ГАЛИНА ДОВБУШ
АНАТОЛИЙ ХОМОНЕНКО

Visual C++

НА ПРИМЕРАХ

СОЗДАНИЕ ПРИЛОЖЕНИЙ В СРЕДЕ VISUAL C++

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА C++

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ОРГАНИЗАЦИЯ ВВОДА-ВЫВОДА И ОБРАБОТКА ИСКЛЮЧЕНИЙ

СОЗДАНИЕ ПРИЛОЖЕНИЙ API WINDOWS И MFC

+CD

bhv®

Галина Довбуш

Анатолий Хомоненко

Visual C++

НА ПРИМЕРАХ

Под редакцией профессора Хомоненко А. Д.

Санкт-Петербург

«БХВ-Петербург»

2007

УДК 681.3.068+800.92VisualC++
ББК 32.973.26-018.1
Д58

Довбуш, Г. Ф.

Д58 Visual C++ на примерах / Г. Ф. Довбуш, А. Д. Хомоненко / Под ред. проф. А. Д. Хомоненко. — СПб.: БХВ-Петербург, 2007. — 528 с.: ил.

ISBN 978-5-94157-918-1

Рассмотрены интерфейс системы программирования Visual C++, техника создания и отладки проектов приложений в среде Visual Studio 2005. Описаны основы языка C++: типы данных и операции, приемы программирования разветвлений и циклов, техника работы со статическими и динамическими массивами, использование функций. Рассмотрены классы и объекты, механизм множественного и одиночного наследования, перегрузка операторов и шаблоны классов, понятия ввода-вывода данных и классификация, принципы работы с потоками и файлами, стандартные классы потоков, форматированный ввод-вывод базовых типов, дополнительные возможности ввода-вывода. Освещена обработка исключений. Показаны особенности создания приложений API Windows и MFC. Представлены внутренняя их организация, создание диалоговых окон и меню, механизм обработки сообщений, работа с картой сообщений. Приводятся многочисленные примеры отлаженных программ. На компакт-диске содержатся тексты листингов примеров программ, приведенных в книге.

Для начинающих программистов

УДК 681.3.068+800.92VisualC++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.09.07.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 42,57.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-94157-918-1

Оглавление

Предисловие

Система программирования Microsoft Visual C++ (MVC++) входит в состав Visual Studio и является популярным инструментом, широко используемым для разработки различного рода приложений, выполняемых под управлением Windows, и обеспечивающим высокую эффективность их кода. Это обусловлено (в частности) тем, что MVC++ разрабатывалась фирмой Microsoft — создателем семейства операционных систем Windows.

По языку C++ и системе программирования MVC++ имеется большое множество учебной литературы. Предлагаемая книга также ориентирована на использование в учебном процессе. При ее подготовке преследовалась цель последовательно и систематично изложить приемы программирования в среде MVC++, приводя сравнительно небольшой объем теоретических сведений и продуманный набор содержательных примеров. Все приводимые примеры отлажены в среде MVC++ из состава Visual Studio 2005.

В системе программирования MVC++ можно создавать достаточно большое число различных видов приложений. Мы рассматриваем три основные разновидности приложений: консольные приложения, приложения API Windows и приложения MFC.

Консольные приложения в книге используются для рассмотрения основных приемов программирования на языке C++ при создании приложений, в которых обмен информацией ведется в основном с помощью клавиатуры и экрана.

Приложения API Windows позволяют обеспечить развитый графический интерфейс, основанный на использовании соответствующих функций и диалоговых окон.

Приложения MFC также позволяют обеспечить развитый графический интерфейс, основанный на использовании диалоговых окон. Вместо функций API здесь используются компоненты библиотеки MFC, при этом повышается удобство разработки приложений.

Книга состоит из шести частей.

В *части I* описываются среда программирования MVC++, вопросы создания, открытия и активизации проекта приложения в новой или существующей рабочей области. Освещаются этапы подготовки программы к исполнению: трансляция, компоновка и отладка. Рассматривается технология создания консольного приложения, описывается функция `main ()`, и затрагиваются вопросы вывода текста на экран.

В *части II* описываются простые типы данных, ввод данных с клавиатуры и вывод данных на экран, операции над операндами простых типов. Рассматриваются операторы: составные, условные, цикла и передачи управления. Описываются массивы (одномерные, многомерные, символьные) и операции над ними, операции с указателями, использование указателей при работе с обычными и динамическими массивами, действия с массивами указателей. Освещаются структуры: операции доступа к элементам структуры, инициализация структур и действия с массивами структур. Рассматриваются функции и их использование: прототип, определение, возвращаемое значение, вызов, область видимости, включение функции в проект приложения. Описываются способы передачи параметров функции, в том числе массивов, функции обработки символов и строк, перегрузка функций и шаблонные функции.

Часть III рассматривает объекты и классы: спецификаторы доступа к объектам класса, объявление класса и объекта класса, доступ к членам объектов, конструкторы и деструктор. Описан указатель `this` на объект класса, данные и методы класса, а также стандартный класс `string` для строкового типа данных. Освещаются композиция, одиночное и множественное наследование, чистые виртуальные функции и абстрактные классы. Описывается перегрузка операторов с помощью операторных функций-членов класса и функций-друзей класса, а также перегрузка операторов в производных классах. Рассматриваются шаблоны классов: объявление шаблона класса и объектов шаблона класса, параметры по умолчанию, наследование при работе с шаблонами.

В *части IV* рассматриваются основы ввода-вывода: классификация способов, принципы работы с потоками и файлами, стандартные классы потоков, форматированный ввод-вывод базовых типов. Освещаются дополнительные возможности ввода-вывода: форматированный ввод-вывод пользовательских типов, файловый ввод-вывод на верхнем уровне, неформатированный ввод-вывод, использование библиотеки `stdio`. Описывается технология обработки исключительных ситуаций.

В части V дается характеристика приложений API Windows: графический интерфейс приложений, контекст устройства, состав приложения, функ-

ция `WinMain()`, оконная процедура обработки сообщений, шаги создания приложения. Описывается технология разработки интерфейса приложения API Windows: создание меню и диалогового окна, использование элементов управления, задание оконных процедур.

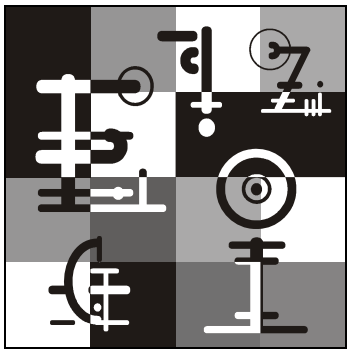
В части VI дается характеристика приложений MFC: библиотека MFC, графический интерфейс. Рассматриваются этапы создания, типы и состав приложений MFC. Описывается организация обработки сообщений: карта и макросы обработки сообщений, типы передаваемых сообщений. Рассматривается технология разработки интерфейса приложения MFC: общая характеристика интерфейса приложения, создание диалогового окна и класса окна, доступ к элементам управления окна. Освещается ввод-вывод с помощью класса `CFile`.

Прилагаемый компакт-диск содержит тексты листингов для примеров программ, приводимых в книге.

Подготовка книги основана на опыте преподавания ряда дисциплин, связанных с изучением приемов и технологий программирования на языке C++ в среде Visual Studio, преподавателями кафедры информационных и вычислительных систем Санкт-Петербургского государственного университета путей сообщений и кафедры математического обеспечения военно-космической академии имени А. Ф. Можайского.

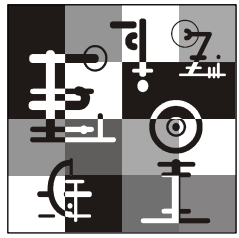
Выражаем признательность Бобровскому А. И., участвовавшему в подготовке материалов четвертой части книги. Благодарим также Сухоногова А. М. и Темплинга А. А. за подготовку материалов с примером консольного приложения (в приложении к книге).

Авторы



Часть I

Простейшая программа на языке C++



Глава 1

Подготовка программы к исполнению

Подготовка прикладных программ к исполнению на компьютере включает в себя три этапа.

Первый из них заключается в создании и редактировании файлов исходной программы, написанной на языке программирования. Как правило, программа на языке C++ содержит множество файлов, которые делятся на файлы спецификации и файлы реализации. Файл спецификации, или заголовочный файл для C++ имеет расширение `.h` и содержит описание используемых в программе типов данных, а также прототипов функций. Файл спецификации включается при помощи специальной директивы препроцессора `include` в соответствующий текст файла реализации, который содержит инструкции языка для выполнения тех или иных действий. Файл реализации имеет расширение `.cpp`.

На втором этапе из исходных файлов формируются объектные программы, то есть программы в машинных кодах, полученные после компиляции исходных файлов. Объектные файлы имеют расширение `.obj`. Каждый исходный файл реализации обрабатывается отдельно. В процессе компиляции в исходной программе могут быть обнаружены синтаксические ошибки, которые следует исправить, иначе объектная программа не будет построена. Если компиляция не выявила ошибок, то можно переходить к следующему этапу.

Третий этап состоит в построении исполняемого файла, имеющего расширение `.exe`. На этом этапе все объектные программы и функции из системной библиотеки объединяются в единое целое, и происходит компоновка программы. В процессе компоновки так же, как и в процессе компиляции, могут возникнуть ошибки. В случае появления таких ошибок их тоже следует исправить, после чего повторить компиляцию исправленных исходных файлов и выполнить компоновку.

После третьего этапа программа готова к исполнению. Сначала необходимо проверить работоспособность программы на заранее подготовленных контрольных (тестовых) примерах. Если в результате проверки будет получен хотя бы один несхожий с ожиданиями результат, следует найти место ошибки исполнения в исходной программе, а затем повторить заново все этапы.

Схема процесса подготовки программы к исполнению показан на рис. 1.1.

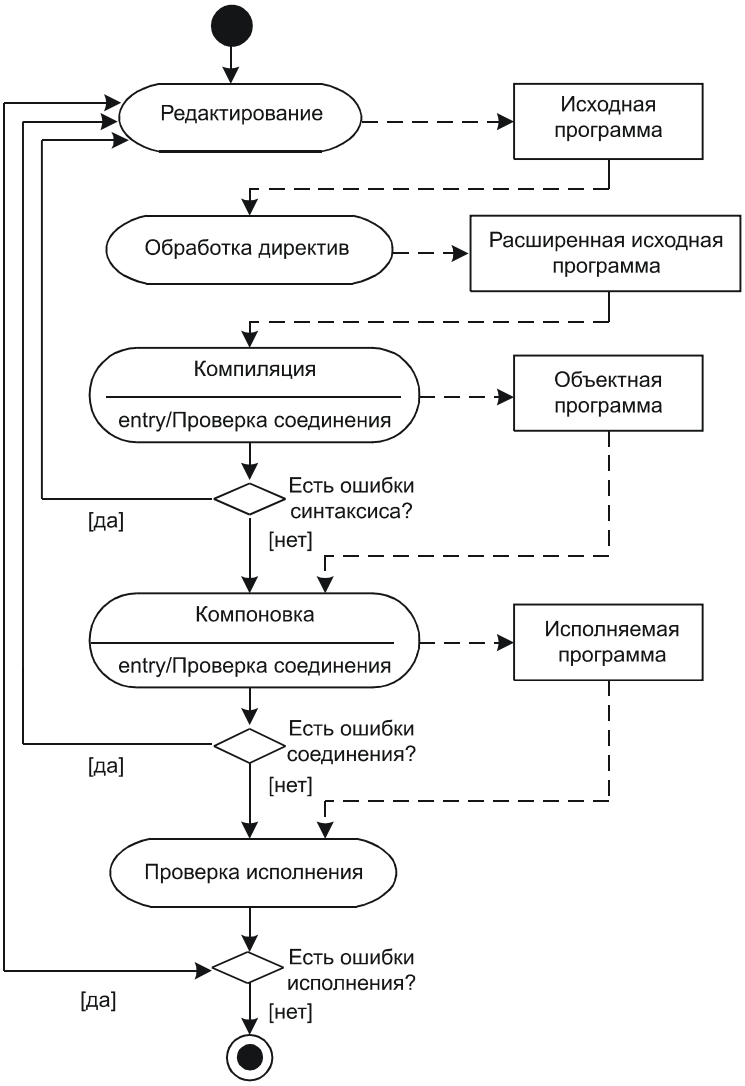
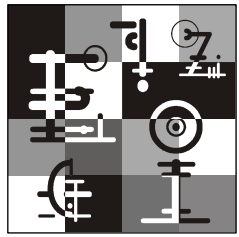


Рис. 1.1. Схема процесса подготовки программы к исполнению



Глава 2

Среда программирования

Среда программирования (program environment) образуется необходимыми для создания программ средствами, к которым относятся:

- редактор;
- препроцессор;
- компилятор;
- компоновщик;
- отладчик.

Редактор (editor) — программа, используемая для написания и изменения исходных программ или данных.

Основные функции редактора среды программирования MVC++:

- ввод и сохранение на диске файлов с текстами исходных программ;
- выделение фрагментов текста, их копирование и перенос;
- синтаксическая раскраска текста программы. По умолчанию редактор MVC++ раскрашивает текст программы в черный цвет с комментариями зеленого цвета и ключевыми словами (служебными словами языка C++) голубого цвета. Синтаксическая раскраска помогает замечать ошибки, допущенные при наборе текстов программ;
- форматирование текста программы. Редактор автоматически расставляет отступы при наборе операторов. Эта функция редактора, так же как и предыдущая, позволяет избежать многих ошибок.

Препроцессор (preprocessor) — программа, которая выполняет предварительную обработку исходной программы для компилятора. Перед тем как попасть на вход компилятора, исходная программа проходит через препроцессор,

работа которого регламентируется директивами (preprocessor directive) и операторами (preprocessor operator).

Компилятор (compiler) — средство для перевода (трансляции) исходной программы, написанной на языке программирования, в совокупность понятных компьютеру машинных команд. Компилятор может обнаружить ошибки трансляции (compile errors), а значит, он не сможет создать объектную программу. Как правило, это синтаксические ошибки (syntax errors). Компилятор также может сообщать предупреждения (warnings), которые необходимо обработать как ошибки. Все ошибки в исходной программе следует исправить в редакторе, после чего повторить ее трансляцию. Для решения одной задачи создается несколько исходных программ, каждая из которых проходит трансляцию. Исходные программы используют различные функции, размещенные в специальных библиотечных файлах (например, в библиотеке стандартного потока ввода/вывода). Компилятором из исходных программ генерируются объектные, требующие соединения для решения поставленной задачи. Точно так же в соединении с объектными программами нуждаются и используемые библиотечные функции.

Компоновщик (linker) — средство связывания объектных программ с кодами функций из стандартных библиотек, обеспечивающее сборку (компоновку) исполняемой программы. Во время связывания объектных файлов могут возникать ошибки компоновки (linker errors). Эти ошибки показывают, что указанные в программе внешние ссылки не могут быть найдены компоновщиком. В таком случае следует уточнить ссылки в исходных программах, а затем повторить трансляцию и компоновку снова.

Результаты работы исполняемой программы проверяются на тестовых примерах. Тестовые примеры (test cases) — контрольные наборы тестовых данных для проверки правильности функционирования исполняемой программы. Тестовые примеры подготавливаются заранее. Для каждого примера записываются контрольные значения необходимых исходных данных, а также записываются ожидаемые выходные данные (результаты работы исполняемой программы). Тестовые примеры должны учитывать следующие возможные варианты для входных данных:

- значения данных принадлежат допустимому диапазону изменения;
- значения данных находятся на границе диапазона изменения;
- значения данных запрещены, то есть находятся вне разрешенного диапазона изменения.

Во время проверки исполняемой программы, могут возникать ошибки исполнения (run-time errors). Если программа делает что-то запрещенное, она

тут же завершается. Однако ошибка не всегда приводит к остановке программы, и только часть результатов может оказаться неверной. Такая форма ошибки исполнения является логической ошибкой (logical error). Как правило, появление логических ошибок происходит вследствие неправильного понимания задачи или допущенных на этапе разработки программы просчетов. Если хотя бы один тестовый пример будет выполнен некорректно, то следует найти ошибку, исправить исходную программу, откомпилировать ее, создать новую исполняемую программу и повторить тестовые примеры.

Для поиска ошибок исполнения проводится отладка, являющаяся существенной частью процесса программирования. Отладка (debugging) позволяет обнаружить, локализовать и устранить ошибки в программе. Ключевым понятием при отладке программ является точка останова.

Точка останова (breakpoint) — это место в исходной программе, где следует остановиться в процессе выполнения программы. В точке останова происходит прекращение исполнения программы перед выполнением отмеченного точкой останова оператора, после чего можно завершить выполнение программы или работать дальше, анализируя передачу управления в программе и изменение значений переменных.

Для устранения ошибок исполнения необходимо проверить алгоритм решения задачи и воспользоваться пошаговой отладкой программы с помощью отладчика.

Отладчик (debugger) — средство, предназначенное для анализа поведения исполняемой программы, обеспечивающее ее пошаговое выполнение (трассировку).

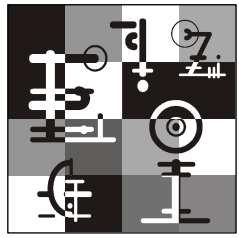
Основные функции отладчика среды программирования MVC++:

- установка точек останова для прерывания выполнения программы в указанных местах с целью последующей трассировки. Установленная точка останова отмечается красной точкой на левой границе окна в строке текста программы, содержащей оператор. После остановки исполнения программы можно добавить или убрать точки останова;
- трассировка с самого начала или с любого места программы. Желтая стрелка на левой границе текста программы показывает оператор, который будет выполняться. При трассировке программы желтая стрелка движется, чтобы показать, какой оператор будет выполняться;
- пошаговое выполнение программы с трассировкой вызываемых ею функций или без их трассировки;
- просмотр значений переменных. В специальном окне отладчика отображаются значения используемых программой в данный момент времени

переменных. Можно проанализировать переменные и продолжить или прервать выполнение;

- изменение значений переменных в процессе отладки. В специальном окне отладчика можно установить новые (другие) значения для переменных, чтобы понять, как изменится процесс выполнения программы с измененными значениями.

Проверенная и отлаженная исполняемая программа может выполняться на компьютере до тех пор, пока она существует. После запуска исполняемой программы для решения задачи загрузчик размещает программу в оперативной памяти. Центральный процессор выбирает и выполняет каждую инструкцию программы, сохраняя новые значения данных по мере выполнения программы.



Глава 3

Создание консольного приложения

Консольное приложение — это программа, которая выполняется из командной строки окна DOS или Windows и не имеет графического интерфейса. Проект консольного приложения создается пустым и предполагает добавление в него исходных файлов вручную.

Среда программирования MVC++ использует концепцию рабочей области (solution), что на один уровень абстракции выше, чем проект (project). Проект — это множество всех файлов, необходимых для построения исполняемого файла. В одной рабочей области может содержаться несколько проектов. Несмотря на наличие в рабочей области нескольких проектов, работать можно только над одним, называемым активным. Для рабочей области создается отдельная папка, включающая каталог \Debug и несколько файлов, основным из которых является конфигурационный файл области с расширением `.sln`. Внутри рабочей области каждый проект имеет собственный каталог, в котором находятся конфигурационный файл проекта с расширением `.vcproj` и исходные файлы проекта с расширениями `.h` (заголовочные файлы) и `.cpp` (файлы реализации). Во время создания проекта в его каталоге создается пустая папка \Debug, где будут храниться объектные файлы `.obj`. Исполняемые файлы `.exe` всех проектов рабочей области записываются в каталог рабочей области.

Запуск MVC++

Для запуска среды создания приложений MVC++ необходимо последовательно выполнить следующие действия:

1. Нажать кнопку **Пуск** (Start).
2. Выбрать пункт меню **Программы** (Programs).

3. Выбрать пункт меню **Visual Studio 2005**. В результате откроется стартовое окно, которое показано на рис. 3.1.

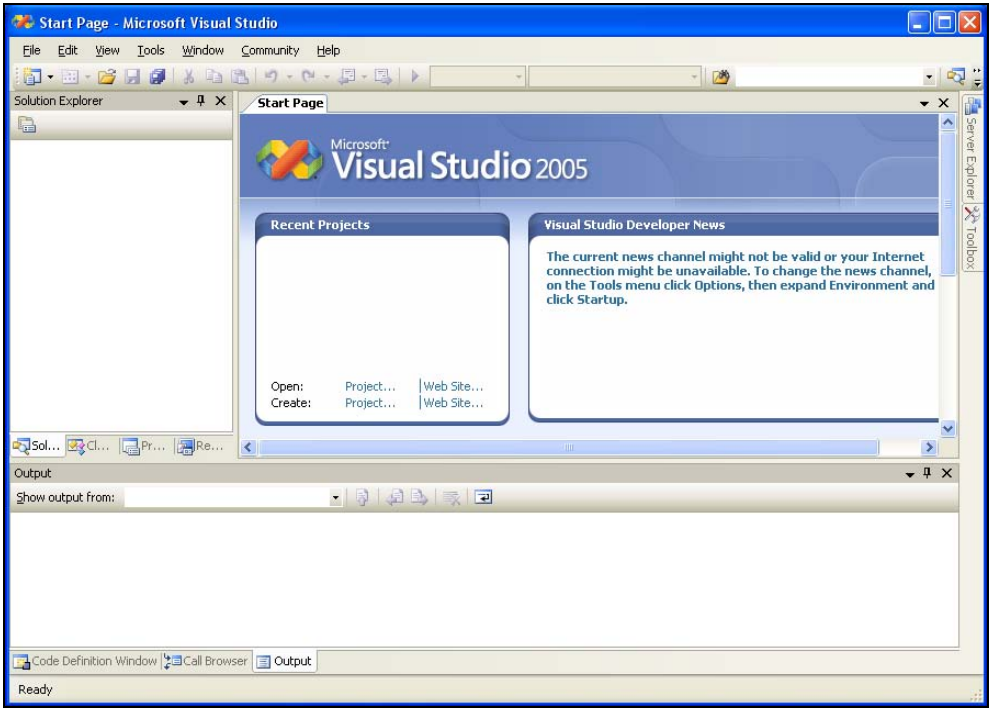


Рис. 3.1. Стартовое окно MVC++

В верхней части окна расположено меню, с помощью которого создаются, изменяются и выполняются программы C++. После выбора пункта меню появляется выпадающее меню, в котором следует выбрать соответствующий пункт. Состояние меню и окон меняется в зависимости от выполняемых программистом действий.

Создание проекта в новой рабочей области

Как упоминалось, рабочая область (solution) создается с целью объединения схожих по тематике проектов. Для создания проекта консольного приложения в новой рабочей области необходимо:

1. Выбрать пункт меню **File → New → Project**.

2. В открывшемся окне **New Project**:
   - в строке **Location** ввести каталог расположения рабочей области или выбрать этот каталог в окне **Project Location**, нажав кнопку **Browse**;
   - в нижней части окна в строке **Solution Name** вместо **<Enter_Name>** набрать название рабочей области;
   - убедиться, что установлен флажок **Create directory for solution**;
   - в строке **Name** вместо **<Enter_Name>** ввести имя проекта;
   - в левой части окна **Project types** выбрать тип проекта **Visual C++ General**;
   - в правой части окна **Templates** выбрать шаблон проекта **EmptyProject**;
   - нажать кнопку **OK**.

На рис. 3.2 демонстрируется вид окна **New Project** при создании проекта p1 в новой рабочей области Exercise, которая будет расположена на диске E.

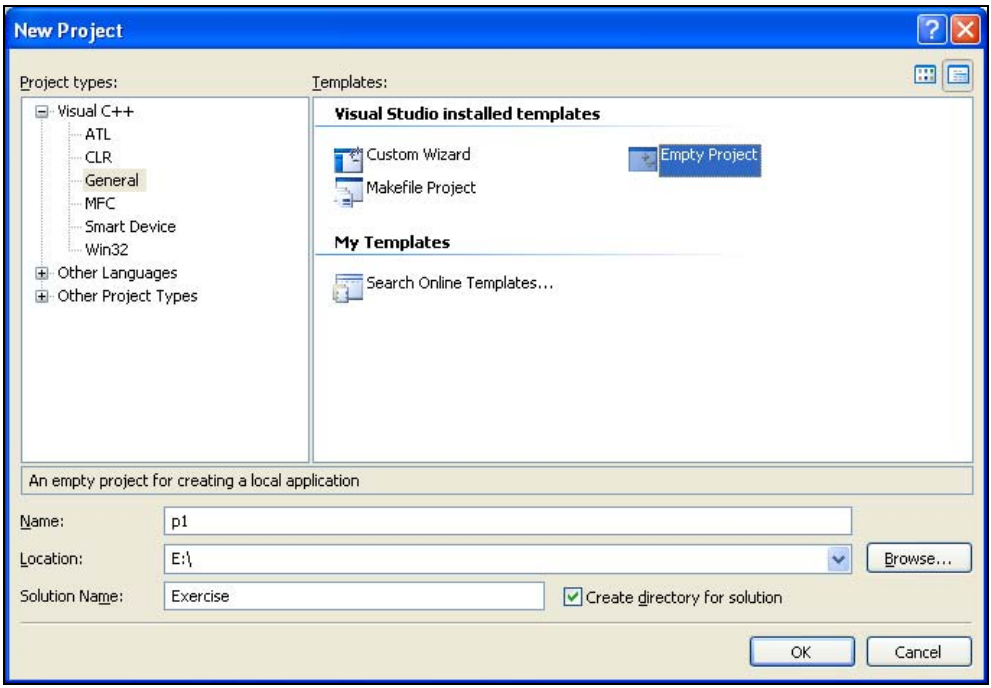


Рис. 3.2. Создание проекта в новой рабочей области

В результате работы мастера MVC++ на диске E в корневом каталоге будет создана папка \Exercise, внутри которой будет сформирован файл рабочей

области Exercise.sln, а также папка \Debug для хранения исполняемых файлов рабочей области. В браузере активной вкладки **Solution Explorer** правой части экрана будет выведено состояние рабочей области, как показано на рис. 3.3.

Рис. 3.3. Состояние рабочей области после создания проекта

Открытие существующей рабочей области

Открыть существующую рабочую область можно различными способами, например, используя MVC++ или проводник.

Для открытия области с помощью MVC++ необходимо выполнить следующие действия:

1. Открыть MVC++.
2. Выбрать нужную область из списка **Recent Project** в стартовом окне.

Если в списке стартового окна нет названия необходимой рабочей области:

1. Выбрать меню **File → Open Project/Solution**.

2. В окне **Open Project** выделить файл рабочей области и открыть его, нажав кнопку **Open**.

Примечание

Если во вкладке **Solution Explorer** браузера MVC++ уже имеется открытая рабочая область, то проекты вновь открываемой рабочей области могут быть добавлены в состав существующей. Для этого следует в окне **Open Project** установить переключатель **Add to Solution** и нажать кнопку **Open**. В результате проекты будут добавлены в уже открытую рабочую область. Если в добавлении проектов нет необходимости, должен быть установлен переключатель **Close Solution**, тогда открытая рабочая область закроется, и будет открыта выбранная в окне **Open Project**.

Для открытия рабочей области при помощи проводника Windows следует выполнить такие шаги:

1. Открыть Проводник.
2. Выделить файл рабочей области (например, **Exercise.sln**).
3. Дважды щелкнуть левой кнопкой мыши по названию файла.

Создание нового проекта в рабочей области

Для создания нового проекта консольного приложения внутри рабочей области проще всего выполнить следующую последовательность действий:

1. Нажать правую кнопку мыши на имени рабочей области в браузере активной вкладки **Solution Explorer**.
2. В появившемся меню выбрать пункт **Add**.
3. В появившемся меню выбрать пункт **New Project**.
4. В окне Add New Project:
   - в левой части окна **Project types** выбрать тип проекта **General**;
   - в правой части окна **Templates** выбрать шаблон проекта **Empty Project**;
   - в нижней части окна в строке **Name** вместо **<Enter_name>** набрать без пробелов название проекта;
   - нажать кнопку **OK**.

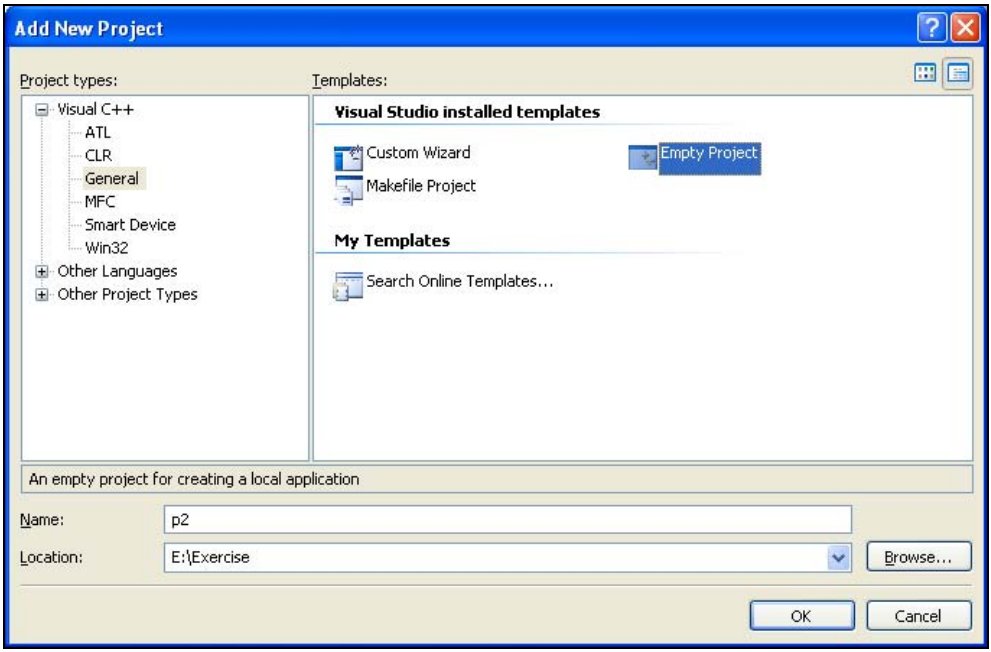


Рис. 3.4. Создание нового проекта в открытой рабочей области

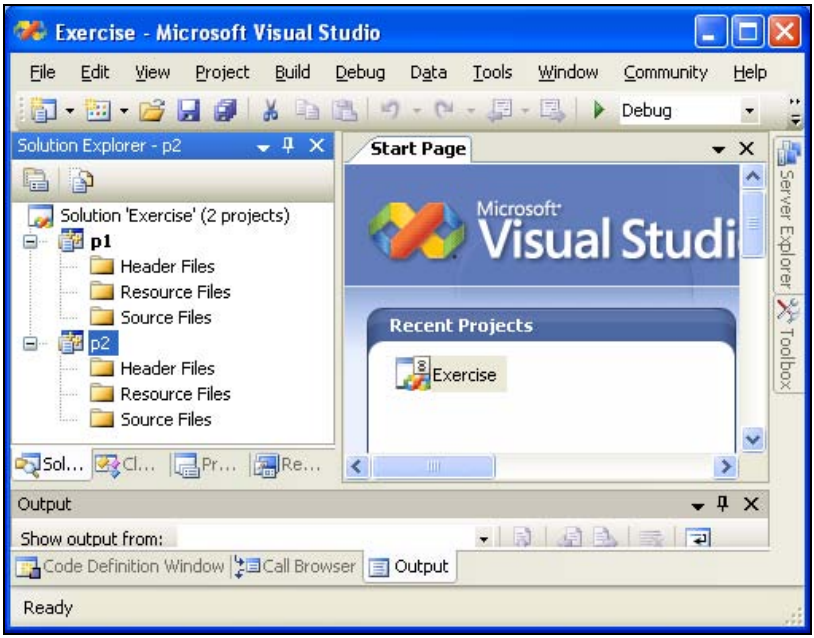


Рис. 3.5. Состояние браузера рабочей области с проектами

На рис. 3.4 показано окно при создании нового проекта p2 в области Exercise.

В результате на диске E в каталоге \Exercise будет создана папка p2, внутри которой будет сформирован файл проекта консольного приложения p2.vcproj. Состояние браузера в активной вкладке **Solution Explorer** показано на рис. 3.5.

Активизация существующего проекта

Рабочая область, как правило, содержит множество проектов. Чтобы сделать проект активным, следует во вкладке **Solution Explorer** выделить имя нужного проекта, нажать правую кнопку мыши и выбрать в меню пункт **Set as StartUp Project**. Название активного проекта в браузере выделяется при этом полужирным шрифтом.

Добавление исходных файлов в проект

В проект консольного приложения необходимо вручную включить исходные файлы: заголовочные файлы (Header Files) и файлы реализации (Source Files).

Включение обоих файлов производится одинаково. Разница состоит только в выборе шаблона: для заголовочного файла шаблон называется Header File (.h), а для файла реализации — C++ File (.cpp).

Далее указаны действия, необходимые для подключения файла реализации:

1. Нажать правую кнопку мыши на имени проекта в браузере Solution Explorer.
2. В появившемся меню выбрать пункт **Add**.
3. В появившемся меню выбрать пункт **Add New Item**.
4. В окне Add New Item:
   - выбрать в правой части окна **Categories** категорию Code;
   - выбрать в левой части окна **Templates** шаблон **C++ File (.cpp)**;
   - ввести в окне **Name** вместо **<Enter_name>** имя файла;
   - нажать кнопку **Add**.

На рис. 3.6 приводится окно для включения файла с именем main в проекте p1 рабочей области Exercise.

В результате на диске в каталоге проекта будет создан файл main.cpp, и в редакторе MVC++ появится окно и вкладка для работы с этим файлом. Состояние главного окна MVC++ после включения файла реализации показано на рис. 3.7.

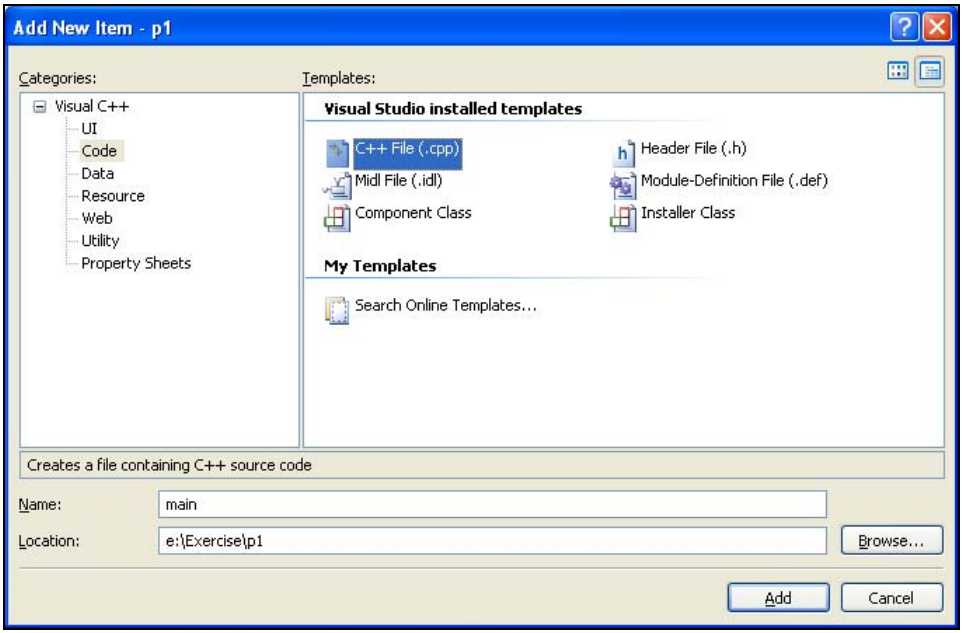


Рис. 3.6. Добавление файла реализации в проект

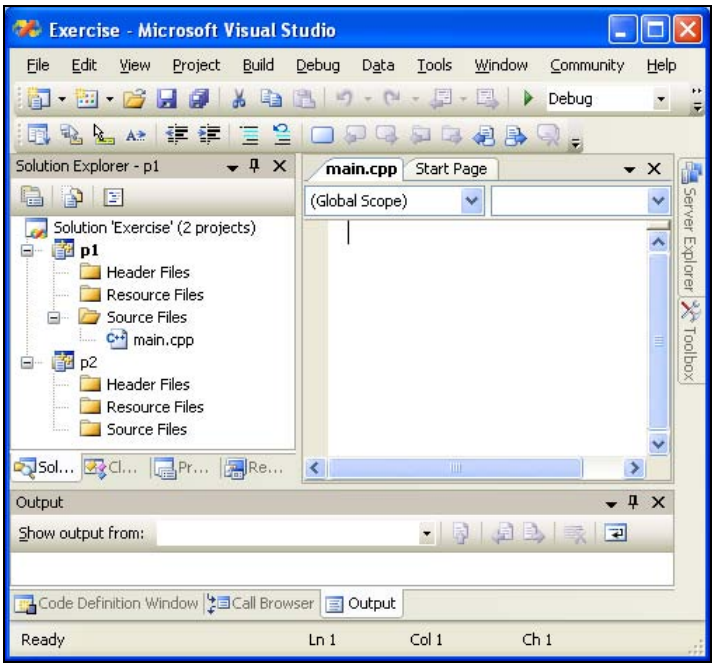


Рис. 3.7. Состояние главного окна после подключения файла реализации

Примечание

Имена исходных файлов задаются автором программы. В начале книги рассматриваются простейшие проекты, состоящие только из одного файла, который содержит исходный код главной функции `main ( )`. Для проектов, состоящих из нескольких файлов, рекомендуется имя заголовочного файла и файла реализации главной функции выбирать таким же, как и имя проекта, а для файлов с описанием и реализацией используемых в программе классов использовать имена самих классов, но без первого символа `C`, с которого обычно они начинаются.

Ввод и редактирование исходного текста осуществляется в активном окне редактора. Редактор поддерживает команды **Cut**, **Copy** и **Paste**, **Undo** и **Redo**, расположенные в меню **Edit**, где также показаны горячие клавиши и пиктограммы для этих действий.

В листинге 3.1 приведен текст первой программы, которая просто выводит текст на экран. Из этой программы будет создаваться исполняемый файл.

Листинг 3.1. Первая программа

```
#include <iostream>
using namespace std ;
int main ( )
{
       cout << "I will be super programmer!" << endl ;
return 0 ;
}
```

Активизация исходного файла для редактирования

Если в окне редактора существует вкладка с именем файла, требующего редактирования, то достаточно щелкнуть левой кнопкой мыши по вкладке. Редактор сделает окно редактирования этого файла активным.

Если такой вкладки не существует, то необходимо дважды щелкнуть левой кнопкой мыши по имени файла в окне **Solution Explorer**. Файл откроется и загрузится в активное окно редактора, в котором появится вкладка с именем открытого файла.

Если файла не существует в проекте, то следует его добавить в проект так, как описано в предыдущем разделе, и приступить к его редактированию.

Сохранение и закрытие файла

Для сохранения файла из активного окна редактора под тем же именем можно либо щелкнуть левой кнопкой мыши по пиктограмме сохранения на панели инструментов, либо выбрать в меню **File → Save**.

Для сохранения файла из активного окна под другим именем необходимо выбрать в меню **File → Save As**. В этом случае в открывшемся окне **Save File As** указать новое имя, а при необходимости выбрать новый каталог, после чего нажать кнопку **Save**. Следует помнить, что сохраненный под другим именем файл не будет являться файлом проекта.

Для закрытия активного файла нажимается кнопка закрытия с крестиком в правом верхнем углу окна редактора.

Трансляция файлов реализации

Следует помнить, что компилируются только файлы реализации, имеющие расширение cpp. Перед трансляцией файл реализации может быть как активным (то есть быть открытым в редакторе), так и нет. Удобнее, если файл будет активным.

Если файл реализации активен, то для его трансляции можно выполнить одно из двух действий:

1. Нажать комбинацию клавиш <Ctrl> + <F7>.
2. Выбрать в меню **Build → Compile**.

Если файл не активен, то выбрать его в списке файлов активной вкладки **Solution Explorer**, нажать правую кнопку мыши и выбрать в меню строку **Compile**.

В случае успешной трансляции исходного файла в папку проекта \Debug компилятор запишет объектный файл с расширением obj и с именем исходного файла. Получив сообщение об успехе (рис. 3.8) после компиляции всех включенных в проект файлов реализации, можно переходить к компоновке.

Компилятор может обнаружить ошибки, тогда внизу экрана будут выведены сообщения о найденных синтаксических ошибках. Для быстрого перемещения к содержащей ошибку строке необходимо дважды щелкнуть левой кнопкой мыши по сообщению об ошибке в нижней части экрана. На рис. 3.9 продемонстрировано состояние окон MVC++ после компиляции программы с ошибками. При вводе текста программы не были набраны кавычки, обрамляющие текстовую константу.

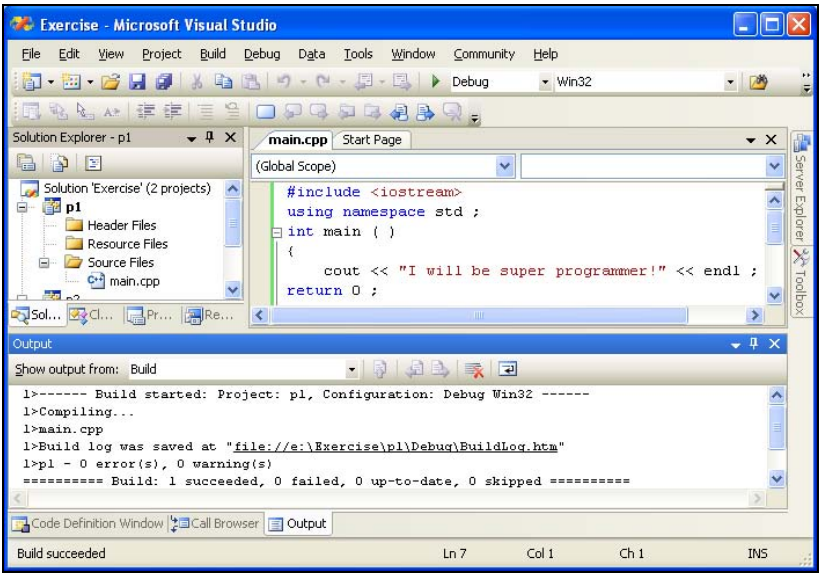


Рис. 3.8. Сообщение об успешной компиляции файла

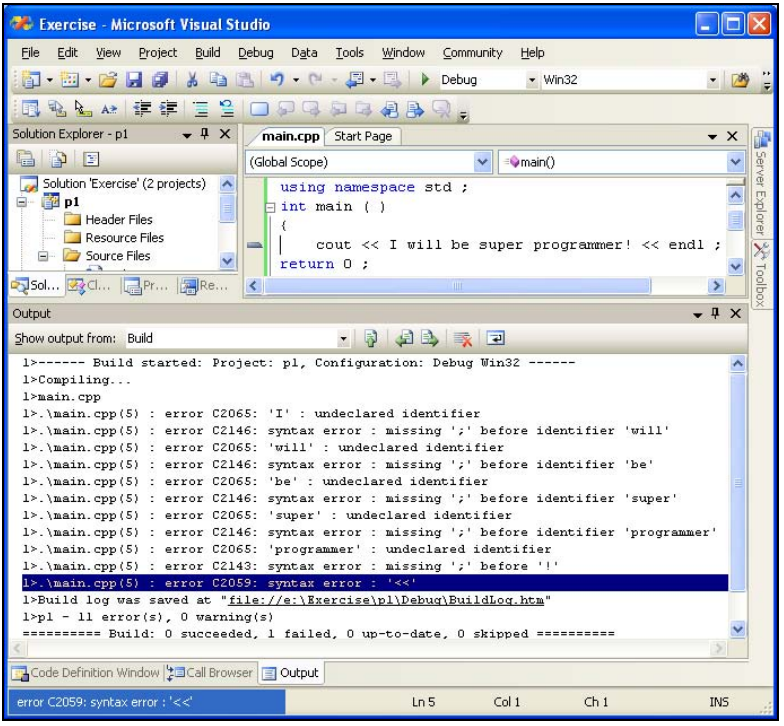


Рис. 3.9. Сообщения об ошибках

Примечание

Компилятор последовательно выполняет проверку синтаксиса программы, поэтому сначала сообщается о неопределенных идентификаторах переменных, а также о пропущенных символах завершения оператора, и только в конце обнаруживает неверную запись операции вывода. Начинающие программисты ошибаются в самых простых операторах. Часто после исправления первой из 127 обнаруженных компилятором ошибок не остается ни одной.

Если компилятор выведет сообщения об ошибках, следует исправить эти ошибки и повторить трансляцию.

Компоновка

Для компоновки приложения из объектных файлов достаточно выбрать в меню **Build → Build**.

Если компоновщик не обнаружит ошибок, то в нижней части экрана появится сообщение, которое показано на рис. 3.10. В результате успешной компоновки в папку проекта \Debug рабочей области будет записан исполняемый файл с расширением exe и с именем проекта. Файл можно вызывать на исполнение.

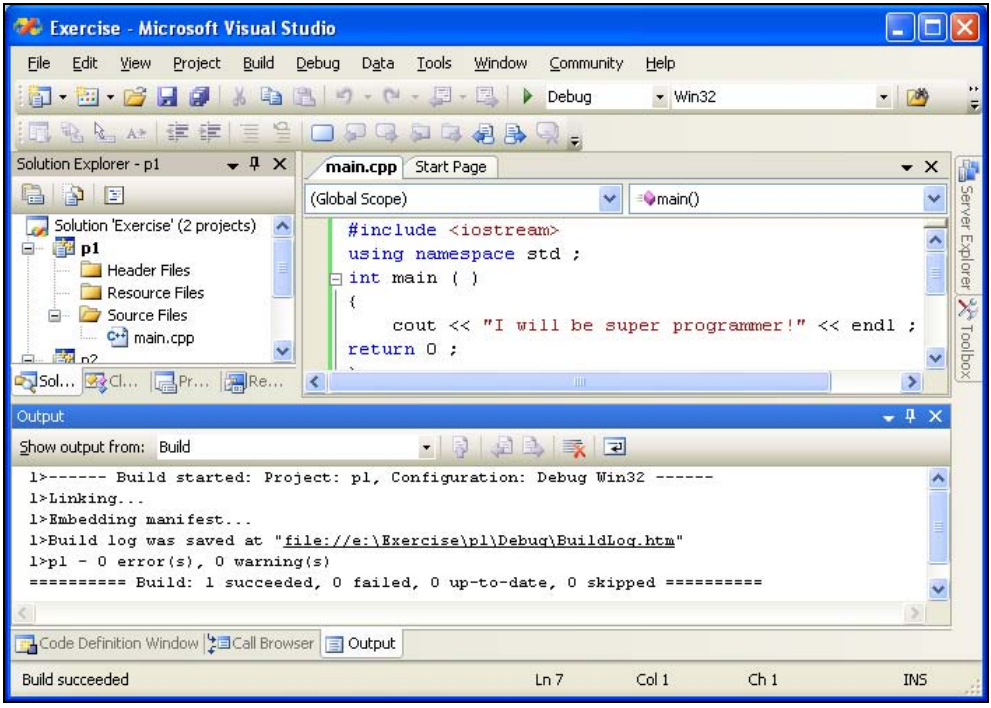


Рис. 3.10. Сообщение об успешной компоновке приложения

Если компоновщик сообщит об ошибках, их следует исправить и повторить трансляцию с последующей компоновкой.

Следует заметить, что для трансляции всех файлов проекта и последующей компоновки приложения можно выбрать в меню **Build → Rebuild**.

Отладка приложения

Во время работы приложения могут возникать ошибки исполнения. Ошибку легче обнаружить при помощи отладчика, который помогает понять, где именно она встречается.

Для начала отладки следует убедиться, что файл реализации с предполагаемым местом ошибки активен в окне редактора.

Пошаговая трассировка

Чтобы запустить отладчик и начать трассировку с главной функции `main ( )`, необходимо нажать клавишу <F10>. Рядом со строкой с фигурной скобкой в главной функции `main ( )` появится желтая стрелка. Если нужно начать трассировку с другой строки, то необходимо установить курсор на начало этой строки и нажать комбинацию клавиш <Ctrl>+<F10>. Желтая стрелка теперь появится рядом с выбранной строкой.

Для продолжения трассировки следует нажать клавишу <F10>. Это приведет к тому, что отладчик перейдет к следующему выражению, которое можно выполнять. Соответственно стрелка сделает то же самое. Каждое нажатие клавиши <F10> означает шаг трассировки, то есть переход к следующему выражению программы. Трассировка позволит проверить логику программы и определить место ошибки.

Пошаговая трассировка с заходом в функции

Если в программе одна функция обращается к другой, можно осуществить пошаговую трассировку каждого шага вызванной функции. Это делается с помощью клавиши <F11>. В отличие от режима трассировки по <F10>, который рассматривает каждую встреченную функцию как одно выражение, такой режим позволяет войти внутрь функции и выявить в ней возможные ошибки. По мере необходимости можно чередовать нажатия <F10> и <F11> в процессе отладки приложения.

Точки останова

Точка останова (breakpoint) позволяет временно прервать выполнение программы в указанном месте и перейти к режиму отладки. Бывают случаи, когда необходимо иметь несколько таких точек.

Чтобы установить точку останова, следует поместить курсор на ту строку, в которой программа при трассировке должна остановиться, нажать правую кнопку мыши и выбрать в появившемся меню **Insert Breakpoint**. Рядом с этой строкой появится красный круг. Соответственно для запрещения точки останова следует выбрать **Disable Breakpoint**, а для удаления — **Remove Breakpoint**. При запрещении круг теряет красную окраску, а при удалении он исчезает.

Если в программе установлены точки останова, то вызов приложения на исполнение происходит при выборе в меню **Debug → Start**. Теперь программа, проработав до точки останова, перейдет в режим отладки.

Просмотр переменных

В процессе пошаговой трассировки существует возможность просмотра значений переменных. Чтобы увидеть локальные переменные, необходимо щелкнуть по вкладке **Locals** в левом нижнем углу экрана. Вкладка **Auto** покажет выборку переменных, сделанную компилятором.

В процессе трассировки программы можно создать собственный набор просматриваемых переменных. Для этого требуется выполнить следующие действия:

1. Щелкнуть правой кнопкой мыши на имени переменной в исходном тексте программы.
2. Из появившегося меню выбрать пункт быстрого просмотра **QuickWatch**.
3. В диалоговом окне щелкнуть добавление **Add watch** для подтверждения выбора.

Если переменная недоступна (например, не определена в программе), окно просмотра выдаст сообщение об ошибке вместо значения рядом с именем переменной.

Прерывание режима отладки

В любой момент времени пошаговая трассировка программы может быть прервана. Для прерывания режима отладки требуется нажать комбинацию клавиш <Shift>+<F5>.

Исполнение приложения

Для исполнения приложения в среде MVC++ необходимо выбрать в меню **Debug → Start Without Debugging**.

Исполняемый файл начнет выполняться на компьютере.

Чтобы завершить работу приложения в ответ на предложение **Press any key to continue**, следует нажать любую клавишу.

Теперь приложение можно выполнять сколько угодно раз до тех пор, пока существует исполняемый файл. Результат работы приложения, построенного из первой программы, показан на рис. 3.11.

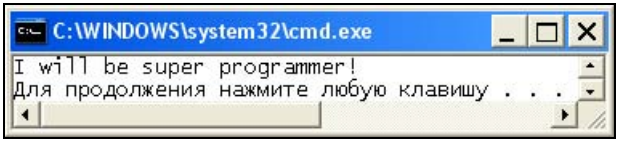


Рис. 3.11. Результат исполнения первой программы

Использование вкладок

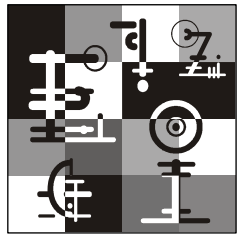
Для удобства работы над проектами программист имеет возможность настроить среду MVC++. В левой части окна можно разместить различные вкладки, чтобы во время работы можно было быстро найти необходимые файлы, классы или справочную информацию.

Для настройки вкладок необходимо выбрать в меню вида **View** соответствующий пункт:

- для вкладки файлов — **Solution Explorer**;
- для вкладки классов — **Class View**.

Для доступа к справочной информации можно выбрать в главном меню пункт **Help → Dynamic Help**, тогда справа появится браузер помощи. Динамическая помощь реагирует на перемещение курсора в тексте редактора, и при наличии соответствующего раздела справки отображает его название в браузере помощи. Можно самостоятельно открывать имеющиеся разделы помощи.

Если в меню выбрать **Help Contents**, то откроется отдельное окно Microsoft Visual Studio 2005, где доступна библиотека справочной информации MSDN Library for Visual Studio 2005.



Глава 4

Функция *main ( )*

В первой части книги рассматриваются возможности языка C++ для создания простейших консольных приложений. Для такого типа приложений функция `main ( )` является обязательной и используется как начальная отметка выполнения программы.

