMore;

   while (!fFinished)
   {
      //Создание нового объекта, представляющего линию
      LineObject *poLine = new LineObject();
      //Его инициализация
      poLine->Initialize();
      //Сохранение его в структуре
      m_paLines->Add(poLine);

      //Вопрос к пользователю: хочет ли он добавить еще линию?
      Console::WriteLine("Нажмите y, чтобы указать координаты
                                          новой линии");
      pszMore = Console::ReadLine();
      if (!pszMore->Equals(S"y"))
      {
         fFinished = true;
      }
   }
}

//Отображение всех линий на экране
void DisplayObject::Draw(Graphics *poG)
{
   //Прссмотр всех объектов, представляющих линии
   IEnumerator *poEnumerator = m_paLines->GetEnumerator();
   while (poEnumerator->MoveNext())
   {
   //Вызов функции Draw для объекта
   static_cast<LineObject*>(poEnumerator->Current)->Draw(poG);
   }
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#eise
int main(void)
#endif
{
   //Структуры для рисования на экране графических элементов
   Form *poForm = new Form();
```

```
    Graphics *poGraphics = poForm->CreateGraphics();
    DisplayObject *poDisplay = new DisplayObject();

    //Инициализация объекта poDisplay
    poDisplay->Initialize();

    //Добавление нозой линии
    poDisplay->Add();

    //Открытие окна, в котором будут нарисованы линии
    poForm->Show();

    //Рисование линий

    poDisplay->Draw(poGraphics);

    //Освобождение памяти, выделенной для отображения графики
    poGraphics->Dispose();

    //Ожидание, пока пользователь не закроет форму
    Application::Run(poForm);

}
```

Управление памятью в неуправляемых программах

При написании программы Draw4 вам не нужно было беспокоиться об освобождении памяти, выделяемой для всех создаваемых объектов. Среда .NET делала это автоматически. Если вы создаете аналогичную неуправляемую программу (используя обычный C++), вам придется самим позаботиться о том, чтобы динамически выделяемая память по завершении выполнения программы была освобождена. Сделать это можно путем создания для каждого класса функции-члена Delete. (Другой возможный вариант рассматривается в главе 18.)

Кроме того, поскольку вы не сможете использовать класс .NET ArrayList, вам придется самостоятельно определить класс, задачей которого будет создание связанного списка для сохранения последовательности объектов. Приведенная ниже программа Draw5 является неуправляемой версией программы Draw4. Она содержит класс LineObjectContainer, предназначенный для создания связанного списка из объектов LineObject:

```
//Класс, представляющий отдельный элемент связанного списка
class LineObjectContainer
{
public:
    LineObject *poLine;
    LineObjectContainer *poNext;
    void Delete();
};
```

Обратите внимание, что этот класс содержит в себе функцию-член Delete, предназначенную для освобождения памяти. Эта функция вызывается тогда, когда список, состоящий из объектов LineObject становится более не нужным.

Поскольку класс LineObjectContainer использует для хранения информации связанный список, функция Delete находит каждый элемент этого списка и освобождает выделенную для него память:

```
//Освобождение памяти, выделенной для всего списка
void LineObjectContainer::Delete()
{
   LineObjectContainer *poCur, poTemp;

   poCur = this;
   while (poCur)
   {
      poTemp = poCur;
      //Освобождение памяти, выделенной для объекта LineObject
      delete poCur->poLine;
      poCur = poCur->poNext;
      //Освобождение памяти, выделенной
      //для объекта LineObjectContainer
      delete poTemp;
   }
}
```

Обратите внимание, что необходимо освобождать память, выделенную как для объекта LineObject (предназначен для хранения информации о координатах линии), так и для объекта LineObjectContainer (предназначен для создания связанного списка из объектов LineObject).

Ниже показана неуправляемая версия объектно-ориентированной программы, рисующей на экране линии. Поскольку здесь не могут быть использованы возможности .NET, автоматически создающие связанные списки и освобождающие динамически выделяемую память, неуправляемая версия будет несколько сложнее, чем управляемая.

```
//Draw5
//Неуправляемая программа

#include "stdafx.h"
#include <iostream.h>


//Сохранение информации о линиях
class LineObject
{
public:
   void Initialize();
   void Draw();
private:
   int m_nXFrom;
   int m_nYFrom;
   int m_nXTo;
   int ra_nYTo;
};

//Инициализация объекта, представляющего линию
void LineObject::Initialize()
```

```
}
   //Чтение координат x и y для начальной и конечной
   //точек линии
   cout << "Координата x начальной точки:";
   cin >> m_nXFrom;
   cout << "Координата y начальной точки:";
   cin >> m_nYFrom;
   cout << "Координата x конечной точки:";
   cin » m_nXTo;
   cout << "Координата y конечной точки:";
   cin >> m_nYTo;
}

//Рисование линии. Здесь просто отображаются значения координат
void LineObject::Draw()
{
   cout << "From " << m_nXFrom « "," « m_nYFrom << " to "
                << m_nXTo « "," << m_nYTo << endl;
}

//Класс, представляющий отдельный элемент связанного списка
class LineObjectContainer
{
public:
   LineObject *poLine;
   LineObjectContainer *poNext;
   void Delete();
};

//Освобождение памяти, выделенной для всего списка
void LineObjectContainer::Delete()
{
   LineObjectContainer *poCur, poTemp;

   poCur = this;
   while (poCur)
   {
      poTemp = poCur;
      //Освобождение памяти, выделенной для объекта LineObject
      delete poCur->poLine;
      poCur = poCur->poNext;
      //Освобождение памяти, выделенной
      //для объекта LineObjectContainer
      delete poTemp;
   }
}

//Создание связанного списка, состоящего из объектов
class ArrayList
{
public:
   void Add(LineObject *poLine);
   LineObject *MoveNext;
   void Delete();
   void Initialize();
```

```
private:
   LineObjectContainer *m_poFirst;
   LineObjectContainer *m_poCur;
   LineObjectContainer *m_poLast;
};

//Инициализация структуры
void ArrayList::Initialize()
{
   m_poFirst = 0;
   m_poCur = 0;
   m_poLast = 0;
}

//Добавление в список нового объекта
void ArrayList::Add(LineObject *poLine)
{
   //Создание нового элемента списка
   LineObjectContainer *poLOC = new LineObjectContainer ();
   //Присвоение ему начальных значений
   poLOC->poLine = poLine;
   poLOC->poNext = 0;

   //Присоединение его к списку

   if (!m_poFirst)
   {
      m_poFirst = poLOC;
      m_poCur = m_poFirst;
   }
   else
      m_poLast->poNext = poLOC;
   //Изменение указателя, ссылающегося на последний элемент списка
   m_poLast = poLOC;
}

//Переход к следующему элементу списка
LineObject *ArrayList::MoveNext()
{
   LineObject *poNext;

   //Если элементов больше нет, функция возвращает нуль
   if (!m_poCur)
      return 0;

   //Поиск следующего объекта, представляющего линию
   poNext = m_poCur->poLine;
   //Переход к следующему элементу списка
   m_poCur = m_poCur->poNext;
   return poNext;
}

//Освобождение памяти
void ArrayList::Delete()
{
```

```
   //Удаление указателя, ссылающегося на первый элемент списка
   m_poFirst->Delete();
}

//Сохранение всех линий, предназначенных для отображения
class DisplayObject
{
public:
   void Add();
   void Draw();
   void Initialize();
   void Delete();
private:
   ArrayList *m_paLines;
};

//Инициализация структуры DisplayObject
void DisplayObject::Initialize()
{
   m_paLines = new ArrayList();
   m_paLines->Initialize();
}

//Освобождение выделенной для него памяти
void DisplayObject::Delete()
{
   m_paLines->Delete();
}

//Добавление нового объекта
void DisplayObject::Add()
{
   bool fFinished = false;
   char cMore;

   while (!fFinished)
   {
      //Создание нового объекта, представляющего линию
      LineObject *poLine = new LineObject();
      //Его инициализация
      poLine->Initialize();
      //Сохранение его в структуре
      m_paLines->Add(poLine);

      //Хочет ли пользователь добавить еще одну линию?
      cout << "Нажмите y, чтобы ввести новую линию" << endl;
      cin >> cMore;
      if (cMore != "y")
      {
         fFinished = true;
      }
   }
}

//Отображение всех линий
```

```
void DisplayObject::Draw()
{
   LineObject *poLine;

   //Обращение к каждому элементу списка
   while (poLine = m_paLines->MoveNext())
   {
      //Вызов метода Draw для объекта
      poLine->Draw();
   }
}

int _tmain(int argc, _TCHAR* argv[])
{
   //Структура для отображения
   DisplayObject *poDisplay = new DisplayObject();

   //Инициализация созданного объекта
   poDisplay->Initialize();

   //Добавление линий
   poDisplay->Add();

   //Отображение линий
   poDisplay->Draw();

   //Освобождение памяти
   poDisplay->Delete();
   delete poDisplay;

   return 0;
}
```

Функции доступа

Вы только что увидели, насколько удобно держать члены данных закрытыми от остальной программы и использовать специальные функции для доступа к ним и для изменения их значений. Visual C++ .NET оптимизирует этот процесс, позволяя автоматически создавать функции доступа. (Это такие функции, которые присваивают значения членам данных и считывают их.) Вот как это делается:

```
__property int get_X();
__property void set_X(int i);
```

Этот код говорит о том, что у объекта есть свойство X. Если нужно прочитать значение этого свойства, можно вызвать функцию get_X () или набрать такой код:

```
n = foo.X;
```

Если же нужно присвоить свойству X какое-то значение, можно вызвать функцию `set_X()` или набрать код

```
foo.X = n;
```

Если вы объявите только функцию `get`, свойство станет доступным только для чтения.

Обратите внимание, что для того, чтобы сделать свойство рабочим, нужно будет еще определить функции get и set. Для этого обычно используются закрытые переменные, как показано в приведенном ниже коле.

```
//Accessors
//Использование функций доступа

#include "stdafx.h"
#using <mscorlib.dll>

using namespace System;

//В этом классе используются свойства для доступа к
//хранящимся в нем данным
__gc class PointObject
{
public:
   __property int get_X();
   __property void set_X(int i);
   __property int get_Y();
   __property void set_Y(int i);
private:
   int m_nX;
   int m_nY;
};

//Здесь определяется, что произойдет, если программа будет
//обращаться к свойствам X и Y. Обратите внимание, что для
//сохранения значений используются закрытые члены данных
int PointObject::get_X()
{
   return m_nX;
}

int PointObject::get_Y()
{
   return m_nY;
}

void PointObject::set_X(int i)
{
   m_nX = i;
}

void PointObject::set_Y(int i)
{
   m_nY = i;
}

//Ожидание, пока пользователь не остановит выполнение программы
void HangOut()
{
   Console::WriteLine(L"Нажмите Enter, чтобы остановить
```

```
                                выполнение программы");
   Console::ReadLine();
}
//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
   PointObject *poPoint = new PointObject();
   //Присвоение значений свойствам X и Y
   poPoint->X = 3;
   poPoint->Y = 2;

   //Отображение этих значений на экране
   Console::WriteLine(S"Значения X и Y: {0}, {1}", poPoint-
            >X.ToString(), poPoint->Y.ToString());

   HangOut();
   return 0;
}
```

Если вы пишете неуправляемую программу, также можно создать подобный эффект наличия свойств у объекта. Для этого можно использовать закрытые члены данных и открытые функции, которые обеспечивают к ним доступ. При этом нельзя будет ссылаться на сами свойства, как это делается в управляемых программах. Вместо этого нужно будет непосредственно вызывать функции-члены get и set.

Если для какой-то переменной определить только функцию get, она станет доступной только для чтения.

Ниже приведен код неуправляемой программы, в котором используются функции доступа.

```
//Accessors2
//Использование функций доступа
//Неуправляемая программа

#include "stdafx.h"
#include <iostream.h>

//В этом классе используются свойства для доступа к
//хранящимся в нем данным
class PointObject
{
public:
   int get_X();
   void set_X(int i);
   int get_Y() ;
   void set_Y(int i) ;
private:
   int m_nX;
   int m_nY;
};
```

```
//Здесь определяется, что произойдет, если программа будет
//обращаться к свойствам X и Y. Обратите внимание, что для
//сохранения значений используются закрытые члены данных
int PointObject::get_X()
(
   return m_nX;
}

int PointObject::get_Y()
{
   return m_nY;
}

void PointObject::set_X(int i)
(
   m_nX = i;
}

void PointObject::set_Y(int i)
(
   m_nY = i;
}

int _tmain(int argc, _TCHAR* argv[])
{
   PointObject *poPoint = new PointObject();
   //Присвоение значений свойствам X и Y
   poPoint->set_X(3);
   poPoint->set_Y(2);

   //Отображение этих значений на экране
   cout << "Значениея X и Y: " << poPoint->get_X() « ", " <<
               poPoint->get_Y() << endl;

   return 0;
}
```

Заголовочные файлы

Если программа состоит из нескольких исходных файлов, вам нужно будет объявить классы в заголовочном файле. Если класс необходимо будет использовать в каком-то исходном файле, наберите в нем директиву `#include` и укажите название заголовочного файла, который должен быть включен в этот исходный файл.

Если вы добавляете в класс или удаляете из класса члены данных «ли функции-члены, не забудьте обновить заголовочный файл. В противном случае вы получите сообщение наподобие такого: `'foo' : is not a member of 'baz'`. Расшифровывается оно приблизительно так: "Вы забыли обновить заголовочный файл для того, чтобы функция `foo` была включена в класс `baz`". Или так: "Вы указали неверный параметр, т.е. то, что было перечислено при объявлении класса, не совпадает с тем, на что вы ссылаетесь при его использовании".

Общие рекомендации

Создавая объектно-ориентированную программу, постоянно учитывайте особенности описываемой ею задачи. Старайтесь создавать объекты, максимально точно отображающие те элементы, которые присутствуют в модели. (Например, если создается объектно-ориентированная программа, моделирующая работу музыкального автомата, нужно создать объект, представляющий сам музыкальный автомат, и объект, представляющий список воспроизводимых записей.)

При создании классов придерживайтесь приведенной ниже последовательности действий.

1. **Проанализируйте проблему.**
2. **Отдельно выделите данные, которыми нужно будет манипулировать.**

   Что это за данные? Как вы будете их обрабатывать?
3. **Разбейте данные и функции на группы, чтобы определить, из чего будет состоять каждый объект.**
4. **Скройте детали выполняемых операций.**

   Для управления объектом используйте функции высокого уровня, так чтобы пользователь не знал, например, что имена сохраняются в массиве или что записи извлекаются из связного списка.

Приведем некоторые общие рекомендации, которыми следует руководствоваться приступая к проектированию классов.

- ✓ **Посмотрите, какие классы создают другие программисты.** Вы можете найти для себя много удачных решений.
- ✓ **Начинайте с простых классов.**
- ✓ **Думайте о том, как можно использовать созданный класс в других частях программы или в тех программах, которые вы** будете **создавать в** будущем. Иногда, чем из меньшего количества элементов состоит класс, тем больше возникает возможностей для его повторного использования. Можете попытаться выделить наиболее общие свойства или характеристики, чтобы на их основе создать базовый класс. Помните, что с помощью возможности наследования можно будет затем построить рабочую среду, максимально отвечающую вашим требованиям.
- ✓ **Проверьте, нет ли в программе таких элементов, которые используются как единое целое.** Другими словами проверьте, есть ли в программе такие фрагменты данных, которые до использования ООП обрабатывались сразу многими процедурами. Именно из них можно создавать хорошие классы. Предположим, например, что у вас есть структура, предназначенная для хранения информации о сотрудниках, и есть набор различных процедур, вычисляющих размеры начисленной заработной платы, количество предоставленных отгулов и т.п. Все это можно объединить для создания класса, управляющего информацией о сотрудниках.
- ✓ **Если вы создали объект, все процедуры, управляющие этим объектом, должны быть его функциями-членами.** Каждый объект должен быть самодостаточным. Он сам должен знать, как обрабатывать содержащиеся в нем данные.
- ✓ **Если в программе есть общие характеристики, которые повторяются снова и снова с небольшими вариациями, создайте базовый класс для представления этих характеристик.**

- ✓ **Решите, должны ли пользователи классов знать о внутренней структуре классов аля того, чтобы иметь возможность их использовать.** Правильнее создавать такие объекты, которые не требуют для понимания их внутреннего устройства. Представим, например, что у вас есть объект Play, содержащий в себе массив для хранения информации о продолжительности музыкальных записей. Пользователь этого объекта должен иметь возможность найти интересующие его данные, не вникая в то, что для их хранения используется массив. В объекте должна быть функция-член, которая сама найдет нужное значение и вернет его в качестве результата. Такая практика значительно упрощает работу с объектами.
- ✓ **Помните, что поначалу все это может показаться несколько сложным.**
- ✓ **Не бойтесь перечеркнуть все сделанное ранее и начать заново: все через это проходят.** Воспринимайте это просто как часть образовательного процесса.

Глава 18

Конструкторы и деструкторы

В этой главе...

- Для чего нужны конструкторы и деструкторы
- Создание нескольких конструкторов
- Советы о том, как правильно читать коды объектно-ориентированных программ

Конструкторы и деструкторы — это специальные функции, предназначенные для инициализации объектов и освобождения выделяемой для них памяти. И эта глава посвящена именно им.

Работа до и после

Каждый из нас любит вкусно поесть, но вот процесс приготовления пищи и последующее мытье посуды способны испортить общий "праздник жизни". В программировании та же картина: создавать хорошие программы — одно удовольствие, если не учитывать всяческую подготовительную работу и последующую уборку программного мусора.

Однако не стоит потакать своей лени, так как подготовительный и завершающий этапы имеют при создании программы очень важное значение. Если, например, вы забудете инициализировать переменную (т.е. не присвоите ей начальное значение), это может привести к таким последствиям, как отображение на экране не того цвета, не тех данных или зависание программы. Аналогично, если вы забудете написать коды, освобождающие выделяемую память, это может привести к тому, что программа будет работать все медленнее и медленнее, пока вовсе не зависнет.

К счастью (а вы ведь знали, что не может все быть настолько ужасно), в C++ есть две встроенные возможности— конструкторы и деструкторы, заботящиеся о том, чтобы переменные инициализировались, а ненужная память вовремя освобождалась. *Конструктор* — это процедура, которая автоматически вызывается в момент создания объекта (он же — экземпляр класса). Вы можете помещать инициализирующую процедуру внутри конструктора, чтобы иметь гарантию, что объект будет настроен должным образом, когда вы приступите к использованию. Можно даже создать несколько конструкторов, чтобы иметь возможность инициализировать объект разными способами.

Когда объект становится более не нужным (это происходит, если использующая его функция заканчивает работу или вы набираете для него команду delete), автоматически вызывается функция, называемая *деструктором*. Если по какой-то причине вас не устраивает тот способ, которым среда .NET автоматически освобождает выделяемую память, можете подкорректировать его, воспользовавшись деструктором.

Подготовительный этап

Конструктор — это функция, вызываемая каждый раз в момент создания нового объекта. Если объект содержит члены данных, которые должны быть инициализированы, добавьте в конструктор соответствующие коды.

Не стоит делать конструктор слишком сложным или добавлять в него коды, без которых можно было бы обойтись. Всегда желательно, чтобы коды конструктора были краткими, простыми и понятными.

Конструктор имеет такое же имя, как и соответствующий класс. Так, конструктор класса `DisplayObject` будет называться `DisplayObject::DisplayObject`. Конструктор класса Dog будет называться `Dog::Dog`.

Конструктор никогда не возвращает значения в качестве результата.

Чтобы создать конструктор, объявите его вначале внутри класса, Ниже, например, показан код объявления класса `DisplayObject`,для которого будет создан конструктор:

```
__gc class DisplayObject
{
public:
   void Add();
   void Draw(Graphics *poG);
   DisplayObject();
private:
   ArrayList *m_paLines;
};
```

Далее нужно написать коды для объявленного конструктора. В программе Draw4 класс `DisplayObject` имеет функцию-член `Initialize`, которая присваивает указателю адрес списка `ArrayList`. Этот же код можно набрать внутри конструктора `DisplayObject`:

```
DisplayObject::DisplayObject()
{
   m_paLines = new ArrayList();
}
```

Каждый раз при создании экземпляра класса `DisplayObject` этот конструктор будет вызываться автоматически. Теперь необходимость в функции `Initialize` отпадает, и более не нужно беспокоиться о том, чтобы она вызывалась для каждого нового объекта. Этот простой прием позволит вам избавиться от лишней головной боли.

Конструктор объявляется внутри класса как одна из его функций-членов. Он может быть закрытым (`private`), открытым (`public`) или защищенным (`protected`).

Много конструкторов — много возможностей

Иногда возникает необходимость по-разному создавать объекты одного и того же класса. Возможно, например, что при создании объекта `LineObject` вам потребуется либо задать начальные координаты для первой фигуры, либо установить ограничение на количество отображаемых фигур. Для этого можно создать сразу несколько конструкторов. Один будет принимать значения параметров и создавать обычный объект `LineObject`. Другой будет также принимать значения параметров и использовать их для инициализации или изменения объекта.

Можете создать столько конструкторов, сколько будет необходимо. Каждый конструктор должен иметь свойственный только ему набор параметров. Например, один может требовать число типа integer, второй — два числа типа double и т.д.

Значения параметров передаются в момент создания объекта. В зависимости от передаваемого набора параметров, вызывается тот или иной конструктор. Например, если при создании объекта вы передаете одно число типа integer, вызывается тот конструктор, которому требуется значение одного параметра типа integer. Если передаются два числа типа double,

вызывается конструктор, которому нужны два числа типа double. Если компилятор не может найти соответствие между передаваемым набором параметров и наборами параметров существующих конструкторов, он выдает сообщение о синтаксической ошибке.

Класс с несколькими конструкторами прост в применении. В действительности его можно многократно использовать в кодах программы. Каждый раз, когда нужно создать новый объект и передать ему значения параметров, вы вызываете параметризованный конструктор. Вот, например, код, взятый из программы Draw4, в котором используется один из нескольких конструкторов класса `Pen`:

```
Pen *poPen = new Pen(Color::Red);
```

При этом для создаваемого объекта `Pen` уже будет определен цвет. Однако одновременно можно задать и толщину отображаемой линии, набрав такой код:

```
Pen *poPen = new Pen(Color::Red, 5);
```

В данном случае также создается объект `Pen`, но при этом уже используется другой конструктор.

Создается класс с несколькими конструкторами также довольно просто. Все, что вам нужно сделать, — это объявить эти несколько конструкторов и затем не забыть написать для каждого из них коды. Ниже, например, приведена одна из версий класса `DisplayObject`, включающего в себя два конструктора:

```
__gc class DisplayObject
{
public:
   void Add();
   void Draw(Graphics *poG];
   DisplayObject();
   //В следующей строке объявляется еще один конструктор
   DisplayObject(Color *poInitalColor);
private:
   ArrayList *m_paLines;
   Color *m_poInitalColor;
};

//Определение конструктора
DisplayObject::DisplayObject()
{
   m_paLines = new ArrayList();
}

//Конструктор, вызываемый, если указывается цвет
DisplayObject::DisplayObject(Color *poInitalColor)
{
   m_paLines = new ArrayList ();
   m_poInitalColor = poInitalColor;
}
```

Помните, что каждый конструктор должен иметь уникальный набор параметров. Например, хотя приведенные ниже конструкторы выглядят по-разному и используют аргументы с разными именами, количество аргументов и их тип совпадают. Это будет воспринято компилятором как синтаксическая ошибка.

```
Window::Window(int Left, int Right);
Window::Window(int Width, int Color);
```

Открытые и закрытые конструкторы

Хорошей практикой является создание открытых и закрытых конструкторов. Открытые конструкторы могут использоваться для создания новых объектов, если это создание инициируется кодами программы, которые к данному классу не относятся. Закрытые конструкторы используются внутри класса как вспомогательные функции. Например, если создается связанный список, состоящий из объектов данного класса, можно использовать закрытый конструктор. (Этот конструктор можно сделать закрытым, поскольку он будет вызываться только функцией-членом этого же класса, отвечающей за создание списка.)

Заметание следов

Деструктор — это функция-член, автоматически вызываемая в момент ликвидации объекта. Объекты могут ликвидироваться по разным причинам. Например объект, создаваемый и используемый какой-нибудь функцией, уничтожается тогда, когда эта функция заканчивает работу. Или же объект, созданный командой new, может быть впоследствии удален командой delete. Или, допустим, программа завершает свою работу и все созданные ею объекты подлежат удалению.

Компилятор вызывает деструктор автоматически. Вам вовсе не обязательно беспокоиться о том, чтобы он был вызван, а также вы никак не сможете передать ему значения каких-то параметров. Конструктор имеет то же имя, что и класс, перед которым стоит символ тильды (~). Класс может иметь только один деструктор, и он должен быть открытым (public).

Вот как выглядят коды деструктора:

```
//Класс для создания связанного списка
class  LineObjectContainer
(
public:
   LineObject  *poLine;
   LineObjectContainer  *poNext;
   ~LineObjectContainer();
};

//Деструктор: освобождение памяти, занимаемой списком
LineObjectContainer::~LineObjectContainer()
{
   //Для наглядности отображается строка на экране
   cout << "Удаление связанного списка\n";

   //Удаление линии
   delete poLine;
   //Удаление следующего элемента списка
   if (poNext)
      delete poNext;
}
```

Когда объект больше не нужен

Если вы создаете управляемую программу (используя возможности .NET), среда .NET автоматически освобождает выделяемую память или по крайней мере делает это для тех объектов, для которых активизирована возможность сборки мусора. Поэтому в .NET не обязательно всегда собственноручно создавать деструкторы. Однако

при написании неуправляемых программ возможность сборки мусора недоступна, поэтому создание деструкторов становится жизненно необходимой задачей.

Поскольку у каждого класса есть свой деструктор, каждый из них в состоянии самостоятельно освободить память, выделяемую для его объектов. Поэтому вам не нужно в конце программы набирать коды для освобождения сразу всей выделенной памяти. Можно делать это для каждого класса отдельно и тогда, когда это нужно.

Если вам сейчас кажется, что на самом деле можно обойтись и без деструкторов, давайте и[illegible] немея к программе Draw5. Классы LineObjectContainer, ArrayList и DisplayOb[illegible] включают в себя функцию Delete для очистки памяти. Программа Draw5 проходит длинный путь, чтобы в нужной последовательности вызвать процедуры Delete для всех объектов, которые были созданы в процессе выполнения этой программы. Использование деструкторов значительно упрощает этот процесс. Вы можете перепоручить им всю выполняемую функциями Dele[illegible] работу, а сами функции Delete отбросить за ненадобностью. Более того, при этом упрощается сама процедура очистки памяти. В программе Draw5 функция LineObjectContainer::Del[illegible] например, выглядит довольно внушительно:

```
void LineObjectContainer::Delete()
{
   LineObjectContainer *poCur, poTemp;

   poCur = this;
   while (poCur)
   {
      poTemp = poCur;
      //Освобождение памяти, выделенной для объекта LineObject
      delete poCur->poLine;
      poCur = poCur->poNext;
      //Освобождение памяти, выделенной
      //для объекта LineObjectContainer
      delete poTemp;
   }
}
```

Как вы помните, эта функция вызывается тогда, когда программа завершает работу. Вызывается она вначале для первого элемента связанного списка, а затем поочередно обращается к каждому последующему элементу. (Ну хорошо, вы, скорее всего, об этом не помните. На самом деле, когда вы просматривали эти коды в главе 17, то наверняка думали о чем-то своем или просто считали овечек.)

С деструкторами все становится намного проще. Когда вы удаляете объект, соответствующий ему деструктор вызывается автоматически, а функцию LineObjectContainer::Delete можно преобразовать к более совершенному виду:

```
//Деструктор: освобождение памяти, выделенной для списка
LineObjectContainer::LineObjectContainer()
{
   //Для наглядности отображается строка на экране
   cout << "Удаление связанного списка\n";

   //Удаление линии
   delete poLine;
   //Удаление следующего элемента списка
   if (poNext)
      delete poNext;
}
```

Этот код очень прост. Первый объект списка ArrayList освобождает занимаемую память и затем удаляет следующий объект списка, который также является объектом класса LineObjectContainer. Следовательно, такой же деструктор вызывается и для этого объекта. Что он делает? Освобождает занимаемую этим объектом память и удаляет следующий объект. И так далее, пока не будет удален последний объект списка.

Каким образом будет вызван деструктор для первого объекта в списке ArrayList? Сделает это деструктор класса ArrayList:

```
ArrayList::~ArrayList()
{
   //Удаление связанного списка
   //Все, что нужно сделать, — удалить первый элемент списка
   delete m_poFirst;
}
```

А каким образом будет вызван этот деструктор? Список ArrayList используется классом DisplayObject. Поэтому, когда удаляется объект класса DisplayObject, соответствующий ему деструктор удаляет объект класса ArrayList:

```
DisplayObject::~DisplayObject()
{
   delete m_paLines;
}
```

Ну и теперь осталось выяснить, каким образом будет вызван деструктор для объекта DisplayObject. Это сделает строка функции main, набранная в самом конце программы:

```
delete poDisplay;
```

Поскольку деструкторы работают именно так. как описано выше, вы можете писать коды для очистки памяти небольшими порциями отдельно для каждого создаваемого класса. Вам не нужно ждать, пока программа завершит всю работу, и затем набирать коды для освобождения всей выделенной памяти. пытаясь вспомнить каждый созданный объект.

Смысл использования деструкторов можно пояснить примером из обычной жизни: если вы чем-то воспользовались и уже забыли об этом. например проехали в трамвае, прочитали книгу или выпили банку пива, то вам не нужно таскать за собой прокомпостированные билеты, старые книги или какие-то жестянки.

Не забывайте также освобождать динамически выделяемую память

Если РЫ создаете объект или переменную динамически, т.е. используете для этого ключевое слово new, вы также должны удалить их, когда потребность в них отпадет. Предположим, например, что вы создаете класс для хранения информации, представляющей фотоснимки, и мри выделении памяти для объектов этого класса используете команду new. Чтобы освободить эту память, нужно будет применить команду delete. В противном случае ее нельзя будет использовать для хранения другой информации. (Если вы просто удалите указатель на этот фрагмент памяти, он останется запятым старыми данными, создавая таким образом эффект "утечки памяти".)

Или предположим, что у вас есть класс, включающий в себя связанный список, состоящий из набора воспроизводимых музыкальных записей. Удаляя объект этого класса вы должны также освободить память, которая выделяется для хранения этого списка. Иными словами, нужно позаботиться о том. чтобы был вызван деструктор этого связанного списка.

Лучший способ реализации такого деструктора — включение в него команды. удаляющей следующий элемент связанною списка. Таким образом, стоит вам только удапить первый

элемент списка, все остальные элементы будет удалены автоматически. Полная аналогия с эффектом домино.

Не забывайте также, что при написании управляемых программ вам обычно не придется слишком беспокоиться о таких вещах, поскольку возможность сборки мусора сделает всю грязную работу автоматически.

Конструкторы и деструкторы классов

По мере приобретения опыта создания объектно-ориентированных программ вы неизбежно придете к выводу, что очень удобно создавать классы, содержащие в себе другие классы. Например, внутри класса Семья вполне логичным было бы использование класса Жена (или Теща).

При создании экземпляра класса первым делом выделяется память для всех его членов данных. Если внутри класса есть члены данных, которые сами являются классами (например, такой член данных, как Теща), для них вызываются конструкторы.

Таким образом, если класс содержит в себе другие классы, первыми вызываются конструкторы для этих классов, поскольку они должны быть созданы до того, как будет создан основной класс.

Когда экземпляр класса уничтожается, вызывается деструктор. Затем вызываются деструкторы для всех членов данных, которые сами являются классами. Например, когда удаляется объект класса Семья, вызывается соответствующий ему деструктор, а после него вызывается деструктор, уничтожающий класс Теща.

Возможно, вам будет легче это понять, если вы просмотрите код, приведенный ниже. В этой программе создается класс foo, который содержит в себе другой класс — bar. Если вы запустите программу, то увидите, что вначале вызывается конструктор класса bar, а затем конструктор класса foo. Когда программа заканчивает работу, вначале вызывается деструктор класса foo и только потом деструктор класса bar:

```
//ConstructorDestructor
//Программа демонстрирует порядок, в котором вызываются

//конструкторы и деструкторы

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;
//Простой класс, состоящий из конструктора и деструктора
class bar
{
public:
   bar();
   ~bar();
};

//Пусть все знают, что создается объект класса bar
bar::bar()
{
   Console::WriteLine(S"Создается объект класса bar");
}
```

```
//Пусть все знают, что объект класса bar уничтожается
~bar::bar()
{
   Console : :WriteLine(S"Объект класса bar уничтожается") ;
}

//foo — это класс, внутри которого содержится другой класс,
//в данном случае — bar. При создании и удалении объекта
//класса foo также вызываются соответствующие ему конструктор
//и деструктор
class foo
{
public:
   bar oTemp;
   foo();
   ~foo();
};

//Пусть весь мир знает, что рождается объект класса foo
foo::foo()
{
   Console::WriteLine(S"Создается объект класса foo");
}

//Пусть весь мир знает, что объект класса foo уничтожается
~foo::foo()
{
   Console::WriteLine(S"Объект класса foo уничтожается");
}

//Это функция main, которая всего лишь создает объект класса
//foo, которому присваивается имя oTemp. При этом автоматически
//вызываются конструкторы. Когда программа завершает свою
//работу, объект oTemp автоматически уничтожается и вы можете
//видеть, в каком порядке вызываются деструкторы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
   foo oTemp;
   return 0;
}
```

Чтение кодовобъектно-ориентированных программ

Вы уже заметили, что объектно-ориентированные программы могут состоять из множества классов. Каждый класс должен быть объявлен для того, чтобы сообщить компилятору, из чего он будет состоять, и затем должны быть определены все его функции-члены. Коды, которыми класс объявляется, обычно лаконичны и не занимают много места, но вот определе-

ние функций-членов может растянуться не на одну страницу. Поскольку вам придется читать коды функций-членов, чтобы точно понять, что они делают, изучение работы класса может потребовать переходов от одного фрагмента кодов к другому, набранному совершенно в другой части файла.

Приведем несколько советов, которые смогут облегчить процесс изучения работы класса.

- ✓ **Начните чтение кодов с функции** `main`, **чтобы увидеть, что она делает.** Начинайте изучение программы с классов верхнего уровня и их функций-членов, затем спускайтесь к классам более низкого уровня для ознакомления с более детальной информацией.
- ✓ **Если файл содержит несколько классов, просмотрите вначале коды, которыми эти классы объявляются.** Читайте все комментарии, оставляемые программистами, которые эту программу создавали, чтобы иметь представление, какой класс для чего нужен. Затем бегло просмотрите, какие задачи выполняют различные функции-члены и члены данных.
- ✓ **Проверьте, сколько конструкторов создано для каждого из классов.** Наверное, вы могли видеть, как некоторые конструкторы используются функцией `main`.
- ✓ **Оставьте рассмотрение закрытых и защищенных функций-членов и членов данных на потом.**
- ✓ **При изучении неуправляемых программ вам, возможно, придется обратиться к заголовочному файлу, чтобы найти коды объявления классов.** Вначале посмотрите, подключает ли исходный файл к себе какие-либо заголовочные файлы. (Эту операцию выполняют строки, которые начинаются с ключевого слова `#include`.) Некоторые классы могут объявляться в одном из заголовочных файлов.
- ✓ **Обычно класс самого верхнего уровня объявляется последним, поэтому лучше начинать читать коды с конца.** Начните с функции `main`. Далее поднимайтесь вверх, чтобы найти объявление класса. (Пропустите коды, которыми определяются функции-члены этого класса.) Просмотрите это объявление. Затем поднимайтесь вверх, чтобы найти объявление следующего класса. Вообще, если у вас возникают с чем-то трудности, попробуйте сделать это задом наперед. Возможно, это не решит проблему, но по крайней мере вы развеселите себя и своих коллег.

Глава 19

Наследование

В этой главе...

- Что такое наследование кодов
- Производные классы
- Наследование закрытых и защищенных членов
- Что такое виртуальные функции
- Абстрактные базовые классы

При обычном способе программирования написание программ может затягиваться на длительное время. Например, если вы создаете процедуру, то не можете использовать ее в дальнейшем, поскольку новая процедура должна чем-то слегка отличаться от этой. Или, допустим, вы создаете процедуру, которая отлично работает в каких-то ситуациях, но есть и другие случаи, когда эта процедура должна работать несколько иначе. Как обычно выходят из такого положения? Копируют коды созданной процедуры, вставляют их в новое место, редактируют и затем присваивают этой процедуре новое имя. Таким образом, вы как бы повторно используете написанные коды, однако данный метод не лишен недостатков. Копируя и вставляя коды, вы не только раздуваете размеры программы, но и способствуете размножению ошибок, ведь если ошибка содержалась в скопированном коде, она многократно проявится во всех частях программы, куда этот код будет вставлен.

Объектно-ориентированное программирование позволяет избежать подобных проблем. Вы можете просто взять существующий фрагмент кодов, наследовать его и внести все необходимые модификации. И никакого копирования — вы просто повторно используете то, что уже создано и работает. Эта возможность называется *наследованием кодов.*

Что происходит при наследовании кодов

Возможность наследования становится особенно полезной при построении объектов. Предположим, например, что у вас есть класс, представляющий информацию о человеке. Этот класс можно наследовать, чтобы создать новый класс, представляющий политика. Затем созданный класс можно снова наследовать, чтобы получить новый класс, представляющий честного политика. (Хорошо, потребность в последнем классе у вас вряд ли когда-нибудь появится, но в данном случае мы просто рассматриваем общую идею.) Все новые объекты, созданные таким образом, могут наследовать свойства тех объектов, из которых они были получены, а также могут иметь модифицированные или добавленные возможности.

Кроме того, если вы обнаружите и исправите ошибку в базовом классе, эта же ошибка автоматически будет исправлена и для всех производных классов. Другими словами, если вы увидите, что в классе, представляющем политика, засела какая-то ошибка, исправьте ее, и она автоматически будет также исправлена в классе, представляющем честного политика.

Чтобы воспользоваться возможностью наследования, при объявлении нового класса после его имени наберите двоеточие (:) и затем укажите название класса, коды которого будут наследоваться.