Данная функция не требует в программе специального объявления, но может быть записана по-разному. Это зависит от используемого типа функции, который может быть `int` или `void`. Тип `int` объявляет `main ( )` как функцию, которая возвращает целое значение. В этом случае последним оператором функции должен быть оператор `return`. Тип `void` сообщает компилятору о том, что `main ( )` не возвращает никакого значения. В этом случае последним оператором должен быть вызов функции выхода `exit ( )`. Целое значение, которое возвращает функция main ( ) в момент окончания работы программы, предназначено для операционной системы. Ненулевое значение, возвращенное `main ( )`, означает аварийное завершение. Ноль, указанный для `return` или `exit ( )`, означает успешное завершение `main ( )` (листинги 4.1 и 4.2).

Листинг 4.1. Функция main ( ) типа `int`

```
int main ( )
{       return 0 ;      }
```

Листинг 4.2. Функция main ( ) типа `void`

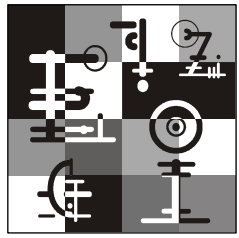
```
void main ( )
{       exit ( 0 ) ;    }
```

Примечание

Язык C++ отдает предпочтение функции целого типа `int`.

Начало функции обозначается символом открывающей фигурной скобки, конец — закрывающей. Для выполнения программой каких-либо действий перед последней операцией передачи управления из функции `main ( )` внутрь фигурных скобок помещаются исполняемые операторы C++, речь о которых пойдет в следующих главах книги.

Представленная функция `main ( )` ничего не делает, но при ее отсутствии консольное приложение работать не будет.



Глава 5

Вывод текста на экран

Вывод текста на экран в процессе выполнения консольного приложения помогает организовать взаимодействие с пользователем программы для ввода необходимых исходных данных, а также просмотра полученных результатов.

Здесь рассматриваются примеры вывода текстовых сообщений о ходе работы программы (листинг 5.1).

Листинг 5.1. Пример вывода сообщения о начале работы программы

```
#include <iostream>
using namespace std ;
int main ( )
{       cout << "Beginning of the program\n" ; return 0 ;         }
```

Первые две строки указывают компилятору об используемой стандартной библиотеке ввода/вывода `iostream` и о пространстве имен `std`. С этих строк начинается любая программа, в которой есть операции ввода/вывода.

Строка внутри функции `main ( )` указывает компьютеру, что текст в кавычках необходимо послать на экран монитора. Присутствие кавычек является обязательным, иначе компилятор будет воспринимать символы как названия переменных. Знак `\n` перед закрывающей кавычкой представляет собой управляющую последовательность и сообщает о необходимости перехода на новую строку после вывода текста.

Листинг 5.2. Пример вывода сообщения о необходимости ввода значения

```
#include <iostream>
using namespace std ;
int main ( )
{       char* msg = "Enter value -> " ; cout << msg ; return 0 ; }
```

Выводимое сообщение хранится по адресу `msg` (листинг 5.2). Содержимое `Enter value ->` , записанное по адресу `msg`, появится на экране в результате работы объекта `cout`. В отличие от предыдущего примера (листинг 5.1), после вывода сообщения перехода на новую строку вывода не произойдет.

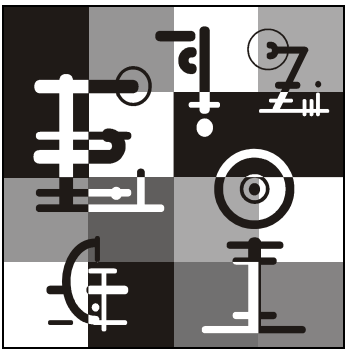
Листинг 5.3. Пример вывода нескольких сообщений друг за другом

```
#include <iostream>
using namespace std ;
int main ( )
{
       cout << "\"Genius is one percent inspiration\n"
              << "and ninety nine percent perspiration.\"\n" ;
       cout << "\tT. A. Edison" << endl ;
       return 0 ;
}
```

Знак `\"` обеспечивает вывод символа кавычки (листинг 5.3). Знак `\t` сообщает о выводе символа табуляции. Манипулятор вывода `endl` точно так же, как и знак `\n`, сообщает о переходе на новую строку. На экран выводятся три строки текста:

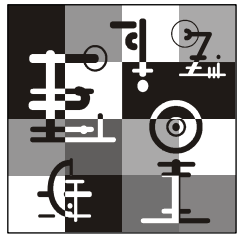
```
"Genius is one percent inspiration
and ninety nine percent perspiration."
       T. A. Edison
```

Подробнее о вводе и выводе данных можно узнать в *главе 7*.



Часть II

Основы языка C++



Глава 6

Простые типы данных

Данные (data), необходимые для работы программы, хранятся в памяти компьютера. Каждая область памяти имеет однозначно определенный адрес, на который ссылаются, когда следует сохранить или прочесть данные. Адрес расположения данных в памяти — это двоичное число в машинном коде. В языках высокого уровня для того, чтобы назвать область памяти, используются идентификаторы, а компилятор транслирует имена в соответствующие адреса. Языки высокого уровня освобождают программиста от необходимости определять действительное расположение данных в памяти компьютера. Каждый элемент данных должен принадлежать к какому-либо определенному типу данных.

Тип данных (data type) — множество допустимых значений данных вместе с набором операций, которые применимы к этим данным. Тип определяет, в каком виде данные представлены в компьютере, сколько байтов они занимают в оперативной памяти, а также — какие преобразования компьютер может к ним применять. Некоторые типы данных известны компилятору языка высокого уровня — их называют встроенными. Встроенные типы данных подразделяются на базовые (простые) и производные типы данных. Помимо встроенных типов программист может определять свои собственные типы — их называют пользовательскими (типы класса). Данные в программе могут быть представлены константами и переменными, которые имеют идентификатор, тип и значение.

Константа (constant) — именованная область памяти компьютера. Значение константы во время выполнения программы никогда не меняется.

Переменная (variable) — именованная область памяти компьютера. В ячейки памяти могут помещаться разные значения переменной, но в каждый момент времени выполнения программы это может быть только единственное значение. В отличие от константы, значение переменной изменяется при работе программы в зависимости от применяемых к ней операций.

Значение (value) — конкретное содержание переменной, константы или выражения. Всегда должно принадлежать области допустимых значений, которая определяется типом переменной, константы или выражения.

Размер элемента данных (item size) — длина переменной или константы, измеряемая в байтах, которая определяет величину области памяти, занимаемую переменной или константой для хранения своего значения.

В C++ имеются простые базовые типы данных, изображенные на рис. 6.1.

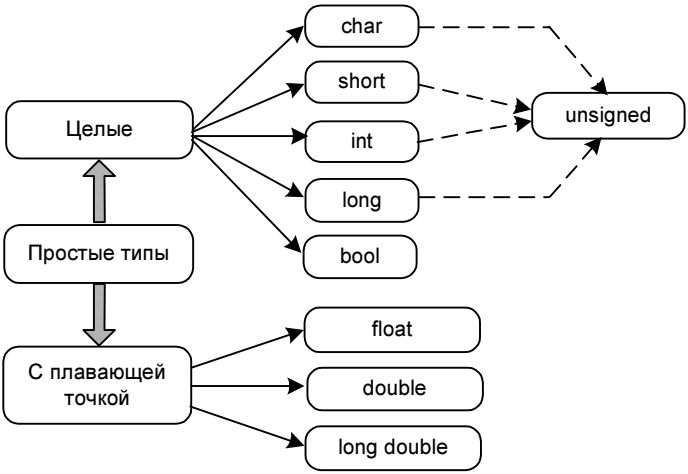


Рис. 6.1. Простые типы данных

Тип `char` используется для представления в программе данных символьного типа. Тип `int` служит для данных целого типа и может быть модифицирован при помощи ключевых слов `short` и `long`. Для типов данных `char`, `int`, `short` и `long` можно ограничиться представлением не отрицательных значений при помощи ключевого слова `unsigned`, которое должно предшествовать названию типа. Типы без знака изменяют диапазон допустимых значений, исключая отрицательные числа. Освободившийся знаковый разряд расширяет диапазон допустимых положительных значений.

К базовым целым типам относится и тип `bool`, служащий для представления логических данных, которые могут принимать значения `true` (истина) и `false` (ложь).

Для использования в программе данных с плавающей точкой существует тип `double`, который может быть модифицирован при помощи ключевых слов `long double` и `float`.

Самым коротким размером в байтах обладает тип `bool`, а самым длинным — `long double`. Язык C++ имеет оператор `sizeof ( )` для определения количества байтов, занятых элементом программы.

Константы простых типов

Объявление констант имеет форматы:

```
const тип идентификатор = значение ;
const тип идентификатор1 = значение1,…, идентификаторN  = значениеN;
```

Используя первый формат, можно объявить одну константу, а второй — несколько констант одного типа (листинг 6.1).

Идентификатор константы рекомендуется записывать прописными буквами.

Листинг 6.1. Пример объявления констант разного типа

```
const float F = 1.25 ;
const double D = 9.42478 ;
const long double LD = 1.7E+300 ;
const unsigned short X = 123 ;
const char A  = 'a' ;
const char COPYA = A ;
const bool T = true ;
const char TRUE [ ] = "true" ;
const char* SAYING = "Slow and steady wins the race." ;
```

Константы с идентификаторами `F`, `D` и `LD` — это числа с плавающей точкой. Разница заключается в размере занимаемой памяти и точности представления, определяемой количеством знаков после запятой. Константа `F` занимает 4 байта, а тип `float` хранит значения с точностью до 7 знаков после запятой. Константы `D` и `LD` занимают 8 байтов. Точность типов `double` и `long double` одинаковая и составляет 15 знаков после запятой, но у типа `long double` шире диапазон допустимых значений.

Константа `X` представляет целое число без знака. Использование перед типом ключевого слова `unsigned` указывает компилятору, что из диапазона изменения такого типа исключаются отрицательные числа.

Символьные константы `A` и `COPYA` имеют одно и то же значение символа `a`, но занимают разные ячейки памяти. Значение символьных констант заключается в апострофы.

Константа `T` имеет логический тип и значение `true`. Логический тип занимает 1 байт оперативной памяти и имеет всего два допустимых значения: `true` (истина) и `false` (ложь).

Константы `TRUE` и `SAYING` объявлены как строковые константы. Значение строковой константы заключается в кавычки. В примере строковые константы объявляются разными способами: константа `TRUE` при помощи символьного массива, а константа `SAYING` с помощью указателя на символьный тип. Более подробно массивы и указатели рассматриваются в *главах 10* и *11*.

Примечание

Следует помнить, что любая константа при объявлении обязательно должна получить конкретное значение.

Переменные простых типов

Объявление переменных имеет следующие форматы:

```
тип идентификатор ;
тип идентификатор1, …, идентификаторN ;
тип идентификатор = значение ;
тип идентификатор1 = значение1, …, идентификаторN  = значениеN ;
```

Согласно указанным форматам можно объявить одну переменную в строке программы или несколько однотипных переменных, указав их идентификаторы через запятую. Кроме того, одновременно с объявлением можно указать первоначальное значение переменной, однако это не обязательно.

Язык C++ допускает объявление переменной в любом месте программы, но перед первым ее использованием (листинг 6.2).

При выборе идентификатора переменной следует учитывать ее назначение в программе, чтобы программный текст читался легче.

Листинг 6.2. Пример объявления переменных разного типа

```
float sum = 0.0 ;
double average ;
unsigned char a ;
bool flag ;
short birthYear = 1703, calendarYear = 2006 ;
int m_capacity, j = 0 ;
long x, y, z ;
```

Переменные с идентификаторами `sum` и `average` объявлены как числа с плавающей точкой обычной и двойной точности, соответственно. Переменная `sum` инициализируется нулевым значением при объявлении.

При помощи идентификатора `a` объявлена символьная переменная без знака.

Логическая переменная `flag` сможет принимать значения `true` или `false`.

Переменные `birthYear`, `calendarYear`, `m_capacity`, `j`, `x`, `y` и `z` — это целые числа. Типы для представления целых чисел `short`, `int`, `long` обеспечивают поддержку разных диапазонов изменения этих чисел. Первые две переменные и переменная `j` при объявлении инициализируются значениями.

Локальные переменные

Локальные переменные (local variables) — это переменные, которые могут быть использованы только в операторах того блока программы, где находится их объявление. Локальные переменные невидимы за пределами блока.

Локальная переменная может быть инициализирована при объявлении каким-либо значением. Это значение будет присваиваться переменной каждый раз при входе в блок. Если локальная переменная не инициализируется при объявлении, ее значение не определено.

В C++ переменные могут объявляться в любом месте программы, но до их первого использования. Важно помнить, если переменная объявлена внутри блока, то после завершения блока она автоматически разрушается, а ее значение теряется.

Как правило, локальные переменные объявляются внутри функций. Функция рассматривается как блок программы, поэтому сохранить значение локальной переменной между вызовами функций невозможно. Сохранение значений между вызовами обеспечивают статические переменные, о которых несколько позднее.

Листинг 6.3. Пример использования локальных переменных

```
int a = 2 ;           // локальная переменная a из внешнего блока
cout << a << endl ;          // выводится 2
{       // вход во внутренний блок
        int a = 7 ;    // локальная переменная a из внутреннего блока
        cout << a << endl ;   // выводится 7 (a из внутреннего блока)
}       // выход из внутреннего блока
cout << ++a << endl ;        // выводится 3 (a из внешнего блока)
```

Переменная a объявлена с инициализацией дважды (листинг 6.3). В обоих случаях она рассматривается как локальная. После выхода из внутреннего блока локальная переменная a этого блока теряет свое значение. Внешний блок использует собственную локальную переменную a, значение которой инкрементируется.

Примечание

Если из текста примера удалить фигурные скобки, компилятор сообщит об ошибке переопределения переменной a и ее многократной инициализации.

Глобальные переменные

Глобальные переменные (global variables) — это переменные, которые доступны из любой части программы и могут быть использованы где угодно, то есть их областью действия является вся программа.

Глобальная переменная должна объявляться вне всех функций один раз.

Глобальная переменная, как и локальная, может быть инициализирована при объявлении каким-либо значением. Это значение может быть изменено в процессе работы программы в любой функции или блоке. Значение неинициализированной глобальной переменной равно нулю (листинг 6.4).

Примечание

Если в глобальных переменных нет особенной необходимости, их следует избегать. Вследствие допустимости изменения значения глобальной переменной в любом месте программы, повышается вероятность появления ошибок. Независимость функций наоборот снижается, так как каждая использующая глобальную переменную функция становится зависимой от значения этой переменной, которая объявлена и может быть изменена вне тела функции.

Листинг 6.4. Пример использования глобальных переменных

```
#include <iostream>
using namespace std ;
int a ;        // глобальная переменная a инициализируется значением 0
void main ( )
{
       cout << a << endl ;          // выводится 0
       {      // вход во внутренний блок
              a = 7 ;               // переменная a меняет значение
```

```
            cout << a << endl ;   // выводится 7 (a во внутреннем бло-
ке)
        }       // выход из внутреннего блока
        cout << ++a << endl ;        // выводится 8 (a во внешнем блоке)
}
```

Глобальная переменная `a` объявлена один раз до начала функции `main ( )` без инициализации, следовательно, она получит значение 0. Во внутреннем блоке переменная получает значение 7. При выходе из внутреннего блока ее значение сохраняется и увеличивается на 1 во внешнем блоке.

Область видимости переменных

Областью видимости идентификатора называется та область программы, в которой на данный идентификатор можно сослаться.

Локальные переменные определяются в определенном блоке программы, следовательно, на них можно ссылаться только в том блоке, где они определены. На глобальные переменные можно ссылаться в любом месте программы.

Любой блок может содержать объявления переменных. Если внешний и внутренний блоки используют один и тот же идентификатор, то идентификатор внешнего блока скрыт до тех пор, пока не закончится внутренний блок.

Если во внутреннем блоке программы необходимо сослаться на глобальную переменную, идентификатор которой совпадает с идентификатором локальной переменной, следует воспользоваться оператором разрешения области видимости, который имеет следующий формат:

```
: : идентификатор
```

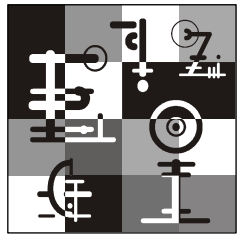
Данный оператор позволяет открыть доступ к той переменной, имя которой скрыто внутри вложенной области видимости (листинг 6.5).

Листинг 6.5. Пример использования оператора разрешения области видимости

```
#include <iostream>
using namespace std ;
int n ;                        // объявление глобальной переменной n
int main ( )
{
        int x = 3 ;            // x - локальная переменная
        n = 2 ;                // n - глобальная переменная
        cout << n * x << endl ;      // выводится 6 ( 2 * 3 )
```

```
    int n = 7 ;           // n - локальная  переменная
    // локальная переменная n скрывает глобальную переменную n
    cout << n * x << endl ;      // выводится 21 ( 7 * 3 )
    // разрешается видимость глобальной переменной :: n
    cout << :: n * x << endl ;   // выводится 6 ( 2 * 3 )
    return 0 ;
}
```

Глобальная переменная `n` получает значение 2 внутри функции `main ( )`. Значение сохраняется. Далее в программе объявляется локальная переменная с точно таким же идентификатором `n` и одновременно инициализируется значением 7. Эта переменная скрывает глобальную переменную, поэтому в вычислении произведения используется значение локальной переменной `n`. В последней строке программы используется оператор разрешения области видимости `:: n`, который открывает доступ к значению глобальной переменной `n`. В результате на экран выводится произведение, для вычисления которого используется глобальная переменная.



Глава 7

Ввод и вывод данных

Ввод/вывод представляет собой дополнение к языку C++ в виде набора средств, который находится в стандартной библиотеке `iostream`. Данная библиотека позволяет программировать ввод данных при помощи объекта `cin` класса `istream` и вывод данных при помощи объекта `cout` класса `ostream`, а также перегруженных операторов поразрядного сдвига `>>` для ввода и `<<` для вывода данных.

Объект `cin` поддерживает последовательный доступ к стандартному устройству ввода, которым является клавиатура. Объект `cout` предоставляет последовательный строковый доступ к устройству стандартного вывода — экрану монитора.

Перегруженный оператор правого поразрядного сдвига `>>` позволяет получить данные из входного потока, а оператор левого поразрядного сдвига `<<` обеспечивает размещение данных в выходной поток.

Для форматирования вывода используются манипуляторы, наиболее часто используемым из которых является `endl`. Манипулятор `endl` сообщает выходному потоку о том, что при выводе будет произведен переход на новую строку.

Форматирование вывода также можно производить при помощи управляющих последовательностей. Чаще всего используется `\n` для перехода на новую строку и `\t` для вывода в выходной поток символа табуляции. Использование `\t` позволяет форматировать расположение данных как в одной строке, так и в последовательно расположенных строках. Например, при выводе матрицы удобно воспользоваться последовательностью `\t` при выводе элементов строки, чтобы элементы столбцов матрицы располагались на экране с одной и той же точки.

Для использования объектов и операторов ввода/вывода необходимо добавить в начало файла, где они используются, следующие две строки:

```
#include <iostream>
using namespace std ;
```

Первая строка служит для подключения виртуального образа стандартной библиотеки `iostream`.

Вторая строка необходима для того, чтобы сообщить компилятору об использовании стандартных потоков ввода/вывода, то есть клавиатуры и монитора (листинг 7.1). Если не указать в программе пространство имен `std`, то придется записывать префикс `std ::` перед каждым использованием объектов `cin` и `cout`.

Листинг 7.1. Пример ввода/вывода данных простых типов

```
#include <iostream>
using namespace std ;
int main ( )
{
       int i ; double x ; char a ;
       cout << "Enter an integer ->\t" ;   cin >> i ;
       cout << "Enter any letter ->\t" ;   cin >> a ;
       cout << "Enter real value ->\t" ;   cin >> x ;
       cout << "int\t" << i << endl ;
       cout << "char\t\'" << a << '\'' << endl ;
       cout << "double\t" << x << endl ;
       cout << "Enter two numbers through a space -> " ; cin >> i >> x ;
       cout << i << '\t' << x << endl ;
       cout << "i = " << i << "\tx = " << x << endl ;
       return 0 ;
}
```

В первой строке функции `main ( )` объявлены три переменные: целая переменная `i`, переменная с плавающей точкой `x` и символьная переменная `a`.

Далее использованы три пары операторов `cout – cin` для вывода подсказки о вводе переменной и ожидание ввода с клавиатуры значения переменной, которое она получит после преобразования из введенной с клавиатуры строки.

Следующие три оператора выводят на экран монитора значения, которые получили переменные в результате ввода. Значение символьной переменной

будет выведено в апострофах, для чего используется управляющая последовательность `\'`. Перед значением каждой переменной выводится наименование типа данных.

Последние операторы перед `return` демонстрируют одновременный ввод и вывод нескольких значений, для чего используется *каскадирование* операций `>>` и `<<`. Новые значения переменных `i` и `x` сначала выводятся через символ табуляции без их названия, а потом с указанием использованных в программе идентификаторов этих переменных. Результат выполнения программы показан на рис. 7.1.

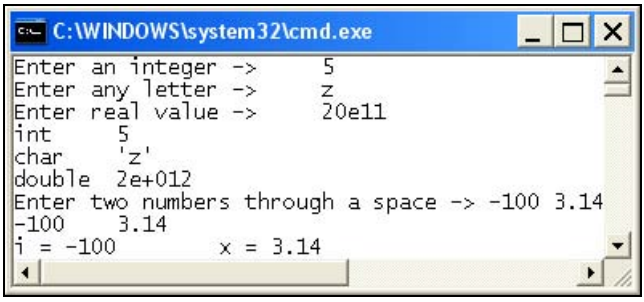
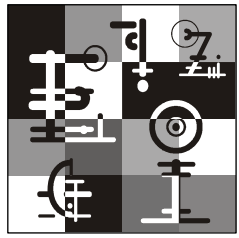


Рис. 7.1. Результат выполнения программы ввода-вывода переменных простых типов

Примечание

Ввод значения происходит после нажатия клавиши <Enter>. Символ пробела при вводе служит разделителем вводимых значений переменных.



Глава 8

Операции над операндами простых типов

Операция (operation) — действие, выполняемое над операндами. Каждая операция обозначается определенным знаком. Например, сложение обозначается знаком `+`, деление — знаком `/`, взятие адреса — знаком `&`.

Операнд (operand) — часть команды, определяющая аргумент, над которым выполняется операция. Операндами являются данные и результаты операции. В качестве операндов используются переменные, константы, вызовы функций.

Выражение (expression) — конструкция языка (формула) для вычисления значения в соответствии со значениями операндов. Значение выражения вычисляется путем выполнения операций над операндами в порядке, соответствующем приоритету операций.

В зависимости от количества операндов (один или два), различают унарные (unary operation) и бинарные (binary operation) операции. Например, изменение знака числа является унарной операцией, а умножение двух чисел — бинарной.

Приоритет операций (operation precedence) — очередность выполнения операций в выражении. Например, в арифметическом выражении сначала выполняются операции умножения и деления, затем — сложения и вычитания.

Далее рассматриваются основные операции языка C++ над операндами простых типов, к которым относятся целые числа и арифметические выражения, которые используют целые числа, символы или числа с плавающей точкой.

Арифметические операции

В табл. 8.1 приводятся все арифметические операции, их обозначение в программе, а также примеры с пояснениями.

Таблица 8.1. *Арифметические операции*

Операция	Знак	Пример	Пояснения
Сложение	+	`j = i + 2 ;`	Устанавливает `j`, равным `i` плюс 2
Вычитание	–	`j = i – 5 ;`	Устанавливает `j`, равным `i` минус 5
Умножение	*	`y = x * 2;`	Устанавливает `y`, равным `x` умноженное на 2
Деление	/	`z = x / 5 ;`	Устанавливает `z`, равным `x` деленным на 5
Деление по модулю	%	`j = i % 3 ;`	Устанавливает `j`, равным остатку от деления `i` на 3
Инкремент	++	`++i ;` `i++ ;`	Устанавливает `i`, равным `i` плюс 1
Декремент	– –	`– –i ;` `i– – ;`	Устанавливает `i`, равным `i` минус 1

Инкремент и декремент

Инкремент увеличивает значение своего операнда на 1, а декремент его уменьшает на 1. Существует два способа записи этих операторов: префиксная (знак операции ставится перед операндом) и постфиксная (знак ставится после операнда).

При префиксной форме записи сначала над операндом производится действие, а затем он используется.

При постфиксной форме — сначала операнд используется, а затем выполняется действие (листинг 8.1).

Листинг 8.1. Пример префиксной и постфиксной форм записи инкремента и декремента

```
int i = 2 ;
cout << ++i << endl ;        // выводится 3
cout << i++ << endl ;        // выводится 3
cout << i << endl ;          // выводится 4
cout << i-- << endl ;        // выводится 4
cout << i << endl ;          // выводится 3
cout << --i << endl ;        // выводится 2
```

В комментариях (текст, который следует за символами //) показан эффект использования разных форм записи операций инкремент и декремент.

Арифметические операции с присваиванием

Значением выражения, стоящего с правой стороны от знака присваивания =, является значение левого операнда после присваивания.

В языке C++ существуют операторы, которые позволяют совмещать арифметические операции с операцией присваивания. Когда арифметическая операция выполняется вместе с присваиванием, происходит замещение уже существующего значения переменной новым значением.

В табл. 8.2 приводится перечень и обозначение арифметических операций с замещением. Для каждой операции дается пример с пояснением.

Таблица 8.2. *Арифметические операции с присваиванием*

Название	Знак	Пример	Пояснения
Присваивание	=	`y = x ;`	Присваивание значения `x` переменной `y`
Сложение с замещением	+=	`y += 2 ;`	Увеличение переменной `y` на 2
Вычитание с замещением	-=	`j -= 5 ;`	Уменьшение переменной `j` на 5
Умножение с замещением	*=	`y *= 7 ;`	Увеличение переменной `y` в 7 раз
Деление с замещением	/=	`z /= 5 ;`	Уменьшение переменной `z` в 5 раз
Деление по модулю с замещением	%=	`i %= 3 ;`	Устанавливает `i`, равным остатку от деления `i` на 3

В табл. 8.3 включены примеры операций с замещением и их альтернативная запись обычным способом.

Таблица 8.3. *Альтернативная запись арифметических операций с замещением*

Знак	Операция с замещением	Альтернативная запись
+=	`i += 2 ;`	`i = i + 2 ;`
-=	`j -= 5 ;`	`j = j - 5 ;`
*=	`x *= x ;`	`x = x * x ;`
/=	`y /= 0.5 ;`	`y = y / 0.5 ;`
%=	`m %= 7 ;`	`m = m % 7 ;`

Операции отношения

Операции отношения сравнивают между собой два значения. Сравнение устанавливает одно из трех возможных отношений между выражениями: равенство, больше или меньше.

Значением выражений, содержащих операции отношения или логические операции, является истина (true) или ложь (false).

В табл. 8.4 дается название операций, их обозначение в программе и приводятся пояснения о значении, которое может являться результатом операции отношения (листинг 8.2).

Таблица 8.4. *Операции отношения*

Название	Знак	Пояснения
Равенство	==	Истина, если операнды равны, иначе — ложь
Неравенство	!=	Истина, если операнды не равны, иначе — ложь
Меньше	<	Истина, если левый операнд меньше правого, иначе — ложь
Больше	>	Истина, если левый операнд больше правого, иначе — ложь
Меньше или равно	<=	Истина, если левый операнд меньше или равен правому, иначе — ложь
Больше или равно	>=	Истина, если левый операнд больше или равен правому, иначе — ложь

Листинг 8.2. Пример операций отношения

```
int n = 5 ;
cout << ( n == 5 ) << endl ;        // выводится 1
cout << ( n != 5 ) << endl ;        // выводится 0
cout << ( n < 5 ) << endl ;         // выводится 0
cout << ( n > 5 ) << endl ;         // выводится 0
cout << ( n >= 5 ) << endl ;        // выводится 1
cout << ( n <= 5 ) << endl ;        // выводится 1
cout << ( ( n + 2 ) <= 5 ) << endl ;       // выводится 0
```

При выводе значения операции отношения на экран ложь представляется целым нулевым значением, истина — единицей.

Логические операции

Логические операции позволяют производить действия только над выражениями, обладающими двумя значениями — истина или ложь.

К логическим операциям (табл. 8.5) относятся: отрицание, конъюнкция и дизъюнкция. Операция логического отрицания — унарная операция и требует одного операнда. Конъюнкция и дизъюнкция являются бинарными операциями, результат этих операций зависит от значений обоих операндов (листинг 8.3).

Таблица 8.5. *Логические операции*

Название	Знак	Пояснения
Отрицание	!	Значение операнда меняется на противоположное
Конъюнкция	&&	Истина, если оба операнда истинны, иначе — ложь
Дизъюнкция	||	Истина, если хотя бы один из операндов истинен, иначе — ложь

Примечание

Логические операции используются совместно с операциями отношений в условных операторах для организации разветвлений в программе. Значения операндов при вычислении результатов отношений или логических операций не изменяются.

Листинг 8.3. Пример логических операций и операций отношения

```
int n = 5, j = 0  ;          // объявление переменных с инициализацией
cout << ! ( n == 5 ) << endl ;              // выводится 0
cout << ( n >= 5 && n < 100 ) << endl ;     // выводится 1
cout << ( n > 5 || n < 7 ) << endl ;        // выводится 1
cout << ! j << endl ;                       // выводится 1
```

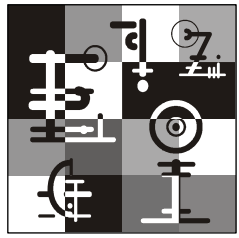
В первой строке объявляются две целочисленные переменные `n` и `j`, которые инициализируются при объявлении значениями 5 и 0 соответственно.

Во второй строке сначала вычисляется значение операции отношения `n == 5`, которое равно истине. Логическая операция отрицания истины дает ложь.

В третьей строке проверяется вхождение переменной `n` в диапазон от 5 до 99 при помощи отношений `n >= 5` и `n < 100`. В результате вычисления значений этих отношений получатся два истинных ответа. Результатом конъюнкции этих отношений так же будет истина.

Далее проверяется дизъюнкция отношений `n > 5` и `n < 7`. Первое отношение ложно, а второе истинно. Следовательно, их дизъюнкция тоже истинна.

В последней строке выводится результат отрицания нулевого значения переменной `j`. Результат очевиден.



Глава 9

Операторы

Оператор (statement) представляет собой установленное синтаксисом языка базовое действие в программе. При выполнении оператора производится некоторое действие. В C++ любое выражение, которое заканчивается символом точки с запятой, является оператором.

Оператор-выражение

К операторам-выражениям относятся следующие: оператор присваивания, оператор вызова функции и пустой оператор.

Формат оператора присваивания имеет вид:

```
идентификатор = выражение ;
```

Например,

```
z = x + y ;
```

Формат оператора вызова функции:

```
имя_функции ( аргумент1, … аргументN ) ;
```

Например,

```
func ( x, 5 ) ;
```

Пустой оператор состоит только из символа точки с запятой. Этот оператор используется, как правило, для обозначения пустого тела в операторах цикла.

Составной оператор

Составной оператор (блок) состоит из одного или большего числа операторов любого типа, заключенных в фигурные скобки. После закрывающейся фигурной скобки не должно быть точки с запятой. Если в блоке только один оператор,

скобки не обязательны. Если в блоке два и больше операторов, наличие фигурных скобок является обязательным. Например,

```
{ z = func ( x, y ) ; z *= z ; }
{ cout << j ; cout << endl ; ++j ; }
```

Примечание

Все операторы C++ имеют свои строгие правила написания. Если правила будут нарушены, компилятор сообщит об ошибке.

Условный оператор *if*

Формат оператора:

```
if  ( выражение )
оператор
```

Если выражение истинно, то выполняется оператор. Если выражение ложно, то программа ничего не делает (листинг 9.1). Логическая схема оператора показана на рис. 9.1.

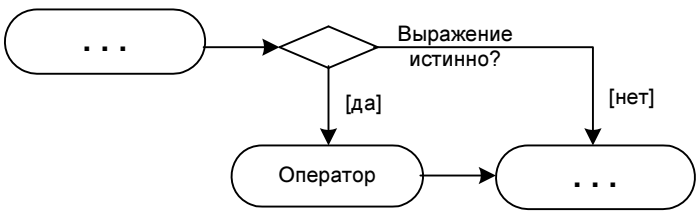


Рис. 9.1. Схема условного оператора `if`

Листинг 9.1. Пример использования оператора `if`

```
if ( a == x )                  // если a  равно x,
       temp = a ;              // то temp  получает значение a
if ( ! c )                     // если отрицание c истина,
{
       temp = x ;              // то temp  получает значение x
       x = y ;                 // x  получает значение y
       y = temp ;              // y  получает значение temp
}
```

Примечание

Обратите внимание на запись условия `a == x` в операторе `if`. Если вместо двух знаков = записать только один, операция отношения равенства изменится на операцию присваивания, что повлечет за собой ошибку.

Условный оператор *if else*

Формат оператора:

```
if  ( выражение )
оператор1
else
оператор2
```

Если выражение истинно, то выполняется `оператор1`, и управление передается на оператор, который следует за `оператором2`. Если выражение ложно, то выполняется `оператор2`, и управление передается на следующий за ним оператор (листинг 9.2).

Логическая схема оператора показана на рис. 9.2.

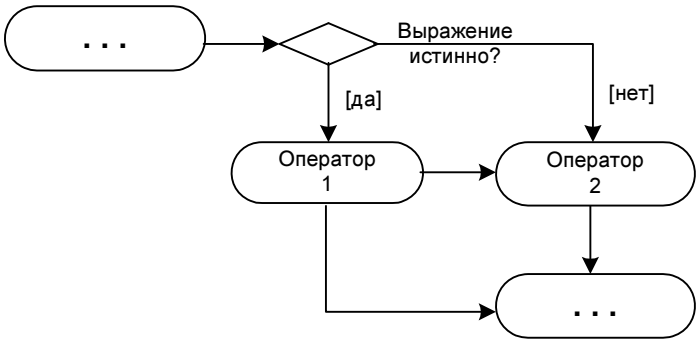


Рис. 9.2. Схема условного оператора `if else`

Листинг 9.2. Пример использования оператора `if else`

```
if ( a == x )                  // если a  равно x,
       temp = a ;              // то temp  получает значение a
else                           // иначе ( a  не равно x )
       temp = -a ;             // temp  получает значение –a
if ( k >= M && k <= N )        // если k ∈ ( N, M ),
```

```
{
       ++k ;                  // то k  увеличивается на единицу,
x *= k ;              // x умножается на k
}
else   x /= k ;              // иначе k ∉ ( N, M ) x делится на k
```

Оператор цикла *while*

Формат оператора:

```
while  ( выражение )
оператор
```

Сначала вычисляется выражение. Если выражение истинно, то тогда выполняется оператор и управление переходит обратно к началу цикла `while`. Оператор будет выполняться до тех пор, пока выражение не станет ложным.

Если выражение ложно, то управление передается следующему оператору (листинг 9.3).

Логическая схема оператора показана на рис. 9.3.

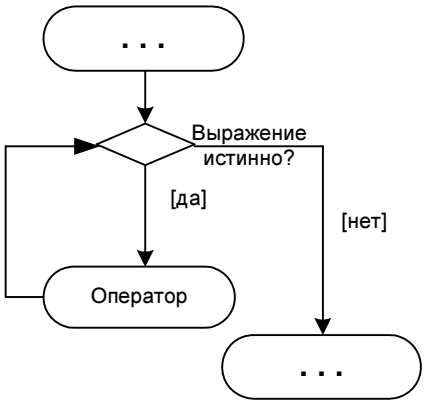


Рис. 9.3. Схема оператора `while`

Листинг 9.3. Пример использования оператора `while`

```
static const int N = 10 ;
int x = 1, sum = 0 ;
while ( x < N + 1 )   // пока x < 11
```

```
{
        sum += x ;      // sum увеличивается на x
        ++x ;           // x увеличивается на единицу
}
cout << sum << endl ; // выводится 55
```

В цикле накапливается сумма целых чисел. Цикл повторится 10 раз. Значение первого числа `x = 1` задается до начала цикла. Переменная `sum` инициализируется значением 0. При каждом проходе цикл `while` увеличивает значение `sum` на текущее значение `x`

```
sum += x ;,
```

затем увеличивает переменную `x` на 1 оператором инкремент

```
 ++x ;.
```

После того как значение `x` станет равным 11, выражение

```
( x < N + 1 )
```

станет ложным, и цикл закончится. Управление перейдет к оператору вывода вычисленного значения суммы.

Примечание

Тело цикла, как правило, представляет собой составной оператор, а это значит, что оно будет заключено в фигурные скобки. Начинающие программисты часто забывают об этом. В результате в цикле остается только один оператор, что может привести к бесконечности цикла. Например, для данного примера, отсутствие фигурных скобок не даст циклу завершиться, так как увеличения `x` не произойдет.

Оператор цикла *for*

Формат оператора:

```
for  (выражение1; выражение2; выражение3)
оператор
```

Сначала вычисляется `выражение1`. Обычно `выражение1` инициализирует переменную, которая используется в цикле.

Затем вычисляется `выражение2`. Если `выражение2` истинно, то выполняется оператор, обрабатывается `выражение3`, и управление переходит к началу цикла, то есть к `выражению2`.

Все повторяется до тех пор, пока `выражение2` не станет ложным. Цикл закончится, и управление получит следующий оператор.

`Выражение1` и `выражение3` могут состоять из нескольких выражений, между которыми ставится запятая (листинг 9.4).

Логическая схема оператора показана на рис. 9.4.

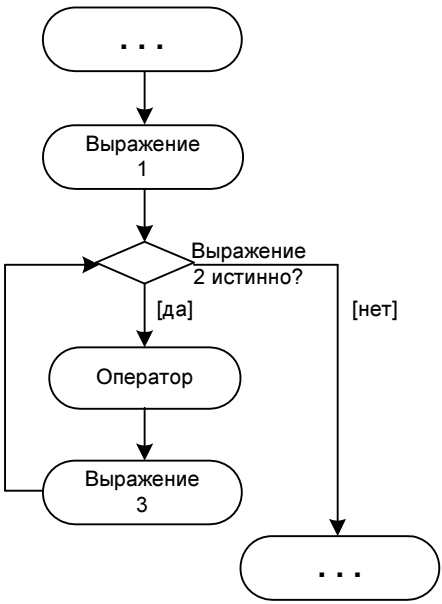


Рис. 9.4. Схема оператора `for`

Примечание

Выражения внутри оператора `for` могут отсутствовать. В этом случае оператор

```
for ( ; ; )
```

представляет собой бесконечный цикл, эквивалентный оператору `while ( true )`. Два символа точки с запятой в операторе `for` являются обязательными.

Листинг 9.4. Пример использования оператора `for`

```
static const int N = 10 ;
for ( int x = 1, sum = 0 ; x < N + 1 ; ++x )
       sum += x ;     // sum увеличивается на x
cout << sum << endl ; // выводится 55
```

В цикле накапливается сумма целых чисел. Цикл повторится 10 раз. Первое выражение в операторе `for` инициализирует переменные `x` и `sum`. Проверяется выражение

```
x < N
```

(1 < 11). Истинный результат приведет к выполнению оператора

```
sum += x,
```

а затем к увеличению переменной `x` на единицу. Выражение

```
x < N
```

(2 < 11) снова будет истинным, следовательно, повторится оператор сложения с замещением и инкремент. Вычисление продолжится до тех пор, пока `x` не станет равным 11. Управление получит оператор вывода значения `sum`.

Оператор цикла *do while*

Формат оператора:

```
do
оператор
while  ( выражение ) ;
```

Сначала выполняется оператор, затем вычисляется `выражение`. Пока `выражение` истинно, управление передается обратно к началу оператора цикла, и процесс повторяется. Когда `выражение` становится ложным, управление переходит к следующему оператору программы (листинг 9.5).

Логическая схема оператора показана на рис. 9.5.

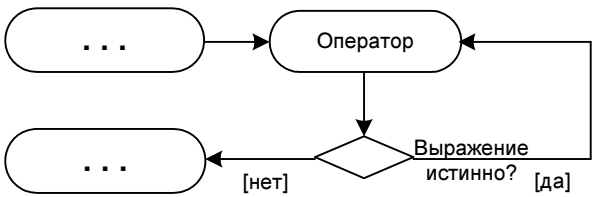


Рис. 9.5. Схема оператора `do while`

Примечание

В отличие от операторов `while` и `for`, оператор `do while` выполняется хотя бы один раз, так как проверка условия цикла происходит в конце оператора. В связи с этим оператор `do while` называют оператором цикла с постусловием, а операторы `while` и `for` — операторами с предусловием.

Следует обратить внимание на символ точки с запятой в конце оператора `do while`, ее присутствие является обязательным.

Листинг 9.5. Пример использования оператора do while

```
static const int N = 10 ;
int x = 1, sum = 0 ;
do                              // выполнять
{
sum += x ;      // увеличивать sum на x
++x ;           // инкрементировать x
} while ( x < N + 1 ) ;         // пока x < N + 1
cout << sum << endl ;           // выводится 55
```

Переменные `x` и `sum`, которые используются в цикле, инициализируются до его начала. В цикле накапливается сумма значений переменной `x`, значение которой увеличивается на единицу в последнем операторе цикла. После выполнения инкремента проверяется условие выполнения цикла

`x < N`.

Цикл выполнится 10 раз, после чего управление передается оператору вывода значения `sum`.

Оператор передачи управления *continue*

Формат оператора:

```
continue ;
```

Оператор направляет поток управления внутри цикла (листинг 9.6). Он заставляет прекратиться текущую итерацию цикла и немедленно начинает следующую. Оператор `continue` может использоваться только внутри тела операторов `while`, `for` и `do while`.

Листинг 9.6. Пример использования оператора continue

```
#include <iostream>
using namespace std ;
int main ( )
{
        int x, sum = 0 ;        // объявление  x и sum, инициализация sum
        bool flag = true ;      // инициализация flag
        while ( flag )          // пока flag истина
        {
                cout << "Enter x -> " ;       // вывод подсказки
```

```
            cin >> x ;                    // ввод значения x
            if ( x > 0 )            // проверка x
            {
                  sum += x ;        // накопление sum для x > 0
                  continue ;        // переход на новую итерацию
            }
            flag = false ;          // изменение flag
      }
      cout << sum << endl ;         // вывод результата
      return 0 ;
}
```

В цикле `while` вводятся значения переменной `x`. Сумма введенных значений накапливается до тех пор, пока вводятся положительные числа. Оператор `continue` обеспечивает передачу управления к оператору вывода подсказки. Оператор

```
flag = false
```

выполняется, когда введено отрицательное число или 0. Выражение `flag` в операторе `while` становится ложным, и цикл `while` завершается, а управление получает оператор вывода результата. Результат выполнения программы с оператором `continue` показан на рис. 9.6.

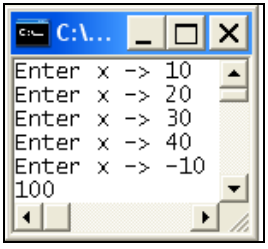


Рис. 9.6. Результат использования оператора `continue`

Оператор передачи управления *break*

Формат оператора:

```
break ;
```

Оператор вызывает выход из цикла оператора `while`, `for` или `do while`, а также из оператора-переключателя `switch`, который будет рассмотрен в следующем разделе. Управление передается оператору, непосредственно сле-

дующему за заканчиваемым оператором цикла или переключателем. Типичное использование оператора `break` состоит в том, чтобы завершить выполнение цикла досрочно (листинг 9.7).

Листинг 9.7. Пример использования оператора `break`

```
#include <iostream>
using namespace std ;
int main ( )
{
       int x, sum = 0 ;      // объявление  x и sum, инициализация sum
       while ( true )        // пока истина
       {
              cout << "Enter x -> " ;      // вывод подсказки
              cin >> x ;                   // ввод значения x
              if ( x <= 0 )         // проверка x
                     break ;        // завершить цикл, когда x ≤ 0
              sum += x ;            // накопление sum для x > 0
       }
       cout << sum << endl ;        // вывод результата
       return 0 ;
}
```

В данном примере выполняются те же вычисления, что и в предыдущем. В цикле `while` вводятся значения переменной `x`. Если введенное значение отрицательное или равно 0, то цикл немедленно завершается, благодаря оператору `break`. Если значение *x* положительное, накапливается сумма. Выход из цикла возможен только при вводе нуля или отрицательного числа. Управление получает оператор вывода результата.

Оператор-переключатель *switch*

Формат оператора:

```
swith  ( выражение )
 {
case константа1 :
        оператор1
case константа2 :
        оператор2
```

```
        …
case константаN :
        операторN
default :
        оператор
}
```

Операторы представляют собой обычные составные операторы, которые содержат метку `case`, а один из них — необязательную метку `default`.

Метка `case` представляет собой константное целочисленное выражение. Все метки должны быть уникальными.

Значением выражения должен быть символ или целое число. Значение выражения сравнивается с константами во всех вариантах `case`, после чего управление передается оператору, который соответствует значению выражения. Выражение определяет, который из вариантов case становится выполняемым.

Обычно действие, которое выполняется после метки `case`, заканчивается оператором `break`, и управление передается следующему за `switch` оператору программы. Если оператор `break` отсутствует, управление передается оператору, следующему за `case`, что делает возможным выполнять одинаковые действия при разных значениях констант.

Если не выбирается ни одна из меток `case`, тогда управление получает `case` с меткой `default`, а после его выполнения оператор `switch` завершается. При отсутствии необязательной метки `default` оператор `switch` заканчивает свою работу. И в первом, и во втором случае управление переходит к следующему оператору программы (листинг 9.8). Логическая схема оператора-переключателя показана на рис. 9.7.

Примечание

Константное выражение не может включать переменные или вызовы функций, а ключевые слова `case` и `default` не могут находиться вне блока `switch`.

Листинг 9.8. Пример использования оператора `switch` с разными действиями для разных констант

```
int x;                                  // объявление целой переменной
char choice ;                           // объявление символьной переменной
cout << "Enter an integer -> " ;        // вывод подсказки
cin >> x ;                              // ввод значения x
cout << "Enter 1...3 -> " ;             // вывод подсказки
cin >> choice ;                         // ввод значения choice
```

```
switch ( choice )                    // анализ значения choice
{
case '1' :                           // если choice равен символу '1',
       cout << x << endl ;                  // то выводится x
       break ;                              // выход из оператора switch
case '2' :                           // если choice равен символу '2',
       cout << x * x << endl ;              // то выводится квадрат x
       break ;                              // выход из оператора switch
case '3' :                           // если choice равен символу '3',
       cout << x * x * x << endl ;          // то выводится куб x
       break ;                              // выход из оператора switch
default :                    // если choice не равен '1','2' или '3'
       cout << "Error of input\n" ;         // то сообщение об ошибке
}
```

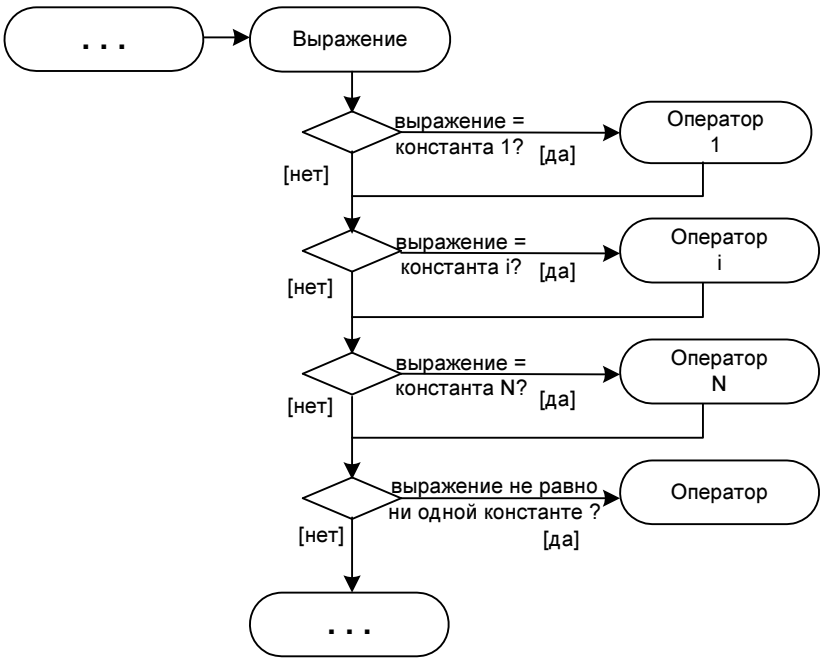


Рис. 9.7. Схема оператора `switch`

Оператор-переключатель анализирует введенное значение символьной переменной `choice`. Оператор `break`, присутствующий в каждом `case`, обеспечивает выход из `switch` после выполнения необходимых действий. Обратите

внимание, что метка `default` не нуждается в операторе `break`, управление будет передано следующему оператору. Если введен символ `1`, то выводится значение переменной `x`. Если `2`, то выводится квадрат `x`. Если `3`, то выводится куб. Наконец, если будет введен символ, отличный от `1`, `2`, или `3`, то выводится сообщение об ошибке ввода.

Листинг 9.9. Пример использования оператора `switch` с одинаковыми действиями для разных констант

```
#include <iostream>
using namespace std ;
int main ( )
{
       char c = ' ' ;                // вводимый символ
       int counterABC = 0 ;          // счетчик символов A, B, или C
       int counterOTHERS = 0 ;       // счетчик остальных символов
       cout << "Enter symbols -> " ;
       while ( c != '.' )            // цикл повторяется, пока c не '.'
       {
              cin >> c ;             // ввод значения символа c
              switch ( c )           // анализ введенного значения
              {
              case 'A' :
              case 'a' :
              case 'B' :
              case 'b' :
              case 'C' :
              case 'c' :
// инкремент счетчика для A, B, или C независимо от регистра
                     ++counterABC ;
                     break ;         // выход из оператора switch
              case '.' :             // если символ является точкой
                     break ;         // выход из оператора switch
// инкремент счетчика для остальных символов
              default :
                     ++counterOTHERS ;
              }                      // конец оператора switch
       }                             // конец оператора while
```

```
        cout << "ABC or abc = " << counterABC << endl ;
        cout << "Other symbols = " << counterOTHERS << endl ;
        return 0 ;
}
```

Программа (листинг 9.9) вычисляет количество прописных символов `A`, `B`, `C` и строчных символов `a`, `b`, `c`, которые введены пользователем, а также количество всех остальных символов. В программе используются три переменные:

- `c` — для хранения значения символа;
- `counterABC` — для хранения числа первых трех букв латинского алфавита;
- `counterOTHERS` — для хранения количества всех остальных символов.

До начала цикла `while` происходит объявление и инициализация переменных. Инициализации переменной `c` значением символа пробела позволяет начать выполнение цикла, который завершится в тот момент, когда переменная `c` получает значение символа точки. Начинается цикл с оператора ввода значения переменной `c`, за которым следует оператор `switch`.

Оператор-переключатель вложен в оператор цикла. Инкремент счетчиков количества тех или иных введенных символов происходит в зависимости от значения переменной `c`.

Первые шесть меток `case` (см. листинг 9.9) приводят к исполнению одного и того же выражения `++counterABC`. Оператор `break` в этой метке необходим для выхода из оператора-переключателя, после чего начинается следующая итерация цикла.

Метка `.` не поддерживает никаких вычислений в программе, а оператор `break` передает управление оператору `while`, который не будет выполняться, так как условие выполнения цикла в данном случае является ложным.

Благодаря метке `default` подсчитывается количество символов, отличных от `A`, `B`, `C`.

После завершения цикла выводятся результаты вычислений, которые демонстрируются на рис. 9.8.

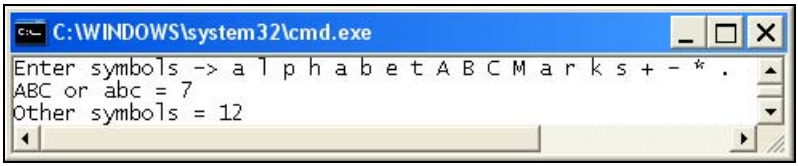


Рис. 9.8. Результаты использования оператора `switch`

Оператор возврата *return*

Форматы оператора:

```
return ;
return выражение ;
```

В первом случае оператор прекращает выполнение текущей функции и возвращает управление вызвавшей функции.

Во втором случае управление вызвавшей функции передается с передачей значения выражения (листинг 9.10).

Функции рассматриваются в книге несколько позднее.

Примечание

Если тип функции не является типом `void`, то в теле такой функции обязательно должен присутствовать оператор `return`.