```
class ПроизводныйКласс : БазовыйКласс
{
};
```

Все функции-члены базового класса (он также называется *родительским*) становятся функциями-членами производного класса (которым называется также *дочерним*). То же самое касается и членов данных. Таким образом, вам не нужно заново набирать для них коды. Если вы хотите добавить в новый класс какие-то элементы, перечислите их при его объявлении. Именно эта особенность всех добавленных элементов будет отличать новый класс от базового.

Вот, например, объявление класса, представляющего политика:

```
class Политик
{
public:
    void ДатьОбещания();
    void БратьДеньги();
};
```

В этом классе представлены основные характеристики, присущие каждому политику. Класс, представляющий мифического честного политика, может быть создан приблизительно так:

```
class ЧестныйПолитик : public Политик
{
public:
    void ГоворитьПравду();
};
```

Поскольку этот класс является производным от класса Политик, процедуры ДаватьОбещания и БратьДеньги являются его составной частью. Но в дополнение к этому он содержит в себе также метод ГоворитьПравду. Ничего сложного, не так ли?

Зачем делать то, что уже сделано?

Нужно приложить некоторые усилия и проявить настойчивость, чтобы научиться извлекать все выгоды из возможности повторного использования кодов. Как уже упоминалось в предыдущей главе, чем больше вы программируете, тем профессиональнее вы это делаете. Ниже приведено несколько общих рекомендаций, которые следует помнить при разработке объектно-ориентированных программ.

- Думайте **с том, какие новые классы можно создать на базе уже имеющихся.** Например, если у вас есть базовый класс, представляющий основную информацию о сотрудниках, можно создать на его основе новый класс, добавив к нему данные о выплатах заработной платы, и получить таким образом объект, представляющий платежную ведомость.
- Посмотрите, **где еще может быть использован имеющийся набор данных.** Например, постарайтесь выделить в один класс некоторые общие свойства или характеристики. Класс, полученный путем наследования данного класса, всегда можно будет модифицировать, приспособив его к какой-то конкретной ситуации.
- Учитывайте **свои будущие потребности.** Возможно, не лишним будет сохранить некоторые старые классы, чтобы в дальнейшем наследовать их для создания новых классов.
- Смотрите, как **аналогичные задачи решаются другими программистами.**
- Помните, **что нужен большой практический опыт, чтобы научиться создавать качественные объектно-ориентированные программы.** Если что-то не получается, многое приходится переделывать заново; при этом, разумеется, возникает масса ошибок. Однако не огорчайтесь— все это является частью образовательного процесса.

Вы не ограничиваетесь возможностью наследовать только классы, написанные собственноручно. С тем же успехом можно наследовать любые встроенные .NET-классы. Например, вы можете набрать код. подобный этому:

```
class MyPen : public Pen
{
};
```

Наследование открытых, закрытых и защищенных элементов

Рано или поздно вам придется обратить внимание на те различия, которые возникают при наследовании членов классов с разными уровнями доступа. Вот правила, регулирующие этот процесс.

- ✓ Открытые элементы (`public`) базового класса доступны для использования как в пределах производного класса, так и за его пределами.
- ✓ Закрытые элементы (`private`) базового класса невидимы для всей остальной программы (в том числе и для производного класса),
- ✓ Защищенные элементы (`protected`) базового класса доступны для производного класса, но невидимы для всей остальной программы.

(В действительности в языке C++ есть еще несколько дополнительных правил, которые касаются наследования открытых, закрытых и защищенных членов класса. О них речь идет в разделе "Защищенное и закрытое наследование" ниже в этой главе.)

Перегрузка функций-членов

При необходимости можно изменить поведение элементов, которые были наследованы. Чтобы сделать это, заново объявите для производного класса функции-члены, работа которых должна отличаться от работы тех же функций-членов базового класса. При этом сами имена функций-членов не меняются. Эта возможность называется *перегрузкой (overriding)* и является одним из самых мощных и часто используемых средств языка C++.

Предположим, например, что у вас есть класс, представляющий фигуры. Он может выглядеть приблизительно так:

```
__gc class ShapeObject
{
public:
   void Draw(Graphics *poG);
   void PrintStats();
private:
   int rUseCount;
};
```

Если теперь вы хотите создать отдельный класс, представляющий линии, можете наследовать его из класса `ShapeObject`, изменив порядок действий, выполняемых функцией `Draw` (воспользовавшись возможностью перегрузки):

```
__gc class LineObject : public ShapeObject
{
public:
   void Draw(Graphics *poG);
};
```

Созданный таким образом класс LineObject будет иметь те же свойства и характеристики, что и класс ShapeObject, за исключением функции Draw, работа которой будет отличаться от работы функции Draw класса ShapeObject.

Родительские связи

Если вы воспользовались возможностью перегрузки и изменили работу наследованных функций-членов, то иногда все же может возникать необходимость доступа к соответствующим функциям-членам базового класса. (Как дети иногда нуждаются в помощи родителей, так и производные классы обращаются к базовым с тем, чтобы получить дополнительные возможности. Может быть, именно поэтому базовые классы иногда называют родительскими.) Сделать это несложно. Просто наберите имя базового класса, затем два двоеточия (: :) и название нужной функции.

Предположим, например, что функции Draw класса LineObject необходимо вызвать функцию Draw класса ShapeObject. Вот как это делается:

```
void LineObject::Draw(Graphics *poG)
{
   //Вызов функции Draw базового класса
   ShapeObject::Draw(poG);
   //Создание пера для рисования
   Pen *poPen = new Pen(Color::Red);
   //Рисование линии
   poG->DrawLine(poPen, m_nXFrom, m_nYFrom, m_nXTo, m_nYTo);
}
```

Это полезная возможность, поскольку если вам необходимо получить функциональность, которая имеется у базового класса, вам не нужно копировать его коды и вставлять их в производный класс. Вместо этого можно просто вызвать нужную функцию родительского класса и затем написать дополнительные коды для выполнения оставшейся работы.

А теперь немного практики

Здесь приведен код небольшой программы, на примере которого вы можете увидеть, как происходит наследование и перегрузка кодов. В программе есть два класса: базовый, выполняющий одну работу, и производный, работающий несколько иначе. Кроме того, вы можете увидеть, как функция производного класса вызывает функцию базового класса.

```
//Наследование

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;

//Базовый класс. В нем содержится информация о цене и функция,
//отображающая значение цены на экране
class Base
{
public:
   int nPrice;
   void PrintMe();
};
```

```
//Функция базового класса, отображающая цену на экране
void Base::PrintMe()
{
   Console::WriteLine(S"Base {0}", nPrice.ToString());
}

//Производный класс, в котором функция PrintMe будет
//определена по-своему
class Derived : public Base
{
public:
   void PrintMe();
};

//Функция производного класса, которая вызывает функцию
//базового класса
void Derived::PrintMe()
{
   Console::WriteLine(S"Derived");
   //Вызов функции базового класса
   Base::PrintMe();
}

//Ожидание, пока пользователь не остановит выполнение
//программы
void HangOut()
{
   Console::WriteLine(L"Нажмите Enter, чтобы завершить
                        выполнение программы");
   Console::ReadLine();
}

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
   Base oBaseClass;
   Derived oDerivedClass;

   oBaseClass.nPrice = 1;
   oDerivedClass.nPrice = 7;

   Console::WriteLine(S"Вызов функции базового класса");
   oBaseClass.PrintMe();

   Console::WriteLine(S"Вызов функции производного класса");
   oDerivedClass.PrintMe();
   HangOut();
   return 0;
}
```

Как процесс наследования отображается на конструкторах и деструкторах

По мере написания более сложных программ вы начнете создавать классы, которые имеют в своем арсенале несколько конструкторов. В этом разделе рассматривается, как вызывать такие конструкторы для классов, полученных путем наследования. Уделите данному вопросу должное внимание, и в будущем вы сможете оценить эффективность этого приема.

Когда вы создаете объект производного класса, вызывается конструктор базового класса. При этом компилятор в первую очередь ищет конструктор, установленный по умолчанию (конструктор, который не имеет параметров).

Иногда возникает необходимость в создании специализированных конструкторов. Например, вы уже могли видеть, что объект класса `Pen` может создаваться с указанием цвета либо с указанием цвета и ширины пера. Если на основе класса `Pen` вы создадите производный класс, то наверняка при создании объектов нового класса вам захочется использовать параметризованные конструкторы базового. Сделать это можно с помощью конструктора производного класса. Для этого просто укажите конструктор базового класса, который должен вызываться при вызове конструктора производного класса, и список используемых параметров:

```
Derived::Derived() : Base(...)
{
}
```

Например:

```
NewPen::NewPen(int nWidth, int Style) : Pen(Color::Red, nWidth)
{
}
```

В данном случае вызов конструктора `NewPen(int, int)` сопровождается вызовом конструктора базового класса Pen, который устанавливает цвет и ширину пера.

При создании или удалении объекта производного класса вызываются конструкторы и деструкторы как производного, так и базового класса. Очень важно понимать порядок, в котором они вызываются. При создании объекта для него вначале выделяется память, затем вызывается конструктор базового класса, после чего конструктор производного класса. При удалении объекта вначале вызывается деструктор его класса, а затем деструктор базового класса.

Универсальный указатель

Если у нас есть указатель, ссылающийся на объект базового класса, *его* же можно использовать и для указания на объекты производных классов. Предположим, например, что `*pShape` является указателем на объекты класса `ShapeObject`:

```
ShapeObject *poShape;
```

Если какой-то объект является производным от класса Shape-Object, то для ссылки на него также можно использовать указатель `poShape`:

```
pShape = new LineObject();
```

"Ну и что?" --- спросите вы. А мы ответим: "Это очень важно, поскольку отсюда следует, что вы можете использовать один и тот же указатель для ссылки на множество различных объектов. Так, вам не обязательно заранее знать, какой объект создаст пользователь данного

класса: специальный, производный или еще какой-то. Вы должны использовать один и тот же указатель, и он будет работать с производными объектами так же хорошо, как и с базовыми".

Предположим, например, что вы пишете программу, которая может рисовать различные фигуры, среди которых могут быть круги и линии. Вы можете создать базовый класс `ShapeObject` и производные классы: `LineObject` для рисования линий и `CircleObject` для рисования кругов. Далее для хранения информации об отображаемых фигурах вы можете создать список `ArrayList` и использовать для ссылки на любой из его элементов указатель на объект класса `ShapeObject`.

Увидеть этот прием в действии вы сможете ниже в этой главе на примере программы Draw8. В список `ArrayList` будут добавляться объекты `LineObject` и `CircleObject`, а затем для доступа к ним вы просто используете такой код (ну хорошо, слово *просто* в данном случае, пожалуй, является не самым подходящим):

```
static_cast<ShapeObject*>(poEnumerator->Current)->Draw(poG);
```

Поскольку классы `LineObject` и `CircleObject` являются производными от класса `ShapeObject` и поскольку класс `ShapeObject` имеет функцию-член Draw, этот код будет работать независимо от того, является ли текущий элемент списка объектом класса `LineObject` или объектом класса `CircleObject`.

Почему указатель может ссылаться на объекты разных классов?

Как вы знаете, C++ следит за корректностью использования типов данных. Например, если у вас есть указатель типа **int** *, то он не может ссылаться на значение типа `float`. Однако в случае с производными классами все обстоит несколько иначе. Когда компилятор создает производный класс, вначале в нем размещаются элементы, которые наследуются из базового класса. Например, если первые четыре элемента базового класса имели тип integer, то в производном классе первые четыре элемента также будут иметь тип integer.

Если компилятор имеет дело с указателем на базовый класс, то он знает, как в нем найти различные члены данных и функции-члены по их местоположению в этом классе. Если вы используете тот же указатель для ссылки на производный класс, то все члены данных и функции-члены, наследованные от базового класса, будет так же легко найти, поскольку производный класс выглядит как копия базового класса с некоторыми незначительными изменениями. Вот почему у вас есть такая возможность. Компилятор сможет найти любую функцию по ее местоположению в базовом классе, несмотря на различия, внесенные в производный класс в результате наследования.

А что произойдет, если в производном классе будет перефужена какая-либо функция базового класса? Об этом речь пойдет ниже.

Защищенное и закрытое наследование

Наследуя классы, можно определить права доступа. До этого момента, создавая производный класс, мы использовали только ключевое слово `public`. Кроме него, можно использовать также ключевые слова `private` и `protected`.

В табл. 19.1 показано, какие права доступа приобретают элементы производного класса при использовании в процессе наследования ключевых слов `public`, `private` и `protected`.

Таблица 19.1. Влияние слов public, private и protected на процесс наследования

Используемое слово	Члены базового класса объявлены как...	Члены производного класса наследуются как ...
public	public	public
	protected	protected
	private	**Недоступны для наследования**
protected	public	protected
	protected	protected
	private	Недоступны для наследования
private	public	private
	protected	private
	private	Недоступны для наследования

Виртуальная реальность

Виртуальные функции используются тогда, когда указатель на объект иногда ссылается на базовый класс, а иногда — на производный. К счастью, все это не так сложно, как выглядит поначалу. Чуть выше, в разделе "Универсальный указатель", вы уже видели фрагмент кода из программы DrawS, где список ArrayList использовался для хранения объектов CircleObjectи LineObject. Это, собственно, и все, о чем пойдет речь.

Конечно, здесь есть одна маленькая загвоздка. Предположим, что в производном классе вы перегрузили одну из функций-членов, изменив тем самым ее поведение. Например, в классе LineObject функция Draw была перегружена таким образом, чтобы в процессе ее выполнения на экране рисовались линии. А в классе CircleObjectта же функция была изменена так, чтобы ее вызов сопровождался рисованием кругов. Теперь, если вы используете указатель типа ShapeObject для ссылки на один из этих объектов и вызовете функцию Draw, вы не получите тот результат, который вам нужен: будет вызвана функция Draw базового класса.

Вот что получается. Метод наследования предоставляет вам великолепную возможность повторного использования кодов, а свойства указателей обеспечивают гибкость при работе с объектами. Но, когда эти два неоспоримых преимущества накладываются друг на друга, возникает серьезная проблема.

И тут на помощь приходят *виртуальные функции*. Когда их видят указатели, они знают, что им придется вызывать перегруженные функции-члены производного класса. Если программисты говорят между собой о *полиморфизме* как о важнейшем преимуществе объектно-ориентированного программирования, на самом деле речь идет о виртуальных функциях.

Когда вы делаете функцию Draw виртуальной, упомянутая выше проблема исчезает. Если у вас есть объект LineObject, но на него ссылается указатель типа ShapeObject, вызов функции Draw будет означать вызов именно функции LineObject:: Draw, а не ShapeObject: : Draw. Более того, при этом не возникает никакой путаницы. Так, если вы ссылаетесь на объект LineObject с помощью указателя типа LineObject, то при вызове функции Draw также будет вызываться функция LineObject: : Draw. Другими словами, с виртуальными функциями происходят только правильные вещи.

Тест на виртуальность

Ответив на перечисленные ниже вопросы, вы сумеете определить, можно ли какую-то конкретную функцию сделать виртуальной. Если ответ хотя бы на один из вопросов будет отрицательным, делать функцию виртуальной нет необходимости.

- ✓ **Является ли данный класс базовым для каких-то производных классов?** (Или станет ли он таким в будущем?)
- ✓ **Отличаются ли действия функции в базовом классе от действий аналогичной функции в производном классе?** (Будут ли они отличаться в будущем?)
- ✓ **Используете ли вы указатель на базовый класс?**
- ✓ **Нужно ли вам использовать тот же указатель для ссылки на производный класс?** Другими словами, будут ли какие-то указатели ссылаться попеременно то на базовый, то на производный классы.

Если ответы на *все* четыре вопроса будут положительными, вам нужна виртуальная функция.

Декларация ваших виртуальных намерений

Виртуальная функция объявляется в базовом, но не в производном классе. Для этого наберите перед названием функции ключевое слово `virtual`:

```
class Base
{
public:
  int Price;
  virtual void PrintMe();
};
```

Этот код говорит о том, что в классе Base есть виртуальная функция PrintMe.

Предположим, **что** на основе этого класса вы создаете производный:

```
class Derived : public Base
{
public:
  virtual void PrintMe();
};
```

Теперь компилятор сможет корректно вызывать функцию PrintMe, если вы используете один и тот же указатель для ссылки на объекты этих двух классов.

Вам не обязательно набирать слово `virtual` перед названием функции при объявлении производного класса. Однако лучше это сделать, поскольку в дальнейшем при чтении кодов программы вам легче будет определить, что в данном случае вы имеете дело с виртуальной функцией.

Например, класс Derived можно объявить так:

```
class Derived : public Base
{
public:
 void PrintMe();
>;
```

Однако следующий код более понятен:

```
class Derived : public Base
{
public:
  virtual void PrintMe();
};
```

Когда без этого не обойтись

Если вы еще не вполне ясно представляете, для чего нужны виртуальные функции, просмотрите приведенный в этом разделе пример программы, где используются виртуальная и невиртуальная функции. Запустив эту программу на выполнение и увидев возвращаемый ею результат, вы поймете, почему при работе с указателями иногда нужно создавать виртуальные элементы.

В программе используются два класса: Bass и Derived. Класс 3ase содержит функцию-член PrintMe, которая не является виртуальной, и виртуальную функцию-член PrintMeV:

```
class Base
{
public:
   void PrintMe();
   virtual void PrintMeV();
};
```

Класс Derived создается путем наследования класса Base. При этом обе функции перегружаются:

```
class Derived : public Base
{
public:
   void PrintMe();
   virtual void PrintMeV();
};
```

Работа функции main начинается с создания объекта класса Base. Чтобы продемонстрировать свойства виртуальной функции, для доступа к объекту используется указатель:

```
Base *po3ase = new Base();
```

Затем с его помощью вызываются функции-члены:

```
poBase->PrintMe();
poBase->PrintMeV();
```

Далее создается объект класса Derived. Для ссылки на этот объект используется ранее объявленный указатель типа Base. И теперь самое интересное: функции-члены производного класса вызываются с помощью указателя, созданного для ссылки на объекты базового класса:

```
poBase = new Derived();
poBase->PrintMe();
po3ase->PrintMeV{);
```

Когда этот код будет выполняться, вы увидите, что будет вызвана функция-член PrintMe класса 3ase несмотря на то, что указатель poBase в данный момент ссылается на объект класса Derived. Это происходит потому, что функция PrintMe не является виртуальной. В то же время вызванная функция PrintMeV будет относиться к классу Derived. Почему? Потому что она является виртуальной. (На рис. 19.1 показан результат выполнения приведенных выше кодов.)

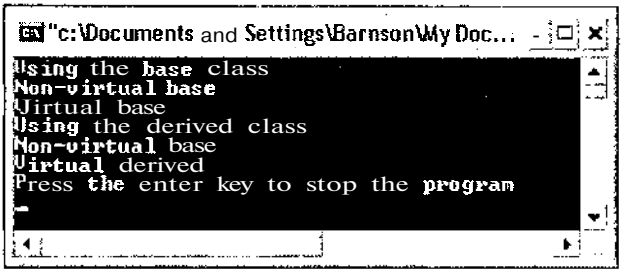


Рис. 19.1. Использование виртуальных функций позволяет указателям находить функции-члены производных классов

После того как вы объявили какую-то функцию виртуальной, она становится таковой для всех производных классов. Например, функция PrintMeV была объявлена как виртуальная для класса Base. Следовательно, она также будет виртуальной и для производного класса Derived. Если вы создадите новый класс (назвав его, допустим, DerivedKiddo), производный от класса Derived, его функция PrintMeV также будет виртуальной. Независимо от того, как много классов будут наследовать коды других классов (а на количество классов, создаваемых путем наследования, ограничений нет), компилятор всегда сможет правильно определить, функцию какого класса вы хотите вызвать.

Вот полная распечатка кодов программы, о которой только что шла речь:

```
//Inherit2

//Использование виртуальной функции

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;
//Объявление базового класса
class Base
{
public:
   void PrintMe();
   virtual void PrintMeV();
};

//Определение функций базового класса. Отображаемый ими текст
//позволяет определить, какая именно функция была вызвана
void Base::PrintMe()
{
   Console::WriteLine(S"Non-virtual base");
}

Void Base::PrintMeV()
{
   Console::WriteLine(S"Virtual base");
}

//Производный класс, создаваемый путем наследования класса Base
class Derived : public Base
{
```

```
public:
   void PrintMe();
   virtual void PrintMeV();
};

//Перегрузка функций производного класса.
//Их выполнение позволяет сделать вывод о том,
//какая именно функция была вызвана
void Derived::PrintMe()
{
   Consile::WriteLine(S"Non-virtual derived");
}

Void Derived::PrintMeV()
{
   Consile::WriteLine(S"Virtualderived");
}

//Ожидание, пока пользователь не остановит выполнение программы
void HangOut()
{
   Console::WriteLine(L"Нажмите Enter, чтобы завершить
                        выполнение программы");
   Console::ReadLine();
}

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
   //Динамическое выделение памяти для объекта базового класса
   Base *poBase = new Base [};

   //Вызов двух функций базового класса
   Console::WriteLine(S"Использование базового класса");
   poBase->PrintMe();
   poBase->PrintMeV{);

   delete poBase;

   //Теперь этот же указатель используется для ссылки
   //на объект производного класса
   poBase = new Derived();

   //Вызов функций класса, на объект которого в данный
   //момент ссылается указатель
   Console::WriteLine(S"Использование производного класса");
   poBase->PrintMe();
   poBase->PrintMeV();

   HangOut();
   return 0;
}
```

Абстрагирование от деталей

Предположим, вам нужно создать программу, рисующую геометрические фигуры. Заранее известно, что будет много разных фигур, таких как круги, линии и квадраты. Для хранения информации обо всех отображаемых фигурах можно использовать список ArrayList. Следовательно, вам нужен базовый класс, из которого путем наследования можно создать производный классы для представления всех этих фигур. Почему? Помните, что указатель на базовый класс может ссылаться также и на объекты производных классов. Имея базовый класс, вы можете сохранять все производные объекты в списке ArrayList, извлекать их оттуда с помощью указателя на базовый класс, вызывать функцию Draw для рисования фигур на экране. Сделав функцию-член Draw виртуальной, вы обеспечиваете корректность выполнения программы: круги будут рисоваться как круги, линии как линии, а квадраты как квадраты. Итак, вам нужно создать базовый класс ShapeObject, который станет основой для создания производных объектов LineObject и CircleObject, а также других объектов, предназначенных для отображения фигур на экране.

Обратите внимание, что в данном примере объекты самого класса ShapeObject никогда не будут сохраняться в списке ArrayList. Действительно, ведь нет некой общей фигуры, которую нужно будет отображать на экране. Отображать понадобится только заранее определенные фигуры, такие как круги или квадраты. Класс ShapeObject предназначен всего лишь для представления общей концепции геометрической фигуры.

Чтобы обобщить эту концепцию и не допустить непреднамеренного прямого использования класса ShapeObject, вы можете сделать его абстрактным базовым классом. *Абстрактный базовый класс* представляет некую концепцию (отсюда и его название — *абстрактный),* но сам не несет никакой функциональности. Вы никогда не сможете создать объект такого класса: если попытаетесь это сделать, компилятор сразу же выскажет все, что думает по этому поводу. Этот класс может быть использован только для создания на его основе производных классов. А сами производные классы должны определить функциональность для каждой функции-члена, наследованной от абстрактного базового класса.

Чтобы сообщить компилятору, что создаваемый класс должен быть абстрактным, нужно все функции-члены этого класса сделать *исключительно виртуальными.* Этот термин означает, что интерфейс класса объявляется, но не определяется. Чисто технически это выполняется следующим образом: при объявлении функций-членов в конце строки наберите = 0;