Листинг 9.10. Пример оператора `return`

```
return ( x + y ) ;    // возврат значения x + y в вызвавшую функцию
```

Тернарный оператор *?:*

Формат оператора:

```
выражение1 ? выражение2 : выражение3 ;
```

Оператор `?:` называется тернарным, так как имеет три операнда. Этот оператор представляет собой компактную форму условного оператора `if else`. Сначала вычисляется `выражение1`. Если оно истинно, вычисляется `выражение2`, и полученное значение становится результатом всего оператора. Если `выражение1` ложно, то тогда вычисляется `выражение3`, и его значение становится результатом оператора (листинг 9.11).

Примечание

В операторе `?:` на месте выражений можно использовать операторы вызова функций. Когда в операторе встречается имя функции, происходит ее вызов, после чего на месте соответствующего выражения используется результат работы функции.

Листинг 9.11. Пример оператора `? :`

```
j = ( i < 0 ) ? ( -i ) : ( i ) ;
max = ( a > b ) ? a : b ;
```

Первый оператор вычисляет абсолютное значение числа и записывает его в переменную j. Если значение переменной i является отрицательным, то происходит смена знака, и переменная j получает значение -i. В противном случае — значение i.

Второй оператор выполняет выбор наибольшего значения из двух. Если a больше b, то переменной max присваивается значение переменной a. Иначе — значение переменной b.

Оператор *sizeof*

Формат оператора:

```
sizeof  (выражение ) ;
```

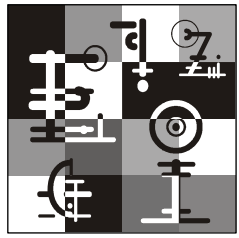
Оператор является статическим и вычисляет длину операнда в байтах. Результат оператора рассматривается как константа целого типа и никогда не бывает равным нулю. Данный оператор позволяет программе автоматически определять размеры встроенных типов, что делает программу независимой от компьютера (листинг 9.12).

Листинг 9.12. Пример оператора `sizeof`

```
cout << sizeof ( double ) << endl ;
int j ; cout << sizeof ( j ) ;
```

В первой строке выводится фактическая длина встроенного типа double, которая определена при помощи оператора sizeof.

Во второй строке определяется и выводится фактическая длина в байтах, занимаемая целочисленной переменной j.



Глава 10

Массивы

Массив (array) является производным типом данных и представляет собой последовательность однотипных данных, имеющих единый идентификатор и хранящихся в смежных ячейках памяти.

Количество элементов в массиве называют *размером* массива.

Необходимое для хранения массива количество байтов оперативной памяти зависит от его типа, а также размера и вычисляется по следующей формуле:

```
число_байтов  = sizeof ( тип_массива ) * число_элементов
```

Размер массива вычисляется как частное от деления количества байтов, занимаемого массивом, на количество байтов, которое занимает данное этого типа. Формула вычисления имеет вид:

```
число_элементов  = sizeof ( имя_массива ) / sizeof ( тип_массива  )
```

Массивы могут быть одномерными и многомерными. Доступ к элементам массива организуется с помощью индексов.

Количество индексов, необходимых для однозначной идентификации любого элемента массива, называют *размерностью* массива.

Операции над массивами

Для массивов определена единственная операция — индексация, при помощи которой происходит обращение к элементам массива. Оператор индексации обозначается квадратными скобками. Внутри квадратных скобок записывается выражение целого типа, задающее индекс элемента.

В зависимости от размерности массива для элемента массива указывается один или несколько индексов, каждый из которых заключается в квадратные скобки. Значение индекса должно находиться в диапазоне от 0 до размера

массива по указанному индексу минус 1. Если значение индекса массива лежит вне этого диапазона, происходит выход за пределы массива, вызывающий ошибку исполнения. Автор программы должен сам позаботиться о том, чтобы индексы оставались внутри своих пределов (листинг 10.1).

Листинг 10.1. Пример индексации элементов массива

```
array [ 2 ] ;          // третий элемент массива array
matrix [ 1 ] [ 4 ] ;   // пятый элемент второй строки матрицы matrix
b [ i ] [ j ] ;        // (i + 1)-ый элемент (j + 1)-ой строки матрицы b
a [ i + 2 ] ;          // (i + 3)-ий элемент массива a
```

В примере показано обозначение разных элементов одномерных и двумерных массивов.

Одномерные массивы

Каждому элементу одномерного массива соответствует один индекс — целое неотрицательное число или выражение, которое определяет позицию элемента в последовательности. Первому элементу массива соответствует нулевой индекс.

Объявление одномерного массива имеет вид:

```
тип идентификатор [ размер ] ;
```

Обращение к элементу одномерного массива производится с помощью оператора индексации, в котором указывается один индекс (листинг 10.2).

Листнng 10.2. Пример ввода/вывода одномерного массива простого типа

```
#include <iostream>
using namespace std ;
const int N = 10 ;
int main ( )
{
        // объявление массива из 10 элементов целого типа
        int array [ N ] ;
        // цикл ввода элементов массива
        for ( int j = 0; j < N; j++ )
        {
                cout << "Enter array [" << j << "] -> " ;
```

```
            cin >> array [ j ] ;
        }
        // вывод подсказки
        cout << "\n\nThe entered array\n" ;
        // цикл вывода элементов массива
        for ( int j = 0; j < N; j++ )
            cout << array [ j ] << '\t' ;
        cout << endl ;
        return 0 ;
}
```

Пример демонстрирует ввод и вывод 10 элементов целочисленного массива. Размер массива задан глобальной константой целого типа

`const int N = 10.`

В начале программы объявляется массив целых чисел `array`, состоящий из 10 элементов. Индекс элементов массива может изменяться от 0 до 9.

Для ввода всех элементов массива организован цикл, который позволяет ввести каждый элемент, изменяя индекс. В качестве индекса массива в операторе `for` используется переменная `j`, которая инициализируется нулевым значением, так как первый элемент имеет индекс 0. Цикл начинается с проверки условия `j < N` и заканчивается, когда переменная `j` принимает значение, равное `N`. Внутри цикла сначала выводится подсказка с указанием значения индекса вводимого элемента, а затем вводится сам элемент `cin >> array [ j ]`. Последний оператор `j++` цикла `for` увеличивает значение индекса `j` на 1.

После завершения цикла выводится сообщение, и в новом цикле `for` выводятся значения, которые элементы массива получили при вводе. Для отделения элементов массива друг от друга при выводе используется управляющая последовательность `\t`.

На рис. 10.1 показан результат работы программы.

Примечание

При выводе элементов массива для отделения одного элемента от другого необходимо использовать какой-либо разделитель (например, `'\t'`), иначе все значения соединятся в одну сплошную линию, и невозможно будет определить, где начинается и где заканчивается каждое значение.

Элементам массива можно непосредственно задать значения внутри любой функции программы. Для инициализации массива вместе с его объявлением достаточно указать тип, идентификатор, после которого в квадратных скоб-

ках записать количество элементов массива, и, воспользовавшись оператором присваивания, в фигурных скобках перечислить значения элементов через запятую (листинг 10.3).

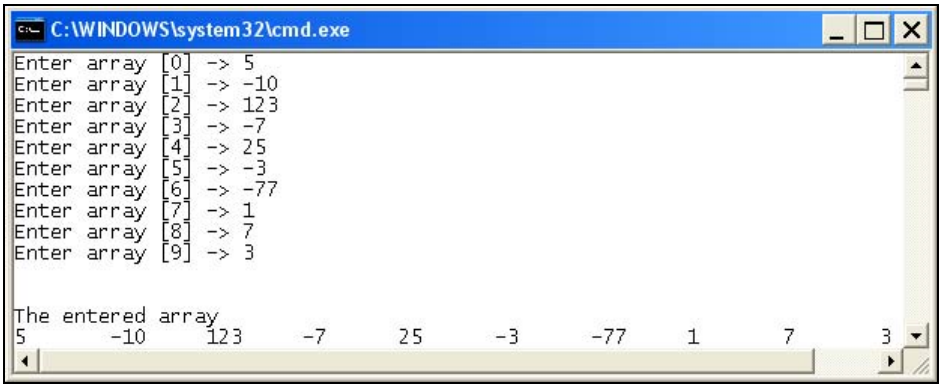


Рис. 10.1. Результат ввода одномерного массива

Листинг 10.3. Пример инициализации одномерного массива простого типа

```
int m [ 3 ] = { 1, 10, 100 } ;
unsigned short x [ 5 ] = { 3, 5 } ;
double y [ ] = { 1.5, 7.38, -8.9, 0.0, 10.5 } ;
char c [ 4 ] = { 'a', 'b', 'c', '\0' } ;
```

Примечание

В конце оператора объявления, совмещенного с инициализацией, после закрывающей фигурной скобки обязательно должна стоять точка с запятой.

В примере массив `m` состоит из трех целых чисел со значениями 1, 10 и 100.

Количество значений в списке может быть меньше объявленного. В этом случае все недостающие элементы инициализируются нулевым значением. Так, в массиве коротких целых чисел без знака с идентификатором `x` будет пять элементов со значениями 3, 5, 0, 0, 0.

Количество элементов инициализируемого массива может вообще не указываться. В примере это соответствует массиву `y`, который содержит 5 чисел с плавающей точкой с указанными в списке значениями. Размер массива вычисляется компилятором автоматически.

В последней строке листинга 10.3 инициализируется символьный массив, последним элементом которого является символ завершения строки \0, которым в C++ заканчиваются все строки.

Примечание

Пустые квадратные скобки не допустимы при обычном объявлении массива без его инициализации.

Листинг 10.4. Пример поиска минимального элемента в одномерном массиве простого типа

```
#include <iostream>
using namespace std ;
int main ( )
{
        // объявление и инициализация массива array
        int array [ ] = { 5, -10, 123, -7, 25, -3, -77, 1, 7, 3 } ;
        // вычисление size — количества элементов в массиве
        int size = sizeof ( array ) / sizeof ( array [ 0 ] ) ;
        // предположение, что минимальный элемент имеет индекс 0
        int jMin = 0 ;
        // цикл проверки элементов массива
        for ( int j = 1; j < size; j++ )
                if ( array [ j ] < array [ jMin ] )
                        jMin = j ;
        // вывод массива
        cout << "Elements of array\n" ;
        for ( int j = 0; j < size; j++ )
                cout << array [ j ] << '\t' ;
        // вывод минимального значения array [ jMin ]
        cout << "\nMinimum = " << array [ jMin ] << endl ;
        return 0 ;
}
```

В программе происходит поиск не самого значения минимума, а его индекса (листинг 10.4). До начала цикла for делается предположение, что наименьшее значение имеет самый первый элемент в массиве, поэтому индекс jMin получает значение 0. Цикл проверки элементов управляет изменением индекса j, увеличивая его на 1 в конце каждой итерации. В теле цикла сравниваются

значения текущего элемента с индексом `j` и элемента с индексом `jMin`. Если текущий элемент меньше, то с помощью выражения `jMin = j` корректируется индекс минимального элемента массива. В противном случае корректировка не производится. После завершения цикла в переменной `jMin` будет находиться индекс наименьшего элемента массива. В конце программы сначала выводится массив, а затем найденное значение `array [ jMin ]`, где записан минимальный элемент. На рис. 10.2 представлен результат работы программы.

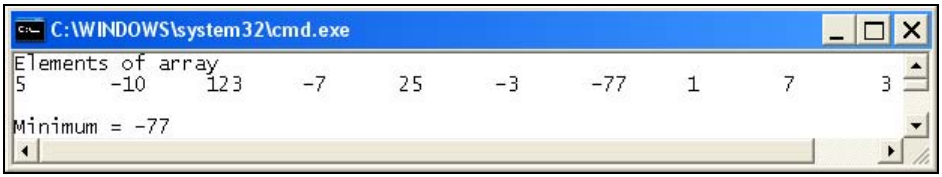


Рис. 10.2. Результат программы поиска минимального элемента в одномерном массиве

Листинг 10.5. Пример вычисления количества элементов из заданного диапазона

```
#include <iostream>
using namespace std ;
int main ( )
{
// объявление и инициализация массива array
       double array [ ] = { 1.1, -5.25, 1.2, -2., -3.4, 2.5, 2.75 } ;
// объявление и инициализация верхней t и нижней b границ диапазона
       double t = -3.5, b = 2.5 ;
// вычисление size — количества элементов в массиве
       int size = sizeof ( array ) / sizeof ( double ) ;
// объявление числа входящих в диапазон элементов n и индекса i
       int n, i ;
// цикл вычисления количества входящих в диапазон элементов
       for ( n = 0, i = 0; i < size; i++ )
              if ( array [ i ] >= t && array [ i ] <= b )
                     ++n ;
// вывод результатов вычислений
       cout << "Is in range " << n << " elements\n" ;
       cout << "No in range " << ( size - n ) << " elements\n" ;
       return 0 ;
}
```

В начале программы вычисляется размер массива `size` при помощи оператора `sizeof` (листинг 10.5). Для вычисления количества элементов, входящих в заданный диапазон, переменная `n` перед началом цикла получает значение 0. Каждая итерация цикла проверяет вхождение очередного элемента массива `array [ i ]` в диапазон `(t,b)`. Если выражение в операторе `if` оказывается истинным, происходит увеличение `n` на 1. Итерации заканчиваются, когда индекс `i` достигает значения `size`. Программа выводит не только значение переменной `n`, но и количество элементов, которые не попадают в заданный диапазон. Результатом работы программы будут следующие две строки:

```
Is in range 5 elements
No in range 2 elements
```

Листинг 10.6. Пример вычисления среднего арифметического значения элементов массива

```
#include <iostream>
using namespace std ;
int main ( )
{
        const int SIZE = 6 ;
        double array [ SIZE ] ;
        cout << "Enter elements through a space -> " ;
        // ввод элементов
        for ( int j = 0; j < SIZE; j++ )
                cin >> array [ j ] ;
        // вычисление суммы элементов
        double total = 0.0 ;
        for ( int j = 0; j < SIZE; j++ )
                total += array [ j ] ;
        // вычисление и вывод среднего арифметического
        double average = total / SIZE ;
        cout << "Average: " << average << endl ;
        return 0 ;
}
```

Количество элементов массива (листинг 10.6) задается константой `SIZE`. Первый цикл `for` предназначен для ввода элементов. Второй — для вычисления суммы `total` всех элементов массива. До начала цикла переменная `total` инициализируется нулевым значением. Переменной `average` присваивается

среднее арифметическое значение элементов массива, вычисляемое как `total / SIZE`. Результаты работы программы показаны на рис. 10.3.

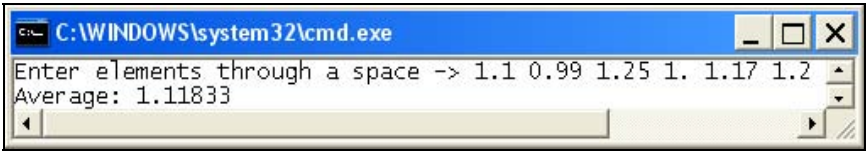


Рис. 10.3. Результаты программы вычисления среднего арифметического значения

Многомерные массивы

Массивы в языке C++ могут иметь не один, а несколько индексов. Оператор объявления многомерного массива имеет вид:

```
тип идентификатор [ размер1 ] [ размер2 ] … [ размерN ] ;
```

где `размерi` — это целое неотрицательное число или выражение, определяющее максимальное значение `i`-го индекса массива.

Количество элементов многомерного массива определяется произведением максимальных значений индексов. Например, оператор

```
int array [ 3 ] [ 4 ] ;
```

объявляет массив из трех элементов целого типа, каждый из которых является массивом из четырех элементов целого типа. Размер массива равен 12.

Размер массива можно вычислить как частное от деления количества байтов, занимаемого массивом, на количество байтов, которое занимает его первый элемент. Формула вычисления имеет вид:

```
число_элементов = sizeof ( идентификатор ) /sizeof ( идентификатор [ 0 ])
```

Для определения количества байтов, которое занимает массив в оперативной памяти, можно воспользоваться следующей формулой:

```
число_байтов  = sizeof ( тип_массива ) * число_элементов
```

Обращение к элементу многомерного массива производится точно так же, как и к элементу одномерного массива, с помощью *оператора индексации*. Внутри квадратных скобок записывается выражение целого типа, задающее индексы элемента. Все индексы первого элемента равны нулю.

Наибольшее применение в программах, помимо одномерных массивов, получили двумерные, которые часто называют матрицами или таблицами. Первый индекс определяет количество строк в матрице, второй — количество столбцов. В памяти компьютера элементы располагаются построчно. Для обра-

щения к элементу матрицы указываются два индекса. Например, `array [ 1 ] [ 3 ]` обозначает четвертый элемент второй строки матрицы.

Массивы, имеющие больше трех индексов, используются крайне редко.

Листинг 10.7. Пример ввода матрицы

```
#include <iostream>
using namespace std ;
int main ( )
{
        // объявление максимального значения индексов матрицы
        const int ROW = 5 ;
        const int COLUMN = 10 ;
        // объявление матрицы вещественных чисел из 5 строк и 10 столбцов
        double matrix [ ROW ] [ COLUMN ] ;
        // ввод элементов матрицы
        // цикл по строкам матрицы
        for ( int i = 0; i < ROW; i++ )
                // цикл по столбцам матрицы
                for ( int j = 0; j < COLUMN; j++ )
                        cin >> matrix [ i ] [ j ] ;
        return 0 ;
}
```

Матрица — это двумерный массив, поэтому для ввода каждого элемента необходимы два цикла, которые вкладываются друг в друга (листинг 10.7). Когда циклы вкладываются друг в друга, первым выполняется самый внутренний цикл. Он будет инициироваться столько раз, сколько выполняется внешний цикл.

В примере внешний цикл с переменной `i` управляет изменением индекса строки, внутренний цикл с переменной `j` изменяет индекс столбца. Каждый цикл имеет собственную инициализацию, проверку и обновление. Внешний цикл повторяет внутренний цикл пять раз и не производит никакой обработки. Во внутреннем цикле происходит ввод значения элемента матрицы с индексами `i`, `j`.

Во время работы программы необходимо ввести 5 строк вещественных чисел по 10 элементов в каждой строке. Действия этой программы неясны пользователю, так как ничего не выводится на экран монитора. Циклы ввода матрицы лучше записать следующим образом (листинг 10.8).

Листинг 10.8. Ввод матрицы по строкам

```
for ( int i = 0; i < ROW; i++ )              // цикл по строкам матрицы
{
       cout << "Enter 10 elements of row " << ( i + 1 ) << " -> " ;
       for ( int j = 0; j < COLUMN; j++ )  // цикл по столбцам матрицы
              cin >> matrix [ i ] [ j ] ;
}
```

Здесь во внешнем цикле сначала выводится подсказка о вводе 10 элементов очередной строки матрицы. Обратите внимание, выводится не сам индекс `i`, а его увеличенное на 1 значение. Это объясняется тем, что индексы в C++ начинаются от 0, а в реальной жизни — от 1.

Листинг 10.9. Ввод матрицы по столбцам

```
for ( int j = 0; j < COLUMN; j++ )           // цикл по столбцам матрицы
{
       cout << "Enter 5 elements of column " << ( j + 1 ) << " -> " ;
       for ( int i = 0; i < ROW; i++ )      // цикл по строкам матрицы
              cin >> matrix [ i ] [ j ] ;
}
```

Результат работы программы ввода матрицы по столбцам (листинг 10.9) будет таким же, как и при вводе матрицы по строкам. Разница только в том, что внутренний цикл заставляет быстрее меняться индекс строки, а внешний — управляет индексом столбца матрицы и повторяет внутренний цикл 10 раз.

Листинг 10.10. Пример инициализации матрицы

```
int a1 [ 2 ] [ 3 ]    = { 1, -2, 2, 3, 1, -2 } ;
int a2 [  ] [ 3 ]     = { { 1, -2, 2 }, { 3, 1, -2 } } ;
int b1 [ ] [ 3 ]      = { { 1, 4, 2 }, { -6, 5, -9 }, { -3, 6, -5 } } ;
int b2 [ 3 ] [ 3 ]    = { { 1, 4, 2 }, { 6, 5, -9 } } ;
int c [ ] [ 2 ]       = { 1, 1, 2, 2, 3, 3, 4 } ;
double f [ 5 ] [ 5 ] = { 0.0 } ;
// определение количества строк матрицы c
int row = sizeof ( c ) / ( ( sizeof ( int ) ) * 2 ) ;
```

Если массив небольшой и заранее известен, значения элементов задаются при объявлении. Инициализация матрицы при объявлении может быть организована в программе разными способами (листинг 10.10).

Матрицы `a1` и `a2` инициализируются по-разному, но являются одинаковыми по значениям элементов. Для инициализации матрицы `a1` элементы перечисляются построчно внутри фигурных скобок через запятую, при этом максимальные значения в объявлении указаны для обоих индексов. Матрица `a1` имеет 2 строки по 3 элемента в каждой и имеет вид:

```
1       -2      2
3       1       -2
```

Матрица `a2` имеет точно такой же вид, однако ее инициализация выполнена иначе. Компилятору для выполнения индексации многомерного массива требуется указание всех размеров, кроме первого, поэтому на месте максимального значения первого индекса ничего не записано. Кроме того, внутри фигурных скобок каждая строка матрицы `a2` заключена в собственные фигурные скобки.

Матрица `b1` состоит из 9 элементов и имеет вид:

```
1       4       2
-6      5       -9
-3      6       -5
```

Матрица `b2` объявляется из 9 элементов с явным указанием двух размеров, однако в списке значений указаны элементы только первых двух строк. В подобных случаях компилятор инициализирует недостающие элементы нулевыми значениями. Матрица `b2` в результате инициализации будет такой:

```
1       4       2
-6      5       -9
0       0       0
```

Матрица `c` объявляется состоящей из двух столбцов. В списке значений указано 7 элементов. Для вычисления количества строк `row` матрицы `c` используется формула, которая записана в последней строке примера (см. листинг 10.10). В результате вычисления оператор `sizeof ( c )` вернет значение 32, а оператор

```
sizeof ( int )
```

значение 4. Эти значения соответствуют числу байтов, которые занимают массив `c` и базовый тип массива `int` соответственно. Так как матрица имеет 2 столбца, в знаменателе дроби число байтов базового типа `sizeof ( int )` умножается на число 2, и переменная `row` получает значение 4.

Последний элемент матрицы `c [ 3 ] [ 1 ]` инициализируется нулем, поскольку в списке его значение не указано. В результате инициализации матрица `c` примет следующий вид:

```
1       1
2       2
3       3
4       0
```

Матрица `f` чисел с плавающей точкой содержит 25 элементов. В каждой из 5 строк матрицы присутствует 5 элементов. Все элементы матрицы инициализируются нулевым значением.

Примечание

При объявлении многомерного массива без инициализации нельзя записывать пустые квадратные скобки, так как компилятор не сможет определить необходимый для массива объем памяти. В конце оператора объявления обязательно должна стоять точка с запятой.

Листинг 10.11. Пример вывода матрицы по строкам

```
#include <iostream>
using namespace std ;
int main ( )
{
        // объявление максимального значения индексов матрицы
        const int R = 2, C = 3 ;
        // объявление и инициализация матрицы из 2 строк и 3 столбцов
        double matrix [ R ] [ C ] = { -1.5, 20.85, 2., 3.17, 1.5, -2.8 }
;
        // вывод матрицы
        // цикл по строкам
        for ( int i = 0; i < R; i++ )
        {
                // цикл по столбцам
                for ( int j = 0; j < C; j++ )
                        cout << matrix [ i ] [ j ] << '\t' ;
                cout << endl ;
        }
        return 0 ;
}
```

Константы R и C в листинге 10.11 определяют максимально допустимые значения индексов строк и столбцов матрицы. Далее объявляется матрица чисел с плавающей точкой matrix, у которой 2 строки и 3 столбца, согласно заданным значениям констант. Одновременно с объявлением происходит инициализация элементов матрицы.

Для вывода матрицы по строкам используются вложенные циклы. Внешний цикл for управляет изменением индекса строк i и включает составной оператор, заключенный в фигурные скобки. Этот оператор содержит внутренний цикл и манипулятор вывода endl. Сначала внутренний цикл, который использует индекс i из внешнего цикла, выводит элементы очередной строки матрицы. Каждая итерация внешнего цикла заставляет меняться индекс столбца j от 0 до C – 1. После завершения внутреннего цикла манипулятор вывода endl обеспечит переход на новую строку вывода так, что следующая строка матрицы начнется с новой строки экрана монитора.

Примечание

При выводе элементов массива следует использовать какой-либо разделитель (например, '\t'), иначе все значения будут выведены слитно и невозможно будет определить, где начинается и где заканчивается каждое значение.

Для вывода многомерного массива во внутреннем цикле необходимо использовать манипулятор endl, в противном случае все значения будут выведены в одну строку, следовательно, возникнет проблема чтения состояния массива.

Листинг 10.12. Пример заполнения матрицы псевдослучайными числами

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std ;
int main ( )
{
        const int SIZE = 5 ;
        // объявление матрицы из 5 строк и 5 столбцов
        int array [ SIZE ] [ SIZE ] ;
        // вычисление стартовой точки при генерации псевдослучайных чи-
сел,
        // возвращаемых функцией rand ( )
        srand( ( unsigned ) time( NULL ) ) ;
        // цикл по строкам матрицы
```

```
        for ( int i = 0; i < SIZE; i++ )
                // цикл по столбцам матрицы
                for ( int j = 0; j < SIZE; j++ )
                        // присваивание элементу матрицы целого числа
                        array [ i ] [ j ] = rand ( ) ;
        // вывод матрицы
        for ( int i = 0; i < SIZE; i++ )
        {
                for ( int j = 0; j < SIZE; j++ )
                        cout << array [ i ] [ j ] << '\t' ;
                cout << endl ;
        }
        return 0 ;
}
```

Для генерации последовательности псевдослучайных чисел (листинг 10.12) используются функции `rand ( )`, `srand ( )` и `time ( )`. Функция `time ( )` возвращает текущее календарное время, установленное операционной системой. Однако если системное время не задано (как в рассматриваемом примере), функция возвращает число –1. Функция `srand ( )` используется для вычисления стартовой точки при генерации последовательности псевдослучайных чисел, возвращаемых функцией `rand ( )`. Каждый раз при вызове функции `rand ( )` возвращается целое число из диапазона от нуля до `0x7fff` (величина `RAND_MAX`). Применение данных функций требует включения заголовочного файла `cstdlib`, где определены `rand ( )` и `srand ( )`, а также файла `ctime`, в котором определена функция `time ( )`. Заголовочные файлы включаются в программу с помощью директивы препроцессора `#include`.

Константа `SIZE` определяет максимально допустимое значение индексов матрицы, которое используется в объявлении матрицы целых чисел `array`. Матрица имеет одинаковое количество строк и столбцов.

Для обработки каждого элемента матрицы использованы два цикла, которые вкладываются друг в друга. Внешний цикл с переменной `i` управляет изменением индекса строки, внутренний цикл с переменной `j` изменяет индекс столбца. Внешний цикл многократно повторяет внутренний цикл и не производит никакой обработки. Во внутреннем цикле происходит присваивание элементам матрицы `array [ i ] [ j ]` значений, которые возвращаются вызовом функции `rand ( )`.

Вывод матрицы организован также с использованием вложенных циклов. Внешний цикл представляет собой составной оператор, который содержит

оператор внутреннего цикла для вывода строки и оператор вывода манипулятора `endl`, благодаря которому каждая строка матрицы отображается на экране монитора в отдельной строке. На рис. 10.4 приведен результат работы программы.

Примечание

Вследствие использования функций для генерации псевдослучайных чисел каждый новый вызов программы на исполнение будет создавать новую последовательность, и массив каждый раз будет заполняться разными значениями.

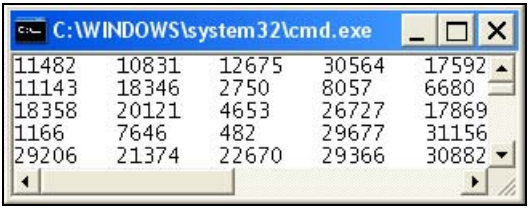


Рис. 10.4. Результат программы заполнения матрицы случайными числами

Листинг 10.13. Пример вычисления сумм столбцов матрицы

```
#include <iostream>
using namespace std ;
int main ( )
{
        const unsigned short SIZE = 3 ;
        // инициализация исходной матрицы
        short matrix [ ][ SIZE ] = { 1, 4, 2, -6, 5, -9, -3, 6, -5 } ;
        // объявление результирующего массива
        short res [ SIZE ] ;
        // цикл по столбцам
        for ( int j = 0; j < SIZE; j++ )
        {
                // подготовка суммирования в столбце
                res [ j ] = 0 ;
                // цикл по строкам
                for ( int i = 0; i < SIZE; i++ )
                        res [ j ] += matrix [ i ] [ j ] ;
        }
```

```
        // вывод результата
        for ( int i = 0; i < SIZE; i++ )
                cout << res [ i ] << ' ' ;
        cout << endl ;
        return 0 ;
}
```

Представленная листингом 10.13 программа вычисляет сумму элементов для каждого столбца матрицы `matrix`, имеющей 3 строки и 3 столбца. Результат вычислений записывается в одномерный массив `res`. Размер массива совпадает с количеством столбцов исходной матрицы.

Как обычно, для работы с двумерным массивом используются вложенные циклы. В примере внешний цикл управляет изменением индекса столбца `j`, так как индекс строки `i` должен изменяться быстрее. Индекс `j` используется и для доступа к элементам одномерного массива `res`, где сохраняется результат вычислений. Перед началом внутреннего цикла элемент массива `res [ j ]` обнуляется для возможности накопления в нем суммы значений всех элементов столбца. В теле внутреннего цикла происходит обращение к элементам, которые имеют второй индекс `j`, а первый индекс `i` изменяется под управлением внутреннего цикла.

В итоге вычислений элементы массива `res` получат следующие значения:

```
res [ 0 ] = -8, res [ 1 ] = 15, res [ 2 ] = -12
```

Для вывода результата в конце программы организован цикл. В качестве разделителя между элементами массива `res` используется символ пробела.

Листинг 10.14. Пример умножения матриц

```
#include <iostream>
using namespace std ;
int main ( )
{
        const int ROW = 2, COL = 3 ;
        // инициализация исходных матриц
        int a [ ROW ][ COL ] = { 1, -2, 2, 3, 1, -2 } ;
        int b [ ][ COL ] = { { 1, 4, 2 }, { -6, 5, -9 }, { -3, 6, -5 } } ;
        // объявление результирующей матрицы
        int res [ ROW ][ COL ] ;
        // умножение матриц a и b
        // цикл по строкам матриц a и res
        for ( int i = 0; i < ROW; i++ )
```

```
            // цикл по столбцам матриц b и res
            for ( int k = 0; k < COL; k++ )
            {
                  // подготовка элемента результирующей матрицы
                  res [ i ][ k ] = 0 ;
                  // цикл для накопления суммы произведений
                  for ( int j = 0; j < COL; j++ )
                        res [ i ][ k ] += a [ i ][ j ] * b [ j ][ k ] ;
            }
      // вывод результирующей матрицы res
      for ( int i = 0; i < ROW; i++ )
      {
            for ( int j = 0; j < COL; j++ )
                  cout << res [ i ][ j ] << '\t' ;
            cout << endl ;
      }
      return 0 ;
}
```

Умножение двух матриц `a` и `b` производится по правилам умножения матриц, принятым в математике (листинг 10.14). В программе для реализации умножения используются три вложенных цикла.

Самый внешний цикл, который выполняется один раз, управляет изменением индекса `i` строк исходной матрицы `a` и строк результирующей матрицы `res`.

Вложенный в него цикл выполняется два раза (по количеству строк матрицы) и управляет изменением индекса `k` столбцов исходной матрицы `b` и результирующей `res`.

Самый внутренний цикл, управляющий индексом `j`, предназначен для вычисления элемента матрицы `res`. Этот цикл выполняется шесть раз (по количеству элементов в результирующей матрице). Перед входом во внутренний цикл элемент `res [ i ] [ k ]` приравнивается нулю, чтобы стало возможным накапливать в нем сумму произведений элементов `a [ i ] [ j ]` и `b [ j ] [ k ]`.

Последний цикл программы выводит вычисленную матрицу `res` по строкам в виде:

```
7       6       10
3       5       7
```

Символьные массивы

Одномерный символьный массив представляет собой строку символов. В языке C++ существует два вида строк:

- строки, завершающиеся нулевым байтом (null-terminated string). Последний элемент такой строки — символ завершения строки `'\0'`;
- строки типа `string`, реализующего объектно-ориентированный подход к программированию. Строки такого типа рассматриваются в *главе 14*.

Строки первого вида требуют при объявлении массива дополнительную ячейку памяти для нулевого байта. Это значит, что размер символьного массива должен быть на единицу больше длины предполагаемой последовательности символов. Например, для объявления строки *s*, содержащей не более 50 символов, потребуется следующий оператор:

```
char s [ 51 ] ;
```

Строка, заключенная в кавычки, называется строковой константой (string constant). В конец строковой константы компилятор автоматически дописывает нулевой байт. Например, `"September"` — это строковая константа, которая занимает в оперативной памяти 10 байтов.

При вводе строковых переменных нулевой байт записывается в конец тоже автоматически. Если же формирование строки происходит в программе, ее автор должен самостоятельно позаботиться о наличии нулевого байта.

Листинг 10.15. Пример удаления всех вхождений символа из строки

```
#include <iostream>
using namespace std ;
int main ( )
{
       char s [ ] = "Taraauaaaamaaaaaraaaaaa!" ;  // исходная строка
       char c = 'a' ;                             // удаляемый символ
       cout << "String =\t" << s << endl ;         // вывод строки
       // удаление из строки s всех вхождений символа 'a'
       int i = 0 ;                  // индекс для исходной строки
       int j = i ;                  // индекс для модифицированной строки
       // цикл модификации исходной строки
       while ( s [ i ] )            // пока в строке не нулевой байт
       {
```

```
            // проверить текущий символ строки
            if ( s [ i ] != c )   // если s [ i ] не равен 'a'
                  // переписать текущий символ по новому индексу j
                  // и увеличить индекс j на единицу
                  s [ j++ ] = s [ i ] ;
            // увеличить индекс i на единицу
            i++ ;
      }
      // запись нулевого байта в конец модифицированной строки
      s [ j ] = '\0' ;
      // вывод результата
      cout << "\nResult =\t" << s << endl ;
      return 0 ;
}
```

Результат работы программы (листинг 10.15) представлен на рис. 10.5.

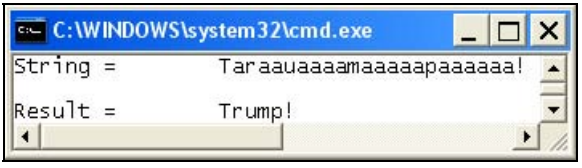


Рис. 10.5. Результат исполнения программы удаления всех вхождений символа из строки

Программа изменяет состояние исходной строки таким образом, что в ней не остается ни одного вхождения заданного символа.

Модификация исходной строки происходит в цикле, который управляет изменением индекса `i` для обращения к элементу строки. Инициализация индекса нулевым значением происходит до начала итераций. Оператор `j = i` определяет значение индекса `j`, который используется для новой нумерации элементов в строке. Если проверяемый элемент строки `s [ i ]` не совпадает с удаляемым символом `c`, происходит запись этого символа по индексу `j`. После записи элемента на новое место индекс `j` увеличивается на 1 для подготовки к записи следующего элемента модифицированной строки. Как только в цикле `while` проверка элемента строки *s* `[ i ]` обнаружит нулевой байт, цикл закончится. Для окончательного формирования нового состояния строки по индексу `j` записывается символ завершения строки `'\0'`.

Листинг 10.16. Пример удаления символов цифр из строки

```
#include <iostream>
using namespace std ;
int main ( )
{
        // исходная строка
        char s [ ] = "0Th123ere a45re no777 d9999igits." ;
        int i, j ;
        // цикл модификации строки
        for ( j = 0, i = 0; s [ i ]; i++ )
                if ( ! ( s [ i ] >= '0' && s [ i ] <= '9' ) )
                        s [ j++ ] = s [ i ] ;
        // запись символа конца строки
        s [ j ] = '\0' ;
        cout << s << endl ;
        return 0 ;
}
```

Данная программа (листинг 10.16) представляет собой модифицированную версию программы предыдущего примера. Из исходной строки удаляются все символы цифры. Вместо оператора цикла `while` здесь используется оператор цикла `for`. Выражение в условном операторе `if` будет истинным тогда, когда анализируемый символ `s [ i ]` не будет входить в диапазон от символа `0` до символа `9`. Результатом работы программы будет строка `There are no digits`.

Листинг 10.17. Пример реверса строки

```
#include <iostream>
using namespace std ;
int main ( )
{
        // исходная строка
        char s [ ] = "dennis margorp" ;
        cout << "String:\t\t" << s << endl ;
        // размер строки без нулевого байта
        int len = sizeof ( s ) - 1 ;
        // индекс i = 0 для перемещения от начала строки
        // индекс j = len - 1  для перемещения от конца строки
        int i = 0, j = len - 1 ;
        // переменная tmp для перестановки элементов
        char tmp ;
```

```
        // цикл реверса строки
        do
        {
                // перестановка символов
                tmp = s [ i ] ;
                s [ i++ ] = s [ j ] ;
                s [ j-- ] = tmp ;
        } while ( i < j ) ;
        // вывод строки после реверса
        cout << "Result:\t\t" << s << endl ;
        return 0 ;
}
```

После объявления и вывода исходной строки (листинг 10.17) вычисляется размер строки в байтах без учета нулевого байта. Вследствие того, что элемент типа `char` занимает в оперативной памяти 1 байт, количество символов в строке совпадает с ее размером. Индекс элемента с нулевым байтом будет равен вычисленному размеру `len`, так как индексация в массиве начинается с 0. Для реверса строки необходимо попарно менять местами символы, имеющие одинаковое смещение от начала и от конца строки. Для организации перестановки используются два индекса `i`, `j` и переменная `tmp`. До начала цикла обработки строки индекс `i` инициализируется значением 0, что соответствует индексу первого элемента, а индекс `j` — значением `len - 1`, которое определяет индекс символа перед нулевым байтом. Переменная `tmp` требуется для перестановки элементов.

В программе (см. листинг 10.17) используется оператор цикла с постусловием `do while`, который управляет изменением индекса `i` для обращения к начальным элементам строки и изменением индекса `j` для обращения к элементам от конца строки. Внутри цикла сначала в переменной `tmp` сохраняется значение символа с индексом `i`, затем на место элемента с этим индексом записывается символ с индексом `j`. Далее совершается инкремент индекса `i`. Наконец, на место символа с индексом `j` записывается значение временно сохраненной переменной `tmp`, после чего индекс `j` декрементируется. Итерации повторяются до тех пор, пока значение `i` меньше значения `j`. По окончании цикла выводится новое состояние строки. Результат работы программы представлен на рис. 10.6.

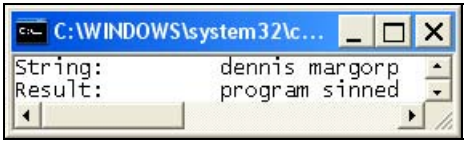
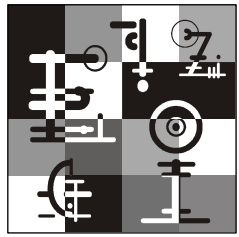


Рис. 10.6. Результат исполнения программы реверса строки



Глава 11

Указатели

Все рассмотренные в предыдущих главах примеры в качестве данных использовали обычные переменные. Для таких переменных память выделяется автоматически при запуске программы, в которой они объявлены, и так же автоматически удаляются при завершении программы. Доступ к значению обычной переменной происходит по ее идентификатору. Язык C++ позволяет обратиться к значению переменной иначе — при помощи адреса, по которому записано значение. Адрес помещается в особую переменную — указатель (pointer).

Указатели относятся к производным типам данных, как и массивы. Применение указателей предоставляет возможность создавать динамические структуры данных, размер которых определяется не при создании программы, а в процессе ее исполнения. На рис. 11.1 проиллюстрирована взаимосвязь между переменной и указателем на нее.

Рис. 11.1. Взаимосвязь между переменной и указателем на нее

Как известно, каждая ячейка оперативной памяти имеет свой уникальный адрес. Переменная, значение которой равно 5, записана в ячейку с адресом 0012FF70. Указатель на эту переменную имеет адрес 0012FF6C, по которому записан адрес переменной 0012FF70.

Указатель (pointer value) — это переменная, значением которой является адрес другой переменной.

Указатель, как любая переменная, должен быть объявлен в программе до его первого использования. Объявление указателя имеет следующие форматы:

```
тип* идентификатор ;
тип *идентификатор ;
тип *идентификатор1, …, *идентификаторN ;
```

Символ звездочки свидетельствует о том, что переменная с указанным идентификатором является указателем.

В одной строке можно объявлять как один указатель, так и несколько указателей одного типа. Если в строке объявляется только один указатель, звездочка может записываться как сразу после типа, так и перед идентификатором переменной. Когда возникает необходимость объявления в одной строке нескольких указателей, звездочка должна располагаться непосредственно перед каждым идентификатором.

Тип указателя обязательно должен совпадать с типом переменной, адрес которой он хранит.

Указатель инициализируется либо при его объявлении, либо с помощью оператора присваивания. Нельзя использовать в программе указатель, значение которого не определено. Допустимый диапазон значений для любого указателя всегда включает специальный адрес 0 и набор положительных целых чисел, которые интерпретируются как машинные адреса.

Указатель может получить в качестве начального значения нуль или адрес. Указатель, получивший в качестве значения 0, ни на что не указывает. При этом не нужно приводить целочисленное значение 0 к типу указателя, так как оно автоматически преобразуется компилятором в указатель соответствующего типа.

Для инициализации указателя нулевым значением следует записать в программе строку следующего вида (листинг 11.1):

```
тип* идентификатор = 0 ;
```

Присваивание адреса переменной указателю производится при помощи операции взятия адреса, которая рассматривается далее.

Листинг 11.1. Пример объявления указателей

```
char* ps ;            // указатель на символьную переменную
float *ptr ;          // указатель на переменную с плавающей точкой
int *px, *py ;        // указатели на целочисленые переменные
// указатель на символьную переменную и символьная переменная
char* p, ch ;
```

В первой строке звездочка расположена сразу после типа, а во второй — перед идентификатором переменной. Несмотря на положение символа в строке, обе переменные ps и ptr являются указателями. В третьей строке показано объявление двух указателей одного типа. Если звездочки не будет перед вторым идентификатором, компилятор воспримет его как имя обычной переменной. В последней строке примера объявляется указатель на символьную переменную с идентификатором p и символьную переменную с идентификатором ch.

Примечание

Идентификатор указателя лучше начинать с символа p, чтобы текст исходного кода программы легче читался.

Листинг 11.2. Пример инициализации указателей нулевым значением

```
char* ps = 0 ;
int *px = 0, *py = 0 ;
char *p = 0 ;
double* pD = 0 ;
```

Все указатели (листинг 11.2) получают значение 0, которое преобразовано компилятором в указатели на переменные символьного типа для ps и p, для указателей px и py — на целочисленные переменные, а для pD — на переменные с плавающей точкой.

Операции с указателями

С указателями можно выполнять операции, аналогичные арифметическим операциям, однако они отличаются от операций с переменными простых типов, так как в качестве операндов используются адреса. В C++ существуют специальные операции, предназначенные для работы с указателями. В табл. 11.1 приведены допустимые операции для указателей.

Таблица 11.1. *Операции с указателями*

Название	Знак	Пояснения
Взятие адреса	&	Получить адрес переменной
Разыменовывание	*	Получить значение переменной по ее адресу
Присваивание	=	Присвоить указателю адрес переменной или 0

Таблица 11.1 *(окончание)*

Название	Знак	Пояснения
Инкремент	`++`	Увеличить указатель на 1. После увеличения указатель будет указывать на следующий элемент массива
Декремент	`--`	Уменьшить указатель на 1. После уменьшения указатель будет указывать на предыдущий элемент массива
Сложение	`+`	Увеличить указатель на целое значение
Сложение с замещением	`+=`	Увеличить существующий указатель на целое значение
Вычитание	`-`	Уменьшить указатель на целое значение или вычесть другой указатель, если оба указателя указывают на элементы одного и того же массива
Вычитание с замещением	`-=`	Уменьшить существующий указатель на целое значение или вычесть другой указатель, если оба указателя указывают на элементы одного и того же массива

Отношения	`==`
	`!=`
	`>`
	`<`
	`>=`
	`<=`
		Сравнить указатели. Возвращается истина, если операция сравнения закончилась успешно, иначе возвращается ложь

Выделение памяти	`new`	Получить указатель на начало выделенного блока памяти
Освобождение памяти	`delete`	Освободить выделенный блок памяти и сделать указатель недоступным

Арифметические операции, связанные с увеличением указателя на целое значение, модифицируют значение указателя на число, соответствующее произведению целого числа на количество занимаемых типом указателя байтов. Инкремент модифицирует указатель не на 1, а на число байтов, которое занимает тип указателя. Арифметические операции, которые уменьшают указатель на целое число, выполняют такую же модификацию, но в сторону уменьшения значения адреса. При работе с указателями программист должен внимательно контролировать их использование, чтобы не произошла ошибка обращения по несуществующему адресу или адресу, значение которого 0.

Листинг 11.3. Пример взятия адреса &

```
#include <iostream>
using namespace std ;
int main ( )
{
        int x = 5 ;
        int * px = & x ;                              // взятие адреса x
        cout << "Value x\t\t" << x << endl ;          // вывод значения x
        cout << "Address x\t" << & x << endl ;        // вывод адреса x
        cout << "Pointer px\t" << px << endl ;        // вывод значения px
        return 0 ;
}
```

В листинге 11.3 объявляется целая переменная `x` и указатель на данные целого типа `px`. При объявлении переменная инициализируется значением 5, а указатель — значением адреса переменной `x`. Для инициализации указателя использована операция взятия адреса `&x`. Результаты работы программы показан на рис. 11.2.

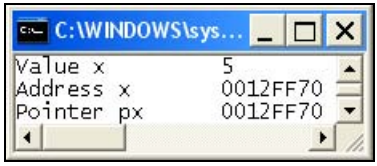


Рис. 11.2. Результаты исполнения программы взятия адреса

Листинг 11.4. Пример разыменовывания указателя *

```
#include <iostream>
using namespace std ;
int main ( )
{
        double v1 = 0.05, v2 = 2.5e32 ;
        double * pv ;                          // объявление указателя
        pv = & v1 ;                            // взятие адреса v1
        cout << "pv =\t" << pv << endl ;       // вывод адреса v1
        cout << "v1 =\t" << v1 << endl ;       // вывод значения v1
```

```
        cout << "*pv = \t" ;
        cout << * pv << endl ;              // вывод значения v1 через pv
        pv = & v2 ;                         // взятие адреса v2
        cout << "pv =\t" << pv << endl ;    // вывод адреса v2
        cout << "v2 =\t" << v2 << endl ;    // вывод значения v2
        cout << "*pv = \t" ;
        cout << * pv << endl ;              // вывод значения v2 через pv
        return 0 ;
}
```

В программе объявляются две переменные с плавающей точкой `v1` и `v2`, а также указатель на тип с плавающей точкой `pv`. Сначала указатель получает значение адреса переменной `v1`. Три следующие строки организуют последовательный вывод указателя `pv`, значения переменной `v1` по ее имени и значения переменной `v1` через разыменовывание указателя `* pv`. Указатель `pv` хранит адрес переменной `v1`, поэтому между выводом `v1` и `* pv` нет никакой разницы. Далее указатель принимает значение адреса переменной `v2`. Хранимый в `pv` адрес и содержимое `* pv` (значение переменной `v2`), которые будут выведены на экран, изменятся. Результаты работы программы представлены на рис. 11.3.

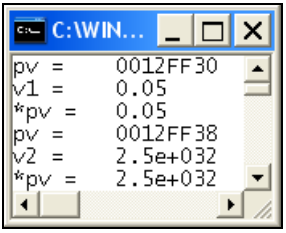


Рис. 11.3. Результат исполнения программы разыменовывания адреса

Указатели и массивы

Доступ к элементам массива можно получить, используя указатель.

Пусть программе известен массив целых чисел `int array [ 10 ]`. Имя массива `array` является его константным адресом, поэтому выражение

`( array + i )`

представляет адрес i-го элемента в массиве `array`. Для того чтобы получить значение элемента массива, необходимо выполнить операцию разыменовывания адреса, то есть записать выражение `* ( array + i )`. Для получения

адреса i-го элемента в массиве следует использовать операцию взятия адреса `& array [ i ]`.

Тип указателя на массив должен совпадать с типом массива, чтобы компилятор мог осуществлять правильный доступ к его элементам. Компилятор при работе с указателем будет умножать значение индекса на величину, равную количеству байтов, которое занимает тип элементов массива. Например, для символьных массивов эта величина будет равна 1, для целых массивов — 2, для вещественных массивов — 8.

Листинг 11.5. Пример вывода элементов одномерного массива с использованием указателя

```
#include <iostream>
using namespace std ;
int main ( )
{
        int array [ ] = { 10, 20, 30, 40, 50 } ;
        int size = sizeof ( array ) / sizeof ( array [ 0 ] ) ;
        // вывод с использованием константного адреса массива
        for ( int i = 0; i < size; i++ )
                cout << * ( array + i ) << '\t' ;
        cout << endl ;
        // вывод с использованием указателя
        int* p = array ;
        int count = 0 ;
        do
        {
                cout << * p++ << '\t' ;
                count++ ;
        } while ( count < size ) ;
        cout << endl ;
        return 0 ;
}
```

Цикл `for` выводит элементы массива `array`, используя его константный адрес, который является указателем (листинг 11.5). Для перемещения к следующему элементу указатель array увеличивается на значение индекса `i`, а затем операция разыменовывания `*` возвращает значение элемента массива с адресом `( array + i )`.

Во второй части программы тот же массив выводится с использованием указателя p, который при объявлении инициализируется адресом array. Цикл do while управляет счетчиком элементов массива count. В цикле для получения значения элемента массива используется указатель p. Сначала берется значение, хранимое по адресу p. Это значение возвращает операция взятия содержимого по адресу * p. После вывода значения элемента указатель инкрементируется p++, делая доступным следующий элемент массива для новой итерации цикла.

Примечание

Оператору инкремент в качестве операнда требуется именующее выражение (l-value). Имя массива является константным адресом, поэтому не подлежит изменению. В связи с этим невозможно использовать имя массива array в операции инкремент. Если же операндом сделать переменную, объявленную как указатель, операция становится возможной. В примере это p++.

Листинг 11.6. Пример вывода строки с использованием указателя

```
#include <iostream>
using namespace std ;
int main ( )
{
        char s [ 20 ] ;
        char* p = s ;
        cout << "-> " ; cin >> p ;
        cout << "\nWizard - pointer!\n\n" ;
        while ( * p )
                cout << p++ << endl ;
        p = s ;
        cout << "\nInitial string -> " << p << endl ;
        return 0 ;
}
```

Приведенный в листинге 11.6 пример наглядно иллюстрирует использование указателя. Строка s, которая вводится в программе cin >> p, не меняется во время исполнения программы. Однако на экран выводятся разные значения, поскольку значение указателя p увеличивается в цикле while на 1, изменяя адрес начала выводимой части строки s. Цикл продолжается до тех пор, пока содержимое по адресу p не станет равным нулевому байту.

После выхода из цикла указатель `p` содержит адрес конца строки, поэтому перед выводом исходной строки через указатель следует присвоить ему значение адреса начала строки `p = s`. На рис. 11.4 показаны результаты выполнения программы.

Рис. 11.4. Результаты выполнения программы вывода строки с использованием указателя

Листинг 11.7. Пример вывода элементов двумерного массива с использованием указателя

```
#include <iostream>
using namespace std ;
int main ( )
{
        const int R = 2, C = 3 ;
        double matrix [ R ] [ C ] = { -1.5, 20.85, 2., 3.17, 1.5, -2.8 } ;
        double* p = & matrix [ 0 ] [ 0 ] ;
        for ( int i = 0; i < R; i++ )
        {
                for ( int j = 0; j < C; j++ )
                        cout << * ( p + i * C + j ) << '\t' ;
                cout << endl ;
        }
        return 0 ;
}
```

В листинге 11.7 объявляется указатель на тип с плавающей точкой `p`, который одновременно с объявлением инициализируется адресом первого элемента матрицы `&matrix [ 0 ] [ 0 ]`. Адрес выводимого элемента матрицы вычисляется при помощи выражения `( p + i * C + j )`. Часть этого выражения `p + i * C` позволяет определить адрес первого элемента в строке `i`, а все выражение целиком — адрес текущего элемента `j` в `i`-ой строке.

Примечание

Для многомерного массива нельзя инициализировать указатель именем массива. Необходимо или выполнить операцию взятия адреса, как это показано в примере, или же воспользоваться оператором приведения типа `reinterpret_cast`. Синтаксис оператора можно увидеть в справочной документации MVC++. В примере вместо выражения `double* p = &matrix [ 0 ] [ 0 ]` следует записать следующее выражение: `double* p = reinterpret_cast < double* > ( matrix )`.

Листинг 11.8. Пример транспонирования квадратной матрицы с помощью указателя

```
#include <iostream>
using namespace std ;
int main ( )
{
        const int N = 3 ;
        int m [ N ] [ N ] = { 1, 3, 4, 2, -1, -2, 7, -5, 8 } ;
        // объявление указателя с инициализацией
        int* pm = & m [ 0 ] [ 0 ] ;
        // вывод начальной матрицы с использованием указателя
        cout << "Initial matrix\n" ;
        for ( int i = 0; i < N; i++ )
        {
                for ( int j = 0; j < N; j++ )
                        cout << * ( pm + i * N + j ) << '\t' ;
                cout << endl ;
        }
        // транспонирование матрицы с использованием указателя
        int tmp ;
        for ( int i = 1; i < N; i++ )
                for ( int j = 0; j < N - 1; j++ )
                {
                        tmp = * ( pm + i * N + j ) ;
```

```
                        * ( pm + i * N + j ) = * ( pm + j * N + i ) ;
                        * ( pm + j * N + i ) = tmp ;
                }
        // вывод транспонированной матрицы обычным способом
        cout << "\nTransposed matrix\n" ;
        for ( int i = 0; i < N; i++ )
        {
                for ( int j = 0; j < N; j++ )
                        cout << m [ i ] [ j ] << '\t' ;
                cout << endl ;
        }
        return 0 ;
}
```

В листинге 11.8 матрица транспонируется сама в себе. Результаты исполнения программы показаны на рис. 11.5.

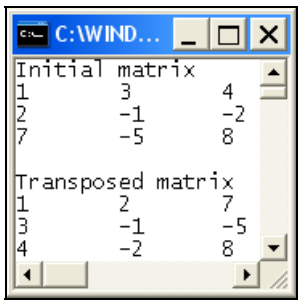


Рис. 11.5. Результаты исполнения программы транспонирования матрицы

Известно, что для транспонирования матрицы необходимо поменять местами строки со столбцами, причем первый элемент первой строки матрицы и последний элемент последней строки остаются на своих местах.

Ввиду того, что транспонируется квадратная матрица, для указания максимально допустимого значения индексов используется одна константа `N`, объявленная в начале программы. Далее матрица `m` инициализируется значениями, и указатель `pm` получает значение адреса первого элемента `& m [ 0 ] [ 0 ]`.

Вывод исходного состояния матрицы выполняется так же, как в предыдущем примере.

Внешний цикл `for` управляет изменением адреса строки, начиная со второй строки матрицы `i = 1` и заканчивая последней `i = N - 1`.

Внутренний цикл `for` управляет изменением адреса столбца, начиная с первого `j = 0` и заканчивая предпоследним столбцом `j = N - 2`. Во внутреннем цикле сначала происходит сохранение во временной переменной `tmp` значения элемента матрицы, который хранится по адресу `( pm + i * N + j )`. Затем по этому адресу записывается значение, хранимое по адресу `( pm + j * N + i )`. Последний оператор тела цикла записывает значение переменной `tmp` по адресу `( pm + j * N + i )`. В процессе перестановки используется операция разыменовывания `*`, потому что необходимо поменять местами не адреса элементов, а их значения.

Транспонированная матрица выводится при помощи вложенных циклов с использованием операции индексации.

Примечание

Представленная в примере программа транспонирует матрицу саму в себя. Если матрица транспонируется в другой массив, необходимо объявить этот массив в программе, а также изменить условия выполнения отвечающих за транспонирование циклов. Кроме того, временная переменная не понадобится. Далее показан фрагмент соответствующего кода, в котором вместо указателя использована операция индексации.