```
//Объявление абстрактного базового класса
__gc class ShapeObject
{
   virtual void Initialize!) = 0;
   virtual void Draw(Graphics *poG) = 0;
};
```

Историческая справка: как раз тогда, когда я писал эти строки, у меня родилась дочь Габриэла.

Класс, функции-члены которого объявлены таким образом, не может иметь собственных объектов, т.е. вы не сможете создать объект класса ShapeObject, набрав код наподобие такого:

```
ShapeObject foo;
```

И это здорово! Ведь вам такие объекты и не нужны. Этот класс вам нужен только как базовый для создания серии производных классов. И уже при создании производных классов вы должны позаботиться о том, чтобы вдохнуть жизнь (т.е. определить функциональность) во

все исключительно виртуальные функции. Если же вы этого не сделаете, компилятор напомнит вам о ваших обязанностях.

Готов ли класс к тому, чтобы обзавестись потомством?

Классы к наследованию нужно подготовить. Вот небольшой список, с которым следует сверяться, чтобы определить, все ли сделано для того, чтобы какой-либо класс смог стать базовым (родительским) и произвести на свет парочку производных (дочерних) классов. (Готовы ли ползунки и погремушки? И как насчет детской коляски и кроватки?}

- ✓ Если вы планируете использовать указатель для ссылки на объекты базового и производного класса, объявите виртуальными все функции-члены, коды которых будут перегружаться.
- ✓ Если базовый класс имеет несколько специализированных конструкторов, убедитесь, что конструкторы производного класса непосредственно вызывают один из этих конструкторов, либо что базовый класс имеет также конструктор, используемый по умолчанию.
- ✓ Если в базовом классе некоторые члены данных или функции-члены объявлены как закрытые (`private`), то их лучше перевести в категорию защищенных (`protected`).
- ✓ Если вы наследуете кучу денег, обязательно вышлите часть из них автору этой книги.

Искусство абстрагирования

Теперь, когда вы уже имеете общее представление о виртуальных функциях и абстрактных базовых классах, пришло время применить эти замечательные возможности на практике и создать настоящую программу. Мы добавим функциональность программе рисования, чтобы ее выполнение действительно сопровождалось отображением на экране кругов и линий.

Как она будет работать? Как и ранее, для сохранения информации об отображаемых фигурах будет использоваться список `ArrayList`. Но теперь в этот список будут помещаться объекты классов `LineObject` и `CircleObject`:

```
Console::WriteLine("Type c to create a circle and 1 to create
                    a line");
//Отображаемое сообщение переводится так: "Наберите с, чтобы
//создать круг, и 1, чтобы создать линию"
pszMore = Console::ReadLine();

if (pszMore->Equals(S"c"))
{
   //Создание объекта, представляющего круг
   poShape = new CircleObject();
}
else
i
   //Создание объекта, представляющего линию
   poShape = new LineObject();
}
```

Оба класса являются производными от абстрактного базового класса `ShapeObject`. И все эти классы имеют виртуальные функции-члены `Initialize` и `Draw`, которые поразному определяются для каждого производного класса. Так, для класса `LineObject` функции-члены с такими именами считывают значения координат двух точек и рисуют между ними прямую линию. Для класса `CircleOoject` такие же функции считывают координаты центра круга и длину радиуса и, исходя из этих данных, рисуют круг на экране.

Как и ранее, при извлечении объектов из списка нужно указать, к какому классу они относятся. В данном случае вы указываете, что все объекты относятся к классу ShapeObject, и затем вызываете функции-члены, полагаясь на то, что компилятор сможет разобраться, к какому классу (LineObjectили CircleObject) на самом деле относится текущий объект и какая функция должна быть вызвана:

```
IEnumerator *poEnumerator = m_paShapes->GetEnumerator();
while (poEnumerator->MoveNext())
{
   //Вызов функции, отображающей фигуру на экране
   static_cast<ShapeObject*>(poEnumerator->Current)->Draw(poG);
}
```

При этом достигается действительно великолепная гибкость и функциональность. Попробуйте набрать коды приведенной ниже программы и запустить ее на выполнение.

Draw8

```
//Использование виртуальных функций и возможности наследования

#include "stdafx.h"

#using <mscorlib.dll>
#using<System.Windows.Forms.dll>
#using <System.dll>
#using <System.Drawing.dll>
using manespace System;
using manespace System::Drawing;
using manespace System::Windows::Forms;
using manespace System:Collections;


//Объявление абстрактного базового класса
__gc class ShapeObject
{
public:
   virtual void Initialize() = 0;
   virtual void Draw(Graphics *poG) = 0;
};

//Сохранение информации о линиях
__gc class LineObject : public ShapeObject
{
public:
   virtual void Initialize();
   virtual void Draw(Graphics "poG);
private:
   int m_nXFrom;
   int m_nYFrom;
   int m_nXTo;
   int mnYTo;
};

//Инициализация объекта, представляющего линию
void LineObject::Initialize()
{
```

```
    //Чтение координат x и y для начальной и конечной
    //точек линии
    Console::WriteLine(S"Координата x начальной точки:");
    m_nXFrom = Int32::Parse(Console::ReadLine());
    Console::WriteLine(S"Координата y начальной точки:");
    m_nYFrom = Int32::Parse(Console::ReadLine());
    Console::WriteLine(S"Координата x конечной точки:");
    m_nXTo = Int32::Parse(Console::ReadLine());
    Console::WriteLine(S"Координата y конечной точки:");
    m_nYTo = Int32::Parse(Console::ReadLine());
}

//Рисование линии
void LineObject::Draw(Graphics *poG)
{
    //Создание пера для рисования
    Pen *poPen = new Pen(Color::Red);
    //Рисование линии
    poG->DrawLine(poPen, m_nXFrom, m_nYFrom, m_nXTo, m_nYTo);
}

//Сохранение информации об окружностях
__gc class CircleObject : public ShapeObject
{
public:
    virtual void Initialize();
    virtual void Draw(Graphics *poG);
private:
    int m_nXCenter;
    int m_nYCenter;
    int m_nRadius;
};

//Инициализация объекта, представляющего круг
void CircleObject::Initialize()
{
    //Чтение координат центра круга и длины его радиуса
    Console::WriteLine(S"Координата x центра круга:");
    m_nXCenter = Int32::Parse(Console::ReadLine());
    Console::WriteLine(S"Координата y центра круга:");
    m_nYCenter = Int32::Parse(Console::ReadLine());
    Console::WriteLine(S"Радиус круга:");
    m_nRadius = Int32::Parse(Console::ReadLine());
}

//Рисование круга
void CircleObject::Draw(Graphics *poG)
{
    //Создание пера для рисования
    Pen *poPen = new Pen(Color::Blue);
    //Рисование круга
    poG->DrawEllipse(poPen, m_nXCenter-m_nRadius, m_nYCenter-
                m_nRadius,    ra_nXCenter+m_nRadius,
                m_nYCenter+m_nRadius);
}
```

```
//Сохранение информации обо всех фигурах,
//которые должны быть нарисованы
__gc class DisplayObject
{
public:
   void Add();
   void Draw(Graphics *poG);
   DisplayObject();
private:
   ArrayList *m_paShapes;
};

//Инициализация структуры DisplayObject
void DisplayObject::Initialize()
{
   m_paShapes = new ArrayList();
}

//Добавление нового объекта
void DisplayObject::Add()
{
   bool fFinished = false;
   String *pszMore;
   ShapeObject *poShape;

   while (!fFinished)
   {
      //Какой объект должен быть создан?
      Console::WriteLine("Type c to create a circle
                          and 1 to create a line");
      //Отображаемое сообщение переводится так: "Наберите
      //c, чтобы создать круг, и 1, чтобы создать линию"
      pszMore = Console::ReadLine();

      if (pszMore->Equals(S"c"))
      {
         //Создание объекта, представляющего круг
         poShape = new CircleObject();
      }
      else
      {
         //Создание объекта, представляющего линию
         poShape = new LineObject();
      }

      //Инициализация созданного объекта
      pcShape->Initialize  ();
      //Сохранение его в структуре
      m_paShapes->Add(poShape);

      //Вопрос к пользователю: хочет ли он добавить еще
      //линию или круг?
      Ccnsole::WriteLine("Нажмите y, чтобы добавить линию
                                  или  круг");
```

```
        pszMore = Console::ReadLine();
        if (!pszMore->Equals(S"y"))
        {
            fFinished = true;
        }
    }
}

//Отображение всех фигур на экране
void DisplayObject::Draw(Graphics *poG)
{
    //Просмотр всех объектов, представляющих круги и линии
    IEnumerator *poEnumerator = m_paShapes->GetEnumerator();
    while (poEnumerator->MoveNext())
    {
    //Вызов функции Draw для объекта
    static_cast<ShapeObject*>(poEnumerator->Current)->Draw(poG);
    }
}

//С этой точки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
    //Структуры для рисование на экране графических элементов
    Form *poForm = new Form();
    Graphics *poGraphics = poForm->CreateGraphics();
    DisplayObject *poDisplay = new DisplayObject();

    //Добавление новой фигуры
    poDisplay->Add();

    //Переход к левому верхнему углу экрана
    poForm->SetDesktopLocation(0,0);
    //Открытие окна, в котором будут нарисованы линии
    poForm->Show();

    //Рисование кругов и линий
    poDisplay->Draw(poGraphics);

    //Освобождение памяти, выделенной для отображения графики
    poGraphics->Dispose();

    //Ожидание, пока пользователь не закроет форму
    Application::Run(poForm);
}
```

Глава 20

Исключительные ситуации

В этой главе...

- Обработка исключений
- Собственное определение ошибок
- Синтаксические правила
- Идентификация ошибок и преобразование типов

Одним из этапов создания программы является ее тестирование и отладка. Но даже если вы устраните все ошибки и программа начнет работать, это еще не означает, что проблемы не возникнут в будущем. Причина в том, что вы не можете заранее предвидеть всех действий пользователей, производимых в отношении вашей программы, и всех ситуаций, с которыми может столкнуться созданный вами программный продукт. Единственный способ проверить, как будет вести себя программа в различных экстремальных условиях, — создать эти условия и посмотреть, что произойдет. Кстати, небезызвестные законы Мэрфи во многом обязаны своим происхождением всем тем проблемам, которые возникают между пользователями и программным обеспечением.

Такие непредвиденные условия называют *условиями ошибки*. Существуют тысячи причин, по которым эти условия возникают. Например, на диске нет свободного места, файл с данными содержит неверную информацию, пользователь вместо радиуса круга указывает координаты его центра и т.п.

Как следствие, *все* перечисленные ситуации (и множество им подобных) могут привести к сбоям в работе программы. К счастью, существует такое средство, как *обработка исключений*, которое поможет вам с достоинством справиться со всеми непредвиденными проблемами.

Как это было раньше

Прежде чем осваивать метод обработки исключений, вам полезно будет узнать, как подобные проблемы решались ранее.

Процесс устранения ошибок можно разбить на два этапа: определение факта наличия ошибки, и се нахождение и ликвидация. Первый этап — определение того, что ошибка действительно есть, • — обычно не представляет никаких сложностей. Просто в программе в нужных местах необходимо добавить несложные коды, определяющие наличие ошибки и сообщающие что-то наподобие; "Ага, недостаточно данных. Есть ошибка!"

Когда установлено, что в программе присутствует ошибка, нужно найти источник ее возникновения и ликвидировать его. Локализация ошибки может оказаться более сложной задачей. Коды программы, обнаружившие существование ошибки, могли быть вызваны функцией, которая была вызвана другой функцией, которая, в свою очередь, была вызвана еще одной функцией, и т.д. Желательно, чтобы все эти функции могли сами определить наличие ошибки и решить, как поступать далее. Обычно к ним добавляются коды, проверяющие наличие

ошибки и в случае ее отсутствия позволяющие функциям работать далее в обычном режиме, а в случае обнаружения ошибки — досрочно прекращающие их работу.

Другими словами, набор кодов

```
//Вызов нескольких функций
ReadSongs();
ReadSizes();
ReadMyLips();
```

желательно заменить такими кодами;

```
//Вызов нескольких функций. Если возвращаемый функцией результат
//меньше нуля, это расценивается как ошибка
nTemp = ReadSongs();
if (nTemp < 0)
{
  reyurn nTemp;
}
nTemp = ReadSizes();
if (nTemp < 0)
{
  reyurn nTemp;
}
nTemp = ReadMyLips();
if (nTemp < 0)
{
  reyurn nTemp;
}
```

Теперь представьте, что подобные коды нужно набирать в каждой процедуре программы после вызова каждой отдельной функции.

Более того, если для функции допустимо возвращать отрицательные значения, вам придется придумать другую схему, определяющую возникновение ошибки после каждого вызова этой функции.

И наконец, после того как вы добавите все эти коды, нужно будет создать еще функцию, которая сможет обработать информацию об ошибке и сообщить что-то вроде: "Ага, я знаю как справиться с этой ошибкой". Такая функция должна уметь различать, ошибка какого типа произошла, и знать, как на это реагировать.

В общем, осуществить все это на практике — задача не из легких. На написание всех этих сложных кодов, обрабатывающих возможные ошибки, может уйти больше времени и усилий, чем на написание реально работающей части программы. К тому же эти дополнительные коды делают программу громоздкой и трудночитаемой.

Новый усовершенствованный способ обработки ошибок

Возможность обработки исключений позволяет избавиться от необходимости добавления проверяющих кодов после каждого вызова какой-либо функции.

Сам механизм обработки исключений состоит из двух частей: одна процедура определяет исключительную ситуацию (ее задача сказать: "О да, здесь есть ошибка!") и вторая процедура (которая, собственно, и обрабатывает найденное исключение) берет на себя управление, если какое-то исключение найдено.

Второе преимущество механизма обработки исключений состоит в том, что любые локальные объекты, т.е. те, которые создаются в процессе выполнения какой-то функции, автоматически уничтожаются. Продемонстрируем это на таком примере. Предположим, что функция A создает локальный объект ObjectA, затем вызывает функцию B, которая создает локальный объект OhjectB и вызывает функцию, которая находит ошибку. Объекты ObjectA и ObjeclB должны быть удалены. Таким образом, все используемые ими файлы, данные, фрагменты памяти и т.п. автоматически освобождаются.

Фрагмент кода, который может послужить причиной возникновения исключительной ситуации, необходимо поместить внутри фигурных скобок, а перед скобками набрать ключевое слово try. Это будет командой компилятору: "Попробуй выполнить этот фрагмент кодов и посмотри, что будет". Далее нужно набрать ключевое слово catch и в фигурных скобках поместить коды, определяющие реакцию на возникновение ошибки. Компилятор это воспримет так: "Если возникнет какая-то проблема, действуй в соответствии с приведенными здесь инструкциями".

Вот как это выглядит на практике

Лучше один раз увидеть, чем сто раз услышать. Поэтому перейдем к демонстрации сказанного на конкретном примере. Приведенная ниже функция предназначена для получения числа от пользователя и преобразования его к значению типа integer. (Функция взята из программы Factorial4, вычисляющей факториал заданного числа.) Казалось бы, ничего сложного здесь нет:

```
int GetNumber()
{
   int Г-Number;

   Console::WriteLine(S"Укажите число");
   nNumber = Int32::Parse(Console::ReadLine());
   return nNumber;
}
```

Но что произойдет, если пользователь наберет что-то неверное или бессмысленое, например слово "печенька"? Поскольку это не может быть преобразовано к значению типа integer (это вообще не число), на экране появится сообщение об ошибке и работа программы будет моментально завершена.

Что в действительности может произойти, если компилятор столкнется с исключительной ситуацией такого рода? Если вы не оставите на этот случай никаких инструкций, как уже отмечалось, появится сообщение об ошибке и программа будет остановлена. Если же вы заранее сообщите компилятору, что нужно делать в такой ситуации, подобного развития событий можно избежать. Вот аналогичная функция, которая самостоятельно может справиться с проблемой, возникшей в результате ввода пользователем некорркгного значения:

```
int GetNumber()
{
   int nNumber;

   Console::WriteLine(S"Укажите число");
   rty
   {
   nNumber = Int32::Parse(Console::ReadLine());
   }
   catch (FormatException *e)
```

```
    {
    Console::WriteLine(e->Message);
    Console::WriteLine("В следующий раз наберите, пожалуйста,
                число. Сейчас введенное вами значение будет
                воспринято как число 1");
    nNumber = 1;
    }
    return nNumber;
}
```

Если пользователь наберает какую-то абракадабру, это означает возникновение ошибки `FermatException`. Оператор `catch` идентифицирует эту ошибку, в результате чего на экране отображается дружественное сообщение и возвращается факториал числа, заданного по умолчанию (в данном случае это число 1).

Вспомните о пользователях!

Сами по себе пользователи хороши тем, что они существуют. Именно благодаря тому, что они используют создаваемое вами программное обеспечение, вы стяжаете себе славу и известность или по крайней мере сохраняете за собой свое рабочее место. Самое плохое в пользователях то, что они зачастую делают с вашей программой такое, о чем вы и подумать не могли. И именно поэтому программу нужно снабжать кодами, управляющими исключительными ситуациями.

Вот несколько наиболее часто возникающих проблемных ситуаций, с которыми ваша программа должна уметь справляться.

- ✓ **Не хватает памяти для выполнения программы.** Вызвано это может быть самыми разными причинами. Например, пользователь пытается открыть файл огромных размеров или у его компьютера просто небольшой объем оперативной памяти.
- ✓ **Пользователь неверно указывает имя файла.** Например, в ответ на приглашение набрать имя файла, который должен быть открыт, пользователь набирает что-то вроде: "Какая разница, открывай любой". Разумеется, что неверное имя файла.
- ✓ **Пользователь вводит число, которое не принадлежит допустимому диапазону.** Например, должно быть введено число в интервале от нуля до пяти, а пользователь вводит число 17.
- ✓ **функция, к которой пользователь доступа не имеет, по какой-то причине получает в качестве аргумента недопустимое значение.** Например, внутри программы используется процедура, требующая для себя значение из определенного диапазона, и в силу каких-то обстоятельств ей передается значение, этому диапазону не принадлежащее. Чтобы программа смогла адекватно отреагировать на такое событие, снабдите эту процедуру кодами обработки исключительных ситуаций.
- ✓ **Пользователь загружает файл недопустимого типа.** Например, созданная вами программа предназначена для обработки текстовых файлов, а пользователь пытается открыть файл с расширением EXE.
- ✓ **Файлы данных утеряны.** Предположим, например, что для работы программы необходимо загрузить файл PASSWD.SCT, содержащий в себе список паролей. Но по какой-то причине пользователь этот файл удалил или переместил в другую папку, и теперь программа не может его найти.
- ✓ **Нет свободного места** на **диске.** Программа должна сохранить файл на жестком диске, но имеющегося свободного пространства для этого не достаточно.

Гибкость — основное свойство механизма обработки исключений

Механизм обработки исключений языка C++ предоставляет поистине неограниченную свободу действий. Заключая отдельные фрагменты кодов в фигурные скобки оператора try. вы можете сами определять, какие из них нужно проврять на наличие ошибок, а какие нет. Для одного и того же фрагмента кодов можно оставить инструкции на случай возникновения самых разных ошибок. И наконец, для одной и той же ошибки, возникающей в разных частях программы, можно оставить различные инструкции, поместив их в фигурные скобки соответствующих операторов catch.

Предположим, например, что в разных частях программы вызывается функция AllocateBuf. В одной части программы эта функция вызывается для того, чтобы выделить память для файла с данными, который должен быть сохранен, а в другой части — чтобы выделить память для фотоизображения.

Это может быть реализовано в таком виде:

```
//Попытка сохранения файла
try
{
  AllocateBuf();
  SaveFile();
}
catch (Errorl e)
{
  Console::WriteLine(S"Файл не может быть сохранен");
}
//Попытка выделить память для фотографии
try
{
  AllocateBuf();
  ProcessPhoto();
}
catch (Errorl e]
{
  Console::WriteLine(S"Фотография не может быть обработана");
}
```

Каждый раз вызывается функция AllocateBuf, и каждый раз проверяется наличие одной и той же ошибки: невозможность выделения памяти. Но в каждом случае программа реагирует по-разному, и на экране отображаются соответствующие случаю сообщения.

Определение собственных исключительных ситуаций

Предположим, что программа вызываст некую функцию fco и эта функция принимает значения нескольких параметров. Допустим, что заранее известно: если значение, скажем, первого параметра будет меньше (или больше) некоторого числа, то ситуация выйдет из под контроля либо будет развиваться в совершенно неприемлемом направлении. Как в этом случае указать комилятору на наличие ошибки?

Чтобы самостоятельно идентифицировать ошибку, воспользуйтесь командой throw. Ее выполнение будет означать сигнал о возникновении исключительной ситуации, и компилятор перейдет к инструкциям catch, следующим за текущим блоком try.

Предположим, например, что вам нужно вычислить квадратный корень числа. Вы знаете, что эта операция некорректна для отрицательных чисел, а потому можете набрать код, подобный следующему:

```
double  SquareRoot(int nNumber)
{
   if (nNumber  <  0)
   {
      throw new SquareRootException(S"Число должно быть
                           положительным");
   i
   return Math::Sqrt(nNumber);
}
```

Эта функция проверяет, не является ли число отрицательным. Если да, фиксируется исключительная ситуация. Если нет, вычисляется квадратный корень этого числа.

Вы можете определить столько собственных исключительных ситуаций, сколько будет необходимо. Каждая команда throw передает отдельный тип данных. Можно создать любое количество типов данных, определить для них конкретные значения и использовать эту информацию как вспомогательную для управления процессом обработки исключения.

Для этого можно использовать даже классы. В таком случае вы можете создать процедуру, управляющую исключительной ситуацией, непосредственно как функцию-член этого класса и передать объект данного класса кодам catch с помощью команды throw. Например, можно создать класс MyError и при возникновении исключительной ситуации воспользоваться командой throw, которая создаст объект класса MyError и передаст его кодам catch.

Команды catch в качестве параметра принимают типы данных. Так, можно создать класс ChoiceError, который будет идентифицироваться одной командой catch, и класс MyError, на который будет реагировать другая команда catch.

При этом также можно использовать механизм наследования. Например, класс MyError может быть производным от класса ChoiceError.

Поговорим о синтаксисе

Вот как выглядит синтаксис кодов обработки исключений:

```
try
{
 //Коды, при выполнении которых может возникнуть ошибка
 инструкции;
}
catch (тип_ошибки_1)
{
 инструкции на случай возникновения этой ошибки;
}
catch (тип_ошибки_2)
{
 инструкции на случай возникновения этой ошибки;
}
```

.
.
.
Чтобы зафиксировать факт возникновения ошибки, наберите
throw тип_ошибки;

Обратите внимание, что на количество блоков `catch` ограничения нет. Когда фиксируется исключительная ситуация (ошибка), ищется тип ошибки, которому она соответствует, и выполняются оставленные на этот случай инструкции.

Все исключения .NET являются классами, производными от класса `Exception`. Объект класса `Exception` содержит свойство Message (Сообщение). Вы можете присвоить значение этому свойству (определить содержание сообщения) в момент создания объекта `Exception`.

Рассмотрим пример небольшой программы. В процессе ее выполнения пользователю снова и снова предлагается ввести число, из каждого введенного числа извлекается квадратный корень, а полученное значение отображается на экране. Если пользователь вводит значение, которое не может быть преобразовано в число (например, он вводит слово), программа реагирует на эту ошибку и вычисляет квадратный корень числа, заданного по умолчанию. Если же пользователь вводит отрицательное число, программа также реагирует на эту ошибку, что сопровождается отображением специального сообщения.

```
//SquareRoot

//Обработка исключительных ситуаций

#include "stdafx.h"

#using <mscorlib.dll>

using namespace System;

__gc class SquareRootException : public Exception

public:
}; SquareRootException(String *s) : Exception(s) {};

//Вычисление квадратного корня
//Фиксирование ошибки, вызванной вводом отрицательного числа
double SquareRoot(int nNumber)
{
    if (nNumber < 0)
    {
       throw new SquareRootException (S"Число должно быть
                          положительным");
    }
    return Math::Sqrt(nNumber);
}

//Предлсжение пользователю ввести число
//Функция возвращает введенное число
int GetNumber()
{
  int nNumber;
```

```
    Console::WriteLine(S"Введите число");
    try
    {
    nNumber = Int32::Parse(Console::ReadLine());
    }
    catch (FormatException *e)
    {
    Console::WriteLine(e->Message);
    Console::WriteLine("В следующий раз наберите, пожалуйста,
                  число. Сейчас введенное вами значение будет
                  воспринято как число 1");
    nNumber = 1;
    }
    return nNumber;
}

//С этой строки начинается выполнение программы
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
    int nNumber;

    //Программа предлагает пользователю ввести новое число до
    //тех пор, пока он не вводит число 0
    while (nNumber = GetNumber())
    {
       //Вычисление результата
       try
       {
       Console::WriteLine(S"Квадратный корень числа {0} равен
                  числу {1}", nNumber.ToString(),
                  SquareRoot(nNumber).ToString());
       }
       catch (Exception *e)
       (
       Console::WriteLine(e->Message);
    }

    //Завершение работы программы
    Console::WriteLine(S"Bye");
    return 0;
}
```

Начните чтение кодов этой программы с блока try, который предназначен для вычисления и отображения значения квадратного корня числа, введенного пользователем:

```
 try
 {
 Console::WriteLine(S"Квадратный корень числа {0} равен
                  числу {1}", nNumber.ToString(),
```

```
                SquareRoot(nNumber).ToString());
}
```

Если пользователь введет отрицательное число, оно будет передано функции `SquareRoot`, которая отреагирует на это фиксированием исключения с помощью команды `throw` и передачей сообщения об ошибке конструктору `SquareRootException`:

```
if (nNumber < 0}
{
    throw new SquareRootException(S"Число должно быть
                                  положительным");
}
```

Коды `catch` набраны сразу же после блока `try`. В данном случае здесь присутствует только одна команда `catch`, которая реагирует на все исключения, фиксируемые внутри блока `try`. Информация об исключениях передастся в виде объекта класса `Exception`, который имеет собственное имя. Процедура `catch` отображает на экране значения свойства Message этого объекта:

```
catch (Exception *e)
{
   Console::WriteLine(e->Message);
}
```

Итак, посмотрим, как работает эта программа. Все начинается с запуска цикла. Перед выполнением каждой итерации пользователю предлагается ввести какое-то число. Если пользователь вводит то, что не может быть преобразовано к значению типа integer, функция `GetNumber` фиксирует ошибку и выдает предупреждающее сообщение. При этом вычисляется квадратный корень значения, заданного по умолчанию (в данном случае это значение равно числу 1). Внутри цикла предпринимается попытка выполнить коды блока `try`, цель которых вычислить квадратный корень введенного числа и отобразить полученное значение на экране.

Если пользователь ввел положительное число, вызываемая функция `SquareRoot` не зафиксирует ошибку, в резельтате чего все коды блока `try` будут благополучно выполнены и пользователь увидит на экране вычисленный результат. Затем программа переходит к следующей итерации, и, если пользователь не вводит нулевое значение, снова происходит попытка выполнить коды блока `try`.

Если пользователь вводит отрицательное значение, функция `SquareRoot` фиксирует возникновение исключительной ситуации и передаст управление колам блока `catch`. В данном случае их выполнение приводит к отображению на экране специального предупреждающего сообщения. Затем программа переходит к следующей итерации. Как только пользователь вводит число 0 (нуль), программа завершает работу.

Все это **хорошо,** но несколько запутанно

Благодаря возможности передавать объекты классов при возникновении исключительных ситуаций решаются две проблемы. Во-первых, можно определить столько исключений, сколько будет необходимо, а во-вторых, с объектами можно передавать всю необходимую информацию об ошибке и мерах, которые следует предпринять в связи с ее возникновением.

Например, если ошибка вызвана отсутствием свободного места на диске, желательно было бы указать название этого диска. Если ошибка связана с тем, что поврежден какой-то файл, неплохо было бы указать его имя и адрес последнего удачно считанного фрагмента. Вы могли бы даже определить процедуру, которая попытается восстановить этот файл.

Но когда вы передаете объект класса с помощью команды throw, как компилятор определит, какой из блоков catch должен быть использован? Это важный момент. Компилятор проверяет тип данных, которые передаются командой throw, и ищет ту команду catch, которая знает, как данные такого типа должны быть обработаны. Так, вы можете создать классы FileCorrupt (повреждение файла), DiskError (ошибка диска), MemoryError (ошибка памяти), каждый из которых будет содержать в себе информацию, описывающую данную проблему. Сами классы могут кардинально отличаться друг от друга.

Затем, на случай обнаружения ошибки диска, наберите следующее:

```
DiskError foo;
//Здесь наберите коды, наполняющие содержанием объект foo
throw foo;
```

На случай, если будет поврежден файл, наберите такой код:

```
CorruptFile bar;
//Здесь наберите коды, наполняющие содержанием объект bar
throw bar;
Затем, благодаря тому что команды catch определяют совпадение типов
данных или классов, управление передается соответствующему блоку
catch:
//Инструкции на случай возникновения ошибки диска
catch (DiskError MyError) { }
//Инструкции на случай повреждения файла
catch (CorruptFile MyError) { }
```

Использование классов позволяет весьма элегантно достичь впечатляющей гибкости в управлении исключительными ситуациями, поскольку классов, описывающих ошибки, можно создать любое количество, а в самих классах можно обозначить всю необходимую информацию.

При желании для определения типа ошибки можно использовать ппростой тип данных, например char *. Но использовать классы все же гораздо предпочтительнее, поскольку с ними вы можете передать полную и четкую информацию о том, в чем именно заключается проблема. Процедура, в которой ошибка была обнаружена, располагает всеми необходимыми данными. Процедура, которая будет обрабатывать эту ошибку, должна получить эти данные, чтобы помочь пользователю справиться с возникшей проблемой.

Еще одно преимущество использования классов состоит в том, что вы можете снабдить их функциями-членами. Например, можно создать внутри класса набор процедур, отображающих сообщения на экране и помогающих корректировать ситуацию или управлять ею.

Наследование классов, описывающих исключения

Уже созданные классы обработки исключений могут быть наследованы для создания новых производных классов. Это позволит повторно использовать готовые работающие коды.

Кроме того, создать собственный, описывающий исключительную ситуацию класс можно путем наследования класса Exception. Это позволит использовать встроенные функциональные возможности, а определенные вами исключения будут выглядеть так же, как и остальные исключения .NET.

При написании неуправляемых кодов вы не сможете наследовать класс Exception. Вам в любом случае придется самостоятельно создавать собственный класс.

Преобразование типов и работа оператора catch

Если тип ошибки может быть легко преобразован к типу, на который реагирует команда `catch`, фиксируется соответствие.

Например, тип `short` можно легко преобразовать к типу `int`. Поэтому такая команда `catch` отреагирует на исключение, генерируемое командой `throw`:

```
catch (int k) { }
. . .
throw (short i = 6);
```

Точно так же, если команда `throw` возвращает класс `Derived`, который является производным класса `Base`, указатель на производный класс может быть воспринят как указатель на базовый класс:

```
catch (Base foo) { }
. . .
Derived foo;
throw foo;
```

В данном случае эта команда `catch` также отреагирует на ошибку. Иаодя из этого, при создании класса, описывающего исключение, функции-члены желательно сразу делать виртуальными.

Компилятор выполняет блок кодов первого оператора `catch`, который фиксирует соответствие с типом возникшей ошибки. Поэтому, если вы используете сразу несколько производных классов, перечислите вначале соответствующие им блоки `catch` и только затем наберите блок `catch`, соответствующий базовому классу. В таком случае программа будет работать правильно, и специфические исключения (представляемые производными классами) будут фиксироваться именно соответствующими им командами `catch`. В противном случае, если первым будет набран блок `catch` базового класса, он будет реагировать на все подряд исключительные ситуации, представляемые производными классами.

Пять правилисключительного благополучия

Вот пять простых правил, которых следует придерживаться при написании кодов обработки исключений.

- ✓ **Командой `throw` предпочтительнее передавать классы, чем обычные типы данных.** Это позволит вам более точно указать, в чем именно заключается проблема.
- ✓ **Классы также хороши тем, что с ними при необходимости можно передавать функции-члены.**
- ✓ **Создайте классы, описывающие все категории ошибок, с которыми может столкнуться ваша программа.**
- ✓ **Убедитесь, что тип, передаваемые командой `throw`, в точности соответствуют типу, который должна идентифицировать команда `catch`.** Например, если команда `throw` передает тип `DiskError *`, команда `catch` также должна реагировать на тип `DiskError *`, а не на `DiskError`.

- ✓ Если вы запутаетесь с типами ошибок, могут возникнуть самые разные последствия. (Обратите внимание, что, если компилятор сможет преобразовать один тип к другому, как, например, может преобразовать тип `float` к типу `int`, он сделает это для того, чтобы найти блок `catch`, соответствующий зафиксированной ошибке. Более подробную информацию вы можете найти в разделе "Преобразование типов и работа оператора catch".)
- ✓ **Если для возникшего исключения не будет найден соответствующий блок** `catch`, **работа программы автоматически прерывается.** Будьте готовы к этому. К счастью, поскольку объекты, сохраненные в стеке, уничтожаются, вызываются их деструкторы и программа благополучно освобождает выделенную для ее выполнения память.

Глава 21

Потоки данных

В этой главе...

- Чтение и запись в файлы и из файлов
- Чтение и запись данных различных типов
- Контроль формата исходящих данных

Почти все программы занимаются тем, что принимают одни данные и возвращают другие. Библиотеки классов среды .NET включают в себя набор *потоковых классов,* предназначенных для организации процесса ввода-вывода. (Иногда ввод-вывод обозначается как I/O, что является сокращением от слов input/output.) Если вы пишете неуправляемую программу, к вашим услугам — библиотека функций *iostream,* которые также обслуживают процесс ввода-вывода данных.

Считывать и сохранять данные можно множеством способов. И библиотека iostream, и .NET-классы ТО обеспечивают режим, при котором вам не нужно вникать во множество технических деталей, так как все они выполняются автоматически.

NET-классы I/O

В пространстве имен (namespace) System.IO среды .NET есть некоторое количество классов, предназначенных для организации процесса ввода-вывода информации в файлы и из файлов. В этой главе вы познакомитесь с классом `StreamReader`, который обеспечивает процесс чтения текста из файла, и с классом `StreamWriter`, который обеспечивает процесс записи текста в файл. Класс `File` используется как вспомогательный для создания объектов `StreamReader` и `StreamWriter`. Вот как открывается потоковый файл для записи текста:

```
StreamWriter *poWStream = File::CreateText(S"test.txt");
```

Запись в потоковый файл осуществляется посредством вызова метода Write:

```
poWStream->Write(S"Hello this is a test");
```

Метод Write является перегружаемым, благодаря чему вы можете записывать в файл значения различных типов, например double или integer. Если нужно записать информацию, которая заканчивается символом окончания строки, вызовите метод `WriteLine`. К тому же не забывайте, что методы Write и `WriteLine` обладают возможностями форматирования, а потому вы можете набирать коды, наподобие этого:

```
poWStream->Write(S"Hello {0}, you are {1} and {2}!", sName,
                                                    nAge, sCute);
```

Записав в файл всю необходимую информацию, закройте поток с помощью метода `Close`:

```
poWStream->Close();
```

Подобным образом осуществляется чтение из файла посредством использования объекта класса `StreamReader`:

```
poRStream = File::OpenText(S"test.txt");
Console::WriteLine(poRStream->ReadLine());
poRStream->Close();
```

Чтобы получить доступ к пространству имен System.IO, наберите в начале программы такой код:

```
using namespace System::IO;
```

Записали — прочитали

В этом разделе рассматривается пример программы, в процессе выполнения которой вначале в файлы записываются текстовые и числовые данные, а затем те же данные считываются из этих же файлов. Текстовые данные записываются в один файл, а числовые — в другой:

```
//Открытие потока для записи текста
StreamWriter *poWStream = File::CreateText(S"text.txt".) ;
//запись текста в файл
poWStream->WriteLine(S"Hello this is a test");
//Закрытие потока
poWStream->Close();

//Открытие другого файла для записи числовых данных
poWStream = File::CreateText(S"number.txt");
poWStream->WriteLine(15);
poWStream->WriteLine(42);
poWStream->WriteLine(1);
poWStream->Close();
```

Далее используется объект класса StreamReader для открытия первого файла с последующим вызовом метода ReadLine для считывания текстовых данных:

```
//Открытие первого файла, считывание строк и отображение
//их на экране
StreamReader *poRStream = File::OpenText(S"text.txt");
String* pszWords = poRStream->ReadLine();
Console::WriteLine(pszWords);
poRStream->Close();
```

Затем считываются данные из второго файла. Обратите внимание, что в файле содержится несколько значений. Метод ReadLine, достигнув конца файла, возвращает нулевое значение, что дает возможность использовать цикл while для того, чтобы прочитать из файла все данные. (О циклах while речь идет в главе ! 1.)

```
//Считывание числовых данных
String* pszLine;
int nTempNum;
poRStream = File::OpenText(S"numbers.txt");

//Считывание значений до тех пор, пока не будет достигнут
//конец файла
while ((pszLine = poRStream->ReadLine()) != 0)
{
   nTempNum = Int32::Parse(pszLine);
   Console::WriteLine(nTempNum);
}
```

```
poRStream->Close();
```

Функции библиотеки iostream

В библиотеке iostream содержится ряд функций, которые помогут записать данные в файл и считать их оттуда. Файл для записи открывается таким несложным кодом:

```
ofstream foo("filename");
```

Специальный класс ofstream предназначен для передачи файлам выходных данных. (Буквы *o* и *f* взяты от слов *output* и *files*. Слово *stream* переводится как *поток* — термин, которым в C++ обозначаются объекты, предназначенные для организации процесса ввода-вывода данных.) В данном случае foo является потоковой переменной (объектом класса ofstream). При создании объекта foo конструктору класса ofstream передается имя файла, который должен быть открыт. Передача данных осуществляется с помощью оператора <<:

```
//Запись в файл текста "Hello World"
foo << "Hello World"
```

Класс ofstream имеет близнеца— класс ifstream, который предназначен для организации процесса чтения данных из файлов. (Здесь буквы *i* и *f* обозначают слова *input* и *files.)* Использование класса ifstreamосуществляется аналогичным образом:

```
ifstream foo("filename");
//Выделение области, куда будут считываться текстовые данные
char buffer[100];
//Считывание текстовых данных из файла
foo >> buffer;
```

Числовая информация считывается несколько иначе:

```
int MyInt;
foo « MyInt;
```

Да, и самое важное! Перед тем как использовать все эти возможности, нужно подключить библиотеку iostream, для чего в начале программы наберите

```
#include <fstream.h>
```

Повторное использование потоковых переменных

Когда потоковая переменная прекращает свое существование, представляемый ею файл закрывается. Но закрыть этот файл можно и раньше, воспользовавшись функцией-членом close:

```
//Закрытие файла, представляемого потоковой переменной foo
foo.close();
```

Обычно это делается в тех случаях, когда хотят одну и ту же потоковую переменную использовать для доступа к нескольким файлам. Чтобы сделать это, нужно вначале закрыть один файл, а затем с помощью функции-члена open открыть другой:

```
//Открытие файла "foo.txt"
foo.open("foo.txt");
```

Если хотите проверить, осталась ли еще в файле какая-то непрочитанная информация, используйте функцию-член eof, которая возвращает значение true (Истина) в момент достижения окончания файла.

Записали — прочитали

Не волнуйтесь, это не дежа-вю. И даже не опечатка. Чуть ранее вы действительно ви-дели раздел с точно таким же названием, но тогда описывался доступ к файлам с помощью .NET-классов I/O, а теперь рассматривается, как выполняются те же действия с использо-ванием функций библиотеки iostream.

В этом разделе также рассматривается пример программы, в процессе выполнения кото-рой вначале в файлы записываются текстовые и числовые данные, а затем эти же данные считываются. Текстовые данные записываются в один файл, а числовые — в другой:

```
//Открытие файла new.txt
ofstream oOutFile ( "new.txt") ;

//Запись Б этот файл текстовой информации
oOutFile << "Row, row, row your boat";

//Теперь этот файл закрывается
oOutFile.close();

//Открытие другого файла и запись в него числовых данных.
//'Обратите внимание, что числа отделяются друг от друга
//пробелами
oOutFile.open("number.txt");
oOutFile << 15 << " " « 42 << " " << 1;
OutFile.close();
```

Далее для открытия файлов и считывания информации используется класс `ifstream`. В данном случае файл открывается не с помощью конструктора при создании потоковой пере-менной, а непосредственно, путем вызова метода `open`. Не имеет значения, какой способ вы выберете, оба работают одинаково хорошо.