```
int res [ N ] [ N ] ;
for ( int i = 0; i < N; i++ )
        for ( int j = 0; j < N; j++ )
              res [ i ] [ j ] = m [ j ] [ i ] ;
```

Листинг 11.9. Пример реверса строки с помощью указателей

```
#include <iostream>
using namespace std ;
int main ( )
{
       char s [ ] = "lisaB on eta tenoB asiL" ;
       cout << "String:\t\t" << s << endl ;
       int len = sizeof ( s ) - 1 ;
       // адрес первого символа строки pTop = &s [ 0 ]
       // адрес предпоследнего символа строки pEnd = pTop + len - 1
       char* pTop = & s [ 0 ], * pEnd = pTop + len - 1 ;
       char tmp ;
       // цикл реверса строки
```

```
        while ( pTop < pEnd )
        {
                // перестановка символов
                tmp = * pTop ;
                * pTop++ = * pEnd ;
                * pEnd-- = tmp ;
        }
        // вывод строки после реверса
        cout << "Result:\t\t" << s << endl ;
        return 0 ;
}
```

Программа (листинг 11.9) представляет собой модифицированную версию рассмотренного в *главе 10* примера реверса строки. Модификация заключается в использовании указателей `pTop` и `pEnd` для перестановки символов в строке.

После вывода исходной строки `s` вычисляется ее длина `len` без учета нулевого байта. Следует заметить, что для определения длины строки в C++ существует специальная функция. Основные функции обработки символьных строк обсуждаются в *главе 13*. Оператор `sizeof` нельзя использовать для вычисления длины строки, если она задана при помощи указателя (в этом случае оператор `sizeof` вернет размер адреса, а не строки).

Указатель `pTop` инициализируется адресом первого элемента строки `& s [ 0 ]`, указатель `pEnd` — адресом предпоследнего элемента `pTop + len - 1`. Последним элементом строки `s` является нулевой байт.

Каждая итерация цикла `while` начинается с сохранения во временной переменной `tmp` значения символа, на который указывает `pTop`. Цикл управляет изменением указателей так, что указатель `pTop` увеличивается на 1 после сохранения по адресу `pTop` символа, на который указывает `pEnd`, а указатель `pEnd` уменьшается на 1 после сохранения по адресу `pEnd` временной переменной `tmp`. Итерации повторяются до тех пор, пока адрес `pTop` меньше адреса `pEnd`. Как только указатели получат одно и то же значение адреса, цикл и перестановки закончатся.

Результатом работы программы будет строка

```
Lisa Bonet ate no Basil.
```

Примечание

Вместо выражения `pTop = & s [ 0 ]` для инициализации можно записать выражение `pTop = s`, так как имя одномерного массива является его адресом.

Листинг 11.10. Пример подсчета числа вхождений подстроки в строку

```
#include <iostream>
using namespace std ;
int main ( )
{
        char* ps = "KING, ARE YOU GLAD YOU ARE KING?" ;
        char* pss = "KING" ;
        int n = 0 ;             // счетчик вхождений подстроки
        char* p, *r ;           // временные указатели
        // цикл просмотра символов в строке ps
        while ( * ps )
        {
                // цикл проверки вхождения подстроки r в строку p
                for ( p = ps, r = pss; * r && * p == * r; p++, r++ )
                        ;
                if ( ! * r )  // если символ подстроки r нулевой байт,
                {
                        ++n ;           // увеличить счетчик вхождений
                        ps = p ;        // сохранить адрес конца проверки
                }
                else            // если конец подстроки r не достигнут,
                        ps++ ;          // увеличить адрес на 1
        }
        cout << "n = " << n << endl ;
        return 0 ;
}
```

В начале листинга 11.10 объявляется указатель на символьный тип `ps` с одновременной инициализацией содержимого строкой `KING, ARE YOU GLAD YOU ARE KING?`. Строка начнется с адреса, на который указывает `ps`. Аналогично `ps` объявляется и инициализируется подстрока, адрес которой находится в переменной-указателе `pss`. Далее объявляется целая переменная `n`, где будет накапливаться количество вхождений подстроки `KING` в строку с адресом `ps`.

Внешний цикл `while` управляет изменением указателя `ps`, указывающего на символ, с которого начинается проверка вхождения подстроки. Цикл `while` закончится, когда содержимое по записанному в `ps` адресу будет равно `'\0'`, что сделает выражение `( * ps )` ложным.

Для реализации вычисления `n` дополнительно требуются два указателя — `p` и `r`. Первый используется для доступа к символам строки, второй — для доступа к символам подстроки. Во внутреннем цикле `for` указатели `p` и `r` перемещаются до тех пор, пока не встретится конец подстроки `* r` и пока символы строки совпадают с символами подстроки `* p == * r`. Пока выражение `( * r && * p == * r )` истинно, выполняется инкремент `p++` и `r++`. Тело внутреннего цикла — пустой оператор точка с запятой. Целью этого цикла `for` является определение адреса подстроки `r`.

Если после окончания цикла `r` указывает на нулевой байт, то это обозначает совпадение всех символов подстроки с проверяемыми символами строки. Отрицание `* r` в этом случае будет истинным. Следовательно, выполняется блок условного оператора `if else`, в котором значение счетчика вхождений `n` увеличится на 1, и указатель `ps` получит значение `p`, который указывает на следующий символ в строке после найденного совпадения. Если же символ, на который указывает `r`, не является нулевым байтом, происходит инкремент указателя `ps`.

После выполнения условного оператора `if else` управление получит оператор внешнего цикла `while`.

Результатом исполнения программы будет значение `n = 2`.

Операторы распределения памяти *new* и *delete*

Для управления свободной памятью, которая предоставляется программе операционной системой, в C++ существуют унарные операторы `new` и `delete`. Эти операторы позволяют динамически распределять память для любых типов данных, и используются для выделения и освобождения памяти в ходе выполнения программы. Программист создает данные, используя `new`, и удаляет их с использованием `delete`.

Оператор `new` выделяет блок памяти и возвращает указатель на первую ячейку этого блока. Если `new` не в состоянии найти необходимое пространство свободной памяти, он вернет указатель 0.

Оператор `new` используется в следующих форматах:

```
указатель = new тип ;
указатель = new тип ( значение ) ;
```

Тип указателя и тип, указанный за ключевым словом `new`, должны совпадать. Когда используется второй формат записи оператора, одновременно

с выделением необходимого количества байт производится инициализация содержимого блока памяти указанным в круглых скобках значением (листинг 11.11).

Листинг 11.11. Пример использования оператора `new`

```
#include <iostream>
using namespace std ;
int main ( )
{
        // объявление указателей
        int* pi ; unsigned short* pu ;
        // выделение памяти
        pi = new int ;
        pu = new unsigned short ( 200 ) ;
        // объявление указателя с выделением памяти
        double* pd = new double ;
        // объявление указателя с инициализацией
        float* pf = new float ( -3.15 ) ;
        // вывод адресов, выделенных new
        cout << "Pointers" ;
        cout << "\npi\t" << pi << "\npu\t" << pu ;
        cout << "\npd\t" << pd << "\npf\t" << pf ;
        // вывод значений
        cout << "\n\nValues with initialization" ;
        cout << "\n* pu\t" << * pu ;
        cout << "\n* pf\t" << * pf ;
        cout << "\n\nValues without initialization" ;
        cout << "\n* pi\t" << * pi ;
        cout << "\n* pd\t" << * pd << endl ;
        // присваивание значений и вывод
        * pi = - * pu ; * pd = - * pf ;
        cout << "\n\nValues after initialization" ;
        cout << "\n* pi\t" << * pi ;
        cout << "\n* pd\t" << * pd << endl ;
        return 0 ;
}
```

Пояснения к программе находятся в комментариях текста. Следует обратить внимание на то, что значение, которое расположено по адресу, возвращенному оператором `new` без инициализации памяти, соответствует содержимому области памяти до начала исполнения программы и фактически не определено.

На рис. 11.6 можно увидеть результаты работы программы с выводом состояния выделенных ячеек памяти при использовании оператора `new` совместно с инициализацией и без таковой.

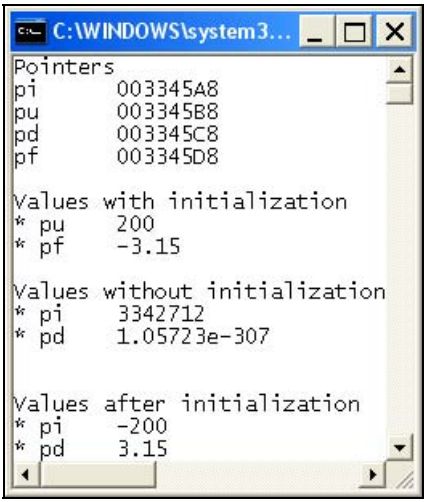


Рис. 11.6. Результаты исполнения программы с оператором `new`

Чтобы освободить блок памяти, выделенный оператором `new`, необходимо воспользоваться операцией динамического освобождения памяти.

Оператор `delete` освобождает память, выделенную ранее оператором `new`. Недопустимо применение оператора `delete` для указателя, который не участвовал в выполнении оператора `new`.

Формат оператора `delete` прост и выглядит следующим образом:

```
delete указатель ;
```

Освобождение памяти не означает удаления связанного с этим блоком памяти указателя. Просто память становится свободной, а указатель теряет силу. После выполнения оператора `delete` операция разыменовывания может иметь непредсказуемый результат или привести к ошибке исполнения. В связи с этим не следует использовать указатели на освобожденную память (листинг 11.12).

Листинг 11.12. Пример использования оператора `delete`

```
#include <iostream>
using namespace std ;
int main ( )
{
        int x = 25 ;
        cout << "Value x\t" << x ;
        // объявление указателя с инициализацией
        int* py = new int ( x ) ;
        cout << "\nValue *py before delete\t" << * py ;
        // освобождение памяти
        delete py ;
        // состояние памяти после освобождения
        cout << "\nValue *py after delete\t" << * py << endl ;
        return 0 ;
}
```

Результаты выполнения программы представлены на рис. 11.7, который показывает, что произойдет при использовании разыменовывания указателя `py` после выполнения оператора `delete`. Если до освобождения памяти по адресу находилось значение 25, то после освобождения оно стало равным 0.

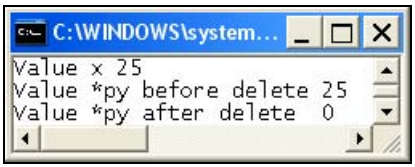


Рис. 11.7. Результаты исполнения программы с оператором `delete`

В примере демонстрируется еще один способ инициализации памяти при ее выделении оператором `new`. В круглых скобках указан идентификатор переменной `x`, значение которой записывается в выделенный блок. При подобном способе инициализации необходимо, чтобы тип переменной совпадал с типом указателя.

Указатели и динамические массивы

Операторы `new` и `delete` позволяют выделять и освобождать память при работе с динамическими массивами, размер которых становится известным в процессе выполнения программы. При работе с динамическими массивами используются следующие форматы операторов `new` и `delete`:

```
указатель = new тип _массива [ размер ] ;
delete [ ] указатель ;
```

Размер в операторе `new` записывается в квадратных скобках и задает число элементов размещаемого массива. В операторе `delete` размер не указывается, но наличие квадратных скобок является обязательным. Их использование сообщает, что одновременно освобождается память, используемая для всех элементов массива.

Листинг 11.13. Пример использования операторов `new` и `delete` для динамического массива

```
#include <iostream>
using namespace std ;
int main ( )
{
        // ввод размера массивов
        int size ; cout << "Enter size of array -> " ; cin >> size ;
        // выделение памяти под массивы
        double* pdArray = new double [ size ] ;
        int* piArray = new int [ size ] ;
        // адреса элементов массивов
        cout << "\nAddresses for double\n" ;
        for ( int i = 0; i < size; i++ )
                cout << ( pdArray + i ) << '\t' ;
        cout << "\n\nAddresses for int\n" ;
        for ( int i = 0; i < size; i++ )
                cout << ( piArray + i ) << '\t' ;
        // значения элементов массива целых чисел
        cout << "\n\nValues of integer array\n" ;
        for ( int i = 0; i < size; i++ )
        {
                * ( piArray + i ) = i * i ;
cout << * ( piArray + i ) << "\t\t" ;
```

```
        }
        cout << endl ;
        // освобождение памяти
        delete [ ] piArray ; delete [ ] pdArray ;
        return 0 ;
}
```

Синтаксис языка C++ требует при объявлении массива указывать его размер константой, иначе компилятор сообщит об ошибке. Однако количество элементов не всегда известно заранее. Чтобы избежать использования памяти, которая может и не понадобиться, следует воспользоваться указателем.

В листинге 11.13 сначала вводится размер `size` для размещения массивов целого и вещественного типов. Выделение необходимой памяти для элементов массивов производится при помощи операторов `new`, которые возвращают начальные адреса для целочисленного массива в `piArray` и для вещественного — в `pdArray`.

Два цикла `for` выводят адреса элементов обоих массивов. Из результатов выполнения программы (рис. 11.8) видно, что адреса отличаются друг от друга на длину типа массива: для типа с плавающей точкой — на 8 байтов, а для целого типа — на 4 байта. Для получения адреса элемента начальный указатель увеличивается на значение индекса этого элемента в массиве.

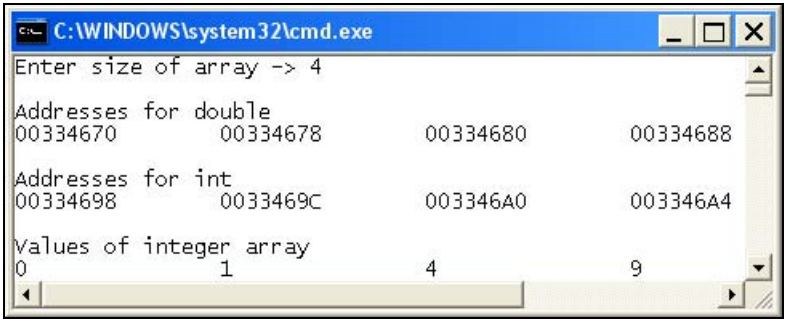


Рис. 11.8. Результаты выполнения программы с операторами `new` и `delete` для массивов

Следующий цикл обеспечивает сохранение по адресу элемента массива, на который указывает `piArray`, значения квадрата индексной переменной `i` и вывод этого значения. Для присваивания значений использована операция разыменовывания указателя `* ( piArray + i )`.

Перед оператором выхода из программы выделенная для массивов память освобождается при помощи операторов `delete`.

Указатели и спецификатор *const*

Спецификатор `const` может находиться в операторе объявления указателя и перед именем типа, и перед идентификатором.

Если спецификатор стоит перед именем типа, то объявляется *указатель на константу*. Следовательно, невозможно изменить значение переменной, на которую указывает указатель, однако можно изменить значение самого указателя.

Если спецификатор стоит непосредственно перед идентификатором, то объявляется *константный указатель*. Значит, нельзя изменить значение указателя, но можно изменять значение того, на что он указывает.

Листинг 11.14. Пример использования указателей со спецификатором `const`

```
// Пример указателей с модификатором const
#include <iostream>
using namespace std ;
int main ( )
{
        // указатель на константу
        cout << "Pointer on constant\n" ;
        const char A = 'A' ; const char* pC = &A ;
        cout << "I pointer " << &pC ;
        cout << "\tmy constant " << *pC << endl ;
        // указатель можно изменить
        const char Z = 'Z' ; pC = &Z ;
        cout << "And now\t\t\tmy constant " << *pC << endl ;
        // константный указатель
        cout << "\nThe constant pointer" ;
        char* const pCP = new char [ 40 ] ;
        char s1 [ ] = "As you brew," ;
        char s2 [ ] = "so must you drink." ;
        strcpy_s ( pCP, 40, s1 ) ;
        cout << '\n' << &pCP << " point on ->\t" << pCP ;
        // содержимое можно изменить
        strcpy_s ( pCP, 40, s2 ) ;
        cout << "\nAnd now I point on ->\t" << pCP << endl ;
        // константный указатель на константу
        const char* const p = "constant pointer on a constant" ;
```

```
        // освобождение памяти
        delete [ ] pCP ;
        // ошибки
//      *pC = 'Z' ;                  // *pC - символьная константа
//      pCP = new char [ 20 ] ;      // pCP - константный указатель
//      p = new char [ 40 ] ;        // p   - константный указатель
//      * ( p + 7 ) = 'T' ;          // * ( p + 7 )- константный символ
//      delete pC ;                  // pC  - не было new
//      delete [ ] s1 ;              // s1  - массив
        return 0 ;
}
```

В листинге 11.14 (результаты исполнения можно увидеть на рис. 11.9) показаны разные способы использования спецификатора `const` вместе с указателями.

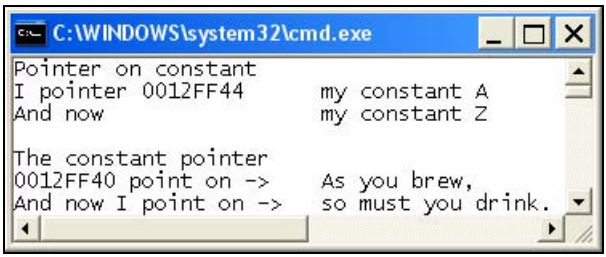


Рис. 11.9. Результаты использования спецификатора `const` и указателей

Неконстантный указатель на константные данные `pC` модифицируется, чтобы сначала указывать символьную константу `A`, потом — на `Z`. Этот указатель можно изменять, чтобы указывать на любые символы, но сами символы, на которые он ссылается, не могут быть модифицированы. Это свойство полезно при передаче аргументов в функцию, когда аргументы нельзя изменять.

Константный указатель на неконстантные данные `pCP` всегда указывает на одну и ту же ячейку памяти, адрес которой вернул оператор `new`, динамически выделивший блок памяти размером в 40 байтов. Далее в программе данные этого блока памяти модифицируются. По адресу из `pCP` копируется и выводится строка `s1`, после чего по этому же адресу копируется строка `s2`.

Константный указатель на константную строку `p` указывает на одну и ту же ячейку памяти. Данные в этой ячейке памяти и в остальных ячейках, относящихся к строковой константе, нельзя модифицировать. Эти данные посредством индекса строки и разыменовывания указателя можно только просматривать или использовать их в качестве входных данных.

Ошибочная запись операций при использовании указателей разного типа приведена в комментариях перед оператором `return`.

Примечание

Указатели, которые объявлены константными, должны получить начальное значение при своем объявлении.

Массивы указателей

Указатели можно помещать в массив. Использование массивов указателей позволяет формировать массивы значений динамически в процессе выполнения программы. Каждый элемент такого массива является указателем на переменную. Тип переменной должен быть таким же, как базовый тип массива указателей. Перед использованием массив указателей необходимо объявить по правилам объявления массивов, его можно проинициализировать при объявлении.

Для присваивания элементу массива указателей значения адреса переменной типа, на который он указывает, следует выполнить операцию взятия адреса `&` для переменной. Когда существует необходимость динамически формировать массив указателей, следует воспользоваться оператором `new` для выделения требуемого блока памяти, который после его использования освободить при помощи оператора `delete`. Чтобы получить значение переменной, на которую указывает элемент массива указателей, необходимо разыменовать указатель (листинг 11.15).

Листинг 11.15. Пример реализации меню при помощи массива указателей

```
#include <iostream>
using namespace std ;
int main ( )
{
        char const* pS [ 4 ] =
        { "1. New array", "2. View", "3. Search", "0. Exit" } ;
        char choice ;
        do
        {
                cout << "Menu\n" ;
                for ( int i = 0; i < 4; i++ ) cout << pS [ i ] << endl ;
                cout << "\nYour choice - > " ;
```

```
            cin >> choice ;
            switch ( choice )
            {
            case '1' : cout << '\n' << pS [ 0 ] << "\n\n" ; break ;
            case '2' : cout << '\n' << pS [ 1 ] << "\n\n" ; break ;
            case '3' : cout << '\n' << pS [ 2 ] << "\n\n" ; break ;
            case '0' : break ;
            default : cout << "\nIllegal choice\n\n";
            }
        } while ( choice != '0' ) ;
        return 0 ;
}
```

В примере приводится простейший способ создания меню для программы. Пункты меню записаны константными значениями, адреса которых помещаются в массив неконстантных указателей `pS`. Во внешнем цикле `do while` выводятся значения констант, после чего пользователю предлагается ввести значение. Введенный символ сохраняется в переменной `choice`. Далее программа при помощи оператора `switch` анализирует переменную `choice`. Если переменная получает значение `1`, `2` или `3`, управление получает соответствующая метка `case` оператора-переключателя, где выводится название выбранного пункта меню. Выход после вывода обеспечивает оператор `break`. При значении `0` `switch` завершает свою работу, передавая управление внешнему циклу, который в этом случае тоже заканчивается по ложному значению условия выполнения цикла. Если же переменная `choice` не равна ни одному из перечисленных значений, по метке `default` выводится предупреждение о недопустимом выборе, и начинается новая итерация внешнего цикла. Этот цикл прервется, когда будет введен `0`. Результаты работы программы показаны на рис. 11.10.

Листинг 11.16. Пример инициализации массива указателей существующими адресами

```
#include <iostream>
using namespace std ;
int main ( )
{
        const int SIZE = 4 ;
        int array [ SIZE ] = { 5, -3, 0, 17 } ;
        // объявление массива указателей на целый тип
        int* p [ SIZE ] ;
```

```
        // инициализация массива указателей адресами
        for ( int i = 0; i < SIZE; i++ )    p [ i ] = & array [ i ] ;
        // вывод состояния массива указателей
        for ( int i = 0; i < SIZE; i++ )    cout << p [ i ] << '\t' ;
        cout << endl ;
        // вывод значений через массив указателей
        for ( int i = 0; i < SIZE; i++ )    cout << * p [ i ] << "\t\t" ;
        cout << endl ;
        return 0 ;
}
```

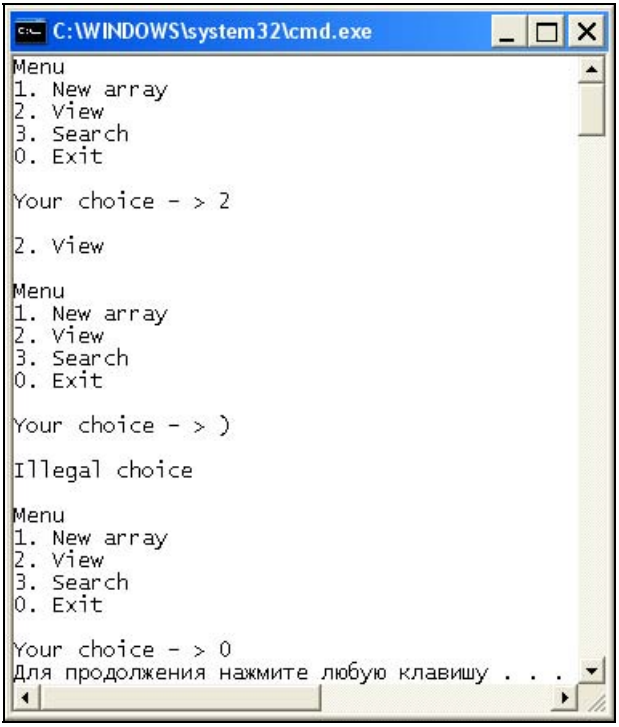


Рис. 11.10. Результаты работы программы

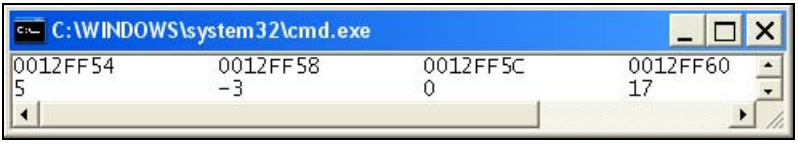


Рис. 11.11. Результаты программы инициализации массива указателей существующими адресами

Элементы массива указателей p получают значения адресов элементов из массива array (листинг 11.16). После инициализации выводится состояние массива указателей и значения элементов, на которые указывают его элементы, как показано на рис. 11.11.

Листинг 11.17. Пример динамического формирования массива указателей

```
#include <iostream>
using namespace std ;
const int SIZE = 4 ;
int main ( )
{
        // инициализация массива указателей адресами
        // и значениями по выделенным адресам
        int* p [ SIZE ], x ;
        for ( int i = 0; i < SIZE; i++ )
        {
                cout << "-> " ;       cin >> x ;
                p [ i ] = new int ( x ) ;
        }
        // вывод состояния массива указателей
        for ( int i = 0; i < SIZE; i++ )    cout << p [ i ] << '\t' ;
        cout << endl ;
        // вывод значений через массив указателей
        for ( int i = 0; i < SIZE; i++ )    cout << * p [ i ] << "\t\t" ;
        cout << endl ;
        // освобождение памяти
        for ( int i = 0; i < SIZE; i++ )    delete p [ i ] ;
        return 0 ;
}
```

В листинге 11.17 элементы массива указателей p получают свои значения при помощи операции динамического выделения памяти. Возвращенный оператором new адрес становится значением элемента p [i]. По этому адресу при помощи инициализатора (x) записывается значение переменной x. Операция разыменовывания в этом случае не нужна. На рис. 11.12 проиллюстрированы результаты исполнения программы.

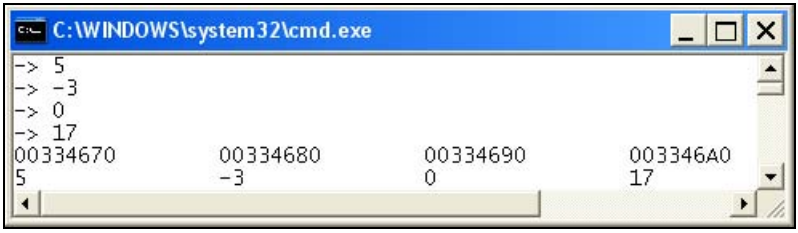


Рис. 11.12. Инициализация массива указателей при помощи оператора `new`

Примечание

Следует помнить, что базовый тип массива указателей, тип в операторе `new` и тип переменной, адрес которой хранится в массиве, должны совпадать. Не стоит забывать, что выделенные блоки памяти должны быть освобождены после их использования. Чтобы освободить оперативную память для массива указателей, необходимо организовать цикл, в каждой итерации которого выполнится оператор `delete` для текущего элемента `p [ i ]` из массива.

Листинг 11.18. Пример обработки массива строк через массив указателей

```
#include <iostream>
using namespace std;
const int SIZE = 10 ; const int LEN = 40 ;
int main ( )
{
        char *pa [ SIZE ] ;                  // массив указателей на строки
        char *str = new char [ LEN ] ;       // строка-буфер
        int n ;                              // количество введенных строк
        char *p ; int i, j ;          // временные указатель и переменные
        while ( true )         // цикл ввода не менее пяти строк
        {
                cout << "Enter not less than 5 lines "
                        << "or empty line for an exit\n" ;
                for ( i = 0; i < SIZE ; i++ )        // цикл ввода строк
                {
                        cin.getline ( str, LEN ) ;
                        if ( ! *str )          // если строка пустая,
                                break ;        // то выйти из цикла ввода for
                        // выделение блока памяти
```

```
                pa [ i ] = new char [ strlen ( str ) + 1 ] ;
                // дублирование str в выделенный блок
                strcpy_s ( pa [ i ], LEN, str ) ;
            }
            if ( i < 5 )                // если введено < 5 строк,
                for ( j = 0; j < i; j++ )
                    delete pa [ j ] ;    // то освободить память
            else  break ;          // иначе выйти из цикла while
        }                   // окончание цикла while
        n = i ;             // сохранение числа введенных строк
        // вывод введенных строк
        cout << "\nENTERED ARRAY\n" ;
        for ( i = 0 ; i < n ; i++ )  cout << pa [ i ] << endl ;
        // поиск самой короткой строки
        int iMin = 0 ;        // индекс строки с минимальной длиной
        for ( i = 1; i < n; i++ )
            if ( strlen ( pa [ i ] ) < strlen ( pa [ iMin ] ) )
                iMin = i ;
        // Обработка массива
        if ( iMin )           // если строка не первая ( iMin != 0 ),
        {
            p = pa [ iMin ] ;     // то запомнить ее адрес
// определить новый порядок строк: с адреса pa [iMin] до адреса pa [1]
// на место текущего адреса pa [i] записывать предыдущий адрес pa [i-1]
            for ( i = iMin; i > 0; i-- ) pa [ i ] = pa [ i - 1 ] ;
            // переопределить адрес первой строки
            pa [ 0 ] = p ;
        }
        cout << "\nARRAY - RESULT\n" ;
        for ( i = 0; i < n; i++ )
        {       cout << pa [ i ] << endl ;   delete pa [ i ] ;     }
        delete str ;
        return 0 ;
}
```

Результаты исполнения программы (листинг 11.18) представлены на рис. 11.13.

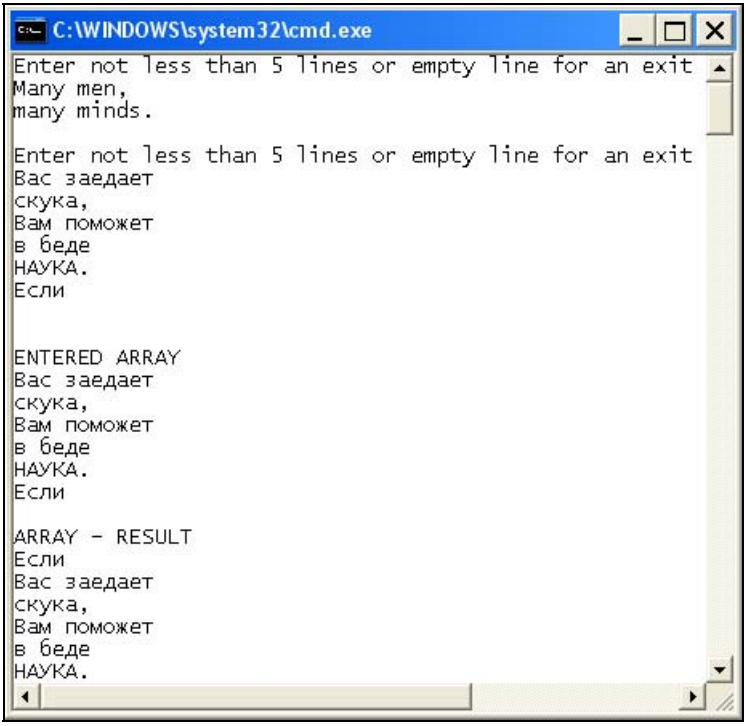


Рис. 11.13. Результаты исполнения программы обработки массива строк

В программе объявлен массив указателей `pa`, максимальный размер которого определяется глобальной переменной `SIZE`. Внешний цикл `while` организует ввод и размещение в оперативной памяти не менее 5 строк. Внутренний цикл `for` отвечает за ввод символов в строку-буфер `cin.getline ( str, LEN )`, выделение необходимого для размещения этой строки блока памяти

```
pa [ i ] = new char [ strlen ( str ) + 1 ]
```

и копирования строки-буфера по новому адресу `strcpy_s ( pa [ i ], LEN, str )`. Если было введено менее 5 строк, выделенная память освобождается и начинается новая итерация внешнего цикла. Иначе управление передается к сохранению количества введенных строк `n = i`. Этот же оператор получает управление, когда вводится пустая строка (нажатием клавиши <Enter>), что проверяется условием

```
( ! *str ).
```

После размещения адресов и строк выводится состояние массива указателей. Далее происходит поиск индекса `iMin` самой короткой строки из всех введенных. Для этого предполагается, что такая строка первая `iMin = 0`. В цикле

сравниваются длины текущей строки с индексом i и строки с предполагаемым индексом iMin. При необходимости iMin корректируется.

Обработка строк заключается в перемещении адреса строки с минимальной длиной в самое начало массива. Чтобы сделать это, во временной переменной p сохраняется указатель на самую короткую строку pa [iMin], и затем в цикле for на место текущего указателя с индексом i записывается указатель с предыдущим индексом i - 1. Цикл управляет изменением индекса i, уменьшая его на 1 от индекса iMin до индекса 1. Когда цикл заканчивается, происходит корректировка первого указателя из массива pa [0], на место которого записывается значение временного указателя p. Следует заметить, что сами строки не копируются.

В конце программы выводится новое состояние массива указателей.

Примечание

Функция getline ( ), вызванная для объекта cin, позволяет ввести строку, в которой могут быть пробелы. Функция strlen ( ) возвращает длину строки без учета нулевого байта. Функция strcpy_s ( ) копирует строку по указанному адресу. Детально эти функции работы со строками рассматриваются в *главе 13*.

Указатели на указатели

Указатель может ссылаться на другой указатель, который содержит адрес обычной переменной. Такой указатель называется указатель на указатель (pointer to pointer), а адресация — косвенной. В случае косвенной адресации один указатель ссылается на другой указатель, где хранится адрес переменной, в котором записано нужное значение. На рис. 11.14 проиллюстрировано состояние памяти при использовании косвенной адресации.

Чтобы объявить переменную указатель на указатель, перед идентификатором записывается дополнительная звездочка. Чтобы извлечь значение переменной при косвенной адресации, необходимо дважды применить операцию разыменовывания (листинг 11.19).

Листинг 11.19. Пример косвенной адресации

```
#include <iostream>
using namespace std;
int main ( )
{
       int x ( 5 ), y ( -x ), *px, ** pPx ;
```

```
    px = & x ;     // инициализация указателя
    pPx = & px ;  // инициализация указателя на указатель
    cout << px << endl ; //выводится адрес x
    cout << pPx << endl ; // выводится адрес указателя px
    cout << x << endl ;   // выводится значение x = 5
    cout << ** pPx << endl ;     // выводится значение x через pPx
    ** pPx *= y ; // x получает значение через указатель на указатель
    cout << x << endl ;          // выводится x = -25
    cout << ** pPx << endl ;     // выводится -25
    * pPx = & y ; // новая инициализация указателя на указатель
    cout << ** pPx << endl ;     // выводится значение y = -5
    return 0 ;
}
```

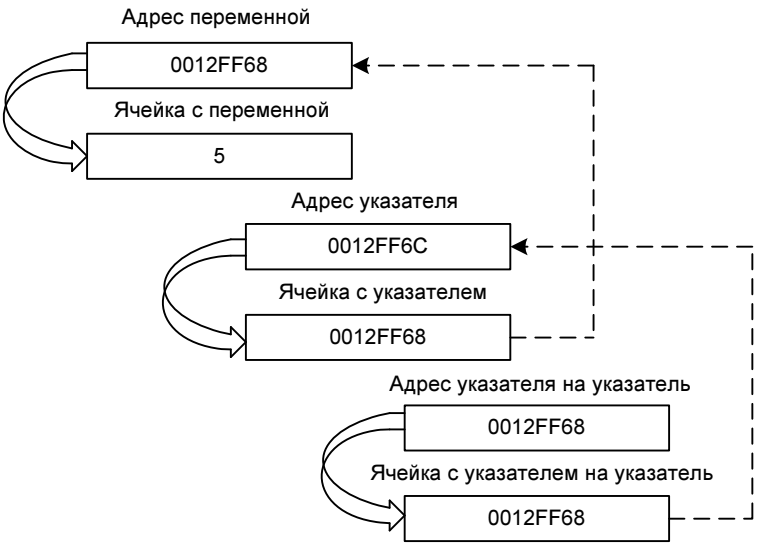


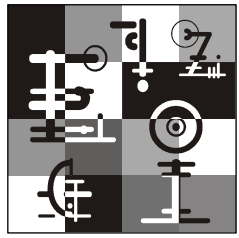
Рис. 11.14. Состояние памяти при косвенной адресации

Двумерный массив создается в процессе исполнения программы (листинг 11.20). Количество строк `m` и столбцов `n` вводятся пользователем. Сначала выделяется блок памяти для массива указателей на строки матрицы размером `m`. Возвращенный оператором `new` адрес становится значением указателя на указатель `p`. Далее в цикле выделяются блоки памяти размером `n` для размещения одномерных массивов (строк матрицы), начальные адреса которых записываются в соответствующий элемент `p [ i ]` массива указателей. Указатель

на указатель p позволяет обращаться к элементам матрицы так, как если бы она была объявлена обычным способом. В конце программы выделенная память освобождается. Сначала освобождается пространство, отведенное для строк, а затем — для указателей на них.

Листинг 11.20. Пример создания динамического двумерного массива

```
#include <iostream>
using namespace std;
int main ( )
{
       double** p ;          // указатель на указатель
       // ввод количества строк m и столбцов n
       int m, n ;
       cout << "Enter quantity of rows\t-> " ; cin >> m ;
       cout << "Enter quantity of columns\t-> " ; cin >> n ;
       // выделение памяти для массива указателей на строки
       p = new double* [ m ] ;
       // выделение памяти для элементов строк
       for ( int i = 0; i < m; i++ ) p [ i ] = new double [ n ] ;
       // заполнение матрицы значениями
       for ( int i = 0; i < m; i++ )
              for ( int j = 0; j < n; j++ ) cin >> p [ i ] [ j ] ;
       // вывод матрицы
       for ( int i = 0; i < m; i++ )
       {
              for ( int j = 0; j < n; j++ )
                     cout << p [ i ] [ j ] << '\t' ;
              cout << endl ;
       }
       // освобождение памяти
       for ( int i = 0; i < m; i++ ) delete [ ] p [ i ] ;
       delete [ ] p ;
       return 0 ;
}
```



Глава 12

Структуры

Структура (structure) объединяет логически связанные данные разных типов и представляет собой набор переменных, которые объединены общим именем.

Объявление структуры начинается с ключевого слова `struct` и соответствует следующему формату:

```
struct идентификатор { объявления_элементов_структуры } ;
```

Идентификатор задает имя структуры (равно тип) и используется при объявлении структурных переменных такого типа. Элементы структуры часто называют переменными-членами. Структуры могут быть вложены друг в друга. Это значит, что элементы сами могут быть структурами. Объявление структуры не резервирует никакого пространства в памяти, оно лишь создает новый тип данных. Память выделяется при объявлении переменной типа структуры или при помощи операции `new`, если используется указатель на структуру.

Операции доступа к элементам структуры

Для доступа к элементам структуры по имени переменной типа структуры используется *операция точка* (обозначается точкой).

Для доступа к элементам структуры через указатель на переменную служит *операция стрелка* (обозначается `->`).

При записи в программе обе операции (`.` и `->`) помещаются между именем переменной типа структуры и идентификатором элемента структурного типа.

Листинг 12.1. Пример программы с простой структурой

```
#include <iostream>
using namespace std ;
int main ( )
{
        // объявление типа структуры
        struct Address {
                char city [ 20 ] ;    // город
                char street [ 30 ] ;  // улица
                int house ;           // номер дома
        } ;
        // объявление переменной
        Address a ;
        // ввод значений элементов
        cin >> a . city ; cin >> a . street ; cin >> a . house ;
        // объявление указателя с инициализацией
        Address *p = new Address ;
        // ввод значений элементов
        cin >> p -> city ; cin >> p -> street ; cin >> p -> house ;
        // присваивание структур
        Address copy ; copy = a ; copy = * p ;
        return 0 ;
}
```

В листинге 12.1 определяется тип структуры `Address`. В первой части программы объявляется переменная `a`, для которой вводятся значения элементов. Чтобы элементы получили свои значения, используется операция точка. Во второй части объявляется указатель `p` на тип структуры. Одновременно с объявлением выделяется память при помощи `new`, и адрес выделенного блока памяти записывается в переменную `p`. Для доступа к элементам теперь используется операция стрелка.

Значения элементов, принадлежащие одной структурной переменной, могут быть присвоены другой структуре. Для этого нет необходимости копировать каждый элемент в отдельности, достаточно воспользоваться оператором присваивания. В примере структура `copy` сначала получает значения, хранящиеся в структурной переменной `a`, после чего структура `copy` приравнивается структуре, адрес которой записан в `p`. В последнем случае понадобилась операция разыменовывания.

Примечание

Операции отношения (==, !=, > и т. д.) над структурами недопустимы. Для сравнения структурных переменных используется перегрузка операторов, рассматриваемая в *главе 17*.

Листинг 12.2. Пример иерархической структуры

```
#include <iostream>
using namespace std ;
int main ( )
{
        // простые структуры
        struct Address {
                char city [ 20 ] ; char street [ 30 ] ; int house ;
        } ;
        struct Person {
                char Fname [ 15 ] ; char Lname [ 20 ] ;
        } ;
        // иерархическая структура
        struct Employee {
                Person p ;              // вложенная структура
                Address addr ;          // вложенная структура
        } ;
        // объявление переменной указателя
        Employee *p = new Employee ;
        // ввод значений элементов
        cin >> p -> p . Fname ; cin >> p -> p . Lname ;
        cin >> p -> addr . city ;
        cin >> p -> addr . street ;
        cin >> p -> addr . house ;
        return 0 ;
}
```

В листинге 12.2 тип иерархической структуры `Employee` включает в качестве элементов две переменные, одна из которых имеет тип структуры `Person`, вторая — тип структуры `Address`. Указатель `p` содержит адрес выделенного блока памяти для размещения переменной типа `Employee`. Чтобы получить доступ к элементам этой переменной, используется операция стрелка, так как `p` — это указатель. Для получения доступа к элементам вложенных структур

p и addr используется операция точка. Важно понимать, что идентификатор p в начале является указателем на тип Employee, а идентификатор p в середине является совсем другой переменной и ссылается на переменную — член структуры типа Employee.

Инициализация структур

Структурная переменная может быть инициализирована. Для этого в операторе объявления переменной необходимо после операции присваивания в фигурных скобках указать значения элементов в порядке их объявления.

Если структура является иерархической, то значения каждой вложенной переменной структуры заключаются в отдельные фигурные скобки.

Если членом структуры является массив, то значения всех элементов массива должны быть последовательно указаны.

Листинг 12.3. Пример инициализации переменных структурного типа

```
#include <iostream>
using namespace std ;
int main ( )
{
        struct Time {  int hour ; int minute ; int second ; } ;
        struct Interval { Time top ; Time end ; } ;
        // инициализация простой структуры
        Time top = { 5, 30, 0 } ; Time end = { 15, 30, 45 } ;
        // инициализация иерархической структуры
        Interval i1 = { { 5, 30, 0 }, { 15, 30, 45 } } ;
        // вывод структурной переменной i1
        cout << i1.top.hour << ':' ;
        cout << i1.top.minute << ':' ;
        cout << i1.top.second << " - " ;
        cout << i1.end.hour << ':' ;
        cout << i1.end.minute << ':' ;
        cout << i1.end.second << endl ;
        // структура с элементом двумерный массив
        struct Book { char author [ 2 ] [ 50 ] ; char title [ 100 ] ; } ;
        // иерархическая структура
        struct BookCard { int number ; Book book ; } ;
        // инициализация структурной переменной
```

```
        BookCard c = { 12, "Harvey M. Deitel", "Payl J. Deitel",
                "C++ How to Program" } ;
        // вывод структурной переменной
        cout << c.number << " - " ;
        cout << c.book.author [ 0 ] << ", " ;
        cout << c.book.author [ 1 ] << "/ " ;
        cout << c.book.title << endl ;
        return 0 ;
}
```

В листинге 12.3 объявлена структура `BookCard` и структурная переменная `c`. Одним из элементов структуры является двумерный массив символьного типа `author`, у которого 2 строки. При инициализации структурной переменной `c` указаны: номер карточки, 2 автора и название. В результате выполнения программы на экран будут выведены следующие строки:

```
5:30:0 - 15:30:45
12 - Harvey M. Deitel, Payl J. Deitel/ C++ How to Program"
```

Массивы структур

Массивы структур ничем не отличаются от массивов встроенных типов данных и обрабатываются как обычно.

Листинг 12.4. Пример использования массива структур

```
#include <iostream>
using namespace std ;
// названия железных дорог
const char* pName [ ] = { "East-Siberian", "Far-East",
        "October ", "Krasnoyarsk", "Moscow ", "Sakhalin",
        "Northern", "Southeast", "Southern-Ural" } ;
// эксплуатационная длина железных дорог
double len [ ] = { 3848.1, 6000, 10143, 3157.9,
        8984, 833.8, 5951.7, 4189.1, 4806.6 } ;
int main ( )
{
        // вычисление размера массива len
        int size = sizeof ( len ) / sizeof ( len [ 0 ] ) ;
        // объявление структуры
```

```
        struct Road { char name [ 30 ] ; double length ; } ;
        // объявление массива типа структуры
        Road roadArray [ 20 ] ;
        // заполнение массива структур
        int i = 0 ;
        while ( i < size )
        {
                strcpy_s ( roadArray [ i ] . name, 30, pName [ i ] ) ;
                roadArray [ i ] . length = len [ i++ ] ;
        }
        // объявление указателя на массив типа структуры
        Road* pRailRoad = new Road [ size ] ;
        // приравнивание массивов
        for ( i = 0; i < size; i++ )
                * ( pRailRoad + i ) = roadArray [ i ] ;
        // вывод значений из массива с адресом pRailRoad
        Road* p = pRailRoad ;
        do
        {
                cout << p -> name << '\t' << p -> length << '\n' ;
                p++ ;
        } while ( p < pRailRoad + size ) ;
        // вычисление общей длины дорог
        double total = 0 ;
        for ( i = 0, p = pRailRoad ; i < size; i++ )
                total += p++ -> length ;
        cout << "\nSummary length = " << total << endl ;
        return 0 ;
}
```

В листинге 12.4 показана работа с массивом структур `roadArray` типа `Road` и с массивом, объявленным при помощи указателя `pRailRoad` на тот же структурный тип `Road`. В массив `roadArray` помещаются названия железных дорог из глобального массива `pName` и эксплуатационная длина дорог из глобального массива `len`. Заполнение массива происходит в цикле `while` до тех пор, пока индекс массива меньше размера имеющихся в глобальных массивов `size`. Для доступа к элементам массива `roadArray` используется операция индексации, а для доступа к элементам структуры — операция точка. Для создания массива структур через указатель `pRailRoad` применяется оператор

`new`, при помощи которого выделяется необходимый блок памяти, и цикл `for`, в котором копируются элементы массива `roadArray`. Каждый элемент `i` этого массива переписывается по адресу `pRailRoad + i`. Для вывода состояния полученного после копирования массива используется указатель `p` и цикл `do while`, управляющий изменением указателя на элементы массива. В цикле сначала выводится очередной элемент массива структур, на который указывает `p`, а затем указатель увеличивается на 1, обеспечивая доступ к следующему элементу в новой итерации цикла. В конце программы вычисляется и выводится общая длина всех дорог, сведения о которых имеются в массиве. Результаты исполнения программы показаны на рис. 12.1.

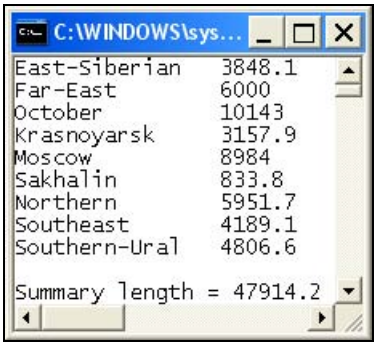
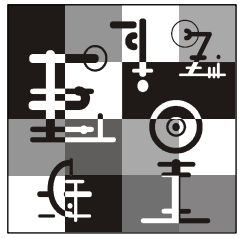


Рис. 12.1. Результаты выполнения программы с использованием массива структур



Глава 13

Функции

Функция (function) представляет собой именованную группу операторов, которая выполняет относительно независимую обработку данных и которую можно неоднократно вызывать из различных частей программы. При вызове функции ей передается последовательность значений, называемых параметрами (parameter).

Когда в программе встречается имя функции, она вызывается, и управление передается функции. После выполнения функции управление программой передается обратно в вызывающую среду, которая продолжает свою работу.

Примечание

Выполнение программы всегда начинается с функции `main ( )`.

Функции можно подразделить на библиотечные и авторские. Исполняемый код библиотечной функции находится в библиотечном файле, содержимое которого присоединяется к исполняемому файлу программы на этапе компоновки. Авторские функции создаются программистом, и должны быть откомпилированы.

Прототип функции

Все используемые программой функции должны быть объявлены. Объявление функции называется прототипом функции, который имеет следующий формат:

```
тип имя ( список типов параметров ) ;
```

`Тип` определяет тип возвращаемого функцией значения. Если функция ничего не возвращает, типом будет `void`. `Имя` указывает название функции, по которому к ней можно обратиться. `Список типов параметров` (через запятую) располагается

в круглых скобках и сообщает компилятору о том, что определение функции будет сделано позднее. Список может быть пустым, если функция использует глобальные переменные, однако круглые скобки сохраняются при отсутствии параметров. Прототип функции заканчивается точкой с запятой.

Определение функции

Определение функции состоит из заголовка и тела функции. Заголовок должен соответствовать прототипу функции. Это значит, что тип и имя должны совпадать с указанными в прототипе. Список параметров должен включать те же типы и следовать в том порядке, в каком они указывались в прототипе. Отличие заключается в том, что после типа параметра указывается идентификатор, который используется в теле функции для обозначения этого параметра. Указанные в круглых скобках заголовки параметры принято называть формальными. В определении функции после заголовка точка с запятой не ставится. Тело функции состоит из последовательности операторов, заключенной в фигурные скобки.

Возвращаемое функцией значение

Функция может возвращать значение. Если это так, тип возвращаемого значения должен быть определен и указан перед именем функции. Типом функции может быть любой допустимый синтаксисом языка C++ тип или тот, который был определен программистом. В теле должен присутствовать оператор `return`, который возвращает значение типа функции и передает управление в вызывающую среду.

Если функция ничего не возвращает, ее типом является тип `void`, который должен быть записан перед именем функции. В теле такой функции оператора `return` не должно быть. Вызывающая среда получит управление сразу же после выполнения функции.

Примечание

Можно не указывать тип функции, тогда по умолчанию ей будет присвоен тип `int`. Однако лучше всегда указывать тип для функций.

Вызов функции

Если функция возвращает значение, то вызов функции — это выражение, значением которого является возвращаемый функцией результат вычислений. Это выражение можно использовать в операторе присваивания в качестве

операнда, стоящего справа от знака равенства. Если же функция имеет тип `void`, ее использование в операторе присваивания недопустимо. Существует два способа вызова функции, которые имеют следующие форматы:

```
имя ( список параметров )
( * указатель_на_функцию ) ( список параметров )
```

Параметры, указанные в списке параметров при вызове функции, принято называть фактическими. Если функция не имеет параметров, круглые скобки сохраняются при ее вызове. В списке параметров могут быть указаны как константы, так и идентификаторы переменных, значения которых заранее вычислены перед вызовом функции. При попытке передать функции неопределенное значение параметра компилятор сообщит предупреждение об ошибке.

Указатель на функцию (function pointer) содержит адрес функции, который задается ее именем без скобок и аргументов. С помощью этого указателя функцию можно вызывать, а также передавать ее другим функциям в качестве параметра.

Область видимости функции

Областью видимости функции является файл, где находится ее определение. Код функции закрыт и недоступен для любых операторов, расположенных в других функциях, за исключением операторов вызова. Данные, определенные внутри какой-либо функции, никак не взаимодействуют с данными из другой функции, поскольку область видимости каждой функции индивидуальна.

Переменные, которые определены в теле функции, называются локальными. Они создаются при входе в функцию и автоматически уничтожаются при выходе из нее. Локальные переменные не сохраняют свои значения между вызовами функции.

Включение функций в проект приложения

Как правило, для больших программ используется раздельная компиляция. Функция `main ( )` помещается в отдельном файле с расширением `.cpp`, в который перед началом функции включаются заголовочные файлы библиотечных функций и заголовочный файл авторских функций. Заголовочный файл авторских функций содержит прототипы вызываемых функций и имеет расширение `.h`. Имя этого файла обычно совпадает с именем файла с функцией `main ( )`.

Определение функций помещается в один или несколько файлов с расширением .cpp. Таким образом, проект задачи включает несколько файлов:

- файл реализации главной функции `main ( )`;
- заголовочный файл с прототипами авторских функций;
- файл или файлы реализации авторских функций.

Для программ небольшого размера определение авторских функций может находиться и в одном файле с функцией `main ( )`. В этом случае прототипы следует указать перед началом `main ( )`, а определение поместить после тела функции `main ( )`. Размещение определения проверенных и отлаженных функций в отдельных файлах дает возможность подключать эти файлы в разные проекты без дублирования кода, поэтому такой вариант является более предпочтительным.

Передача параметра по значению

В случае передачи параметра по значению формальному параметру функции присваивается копия значения фактического параметра, и все изменения внутри функции никак не отражаются на фактическом параметре. Это значит, что внутри функции изменения происходят с копией фактического параметра, а не с ним самим.

Передача параметра по ссылке посредством указателя

Такой способ передачи параметра в функцию требует указания адреса фактического параметра, который при входе в функцию присваивается формальному параметру. Внутри функции открывается доступ к фактическому параметру, причем копия не создается. Все изменения, произошедшие при работе функции, отражаются на состоянии фактического параметра.

Передача параметра по ссылке посредством ссылки

Ссылка (reference) представляет собой указатель с константным значением адреса и так же, как указатель, является переменной, которая содержит адрес другой переменной. Однако в отличие от указателя сама ссылка не имеет адреса. Для получения данных по ссылке не надо пользоваться операцией разыменовывания `*`.

Ссылки можно использовать как псевдонимы для других переменных. В этом случае ссылки называют независимыми. При объявлении независимая ссылка обязательно должна получить значение, которое невозможно изменить в ходе выполнения программы.

Формат объявления независимой ссылки следующий:

```
тип & идентификатор_псевдоним = идентификатор ;
```

Независимые ссылки практически не используются в программах, так как в этом нет особенной необходимости. Предназначение этого типа заключается в организации передачи параметров в функцию. При передаче параметров посредством ссылок происходит то же самое, что и при передаче посредством указателей. Ссылка становится вторым именем фактического параметра и все изменения, происходящие внутри функции, сказываются также на фактическом параметре. В отличие передачи по значению копия не создается, следовательно, значение параметра после обработки в функции сохранит свое новое значение.

Параметры по умолчанию

При вызове функции можно не указывать фактические параметры, если в заголовке функции с помощью оператора присваивания заданы значения параметров по умолчанию. В прототипе функции после типа и знака = записывается константное значение, которое будет использоваться в теле функции в том случае, когда при вызове не указывается фактический параметр. В определении функции указывается только тип и идентификатор параметра по умолчанию. Если в определении функции записать значение, компилятор сообщит об ошибке переопределения заданного по умолчанию параметра.

Параметры по умолчанию следует располагать в конце списка параметров, так как опускать при вызове можно только параметры, стоящие в конце списка при объявлении функции. Например, если в прототипе имеется в конце списка три значения по умолчанию, то можно не указывать при вызове последний параметр, два последних или все три, но невозможно указать первый и последний без указания второго параметра.

Передача массива в качестве параметра функции

При передаче массива в качестве параметра функции ей передается адрес массива. Получая адрес, функция имеет доступ ко всем элементам массива и может их модифицировать. Если функция не должна изменять значения

элементов массива, необходимо воспользоваться спецификатором const, который сообщит, что массив постоянный и не подлежит изменению.

Часто для работы с массивом при помощи функции в качестве еще одного параметра передается так же и размер массива, чтобы функция могла обрабатывать заданное число его элементов.

Примеры функций

Раздел книги включает ряд примеров с функциями разного типа, которые получают разнотипные параметры и используют различные способы передачи параметров.

Листинг 13.1. Пример функции вывода состояния одномерного массива

```
#include <iostream>
using namespace std ;
// прототип функции вывода состояния
void view ( const int*, int = 5 ) ;
// главная функция
int main ( )
{
        int const SIZE = 12 ;
        int a [ SIZE ] = { 1, 2, 3, 4, 5, 6 } ;
        cout << "Using default value\n" ;
        view ( a ) ;          // вызов по умолчанию
        cout << "Using identifier\n" ;
        view ( a, SIZE ) ;    // вызов с использованием идентификатора
        cout << "Using constant\n" ;
        view ( a, 3 ) ;       // вызов с использованием константы
        return 0 ;
}
// определение функции вывода состояния
void view ( const int* p, int size )
{
        int n = 5 ;
        for ( int i = 0; i < size; i++ )
        {
                for ( int j = 0; j < n && ( i + j ) < size; j++ )
                        cout << p [ i + j ] << '\t' ;
```

```
            i += n - 1 ;
            cout << endl ;
        }
}
```

Функция `view ( )` в листинге 13.1 не возвращает никакого значения, поэтому ее тип `void`. Эта функция выводит состояние одномерного массива целых чисел так, что в каждой строке выводится пять элементов. Параметрами функции являются указатель на константу `int*` и количество элементов, которое необходимо вывести `int = 5`. Первый параметр обеспечивает невозможность модификации элементов массива внутри функции. Второй параметр имеет значение, которое определяет число выводимых элементов по умолчанию.

В главной функции `main ( )` объявляется константа `SIZE` для определения максимального числа элементов в массиве. Далее объявляется массив целых чисел `a`, первые шесть элементов которого инициализируются указанными значениями, а остальные шесть согласно правилам инициализации — нулевыми значениями.

Первый вызов функции `view ( a )` приведет к тому, что параметр `size` функции получит значение по умолчанию, равное 5. Формальный указатель `p` станет равным адресу массива `a`. В результате будут выведены первые пять элементов массива `a`, как показано на рис. 13.1.

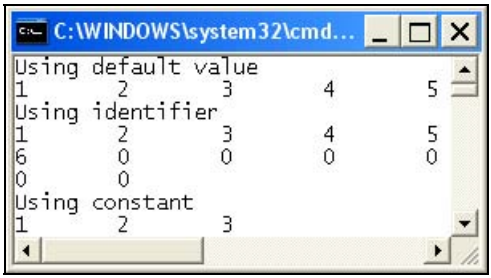


Рис. 13.1. Результаты исполнения функции вывода состояния одномерного массива

Второй вызов функции `view ( a, SIZE )` использует тот же массив, поэтому формальный параметр `p` получит значение адреса массива `a`. Параметр `size` при этом вызове получает значение константы `SIZE` и становится равным 12. Функция выводит три строки со значениями элементов массива.

Последний вызов `view ( a, 3 )` ограничивает число выводимых элементов константой 3, значение которой становится значением формального параметра `size`.

Для вывода состояния другого целочисленного массива следует определить его в главной функции, после чего передать имя массива и количество выводимых элементов при вызове функции `view ( )` одним из рассмотренных способов.

Листинг 13.2. Пример функции вывода состояния матрицы

```
#include <iostream>
using namespace std ;
const int COL = 3 ;
// прототип функции вывода состояния
void show ( const short [ ] [ COL ], int = 3 ) ;
// главная функция
int main ( )
{
        short a [ 3 ] [ COL ] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 } ;
        short b [ 4 ] [ COL ] = { 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4 } ;
        // вызов по умолчанию
        cout << "Matrix a\n" ; show ( a ) ;
        cout << "Part of matrix b\n" ; show ( b ) ;
        // вызов с использованием констант
        cout << "Part of matrix a\n" ; show ( a, 2 ) ;
        cout << "Matrix b\n" ; show ( b, 4 ) ;
        return 0 ;
}
// определение функции вывода состояния
void show ( const short m [ ] [ COL ], int row )
{
        for ( int i = 0; i < row; i++ )
        {
                for ( int j = 0; j < COL; j++ )
                        cout << m [ i ] [ j ] << '\t' ;
                cout << endl ;
        }
        cout << endl ;
}
```

Функция `show ( )` в листинге 13.2 выводит по строкам двумерный массив коротких целых чисел, в котором может быть любое количество строк, но количество столбцов ограничено глобальной константой `COL` и равно 3.

Минимально функция выводит три строки массива, что задается параметром по умолчанию. Если необходимо вывести все строки, при вызове функции необходимо явно указать их число. При передаче двумерного массива в качестве аргумента обязательно следует задавать второй размер. При вызове функции указывается только имя массива, например, `show ( a )`. Несмотря на то, что функция использует другое имя `m`, на самом деле она работает с оригинальным массивом. Это положение справедливо только для массивов, так как из-за возможных больших размеров массивов дублирование элементов может занять много времени и занять значительное пространство в оперативной памяти. Результат исполнения программы показан на рис. 13.2.

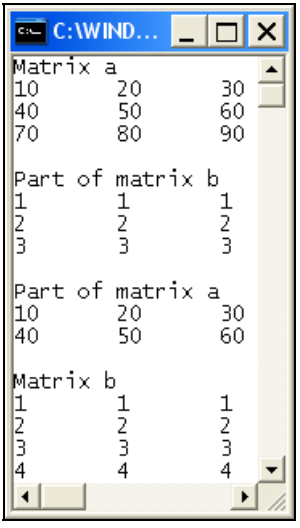


Рис. 13.2. Результаты исполнения функции вывода состояния матрицы

Листинг 13.3. Пример функции, возвращающей значение

```
#include <iostream>
using namespace std ;
// прототипы функций
long double fact ( const int& ) ;
long double factRec ( const int& ) ;
// главная функция
int main ( )
{
```

```
        int x = 8 ;
        cout << fact ( x ) << endl ;
        x = fact ( x ) ; cout << x << endl ;
        long double y ;
        y = factRec ( 150 ) ; cout << y << endl ;
        return 0 ;
}
// определение вычисления без рекурсии
long double fact ( const int& n )
{
        int i ( n ) ;
        long double res = 1 ;
        while ( i > 1 ) res *= i-- ;
        return res ;
}
// определение вычисления с рекурсивным вызовом
long double factRec ( const int& n )
{
        if ( n == 1 || n == 0 ) return 1 ;
        long double res ;
        res = factRec ( n - 1 ) * n ;
        return res ;
}
```

В листинге 13.3 приведены две функции вычисления факториала целого числа, каждая из которых реализует собственный алгоритм. Выбранный тип обеих функций поддерживается типом `long double` возвращаемого значения `res`. Тип `long double` обеспечит вычисление факториалов достаточно больших чисел. Параметр функций задан при помощи ссылки на константу целого типа `const int&` и определяет число, факториал которого вычисляется.

Функция `fact ( )` вычисляет результат в цикле `while`, который управляет изменением значения локальной переменной `i`, уменьшая ее значение на 1 после подсчета произведения `res` на текущее значение `i`. Перед началом цикла переменная `res` получает значение 1, а переменная `i` инициализируется значением параметра `n`. Использовать параметр `n` в цикле `while` невозможно, так как его тип не позволяет изменять значение внутри функции. Оператор `return` возвращает полученное значение `res` и передает управление в функцию `main ( )`.

Функция `factRec ( )` реализует рекурсивный алгоритм. В начале функции проверяется полученное функцией значение. Если оно равно 0 или 1, возвра-

щается 1, так как факториал этих чисел равен 1. В противном случае функция возвращает значение `factRec ( n - 1 ) * n`. Для вычисления этого выражения функция вызывается `n - 1` раз до тех пор, пока `n` не станет равным 1. С этого момента начинается выполнение операторов `return`, относящихся к разным вызовам функции `factRec ( )`. При рекурсивном вызове локальные переменные и параметры размещаются в стеке, новая копия функции не создается. Копии локальных переменных и параметров удаляются из стека после возврата управления из каждого рекурсивного вызова.

В главной функции `main ( )` вычисляется значение факториала переменной `x`, значение которой 8. Первый вызов `fact ( x )` используется для вывода вычисленного значения. Результат вычисления нигде не сохраняется. Следующий вызов функции `fact ( )` использует первоначальное значение переменной `x`, а оператор присваивания замещает его на значение, вычисленное функцией. Результат вычислений выводится на экран. Вызов рекурсивной функции `factRec ( )` сохраняется в переменной `y` и тоже выводится. В качестве параметра при вызове указывается константа 150. В итоге будут выведены следующие значения:

```
40320
40320
5.71388e+262.
```

Листинг 13.4. Пример функции, возвращающей указатель

```
#include <iostream>
using namespace std ;
char* fSubStr ( char*, char* ) ;
int main ( )
{
        char* s = "A Toyota! Race fast... safe car: a Toyota" ;
        char* ss = "Toyota" ;
        cout << "String:\t\t" << s << endl ;
        cout << "Substring:\t" << ss << endl ;
        char* p ( s );
        while ( p = fSubStr ( p, ss ) )
        {
                cout << "Substring entry:\t" << p << endl ;
                p += strlen ( ss ) ;
        }
        return 0 ;
}
```

```
// определение функции поиска подстроки в строке
char* fSubStr ( char* ps, char* pss )
{
        if ( ! *pss || strlen ( pss ) > strlen ( ps ) )   return 0 ;
        char *p, *r ;
        while ( *ps )
        {
                for ( p = ps, r = pss; *r && *p == *r; p++, r++ ) ;
                if ( ! *r )    return ps ;
                ps++ ;
        }
        return 0 ;
}
```

Функция `fSubStr ( )` в листинге 13.4 ищет в строке первое вхождение подстроки. В случае успеха возвращает указатель на вхождение подстроки, иначе возвращает 0. Параметрами функции являются указатель на строку `ps` и указатель на подстроку `pss`. Чтобы вернуть указатель, в объявлении функции указан соответствующий тип возвращаемого значения `char*`. Алгоритм функции рассмотрен в *главе 11* в примере подсчета количества вхождений подстроки в строку. В главной функции `main ( )` организован цикл, который позволяет вывести все вхождения подстроки `Toyota` в строку. Цикл `while` повторяется до тех пор, пока функция `fSubStr ( )` возвращает ненулевой указатель. Найденное вхождение выводится. Фактический параметр `p` для каждого вызова корректируется на длину искомой подстроки. Результаты исполнения программы приводятся на рис. 13.3.

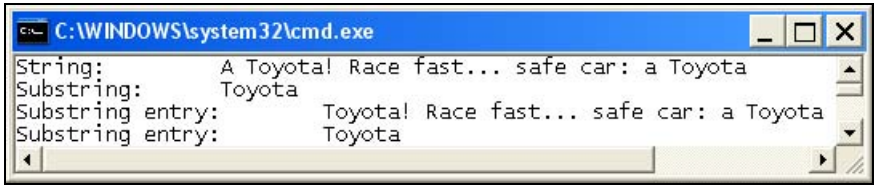


Рис. 13.3. Результаты исполнения функции поиска подстроки в строке

Листинг 13.5. Пример передачи параметров по значению и по ссылке

```
#include <iostream>
using namespace std ;
// прототипы функций
```

```
void swapV ( int, int ) ;
void swapP ( int*, int* ) ;
void swapR ( int&, int& ) ;
void reverseArray ( int*, int = 10 ) ;
void view ( const int*, int = 5 ) ;
// главная функция
int main ( )
{
       int aIn, bIn ;
       cout << "Enter a -> " ; cin >> aIn ;
       cout << "Enter b -> " ;      cin >> bIn ;
       // перестановка по значению
       int a = aIn, b = bIn ;       swapV ( a, b ) ;
       cout << "Result swapV ( ): " ; cout << a << ' ' << b << endl ;
       // перестановка посредством указателей
       a = aIn, b = bIn ;           swapP ( &a, &b ) ;
       cout << "Result swapP ( ): " ; cout << a << ' ' << b << endl ;
       // перестановка посредством ссылок
       int &ra = aIn, &rb = bIn ;
       swapR ( ra, rb ) ;    // swapR ( aIn, bIn ) ;
       cout << "Result swapR ( ): " ; cout << aIn << ' ' << bIn << endl ;
       // реверс массива
       int ar [ 20 ] = { 1, 2, 3, 4, 5, 6 } ;
       cout << "\nInitial array\n" ;       view ( ar, 5 ) ;
       cout << "\nReverse array\n" ;
       reverseArray ( ar, 5 ) ;            view ( ar, 5 ) ;
       cout << "\nOne more reverse\n" ;
       reverseArray ( ar, 12 ) ;           view ( ar, 12 ) ;
       return 0 ;
}
// перестановка по значению (только внутри функции)
void swapV ( int x, int y )
{      int t = x ; x = y ;  y = t ;       }
// перестановка при помощи указателей
void swapP ( int* px, int* py )
{      int t = * px ; * px = * py ; * py = t ;    }
// перестановка при помощи ссылок
void swapR ( int& rx, int& ry )
```

```
{       int t = rx ; rx = ry ; ry = t ;     }
// реверс массива
void reverseArray ( int* p, int size )
{
        for ( int i = 0, j = size - 1; i < j; i++, j-- )
                swapP ( p + i, p + j ) ; // swapR ( p [ i ], p [ j ] ) ;
}
// вывод состояния массива
void view ( const int* p, int size )
{
        int n = 5 ;
        for ( int i = 0; i < size; i++ )
        {
                for ( int j = 0; j < n && ( i + j ) < size; j++ )
                        cout << p [ i + j ] << '\t' ;
                i += n - 1 ;
                cout << endl ;
        }
}
```

В листинге 13.5 рассматриваются разные способы передачи параметров для перестановки значений двух переменных и организация вызова функций для этих способов. Результаты исполнения программы представлены на рис. 13.4.

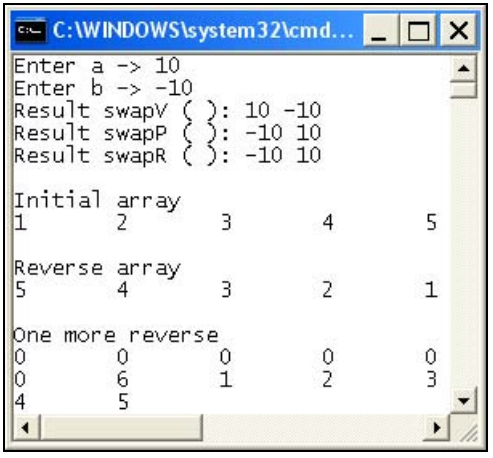


Рис. 13.4. Результаты исполнения программы с передачей параметров по значению и по ссылке

Функция `swapV ( )` хотя и выполняет поставленную задачу внутри функции, но не в состоянии обеспечить фактическую перестановку значений. Параметры передаются в эту функцию по значению, следовательно, функция работает с их копиями. Результат вызова `swapV ( a, b )` никак не отразится на переменных `a` и `b`, которые сохранят свои первоначальные значения.

Функция `swapP ( )` использует указатели на переставляемые переменные, поэтому при вызове указываются адреса этих переменных `swapP ( &a, &b )`. При таком способе передачи параметров копии внутри функции не создаются, и она имеет непосредственный доступ к значениям переменных. При перестановке используется разыменовывание указателей. После выполнения функции значения `a` и `b` поменяются местами.

Функция `swapR ( )` использует в качестве параметров ссылки. В примере перед вызовом функции объявляются независимые ссылки `ra` и `rb`, после чего происходит вызов функции `swapR ( ra, rb )`. Поскольку независимая ссылка не что иное, как второе имя переменной, объявление ссылок было не обязательным. Можно было обратиться к функции, указывая имена переменных `swapR ( aIn, bIn )`. Результат один и тот же при обоих вызовах, так как меняются местами переменные `aIn` и `bIn`. При использовании ссылок копии не создаются, и в разыменовывании нет нужды.

Функция `reverseArray ( )` переписывает элементы одномерного массива целых чисел в обратном порядке. Параметрами функции являются указатель на массив и размер массива. Второй параметр имеет значение по умолчанию. Для реверса массива в теле функции используется вызов функции `swapP ( p + i, p + j )` с указанием адресов переставляемых элементов массива. Этот вызов можно заменить на вызов функции `swapR ( p [ i ], p [ j ] )`, тогда на месте фактических параметров следует указать сами элементы, подлежащие перестановке.

Функция вывода состояния `view ( )` рассмотрена ранее в начале главы.

Функции обработки символов

В разделе представлены стандартные функции для обработки символов из библиотеки функций. Для символьных функций предусмотрен заголовок `<cctype>`, который в Visual C++ 2005 подключать в текст программ не обязательно.

Функции классификации символов

Символьные функции работают с переменными типа `char`. Параметром каждой функции считается целое число, которое она автоматически преобразует в тип `unsigned char`. Фактический параметр типа `char` в вызове функции

автоматически приводится к целому типу. Если значение параметра входит в состав проверяемого функцией множества символов, возвращается истина, иначе — ложь. В табл. 13.1 включены прототипы функций классификации и описание проверяемого символа.

Таблица 13.1. *Функции классификации символов*

Прототип	Проверяемый символ
`isalnum ( int c )`	Алфавитно-цифровой символ: цифра, прописная, строчная буква
`isalpha ( int c )`	Алфавитный символ: прописная или строчная буква
`isascii ( int c )`	Символ ASCII `0 <= c <= 0x7e`
`iscntrl ( int c )`	Управляющий код ASCII `0x7f` или от `0x00` до `0x1f`
`isdigit ( int c )`	Цифра от `0` до `9`
`isgraph ( int c )`	Печатный символ, кроме пробела `0x21 <= c <= 0x7e`
`islower ( int c )`	Строчная буква от `a` до `z`
`isprint ( int c )`	Печатный символ или пробел `0x20 <= c <= 0x7e`
`ispunct ( int c )`	Знак пунктуации
`isspace ( int c )`	Пробельный символ: пробел, табуляция, возврат каретки, перевод строки
`isupper ( int c )`	Прописная буква от `A` до `Z`
`isxdigit ( int c )`	Шестнадцатеричная цифра: от `0` до `9`, от `A` до `F` или от `a` до `f`

Листинг 13.6. Пример использования функций классификации символов

```
#include <iostream>
using namespace std ;
void show ( const char&, const char* ) ;
void delSpace ( char* ) ;
int main ( )
{
        char* pS = "x = 20*y ;" ;
        cout << "Initial string:\t" << pS << endl << endl ;
        // анализ символов строки
        char* p = pS ;
```

```
        while ( *p )
        {
                if ( isalpha ( *p ) ) show ( *p, "letter" ) ;
                if ( isdigit ( *p ) ) show ( *p, "digit" ) ;
                if ( ispunct ( *p ) ) show ( *p, "punctuation symbol" ) ;
                if ( iscntrl ( *p ) ) show ( *p, "control symbol" ) ;
                else if ( isspace ( *p ) ) show ( *p, "space" ) ;
                p++ ;
        }
        // преобразование строки
        char s [ ] = "What is done \t\n\rcannot \t\t be undone." ;
        cout << "\nInitial string: " << s << endl ;
        delSpace ( s ) ; cout << "Result string: " << s << endl ;
        return 0 ;
}
// функция вывода сообщения о символе
void show ( const char& c, const char* msg )
{       cout << '\'' << c << "\'\t" << msg << endl ;       }
// функция удаления лишних пробельных символов из строки
void delSpace ( char* p )
{
        char* r = p ;
        while ( *p )
                if ( isspace ( *p ) )
                {
                        * r++ = *p++ ;          // запись первого пробела
                        while ( isspace ( *p ) ) p++ ;
                }
                else * r++ = *p++ ;             // запись непробельного симво-
ла
        * r = '\0' ;
}
```

Главная функция в листинге 13.6 анализирует символы строки `x = 20*y ;` в цикле `while`. Доступ к очередному элементу строки организован через указатель `p`, который до начала цикла указывает на начало исходной строки `pS`. Для анализа принадлежности текущего символа к тому или иному множеству поочередно вызываются разные функции классификации символов. Если

функция возвращает 1, вызывается функция вывода сообщения о символе, параметрами которой являются символ и сообщение. Чтобы после вызова функции `isspace ( )` для табуляции не выводилось сообщение `space`, используется оператор `if else if`, который подавляет вывод этого сообщения. За вывод символа и соответствующего сообщения отвечает функция `show ( )`.

Функция `delSpace ( )` удаляет из строки лишние пробельные символы. В теле функции первый оператор цикла `while` проверяет, является ли символ пробельным. Если `isspace ( )` возвращает истину, то этот первый пробел переписывается в результирующую строку при помощи указателя `r`, а все последующие пробельные символы игнорируются в новом цикле `while`, который управляет изменением указателя `p` на символы в исходной строке. Если же `isspace ( )` возвращает ложь, то текущий непробельный символ записывается в строку результата. Последний оператор функции `delSpace ( )` записывает по нужному адресу нулевой байт. Результаты исполнения программы показаны на рис. 13.5.

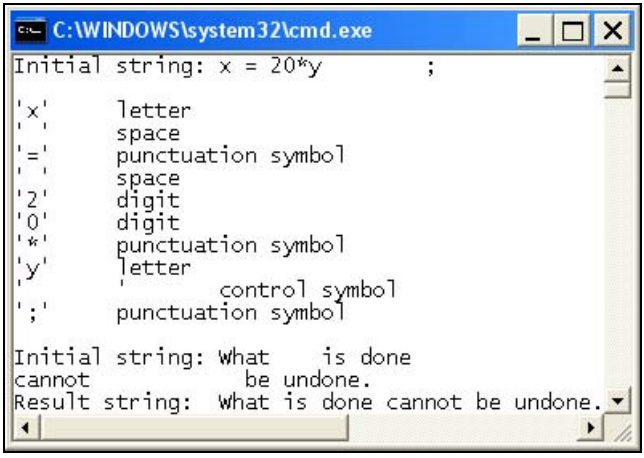


Рис. 13.5. Результаты выполнения программы с функциями классификации символов

Функции преобразования символов

Для преобразования символов в программе используются функции, краткое описание которых приводится в табл. 13.2. Все эти функции точно так же, как и функции классификации символов, в качестве параметра имеют целое число, автоматически приводимое к типу `unsigned char`. Фактический параметр функций должен иметь символьный тип `char` или `unsigned char`.

Таблица 13.2. *Функции преобразования символов*

Прототип	Краткое описание
`__toascii ( int c )`	Преобразует целое `c` и возвращает семибитовое значение кода ASCII
`tolower ( int c )`	Если `c` является буквой от `A` до `Z`, возвращает ее строчный эквивалент
`_tolower ( int c )`	Более быстрая ограниченная регионом версия функции `tolower ( )`
`toupper ( int c )`	Если `c` является буквой от `a` до `z`, возвращает ее прописной эквивалент
`_toupper ( int c )`	Более быстрая ограниченная регионом версия функции `toupper ( )`

Листинг 13.7. Пример использования функций преобразования символов

```
#include <iostream>
using namespace std ;
int main ( )
{
       // преобразование символов строки
       char s [ 20 ] = "As Old As The Hills" ;
       cout << "Initial string:\t\t" << s << endl ;
       char r [ 20 ] ; int i ;
       for ( i = 0; s [ i ]; i++ )
              if ( islower ( s [ i ] ) ) r [ i ] = _toupper ( s [ i ] ) ;
              else r [ i ] = s [ i ] ;
       r [ i ] = '\0' ;
       cout << "Only capitals:\t\t" << r << endl ;
       for ( i = 0; r [ i ]; i++ )
              if ( isupper ( r [ i ] ) ) r [ i ] = _tolower ( r [ i ] ) ;
       cout << "Only lower case:\t" << r << endl ;
       // получение ASCII кода символа
       cout << "\nchar\tASCII\n" ;
       char c [ 6 ] = { 'A', 'B', 'C', '+', '1', '2' } ;
       for ( i = 0; i < 6; i++ )
              cout << c [ i ] << '\t' << __toascii ( c [ i ] ) << endl ;
       return 0 ;
}
```

В листинге 13.7 в первом цикле `for` формируется строка `r` из строки `s` путем преобразования строчных символов исходной строки `s` в их прописной эквивалент. После цикла в строку `r` записывается нулевой байт, чтобы ограничить полученную строку. Во втором цикле `for` прописные символы строки `r` заменяются строчным эквивалентом. Символ завершения строки не записывается, так как строка существует и уже содержит этот символ. В конце программы демонстрируется вызов функции получения ASCII кода символа `__toascii ( )`. Результаты работы программы представлены на рис. 13.6.

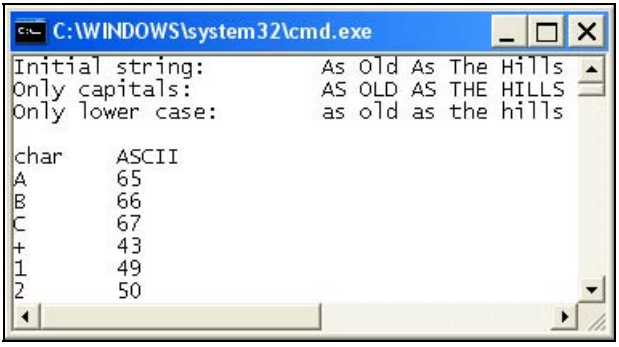


Рис. 13.6. Результаты выполнения программы с использованием функций преобразования символов

Основные функции обработки строк

Здесь рассматриваются важнейшие универсальные функции обработки строк из библиотеки функций. Для их использования в программе необходимо включить заголовок `<string>` в исходный текст. Все рассматриваемые в разделе функции применяются к строкам, которые заканчиваются нулевым байтом.

В табл. 13.3 включены названия основных строковых функций и их назначение.

Таблица 13.3. *Основные строковые функции*

Название функции	Назначение функции
`strcat_s ( )`	Сцепить две строки
`strcmp ( )`	Сравнить две строки в лексикографическом порядке
`strcpy_s ( )`	Копировать одну строку в другую
`strlen ( )`	Определить длину строки

Функция *strcat_s ( )*

Прототип:

```
errno_t strcat_s ( char *strDest, size_t sizeInBytes, const char *strSrc ) ;
```

Описание: Объединяет инициализированную строку-приемник `strDest` и исходную строку `strSrc`, присоединяя ее к концу строки `strDest`. Нулевой байт строки `strDest` при объединении замещается первым символом исходной строки. Функция возвращает 0 при успешном объединении и код ошибки при неудаче. Тип возвращаемого значения `errno_t` по сути является типом `int` и используется для функционального типа, который имеет дело с кодами ошибки `errno`.

Параметры:

- `char* strDest` — указатель на инициализированную строку-приемник;
- `size_t sizeInBytes` — размер строки-приемника. Тип `size_t` является по существу типом `unsigned int`;
- `const char *strSrc` — указатель на строку, которая присоединяется.

Примечание

Программист должен сам позаботиться о необходимом размере строки-приемника, который должен быть равен сумме длин обеих строк плюс 1 байт.

Листинг 13.8. Пример использования функции конкатенации строк

```
char s1 [ 20 ] = "I like" ;  char s2 [ 20 ] = " C++" ;
strcat_s ( s1, 20, s2 ) ;                  // в s1 I like C++
char* p = new char [ 80 ] ;  * p = '\0' ;
strcat_s ( p, 80, s1 ) ;                   // в p I like C++
strcat_s ( p, 80, " very much!" ) ;        // в p I like C++ very much!
```

Результатом первого вызова функции `strcat_s ( )` (листинг 13.8) будет строка `s1`, где будет записано `I like C++`, строка `s2` сохранит свое состояние. Указатель `p` получает значение выделенного блока памяти. В первый байт этого блока записывается нулевой байт. Второй вызов функции объединяет строку с адресом `p` со строкой `s1` так, что они становятся равными друг другу. Строка-приемник обязательно должна быть инициализирована, поэтому туда и был записан символ `\0`. Последний вызов `strcat_s ( )` делает строку, на которую указывает `p`, равной `I like C++ very much!`, приписывая в конец существующей строки константную строку-литерал.

Функция *strcmp ( )*

Прототип:

```
int strcmp ( const char *s1, const char *s2 ) ;
```

Описание: Сравнивает две строки. Возвращает отрицательное значение, если `s1` в лексикографическом порядке меньше `s2`; нуль при равенстве строк; положительное значение, если `s1` больше `s2`.

Примечание

Все символы представляются внутри компьютера как численные коды. Когда строки сравниваются, компьютер сравнивает численные коды символов в строке.

Параметры:

- `const char* s1` — указатель на первую сравниваемую строку;
- `const char *s2` — указатель на вторую сравниваемую строку (листинг 13.9).

Листинг 13.9. Пример использования функции сравнения строк

```
char *p1 = "Good luck!" ;
char *p2 = "Good luck!" ;
char *p3 = "good luck" ;
cout << strcmp ( p1, p2 ) << endl ;        // выводится 0
cout << strcmp ( p1, p3 ) << endl ;        // выводится -1
cout << strcmp ( p3, p1 ) << endl ;        // выводится 1
```

Функция *strcpy_s ( )*

Прототип:

```
errno_t strcpy_s ( char *strDest, size_t sizeInBytes, const char *strSrc ) ;
```

Описание: Копирует исходную строку `strSrc` и ее нулевой байт в строку-приемник `strDest`, перезаписывая символы строки-приемника. Возвращается 0 при удачном копировании, иначе возвращается код ошибки.

- Параметры:
- `char* strDest` — указатель на строку-приемник, перезаписываемую исходной строкой;
- `size_t sizeInBytes` — размер строки-приемника;
- `const char *strSrc` — указатель на копируемую строку, которая заканчивается нулевым байтом (листинг 13.10).

Примечание

Программист должен сам позаботиться о необходимом размере строки-приемника, который должен быть равен длине исходной строки плюс 1 байт.

Листинг 13.10. Пример использования функции копирования строк

```
char *p = "Slavic bazar" ;
char s1 [ 30 ] , s2 [ 13 ] ;
strcpy_s ( s1, 30, p ) ;
cout << s1 << endl ;          // выводится Slavic bazar
strcpy_s ( s2, 13, p ) ;
cout << s2 << endl ;          // выводится Slavic bazar
strcpy_s ( s2, "Vitebsk" ) ;
cout << s2 << endl ;          // выводится Vitebsk
```

В результате второго копирования в строку `s2` ее состояние изменится: вместо `Slavic bazar` в ней будет записано `Vitebsk`. Если у строки-приемника `s1` или `s2` длина будет менее 13 байтов, поведение функции `strcpy_s ( )` будет непредсказуемым. При копировании строк следует внимательно следить за размером строки, куда записывается результат.

Функция *strlen ( )*

Прототип:

```
size_t strlen ( const char *str ) ;
```

Описание: Определяет длину строки. Возвращает число символов в строке без учета нулевого байта.

Параметр:

`const char* str` — указатель на завершающуюся нулевым байтом строку (листинг 13.11).

Листинг 13.11. Пример использования функции определения длины строки

```
char *p = "Football world championship 2006" ;
char s1 [ ] = "Part of the game" ;
char s2 [ 80 ] = "What? Where? When?" ;
cout << strlen ( p ) << endl ;      // выводится 32
cout << strlen ( s1 ) << endl ;     // выводится 16
cout << strlen ( s2 ) << endl ;     // выводится 18
cout << strlen ( "" ) << endl ;     // выводится 0
```

Служебные функции преобразования строк

В разделе рассматриваются служебные функции преобразования строк из библиотеки утилит общего назначения с заголовком <cstdlib>, обязательно подключаемым в программу при использовании этих функций, краткие сведения о которых содержит табл. 13.4. Функции преобразования строк задают значение встроенной глобальной целочисленной переменной errno. При возникновении ошибки эта переменная позволяет получить доступ к детальной информации о возникшей проблеме. Для получения кода ошибки можно воспользоваться функцией _get_errno ( ). Известные значения переменной errno расположены в библиотеке с заголовком <cerrno>.

Таблица 13.4. *Функции преобразования строк*

Название функции	Назначение функции
atoi ( )	Преобразовать ASCII-строку в целое число
atof ( )	Преобразовать ASCII-строку в число с плавающей точкой
strtod ( )	Преобразовать строку в число с плавающей точкой
atol ( )	Преобразовать ASCII-строку в длинное целое число
strtol ( )	Преобразовать строку в длинное целое число

Функция *atoi ( )*

Прототип:

```
int atoi ( const char *str ) ;
```

Описание: Преобразует строку в значение типа int. Возвращает значение или 0, если строку невозможно преобразовать. Функция останавливает чтение строки на первом символе, который не может интерпретироваться как часть числа. Таким символом может быть разделитель, знак пунктуации и не являющийся цифрой символ. В Visual C++ 2005 при переполнении результата устанавливается код ошибки ERANGE.

Параметр:

const char* str — указатель на преобразуемую строку (листинг 13.12).

Примечание

Строка может содержать знак плюс или минус, одну или большее количество цифр, а также символы пробела и табуляции. Последние два символа при преобразовании строки игнорируются.

Листинг 13.12. Пример использования функции `atoi ( )`

```
#include <iostream>
#include <cstdlib>
#include <cerrno>
using namespace std ;
void view ( const char*, const char* ) ;
int main( )
{
        errno_t err ; char *str = 0 ;       int v ( 0 ) ;
        str = " -12345+ " ;          v = atoi ( str ) ;
        view ( "atoi", str ) ;       cout << v ;
        str = "+2147483647" ;        v = atoi ( str ) ;
        view ( "atoi", str ) ;       cout << v ;
        // переполнение при преобразовании
        str = "3333333333" ;         v = atoi ( str ) ;
        view ( "atoi", str ) ;       cout << v ;
        _get_errno ( &err ) ;
        if ( err == ERANGE ) cout << "\tOverflow!\n" ;
        return 0 ;
}
// вывод названия функции и строки преобразования
void view ( const char* f, const char* s )
{       cout << '\n' << f << " ( \"" << s << "\" ) = " ;  }
```

Результаты преобразований показаны на рис. 13.7.

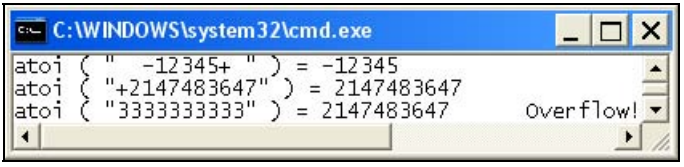


Рис. 13.7. Результаты исполнения преобразований при помощи функции `atoi ( )`

Функция *atof ( )*

Прототип:

```
double atof ( const char *str ) ;
```

Описание: Преобразует строку в число с плавающей точкой. Возвращает значение типа `double`. Если значение функции превышает пределы допустимого диапазона чисел, возникает переполнение. В этом случае возвращается константа `±HUGE_VAL`, для использования которой необходим заголовок `<cmath>` стандартной библиотеки математических функций. Если строка содержит некорректное представление числа с плавающей точкой, поведение функции не определено. В отдельных случаях при невозможности преобразования строки может возвращаться 0.0. Чтение строки останавливается на первом символе, который не может интерпретироваться как часть числа с плавающей точкой, или же на символе завершения конца строки.

Параметр:

`const char* str` — указатель на преобразуемую строку с нулевым байтом в конце (листинг 13.13).

Примечание

Строка может содержать знаки плюс или минус перед символом-цифрой, одну или большее количество цифр, символ десятичной точки, а также символы пробела и табуляции. Последние два символа при преобразовании строки игнорируются. Допускается научная форма записи числа в строке с использованием символов `d`, `D`, `e` и `E` перед порядком, заданным в виде целого числа со знаком.

Листинг 13.13. Пример использования функции `atof ( )`

```
// Пример использования функции atof ( )
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std ;
void view ( const char*, const char* ) ;
int main( )
{
       char *str = 0 ;              double d ( 0.0 ) ;
       str = "3333333333" ;         d = atof ( str ) ;
       view ( "atof", str ) ;       cout << d ;
       str = "7.2431851239d108" ;   d = atof ( str ) ;
       view ( "atof", str ) ;       cout << d ;
       str = " -1798.15E-17 " ;     d = atof ( str ) ;
       view ( "atof", str ) ;       cout << d ;
       str = ",8" ;                 d = atof ( str ) ;
```

```
        view ( "atof", str ) ;        cout << d ;
        // переполнение при преобразовании
        str = "-1.7e+509" ;           d = atof ( str ) ;
        view ( "atof", str ) ;        cout << d ;
        if ( d == -HUGE_VAL || d == HUGE_VAL )
               cout << "\tOVERFLOW!" ;
        cout << endl ;
        return 0 ;
}
// вывод названия функции и строки преобразования
void view ( const char* f, const char* s )
{       cout << '\n' << f << " ( \"" << s << "\" ) = " ;  }
```

Для проверки переполнения при преобразовании строки в программе используется константа `HUGE_VAL`, которая определяет допустимое значение числа с плавающей точкой. Результаты преобразований показаны на рис. 13.8.

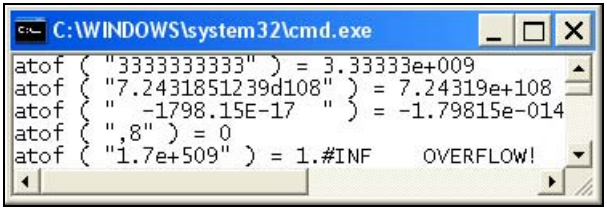


Рис. 13.8. Результаты исполнения преобразований при помощи функции `atof ( )`

Функция *strtod ( )*

Прототип:

```
double strtod ( const char *nptr, char **endptr ) ;
```

Описание: Преобразует содержащееся в строке представление числа с плавающей точкой в его двоичное представление типа `double`. В случае успеха возвращается полученный результат, иначе возвращается `±HUGE_VAL`, а глобальной переменной `errno` присваивается значение константы `ERANGE`. Константа обозначает выход за пределы допустимых значений. Если какой-либо из считанных символов не является элементом записи числа, процесс чтения строки прекращается. Это относится к разделителям, знакам пунктуации (кроме точки) и символам, отличным от `d`, `D`, `e` или `E`. Указатель `**endptr` устанавливается на остаток исходной строки, если таковой имеется.

Параметры:

- `const char*nptr` — указатель на преобразуемую строку;
- `char**endptr` — указатель на адрес символа, расположенного непосредственно после последнего символа в строке `nptr`, который участвует в преобразовании. Этот параметр является необязательным и используется при разборе строк, содержащих несколько представлений чисел с плавающей точкой, которые, возможно, отделяются друг от друга разделителями, запятыми или другими символами.

Листинг 13.14. Пример использования функции `strtod ( )`

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std ;
void view ( const char*, const char* ) ;
int main( )
{
       char *p = "3.14,-0.3e-409, .9, .3D550 89.36d150" ;
       double d ( 0.0 ) ;    char *pEnd ;
       while ( *p )
       {
              while ( *p == ',' ) p++ ;    // пропуск запятых
              d = strtod ( p, &pEnd ) ;
              view ( "strtod", p ) ;       cout << d ;
              if ( errno == ERANGE )
              {      cout << " Overflow!" ;       errno = 0 ;    }
              p = pEnd ;
       }
       cout << endl ;
       return 0 ;
}
// вывод названия функции и строки преобразования
void view ( const char* f, const char* s )
{      cout << '\n' << f << " ( \"" << s << "\" ) = " ;  }
```

В листинге 13.14 демонстрируется разбор строки, которая содержит несколько чисел с плавающей запятой, разделенных запятой и пробелом. В вызове функции `strtod ( )` указан адрес указателя `pEnd`, куда будет записываться адрес того символа, где закончилось чтение исходной строки.

Для проверки переполнения при преобразовании используется глобальная переменная `errno`. Когда преобразование заканчивается переполнением, функция `strtod ( )` устанавливает значение `errno`, равным константе `ERANGE`. Однако успешный вызов никак не влияет на состояние переменной `errno`. Именно поэтому после вывода сообщения о переполнении `errno` приравнивается нулю, чтобы анализ результата следующего вызова функции `strtod ( )` соответствовал действительности. Результаты преобразований показаны на рис. 13.9.

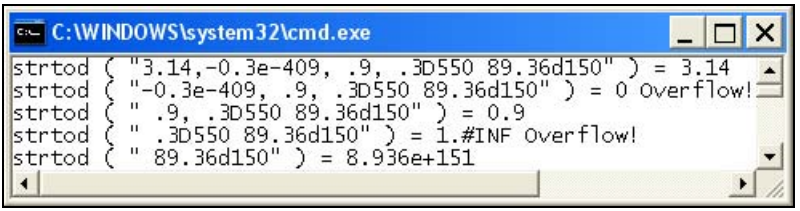


Рис. 13.9. Результаты исполнения преобразований при помощи функции `strtod ( )`

Функция *atol ( )*

Прототип:

```
long atol ( const char *str ) ;
```

Описание: Преобразует строку в значение типа `long`. Возвращает значение или 0, если строку невозможно преобразовать (листинг 13.15). Функция останавливает чтение строки на первом символе, который не может интерпретироваться как часть числа. Таким символом может быть разделитель, знак пунктуации и не являющийся цифрой символ. В Visual C++ 2005 при переполнении результата большим положительным значением возвращается константа `LONG_MAX`, при переполнении большим отрицательным значением — константа `LONG_MIN`, а глобальная переменная `errno` получает значение кода ошибки `ERANGE`.

Параметр:

`const char* str` — указатель на преобразуемую строку.

Примечание

Строка может содержать знак плюс или минус, одну или большее количество цифр, а также символы пробела и табуляции. Последние два символа при преобразовании строки игнорируются.

Листинг 13.15. Пример использования функции `atol ( )`

```
#include <iostream>
#include <cstdlib>
#include <cerrno>
using namespace std ;
void view ( const char*, const char* ) ;
int main( )
{
       char *str = 0 ;              int l ( 0L ) ;
       str = " 47.24318 " ;         l = atol ( str ) ;
       view ( "atol", str ) ;       cout << l ;
       str = " 215 483647" ;        l = atol ( str ) ;
       view ( "atol", str ) ;       cout << l ;
       str = "-3647" ;              l = atol ( str ) ;
       view ( "atol", str ) ;       cout << l ;
       str = "x" ;                  l = atol ( str ) ;
       view ( "atol", str ) ;       cout << l ;
       // переполнение при преобразовании
       str = "-2147483649" ;        l = atol ( str ) ;
       view ( "atol", str ) ;       cout << l ;
       if ( errno == ERANGE )
       {      cout << "\tOverflow!" ; errno = 0 ; }
       str = "2147483648" ;         l = atol ( str ) ;
       view ( "atol", str ) ;       cout << l ;
       if ( errno == ERANGE )       cout << "\tOverflow!\n" ;
       return 0 ;
}
// вывод названия функции и строки преобразования
void view ( const char* f, const char* s )
{      cout << '\n' << f << " ( \"" << s << "\" ) = " ;  }
```

На рис. 13.10 проиллюстрированы результаты исполнения преобразований строки в число типа `long`. Чтение строки `" 47.24318 "` закончится на символе точка. Строку `"x"` преобразовать невозможно, поэтому выводится 0. При возникновении переполнения для строки `"-2147483649"` выводится минимальное допустимое значение, для строки `"2147483648"` — максимальное допустимое значение.

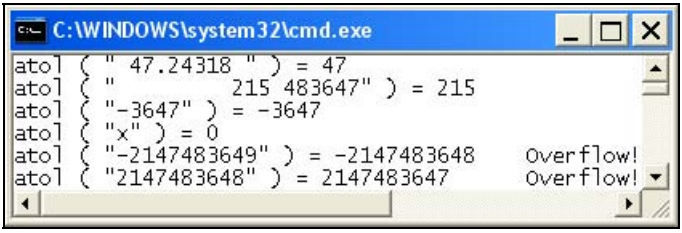


Рис. 13.10. Результаты исполнения преобразований при помощи функции `atol ( )`

Функция *strtol ( )*

Прототип:

```
long strtol ( const char *nptr, char **endptr, int base ) ;
```

Описание: Преобразует символьное представление значения типа `long` в его двоичное представление. Значение в строке может быть записано в десятичном виде или в виде любой другой системы счисления от двоичной до тридцатишестиричной. В случае успеха функция возвращает преобразованный результат. Указатель `**endptr` после преобразования устанавливается равным адресу, следующему за последним символом, который входит в символьное представление числа. В случае ошибки функция возвращает 0. При переполнении происходит то же, что и при использовании функции `atol ( )` (листинг 13.16).

Параметры:

- `const char*nptr` — указатель на преобразуемую строку;
- `char**endptr` — указатель на адрес символа, расположенного непосредственно после последнего символа в строке `nptr`, который участвует в преобразовании. Параметр является необязательным и используется при разборе строк, содержащих несколько представлений длинных целых чисел;
- `int base` — целое число, которое задает используемое для преобразования основание системы счисления и может принимать значения от 2 до 36. Буквы от `A` до `Z` распознаются как числовые символы для оснований, превышающих 10. Если `base` равен 0, функция будет автоматически распознавать и преобразовывать числа, используя стандарт языка C++, то есть десятичные числа должны начинаться с цифры, восьмеричные с 0, а шестнадцатеричные — с префикса `0x` или `0X`.

Листинг 13.16. Пример использования функции `strtol ( )`

```
#include <iostream>
#include <cstdlib>
using namespace std ;
void view ( const char*, const char* ) ;
int main( )
{
        char *p = "-101001234791" ;  long l ( 0L ) ;
        char *pEnd ;   int radix ;
        for ( radix = 2; radix < 9; radix *= 2 )
        {
                l = strtol( p, &pEnd, radix ) ;
                view ( "strtol", p ) ; cout << l ;
                cout << "\t( base = " << radix << " )" ;
                cout << "\n\tStopped at: " << pEnd ;
        }
        l = strtol( "0x777", 0, 0 ) ;
        view ( "strtol", "0x777" ) ; cout << l ;
        l = strtol( "0777", 0, 0 ) ;
        view ( "strtol", "0777" ) ; cout << l ;
        l = strtol( "777", 0, 0 ) ;
        view ( "strtol", "777" ) ; cout << l << endl ;
        return 0 ;
}
// вывод названия функции и строки преобразования
void view ( const char* f, const char* s )
{       cout << '\n' << f << " ( \"" << s << "\" ) = " ;  }
```

В цикле `for` изменяется значение переменной `radix`, определяющей основание системы счисления в вызове функции `strtol ( )`. За вызовом следует вывод результата преобразования, вывод состояния `radix` и остатка строки, адрес начала которого установлен функцией в указателе `pEnd`. После цикла `for` иллюстрируется вызов без указания указателя для хранения адреса символа, на котором закончилось преобразование строки, и без указания основания системы счисления. Полученные переменной `l` значения (рис. 13.11) обусловлены интерпретацией исходной строки. Для строки `0x777` выбирается основание 16, для строки `0777` — основание 8 и для строки `777` — основание 10.