```
ifstream oInFile;
oInFile.open{"new.txt") ;
```

Для сохранения информации, считываемой из файла, создается буфер. В данном случае размер буфера равен пятидесяти байтам. Убедитесь, что размер используемого вами буфера больше самого большого элемента, который должен быть считан. Если вы точно знаете, эле-менты какого размера должны считываться, сделайте буфер больше, чем размер любого из этих элементов. С другой стороны, установив размер буфера, вы можете ограничить макси-мальный размер принимаемых программой данных.

Данные из файла считываются в буфер и затем по одному элементу за раз отображаются на экране:

```
oInFile >> p;
cout « p << endl;
oInFile >> p;
cout << p << endl;
```

Затем данные считываются из файла, содержащего числовую информацию. Чтобы прочи-тать все числа, используется цикл `while`. Функция `eof` на каждой итерации цикла проверя-ет, остаются ли еще данные для считывания. Поскольку символ окончания файла интерпре-тируется компилятором как число, чтобы он тоже не был отображен на экране как одно из чисел, считываемых из файла, перед командой отображения данных на экране дополнительно используется оператор `if`:

```
//Считывание данных из файла
while (!oInFile.eof())
```

```
{
   //Чтение числа
   oInFile » nTempNum;
   //Если число не является символом окончания файла, оно
   //отображается на экране
   if ( !oInFile.eof())
      cout << nTempNum << endl;
}
oInFile.close();
```

Обратите внимание также на то, что информация из текстовых файлов считывается по одному слову за раз. Это очень удобно для последующей обработки считываемых данных.

Специальные команды, погружаемые в потоки данных

Ниже вы можете увидеть некоторые команды, которые включаются в поток для того, чтобы управлять способом чтения и записи данных. Сами по себе эти команды (называемые *потоковыми манипуляторами)* никуда не записываются и не считываются. Они лишь определяют особенности считывания или записи следующих за ними элементов:

dec — число считывается или записывается как значение типа decimal;
hex — число считывается или записывается как значение типа hex;
oct — число считывается или записывается как значение типа octal.

Например, если вводимое пользователем число должно быть интерпретировано как значение типа hex, наберите такой код:

```
cin » hex >> TempNum;
```

Можно также использовать приведенный ниже код как вариант преобразования значения типа hex к значению типа decimal:

```
//Чтение числа как значения типа hex
cin >> hex >> TempNum;
//Отображение этого же числа как значения типа decimal
cout << dec « TempNum;
```

Формат передаваемых данных

По умолчанию числа выводятся на экран, на печать или записываются в файлы без добавления между ними каких-либо символов или пробелов. Если необходимо, их можно разделять между собой дополнительными пробелами. Это особенно удобно, если вы хотите разместить числа по столбцам. Чтобы отделить числа друг от друга пробелами, используйте функцию-член width:

```
//Отображение чисел с интервалом в 20 пробелов
cout.width(20);
```

Вместо пробелов могут быть использованы какие-то другие символы. Например, можно заполнить пространство между числами звездочками (*), чтобы никто не смог дописать, скажем, к платежному чеку парочку нулей:

```
//Заполнение звездочками пространства между отображаемыми числами
cout.fill('*');
```

Этот же прием можно использовать и при записи числовых данных в потоковые файлы.

Записывать данные в файлы, отображать на экране, а также считывать их оттуда можно тысячами способов. Есть еще множество используемых для этого дополнительных классов, которые в книге не рассматриваются. Если вы хотите стать мастером ввода-вывода, обрати-

тесь к разделу справочной системы Visual C++. который посвящен функциям iostream. Помимо чтения общих разделов о потоках данных и потоковых классах, можете обратить особое внимание на разделы, посвященные методам форматирования, форматирующим флагам и манипуляторам.

Кое-что о работе с файлами

Ниже приведено несколько советов, которые помогут вам при работе с файлами данных.

- ✓ **Старайтесь одновременно не записывать данные в файл и не считывать их** оттуда. Формально это делать можно, но иногда такие действия (особенно это касается начинающих программистов) могут привести к тому, что возникнет путаница в определении места, с которого данные должны считываться, и места, начиная с которого данные должны записываться.
- ✓ **Если для записи числовых данных используется команда << библиотеки iostream, числа записываются как текст без каких-либо пробелов между ними.** Так, если вы хотите, чтобы результатом записи чисел в файл была строка

  3,4 5,6 66,28 8,

  а не

  3,45,666,288,

  нужно позаботиться о том, чтобы после каждого записываемого числа был добавлен пробел. Для этого можете использовать, например, такой код:

  ```
  cout << foo « " ";
  ```
- ✓ **В конце файла размещается специальный символ (называемый символом окончания файла и иногда обозначаемый как *eof*), который говорит компилятору: "Эй! На мне файл заканчивается".** Как только компилятор видит этот символ, он сразу же определяет, что данных для считывания в этом файле больше нет.
- ✓ **Строка считывается до тех пор, пока не будет достигнут символ окончания файла или символ перехода на новую строку** (\n). Если последняя строка файла не заканчивается символом перехода на новую строку, сразу же после ее считывания фиксируется факт окончания файла.
- ✓ **Если для считывания числовых данных используется команда » библиотеки iostream, символ окончания файла не распознается при считывании последнего числа.** (В этом, например, отличие C++ от языка C, в котором при считывании последнего числа определяется факт окончания файла.) Поэтому, если вы не обратите внимание на данное обстоятельство, программа прочитает символ окончания файла как еще один элемент и попытается преобразовать его в число.

Глава 22

Создаем пользовательский интерфейс

В этой главе...

- Что обозначает термин WinForms
- Создание форм и управляющих элементов
- Обработка событий

Наверное, компьютерные гуру компании Microsoft, работавшие над созданием среды .NET, хотели быть уверены, что их творение полюбят все. Среда .NET способна решать разнообразнейшие задачи: от управления вводом-выводом данных, до создания программ, работающих в Web и оснащенных самым потрясающим графическим интерфейсом. Эта глава посвящена вопросам разработки графического пользовательского интерфейса (Graphical User Interface — GUI) с использованием .NET-классов GUI, которые обычно обозначаются термином WinForms. (Формально эти классы называются Windows Forms, но программисты, которые с ними работают, называют их просто WinForms.)

.NET-классы WinForms подключаются к интерфейсу среды Windows и взаимодействуют с его элементами (диалоговыми окнами, кнопками, меню, списками и т.п.). Они непосредственно обращаются к компонентам среды Windows, которые представляют отдельные элементы пользовательского интерфейса. Например, объект класса System.Windows.Forms.CheckBox выглядит и работает точно так же, как и любой другой флажок (check box), расположенный где-нибудь в диалоговом окне Windows. Это означает, что вы можете создавать .NET-приложения, используя все преимущества среды .NET, CLR и всего, что им сопутствует; в то же время они будут иметь все характеристики и возможности приложений Windows.

Основной набор инструментов

Классы WinForms представляют одну из частей Windows API (Application Programming Interface— программный интерфейс приложения). Другие части Windows API представлены другими компонентами .NET. Например, интерфейсу графических устройств (Graphical Device Interface — GDI) в среде .NET соответствуют классы System.Drawing.

Большинство классов WinForms используются для создания визуальных средств управления, которые являются составными компонентами среды Windows. Некоторые из них, например объекты класса ContextMenu, не всегда отображаются на экране. Способ отображения других заранее определен. Так, например, объекты класса MainMenu всегда отображаются в верхней части окна программы.

В табл. 22.1 приведен список тех классов WinForms, которые, вероятнее всего, вы будете использовать в повседневной работе.

Таблица 22.1. Наиболее часто используемые классы WinForms

Класс System.Windows.Form	Описание
Application	Управляет программой, в том числе ее запуском и остановкой
Button	Кнопка, иногда называемая также командной кнопкой. Щелчок на кнопке влечет за собой выполнение каких-то действий
CheckBox	Флажок. Можно выбирать сразу несколько флажков, принадлежащих одной группе (в отличие от переключателей)
CheckedListBox	Список флажков, который можно прокручивать. Элемент может вмещать в себя множество флажков, занимая при этом небольшое пространство
ComboBox	Поле со списком. В нем вы можете либо набрать собственное значение, либо выбрать из списка одно из предлагаемых значений. Существует несколько видов полей со списком
ContextMenu	Контекстное меню (открывается после щелчка правой кнопкой мыши). Позволяет пользователю выбрать команды или опции, относящиеся к тому элементу, на котором он щелкнул
DataGrid	Таблица данных, в которой отображается информация, сохраненная в источнике данных ADO.NET. Столбцы таблицы данных соответствуют полям в источнике данных, а строки – записям
FileDialog	Диалоговое окно, предоставляющее пользователю возможность обзора для выбора файла, который должен быть открыт, или папки, в которой должен быть сохранен новый файл
Form	Базовый класс WinForms, из которого путем наследования создаются классы, описывающие все формы вашей программы. Производные классы могут представлять как постоянные, так и диалоговые окна
Label	Надпись. Отображает текст, который обычно относится к другим элементам управления. Пользователь не может изменить текст надписи
LinkLabel	Гиперссылка, щелчок на которой приводит к открытию Web-страницы или к другим действиям
UstBox	Список. Пользователю предоставляется возможность выбрать одно из предлагаемых значений
ListView	Список, содержащий в себе элементы и соответствующие им пиктограммы. Такие списки широко используются в среде Windows, и вы наверняка часто с ними сталкивались
MainMenu	Главное меню. Отображается в виде строки в верхней части окна программы
PictureBox	Рисунок. Используется для отображения графики в окне формы
ProgressBar	Индикатор выполнения. Отображает сообщения о выполняемой операции

Класс System.Windows.Form	Описание
RadioButton	Переключатель. Из группы переключателей может быть выбран только один переключатель
RichTextBox	Окно с плотным текстом. Отображает форматированный текст и позволяет пользователю набирать новый текст, который в дальнейшем может быть использован программой. Форматирование может включать полужирное начертание, курсив, маркеры абзаца и гиперссылки
Splitter	Разделитель. Позволяет пользователям изменять в окне формы размеры элементов управления
StatusBar	Строка состояния, располагающаяся в нижней части окна формы. Отображает информацию и сообщения для пользователя
TabControl	Элемент управления, содержащий набор вкладок. Пользователь может переходить от одной вкладки к другой, щелкая на их корешках
TextBox	Текстовое поле. Отображает текст и позволяет пользователю набрать свой текст, который в дальнейшем будет использован программой. Текстовые поля могут состоять из нескольких строк
ToolBar	Панель инструментов, на которой размещаются командные кнопки. Щелкая на кнопках, пользователь может инициировать те или иные действия
TreeView	Дерево, представляющее информацию в иерархическом виде, как, например, дерево папок в окне Windows Explorer

Формы

Сейчас, наверное, перспектива создания программ с использованием инструментов WinForms кажется вам весьма сомнительной. Если вы уже пытались создавать программы для Windows и использовали для этого такие приложения, как, например, MFC, вы можете подумать, что при использовании классов WiiiForms вам придется столкнуться с теми же проблемами. Не переживайте, это не так. Компания Microsoft учла свой предыдущий опыт, и ее новое творение — среда .NET — объектно-ориентированна и отличается особой простотой в использовании.

Чтобы создать форму, нужно выполнить три основных шага.

1. Создать новый класс путем наследования класса System.Windows.Forms.Form.
2. Настроить новую форму (разместить на ней управляющие элементы и т.п.).
3. Определить порядок взаимодействия программы и созданной формы.

Наследование форм

Все формы, которые используются в вашей программе, создаются путем наследования класса WiiiForms, который так и называется — Form. В процессе наследования определяется, как новая форма будет выглядеть и как она будет работать. Производный класс от класса Form создается очень просто.

```
 gc class MyForm : public Form
T
public:
  MyForm();
};
```

Используйте код gc, чтобы активизировать возможность автоматического управления памятью, выделяемой для объектов создаваемого класса.

Для нового класса объявите конструктор, с помощью которого впоследствии будут определены все характеристики создаваемой формы.

Настройка формы

Основная работа, которая касается определения параметров новой формы, состоит в добавлении к ней элементов управления. Осуществляется это следующим образом.

1. **Создайте объекты классов, которые описывают отдельные элементы управления.**

```
Button *poOK = new Button();
RichTextBox *poTE = new RichTextBox();
PictureBox *poImg = new PictureBox();
```

2. **Установите свойства элементов управления.**

```
//Определение свойств кнопки
poOK->Text = "OK";
poOK->Left = 280;
poOK->Top = 350;
poOK->BackColor = Color::Blue;
```

3. **Добавьте элементы управления к коллекции Controls вашей формы.**

```
this->Controls->Add(poOK);
this->Controls->Add(poTE);
this->Controls->Add(poImg);
```

Указатель this-> используется в тех случаях, когда вы набираете колы функций-членов, к числу которых относится, например, конструктор. Здесь этот указатель используется, чтобы с помощью возможности редактора Visual C++ выделять цветом отдельные синтаксические элементы, тем самым обозначив, что набираемые коды относятся ко всей форме, а не только к одному из ее элементов управления.

Кроме того, вы можете определить свойства самой формы:

```
//Определение размеров формы
this->Width = 600;
this->Height = 400;
```

Открытие формы

После того как вы создадите форму и определите ее характеристики, вам еще нужно будет сообщить компилятору, что эта форма должна быть главной формой вашей программы. Для этого используется класс System.Windows.Forms.Application. Метод Run этого класса принимает в качестве аргумента указатель созданной формы и делает ее главной для вашей программы. Обычно метод Run вызывается функцией main.

```
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  Application::Run(new MyForm());
  return 0;
}
```

Обработка событий

Создание любого графического пользовательского интерфейса подразумевает разработку программ, обрабатывающих события. Осуществляется это посредством объектно-ориентированного программирования. Сами события в Windows и в среде .NET происходят постоянно и в основном являются следствием действий пользователя. Например, перемещение окна, щелчок на кнопке, выбор пункта меню — все это воспринимается как событие.

Создавая программу с использованием инструментов WinForms, вы должны определить, на какие события должна реагировать программа. Обрабатывать все события подряд не имеет никакого смысла. В действительности в процессе работы программы могут происходить тысячи событий, поэтому, если бы все их нужно было обрабатывать, только на написание программ уходили бы годы, не говоря уже о процессе отладки и об устранении ошибок.

Обрабатывать события можно двумя способами: либо использовать для этого классы, полученные путем наследования, либо делегировать управление событием другому классу. Например, функции-члены производного класса, представляющего форму, могут непосредственно обрабатывать события, касающиеся самой формы. Но допустим, что в окне этой формы есть кнопка, на которой должен щелкнуть пользователь. Вам, наверное, не хотелось бы создавать еще один отдельный класс, представляющий кнопку, только для того, чтобы определить, что должно произойти после щелчка на ней. Поэтому среда .NET включает в себя концепцию *делегирования,* которая позволяет событие, относящееся, например, к кнопке, обрабатывать функцией-членом формы, на которой эта кнопка расположена.

Как видите, процесс обработки событий тесно связан с объектно-ориентированным программированием. Создавая путем наследования класс, представляющий форму, вы добавляете к нему функции-члены, делающие эту форму рабочей, включая делегированные, которые делают рабочими элементы управления, расположенные на этой форме.

Объекты, которые умеют реагировать

Представляющие формы и элементы управления классы имеют множество встроенных функций-членов, предназначенных для обработки событий. Таким образом значительно упрощается процесс создания собственных программ, обрабатывающих события. Еще один плюс состоит в том, что, если вам необходимо как-то изменить поведение производного класса, просто воспользуйтесь возможностью перегрузки и измените нужные вам функции-члены. Имена всех функций-членов, предназначенных для обработки событий, начинаются с букв On. Например, функция-член, предназначенная для обработки события `MouseMove` (перемещение курсора мыши), называется `OnMouseMove`. Список названий всех функций-членов, обрабатывающих события, вы можете найти в справочной системе Visual C++ .NET в разделе Protected Instance Methods. Поскольку все эти функции являются защищенными (protected), они могут быть перегружены и вызваны только из производных классов.

Предположим, например, что вы хотите, чтобы при перемещении пользователем курсора мыши в окне созданной вами формы для курсора постоянно отображалось текущее значение координаты X. Чтобы сделать это, вам нужно перегрузить функцию-член `OnMouseMove`, но для этого вы должны знать ее сигнатуру. (Сигнатура функции определяет тип возвращаемого ею результата, а также количество и тип передаваемых ей аргументов. Дополнительную информацию о сигнатуре функций вы найдете в главе 18.) Вы можете найти описание функции OnMouseMove в справочной системе Visual C++ или просто воспользоваться возможностью IntelliSense редактора Visual C++. (Попробуйте набрать On и, удерживая клавишу <Ctr!>, нажмите пробел.) И в том и в другом случае вы увидите, что функция OnMouseMove должна быть объявлена таким образом:

```
void OnMouseMove(MouseEventArgs *e);
```

Вы, наверное, спросите: "Что это еще за MouseEventArgs?" Дело в том, что каждая функция, обрабатывающая событие, должна иметь какую-то информацию о самом событии. Так, например, функция OnMouseMove вызывается каждый раз, когда пользователь перемещает курсор мыши, но она также должна знать, *куда* курсор перемещается. Именно для этого и существует аргумент MouseEventArgs, у которого есть свойства, позволяющие определить текущие координаты X и Y курсора мыши. Для других событий предусмотрены другие аргументы.

Каждая функция обработки событий в качестве аргумента получает указатель, ссылающийся на объект производного класса, полученного путем наследования класса EventArgs (или же на объект самого класса EventArgs). Существует почти сотня классов, производных от класса EventArgs.

После того как функция OnMouseMove объявлена, она должна быть определена. Чтобы отображать значение координаты X курсора мыши, добавьте к классу в качестве члена данных элемент управления Label (Надпись) и свойству Text этого объекта присвойте значение свойства X объекта MouseEventArgs:

```
void MyForm::OnMouseMove(MouseEventArgs *e)
{
   //Отображение координаты X
   m_poXPosition->Text = e->x.ToString();
```

В процессе выполнения этой программы вы увидите, что надпись со значением координаты X обновляется каждый раз, когда курсор перемещается над пустыми участками формы. Дело в том, что при размещении курсора над любым элементом управления, событие `MouseMove` будет относиться уже к этому элементу и сама форма не будет реагировать на перемещение курсора.

Хорошие менеджеры умеют делегировать свои обязанности

Создание программ, обрабатывающих события, было бы весьма утомительным, если бы для каждого события нужно было создавать новый производный класс и перегружать функции-члены. Среда .NET упрощает эту задачу, позволяя делегировать другим классам возможность реагировать на то или иное событие. Делегирование можно использовать в отношении как форм, так и элементов управления. Так, вам не нужно создавать новые классы для каждого отдельного элемента управления. Если программа должна как-то отреагировать на событие, совершенное пользователем в отношении какого-то элемента управления (например, пользователь щелкнул на кнопке Выход), делегируйте эту обязанность классу, представляющему форму, где эта кнопка расположена. Функции-члены, которые обрабатывают делегированные события, называются делегатами (delegate).

Чтобы добавить к какому-то классу делегированную функцию, нужно выполнить два основных шага.

1. **Объявить и определить функцию-член, обрабатывающую событие.**
2. **Добавить** делегата.

Объявление и определение обрабатывающей событие функции

Каждому событию соответствует делегирующий класс, который определяет, как должна выглядеть обрабатывающая это событие функция-член. Событию Click, например, соответствует делегирующий класс EventHendler, функция-член которого требует для себя значения двух аргументов:

```
void Change_Clicked(Object *sender, EventArgs *e);
```

Аргумент sender является объектом, который генерирует событие. В нашем примере это будет кнопка, на которой щелкнул пользователь. Аргумент EventArgs содержит в себе информацию о событии. В данном случае все, что вам нужно знать, — на какой кнопке щелкнул пользователь, поэтому здесь используется стандартный объект класса EventArgs, который не несет в себе никакой дополнительной информации.

В .NET отсутствуют какие-либо дополнительные требования относительно наименования функций-членов, обрабатывающих события. Однако лучше присваивать такие имена, которые как-то будут указывать на обрабатываемое событие и на объект, к которому это событие относится. Например, название Change_Clicked говорит о том, что здесь обрабатывается событие Clickдля кнопки Change.

Из определения функции Change_Clickedвидно, что после щелчка на кнопке Change изменяется рисунок, отображаемый объектом PictureBox:

```
void MyForm::Change_Clicked(Object *sender, EventArgs *e)

    //Изменение отображаемого рисунка
    ra_polmg->Image = Image::FromFile("..\\desert5.jpg");
}
```

Добавлениеделегата

Просто создать процедуру обработки события недостаточно. Нужно еще сообщить компилятору, когда эта процедура должна выполняться. Нужно связать событие Click с только что созданным делегатом.

Процесс добавления делегата включает в себя четыре основных элемента:

- ✓ элемент управления, к которому относится событие;
- ✓ событие, которое должно быть обработано;
- ✓ делегирующий класс этого события;
- ✓ функция-член, обрабатывающая событие.

В нашем примере элементом управления является кнопка Change, представляемая членом данных poChange. Обрабатываемое событие— Click (щелчок кнопкой мыши), которому соответствует делегирующий класс EventHandler. Функцию-член мы назвали именем Change_Clicked. Итак, чтобы добавить этого делегата, наберите такой код:

```
poChange->Click += new EventHandler(this, Change_Clicked);
```

Этот принцип распространяется и на все другие события. Например, чтобы обработать событие ChangeUICues (смысл которого в том, что пользователь нажимает клавишу <Tab> или щелкает кнопкой мыши на элементе управления, делая его таким образом активным), вам нужно просмотреть справочную информацию об этом событии. Из нее вы узнаете, что данному событию соответствует делегирующий класс UlCuesEventHandler, которому нужна обрабатывающая событие функция-член с такой сигнатурой:

```
void Change_UICues(Object *sender, UICuesEventArgs *e);
```

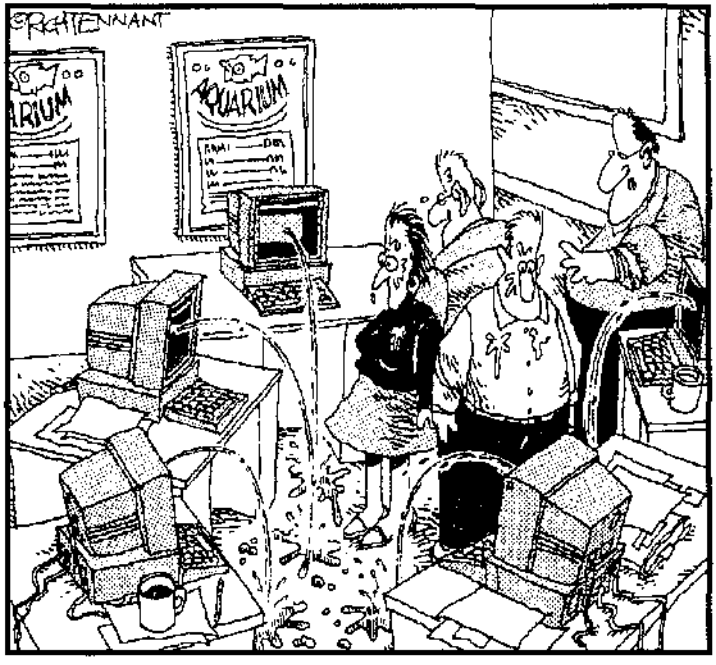
Добавление делегата для этого события будет выглядеть следующим образом:

```
poChange->ChangeUICues += new UICuesEventHandler(this,
        Change_UICues);
```

Почему для добавления делегата нужно использовать оператор +=? Дело в том, что .NET позволяет создавать для одного события множество процедур, которые это событие обрабатывают. Если вы используете просто оператор присвоения (=), одна процедура заменит собой все другие. Использование оператора += позволяет *добавлять* новые процедуры к уже существующим.

Часть IV

Горячие десятки



Ах да, извините, я забыл предупредить, что теперь у нас есть особая функция, которая издает специальный сигнал тревоги, как только нарушается связь с Web-узлом Aquarium.

В этой части...

Здесь вы найдете некоторые полезные сведения и советы, которые могут очень пригодиться при написании ваших собственных программ.

Первые главы этой части содержат советы, которые помогут вам справиться с некоторыми типичными (и весьма назойливыми) проблемами, возникающими в процессе создания программ. Помните, что рано или поздно все (и даже видавшие виды хакеры) сталкиваются с проблемами или совершают досадные ошибки. Здесь же вы узнаете о наиболее вероятных причинах возникновения проблем, что позволит вам более эффективно с ними бороться.

В последней главе приводится список десяти полезнейших функций, являющихся функциями-членами встроенных классов среды .NET. Зная, какой функциональностью обладают наиболее часто используемые классы среды .NET, вы без труда разберетесь в примерах программ, поставляемых компанией Microsoft вместе с установочным пакетом Visual C++ .NET, и сможете использовать их для создания собственных программ.

Глава 23

Десять синтаксических ошибок

В этой главе...

- Подключаемый файл не может быть найден
- Пропущена точка с запятой
- Не подключен заголовочный файл
- Не обновлено объявление класса
- Использование названия класса вместо названия объекта
- После объявления класса не поставлена точка с запятой
- В определении класса пропущено слово `public:`
- Неправильно набраны имена переменных
- Использование точки вместо стрелки (->) и наоборот
- Пропущена закрывающая фигурная скобка

Существуют тысячи причин возникновения синтаксических ошибок. В этой главе рассматриваются некоторые наиболее распространенные ошибки, сопровождающие их сообщения и методы их устранения.

Если вы не можете справиться с какой-то ошибкой и вам нужна дополнительная помощь, щелкните на сообщении об этой ошибке в окне Output и нажмите клавишу <F1>. Откроется окно справочной системы, и вы сможете получить развернутую информацию о том, что именно в программе работает неправильно и каким образом это может быть исправлено.

Подключаемый файл не может быть найден

Сообщение:

```
fatal error C1083: Cannot open include file: 'foo.h': No such file
or directory
```

Эта ошибка является одной из самых распространенных. Обычно ей предшествует целый набор сообщений о том, что тот или иной элемент не может быть определен.

Проверьте, расположен ли заголовочный файл, который должен быть подключен к вшей программе, в одной папке с исходными файлами. Если его там нет, то он обязательно должен быть размещен в одной из папок, содержащих подключаемые файлы. Чтобы проверить, какие именно папки содержат подключаемые файлы, выберите команду Tools⇨Options (Сервис⇨Параметры) и в открывшемся диалоговом окне в разделе Projects выберите категорию VC++ Directories. В списке Show Directories For (Показать папки для...) выберите пункт Include Files (Подключаемые файлы).

Кроме того, эта ошибки может быть вызвана тем, что, набирая название заголовочного файла, вы заключили его не в кавычки (" "). а в угловые скобки (< >). Используйте кавычки в тех случаях, если заголовочный файл размещен в той же папке, что и исходные файлы, поскольку те файлы, названия которых набраны в УГЛОВЫХ скобках, компилятор ищет только в папках, которые определены как содержащие подключаемые файлы.

Пропущена точка с запятой

Сообщении:

```
error C2628: 'bar' followed by 'int' is illegal (did you
                forget a ';'?)
error C2144: syntax error : 'int' should be preceded by ';'
error C2143: syntax error : missing ';' before '}'
```

Пропущенная точка с запятой является второй самой распространенной проблемой. Обычно в таких случаях появляются сообщения, показанные выше, однако могут быть сообщения и другого содержания.

По сути, эта ошибка возникает из-за того, что компилятор не знает, где остановиться, поэтому обычно в качестве источника возникновения ошибки будет указываться строка, следующая за той, в которой действительно пропущена точка с запятой. Решается эта проблема очень просто. Просмотрите коды программы, начиная со строки, на которую указывает компилятор, и выше, и определите, где не хватает точки с запятой.

Очень часто забывают набирать точку с запятой после фигурной скобки, которой заканчивается определение класса. (В этом случае сообщение об ошибке будет выглядеть примерно как `unexpected 'class' 'foo'`.) Если вам нужна дополнительная информация о правилах использования точек с запятой, обратитесь к главе 12.

Не подключен заголовочный файл

Сообщения:

```
error C2G65: 'a' : undeclared identifier
error C2065: 'sin' : undeclared identifier
error C2440: '=' : cannot convert from ''unknown-type'' to
                ''unknown-type''
```

Еще одна классическая ошибка — неподключенный заголовочный файл. Результатом будет появление самых различных сообщений, среди которых обычно есть сообщения о том, что класс, тип или функция не определены или что возвращаются данные не того типа.

Подобные проблемы очень часто возникают тогда, когда вы используете библиотечные функции, но забываете подключить соответствующие им заголовочные файлы. Например, если для отображения информации на экране вы используете функцию `cout`, не забудьте подключить файл iostream.h; если для вычисления квадратного корня вы используете функцию `sqrt`, не забудьте подключить файл math.h.

Просмотрите строку, на которую указывает компилятор. Если в ней используется библиотечная функция, проверьте, подключен ли соответствующий заголовочный файл. Если вы не уверены, какой именно файл должен быть подключен для того, чтобы можно было использовать эту функцию, обратитесь к справочной системе.

Эта проблема может касаться также созданных вами функций или классов. Если вы определили функцию, класс или переменную в одном файле, а хотите использовать в другом, вам придется создать и подключить заголовочный файл.

Не обновлено объявление класса

Сообщения:

```
error C2039: 'g' : is not a member of 'test5'
error LNK2001: unresolved external symbol "?bar@foo@ AEXXZ
(public: void___thiscall foo::bar(void))"
```

Очень часто программисты (особенно те, которые раньше программировали на языке C, а потом перешли на C++) забывают обновлять объявления классов. В результате появляются сообщения о том, что один элемент не является членом другого — baz is nor a member of foo (эти сообщения генерирует компилятор), или об использовании недопустимой внешней ссылки (unresolved external symbol — эти сообщения генерирует компоновщик).

Сообщение о недопустимой внешней ссылке на самом деле подразумевает возникновение проблем при использовании скорректированных имен. Скорректированные имена рассматривались в паве 7: компилятор создает для каждого элемента скорректированное имя с тем, чтобы все они были уникальны. Например, если вы используете имя foo : :bar (), компилятор преобразует его в ?bar@foo@ AEXXZ.

C++ очень строго относится к корректности использования типов данных. Если вы вносите изменения в список параметров, которые передаются функции-члену, не забудьте внести такие же изменения и в объявление класса. Если вы поместили объявление класса в заголовочный файл {что является довольно распространенной практикой), не забудьте обновить также и его. Также, если вы добавляете функцию-член к какому-то классу, не забудьте внести соответствующие дополнения в объявление этого класса.

использование названия класса вместо названия объекта

Сообщения:

```
warning C4832: token '.' is illegal after UDT 'error7'
error C2 275: 'error7' : illegal use of this type as an
          expression
error C2143: syntax error : missing ';' before '.'
```

Еще одна ошибка, которую обычно совершают программисты, которые ранее программировали на языке C, состоит в том, что при доступе к функции-члену или члену данных класса используется не название объекта, а название самого класса. В таких случаях обычно появляется сообщение о том, что использован неверный типа данных. Также может появиться сообщение о том, что проблемы вызваны неправильным использованием точки (.) или оператора ->.

Помните, что название класса и название объекта этого класса— совершенно разные имена. Предположим, например, что у вас есть такой код:

```
CDialog foo;
```

В данном случае foo является объектом класса CDialog. Если вам нужно вызвать функцию-член DoModal, наберите

```
foo.DoModal();
```

Но не набирайте

```
CDialog.DoModal();
```

После объявления класса не поставлена точка с запятой

Сообщения:

```
error C2146: syntax error : missing ';' before identifier 'foo'
fatal error C1004: unexpected end of file found
```

Если вы забудете поставить точку с запятой после объявления класса, это приведет к появлению множества сообщений об ошибках. Такая ошибка случается довольно часто, а потому ее можно назвать одной из самых распространенных. Более подробную информацию об этом вы можете найти в разделе "Пропущена точка с запятой".

В определении класса пропущено слово public:

Сообщения:

```
error C2248: 'error8a::a' : cannot access private member
             declared in class 'error8a'
```

Еще одна распространенная ошибка — не набранное слово `public:` при определении класса. При этом может появиться сообщение наподобие такого; `error C2248: 'bar' : cannot access private member declared in class 'foo'`.

По умолчанию все функции-члены и члены данных класса являются закрытыми. Именно поэтому, если вы не сделаете их открытыми, набрав перед их определением ключевое слово `public:`, вы получите сообщение, показанное выше.

Например, если вы наберете такой код, доступ к переменной `bar` будут иметь только функции-члены этого же класса:

```
class foo {
  int bar;
};
```

Поэтому такое обращение к переменной `bar` неправомочно:

```
foo salad;
salad.bar = 1;
```

Если же вы наберете код, приведенный ниже, переменную `bar` можно будет использовать везде, где применяются объекты класса `foo`, и к обращению `salad.bar` у компилятора не будет никаких претензий.

```
class foo {
public:
  int bar;
};
```

Неправильно набранные имена переменных

Сообщение:

```
error C2065: 'b' : undeclared identifier
```

Запутаться в именах переменных очень легко. При этом вы можете получить сообщение наподобие 'SongsNum' : undeclared identifier. Если вы пишете большую программу, то вполне можно забыть, как именно переменная была названа: SongsNum или NumSongs. Если ошибиться хотя бы на один символ, это тут же приведет к возникновению синтаксической ошибки. В таком случае вернитесь к объявлению переменной и уточните ее имя. Те же проблемы возникают и при неправильном наборе имени класса или функции.

использование точки вместо стрелки и наоборот

Сообщения:

```
error C281S: type 'error6' does not have an overloaded member
             'operator ->'
error C2227: left of '->f' must point to class/struct/union
error C2143: syntax error : missing ';' before '.'
```

Вы можете по ошибке использовать оператор -> вместо точки, если забудете, что имеете дело со ссылкой на класс, а не с указателем. Это происходит, если кто-то постоянно работает с указателями и в какой-то момент начинает к каждому элементу относиться как к указателю. Результатом будет появление приблизительно такого сообщения об ошибке: error C22 31: '.foo::bar' : left operand points to 'class', use '->'.

Например, такой код содержит в себе ошибку:

```
CDialog foo;
foo->DoModal();
```

Проблема в том. что foo является названием объекта класса CDialog, а не указателем на такой объект. Правильно будет использовать такой код:

```
foo.DoModal() ;
```

С другой стороны, проблемы возникают при использовании точки там, где в действительности нужно было использовать оператор ->. Это происходит тогда, когда с указателем обращаются так, как будто он является самим объектом. При этом сообщение об ошибке будет выглядеть приблизительно так: error C2227: left of '->bar' must point to class/struct/union. Например, следующий код неверен:

```
CDialcg *foo;
foo.DoModal();
```

В данном случае foo является ссылкой на объект класса CDialog, а не самим объектом. Чтобы вызвать функцию DoModal, нужно набрать такой код:

```
foo->DcModal();
```

Пропущена закрывающая фигурная скобка

Сообщение:

```
fatal error C1004: unexpected end of file found
```

Пропущенная закрывающая фигурная скобка — это более простительная ошибка, чем не поставленная точка с запятой. Обычно это случается при создании функций, содержащих в себе много вложенных операторов if или других блоков команд. Если количество открывающих скобок не соответствует количеству закрывающих, компилятор выдает сообщение о неисправимой ошибке (`fatal error`); в действительности это означает, что в поисках закрывающей фигурной скобки уже достигнут конец файла, а скобка так и не найдена.

В таком случае вам придется просмотреть все коды и проверить, все ли открывающие скобки имеют соответствующие им закрывающие. Для облегчения этой задачи можете воспользоваться дополнительными возможностями редактора кодов.

Глава 24

Вторая десятка синтаксических ошибок

В этой главе...

- Использование конструкторов, которые не требуют значений аргументов
- Незакрытые комментарии
- Несоответствие типов данных
- То, что работало в C, не работает в C++
- Использование ключевого слова `void`
- Неподключенные библиотеки функций
- Конструкторы для производных классов
- Точка с запятой в конце строки `#define`
- Отсутствие свободного места на диске
- Когда причина не известна

А вы подумали, что есть только десять наиболее распространенных ошибок? Конечно же, это не так! Если вы считаете себя программистом, то — как и любой другой программист — вы должны любить синтаксические ошибки. Когда я был помоложе, у нас было около двухсот наиболее распространенных синтаксических ошибок (и все любимые). С тех пор мало что изменилось.

Конструкторы, которые не требуют аргументов

Сообщение:

```
error C2228: left of '.getFrobozz' must have
      class/struct/union type
```

Допустим, у одного из ваших классов есть конструктор, не требующий для себя значений аргументов:

```
class Doohickey
{
public:
  Doohickey();
  int getFrobozz();
}
```

Если вы думаете, что новый объект этого класса может быть создан следующим кодом, то вы ошибаетесь:

```
Doohickey myDoohickey();
```

В действительности компилятор сразу не обнаружит ошибку. Но как только вы попытаетесь использовать новый объект, сразу же получите сообщение об ошибке. Дело в том, что компилятор интерпретирует приведенный выше код как объявление функции (их названия

ведь тоже заканчиваются парой круглых скобок), которая в качества результата возвращает объект класса Doohickey.

Чтобы компилятор понял вас правильно и создал объект, при использовании конструктора, не требующего значений аргументов, не набирайте пустую пару скобок;

```
Doohickey myDoohickey;
```

Незакрытые комментарии

Признак:

Появление сообщений о самых разных ошибках.

Эту ошибку могут допускать те программисты, которые ранее писали программы на языке С и, перейдя на С++, продолжают использовать старый формат комментариев (т.е. продолжают выделять комментарии символами /* и */). Поскольку такие комментарии могут занимать несколько строк, нет ничего удивительного в том, что иногда программисты могут забывать фиксировать их окончание символами */. Последствия такой ошибки могут быть самыми непредсказуемыми. Компилятор выдаст множество сообщений об ошибках, которые, впрочем, можно будет устранить простым закрытием комментария.

Если при наборе кодов в редакторе Visual C++ активизирована возможность выделения цветом отдельных синтаксических элементов (эта возможность включена по умолчанию), вы сразу обнаружите такую ошибку, поскольку при этом целые фрагменты кодов будут окрашены не в тот цвет. По умолчанию все комментарии отображаются зеленым цветом, следовательно, при незакрытом комментарии все последующие за ним строки также будут окрашены в зеленый цвет.

В этом случае вам придется просто просмотреть "зеленые" коды, найти место, где должен заканчиваться комментарий, и набрать там символы */.

Несоответствие типов данных

Сообщения:

```
error C2664: 'error11f' : cannot convert parameter 1 from
             'char *' to 'int'
warning C4244: '=' : conversion from 'float' to 'char',
             possible loss of data
```

Обычно такие ошибки случаются из-за невнимательности программистов. Если, например, вы объявляете переменную одного типа, а затем пытаетесь присвоить ей значение другого типа (которое к тому же не может быть преобразовано к объявленному типу), компилятор выдаст сообщения, подобные приведенным выше. Так, следующий код содержит в себе ошибку:

```
int i;
char* p = "Hello";
i = p;
```

Вернитесь к строке, на которую указывает компилятор, и проверьте правильность использования типов данных. Обычно такие ошибки возникают как следствие неправильного вызова функции или из-за того, что вы что-то перепутали.

Реже причиной возникновения ошибки несоответствия типов может быть использование устаревшего заголовочного файла. Так, очень велика вероятность появления ошибки при компиляции кодов, написанных на языке С, компилятором С++. Дело в том, что язык С довольно либерален в отношении использования типов данных, в то время как С++ скрупулезно следит за соответствием типов и фиксирует все (лаже потенциальные) проблемы.

Подобные ошибки будут возникать и при передаче функциям в качестве аргументов значений не тех типов, которые были для них объявлены. При этом возвращаемое сообщение об ошибке будет выглядеть приблизительно так: `error C2664: 'bar' : cannot convert parameter 1 from 'char*' to 'int'`. Вернитесь к кодам программы и проверьте, значения каких типов передаются функции при ее вызове. Для этого вначале просмотрите коды, где эта функция объявляется, а затем посмотрите, как она вызывается.

Жо, что работало в С, может не работать в C++

Признак:

Программа не может быть откомпилирована компилятором C++.

Если ваша программа была написана на языке С и прекрасно работала, но при попытке откомпилировать ее как программу C++ вдруг возникают какие-то ошибки, вероятнее всего, это происходит из-за несоответствия типов данных. В языке C++ правильность использования типов данных отслеживается намного строже, чем в С. В девяти случаях из десяти ошибка будет относиться к одной из тех, что описаны в разделе "Несоответствие типов данных".

Использование ключевогословаvoid

Сообщения:

```
error C2556: 'int error12::b(void)' : overloaded function differs
             only by return type from 'void error12::b(void)'
error C2371: 'error12::b' : redefinition; different basic types
```

Если функция-член объявлена как возвращающая результат типа void (пустой), но определена без использования слова void, это приведет к возникновению ошибки. Чаще такую ошибку допускают программисты, которые ранее писали программы на языке С. Сообщение об ошибке может выглядеть приблизительно так: `error C25S6: 'bar' : overloaded functions only differ by return type.`

Например, такой код содержит в себе ошибку:

```
class foo {
void Bummer();
};
foo::Bummer() {
}
```

Функция Bummer была объявлена как возвращающая результат типа void (т.е. не возвращающая никакого результата), но определена без указания типа (компилятор по умолчанию предположит, что она возвращает результат типа int). Таким образом, возвращение результата типа void — это не то же самое, что не указание никакого результата. Вот правильное определение функции Bummer:

```
void
foo::Bummer() {
}
```

Неподключенные библиотеки функций

Сообщение:

```
error C2871: 'System' : does not exist or is not a namespace
```

При написании программ .NET не забывайте использовать команду `#using` для подключения библиотек со стандартными функциями, которые вызываются вашей программой. В противном случае вы столкнетесь с многими ошибками, среди которых будут, например, такие, как невозможность обнаружения чего-то, что в действительности существует.

Конструкторы для производных классов

Сообщение:

```
error C2512: 'error13b' : no appropriate default constructor
               available
```

Ошибка может появиться в том случае, если вы не создадите открытый конструктор для производного класса или явно не вызовете конструктор базового класса. Такие ошибки случаются редко, но, столкнувшись с ними, вы можете не сразу понять, в чем дело. Подобные проблемы возникают только при использовании возможности наследования для создания производных классов. Суть проблемы в том, что вызываемый конструктор для создания объекта производного класса не может быть найден. Возможная причина: для производного класса не был определен конструктор, вызываемый по умолчанию. Или, возможно, все конструкторы производного класса требуют передачи им значений аргументов и ни один из них не вызывается явно.

Вернитесь к главе 18, чтобы получить более детальную информацию об инициализации конструкторов и создании конструкторов, используемых по умолчанию.

Точка с запятой в конце строки #define

Признак:

Возникновение самых разных ошибок.

Если в конце строки `#define` вы поставите точку с запятой, это приведет к появлению самых непредсказуемых проблем. Это одна из наиболее неприятных ошибок, так как компилятор при этом будет указывать на ошибки в строках, набранных где-то посреди кодов программы, хотя сами эти строки в действительности будут вполне корректны. Если где-то среди строк, на которые указывает компилятор, используется макрос (вообще говоря, это плохой инструмент), вполне возможно, что проблема именно в нем. Проверьте, правильно ли вы набрали коды, которыми этот макрос определяется, и поставили ли вы в конце этих кодов точку с запятой.

В целом использовать команду `#define` для создания макроса — плохая идея, поскольку, как уже только что было сказано, сами макросы — плохие инструменты. Именно поэтому в данной книге они не рассматриваются.

Отсутствие свободного места на диске

Сообщения:

```
fatal error C1088: Cannot flush precompiled header file:
'WinDebug/NOHANDS.pch': No space left on device
fatal error C1088 : Cannot flush compiler
intermediate file: '': No space left on device
fatal error C1033: cannot open program database
          'd:\project\debug\vc70.pdb'
error LNK1104: cannot open file "FOO.exe"
```

Довольно редко случается так, что оставшегося на диске свободного пространства становится недостаточно для решения каких-то задач, но, когда это происходит, система начинает генерировать сообщения об ошибках. Если вы получаете сообщения, подобные приведенным выше, убедитесь, что на диске по-прежнему достаточно свободного места. Если не достаточно, удалите ненужные файлы, чтобы освободить немного памяти, Или зайдите в ближайший компьютерный магазин и купите новый винчестер гигабайт этак на сорок — цены на них сейчас вполне доступные.

Жак в чем же проблема

Признак:

Сообщения об ошибках указывают на корректно набранные строки.

Если будут повреждены некоторые ключевые файлы Visual C++ .NET, это может привести к появлению необъяснимых синтаксических ошибок. Например, может случиться так, что программа работала правильно, но, когда вы вернулись к ней на следующий день, она почему-то уже не компилируется. Или какая-то вполне безобидная и простая программа вдруг выдает ошибку с сообщением наподобие `missing ; in stdio.h`. Хотя вы точно знаете, что эта программа не имеет никакого отношения к файлу stdio.h (или к windows.h, или к любому другому файлу, который не является частью ваших исходных кодов).

Обычно реальная причина появления таких ошибок заключается в том, что были повреждены некоторые информационные файлы Visual C++. В этом случае откройте папку со своими файлами и удалите все файлы с расширением .PCH. Если это не поможет, удалите также файл с расширением .SUO (это скрытый файл), файл .VCPROJ или файл *.SLN. После этого проект должен быть создан (откомпилирован) заново, но зато теперь программа почти наверняка будет работать правильно. Можете также попробовать заново построить все исходные файлы: по одному за раз, начиная с второстепенных файлов и заканчивая тем, в котором содержится функция `main`.

Глава 25

Десять функций .NET

В этой главе...

- WriteLine
- ReadLine
- Parse
- Run
- DrawLine
- FromArgb
- DrawString
- FromFile
- OnMouseMove
- Add

В этой главе вы можете найти краткое описание десяти наиболее важных функций .NET, к помощи которых вы постоянно будете прибегать при написании программ. Функции, названия которых показаны в формате класс::функция, обычно вызываются непосредственно путем набора их имени, в то время как те из них, которые показаны как класс->функция, вызываются через указатели на объекты соответствующих классов.

Console::WriteLine

Функция Console::WriteLine используется для отображения информации на экране.

```
Console::WriteLine(S"Введите значение координаты X");
```

Console::ReadLine

Функция Console::ReadLine используется для считывания информации, поступающей от пользователя. Обратите внимание, что, если значение должно быть преобразовано к числовом; виду, необходимо также вызвать одну из функций Parse.

```
m_nXFron = Int32::Parse(Console::ReadLine());
```

Int32::Parse

Функция Int32::Prase преобразует строки к значениям типа integer. Используется в основном в процессе обработки данных, поступающих от пользователя.

```
m_nXFron = Int32::Parse(Console::ReadLine());
```

Application: :Run

Форма, принимаемая функцией Application::Run **в качестве** аргумента, **становится главной (main-формой) для данного приложения.**

```
#ifdef _UNICODE
int wmain(void)
#else
int main(void)
#endif
{
  Application::Run(new MyForm());
  return 0;
```

Graphics->DrawLine

Функция Graphics->DrawLine **рисует линию на экране.**

```
Void LineObject::Draw(Graphics *poG)
{
   //Создание пера для рисования
   Pen *poPen = new Pen(Color::Red);
   //Рисование линии
   poG->DrawLine(poPen, m_nXFrom, m_nYFrom, m_nXTo, m_nYTo);
}
```

Color::FromArgb

Функция Color::FromArgb определяет цвет объекта по набору передаваемых ей значений.

```
Void LineObject::Draw(Graphics *poG)
{
   //Создание пера для рисования
   Pen *poPen = new Pen(Color::FromArgb( 128, 0, 128));
   //Рисование линии
   poG->DrawLine(poPen, m_nXFrom, m_nYFrom, n_nXTo, m_nYTo);
}
```

Graphics->DrawString

Функция Graphics->DrawString **отображает текст в окне, открытом для рисования.**

```
Void Factorial::UpdateDisplay(Graphics *g)
{
   Brush *poBrush = new SolidBrush(Color::Black);
   g->Clear(Color::AntiqueWhite);
   g->DrawString(fltResult.ToString(), DefaultFont, poBrush,
                 0, 0);
}
```

Image::FromFile

Функция Image::FromFile загружает изображение, сохраненное на диске.

```
PoImg = new PictureBox( > ;

//Настройка параметров изображения
poImg->Width = 160;
poImg->Height = 120;
poImg->Image = Image :: FromFile("..\\desert4.jpg");
poImg->SizeMode = PictureBoxSizeMode::StetchImage;
```

Form::OnMouseMove

Функция Form::OnMouseMove используется для отслеживания перемещения курсора мыши в окне формы.

```
Void MyForm::OnMouseMove(MouseEventArgs *e)
{
   //Отображение координаты x курсора мыши
   poXPosition->Text = e->X.ToString();
}
```

Controls->Add

Функция Controls->Add используется для добавления в окно формы элементов управления.

```
This->Controls->Add(poOK);
This->Controls->Add(poTE);
```

Предметный указатель

A

ArrayList, 159; 193; 225; 231

C

CLR, 23; 27; 40; 101

D

DataTips, 177

G

GD1+. 141
GPF, 150
GUI, 255

N

n-факториал. 110

S

Stack, 161; 193

А

Аргументы, 118; 119

Б

Библиотеки, 21; 28
  iostream, 45; 249
  math, 103
  mscorlib, 40
  RLT, 21
  динамические, 28
  статические, 28
Битовые поля, 98

В

Вложение операторов, 109
Вложенные структуры, 86
Внешнее имя, 74
Вызов функции, 118
Выполняемые коды, 20
Выражение, 26
Выражения, 35; 89

Г

Гексит, 100
Глобальные переменные, 122; 165

Д

Делегаты, 260
Делегирование, 259
Деструктор, 209; 212
Динамическое выделение памяти, 136
Директивы препроцессора, 45

З

Зависимости, 50
Значения переменных, 130

И

Имена указателей, 133
Имя переменной, 81
Индекс, 155
Инициализация массивов, 157
Инициализация переменных, 83
Инкапсуляция, 31; 32
Интегрированная среда разработки, 21
Исключения. 243
Истина, 93
Исходные коды, 20
Итерация. 109

К

Класс, 189
Классы, 31; 85
  Exception, 243
  Stream Reader. 249
  StreamWriter. 249
  Windows Forms, 255
  абстрактный, 231
  базовые. 32
  базовый, 220
  внешний интерфейс, 192

дочерние, 32; 220
потоковые, 249
производные, 32; 220
родительские, 32
родительский, 220
Коды
неуправляемые, 23
управляемые, 23
Комментарии, 27; 40
Компилятор, 20; 43
Компиляция, 65
Композиция. 31
Компоновщик. 43
Константы, 77
Конструктор, 209
Корректировка имен, 74

Л

Логические выражения. 92
Логические ошибки, 172
Ложь, 93
Локальные переменные, 165

М

Массив, 155
Машинные коды, 65
Методы классов, 32
Механизм обработки исключений, 241
Многомерные массивы, 158
Многооконный интерфейс MDI, 55

Н

Надстройки, 45
Наследование, 32
Наследование кодов, 219
Неизменяемый ссылочный аргумент, 150
Нумератор, 161

О

Обеспечение типовой безопасности, 75
Область видимости переменных, 165
Обработка исключений, 237
Общее нарушение защиты, 150
Объектно-ориентированное программирование, 31
Объекты. 31; 193
Объявление
классов, 88; 191
конструкторов, 210
массивов. 158
переменных, 72
структур, 85
указателей, 133
Окно
Add Existing item, 52
Autos. 175
Find, 62
Locals, 175
New File, 56
New Project, 50
Options, 173; 175
Output, 37
QuickWatch, 177
Replace. 62
Solutin Explorer, 37
Solution Explorer, 50; 53
Watch, 173; 176
консоли, 39
отладчика, 173
редактора кодов, 53; 55
Оператор
switch, 113
квалифицирующий, 192
присвоения, 94
равенства, 95
условный, 98
Операторы, 89
логические, 93
присвоения, 96
сдвига, 91
условные, 26
Определение
массивов, 156
многомерных массивов, 159
переменных, 72; 83
структур, 86
функции, 118
функций-членов, 192
Отладчик, 21; 171

Ошибки
логические, 20
синтаксические, 20

П

Панель перехода, 60
Параметры
командной строки, 48
Параметры функции. 119
Перегрузка, 221
Переменные. 27; 81; 130
Перечислимый тип, 162
Подпрограммы, 117
Подстановочные знаки, 63
Полиморфизм, 32; 33; 194; 226
Поток, 25 1
Потоковые манипуляторы, 253
Потоковые переменные. 25 1
Пошаговое выполнение, 172; 174
Предупреждения, 67
Приоритет операторов, 99
Программы
выполнение, 25
выполняемые, 65
компиляция, 25; 68
написание. 25
неуправляемые. 28; 35; 42
объектно-ориентированные, 31
отладка, 25
повторное построение, 68
построение, 68
проектирование. 25
процесс отладки, 172
управляемые, 35
Проекты, 49
Произвольный доступ к данным. 155
Просмотр текущих значений, 173
Пространство имен, 40
Процедуры, 117

Р

Разыменовывание, 131; 134
Регулярные выражения, 62
Регулярные выражения GREP, 62
Редактор кодов, 55
Рекурсия, 124
Решения. 49

С

Свойства классов, 32
Свертывание кодов, 61
Связанный список. 132; 144
Сигнатура функции, 260
Символы
новой строки, 44
специальные, 44
Синтаксические ошибки, 66; 171; 265
Система счисления
двоичная, 100
десятичная, 100
шестнадцатеричная, 100
События, 259
Справочная система, 22
Среда .NET, 23
Ссылочные аргументы, 149
Строки, 42; 78; 152
Unicode. 42
с завершающим нулем, 152
Строковый компилятор, 20
Структурная проверка, 180
Структуры, 85

Т

Таблица имен, 74
Типы данных, 72; 75
Точки останова. 172

У

Указатели, 130
Условия ошибки, 237
Условный оператор, 106
Установочный пакет Visual C++, 19
Утилиты, 22
NMAKE, 22
Spy++, 22
WINDIFF, 22
ZOOMIN, 22

ф

Файлы
iostream.h, 45
заголовочные, 42; 206
исходные. 35
потоковые, 249
проекта, 49; 50
реализации. 35
ресурсов, 54
формирующие, 49
Форматирование кодов, 108
Формы, 257
Фрагменты кодов, 61
Функции, 117
cin, 45
Console, 39
cout, 44
Graphics, 137
main, 26; 38
библиотечные, 45; 117
виртуальные. 226
глобальные, 117
доступа, 203
математические, 101
перегрузка, 34
Функции-члены, 117; 190
исключительно виртуальные, 23 i
Функция, 26; 38

Ц

Цикл
for, 109
while, 112
бесконечный. 112

Ч

Члены данных, 190
закрытые, 191
открытые, 191

Э

Экземпляры классов. 193
Элементы управления, 258

Я

Языки машинного уровня, 74
Языки программирования
нестрогие, 71
строгие, 71

Научно-популярное издание

Майкл Хаймен, Боб Арнсон

Visual C++ .NET для "чайников"

В издании использованы карикатуры американского художника Рича Теннанта

Литературный редактор *Т.П. Кайгородова*
Верстка *О.В. Мишутина*
Художественный редактор *Е.П. Дынник*
Корректоры *Л.А. Гордиенко, О.В. Мишутина*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 288.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 08.07.02. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 18,83. Уч.-изд. л. 14,60.
Тираж 5000 экз. Заказ № 656.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых
коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

...FOR Dummies™

COMPUTER BOOK SERIES

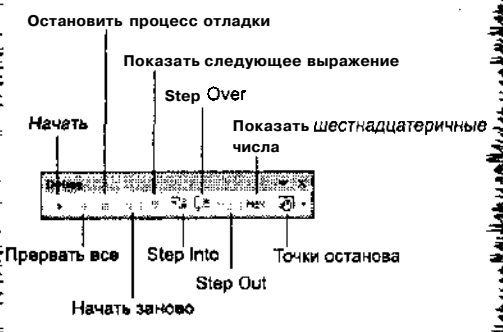
Visual C++ .NET для "чайников"

...ДЛЯ "чайников"™

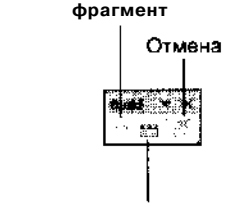
СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Панель Debug



Панель Build



Команды сопоставления регулярных выражений

.	Заменяет собой один из символов
*	Заменяет любое количество произвольных символов
+	Заменяет любое количество символов, предшествующих (по алфавиту) символу, после которого эта команда набрана
^	Ищет соответствие по началу строки
$	Ищет соответствие по концу строки
[]	Заменяет один из символов, набранных внутри скобок
[^]	Заменяет любой символ, кроме набранных **внутри** скобок
[-]	Заменяет любую букву из диапазона, указанного в скобках

Горячие клавиши

Одновременный сдвиг нескольких строк	Выделите несколько строк и нажмите клавишу <Tab>
Отмена сдвига нескольких строк	Выделите строки и нажмите <Shift-Tab>
Переход ко второй скобке из пары скобок	Нажмите <Ctrl-]>, чтобы перейти от открывающей скобки (, {, < или [к соответствующей ей закрывающей скобке:), }, > или]
Использование закладок	Поместите курсор в том месте, где вы хотите оставить закладку, и дважды нажмите <Ctrl+K>
Переход к закладке	Нажмите <Ctrl+K> и <Ctrl-N>, чтобы перейти к следующей закладке в файле. Нажмите <Ctrl+K> и <Ctrl-P>, чтобы перейти к предыдущей закладке
Переключение между окнами редактора	Нажмите <Ctrl+Tab> или <Ctrl-F6>, чтобы перейти к следующему окну редактора (которое было открыто после текущего). Нажмите <Shift-Ctrl+Tab> или <Shift+Ctrl-F6>, чтобы перейти к предыдущему окну редактора (которое было открыто перед текущим)
Построение решения	<Ctrl-Shift+B>
Компиляция текущего файла	<Ctrl-F7>
Отладка программы	<F5>
Прекращение процесса отладки	<Shitf+F5>
Запуск программы без отладки	<Ctrl+F5>
Установка или удаление точки останова	<Ctrl-B>
Step over	<F10>
Step into	<F11>
Step out	<Shift+F11>
Получение справочной информации	<F1>
Переход к следующему окну редактора кодов	<Ctrl-Tab> или <Ctrl+F6>
Закрытие текущего окна	<Ctrl+F4>

COMPUTER BOOK SERIES

Visual C++ .NET для "чайников"

СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Создание управляемой программы C++

1. Выберите команду File⇨New⇨Project.
2. В разделе Project Types выберите Visual C++ Projects.
3. В разделе Templates выберите Managed C++ Application.
4. В поле Name укажите название вашей программы. В папке Visual Studio Projects будет создана новая папка с тем же названием, которое вы дали своей программе, Если вы хотите, чтобы папка была создана в каком-либо другом месте, наберите ее имя в поле Location или щелкните на кнопке Browse и выберите нужную папку.
5. Щелкните на кнопке OK.

Типы данных

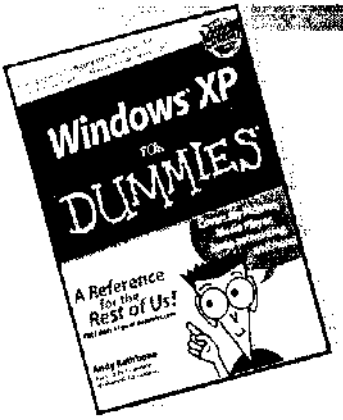
тип	Описание
double	Число с плавающей запятой
int	Целое число
string	Строка, Например, "Hello World"
char	Символ
float	Число с плавающей запятой. Диапазон принимаемых значений значительно меньше, чем для типа double
long	Длинное целое (32 бита)
short	Короткое целое (16 бит)
unsigned	Это слово может быть добавлено перед словом int, long или short, чтобы обозначить, что все принимаемые значения будут только позитивными

Создание неуправляемой программы C++

1. Выберите команду File⇨New⇨Project.
2. В разделе Project Types выберите Visual C++ Projects.
3. В разделе Templates выберите Win32 Project.
4. В поле Name укажите название вашей программы.
5. Щелкните на кнопке OK.
   Откроется окно мастера Win32 Application Wizard.
6. Щелкните на вкладке Application Settings.
   Мастер отобразит настройки, которые можно установить для создаваемой программы.
7. Щелкните на кнопке Finish.

Hungry Minds™

Энди Ратбон



Windows XP для "чайников"

В продаже

Перед вами замечательная книга о самой последней редакции операционной системы от компании Microsoft, предназначенной для домашних пользователей, — Windows XP. Написанная известным автором Энди Ратбоном. эта книга поможет вам сделать первые шаги в освоении новой операционной системы. Из материала книги вы узнаете, что представляет собой Windows XP и на что она способна. Здесь вы встретите описания компонентов Windows XP среди которых Проигрыватель Windows Media, Internet Explorer 6.0, Outlook Express 6.0, Мастер новых подключений и др.

Книга рассчитана на пользователей с различным уровнем подготовки. Легкий и доступный стиль изложения поможет даже начинающим быстро освоить Windows XP.