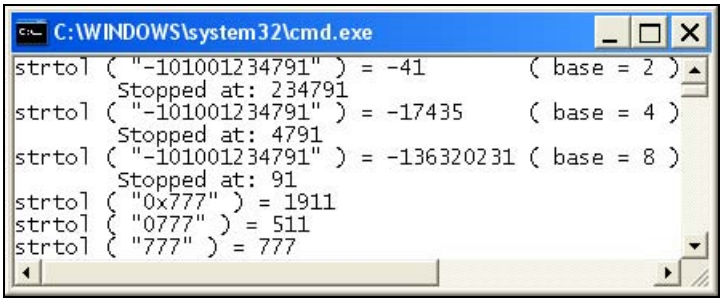


Рис. 13.11. Результаты исполнения преобразований при помощи функции `strtol ( )`

Перегрузка функций

Перегрузка функций (function overloading) представляет собой использование одного и того же имени для нескольких функций. При вызове перегруженной функции компилятор определяет соответствующую функцию путем анализа количества, типов и порядка следования параметров в вызове. Переопределение функций должно различаться либо другими типами параметров, либо их количеством. Если при одинаковом списке параметров функции будут различаться только типом возвращаемого значения, компилятор сообщит об ошибке неоднозначности. Параметры по умолчанию так же могут стать источником ошибок по той причине, что вызов одной функции может оказаться аналогичным вызову другой перегруженной функции.

Листинг 13.17. Пример перегрузки функций

```
#include <iostream>
using namespace std ;
void swapR ( int&, int& ) ;
void swapR ( char&, char& ) ;
void reverseArray ( int*, int = 10 ) ;
int* reverseArray ( const int*, int, int ) ;
char* reverseArray ( const char* ) ;
void view ( const int*, int = 5 ) ;
int main ( )
{
        int ar [ 10 ] = { 1, 2, 3, 4, 5, 6 } ;
        cout << "Initial array\n" ;  view ( ar, 10 ) ;
        // реверс всего массива
```

```
        reverseArray ( ar ) ;
        cout << "\nReverse array\n" ;       view ( ar, 10 ) ;
        // реверс части массива
        int* p = reverseArray ( ar, 5, 9 ) ;
        cout << "\nReverse part array\n" ;  view ( p ) ;
        // реверс строки
        char* s = "NOMEL ON NOLEM ON" ;
        char* pr = reverseArray ( s ) ;
        cout << "\nInitial string\n" ;      cout << s << endl ;
        cout << "\nReverse string\n" ;      cout << pr << endl ;
        return 0 ;
}
// перестановка посредством ссылок
void swapR ( int& rx, int& ry )
{       int t = rx ; rx = ry ; ry = t ;     }
void swapR ( char& rx, char& ry )
{       char t = rx ; rx = ry ; ry = t ;    }
// реверс массива в самом себе
void reverseArray ( int* p, int size )
{
        for ( int i = 0, j = size - 1; i < j; i++, j-- )
                swapR ( p [ i ], p [ j ] ) ;
}
// формирование обратного массива из части другого массива
int* reverseArray ( const int* pIn, int top, int end )
{
        if ( end <= top ) return 0 ;
        int size = end - top + 1 ; int* pRes = new int [ size ] ;
        int i, j, ir, jr ;
        for ( i = top, j = end, ir = 0, jr = size - 1; i < j; )
        {
                pRes [ ir++ ] = pIn [ j-- ] ;
                pRes [ jr-- ] = pIn [ i++ ] ;
        }
        if ( i == j ) pRes [ ir ] = pIn [ i ] ;
        return pRes ;
}
// реверс строки
char* reverseArray ( const char* p )
```

```
{
        char* pOut = new char [ strlen ( p ) + 1 ] ;
        strcpy_s ( pOut, strlen ( p ) + 1, p ) ;
        char *pTop = pOut, *pEnd = pOut + strlen ( pOut ) - 1 ;
        while ( pTop < pEnd )
                swapR ( *pTop++, *pEnd-- ) ;
        return pOut ;
}
```

В листинге 13.17 есть две перегруженные функции: `swapR ( )` и `reverseArray ( )`. Функция `swapR ( )` перестановки значений перегружена для параметров ссылок на целый и символьный тип. Таким образом, функции различаются типом параметров из списка параметров. Версия со ссылками на целый тип вызывается внутри функции реверса массива целых чисел `swapR ( p [ i ], p [ j ] )`, а со ссылками на символьный тип — в функции реверса строки `swapR ( *pTop++, *pEnd-- )`. Второй вызов требует разыменования, так как `pTop` и `pEnd` являются указателями.

Перегруженная функция `reverseArray ( )` имеет три варианта реализации, которые различаются типом возвращаемого значения, типом параметров, а также их количеством.

Прототип `void reverseArray ( int*, int = 10 ) ;` объявляет функцию, которая используется для реверса массива целых чисел в самом себе. В главной функции `main ( )` она вызывается как `reverseArray ( ar )`, используя значение по умолчанию 10 для количества элементов.

Прототип

```
int* reverseArray ( const int*, int, int ) ;
```

отличается от предыдущего типом возвращаемого значения и не использует значений по умолчанию. Функция предназначена для создания нового обратного массива из части исходного. Адрес исходного массива задается первым параметром. Два последних параметра определяют начальный и конечный индекс элементов в исходном массиве, которые должны быть переписаны в новый массив в обратном порядке. Функция возвращает указатель на новый массив. Первый параметр является указателем на константу, что запрещает изменение элементов в исходном массиве. В предыдущем варианте такое решение невозможно потому, что реверс внутри массива требует перестановки существующих значений. Два последних параметра нельзя объявить параметрами по умолчанию, так как возникнет неопределенность, и компилятор не сможет различить вызовы этих функций.

Обработка массива начинается с проверки заданных в вызове индексов. Возвращается 0, если конечный индекс меньше или равен начальному. Иначе вычисляется размер необходимой памяти, динамически выделяется блок памяти в соответствии с вычисленным значением и запускается цикл `for`, в котором формируется новый массив. Большое количество переменных в цикле обусловлено тем, что необходимы два индекса `i`, `j` для перемещения по исходному массиву и еще два индекса `ir`, `jr` для перемещения по создаваемому массиву. Индекс `i` изменяется от значения, заданного параметром `top`, в сторону увеличения на 1 до тех пор, пока он будет меньше индекса `j`. Индекс `j` изменяется от значения, заданного параметром `end`, в сторону уменьшения на 1. Индекс `ir` изменяется от 0, так как индексация в массиве начинается именно с этого значения, и увеличивается на 1 для новой итерации цикла. Индекс `jr` уменьшается на 1 в конце каждой итерации, начиная от значения индекса последнего элемента нового массива `size − 1`. Когда количество подлежащих реверсу элементов нечетное, на выходе из цикла индексы `i` и `j` будут равны. При истинном отношении `i == j` в новый массив переписывается центральный элемент заданного интервала. Функция возвращает указатель на сформированный массив. В главной функции оператор

```
int* p = reverseArray ( ar, 5, 9 ) ;
```

предназначен для вызова реверса части массива `ar`, ограниченного индексами 5 и 9. В результате указатель `p` получит возвращенное функцией значение, а `main ( )` — доступ через этот указатель к элементам обратной части массива `ar`. Сам массив `ar` не меняется. Следует помнить, что операцию `delete` нельзя использовать для указателя `p` в главной функции, потому что динамического выделения памяти она не производила.

Прототип

```
char* reverseArray ( const char* ) ;
```

объявляет перегруженную функцию для реверса строки. Функция не изменяет исходную строку, указатель на которую является ее параметром. Длину строки передавать нет необходимости, так как она может быть вычислена. Функция возвращает указатель на динамически выделенный блок памяти, в котором расположен результат. В начале функции выделяется необходимое пространство оперативной памяти, куда копируется исходная строка. Реверс производится со строкой, расположенной по выделенному адресу `pOut`, для чего в цикле `while` многократно вызывается перегруженная функция перестановки элементов `swapR ( )`. В главной функции объявляется указатель `pr`, которому присваивается значение, возвращенное вызовом функции `reverseArray ( s )`. Состояние исходной строки `s` и результирующей `pr` выводится после вызова функции, причем строка `s` не изменяет своего значения после вызова. Результаты исполнения программы можно увидеть на рис. 13.12.

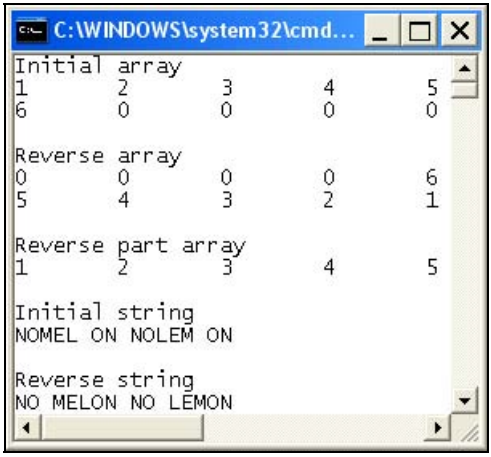


Рис. 13.12. Результаты исполнения программы с использованием перегруженных функций

Шаблонные функции

Некоторые перегруженные функции имеют один и тот же код, независимо от типа параметров, например, перестановка элементов или копирование содержимого одного массива в другой. В языке C++ существует возможность вместо нескольких идентичных по операциям функций создавать только одну универсальную функцию, которая называется шаблоном или шаблонной функцией.

Шаблонная функция (template function) — это функция, полностью контролирующая соответствие типов данных, которые задаются ей как параметры.

Объявление шаблона функции повторяет определение обычной функции, но первой строкой в объявлении обязательно должна быть строка следующего вида:

`template < class имя_типа`$_1$`, …, class имя_типа`$_N$ `>`

В угловых скобках `< >` после ключевого слова `template` указывается список формальных параметров шаблона. `Имя_типа` называют параметром типа, который представляет собой идентификатор типа и используется для определения типа параметра в заголовке шаблонной функции, типа возвращаемого функцией значения и типа переменных, объявляемых в теле функции. Каждый параметр типа должен иметь уникальный идентификатор, который может быть использован в разных шаблонах. Каждому параметру типа должно предшествовать ключевое слово `class`.

За строкой `template` следует строка со стандартным объявлением функции, где в списке параметров используются как параметры типа, так и любые другие допустимые базовые или производные типы данных.

Шаблонная функция может быть перегружена другим шаблоном или не шаблонной функцией с таким же именем, но с другим набором параметров (листинг 13.18).

Компилятор выбирает вариант функции, соответствующий вызову. Сначала подбирается функция, которая полностью соответствует по имени и типам параметров вызываемой функции. Если попытка заканчивается неудачей, подбирается шаблон, с помощью которого можно сгенерировать функцию с точным соответствием типов всех параметров и имени. Если же и эта попытка неудачна, выполняется подбор перегруженной функции.

Листинг 13.18. Пример шаблона для вывода одномерного массива любого типа

```
#include <iostream>
using namespace std ;
// шаблонная функция
template < class T > void view ( const T *p, const size_t size )
{
        for ( int i = 0; i < size; i++ )
                cout << p [ i ] << '\t' ;
        cout << endl ;
}
// главная функция
int main ( )
{
        int ia [ ]= { 1, -3, 0, -9, 2, 7, 0, -19 } ;
        double da [ ] = { 0.0875, 1.25, -3, 0.0, -7.986 } ;
        long la [ ] = { 1L, -2L, 3L, -100L } ;
        float fa [ ] = { -2.12, 1.5, -3.25, -17.2 } ;
        char s [ ] = "Vivat!" ;
        cout << "\nint Array\n" ;
        view ( ia, sizeof ( ia ) / sizeof ( int ) ) ;
        cout << "\ndouble Array\n" ;
        view ( da, sizeof ( da ) / sizeof ( double ) ) ;
        cout << "\nlong Array\n" ;
        view ( la, sizeof ( la ) / sizeof ( long ) ) ;
```

```
        cout << "\nchar Array\n" ;
        view ( s, strlen ( s ) ) ;
        return 0 ;
}
```

На рис. 13.13 продемонстрированы результаты работы шаблона для вывода массива. В качестве параметра типа шаблонная функция использует идентификатор `T`, который применяется для указателя `p` в списке параметров шаблонной функции. Указатель определяет адрес массива, элементы которого необходимо вывести. Второй параметр имеет явный тип `size_t`. Функция не возвращает значения и имеет тип `void`. Для каждого массива генерируется функция с соответствующим типом. Вместо параметра типа `T` подставляются значения `int` для массива `ia`, `double` для массива `da`, `long` для массива `la` и `char` для массива `s`. В каждом вызове функции `view ( )` указывается имя массива, соответствующее его адресу, а также размер массива, который вычисляется при помощи оператора `sizeof ( )`.

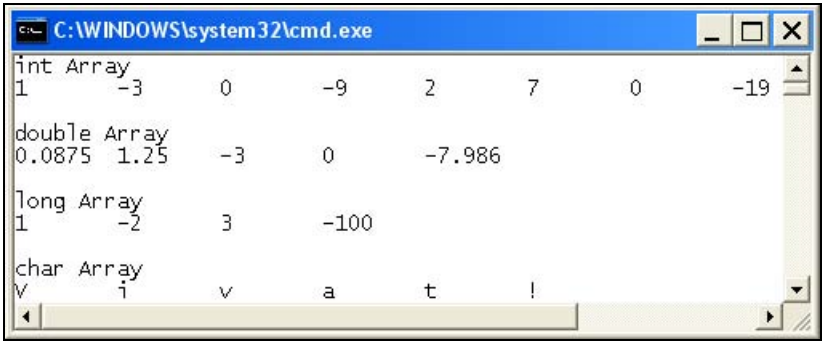


Рис. 13.13. Результаты работы шаблона для вывода массива

Листинг 13.19. Пример шаблона для определения абсолютного значения

```
#include <iostream>
using namespace std ;
// шаблонная функция
template < class T > T abs ( const T n ) { return n < 0 ? -n : n ; }
// главная функция
int main ( )
{
        int i ( -5 ) ; float f ( -1.125e-5 ) ; long l ( 123L ) ;
```

```
        cout << abs ( i ) << endl ;          // выводится 5
        cout << i << endl ;                  // выводится -5
        cout << abs ( f ) << endl ;          // выводится 1.125e-005
        cout << abs ( l ) << endl ;          // выводится 123
        cout << abs ( -123L ) << endl ;      // выводится 123
        return 0 ;
}
```

Шаблонная функция `abs ( )` в листинге 13.19 меняет знак только у отрицательных чисел любого типа. Возвращает положительное значение. Параметр типа `T` определяет как тип входного параметра `n`, так и тип возвращаемого функцией значения. Входной параметр указан со спецификатором `const`, поэтому в теле функции значение `n` не меняется. Это значит, что фактический параметр сохранит свое значение после выполнения функции. Например, если потребуется изменить значение переменной `i`, в главной функции следует воспользоваться оператором присваивания `i = abs ( i )`.

Листинг 13.20. Пример шаблона для перестановки элементов

```
template < class T > void swap ( T *a, T *b )
{       T tmp ( *a ) ; *a = *b ; *b = tmp ;        }
#include <iostream>
using namespace std ;
int main ( )
{
        int unsigned short x ( 150 ), y ( 15 ) ;
        double a ( 0 ), b ( -25.4 ) ;
        char t ( 'a' ), e ( 'z' ) ;
        cout << x << '\t' << y << endl ;    // выводится 150      15
        swap ( x, y ) ;
        cout << x << '\t' << y << endl ;    // выводится 15       150
        cout << a << '\t' << b << endl ;    // выводится 0        -25.4
        swap ( a, b ) ;
        cout << a << '\t' << b << endl ;    // выводится -25.4    0
        cout << t << '\t' << e << endl ;    // выводится a        z
        swap ( t, e ) ;
        cout << t << '\t' << e << endl ;    // выводится z        a
        return 0 ;
}
```

В листинге 13.20 шаблонная функция `swap ( )` типа `void` имеет параметр типа `T`, который обозначает тип указателей на переменные, значения которых необходимо поменять местами. В теле функции параметр типа используется в объявлении переменной `tmp` для временного хранения значения. Функция `swap ( )` переставляет элементы любого типа данных.

Листинг 13.21. Пример шаблона для удаления вхождений указанного элемента из массива

```
template < class T, class S >       T* del ( T x, T *in, S &size )
{
       T *out = in ; T *p = in ; T *end = in + size ;
       while ( in < end )
       {
              if ( *in != x ) *p++ = *in ;
              else --size ;
              in++ ;
       }
       return out ;
}
#include <iostream>
using namespace std ;
int main ( )
{
       int ia [ ]= { 1, -3, 0, -9, 0, -19, 0, 7 } ;
       double da [ ] = { 0.0875, 1.25, 0.0, 0.0, 0.0, -7.986 } ;
       char s [ ] = "aaaaHaaaaaaiaaa!aaa" ;
       size_t n ;
       cout << "\nDeleting 0 (int Array)\n" ;
       n = sizeof ( ia ) / sizeof ( int ) ;
       view ( ia, n ) ; del ( 0, ia, n ) ; view ( ia, n ) ;
       cout << "\nDeleting 0.0 (double Array)\n" ;
       n = sizeof ( da ) / sizeof ( double ) ;
       view ( da, n ) ; del ( 0.0, da, n ) ; view ( da, n ) ;
       cout << "\nDeleting 'a' (char Array)\n" ;
       n = strlen ( s ) ;
       cout << s << endl ; del ( 'a', s, n ) ;
       s [ n ] = '\0' ;      // необходимый для строки оператор
       cout << s << endl ;
       return 0 ;
}
```

На рис. 13.14 проиллюстрированы результаты использования шаблонной функции удаления всех вхождений указанного элемента из одномерного массива любого типа (листинг 13.21).

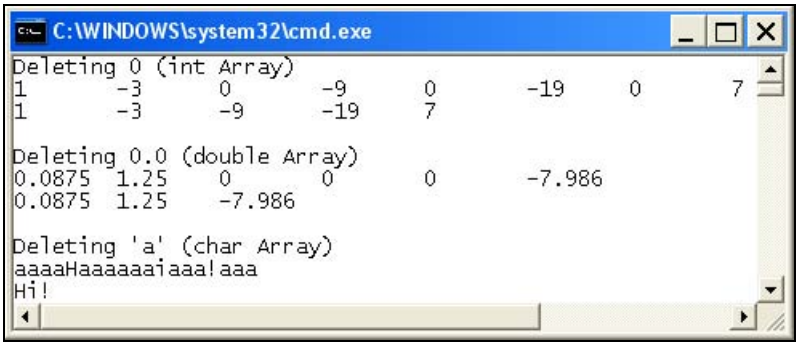


Рис. 13.14. Результаты работы шаблона для удаления всех вхождений элемента из массива

Шаблонная функция `del ( )` использует два параметра типа. Параметр типа `T` предназначен для обозначения типов указателя на возвращаемое значение функции, указателя на входной массив `in` и типа удаляемого элемента `x`. Второй параметр типа `S` используется для обозначения типа ссылки `size` на размер массива. Наличие этого параметра позволяет при вызове шаблона указывать длину массива как с типом `int`, так и с типом `size_t`. Параметр типа `T` применяется в теле функции при объявлении указателей, необходимых для обработки массива. Формальный параметр `size` не случайно является ссылкой, поскольку функция меняет размер массива в процессе обработки, и на выходе, скорее всего, будет записано другое значение.

Следует заметить, что занимаемое массивом пространство оперативной памяти после удаления вхождений элемента не изменяется. Недостатком шаблонной функции `del ( )` является некорректное завершение обработки символьного массива, если он представляет собой строку. Известно, что последним элементом строки должен быть нулевой байт, но в шаблоне отсутствует модификация завершения строки. Для исправления возможной ошибки при работе со строкой `s` в главной функции `main ( )` нулевой байт записывается на нужное место после вызова шаблонной функции. Как правило, для работы со строками шаблон перегружается при помощи явной функции, которая и вызывается в случае вызова с параметром строкового типа.

Текст шаблона `view ( )` представлен в первом примере шаблонных функций.

Листинг 13.22. Пример шаблона для реорганизации числового массива

```
// в начало массива записываются отрицательные элементы
// после отрицательных элементов записываются положительные элементы
template < class T > void reorg ( T *p, const int size )
{
       T tmp ;
       for ( int i = 0, j = 0 ; i < size ; i++ )
       {
              if ( p [ i ] > 0 )    continue ;
              tmp = p [ i ] ;
              for ( int k = i ; k > j ; k-- )     p [ k ] = p [ k - 1 ] ;
              p [ j ] = tmp ;
              j++ ;
       }
}
template < class T > void view ( const T* p, const int size ) ;
#include <iostream>
using namespace std ;
int main ( )
{
       long la [ ] = { 1L, -2L, 3L, -100L, -200L } ;
       float fa [ ] = { -2.12, 1.5, -3.25, -17.2 } ;
       int n = sizeof ( la ) / sizeof ( long ) ;
       cout << "\nInitial long array\n" ;  view ( la, n ) ;
       cout << "Reorganization\n" ; reorg ( la, n ) ; view ( la, n ) ;
       n = sizeof ( fa ) / sizeof ( float ) ;
       cout << "\nInitial float array\n" ; view ( fa, n ) ;
       cout << "Reorganization\n" ; reorg ( fa, n ) ; view ( fa, n ) ;
       return 0 ;
}
```

Шаблон `reorg ( )` (листинг 13.22) позволяет изменять порядок следования элементов в массиве любого числового типа так, что в начале располагаются отрицательные элементы, после которых записываются положительные. Важно помнить, что указатель `p` на исходный массив нельзя объявлять со спецификатором `const`, так как изменения осуществляются непосредственно в исходном массиве. Функция использует единственный параметр типа `T`, применяемый в заголовке и в теле шаблона. Главная функция проверяет

работу шаблона для массива длинного целого типа `long` и массива чисел с плавающей точкой `float`. Результаты обработки массивов при помощи функции `reorg ( )` показаны на рис. 13.15.

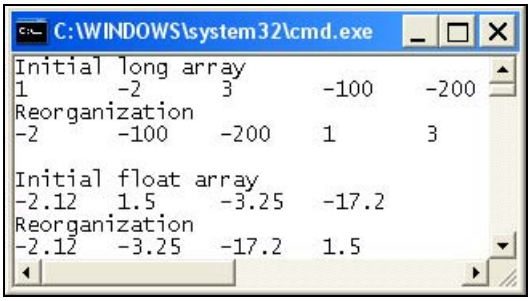


Рис. 13.15. Результаты работы шаблона для реорганизации массива

Листинг 13.23. Пример перегруженной шаблонной функции

```
// шаблон для сравнения на равенство двух значений
template < class T > static bool eq ( const T &x, const T &y )
{       return x == y ;       }
// шаблон для сравнения на равенство двух массивов
template < class T > static bool eq ( const T *px, const T *py, const
int size )
{
       for ( int i = 0; i < size; i++ )
              if ( ! eq ( px [ i ], py [ i ] ) ) return false ;
       return true ;
}
// не шаблонная функция для сравнения строк на равенство
static bool eq ( const char *s1, const char *s2 ) { return ! strcmp ( s1,
s2 ) ; }
#include <iostream>
using namespace std ;
int main ( )
{
       bool b1 [ ] = { true, false, false, true } ; bool b2 [ ] = {
true, false, false, true } ;
       char s [ ] = "C++" ; char c1 ( 'M' ), c2 ( 'm' ) ;
        cout << eq ( c1, c2 ) << endl ;    // выводится 0
```

```
        cout << eq ( b1, b2, 4 ) << endl ;  // выводится 1
        cout << eq ( "C++", s ) << endl ;   // выводится 1
        return 0 ;
}
```

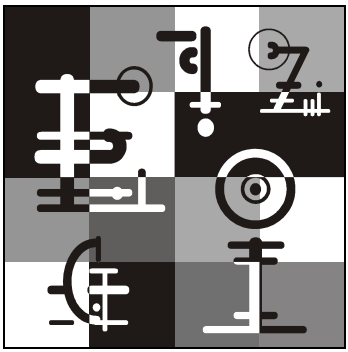
В листинге 13.23 шаблон `eq ( )` перегружен двумя способами: другим шаблоном и явной функцией.

Первая шаблонная функция возвращает `true`, если два значения одного и того же типа, заданного параметром типа `T` и формальными параметрами ссылок на тип `T`, равны. Иначе результат отношения `x == y` будет ложным, и функция вернет `false`.

Второй шаблон проверяет все элементы двух массивов на равенство и возвращает `true` только при совпадении значений всех элементов. Первая пара элементов, где разные значения, приводит к выходу из функции со значением `false`. В теле шаблона для сравнения элементов вызывается первая шаблонная функция. Формальные параметры представлены указателями на параметр типа `T`, которые при вызове должны содержать адреса сравниваемых массивов, и переменной целого типа, которая задает их размер.

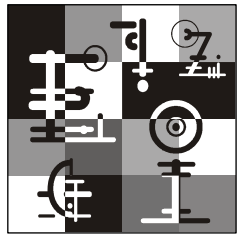
Для сравнения строк существует стандартная функция `strcmp ( )`, поэтому шаблон перегружен явной функцией, которая будет вызываться при обращении с типом параметров `char*`. Библиотечная функция `strcmp ( )` возвращает 0 при равенстве строк, поэтому в операторе `return` использовано логическое отрицание результата библиотечной функции.

В главной функции перегруженные функции тестируются на примере двух символьных переменных `c1` и `c2`, массивов логического типа `b1` и `b2`, а также двух строк, одна из которых задана литералом `"C++"`, а другая инициализирована через указатель `s`.



Часть III

Классы



Глава 14

Объекты и классы

C++ принадлежит к объектно-ориентированным языкам программирования, в которых главными элементами программ являются объекты и классы.

В окружающей людей реальности объект является ее частью, он существует во времени и пространстве. Объект можно увидеть, потрогать, либо знать, что он есть, например, файл на магнитном диске или погрузка конкретного вагона на железнодорожной станции. На рис. 14.1 показаны четыре объекта из библиотеки — книги по программированию.

Рис. 14.1. Объекты из библиотеки

У каждой книги свои автор, название, год издания, библиотечный шифр и стоимость, которые являются свойствами книги. Каждая книга в библиотеке может обладать некоторым состоянием: книга свободна (значит, она может быть выдана читателю), книга выдана (значит, она находится у читателя). Книги отличаются друг от друга конкретными значениями свойств, набор которых является уникальным для каждой отдельной книги, а также конкретным состоянием, текущее значение которого может изменяться во времени. Над книгами в библиотеке можно производить разные действия: искать нужные книги, создавать в библиотечном фонде новые, выдавать читателям, возвращать в библиотеку. Конкретное действие может повлечь за собой

изменение состояния. Например, действие "выдать книгу" изменяет ее состояние из "свободна" в состояние "выдана".

Набор свойств книги (автор, название и т. д.) не что иное, как данные. Действия (искать, создать и т. д.) — функции, которые выполняются над книгами. Ключевая идея объектно-ориентированного программирования состоит в том, что данные и производимые над этими данными действия объединяются и представляют единое неделимое целое, которое и называют объектом. Объектно-ориентированная программа представляет собой множество взаимодействующих друг с другом объектов. Взаимодействие объектов осуществляется через вызов некоторой функции этого объекта. В терминах объектно-ориентированного программирования вызов функции называют передачей сообщения. Например, сообщениями для книги могут быть следующие: искать, выдать. Динамика изменения состояний объекта и его реакция на поступающие сообщения отражает поведение объекта.

Общее описание для всех объектов с общими свойствами и поведением называется классом объектов. Класс вводит понятие абстракции, которая позволяет установить существенные характеристики объекта, которые отличают его от всех других объектов и четко определяют особенности этого объекта с точки зрения дальнейшего рассмотрения. Класс нельзя ни увидеть, ни потрогать, но можно знать, что он есть (например, в программном файле на диске, в библиотеке классов). Класс показывает, как построить существующую во времени и пространстве переменную этого класса — объект.

Класс (class) — определяемый программистом тип данных, который описывает множество объектов с общими свойствами, поведением и семантикой. Каждый класс имеет уникальное имя, определяющее название нового типа данных. Например, для множества книг можно определить класс `CBook`.

Объект (object) — переменная типа класс или экземпляр класса. По правилам объявления переменных для класса `CBook` можно, например, объявить объект этого класса как `CBook book`.

В связи с тем, что тип класса объединяет данные и операции обработки этих данных в единое целое, для классов вводится понятие членов класса (class members). К членам класса относятся данные (data members) и функции (members function).

Член класса (данное) — именованная характеристика или свойство объектов класса, которое имеет собственное значение для каждого объекта. Например, в классе `CBook` можно объявить данные `author`, `title`, `year`, которые будут определять автора, название и год издания для объектов-книг класса. Для каждой книги эти данные будут иметь конкретные значения.

Член класса (функция) — это функция, принадлежащая классу, при помощи которой объект класса взаимодействует с объектами других классов или того же самого класса. Функции-члены принято называть методами. Метод представляет собой действие, которое должен выполнить объект для реализации своего поведения, или сервис, который может быть востребован одним объектом у другого. У каждого метода есть свой объект-получатель, то есть объект, для которого метод применяется. Если он использует только данные своего класса, то у него не будет параметров. Если же для выполнения метода требуются дополнительные аргументы, то он является параметризованным. Некоторые методы могут приобретать разные формы в различных классах, такие методы называются полиморфными. Определение методов находится в классе. Исполняют методы объекты классов, когда эти методы ими вызываются.

Примечание

Многие авторы книг по программированию на языке C++ называют объектами переменные базовых типов данных (`int`, `double` и т. д.), а также переменные производных от базовых типов (массивы, указатели, структуры и т. д.).

Спецификаторы доступа к членам класса

Спецификаторы доступа определяют доступность данных и методов в программе. Существуют следующие спецификаторы доступа:

- `public` — открытый режим доступа. Члены класса доступны за пределами класса любым функциям программы;
- `protected` — защищенный режим доступа. Члены класса доступны для методов этого класса, производных классов и дружественных классов. Производные классы и классы–друзья рассматриваются в следующих главах книги;
- `private` — закрытый режим доступа. Члены класса доступны только для методов этого класса и классов-друзей.

Особенность объектно-ориентированного программирования заключается в сокрытии данных, что защищает данные класса от несанкционированного доступа функций, расположенных за пределами класса. В связи с этим члены-данные, как правило, имеют спецификатор доступа `private`. В отдельных случаях возникает необходимость данные класса делать открытыми.

Методы, которые предоставляют сервисные услуги класса, всегда объявляются со спецификатором `public`. Методы, используемые классом для собст-

венных нужд, имеют спецификатор доступа `private`. Эти методы обеспечивают реализацию открытых функций-членов и скрывают сложности их реализации.

Объявление или спецификация класса

Объявление класса (class declaration) представляет собой описание членов класса: данных и методов. Часто объявление класса называют спецификацией. Члены-данные объявляются согласно правилам объявления переменных и могут иметь любой тип, включая тип класса и указателя на тип класса. Функции-члены в спецификации представлены прототипами.

Спецификация класса, как правило, размещается в отдельном заголовочном файле с расширением `.h` и имеет следующий формат:

```
class имя_класса
{
private:
        // закрытые члены класса
protected:
        // защищенные члены класса
public:
        // открытые члены класса
} ;
```

Объявление класса начинается с ключевого слова `class`, за которым следует имя класса, и заканчивается точкой с запятой. Внутри фигурных скобок может находиться любое количество секций со спецификаторами доступа, причем спецификаторы могут повторяться (листинг 14.1).

Примечание

Если после открывающей фигурной скобки не стоит никакой спецификатор доступа, члены класса по умолчанию становятся private (закрытыми).

Листинг 14.1. Пример объявления класса

```
// Book.h - спецификация класса CBook
#pragma once
class CBook
{
private:
       char m_author [ 50 ] ;       // автор
```

```
        char *m_pTitle ;            // указатель на название
        int m_year ;                // год издания
public:
        // методы установки значений
        void setAuthor ( const char* ) ;
        void setTitle ( const char* ) ;
        void setYear ( const int ) ;
        // методы возврата значений
        char* getAuthor ( void ) ;
        char* getTitle ( void ) ;
        int getYear ( void ) ;
} ;
```

Файл спецификации `Book.h` содержит объявление типа класса `CBook`. Чтобы компилятор включал объявление типа только один раз при трансляции программы, используется директива препроцессора `#pragma once`.

В классе объявлены закрытые данные с типом массива `m_author`, указателя на символьный тип `m_pTitle` и целого `m_year`, которые описывают автора, название и год издания, соответственно. Тип `CBook` сообщает компилятору, что для создания объекта типа `CBook` потребуется 50 байтов для хранения автора, 4 байта для адреса, по которому будет записано название книги, и 4 байта для хранения года издания.

Чтобы можно было изменять значения данных у объектов класса и получать эти значения в программе, в классе объявлены открытые методы. Функции-члены с типом возвращаемого значения `void` имеют параметр, тип которого соответствует типу данного. Методы для возврата значений не нуждаются в параметрах, поскольку переменные, значения которых они возвращают, известны классу. Тип каждого метода обусловлен типом возвращаемой переменной.

Реализация класса

Определение объявленных в спецификации класса методов располагается в отдельном файле с расширением `.cpp`, называемом файлом реализации класса.

Когда файлы спецификации и реализации класса находятся в разных файлах, заголовок функции-члена должен включать область видимости согласно формату:

```
тип_функции-члена имя_класса : : имя_функции-члена ( список параметров )
```

Оператор разрешения области видимости : : используется в заголовке с той целью, чтобы все объявленные внутри класса идентификаторы обрабатывались внутри их собственного пространства имен, а также во избежание неоднозначности использования имен. Запись тела метода ничем не отличается от записи в обычной функции. В файл реализации класса следует включить файл со спецификацией класса, а также необходимые при определении функций заголовки стандартных библиотек (листинг 14.2).

Листинг 14.2. Пример реализации класса

```
// Book.cpp - реализация класса CBook
#include "Book.h"
#include <string>
// установить автора
void CBook :: setAuthor ( const char* author )
{
       strncpy_s ( m_author, 50, author, 49 ) ;
       if ( strlen ( author ) > 49 ) m_author [ 49 ] = '\0' ;
}
// установить название
void CBook :: setTitle ( const char* title )
{
       delete [ ] m_pTitle ;
       m_pTitle = new char [strlen ( title )+ 1] ;
       strcpy_s ( m_pTitle, strlen ( title ) + 1, title ) ;
}
// установить год издания
void CBook :: setYear ( const int year )
{      m_year = year ;       }
// вернуть автора
char* CBook :: getAuthor ( void )
{      return m_author ;     }
// вернуть название
char* CBook :: getTitle ( void )
{      return m_pTitle ;     }
// вернуть год издания
int CBook :: getYear ( void )
{      return m_year ;       }
```

Методы, показанные в листинге 14.2, не нуждаются в особенных комментариях. Следует обратить внимание на определение методов для установки значения автора и значения названия книги.

В методе `setAuthor ( )` в переменную `m_author` можно скопировать только 49 байтов из параметра `author`. В случае превышения этого значения остаток строки `author` теряется, и в последний байт массива `m_author` записывается нулевой байт.

В методе `setTitle ( )` данное `m_pTitle` является указателем, поэтому сначала необходимо освободить память, занимаемую существующим значением названия книги, выделить достаточный для размещения нового названия книги блок памяти и только потом копировать значение параметра `title`. При выделении памяти к длине строки `title` добавляется 1 байт для записи нулевого байта.

Рекомендации по выбору имен

Для облегчения чтения исходного кода рекомендуется придерживаться следующих соглашений по выбору идентификаторов:

- Имя класса начинать с заглавной буквы `C`. Будет понятно, что этот идентификатор имеет отношение к типу данных класс. Например, `CLocomotive`, `CArray`, `CStudent`.
- Файлы спецификации и реализации класса называть именем класса, но без первой буквы `C`.
- Имя члена-данного начинать с префикса `m_`, например, `m_name`, `m_state`.
- Метод называть адекватно его назначению, например, `find ( )`, `isState ( )`.
- Формальный параметр, имеющий отношение к члену-данному класса, называть точно так же, как и данное, но без префикса `m_`. Например, `name`, `state`. Это поможет в написании кода программы.
- Идентификатор переменной подбирать так, чтобы был ясен физический смысл. Например, если переменная обозначает город, то не стоит называть ее `gor` или `x1`, лучше дать ей имя `town`.

Объявление объекта класса

Объявление объекта класса (object definition) не стоит путать с объявлением класса. Важно понимать, что класс не содержит никаких значений данных, а лишь описывает общую структуру и поведение объектов, являясь как бы трафаретом для их создания.

Описание объекта задает его тип (имя класса) и, возможно, необходимые для инициализации членов-данных значения. При объявлении объекта компилятор получает указание на создание переменной класса на основании заданного типа.

Когда объект объявляется, то согласно описанию класса для объекта происходит выделение оперативной памяти, а также при указании значений данных осуществляется инициализация членов-данных указанными значениями. Всю эту работу делает специальный метод класса, называемый конструктором. Конструкторы рассматриваются в книге в отдельном разделе с одноименным названием.

Основными форматами объявления объекта класса являются следующие:

```
имя_класса имя_объекта ;
имя_класса имя_объекта ( список параметров ) ;
имя_класса имя_объекта ( имя_объекта_копирования ) ;
```

Листинг 14.3. Пример объявления объектов класса

```
CBook book, aBook [ 100 ] ;
CBook obj ( "Carrol L.", "Alice in Wonderland", 2002 ) ;
CBook copy ( obj ) ;
```

В листинге 14.3 использованы все три формата для объявления объектов типа класса `CBook`. Согласно первому, объявляется объект `book` и массив из 100 объектов `aBook`. По второму формату объявлен объект `obj`, и по третьему — объект `copy`.

Доступ к членам объектов

Любой объект, как переменная своего класса, включает члены-данные и знает о методах обработки этих данных. Доступ к членам объекта обеспечивается операторами точка и стрелка, как и доступ к элементам структур. Доступ к данным и методам определяется значением спецификатора доступа `private`, `protected` или `public`. Только спецификатор `public` позволяет обращаться к элементам из любого места в программе. Два других спецификатора разрешают доступ лишь объектам того же самого класса или его друзьям.

Для получения доступа по имени объекта необходимо использовать оператор точка. Например, `book.m_author`, `obj.m_pTitle`, `copy.m_author`, `obj.getTitle ( )`.

Для доступа через указатель используется оператор стрелка. Например, если известен указатель `p` на тип `CBook`, то записывается `p -> m_author`, `p -> getTitle ( )`.

Конструкторы класса

При создании объекта автоматически вызывается специальный метод, который называется конструктором (constructor). У конструктора имя класса и обязательно открытый спецификатор доступа. Конструктор управляет построением объекта в оперативной памяти. Процесс построения включает в себя выделение памяти для членов-данных типа указателей, а также инициализацию данных.

Конструктор никогда не возвращает никакого значения и не имеет типа, даже `void`. Конструктор перегружается, так как объекты можно инициализировать по-разному. Количество конструкторов практически не ограничивается и зависит от целесообразности их наличия в классе. Однако существуют конструкторы, без которых не обойтись ни одному классу. Это конструктор по умолчанию, конструктор с параметрами и конструктор копирования.

Важно помнить, никакой конструктор не должен делать ничего, что касается обработки членов-данных. За обработку членов-данных несут ответственность не конструкторы, а другие методы в классе.

Инициализаторы в конструкторах

Главной задачей конструктора класса является инициализация данных создаваемых им объектов. Однако производить инициализацию данных при помощи оператора присваисания в теле конструктора не рекомендуется.

Инициализация осуществляется при помощи списка инициализации, расположенного между двоеточием, которое следует за закрывающей круглой скобкой в заголовке конструктора, и телом конструктора. Присваиваемые при инициализации данных значения записываются в круглых скобках после идентификаторов членов-данных и отделяются друг от друга запятыми.

Порядок следования идентификаторов в списке инициализации не имеет значения.

Конструктор по умолчанию

Конструктор по умолчанию — конструктор, не требующий параметров. Этот конструктор всегда должен быть определен для любого класса. Конструктор по умолчанию может не выполнять никаких действий, но чаще всего он инициализирует данные класса нулевыми значениями.

Объявление конструктора по умолчанию имеет следующий формат:

```
public:имя_класса ( ) ;
```

Например, для класса CBook прототип конструктора по умолчанию записывается со спецификатором открытого доступа

```
public: CBook ( ) ;
```

Листинг 14.4. Пример реализации конструктора по умолчанию

```
CBook :: CBook ( ) : m_year ( 0 ), m_pTitle ( "" )
{       m_author [ 0 ] = '\0' ;       }
```

В результате работы этого конструктора (листинг 14.4) будет построен объект-книга, у которого данное year получит нулевое значение, массив author начнется с нулевого байта точно так же, как и название книги, на которое указывает pTitle. Для инициализации автора не может быть использован инициализатор, так как этот член-данное объявлен в классе как символьный массив.

Конструктор с параметрами

Конструктор с параметрами инициализирует значения данных объекта значениями полученных параметров. Параметров будет столько, сколько данных необходимо проинициализировать. Прототип такого конструктора имеет формат:

```
public:имя_класса ( список формальных параметров ) ;
```

Для класса CBook конструктор с параметрами может иметь следующий прототип: `CBook ( char*, char*, int ) ;`

Листинг 14.5. Пример реализации конструктора с параметрами

```
CBook :: CBook ( char *author, char *title, int year )
: m_year ( year ), m_pTitle ( title )
{
        strncpy_s ( m_author, 50, author, 49 ) ;
        if ( strlen ( author ) > 49 ) m_author [ 49 ] = '\0' ;
}
```

В результате работы этого конструктора (листинг 14.5) будет построен объект-книга, у которого данное m_year получит значение параметра year, в массив m_author будет скопировано не более 49 байтов из строки author, и указатель m_pTitle будет содержать адрес, по которому скопировано значение title.

Конструктор копирования

Конструктор копирования создает копию объекта в оперативной памяти с помощью другого объекта того же класса. В качестве параметра этот конструктор получает ссылку на объект, копию которого необходимо создать. Для конструктора копирования необходимо указать прототип следующего формата:

```
public:имя_класса ( имя_класса & ) ;
```

Для класса `CBook` конструктор копирования объявляется в спецификации класса со следующим прототипом:

```
CBook ( CBook & ) ;
```

Листинг 14.6. Пример конструктора копирования

```
CBook :: CBook ( CBook &o ) : m_year ( o.m_year )
{
       strcpy_s ( m_author, strlen ( o.m_author ) + 1, o.m_author ) ;
       m_pTitle = new char [strlen ( o.m_pTitle ) + 1 ] ;
       strcpy_s ( m_pTitle, strlen ( o.m_pTitle ) + 1, o.m_pTitle ) ;
}
```

В результате работы конструктора копирования (листинг 14.6) будет построен новый объект-книга, все члены-данные которого получат значения данных, принадлежащих копируемому объекту `o`, переданному по ссылке. Чтобы присвоить значение названия книги вновь создаваемому объекту, нужно сначала выделить необходимый блок памяти при помощи оператора `new`.

Деструктор

Деструктор `(destructor)` — специальный метод класса, используемый для разрушения объектов класса. Имя деструктора совпадает с именем конструктора (именем класса), которому предшествует символ тильда `~`. Он всегда имеет открытый спецификатор доступа и не имеет ни типа, ни параметров. Деструктор управляет уничтожением объекта из оперативной памяти.

Вызывается деструктор автоматически при разрушении объекта. Если объект создавался динамически через указатель при помощи оператора `new`, то для уничтожения такого объекта следует использовать оператор `delete` для указателя.

Типовым использованием деструктора является освобождение динамической памяти, которая ранее была выделена конструктором.

В отличие от конструктора, деструктор не перегружается и может быть в классе только один. Лучше всегда определять деструктор класса, даже если он не производит никаких действий и имеет пустое тело.

Объявление деструктора имеет следующий формат:

```
public:~имя_класса ( ) ;
```

Например, для класса `CBook` прототип деструктора в спецификации класса имеет вид:

```
~CBook ( ) ;
```

Листинг 14.7. Пример реализации деструктора

```
CBook :: ~CBook ( ) {        delete [ ] m_pTitle ;        }
```

Результатом действия деструктора `~CBook ( )` является освобождение блока памяти с начальным адресом из `m_pTitle`, выделенного ранее конструктором класса (листинг 14.7).

Вызов конструктора и деструктора

Как упоминалось ранее, конструктор и деструктор вызываются в программе автоматически при объявлении объектов класса. Не имеет значения, объявляется ли объект явно или создается динамически с помощью `new`.

Вызов того или иного конструктора зависит от формы описания объекта:

- Если после имени объекта или типа класса в операторе `new` ничего не указано или стоят пустые круглые скобки, вызовется конструктор по умолчанию.
- Если в скобках записаны параметры, вызывается конструктор с параметрами.
- Если в скобках находится имя объекта, то происходит вызов конструктора копирования.

Кроме того, конструктор вызывается внутри функций для создания копии фактического параметра, если функция получает этот параметр по значению. Известно, что при выходе из функции копия уничтожается, следовательно, автоматически будет вызван деструктор для разрушения копии.

Конструкторы вызываются в программе в порядке объявления объектов. Когда объявляется массив объектов, конструктор по умолчанию будет вызван столько раз, сколько элементов указано в объявлении массива.

При объявлении массива указателей на тип класса или единичного указателя никакие конструкторы не вызываются. Они будут вызваны позже при инициализации указателей оператором `new`.

Деструктор вызывается автоматически при разрушении объявленного по имени объекта, поэтому нет необходимости в явном вызове деструктора для разрушения локального объекта или для разрушения объектов перед выходом из программы. Деструктор также вызывается автоматически, когда с помощью оператора `delete` освобождается указатель на объект класса. Без оператора `delete` объект разрушен не будет. Разрешается и явный вызов деструктора для объекта, который выполняется, как и вызов любого другого метода класса.

Деструкторы вызываются в порядке, обратном порядку объявления именованных объектов в программе. Вызов деструкторов объектов, объявленных через указатели, определяется следованием операторов `delete` для этих указателей (листинг 14.8).

Листинг 14.8. Иллюстрация вызовов конструкторов и деструктора

```
// конструктор по умолчанию
CBook :: CBook ( ) : m_year ( 0 ), m_pTitle ( "" )
{
       m_author [ 0 ] = '\0' ;
       cout << "default CONSTRUCTOR\nthis = " << this ;
}
// конструктор с параметрами
CBook :: CBook ( char *author, char *title, int year )
: m_year ( year ), m_pTitle ( title )
{
       strncpy_s ( m_author, 50, author, 49 ) ;
       if ( strlen ( author ) > 49 ) m_author [ 49 ] = '\0' ;
       cout << "with parameters CONSTRUCTOR\nthis = " << this ;
}
// конструктор копирования
CBook :: CBook ( CBook &o ) : m_year ( o.m_year )
{
       strcpy_s ( m_author, strlen ( o.m_author ) + 1, o.m_author ) ;
       m_pTitle = new char [strlen ( o.m_pTitle ) + 1 ] ;
       strcpy_s ( m_pTitle, strlen ( o.m_pTitle ) + 1, o.m_pTitle ) ;
       cout << "CONSTRUCTOR of copying\nthis = " << this ;
```

```
}
// деструктор
CBook :: ~CBook ( )
{       delete [ ] m_pTitle ; cout << "DESTRUCTOR\t " << this << endl ; }
// главная функция
#include <iostream>
using namespace std ;
#include "Book.h"
void view ( char*, CBook& ) ;
int main ( )
{
        // объявление с вызовом конструктора по умолчанию
        CBook book ;
        book.setAuthor ( "Robert Lafore" ) ;
        book.setTitle ( "Object-Oriented Programming in C++" ) ;
        book.setYear ( 2004 ) ;
        view ( "book", book ) ;
        // объявление с вызовом конструктора с параметрами
        CBook obj ( "Carrol L.", "Alice in Wonderland", 2002 ) ;
        view ( "obj", obj ) ;
        // объявление с вызовом конструктора копирования
        CBook copy ( obj ) ;
        view ( "copy obj", copy ) ;
        // объявление через указатель с вызовом конструктора с параметрами
        CBook *p ;
        p = new CBook ( "Herbert Schildt", ↳
"C++ The Complete Reference", 2003 ) ;
        view ( "pointer p", *p ) ;
        // разрушение объекта, объявленного через указатель
        delete p ;
}
// функция вывода состояния объекта класса CBook
void view ( char *s, CBook &o )
{
        cout << "\nState of object \' " << s << " \'\n" ;
        cout << "Author:\t" << o.getAuthor ( ) << endl ;
        cout << "Title:\t" << o.getTitle ( ) << endl ;
        cout << "Year:\t" << o.getYear ( ) << endl << endl ;
}
```

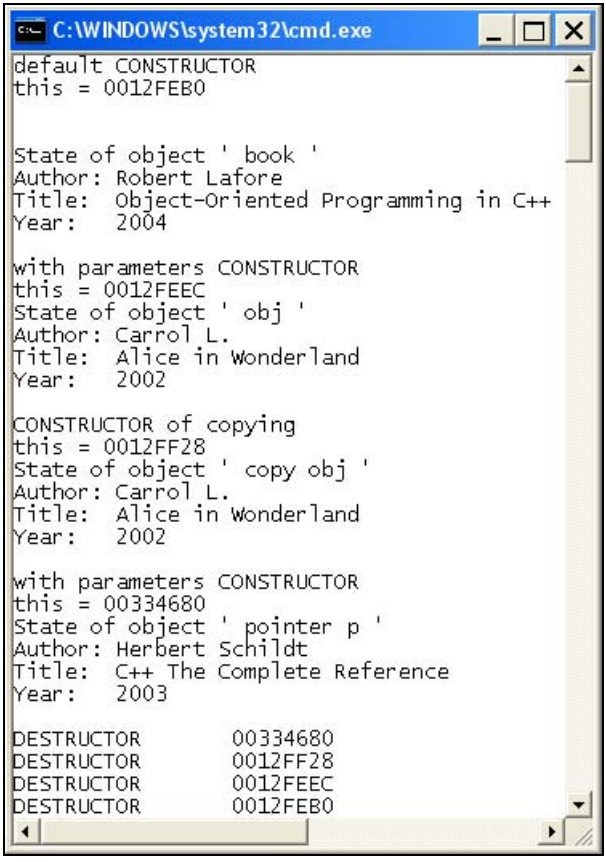


Рис. 14.2. Результаты работы конструкторов и деструктора класса `CBook`

На рис. 14.2 показаны результаты работы конструкторов и деструктора класса `CBook`. В связи с тем, что эти методы вызываются автоматически, в реализацию каждого из них добавлен вывод сообщения с названием метода, а также вывод адреса начала объекта в оперативной памяти. В примере определена дополнительная функция `view ( )`, при помощи которой выводится состояние объекта, чей адрес получает функция.

В главной функции первым вызывается конструктор по умолчанию, когда объявляется объект `book`. Далее для объекта `book` при помощи методов класса `setAuthor ( )`, `SetTitle ( )` и `setYear ( )` устанавливаются значения данных `m_author`, `m_pTitle` и `m_Year`, после чего выводится состояние объекта. Вторым вызывается конструктор с параметрами, создающий объект `obj` с инициализацией данных этого объекта. Третий вызываемый конструктор, который порождает являющийся копией `obj` объект `copy` — это конструктор

копирования. После вывода состояния `copy` при помощи оператора динамического выделения памяти и конструктора с параметрами создается объект, адрес которого возвращает оператор `new`.

Оператор `delete` в конце программы освобождает указатель `p` и вызывает деструктор класса, который разрушает последний созданный объект. Далее деструктор автоматически вызывается три раза, разрушая объекты `copy`, `obj` и `book`.

Указатель *this*

Указатель, обозначаемый ключевым словом `this`, представляет собой неявно определенный указатель на сам объект и является скрытой внутренней переменной класса. Отсюда следует:

- `this` — это адрес активного объекта в оперативной памяти.
- `*this` — сам активный объект.
- `this -> имя члена-данного класса` указывает на данное активного объекта. Как правило, `this ->` не пишется для активного объекта в методах класса, которые обрабатывают это данное активного объекта.
- `this -> имя метода ( список фактических параметров)` вызывает метод для активного объекта. Как правило, `this ->` не пишется при вызове метода активным объектом класса.
- Получить значение указателя в методах класса можно с помощью ключевого слова `this`.
- `this` — это локальная переменная, недоступная за пределами класса. Узнать значение указателя `this` какого-либо объекта класса в функциях программы можно только с помощью открытого метода класса, который возвращает это значение, или же просто воспользоваться оператором взятия адреса `&`.

Абсолютно каждый объект имеет свой собственный указатель `this`. Этот указатель не нужно объявлять в классе, так как объявление `имя_класса *const this` скрыто в классе как объявление указателя на константу. Для любого принадлежащего классу метода `this` неявно объявлен точно так же. Изменить `this` невозможно.

На рис. 14.3 показана схема расположения в оперативной памяти объектов `book` и `obj`, а также методов класса `CBook`. Оперативная память для объектов выделяется при их создании. Размер объекта в памяти определяется суммой размеров членов-данных класса. Если в классе член-данное объявляется при помощи указателя или ссылки, то блок оперативной памяти для размещения

значения этого члена-данного будет выделен динамически в процессе работы программы. Функции-члены помещаются в оперативную память в единственном экземпляре при объявлении класса и совместно используются всеми существующими объектами этого класса. Функции-члены не занимают область памяти, выделенную для объекта.

Рис. 14.3. Схема расположения в памяти объектов и методов класса

Статические данные класса

Иногда бывает полезно выделить общую память для всех объектов одного класса. Чтобы получить такую память, следует объявить член-данное со спецификатором `static`. Ключевое слово `static` помещается перед типом того данного, которое объявляется статическим. Чтобы статические данные класса стали доступны за пределами класса, необходимо назначить им открытый спецификатор доступа `public`. Если же данные объявить закрытыми, то потребуются открытые методы для получения их значений.

Перечислим основные свойства статических данных класса:

- Статические данные не являются частью объектов класса и хранятся за его пределами. Память для статической переменной-члена при объявлении объекта класса не выделяется.
- Одна копия используется всеми объектами класса, следовательно, все объекты имеют одно и то же значение статического данного.
- Объявление статических данных в классе не является их описанием, поэтому присваивание им значений производится вне класса и вне функций программы.

- Статические данные существуют даже при отсутствии объектов класса, для которого они объявлены как статические.
- Доступ к статической переменной-члену со спецификатором `public` может быть реализован без всякого объекта.

Листинг 14.9. Пример класса со статическими данными

```
class C
{
public:
       static double A ;     // открытый статический член-данное
       int m_x, m_y ;        // нестатические члены-данные
private:
       static int B ;        // закрытый статический член-данное
public:
       // конструкторы и деструктор
       C ( ) : m_x ( 0 ), m_y ( 0 ) {      }
       C ( int x, int y ) : m_x ( x ), m_y ( y ) {       }
       ~C ( ) {       }
       // методы для статического данного B
       void setB ( int _b ) { B = _b ; }
       int getB ( ) { return B ; }
} ;
// инициализация статических данных
double C :: A = 2.75 ;
int C :: B = 777 ;
// главная функция
#include <iostream>
using namespace std ;
int main ( )
{
       C a ( 10, 10 ), b ( -10, -10 ) ;
       // вывод состояния нестатических данных
       cout << "a\t" << a.m_x << '\t' << a.m_y << endl ;
       cout << "b\t" << b.m_x << '\t' << b.m_y << endl ;
       // вывод состояния статических данных
       cout << "for a and b static A =\t" << C :: A << endl ;
       cout << "for a static B =\t" << a.getB ( ) << endl ;
       cout << "for b static B =\t" << b.getB ( ) << endl ;
```

```
        // изменение и вывод значения A
        C :: A = 1.2e308 ;
        cout << "after change A =\t" << C :: A << endl ;
        // изменение значения B в объекте a
        a.setB ( 100 ) ;
        // вывод значения для B для объекта b
        cout << "after change in a b.B =\t" << b.getB ( ) << endl ;
        // изменение значения B в объекте b
        b.setB ( -100 ) ;
        // вывод значения B для объекта a
        cout << "after change in b a.B =\t" << a.getB ( ) << endl ;
}
```

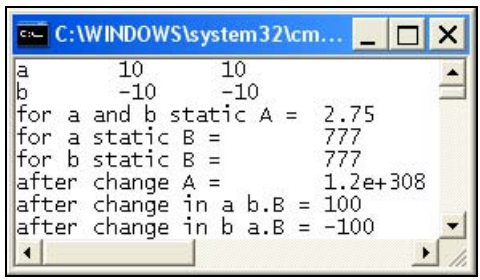


Рис. 14.4. Результаты использования статических данных в классе

В классе `C` в качестве статических данных объявлены открытая `A` и закрытая `B` статические переменные (листинг 14.9). Инициализация значениями происходит за пределами класса и функции `main ( )`, как того требуют правила C++. На рис. 14.4 видно, что изменения значения закрытого статического данного `B` в объекте `a` отражаются на объекте `b`, и наоборот. Для доступа к значениям `B` требуются методы из-за спецификатора `private`. Доступ к `A` происходит обычным способом, так как этот статический член-данное имеет спецификатор доступа `public`.

Статические методы класса

Статическими могут быть не только данные класса, но и методы. Ключевое слово `static` ставится перед типом функции-члена. Статические методы в основном используются для работы со статическими членами-данными класса.

Существуют следующие ограничения на статические методы:

- Методы имеют прямой доступ лишь к статическим данным класса.
- Указатель `this` не виден в статических методах, поэтому в их определении нельзя обратиться к нестатическим данным класса.
- Невозможно объявить в классе статическую и нестатическую версии функции.
- Статические методы не могут быть виртуальными (виртуальные методы рассматриваются в *главе 16*) и не могут быть константными.

Листинг 14.10. Пример класса со статическими методами

```
#include <iostream>
using namespace std ;
class C
{
public:
        // конструкторы и деструктор
        C ( char* s ) { strcpy_s ( m_s, 40, s ) ; incObj ( ) ; }
        C ( )  { m_s [ 0 ] = '\0' ; incObj ( ) ; }
        ~C ( ) { --objAmount ; }
        // открытые статические методы
        static void initAmount ( const int v ) { objAmount = v ; }
        static int getAmount ( ) { return objAmount ; }
private:
        // закрытые статические метод и член-данное
        static int incObj ( ) { return ++objAmount ; }
        static int objAmount ;
        // закрытые нестатический метод и член-данное
        int lenS ( ) { return strlen ( m_s ) ; }
        char m_s [ 40 ] ;
} ;
// объявление статической переменной
int C :: objAmount ;
// главная функция
int main ( )
{
        C :: initAmount ( 0 ) ;      // инициализация objAmount
```

```
        // объявление и создание объектов
        C o1 ( "St. Petersburg" ), o2 ( "Yalta" ) ;
        C o [ 12 ] , o3 ;
        C* p = new C [ 25 ] ;
        // вывод состояния статической переменной
        cout << C :: getAmount ( ) << endl ;       // выводится 40
        // разрушение объектов
        delete [ ] p ;
        // вывод состояния статической переменной
        cout << C :: getAmount ( ) << endl ;       // выводится 15
        return 0 ;
}
```

В листинге 14.10 для закрытой статической переменной `objAmount` класса `C` объявлены три статических метода, два из них являются открытыми, а один — закрытым. Член-данное `objAmount` используется для подсчета существующих в оперативной памяти объектов класса. Для инициализации `objAmount` целым значением реализован открытый статический метод `initAmount ( )`, для получения значения — `getAmount ( )`. Закрытый статический метод `incObj ( )` предназначен для увеличения на 1 значения статической переменной класса `objAmount`. Этот метод вызывается всеми конструкторами класса. Чтобы программа определяла правильно количество существующих объектов, в деструктор добавлен декремент `objAmount`.

До начала главной функции объявляется статическая переменная, которая инициализируется первым оператором `main ( )`, устанавливая для `objAmount` нулевое значение. При объявлении объектов `o1`, `o2`, `o3` класса `C`, массива `o` из 12 объектов и указателя `p` на массив из 25 объектов типа класса с одновременной инициализацией автоматически вызываются конструкторы класса, внутри которых инкрементируется число созданных объектов `objAmount`. Вызов статической функции `getAmount ( )` возвращает значение 40, которое выводится. Оператор `delete` заставит работать деструктор класса, и 25 объектов будут разрушены. Значение `objAmount` 25 раз уменьшится на 1, после чего повторный вызов `getAmount ( )` вернет значение 15.

В связи с тем, что областью видимости статических методов является класс, в котором они объявлены, при всех вызовах указывается оператор расширения области видимости:

```
C :: initAmount ( 0 ) ; C :: getAmount ( ) ;
```

Примечание

Если объявить статическую переменную класса `objAmount` открытой со спецификатором `public`, то можно получить значение без оператора области видимости, используя любой объект программы. Например, все показанные далее операторы выведут значение `objAmount` на экран:

```
cout << p -> objAmount << endl ;
cout << o1 . objAmount << endl ;
cout << o [ 2 ] . objAmount << endl ;
cout << o [ 24 ] . objAmount << endl ;
```

Константные методы класса

Функция класса в C++ может быть объявлена со спецификатором `const`, который записывается после списка параметров перед телом функции. Подобные функции-члены могут только читать значения данных класса, но им не разрешается изменять эти значения. Спецификатор `const` записывается как в прототипе, так и при определении метода. Использование константных методов запрещает случайное изменение данных класса (листинг 14.11). Перечислим некоторые свойства этих функций:

- Метод со спецификатором `const` можно вызывать для любых объектов.
- Для объекта, являющегося константой, можно вызвать константный метод, но нельзя вызывать метод, объявленный без спецификатора `const`.
- Константный метод не может изменять значения членов-данных объекта.

Листинг 14.11. Пример использования методов со спецификатором `const`

```
// Main.cpp - главная функция
#include "Menu.h"
int main ( )
{       CMenu menu ; menu.showMenuAndSelect( ) ; return 0 ;       }
// Menu.h - спецификация класса CMenu
#pragma once
class CBook ;
class CMenu
{
public:
       CBook* m_p ;  // указатель на объект типа класса CBook
       CMenu ( ) ;   ~CMenu ( ) ;
```

```
        void showMenuAndSelect ( ) ; // активизация и выполнение услуги
private:
        // методы для ввода и установки данных объекта-книги
        void setAuthor ( ) const ;   void setTitle ( ) const ;
        void setYear ( ) const ;
        // методы для просмотра состояния объекта-книги
        void viewAuthor ( CBook* ) const ;
        void viewTitle ( CBook* ) const ;
        void viewYear ( CBook* ) const ;
        void view ( CBook* ) const ;
        void createAndViewBook ( ) ;        // создание нового объекта
        void createAndViewCopy ( ) const ;  // создание копии объекта
} ;
// Menu.cpp - реализация класса CMenu
#include <iostream>
using namespace std ;
#include "Menu.h"
#include "Book.h"
const static int N ( 512 ) ;
// конструктор и деструктор
CMenu :: CMenu ( ) : m_p ( 0 ) {    }
CMenu :: ~CMenu ( ) { }
// вывод пунктов меню, активизация и выполнение услуги
void CMenu :: showMenuAndSelect ( )
{
        char choice;
        do
        {
                cout << "\n1  Create and show new Book" ;
                cout << "\n2  Create and show copy" ;
                cout << "\n3  Set Author" ;
                cout << "\n4  Set Title" ;
                cout << "\n5  Set Year" ;
                cout << "\n6  View Book" ;
                cout << "\n0  Exit\n" ;
                cout << "Please, your choice -> " ; cin >> choice ;
                cin.ignore ( ) ;
                if ( m_p )
                        switch ( choice )
```

```
                    {
                    case '1': createAndViewBook ( ) ; break ;
                    case '2': createAndViewCopy ( ) ; break ;
              case '3': setAuthor ( ) ; break ;
                    case '4': setTitle ( ) ; break ;
                    case '5': setYear ( ) ; break ;
                    case '6': view ( m_p ) ; break ;
              }
              else
                    if ( choice == '1' ) createAndViewBook ( ) ;
       } while ( choice != '0' ) ;
}
// создание и просмотр нового объекта-книги
void CMenu :: createAndViewBook ( )
{
       if ( ! m_p )  {      delete m_p ; m_p = 0 ;       }
       m_p = new CBook ;
       setAuthor ( ) ; setTitle ( ) ; setYear ( ) ; view ( m_p ) ;
}
// создание и просмотр копии объекта-книги
void CMenu :: createAndViewCopy ( ) const
{
       CBook* pCopy = new CBook ( *m_p ) ;
       view ( pCopy ) ; delete pCopy ;
}
// методы для ввода и установки данных объекта-книги
void CMenu :: setAuthor ( ) const
{
       char author [ N ] ;          cout << "\nEnter Author ->\t" ;
       cin.getline ( author, N ) ; m_p -> setAuthor ( author ) ;
}
void CMenu :: setTitle ( ) const
{
       char title [ N ] ;           cout << "\nEnter Title ->\t" ;
       cin.getline ( title, N ) ;   m_p -> setTitle ( title ) ;
}
void CMenu :: setYear ( ) const
{
       int year ;                   cout << "\nEnter Year ->\t" ;
```

```
        cin >> year ;                m_p -> setYear ( year ) ;
}
// методы для просмотра состояния объекта-книги
void CMenu :: viewAuthor ( CBook* p ) const
{       cout << "\nAuthor:\t" << p -> getAuthor ( ) << endl ;     }
void CMenu :: viewTitle ( CBook* p ) const
{       cout << "Title:\t" << p -> getTitle ( ) << endl ;         }
void CMenu :: viewYear ( CBook* p ) const
{       cout << "Year:\t" << p -> getYear ( ) << endl ;           }
void CMenu :: view ( CBook* p ) const
{       viewAuthor ( p ) ; viewTitle ( p ) ; viewYear ( p ) ;     }
// Book.h - спецификация класса CBook
#pragma once
class CBook
{
public:
        CBook ( ) ; CBook ( char*, char*, int ) ; CBook ( CBook & ) ;
        ~CBook ( ) ;
private:
        char *m_pAuthor ;            // указатель на автора
        char *m_pTitle ;             // указатель на название
        int m_year ;                 // год издания
public:
        void setAuthor ( const char* ) ;
        void setTitle ( const char* ) ;
        void setYear ( const int ) ;
        // константные методы возврата значений
        char* getAuthor ( void ) const ;
        char* getTitle ( void ) const ;
        int getYear ( void ) const ;
} ;
// Book.cpp - реализация класса CBook
#include <iostream>
using namespace std ;
#include "Book.h"
void CBook :: setAuthor ( const char* author )    // установить автора
{
        delete [ ] m_pAuthor ;
        m_pAuthor = new char [ strlen ( author ) + 1 ] ;
```

```
        strcpy_s ( m_pAuthor, strlen ( author ) + 1, author ) ;
}
void CBook :: setTitle ( const char* title )      // установить название
{
        delete [ ] m_pTitle ;
        m_pTitle = new char [ strlen ( title ) + 1 ] ;
        strcpy_s ( m_pTitle, strlen ( title ) + 1, title ) ;
}
void CBook :: setYear ( const int year )          // установить год
{       m_year = year ;       }
char* CBook :: getAuthor ( void ) const           // вернуть автора
{       return m_pAuthor ;    }
char* CBook :: getTitle ( void ) const            // вернуть название
{       return m_pTitle ;     }
int CBook :: getYear ( void ) const               // вернуть год издания
{       return m_year ;       }
// конструктор по умолчанию
CBook :: CBook ( ) : m_year ( 0 ), m_pTitle ( "" ), m_pAuthor ( "" ) { }
// конструктор с параметрами
CBook :: CBook ( char *author, char *title, int year )
: m_year ( year ), m_pTitle ( title ), m_pAuthor ( author ) {    }
// конструктор копирования
CBook :: CBook ( CBook & o ) : m_year ( o.m_year )
{
        m_pAuthor = new char [ strlen ( o.m_pAuthor ) + 1 ] ;
        strcpy_s ( m_pAuthor, strlen ( o.m_pAuthor ) + 1, o.m_pAuthor ) ;
        m_pTitle = new char [strlen ( o.m_pTitle ) + 1 ] ;
        strcpy_s ( m_pTitle, strlen ( o.m_pTitle ) + 1, o.m_pTitle ) ;
}
// деструктор
CBook :: ~CBook ( )
{       delete [ ] m_pTitle ; delete [ ] m_pAuthor ;      }
```

Пример включает ряд константных методов, объявленных в классах `CMenu` и `CBook`. Класс `CMenu` описывает меню программы, а знакомый по предыдущим примерам класс `CBook` — книги.

Для класса `CMenu` со спецификатором `const` определены методы для просмотра состояния объекта типа `CBook`, имеющего адрес, который хранится в члене-

данном `m_p` этого класса, а также для установки значений для активного объекта и для создания копии активного объекта. Все эти методы являются константными, потому что не изменяют значения единственного данного класса — указателя `m_p` на тип `CBook`. Метод `createAndViewBook ( )` для создания нового объекта-книги не является константным, так как изменяет значение указателя `m_p`. Открытый метод `ShowMenuAndSelect ( )` для вывода меню и активизации выбранного пользователем пункта меню тоже не константный в связи с тем, что внутри этого метода происходит вызов `createAndViewBook ( )`.

В классе `CBook` константными определены только те методы, которые связаны с возвратом значений данных: `getAuthor ( )`, `getTitle ( )` и `getYear ( )`.

Исполнение программы начинается с объявления объекта `menu` класса `CMenu`, для которого вызывается метод `ShowMenuAndSelect ( )`. Программа завершается, когда заканчивается работа вызванного метода. Чтобы метод выполнил какое-либо из предложенных в списке действий, следует ввести соответствующий действию символ. Введенное значение получает переменная `choice`, после чего оператор `switch` анализирует выбор пользователя. Если значение совпадает с какой-либо меткой `case`, управление передается указанному в `case` методу класса `CMenu`. Методы класса `CMenu` отвечают лишь за взаимодействие с пользователем, ввод и вывод переменных, а также посредством указателя `m_p` активизирует методы класса `CBook`. За создание объектов-книг, изменение значений данных объектов и возврат значений отвечает класс `CBook`.

В принципе, программа не выполняет никакой особенной обработки данных объекта, тем не менее, это первая объектно-ориентированная программа в книге. Объект `menu` типа `CMenu` получает от пользователя необходимые значения простых переменных, взаимодействует с объектом типа `CBook` (адрес которого хранится в `m_p`), передавая полученные значения методам класса `CBook`. Объект, на который указывает `m_p`, выполняет вызванные методы и возвращает объекту `menu` значения, которые `menu` использует для предоставления их пользователю.

В листинге предлагается протокол исполнения программы, в котором вводимые пользователем значения выделены курсивом. Для экономии места в книге пункты меню показаны только один раз, хотя в программе они выводятся каждый раз после выполнения выбранного пользователем действия (листинг 14.12).

Листинг 14.12. Протокол выполнения программы

```
1  Create and show new Book
2  Create and show copy
3  Set Author
4  Set Title
```

```
5  Set Year
6  View Book
0  Exit
Please, your choice -> 1
Enter Author -> Артюр Рембо
Enter Title -> По
Enter Year -> 1988
Author: Артюр Рембо
Title: По
Year: 1988
Please, your choice -> 4
Enter Title -> Поэзия
Please, your choice -> 6
Author: Артюр Рембо
Title: Поэзия
Year: 1988
Please, your choice -> 2
Author: Артюр Рембо
Title: Поэзия
Year: 1988
Please, your choice -> 1
Enter Author -> Марина Цветаева
Enter Title -> Избранное
Enter Year -> 1990
Author: Марина Цветаева
Title: Избранное
Year: 1990
Please, your choice -> 6
Author: Марина Цветаева
Title: Избранное
Year: 1990
Please, your choice -> 0
```

Можно создавать новый объект сколько угодно раз, редактировать данные этого объекта и получать его копию. Однако программа не способна сохранить множество объектов, все ранее введенные объекты-книги теряются. Конечно, одним из выходов является организация массива объектов, но существует лучшее решение проблемы. Как можно работать с набором объектов, рассматривается в *главе 15*.

Класс *string*

В разделе рассматриваются отдельные возможности стандартного класса `string`, который позволяет определить в программе строковый тип данных. Чтобы получить доступ к методам класса, следует включить в исходный текст программы библиотечный заголовок `<string>`.

Объекты класса `string` не заканчиваются нулевым байтом, как это принято при представлении строк с помощью символьных массивов. При обработке строк типа `string` необходимо использовать методы класса `string`.

Объявление и инициализация строк

В листинге 14.13 объявлены три строки.

Листинг 14.13. Пример объявления и инициализации строк типа `string`

```
string s1 ; string s2 ( "Hi" ) ; string s3 = "world" ;
```

При создании строки `s1` аргументы не задаются, и конструктор класса создает пустую строку. Строка `s2` создается конструктором с одним аргументом и инициализируется при создании значением константы `"Hi"`. В объявлении `s3` для инициализации используется операция присваивания.

Операторы для объектов класса

В классе `string` определены операторы для работы со строками, которые позволяют использовать объекты класса в выражениях программы и не использовать специальные функции, как это было с символьными массивами. Перечень и примеры операторов представлены в табл. 14.1. В качестве операндов используются строки `s1, s2, s3` из предыдущего примера.

Таблица 14.1. *Операторы для объектов класса* ***`string`***

Оператор	Название операции	Пример	Результат
`=`	Присваивание	`s1 = s3 ;`	`s1 = "world"`
`+`	Конкатенация	`s1 = s2 + s3 ;` `s1 = s2 + ", " + s3 ;`	`s1 = "Hiworld"` `s1 = "Hi, world"`
`+=`	Конкатенация с замещением	`s1 += "!!!" ;`	`s1 = "Hi, world!!!"`

Таблица 14.1 *(окончание)*

Оператор	Название операции	Пример	Результат
==	Равенство	`if ( s2 == "Hi" ) cout << "EQ\n" ;`	EQ
!=	Неравенство	`if ( s3 != "Hi" ) cout << "NO\n" ;`	NO
< <= > >=	Отношения	`if ( s1 > s2 ) cout << "YES\n" ;`	YES
[ ]	Индексация	`char c = s1 [ 2 ] ;`	`c = ','`
<<	Вывод	`cout << s1 << endl ;`	`Hi, world!!!`
>>	Ввод	`string s ; cin >> s ;`	`s` получит введенное значение

Примечание

В операциях отношения строки сравниваются в лексикографическом порядке.

Основные методы обработки строк

Удаление символов

Прототип:

```
basic_string& erase ( size_type _Pos = 0, size_type _Count = npos ) ;
```

Описание: Удаляет `_Count` символов из вызывающего объекта, начиная с позиции, заданной параметром `_Pos`. По умолчанию `_Count` принимает значение константы `string::npos`, которая равна количеству символов в вызывающем объекте (листинг 14.14).

Листинг 14.14. Пример использования метода `erase ( )`

```
string s = "My program" ;
s.erase ( 3, 4 ) ;
cout << s << endl ;          // выводится "My ram"
cout << s.erase ( ) ;        // выводится пустая строка
```

Вставка символов

Прототип:

```
basic_string& insert ( size_type _Pos, const basic_string& _Str ) ;
```

Описание: Вставляет строку, заданную параметром `_Str`, в вызывающий объект, начиная с индекса, заданного параметром `_Pos` (листинг 14.15).

Листинг 14.15. Пример использования метода `insert ( )`

```
string s = "My ram", ss ( "prog" ) ;
cout << s << endl ;  // выводится "My ram"
s.insert ( 3, ss ) ;
cout << s << endl ;  // выводится "My program"
```

Замена символов

Прототип:

```
basic_string& replace ( size_type _Pos, size_type _Num,
const basic_string& _Str ) ;
```

Описание: Заменяет `_Num` символов вызывающего объекта, начиная с позиции, заданной `_Pos`, строкой, заданной параметром `_Str` (листинг 14.16).

Листинг 14.16. Пример использования метода `replace ( )`

```
string s1 ( "I like ice cream" ), s2 ( "apples" ), s3 ( "a vacation!" ) ;
s1.replace ( 7, 6, s2 ) ;
cout << s1 << endl ;          // выводится "I like appleseam"
s1.replace ( 7, 9, s2 ) ;
cout << s1 << endl ;          // выводится "I like apples"
s1.replace ( 7, 6, s3 ) ;
cout << s1 << endl ;          // выводится "I like vacation!"
```

Присваивание части строки

Прототип:

```
basic_string& assign ( const basic_string& _Str, size_type _Off,
size_type _Count ) ;
```

Описание: Присваивает вызывающему объекту `_Count` символов из строки `_Str`, начиная с индекса, заданного параметром `_Off`. Возвращает ссылку на вызывающий объект. Если количество символов в строке `_Str`, начиная с индекса `_Off`, будет меньше значения `_Count`, вызывающему объекту будут присвоены только оставшиеся в строке `_Str` символы (листинг 14.17).

Листинг 14.17. Пример использования метода `assign`

```
string s1 = "As old as the hills" ;
string s2 ;
s2.assign ( s1, 10, 9 ) ;
cout << s1 << endl ;          // выводится "As old as the hills"
cout << s2 << endl ;          // выводится "the hills"
```

Определение длины строки

Прототипы:

```
size_type length ( ) const ; size_type size ( ) const ;
```

Описание: Оба метода возвращают количество символов в строке типа `string` (листинг 14.18).

Листинг 14.18. Пример использования методов `length ( )` и `size ( )`

```
string s = "Look before you leap." ;
cout << s.size ( ) << endl ;                // выводится 21
cout << s.length ( ) << endl ;              // выводится 21
s = "" ; cout << s.size ( ) << endl ;       // выводится 0
```

Поиск символов от начала строки

Прототип:

```
size_type find ( const basic_string& _Str, size_type _Off = 0 ) const ;
```

Описание: Выполняет поиск первого вхождения подстроки, содержащейся в объекте `_Str`, начиная с позиции, которая задана параметром `_Off`. Возвращает индекс первого вхождения подстроки, иначе возвращает –1 (листинг 14.19).

Листинг 14.19. Пример использования метода `find ( )`

```
#include <iostream>
#include <string>
using namespace std ;
int main ( )
{
        string s ( "In twisting a twist one twist should untwist" ) ;
        string ss = "twist" ; string rest ; int i = 0 ;
        cout << s << endl << endl ;
```

```
        do
        {
                i = s.find ( ss, i ) ;        if ( i == -1 ) break ;
                cout << "index = " << i << endl ;
                rest.assign ( s, i, s.size ( ) ) ;
                cout << rest << endl << endl ;
                ++i ;
        } while ( i != -1 ) ;
        return 0 ;
}
```

В цикле `do while` ищутся все вхождения подстроки `twist` в исходной строке `s`. Если метод `find ( )` возвращает –1, цикл завершается. В теле цикла выводится индекс `i` каждого вхождения подстроки и остаток строки `rest`, который формируется при помощи метода `assign ( )`. Результаты исполнения программы показаны на рис. 14.5.

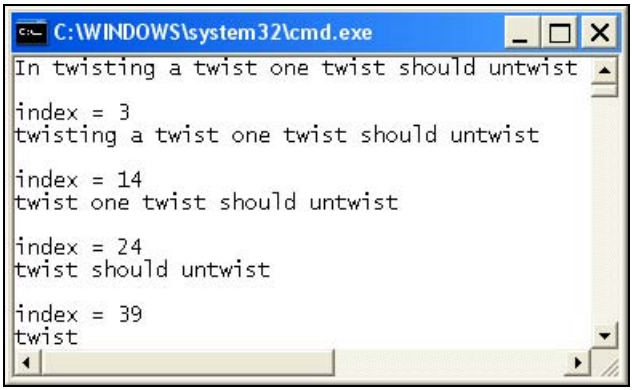


Рис. 14.5. Результаты использования метода `find ( )`

Поиск символов от конца строки

Прототип:

```
size_type rfind ( const basic_string& _Str, size_type _Off = npos )const;
```

Описание: Выполняет поиск последнего вхождения подстроки, содержащейся в объекте `_Str`, начиная с позиции, которая задана параметром `_Off`. Возвращает индекс последнего вхождения подстроки. Если подстрока не найдена, возвращает константу `string :: npos`, равную –1 (листинг 14.20).

Листинг 14.20. Пример использования метода `rfind ( )`

```
string s = "The untwisted twist would untwist the twist" ;
cout << s.rfind ( 'u' ) << endl ;          // выводится 26
cout << s.rfind ( "he" ) << endl ;         // выводится 35
cout << s.rfind ( "he", 30 ) << endl ;     // выводится 1
```

Преобразование в строку с нулевым байтом

Прототип:

```
const value_type *c_str ( ) const ;
```

Описание: Возвращает указатель на строку, которая содержится в вызывающем объекте, приписывая в конец нулевой байт. Полученную строку нельзя изменять (листинг 14.21).

Листинг 14.21. Пример использования метода `c_str ( )`

```
string s = "ABC" ;    const char *ps ;      // указатель только константный
ps = s1.c_str ( ) ;   cout << ps << endl ;  // выводится "ABC"
```

Примечание

Каждый рассмотренный метод класса `string` имеет несколько прототипов, и в классе определено очень большое количество других методов обработки строк, исчерпывающую информацию о которых следует искать в справочной документации.

Объектно-ориентированная модель системы

Процесс разработки объектно-ориентированных программных систем объединяет три последовательно сменяющих друг друга этапа:

1. Объектно-ориентированный анализ (Object Oriented Analysis), который состоит в исследовании задачи с точки зрения объектов реального мира и в определении требований к программной системе.
2. Объектно-ориентированное проектирование (Object Oriented Design), когда внимание разработчиков акцентируется на программных классах и поиске логических путей выполнения установленных на этапе анализа требований.

3. Объектно-ориентированное программирование (Object Oriented Programming), которое обеспечивает реализацию классов проектирования на выбранном языке программирования для получения конкретных результатов.

Основными элементами этих этапов являются объекты, организованные в классы. Современный стиль программирования поддерживает идею взаимодействия объектов, управляемого событиями (event-driven). Исполняемая программа состоит из программных объектов, которые реагируют на случайные события, наступающие под воздействием пользователей, электромеханических устройств, времени или других систем. Чтобы выполнить задачу, активные объекты могут запросить другие объекты о предоставлении им тех или иных услуг. Выполнив задачу, объекты переходят в ожидание нового события.

Создание программной системы — сложный процесс. Для борьбы со сложностью используется моделирование, результатом которого является построение набора моделей системы. Каждая модель (model) представляет собой абстракцию, которая описывает суть сложной проблемы без учета несущественных деталей, что делает ее более понятной. Визуальным моделированием называется процесс графического представления моделей с помощью принятой нотации, например, нотации унифицированного языка моделирования UML (Unified Modeling Language). Разрабатываемые модели должны показывать архитектуру системы на уровне взаимодействия между пользователями и системой, объектов внутри системы, а также на уровне взаимодействия между различными системами.

Популярным архитектурным стилем объектно-ориентированных приложений является трехзвенный стиль (three-tiered architecture). Этот стиль включает три базовых уровня: уровень представления (user service), уровень бизнес-правил (business service) и уровень управления данными (data service). Согласно трехзвенному стилю архитектуры, все программные классы приложения можно разделить на следующие категории:

- ❒ классы взаимодействия окружающей среды с элементами приложения;
- ❒ классы логики приложения (бизнес-правила);
- ❒ классы управления данными.

Классы уровня представления описывают объекты, которые получают входные данные от внешнего источника и предоставляют выходные данные внешнему источнику. В большинстве случаев в качестве внешнего источника выступает пользователь, хотя это может быть также некоторое устройство автоматизированного ввода-вывода. Объекты уровня представления осуществляют навигацию пользователя через приложение.

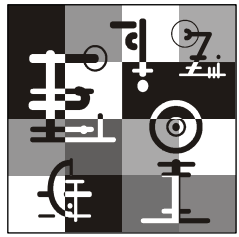
Классы уровня бизнес-правил описывают объекты, которые управляют функциями и процессами, выполняемыми приложением. Эти функции и процессы реализуются методами множества объектов уровня бизнес-правил. Методы активных объектов получают данные или от объектов уровня представления (когда пользователь запрашивает информацию), или от каких-либо других объектов уровня бизнес-правил. В общем, объекты бизнес-правил выполняют некоторое манипулирование данными, которое определяется требованиями к приложению.

Классы уровня управления данными описывают объекты, которые по существу взаимодействуют с системой управления хранилищем данных. Например, с системой управления базами данных, иерархической файловой системой либо с другим источником данных внешней программной системы. Методы объектов этого уровня получают данные от объектов уровня бизнес-правил, хотя в простых приложениях данные могут непосредственно передаваться объектами уровня представления.

Объектно-ориентированные модели программных систем предполагают исследование решаемых задач с точки зрения реальных объектов, определение логики программного решения — с точки зрения программных классов (их свойств, поведения, взаимосвязей), реализацию логики программного решения — с точки зрения языков программирования, которые поддерживают принципы инкапсуляции, наследования и полиморфизма.

Примечание

В очень многих книгах по программированию на C++ просмотр состояния программных объектов, не имеющих отношения к взаимодействию с пользователем, реализуется как метод класса объектов уровня бизнес-правил, а также зачастую в конструктор класса помещаются операторы ввода значений для инициализации данных. Такой подход является неверным с точки зрения распределения обязанностей между классами для объектно-ориентированных приложений, использующих трехзвенный стиль архитектуры. В этой книге отдается предпочтение организации ввода необходимых данных и просмотра состояния программных объектов в специальном классе пользовательского интерфейса (см. класс `CMenu` в примерах). Когда измененяются требования к вводу/выводу или изменяется функциональность программы, класс объектов в большинстве случаев не меняется (изменяется только класс интерфейса пользователя), что дает возможность повторного использования класса объектов без перекомпиляции как в одном и том же проекте, так и в разных проектах.



Глава 15

Композиция

Между объектами программы существуют связи, которые поддерживают их взаимодействие. Множество однотипных связей между объектами двух разных классов называют отношением. Распространенным видом отношений является отношение типа часть — целое (part of или has), когда один класс представляет целое, а другие — его части. Такой тип отношений называют агрегацией (aggregation). Пусть класс целого называется `CWhole`, а класс части — `CPart`. При помощи языка моделирования UML (Unified Modeling Language), используемого для проектирования программных классов, это отношение графически изображается как показано на рис. 15.1.

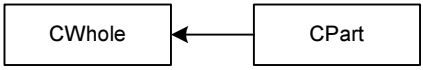


Рис. 15.1. Изображение агрегации в UML

Классы в UML изображаются прямоугольником, агрегация изображается линией с ромбиком на конце, который располагается у класса, представляющего целое. На рисунке два класса `CWhole` и `CPart` соединены отношением агрегации. Особенности реализации агрегации, связанные с созданием и уничтожением целого и частей, в книге не рассматриваются. В этой главе исследуется композиция — возможность включения в класс в качестве членов-данных объектов другого класса.

Для включения объекта части `m_part` класса `CPart` в класс `CWhole` есть две альтернативы:

- явно объявить данное — `CPart m_part`;
- объявить указатель или ссылку на тип части — `CPart* m_p` или `CPart& m_r`.

Объекты-части создаются в том порядке, в котором они объявлены во включающем их классе целого и до того, как будет создан объект целого.

Можно задавать или нет начальные значения объекту части, но при их отсутствии класс частей обязательно должен иметь конструктор по умолчанию. Для инициализации объекта части в списке инициализации конструкторов класса целого явно вызываются конструкторы класса части (листинг 15.1).

Листинг 15.1. Пример композиции с явным объявлением части

```
// Train.h — спецификация класса CTrain
#pragma once
#include "Station.h"
class CTrain                    // описание поезда
{
public:
       int m_num ;              // номер поезда
       CStation m_sd ;          // станция отправления
       CTrain ( ) ; CTrain ( int, char* ) ; CTrain ( CTrain& ) ;
       ~CTrain ( ) ;
} ;
// Train.cpp — реализация класса CTrain
#include "Train.h"
CTrain :: CTrain ( ) : m_num ( 0 ) { }
CTrain :: CTrain ( int num, char* p ) : m_num ( num ), m_sd ( p ) { }
CTrain :: CTrain ( CTrain& t ) : m_num ( t.m_num ), m_sd ( t.m_sd ) { }
CTrain :: ~CTrain ( ) { }
// Station.h — спецификация класса CStation
#pragma once
class CStation                  // описание ж/д станции
{
public:
       char* m_pName ;          // указатель на название станции
       CStation ( ) ; CStation ( char* ) ; CStation ( CStation& ) ;
       ~CStation ( ) ;
} ;
// Station.cpp — реализация класса CStation
#include <string>
#include "Station.h"
CStation :: CStation ( ) : m_pName ( "" ) { }
```

```
CStation :: CStation ( char* pName ) : m_pName ( pName ) { }
CStation :: CStation ( CStation& s ) : m_pName ( s.m_pName ) { }
CStation :: ~CStation ( ) { }
// main.cpp - главная функция
#include <iostream>
using namespace std ;
#include "Train.h"
void view ( CTrain* ) ;      // просмотр состояния
int main ( )
{
       CTrain t ;                                  view ( &t ) ;
       CTrain* p = new CTrain ( 2, "Moscow" ) ;    view ( p ) ;
       CTrain copy ( *p ) ;                        view ( &copy ) ;
       delete p ;
}
void view ( CTrain* p )
{
       cout << "Train number:\t" << ( p -> m_num ) << '\t' ;
       cout << "Departure station:\t" << p -> m_sd.m_pName << endl ;
}
```

Главная функция при помощи функции `view ( )` выведет следующие строки:

```
Train number: 0  Departure station:
Train number: 2  Departure station: Moscow
Train number: 2  Departure station: Moscow
```

Первым будет выполняться конструктор по умолчанию класса `CStation` для создания члена-данного класса `CTrain` с идентификатором `m_sd`. Затем продолжает свою работу конструктор по умолчанию класса `CTrain`, который создает объект `t`, состояние которого выводится функцией `view ( )`. На следующем шаге исполнения выделяется блок оперативной памяти, необходимый для размещения объекта типа класса `CTrain`. Чтобы создать этот объект, вызывается конструктор с параметром `"Moscow"` класса `CStation` и конструктор с параметрами класса `CTrain` завершает формирование объекта. Состояние созданного объекта выведено во второй строке.

Наконец, при помощи выражения `CTrain copy ( *p )` создается копия только что созданного объекта, последовательно вызывая конструкторы копирования классов `CStation` и `CTrain`. Состояние копии также выводится. В конце главной функции для разрушения объекта, на который указывает `p`, выполняется оператор `delete`, вызывая деструкторы классов `CTrain` и `CStation`. Перед

выходом из программы автоматически будут вызваны деструкторы `~CTrain ( )`, `~CStation ( )` сначала для объекта `copy`, потом для объекта `t`.

Листинг 15.2. Пример композиции с объявлением части через указатель

```
// Train.h — спецификация класса CTrain
#pragma once
#include "Station.h"
class CTrain                    // описание поезда
{
public:
       int m_num ;              // номер поезда
       CStation* m_ps ;         // указатель на станцию отправления
       CTrain ( ) ; CTrain ( int, char* ) ; CTrain ( CTrain& ) ;
       ~CTrain ( ) ;
} ;
// Train.cpp — реализация класса CTrain
#include "Train.h"
CTrain :: CTrain ( ) : m_num ( 0 ), m_ps ( new CStation ) { }
CTrain :: CTrain ( int num, char* p )
: m_num ( num ), m_ps ( new CStation ( p ) ) { }
CTrain :: CTrain ( CTrain& t )
: m_num ( t.m_num ), m_ps ( new CStation ( *t.m_ps ) ) { }
CTrain :: ~CTrain ( ) { delete m_ps ; }
// изменение в функции view ( ) в файле main.cpp
cout << "Departure station:\t" << p -> m_ps -> m_pName << endl ;
```

В листинге 15.2 показаны изменения спецификации и реализации класса `CTrain`, а также изменения в последней строке исходного текста функции `view ( )`. Остальной исходный код остался таким же, как в предыдущем примере.

Когда объект другого класса объявляется через указатель, для его создания требуется динамическое выделение памяти, которое и выполняют конструкторы класса `CTrain`. Деструктор класса разрушает объект, выполняя операцию `delete` для указателя `m_ps`. В функции `view ( )` операция точка для доступа к данному `m_pName` заменяется операцией стрелка, потому что `m_ps` — указатель. Результат исполнения новой версии программы будет идентичен версии с явным объявлением объекта (листинг 15.3).

Листинг 15.3. Пример композиции с объявлением части через ссылку

```
// Train.h — спецификация класса CTrain
#pragma once
#include "Station.h"
class CTrain                    // описание поезда
{
public:
       int m_num ;              // номер поезда
       CStation& m_rs ;         // ссылка на станцию отправления
       CTrain ( ) ; CTrain ( int, char* ) ; CTrain ( CTrain& ) ;
       ~CTrain ( ) ;
} ;
// Train.cpp — реализация класса CTrain
#include "Train.h"
CTrain :: CTrain ( ) : m_num ( 0 ), m_rs ( * ( new CStation ) ) { }
CTrain :: CTrain ( int num, char* p )
: m_num ( num ), m_rs ( * ( new CStation ( p ) ) ) { }
CTrain :: CTrain ( CTrain& t )
: m_num ( t.m_num ), m_rs ( * ( new CStation ( t.m_rs ) ) ) { }
CTrain :: ~CTrain ( ) { delete &m_rs ; }
// изменение в функции view ( ) в файле main.cpp
cout << "Departure station:\t" << p -> m_rs.m_pName << endl ;
```

В примере показаны изменения спецификации и реализации класса `CTrain`, а также изменения в последней строке исходного текста функции `view ( )`. Остальной исходный код остался таким же, как в примере с явным объявлением.

Когда объект другого класса объявляется через ссылку, необходимо создавать содержимое по адресу, которого пока не существует. Поэтому конструкторы класса `CTrain` используют разыменовывание указателя, возвращаемого оператором динамического выделения памяти `new`. В деструкторе класса для разрушения созданного объекта операция `delete` получает его адрес. Функция `view ( )` для доступа к `m_pName` использует точку, так как `m_rs` — это ссылка. Результат исполнения программы будет аналогичен версии с явным объявлением объекта (листинг 15.4).

Листинг 15.4. Пример использования композиции для работы с объектами

```
// главная функция
#include "Menu.h"
int main ( )
{       CMenu menu ; menu.showMenuAndSelect ( ) ; return 0 ;      }
// Menu.h — Спецификация класса меню CMenu
#include "Catalogue.h"
class CMenu
{
public:
        CCatalogue* m_p ;             // указатель на каталог книг
        CMenu ( ) ; ~CMenu ( ) ;      // конструктор и деструктор
        void showMenuAndSelect ( ) ; // активизация меню
private:
        void find ( ) const ;         // поиск книги по названию
        void view ( ) const ;         // просмотр состояния каталога
        void ins ( ) ;                // добавление книги
        void del ( ) ;                // удаление книги по названию
        void sort ( ) ;               // сортировка в каталоге по авторам
        void align ( int ) const ;    // выравнивание при выводе
} ;
// Menu.cpp — Реализация класса CMenu
#include <iostream>
#include <iomanip>
using namespace std ;
#include "Menu.h"
const static int N ( 512 ) ;
CMenu :: CMenu ( ) : m_p ( new CCatalogue ) { }    // конструктор
CMenu :: ~CMenu ( ) { delete m_p ; }               // деструктор
void CMenu :: showMenuAndSelect ( )                // активизация меню
{
        char choice ;
        do
        {
                cout << "\n1. Find and look up" ;
                cout << "\n2. Insertion" ;
                cout << "\n3. Deletion" ;
                cout << "\n4. Sorting" ;
```

```
                cout << "\n5. View" ;
                cout << "\n0. Exit\n" ;
                cout << "Please, your choice -> " ;
                cin >> choice ; cin.ignore ( ) ;
                if ( m_p -> getNumberOfBook ( ) )
                        switch ( choice )
                        {
                        case '1': find ( ) ; break ;
                        case '2': ins ( ) ; break ;
                        case '3': del ( ) ; break ;
                        case '4': sort ( ) ; break ;
                        case '5': view ( ) ; break ;
                        }
                else
                        if ( ! m_p -> getNumberOfBook ( ) && choice == '2'
)
                                ins ( ) ;
                } while ( choice != '0' ) ;
}
void CMenu :: find ( ) const                        // поиск по ключу
{
        string title ;
        cout << "\nEnter Title -> " ; getline ( cin, title ) ;
        int iKey = m_p -> find( title ) ;
        if ( iKey != -1 )
        {
                cout << "Author: " ;
                cout << m_p -> m_pBook [ iKey ] -> getAuthor( ) << endl ;
                cout << "Title: " ;
                cout << m_p -> m_pBook [ iKey ] -> getTitle( ) << endl ;
                cout << "Year: " ;
                cout << m_p -> m_pBook [ iKey ] -> getYear( ) << endl ;
        }
        else    cout << title << " - NOT FOUND\n" ;
}
void CMenu :: view ( ) const                        // просмотр состояния
{
        int n = m_p -> getNumberOfBook ( ) ;
        align ( 20 ) ; cout << "Author" ; align ( 50 ) ; cout << "Title" ;
```

```
        align ( 10 ) ; cout << "Year" ; cout << endl ;
        for ( int i = 0; i < n ; i++ )
        {
                align ( 20 ) ; cout << m_p -> m_pBook [i] -> getAuthor( )
;
                align ( 50 ) ; cout << m_p -> m_pBook [i] -> getTitle( ) ;
                align ( 10 ) ; cout << m_p -> m_pBook [i] -> getYear( ) ;
        }
}
void CMenu :: ins ( )                                   // добавление
{
        string author, title ; int year ;
        cout << "\nEnter Title to add -> " ; getline ( cin, title ) ;
        if ( m_p -> find ( title ) < 0 )
        {
                cout << "\nEnter Author to add -> " ;
                getline ( cin, author ) ;
                cout << "\nEnter Year to add -> " ; cin >> year ;
                if ( m_p -> ins ( author, title, year ) )
                        cout << "\nINSERTION IS COMPLETED!\n";
        }
        else    cout << title << " - INSERTION IS IMPOSSIBLE\n" ;
}
void CMenu :: del ( )                                   // удаление
{
        string title ;
        cout << "\nEnter Title -> " ; getline ( cin, title ) ;
        if ( m_p -> del ( title ) )  cout << "\nDELETION IS COMPLETED!\n";
        else    cout << title << " - DELETION IS IMPOSSIBLE\n" ;
}
void CMenu :: sort ( )                                  // сортировка
{       m_p -> sort ( ) ;     cout << "\nSORTING IS COMPLETED!\n" ;       }
void CMenu :: align ( int n ) const                     // выравнивание
{       cout.width ( n ) ; cout.setf ( ios_base::left ) ;         }
// Catalogue.h — Спецификация класса CCatalogue
#pragma once
static const int MAX_SIZE = 100 ;
#include "Book.h"
class CCatalogue
```

```
{
       int m_q ;                              // фактическое число книг
public:
       CBook* m_pBook [ MAX_SIZE ] ;          // массив указателей на книги
public:
       CCatalogue ( ) ; ~CCatalogue ( ) ;  // конструктор и деструктор
       bool ins ( const string&, const string&, int ) ;  // добавление
       bool del ( const string& ) ;                       // удаление
       int find ( const string& ) const ;                 // поиск
       void sort ( ) ;                                    // сортировка
       int getNumberOfBook ( ) const ;     // возврат количества книг
} ;
// CCatalogue.cpp — Реализация класса CCatalogue
#include "Catalogue.h"
CCatalogue :: CCatalogue ( ) : m_q ( 0 ) { }              // конструктор
CCatalogue :: ~CCatalogue ( )                             // деструктор
{      for ( int i = 0; i < m_q; i++ )     delete m_pBook [ i ] ;       }
// возврат фактического количества книг
int CCatalogue :: getNumberOfBook ( ) const       { return m_q ; }
/*     добавление новой книги
1) проверить наличие книги в массиве;
2) если книга уже есть в массиве, вернуть ложь
 если книги нет ->
       - создать новый объект и занести в массив;
       - увеличить число книг в массиве и вернуть истину        */
bool CCatalogue ::
ins ( const string& author, const string& title, int year )
{
       if ( find ( title ) != -1 ) return false ;
       m_pBook [ m_q++ ] = new CBook ( author, title, year ) ;
       return true ;
}
/*     удаление книги из массива
1) проверить наличие книги в массиве
2) если книги нет в массиве, вернуть ложь
 если книга найдена ->
       - разрушить найденный объект;
       - сжать массив;
       - уменьшить число книг в массиве и вернуть истину.       */
```

```
bool CCatalogue :: del ( const string& key )
{
       int i = find ( key ) ;       if ( i < 0 )  return false ;
       delete m_pBook [ i ] ;
       while ( i < m_q - 1 ) m_pBook [ i++ ] = m_pBook [ i + 1 ] ;
       m_q-- ;       return true ;
}
/*     поиск книги по названию
1) проверить название каждой книги;
2) если есть совпадение, вернуть индекс книги
 если нет совпадений, вернуть -1.          */
int CCatalogue :: find ( const string& key ) const
{
       for ( int i = 0; i < m_q; i++ )
              if ( m_pBook [ i ] -> check ( key ) )      return i ;
       return -1 ;
}
// сортировка массива по авторам (метод пузырьковой сортировки)
void CCatalogue :: sort ( )
{
       for ( int j = 1; j < m_q; j++ )
              for ( int i = 0; i < m_q - 1; i++ )
                     if ( m_pBook [ i ] -> getAuthor ( ) >
                            ↳m_pBook [ i + 1 ] -> getAuthor ( ) )
                     {
                            CBook* tmp = m_pBook [ i ] ;
                            m_pBook [ i ] = m_pBook [ i + 1 ] ;
                            m_pBook [ i + 1 ] = tmp ;
                     }
}
// Book.h - спецификация класса CBook
#pragma once
#include <string>
using namespace std ;
class CBook
{
public:
       CBook ( ) ; CBook ( const string&, const string&, int ) ;
       ~CBook ( ) ;
```

```
private:
        string m_author ;              // автор
        string m_title ;               // название
        int m_year ;                   // год издания
public:
        // методы установки значений
        void setAuthor ( const string & ) ;
        void setTitle ( const string & ) ;
        void setYear ( const int ) ;
        // методы возврата значений
        string getAuthor ( ) const { return m_author ; }
        string getTitle ( ) const { return m_title ; }
        int getYear ( ) const { return m_year ; }
        // метод проверки названия книги
        bool check ( const string& ) const ;
} ;
// Book.cpp - реализация класса CBook
#include "Book.h"
CBook :: CBook ( ) : m_year ( 0 ), m_title ( "" ), m_author ( "" ) {    }
CBook :: CBook ( const string& author, const string& title, int year )
: m_year ( year ), m_title ( title ), m_author ( author ) {      }
CBook :: ~CBook ( ) {        }
// установить автора
void CBook :: setAuthor ( const string& author ) { m_author = author ; }
// установить название
void CBook :: setTitle ( const string& title ) { m_title = title ; }
// установить год издания
void CBook :: setYear ( const int year ) { m_year = year ; }
// проверка книги по названию
bool CBook :: check ( const string& key ) const
{       if ( m_title == key ) return true ; return false ;        }
```

В листинге 15.4 реализовано взаимодействие объектов трех классов: `CMenu` (описывает пользовательский интерфейс), `CCatalogue` (описывает массив из 100 указателей на объекты типа `CBook`) и `CBook` (описывает книги). Спецификации и реализации классов размещаются в отдельных файлах. Главная функция `main ( )` тоже находится в отдельном файле.

В главной функции объявляется объект класса `CMenu` с идентификатором `menu`, который вызывает открытый метод класса `ShowMenuAndSelect ( )`.

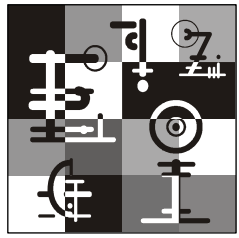
Дальнейшие действия программы зависят от реакции пользователя на предлагаемые этим методом услуги.

Класс CMenu поддерживает ввод исходных данных для взаимодействия с объектом, заданным в классе через указатель m_p, который указывает на объект класса CCatalogue. Помимо ввода, CMenu служит для вывода данных, полученных в результате работы объекта типа CCatalogue. Конструктор класса CMenu создает этот объект, а деструктор разрушает его. Объект menu активизирует меню программы, в котором указаны возможные операции с каталогом книг: поиск по названию, добавление, удаление книги с указанным названием, сортировка по авторам, просмотр состояния каталога книг. По желанию пользователя может выполняться та или иная операция, для чего объектом с указателем m_p вызывается соответствующий метод класса CCatalogue. Если в каталоге нет ни одной книги, возможна только операция добавления. Иначе доступной становится любая операция. В классе CMenu есть вспомогательный метод align ( ) для форматирования вывода. Этот метод вызывается из метода view ( ), при помощи которого выводится состояние каталога книг. Чтобы значения данных одинаково позиционировались в каждой строке, метод align ( ) применяет функции width ( ) и setf ( ) стандартного класса ios, позволяющие изменять ширину поля вывода и устанавливает флаг выравнивания для выводимых данных. Обе функции вызываются стандартным объектом cout. Ширина поля устанавливается равной значению параметра метода align ( ), а флаг выравнивания left обеспечивает выравнивание по левому краю поля вывода.

Класс CCatalogue отвечает за все операции из меню, но для их выполнения он обращается за услугами к классу CBook через указатели на объекты этого класса, которые являются его неотъемлемой составляющей. В открытой части класса объявлен массив указателей m_pBook типа CBook* размером 100, а в закрытой части объявлено фактическое число m_q сохраненных в массиве за время работы программы книг. Несмотря на то, что некоторые методы класса CCatalogue называются точно так же, как и методы класса CMenu, они принадлежат другой области видимости (CCatalogue :: для класса CCatalogue и CMenu :: для класса CMenu), поэтому представляют совершенно другие функции. Конструктор класса инициализирует нулем фактическое количество книг в каталоге. Деструктор (соответственно значению m_q) вызывает оператор delete для разрушения объектов типа CBook, указатели на которые хранятся в массиве m_pBook.

Класс CBook уже много раз рассматривался в книге. В этом примере предложена его новая версия с использованием стандартного класса string для опи-

сания автора и названия книги, а также в класс добавлен новый метод `check ( )`, проверяющий совпадение значения параметра метода и значения члена-данного `m_title`. Метод используется объектом класса `CCatalogue` при поиске книги в каталоге по названию. Объект класса `CCatalogue` не в состоянии самостоятельно выполнить проверку на совпадение значения переменной `key` с названием книги `m_title`, потому что название принадлежит книге `(*m_pBook [ i ])`, а не каталогу `(m_p)`, которому известен лишь адрес книги `(m_pBook [ i ])`. Только сама книга знает свое название и может ответить, совпадает ли оно со значением `key`. Такая организация взаимодействия программных объектов `menu`, `*m_p` и `*m_pBook [ i ]` делает программу объектно-ориентированной, когда обязанности распределяются между объектами и каждый из них несет персональную ответственность за их исполнение.



Глава 16

Наследование

Наследование (inheritance) — это механизм C++, с помощью которого один класс может приобретать свойства (данные и методы) другого класса. При помощи наследования в программах реализуется отношение типа является (is-a), называемое обобщением (generalization). Это отношение, в отличие от агрегации, не обозначает наличия каких-либо связей между объектами классов. Обобщение указывает лишь на то, что между объектами существует некоторая общность в их структуре. Например, общие данные, методы, или и то, и другое. Обобщение позволяет строить иерархию классов, переходя от общих классов к специализированным версиям.

В терминах объектно-ориентированного программирования класс, где сосредоточены наиболее общие данные и методы объектов разных классов, называется базовым (base), или родительским (parent). Класс, объекты которого наследуют данные и методы базового класса, называется производным (derived), или дочерним (child). Открытые (public) и защищенные (protected) данные и методы базового класса автоматически присваиваются производным классам. Закрытые данные и методы не видны за пределами базового класса. Производный класс может расширять набор данных, добавляя характерные именно для объектов только этого производного класса, расширять набор методов базового класса, добавляя новые, а также может переопределять методы базового класса.

Пусть существуют классы `CBase` (базовый) и его модификация `CDerived` (производный). На рис. 16.1 представлена нотация отношения обобщения между этими классами в UML. Обобщение обозначается линией с полым треугольником, вершина которого направлена в сторону базового класса. Прямоугольник каждого класса поделен на три части. Первая часть содержит имя класса, вторая — данные класса, третья — методы. Знаки плюс, решетка и минус обозначают спецификаторы `public`, `protected`, `private`, соответственно.

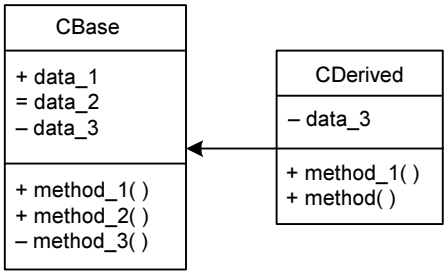


Рис. 16.1. Нотация отношения обощения в UML

Доступ к членам-данным и методам базового класса в производном классе зависит от спецификатора доступа членов базового класса, а также от спецификатора наследования. Спецификатор наследования может быть открытым `public`, защищенным `protected` и закрытым `private`. В табл. 16.1 проиллюстрированы характеристики доступа к членам базового класса из производного при разных спецификаторах наследования.

Таблица 16.1. *Спецификаторы доступа в производном классе к членам базового класса*

Спецификатор наследования	**Спецификатор члена базового класса**		
	`public`	`protected`	`private`
`public`	в производном классе `public`	в производном классе `protected`	в производном классе невидим
`protected`	в производном классе `protected`		
`private`	в производном классе `private`	в производном классе `private`	

Пусть класс `CDerived` открыто наследует данные и методы класса `CBase`. Тогда можно прокомментировать рис. 16.1 следующим образом. Класс `CDerived` наследует данные `data_1` и `data_2` базового класса `CBase`, сохраняя в классе `CDerived` открытый и защищенный спецификаторы доступа. Закрытый член-данное `data_3` может использоваться только объектами базового класса, а `data` — только объектами производного класса. Метод `method_2 ( )` наследуется производным классом, и используется как объектами базового класса, так и объектами производного, выполняя одни и те же действия для объектов этих классов. Вызов метода может осуществляться из любого места программы, так как открытый доступ сохраняется для объектов обоих классов.

Метод `CBase :: method_1 ( )` переопределяется в производном классе методом `CDerived :: method_1 ( )`. Каждый метод имеет собственную реализацию. Объекты классов вызывают метод `method_1 ( )` того класса, которому они принадлежат. Метод `method_3 ( )` может использоваться только объектами базового класса и не виден за пределами класса. Метод `method ( )` может вызываться из любого места программы только объектами производного класса.

Далее в книге рассматриваются возможности реализации отношения обобщения в C++, а также характерные особенности механизма наследования.

Одиночное наследование

Одиночное наследование предполагает, что у каждого производного класса есть только один базовый класс, от которого он наследует данные и методы обработки этих данных. Производный класс, в свою очередь, может быть базовым классом для других классов программы. Такой набор классов называется иерархией классов. Иерархическая организация наследования может иметь много уровней. Производный класс наследует открытые и защищенные данные того базового класса, который расположен выше по иерархической лестнице. Большое число уровней иерархии усложняет модификацию программы, поэтому в программе следует ограничиваться тремя уровнями иерархии. Пример наследования с тремя уровнями иерархии показан на рис. 16.2.

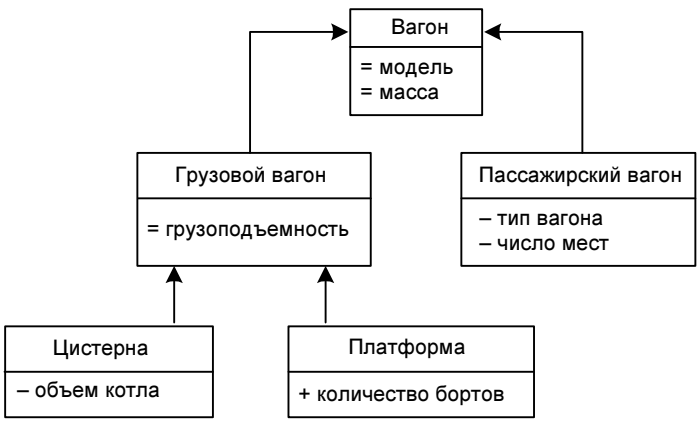


Рис. 16.2. Наследование с тремя уровнями иерархии

На рис. 16.2 базовыми классами являются два класса: `Вагон` и `Грузовой вагон`. Класс `Грузовой вагон` одновременно и базовый (от него наследуют классы

Цистерна и Платформа), и производный (он наследует свойства класса Вагон). Итак, производными от базового класса Вагон являются классы Грузовой вагон и Пассажирский вагон, котрые наследуют данные модель и масса. Производными от базового класса Грузовой вагон являются классы Цистерна и Платформа, наследуя от него данные модель, масса и грузоподъемность. Во всех производных классах существуют данные, которые специфицируют объекты своего класса. Например, у класса Платформа определяется член-данное количество бортов. Такого данного нет ни в одном из имеющихся классов.

Объявление классов при одиночном наследовании

Объявление базового класса производится как и для обычного класса. Объявление производного класса имеет следующий формат:

```
class имя_производного_класса : [ public | protected | private ]opt
имя_базового_класса
{ описание членов производного класса } ;
```

где opt — обозначает, что в объявлении может быть указан один из спецификаторов, перечисленных в квадратных скобках через символ |.

После имени производного класса ставится двоеточие, за которым следует ключевое слово спецификатора (для обозначения вида наследования) и имя базового класса. Чаще всего при наследовании используется спецификатор public.

В листинге 16.1 показано объявление базового и производного классов, которое включает только описание данных. Описание конструкторов и деструктора имеет особенности, которые будут рассмотрены далее.

Листинг 16.1. Пример объявления базового и производных классов при одиночном наследовании

```
// Car.h — Спецификация базового класса CCar (Вагон)
#pragma once
#include <string>
using namespace std ;
class CCar
{
protected:
        string m_model ;        // модель
        double m_mass ;         // масса
```

```
        // описание методов класса
} ;
// Cargo.h — Спецификация производного класса CCargo (Грузовой вагон)
#pragma once
#include "Car.h"
class CCargo : public CCar
{
protected:
        int m_carCapacity ;   // грузоподъемность
        // описание методов класса
} ;
// Tank.h — Спецификация производного класса CTank (Цистерна)
#include "Cargo.h"
class CTank : public CCargo
{
        double m_volume ;     // объем котла
        // описание методов класса
} ;
// Platform.h — Спецификация производного класса CPlatform (Платформа)
#pragma once
#include "Cargo.h"
class CPlatform : public CCargo
{
public:
        int m_boards ;        // количество бортов
        CPlatform ( ) ;
        // описание методов класса
} ;
// Carriage.h
// Спецификация производного класса CCarriage (Пассажирский вагон)
#pragma once
#include "Car.h"
class CCarriage : public CCar
{
        string m_type ;       // тип
        int m_places ;        // число мест
        // описание методов класса
} ;
```

В примере объявляются классы `CCar`, `CCargo`, `CTank`, `CPlatform` и `CCarriage`, которые на рис. 16.2 имеют имена `Вагон`, `Грузовой вагон`, `Цистерна`, `Платформа` и `Пассажирский вагон` соответственно. Из примера понятно, что объявление производных классов при одиночном наследовании отличается от объявления обычного класса двоеточием после имени класса, спецификатором наследования и указанием имени базового класса. Описание данных производных классов, базового класса и обычных классов ничем не отличаются друг от друга.

Примечание

При раздельной компиляции в заголовочный файл производного класса необходимо включить заголовочный файл базового класса.

Конструкторы при одиночном наследовании

Во избежание ошибок при создании объектов производного класса базовый и производные классы должны иметь три конструктора: по умолчанию, с параметрами и копирования.

Для производного класса объявление конструкторов по умолчанию и копирования выполняется обычным способом. В список инициализации каждый из этих конструкторов включают явный вызов соответствующего конструктора базового класса. Конструктор с параметрами для производного класса в списке параметров должен содержать параметры и производного, и базового классов. Параметры базового класса передаются конструктору с параметрами базового класса, который явно вызывается в списке инициализации производного класса.

Порядок конструирования объекта производного класса определяет следующая последовательность:

1. Сначала конструируется базовый подобъект.
2. Последовательно конструируются специфические данные, указанные в списке инициализации (если список инициализации не пустой).
3. В конце конструируется производный объект.

Приводимый в листинге 16.2 пример конструкторов продолжает рассмотрение классов `CCar`, `CCargo`, `CTank`, `CPlatform` и `CCarriage`.

Листинг 16.2. Пример объявления и реализации конструкторов при одиночном наследовании

```
// Прототипы конструкторов CCar
CCar ( ) ;
CCar ( const string&, const double ) ;
CCar ( const CCar& ) ;
// Реализация конструкторов CCar
CCar :: CCar ( ) : m_model ( "" ), m_mass ( 0.0 ) { }
CCar :: CCar ( const string& md, const double m )
: m_model ( md ), m_mass ( m ) { }
CCar :: CCar ( const CCar& c )
: m_model ( c.m_model ), m_mass ( c.m_mass ) { }
// Прототипы конструкторов CCargo
CCargo ( ) ;
CCargo ( const string&, const double, const int ) ;
CCargo ( const CCargo& ) ;
// Реализация конструкторов CCargo
CCargo :: CCargo ( ) : CCar ( ), m_carCapacity ( 0 ) { }
CCargo :: CCargo ( const string& md, const double m, const int c )
: CCar ( md, m ), m_carCapacity ( c ) { }
CCargo :: CCargo ( const CCargo& c )
: CCar ( c ), m_carCapacity ( c.m_carCapacity ) { }
// Прототипы конструкторов CTank
CTank ( ) ;
CTank ( const string&, const double, const int, const double ) ;
CTank ( const CTank& ) ;
// Реализация конструкторов CTank
CTank :: CTank ( ) : CCargo ( ), m_volume ( 0.0 ) { }
CTank :: CTank ( const string& md, const double m, const int c,
↳const double v )
: CCargo ( md, m, c ), m_volume ( v ) { }
CTank :: CTank ( const CTank& c )
: CCargo ( c ), m_volume ( c.m_volume ) { }
// Прототипы конструкторов CPlatform
CPlatform ( ) ;
CPlatform ( const string&, const double, const int, const int ) ;
CPlatform ( const CPlatform& ) ;
// Реализация конструкторов CPlatform
CPlatform :: CPlatform ( ) : CCargo ( ), m_boards ( 0 ) { }
```

```
CPlatform :: CPlatform ( const string& md, const double m, const int c,
↳const int b )
: CCargo ( md, m, c ), m_boards ( b ) { }
CPlatform :: CPlatform ( const CPlatform& c )
: CCargo ( c ), m_boards ( c.m_boards ) { }
// Прототипы конструкторов CCarriage
CCarriage ( ) ;
CCarriage ( const string&, const double, const string&, const int ) ;
CCarriage ( const CCarriage& ) ;
// Реализация конструкторов CCarriage
CCarriage :: CCarriage ( ) : CCar ( ), m_type ( "" ), m_places ( 0 ) { }
CCarriage :: CCarriage ( const string& md, const double m,
       ↳const string& t, const int p )
: CCar ( md, m ), m_type ( t ), m_places ( p ) { }
CCarriage :: CCarriage ( const CCarriage& c )
: CCar ( c ), m_type ( c.m_type ), m_places ( c.m_places ) { }
```

Примечание

Описание конструкторов должно быть записано в заголовочном файле своего класса внутри фигурных скобок объявления класса и начинаться со спецификатора доступа `public`. Реализация конструкторов должна находиться в файле реализации своего класса.

Виртуальные функции класса

Виртуальная функция (virtual function) — это функция-член, объявленная в базовом классе и переопределенная в производном. В производном классе эта функция также будет виртуальной. Класс, содержащий хотя бы одну виртуальную функцию, называется полиморфным. Чтобы создать виртуальную функцию, необходимо начать описание функции в базовом классе с ключевого слова `virtual`.

Объявление виртуальной функции в базовом классе определяет способ ее вызова, который нельзя изменять в производных классах. Для виртуальных функций свойственно следующее:

- Виртуальные функции могут принадлежать только классу.
- Виртуальные функции не могут быть статическими.
- Виртуальные функции позволяют создавать различные версии функции для базового и производных классов, однако должны иметь одинаковое

число и одинаковые типы параметров как в базовом, так и в производных классах. В противном случае функция будет перегруженной, а не виртуальной.

- Нужная версия виртуальной функции выбирается на этапе исполнения, а не на этапе компиляции программы.
- Функция может быть объявлена виртуальной в базовом классе, но при этом не иметь переопределения в производном классе. В таком случае для объекта производного класса будет вызываться функция из базового класса.
- Когда виртуальная функция вызывается с указанием области видимости, например, `CCar :: mod ( n )`, то виртуальный механизм не применяется.
- Доступ к виртуальной функции для базового класса происходит через указатель на объект базового класса, а доступ для производного класса — через указатель на объект производного класса или через указатель на базовый класс, который тогда должен содержать адрес объекта производного класса.

Одна важная особенность механизма виртуальных функций заключается в следующем. Пусть в базовом классе объявлена виртуальная функция `func ( )`, и производный класс содержит ее переопределение `func ( )`, внутри которого явно вызывается `CBase :: func ( )`. Пусть в программе объявлен указатель `*pBase` на тип `CBase`, которому присвоен адрес объекта производного класса:

```
pBase = &objDerived.
```

Вызов функции для объекта производного класса `pBase -> func ( )` приведет к вызову версии из производного класса `CDerived :: func ( )`, которая в процессе исполнения вызовет явно функцию `CBase :: func ( )`. Вызовы функции для этого случая показаны на рис. 16.3 пунктирными линиями со стрелками в сторону вызываемой версии. Если указателю `*pBase` присвоить значение адреса объекта базового класса `pBase = &objBase`, то произойдет вызов версии `func ( )` из базового класса (на рис. 16.3 вызов показан сплошной линией со стрелкой на конце).

Если же функция в базовом классе будет объявлена без ключевого слова `virtual`, то вызов для объекта производного класса `pBase -> func ( )` активизирует функцию базового класса, а функция `CDerived :: func ( )` рассматриваться вообще не будет. Объяснение очевидно — базовому классу ничего неизвестно о производных классах.

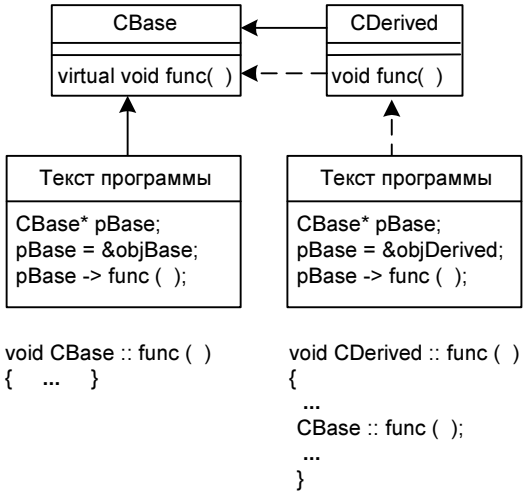


Рис. 16.3. Схема вызова виртуальной функции через указатель на базовый тип

Виртуальный деструктор

Базовый класс не знает о производных классах и объектах. Разрушение объекта базового класса никак не затрагивает объекты производных классов. Отсюда следует, что при работе с объектами производных классов через указатель произойдет ошибка при разрушении объекта производного класса. В этом случае будет разрушен объект базового класса, а объект производного класса нет. Чтобы при исполнении программы такой ошибки не возникало, деструкторы базовых классов всегда следует объявлять виртуальными, то есть в спецификации класса перед именем деструктора необходимо указывать ключевое слово `virtual` (листинг 16.3).

Если базовый класс содержит виртуальный деструктор, то деструктор производного класса тоже будет виртуальным.

Примечание

Деструктор может быть виртуальным, а конструкторы — нет.

Листинг 16.3. Пример объявления деструкторов при наследовании

```
virtual ~CCar ( ) ;
~CCargo ( ) ;
~CTank ( ) ;
~CPlatform ( ) ;
~CCarriage ( ) ;
```

Ключевое слово `virtual` перед деструкторами классов `CCargo` и `CCarriage` указывать не нужно, так как они виртуальны из-за деструктора базового класса `CCar`. Деструкторы `~CTank ( )` и `~CPlatform ( )` являются виртуальными потому, что деструктор их базового класса `CCargo` виртуален.

Далее представлен пример, в котором используются классы с одиночным наследованием. В примере используются известные классы `CCar`, `CCargo`, `CCarriage`, `CPlatform` и `CTank`. Реализация конструкторов этих классов не приводится, так как она показана в разделе о конструкторах при одиночном наследовании. В спецификации всех классов включен оператор с выражением `friend class CMenu`. Ключевое слово `friend` указывает компилятору, что классы разрешают объектам класса `CMenu` получить доступ к своим закрытым и защищенным членам. Все выражение целиком (`friend class CMenu`) объявляет тип `CMenu` дружественным классом (другом). В программах на C++ чаще используются дружественные функции, которые рассматриваются в *главе 17*.

Листинг 16.4. Пример программы с одиночным наследованием

```
// Car.h — Спецификация CCar
#pragma once
#include <string>
using namespace std ;
class CCar
{
protected:
       string m_model ;      // модель
       double m_mass ;       // масса
public:
       CCar ( ) ; CCar ( const string&, const double ) ;
       CCar ( const CCar& ) ; virtual ~CCar ( ) ;
       virtual void mod ( int ) ;   // виртуальная функция
       friend class CMenu ;         // CMenu — друг класса CCar
} ;
// Car.cpp - Реализация CCar
#include "Car.h"
CCar :: ~CCar ( ) { }
void CCar :: mod ( int n )          // виртуальная функция
{      m_mass += n ; }
// Cargo.h — Спецификация CCargo
#pragma once
```

```
#include "Car.h"
class CCargo : public CCar
{
protected:
       int m_carCapacity ;   // грузоподъемность
public:
       CCargo ( ) ; CCargo ( const string&, const double, const int ) ;
       CCargo ( const CCargo& ) ; ~CCargo ( ) ;
       virtual void mod ( int ) ;   // виртуальная функция
       friend class CMenu ;         // CMenu — друг класса CCargo
} ;
// Cargo.cpp - Реализация CCargo
#include "Cargo.h"
CCargo :: ~CCargo ( ) { }
void CCargo :: mod ( int n )        // виртуальная функция
{
       CCar :: mod ( n ) ;          // вызов функции базового класса
       m_carCapacity += n ;
}
// Carriage.h — Спецификация CCarriage
#pragma once
#include "Car.h"
class CCarriage : public CCar
{
       string m_type ;       // тип вагона
       int m_places ;        // число мест
public:
       CCarriage ( ) ;
       CCarriage ( const string&, const double, const string&,
↳const int ) ;
       CCarriage ( const CCarriage& ) ;    ~CCarriage ( ) ;
       void setType ( const string& ) ;    void setPlaces ( const int ) ;
       string getType ( ) const ;          int getPlaces ( ) const ;
       void mod ( int ) ;           // виртуальная функция
       friend class CMenu ;         // CMenu — друг класса CCarriage
} ;
// Carriage.cpp - Реализация CCarriage
#include "Carriage.h"
CCarriage :: ~CCarriage ( ) { }
```

```
void CCarriage :: setType ( const string& t ) { m_type = t ; }
void CCarriage :: setPlaces ( const int p ) { m_places = p ; }
string CCarriage :: getType ( ) const { return m_type ; }
int CCarriage :: getPlaces ( ) const { return m_places ; }
void CCarriage :: mod ( int n )     // виртуальная функция
{
       CCar :: mod ( n ) ;          // вызов функции базового класса
       ++m_places ;
}
// Platform.h — Спецификация CPlatform
#pragma once
#include "Cargo.h"
class CPlatform : public CCargo
{
public:
       int m_boards ;        // количество бортов
       CPlatform ( ) ;
       CPlatform ( const string&, const double, const int, const int ) ;
       CPlatform ( const CPlatform& ) ; ~CPlatform ( ) ;
       void mod ( int ) ;           // виртуальная функция
       friend class CMenu ;         // CMenu — друг класса CPlatform
} ;
// Platform.cpp - реализация CPlatform
#include "Platform.h"
CPlatform :: ~CPlatform ( ) { }
void CPlatform :: mod ( int n )     // виртуальная функция
{      m_boards -= n ;       }
// Tank.h — Спецификация CTank
#pragma once
#include "Cargo.h"
class CTank : public CCargo
{
       double m_volume ;            // объем котла
public:
       CTank ( ) ;
       CTank ( const string&, const double, const int, const double ) ;
       CTank ( const CTank& ) ; ~CTank ( ) ;
       void setVolume ( const double ) ;   double getVolume ( ) const ;
       void mod ( int ) ;           // виртуальная функция
```

```
        friend class CMenu ;          // CMenu — друг класса CTank
} ;
// Tank.cpp - Реализация CTank
#include "Tank.h"
CTank :: ~CTank ( ) { }
void CTank :: setVolume ( const double v ) { m_volume = v ; }
double CTank :: getVolume ( ) const { return m_volume ; }
void CTank :: mod ( int n )           // виртуальная функция
{       m_volume += n ;       }
// Menu.h — Спецификация CMenu
#include "Car.h"
class CMenu
{
public:
        CCar* m_p ;                   // указатель на базовый тип
        CMenu ( ) { }         ~CMenu ( ) { }
        void showMenuAndSelect ( ) ;
private:
        void tank ( ) ; void platform ( ) ; void carriage ( ) ;
} ;
// Menu.cpp — Реализация класса CMenu
#include <iostream>
using namespace std ;
#include "Menu.h"
#include "Tank.h"
#include "Platform.h"
#include "Carriage.h"
// активизация меню
void CMenu :: showMenuAndSelect ( )
{
        char choice ;
        do
        {
                cout << "\n1. Tank\t2. Platform\t3. Carriage\t0. Exit\n" ;
                cout << "Please, your choice -> " ; cin >> choice ;
                switch ( choice )
                {
                case '1': tank ( ) ; break ;
                case '2': platform ( ) ; break ;
```

```
                case '3': carriage ( ) ; break ;
                }
        } while ( choice != '0' ) ;
}
// проверка класса CTank
void CMenu :: tank ( )
{
        cout << "\nTank\n" ;
        CTank* p = new CTank ( "15-1547", 24.5, 68, 85 ) ;
        cout << "Model\t" << p -> m_model << endl ;
        cout << "Mass\t" << p -> m_mass << endl ;
        cout << "Carriage capacity\t" << p -> m_carCapacity << endl ;
        cout << "Volume of the boiler\t" << p -> m_volume << endl ;
        cout << "\nTANK UPDATING\n" ;
        m_p = p ;
        m_p -> mod ( 2 ) ;    // вызов через указатель на базовый тип
        cout << "Volume of the boiler\t" << p -> m_volume << endl ;
        p -> mod ( 2 ) ;      // вызов через указатель на производный тип
        cout << "Volume of the boiler\t" << p -> m_volume << endl ;
        CTank o ( *p ) ; m_p = & o ;
        m_p -> mod ( 4 ) ;    // вызов через указатель на базовый тип
        cout << "Volume of the boiler\t" << o.m_volume << endl ;
        delete p ;
}
// проверка класса CPlatform
void CMenu :: platform ( )
{
        cout << "\nPlatform\n" ;
        CPlatform* p = new CPlatform ( "13-4012", 21.4, 71, 4 ) ;
        cout << "Model\t" << p -> m_model << endl ;
        cout << "Mass\t" << p -> m_mass << endl ;
        cout << "Carriage capacity\t" << p -> m_carCapacity << endl ;
        cout << "Quantity of boards\t" << p -> m_boards << endl ;
        cout << "\nMODIFY PLATFORM\n" ;
        m_p = p ; m_p -> mod ( 2 ) ;
        cout << "Quantity of boards\t" << p -> m_boards << endl ;
        delete p ;
}
// проверка класса CCarriage
```

```
void CMenu :: carriage ( )
{
        cout << "\nCarriage\n" ;
        CCarriage* p = new CCarriage ( "61-807", 55, "Compartment", 36 )
;
        cout << "Model\t" << p -> m_model << endl ;
        cout << "Mass\t" << p -> m_mass << endl ;
        cout << "Type\t" << p -> m_type << endl ;
        cout << "Number of places\t" << p -> m_places << endl ;
        cout << "\nMODIFY CARRIAGE\n" ;
        m_p = p ; m_p -> mod ( 2 ) ;
        cout << "Mass\t" << p -> m_mass << endl ;
        cout << "Number of places\t" << p -> m_places << endl ;
        delete p ;
}
// main.cpp — главная функция
#include "Menu.h"
int main ( )
{
        CMenu* p = new CMenu; p -> showMenuAndSelect ( ) ; delete p ;
        return 0 ;
}
```

В листинге 16.4 проверяется работа методов базовых и производных классов. На рис. 16.4 показаны результаты исполнения программы с одиночным наследованием.

Программа начинается с выполнения функции `main ( )`, где динамически создается объект типа `CMenu`. Через указатель `p` на объект класса `CMenu` вызывается открытый метод `ShowMenuAndSelect ( )`, которому передается управление. Главная функция получает управление обратно, когда завершает работу вызванный метод. Получив управление, `main ( )` заканчивается.

Работа метода `ShowMenuAndSelect ( )` начинается с вывода пунктов меню, каждый из которых может быть активизирован пользователем. Меню программы предлагает проверить работоспособность объектов классов `CTank`, `CPlatform` и `CCarriage`. На рис. 16.4 видно, что сначала активизируется первый пункт меню, следовательно, управление получает метод `tank ( )`.

В теле метода `tank ( )` класса `CMenu` динамически создается объект класса `CTank`, адрес которого записывается в указатель `p` (этот указатель и указатель `p` из главной функции — абсолютно разные указатели, потому что принадле-

жат разным областям видимости). Оператор `new` вызывает конструктор с параметрами класса `CTank` для инициализации объекта. Этот конструктор вызовет конструктор с параметрами класса `CCargo`, который в свою очередь вызовет конструктор с параметрами класса `CCar`. Первым выполнится конструктор класса `CCar`, он выполнит инициализацию `m_model` значением `"15-1547"` и `m_mass` значением `24.5`. После этого конструктор класса `CCargo` выполнит инициализацию `m_carCapacity` значением `68`. Закончит конструирование объекта конструктор класса `CTank`, который выполнит инициализацию `m_volume` значением `85`. Состояние объекта выводится через переменную `p` (указатель на производный тип).

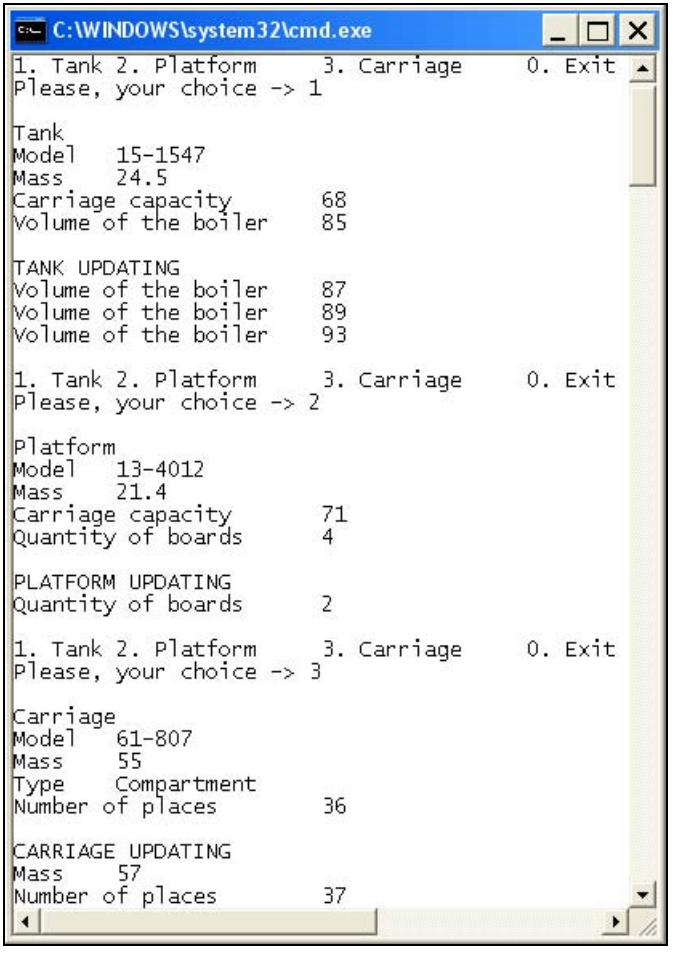


Рис. 16.4. Результаты исполнения программы с одиночным наследованием

Примечание

Нельзя использовать указатель на базовый тип для доступа к членам производных объектов.

Следующим действием функции `tank ( )` является присваивание указателю `m_p`, объявленного в классе `CMenu` в качестве указателя на базовый тип `CCar`, значения указателя на производный тип. Вызов виртуального метода `mod ( )` через указатель на базовый тип по правилам работы механизма виртульных функций приведет к вызову метода `mod ( )` из класса `CTank`. То же самое произойдет и при вызове через указатель на свой тип `p -> mod ( 2 )`. После вывода модифицированных значений члена-данного `m_volume` для копии объекта `o`, разрушается указатель `p`, и метод `tank ( )` заканчивается. Перед возвратом управления сначала будут вызваны деструкторы `~CTank ( )`, `~CCargo ( )`, `~CCar ( )` для объекта с указателем `p`, а затем те же самые деструкторы для объекта `o`.

Действия, похожие на только что рассмотренные, производятся при выборе пользователем пунктов 2 и 3 в меню программы. Эти действия реализуются методами `platform ( )` и `carriage ( )`. Каждый вызов виртуального метода `mod ( )` через указатель на базовый тип настраивается на активизацию версии, находящейся в классе производного объекта. Каждая версия виртуального метода выполняет действия над данными, доступными в своем классе.

Следует обратить внимание на реализацию метода `mod ( )` в классах `CCargo` и `CCarriage`. Если не указать область видимости `CCar ::`, то метод бесконечно будет вызывать сам себя, что приведет к ошибке.

Множественное наследование

Множественное наследование — это иерархическая структура, в которой производный класс имеет два или более базовых классов. При таком наследовании получается новый класс из других классов, которые никак не связаны между собой. Этот класс наследует от базовых классов открытые и защищенные данные и методы, и при необходимости может их дополнить своими собственными. Пример множественного наследования показан на рис. 16.5, где производный класс `CDerived` наследует от базового класса `CBase_1` данные `data_1` и `data_2`, а от базового класса `CBase_2` наследует данное `data_3`.

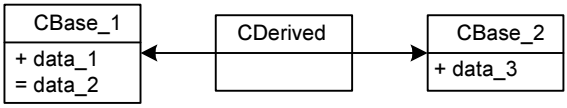


Рис. 16.5. Пример множественного наследования

Объявление классов при множественном наследовании

Объявление базового класса производится обычным способом. Объявление производного класса имеет следующий формат:

```
class имя_производного_класса :
       [ public | protected | private ]opt имя_базового_класса1,
       [ public | protected | private ]opt имя_базового_класса2,
       …,
       [ public | protected | private ] opt имя_базового_классаN,
{ описание членов производного класса } ;
```

где `opt` — обозначает, что в объявлении может быть указан один из спецификаторов, перечисленных в квадратных скобках через символ |.

После имени производного класса ставится двоеточие, за которым следует список указанных через запятую имен базовых классов, каждое из которых предваряется ключевым словом `public`, `protected` или `private`, которое будет определять доступ к данным базового класса внутри производного.

Виртуальный базовый класс

При множественном наследовании может возникать несколько копий унаследованных данных. На рис. 16.6 проиллюстрирована ситуация, когда в производном классе `CDerived` появляется две копии базового класса `Cbase` и, как следствие, две копии унаследованных данных этого класса.

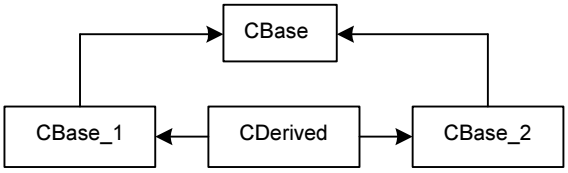


Рис. 16.6. Пример копирования унаследованных данных

Для преодоления такого типа неоднозначности при множественном наследовании используются виртуальные базовые классы. Появление более чем двух копий базового класса в производном классе можно предотвратить, если наследовать базовый класс как виртуальный. Такое наследование не дает появиться двум или более копиям базового класса в любом следующем производном классе, косвенно наследующем базовый класс. В объявлении производного класса перед спецификатором доступа базового класса необходимо поставить

ключевое слово `virtual`. Например, чтобы в классе `CDerived` не было копий базового класса `CBase`, необходимо объявить классы следующим образом:

```
class CBase { … } ;
class CBase_1 : virtual public CBase { … } ;
class CBase_2 : virtual public CBase { … } ;
class CDerived : public CBase_1, public CBase_2 { … } ;
```

При таком объявлении класс `CDerived` имеет один наследуемый подобъект класса `CBase`, который является общим для классов `CBase_1` и `CBase_2`.

Конструкторы и деструкторы при множественном наследовании

Конструкторы базовых классов объявляются и реализуются как обычно. Конструктор производного класса имеет следующий формат:

```
        Конструктор_производного_класса ( Список_формальных_параметров )
:
                имя_базового_класса1 ( Список_параметров1 ),
                имя_базового_класса2 ( Список_параметров2 ),
                …,
                имя_базового_классаN ( Список_параметровN ),
        { // Тело конструктора производного класса } ;
```

где `имя_базового_классаi ( Список_параметровi )` в списке инициализации обозначает явный вызов конструктора соответствующего базового класса.

Конструктору производного класса, помимо собственных параметров, следует передать все параметры, необходимые конструкторам всех базовых классов. В списке инициализации конструктора производного класса явно вызываются конструкторы базовых, которым передаются требуемые ими значения параметров. При создании объекта производного класса конструкторы вызываются слева направо в порядке записи базовых классов при объявлении производного класса.

Конструктор базового класса, объявленного с ключевым словом `virtual`, не инициализирует подобъект при конструировании объекта производного класса. Об инициализации данных, которые наследуются от такого базового класса, автор программы должен позаботиться самостоятельно.

Деструкторы базовых классов следует объявлять виртуальными. Вызываются деструкторы в порядке, обратном вызову конструкторов.

В листинге рассматривается пример программы с множественным и одиночным наследованием. Модель классов, реализованных в программе, показана на рис. 16.7.

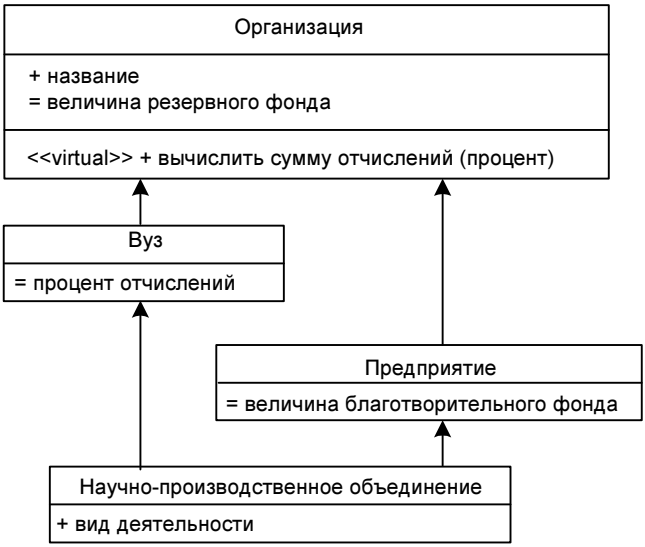


Рис. 16.7. Модель классов программы с множественным и одиночным наследованием

Листинг 16.5. Пример программы с множественным и одиночным наследованием

```
// Organization.h - Спецификация COrganization (Организация)
#include <string>
using namespace std ;
#pragma once
class COrganization
{
protected:
       double m_fund ;              // величина резервного фонда
public:
       string m_name ;              // название организации
       COrganization ( ) ;
       COrganization ( const string&, const double ) ;
       virtual ~COrganization ( ) ;
       void setFund ( double ) ; double getFund ( ) const ;
       // вычисляет сумму ежемесячных отчислений
       virtual double calcPayment ( const int = 0 ) const ;
} ;
// Organization.cpp - Реализация COrganization
```

```
#include "Organization.h"
COrganization :: COrganization ( ) : m_name ( "" ), m_fund ( 0.0 ) { }
COrganization :: COrganization ( const string& n, const double f )
: m_name ( n ), m_fund ( f ) { }
COrganization :: ~COrganization ( ) { }
void COrganization :: setFund ( double f ) { m_fund = f ; }
double COrganization :: getFund ( ) const { return m_fund ; }
double COrganization :: calcPayment ( const int p ) const
{       return ( m_fund * p / 1200 ) ;      }
// HighSchool.h - Спецификация CHighSchool (Вуз)
#pragma once
#include "Organization.h"
class CHighSchool : virtual public COrganization
{
protected:
        int m_rate ;                    // % отчислений из резервного фонда
public:
        CHighSchool ( ) ;
        CHighSchool ( const string&, const double, const int ) ;
        ~CHighSchool ( ) ;
        void setRate ( int ) ; int getRate ( ) const ;
        double calcPayment ( const int = 0 ) const ;
} ;
// HighSchool.cpp - Реализация CHighSchool
#include "HighSchool.h"
CHighSchool :: CHighSchool ( ) : COrganization ( ), m_rate ( 0 ) { }
CHighSchool ::
ChighSchool ( const string& n, const double f, const int r )
: COrganization ( n, f ), m_rate ( r ) { }
CHighSchool :: ~CHighSchool ( ) { }
void CHighSchool :: setRate ( int r ) { m_rate = r ; }
int CHighSchool :: getRate ( ) const { return m_rate ; }
double CHighSchool :: calcPayment ( const int p ) const
{
        if ( p > m_rate ) return ( m_fund * m_rate / 1200 ) ;
        else return ( COrganization :: calcPayment ( p ) ) ;
}
// Enterprise.h - Спецификация CEnterprise (Предприятие)
#pragma once
```

```
#include "Organization.h"
class CEnterprise : virtual public COrganization
{
protected:
       double m_alloc ;             // величина благотворительного фонда
public:
       CEnterprise ( ) ;
       CEnterprise ( const string&, const double, const double ) ;
       virtual ~CEnterprise ( ) ;
       void setAlloc ( const double ) ; double getAlloc ( ) const ;
       double calcPayment ( const int = 0 ) const ;
} ;
// Enterprise.cpp - Реализация CEnterprise
#include "Enterprise.h"
CEnterprise :: CEnterprise ( ) : COrganization ( ), m_alloc ( 0.0 ) { }
CEnterprise ::
CEnterprise ( const string& n, const double f, const double a )
: COrganization ( n, f ), m_alloc ( a ) { }
CEnterprise :: ~CEnterprise ( ) { }
void CEnterprise :: setAlloc ( const double a )
{      m_alloc = a ; }
double CEnterprise :: getAlloc ( ) const { return m_alloc ; }
double CEnterprise :: calcPayment ( const int p ) const
{      return ( m_alloc * p / 1200 ) ;     }
// Union.h - Спецификация CUnion (Научно-производственное объединение)
#pragma once
#include <string>
using namespace std ;
#include "HighSchool.h"
#include "Enterprise.h"
class CUnion : public CHighSchool, public CEnterprise
{
public:
       string m_kind ;              // вид деятельности
       CUnion ( ) ;
       CUnion ( const string&, const double, const string&, const int,
↳const double ) ;
       ~CUnion ( ) ;
       double calcPayment ( const int = 0 ) const ;
```

```
} ;
// Union.cpp - Реализация CUnion
#include "Union.h"
CUnion :: CUnion ( )
: m_kind ( "" ), CHighSchool ( ), CEnterprise ( ) { }
CUnion :: CUnion ( const string& n, const double f, const string& k,
↳const int r, const double a )
: m_kind ( k ), CHighSchool ( "", 0.0, r ), CEnterprise ( "", 0.0, a )
{       m_name = n ; setFund ( f ) ;         }
CUnion :: ~CUnion ( ) { }
double CUnion :: calcPayment ( const int ) const { return 0.0 ; }
// main.cpp — главная функция
#include <iostream>
using namespace std ;
#include "Union.h"
int main ( )
{
        CHighSchool* pH =
                new CHighSchool ( "UNIVERSITY OF FANS C++", 1.0e6, 2 ) ;
        cout << "\nName\t" << pH -> m_name << endl ;
        cout << "Fund\t" << pH -> getFund ( ) << endl ;
        cout << "Rate\t" << pH -> getRate ( ) << endl ;
        CEnterprise* pE =
                new CEnterprise ( "CLEVER OWL Ltd.", 1.0e8, 1.0e7 ) ;
        cout << "\nName\t" << pE -> m_name << endl ;
        cout << "Fund\t" << pE -> getFund ( ) << endl ;
        cout << "Allocation\t" << pE -> getAlloc ( ) << endl << endl ;
        CUnion* pU = new CUnion ( "VICTORY", 1.25e8, "Miracles",
                ↳pH -> getRate ( ), pE -> getAlloc ( ) ) ;
        cout << "\nName\t" << pU -> m_name << endl ;
        cout << "Kind\t" << pU -> m_kind << endl ;
        cout << "Fund\t" << pU -> getFund ( ) << endl ;
        cout << "Rate\t" << pU -> getRate ( ) << endl ;
        cout << "Allocation\t" << pU -> getAlloc ( ) << endl ;
        cout << "\nPayment\n" ;
        COrganization* p ;
        p = pH ;
        cout << pH-> m_name << "\t= " << p-> calcPayment ( 5 ) << endl ;
```

```
        p = pE ;
        cout << pE-> m_name << "\t= " << p-> calcPayment ( 17 ) << endl ;
        p = pU ;
        cout << pU-> m_name << "\t\t= " << p-> calcPayment ( ) << endl ;
}
```

В листинге 16.5 производные классы `CEnterprise` и `CHighSchool` виртуально наследуют от базового класса `COrganization`, чтобы в производном классе `CUnion` не было копий подобъектов базового класса `COrganization`. Деструктор этого класса объявлен виртуальным, чтобы при разрушении объектов производных классов вызывались их деструкторы. Метод `calcPayment ( )`, который вычисляет сумму ежемесячных отчислений в зависимости от указанного параметром `p` процента, объявлен как виртуальный. Этот метод переопределен во всех производных классах, и для каждого класса выполняет свои расчеты.

Следует обратить особое внимание на конструктор с параметрами производного класса `Cunion`. В списке параметров указаны параметры всех классов, стоящих выше по иерархии. Список инициализации начинается с инициализации члена-данного `m_kind`. Далее указаны вызовы конструкторов с параметрами классов `CHighSchool` и `CEnterprise`. Два первых параметра для этих конструкторов не имеют никакого значения, так как наследование виртуальное, и никакой инициализации для объекта производного класса `CUnion` автоматически производиться не будет. Чтобы данные `m_name` и `m_funds` получили значения в результате конструирования объекта, в теле конструктора записаны соответствующие операторы. Для инициализации `m_funds` вызывается метод базового класса `setFund ( )`.

В главной функции через указатели `pH`, `pE`, `pU` динамически создаются объекты производных классов. При помощи этих указателей происходит обращение к открытым данным для вывода их значения и к открытым методам для получения значений защищенных данных. В конце главной функции показано, как организовать обращение к виртуальному методу через указатель на базовый тип. Однако нельзя использовать этот указатель для доступа к личным данным производного класса. Базовый класс, как упоминалось ранее, ничего не знает о производных классах. Через указатель на базовый тип можно получить доступ только к той части производного класса, которая унаследована от базового. В программе примера можно было использовать указатель `p` для доступа к `m_name`, так как объявление этого члена-данного расположено в базовом классе. Результаты исполнения программы представлены на рис. 16.8.

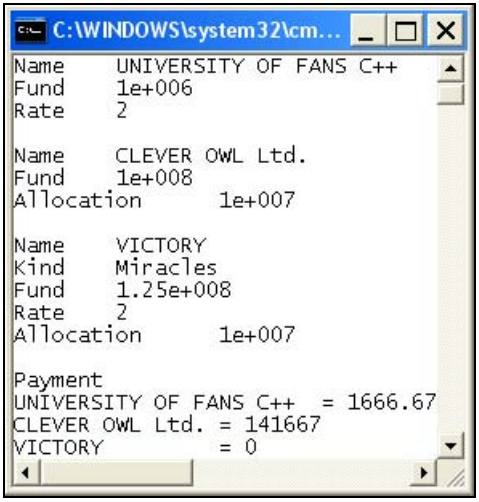


Рис. 16.8. Результаты исполнения программы с множественным и одиночным наследованием

Чистые виртуальные функции и абстрактные классы

Чистая виртуальная функция (pure virtual function) — это виртуальная функция, которая не имеет реализации в базовом классе. Чистая виртуальная функция представляет собой общий прототип для определений функций, которые располагаются в производных классах.

Класс, в котором объявлена хотя бы одна чистая виртуальная функция, называется абстрактным (abstract class). Для таких классов невозможно создать объекты. Абстрактные классы могут использоваться только в качестве базовых для других классов. Если в производном классе, который наследует от абстрактного базового класса, отсутствует реализация чистой виртуальной функции, то этот производный класс тоже будет являться абстрактным.

Примечание

Объекты конструируются только для классов, в которых чистая виртуальная функция имеет реализацию.

Объявление прототипа чистой виртуальной функции в базовом классе имеет следующий формат:

```
virtual тип_функции имя_функции ( список_параметров_функции ) = 0 ;
```

В производных классах прототип указывается как обычно, однако он не может быть изменен. Это значит, что тип возвращаемого значения (тип функции), типы и количество ее параметров должны строго соответствовать описанию, которое задано в базовом классе (листинг 16.6).

Листинг 16.6. Пример использования чистой виртуальной функции

```
// Specifications.h - Спецификации классов
// спецификация CFigure — геометрическая фигура (базовый класс)
class CFigure
{
protected:
       double m_a, m_b ;            // две стороны фигуры
public:
       CFigure ( ) ; CFigure ( const double, const double ) ;
       virtual ~CFigure ( ) ;
       // виртуальные функции
       virtual void setFigure ( double, double ) ;
       virtual void getFigure ( double&, double& ) const ;
       // чистая виртуальная функция - вычисление площади фигуры
       virtual double getArea ( ) const = 0 ;
} ;
// спецификация CTriangle — треугольник (производный класс)
class CTriangle : public CFigure
{
protected:
       double m_c ;                 // третья сторона
public:
       CTriangle ( ) ; CTriangle ( double, double, double ) ;
       ~CTriangle ( ) ;
       void setFigure ( double, double, double ) ;
       void getFigure ( double&, double&, double& ) const ;
       double getArea ( ) const ;   // вычисление площади треугольника
} ;
// спецификация CRectangular — прямоугольник (производный класс)
class CRectangular : public CFigure
{
public:
       CRectangular ( ) ; CRectangular ( double, double ) ;
       ~CRectangular ( ) ;
```

```
        double getArea ( ) const ;   // вычисление площади прямоугольника
} ;
// спецификация CParallelogram — параллелограмм (производный класс)
class CParallelogram : public CTriangle
{
public:
        CParallelogram ( ) ; CParallelogram ( double, double, double ) ;
        ~CParallelogram ( ) ;
        double getArea ( ) const ;   // вычисление площади параллелограмма
} ;
// спецификация CTrapeze — трапеция (производный класс)
class CTrapeze : public CTriangle
{
protected:
        double m_d, m_h ;            // четвертая сторона и высота
public:
        CTrapeze ( ) ;
        CTrapeze ( double, double, double, double, double ) ;
        ~CTrapeze ( ) ;
        void setFigure ( double, double, double, double, double ) ;
        void getFigure ( double&, double&, double&, double&, double& )
↳const ;
        double getArea ( ) const ;   // вычисление площади трапеции
} ;
// Realizations.cpp - Реализации классов
#pragma once
#include "Specifications.h"
// реализация CFigure — геометрическая фигура (базовый класс)
CFigure :: CFigure ( ) : m_a ( 0.0 ), m_b ( 0.0 ) { }
CFigure :: CFigure ( double a, double b ) : m_a ( a ), m_b ( b ) { }
CFigure :: ~CFigure ( ) { }
void CFigure :: setFigure ( double a, double b ) { m_a = a ; m_b = b ; }
void CFigure :: getFigure ( double& a, double& b ) const
{       a = m_a ; b = m_b ;   }
// чистая виртуальная функция не имеет реализации в базовом классе
// реализация CTriangle — треугольник (производный класс)
CTriangle :: CTriangle ( ) : CFigure ( ), m_c ( 0.0 ) { }
CTriangle :: CTriangle ( double a, double b, double c )
: CFigure ( a, b ), m_c ( c ) { }
CTriangle :: ~CTriangle ( ) { }
```

```
void CTriangle :: setFigure ( double a, double b, double c )
{       CFigure :: setFigure ( a, b ) ; m_c = c ;  }
void CTriangle :: getFigure ( double& a, double& b, double& c ) const
{       CFigure :: getFigure ( a, b ) ; c = m_c ;  }
// реализация чистой виртуальной функции
// площадь треугольника вычисляется по формуле Герона
#include <cmath>
using namespace std ;
double CTriangle :: getArea ( ) const
{
        double p = ( m_a + m_b + m_c ) / 2 ;
        return sqrt ( p * ( p - m_a ) * ( p - m_b ) * ( p - m_c ) ) ;
}
// реализация CRectangular — прямоугольник (производный класс)
CRectangular :: CRectangular ( ) : CFigure ( ) { }
CRectangular :: CRectangular ( double a, double b )
: CFigure ( a, b ) { }
CRectangular :: ~CRectangular ( ) { }
// реализация чистой виртуальной функции
double CRectangular :: getArea ( ) const { return m_a * m_b ; }
// реализация CParallelogram — параллелограмм (производный класс)
// член-данное m_c базового класса CTriangle — это высота параллелограмма
CParallelogram :: CParallelogram ( ) : CTriangle ( ) { }
CParallelogram :: CParallelogram ( double a, double b, double h )
: CTriangle ( a, b, h ) { }
CParallelogram :: ~CParallelogram ( ) { }
// реализация чистой виртуальной функции
double CParallelogram :: getArea ( ) const { return m_a * m_c ; }
// реализация CTrapeze — трапеция (производный класс)
CTrapeze :: CTrapeze ( ) : CTriangle ( ), m_d ( 0.0 ), m_h ( 0.0 ) { }
CTrapeze :: CTrapeze ( double a, double b, double c, double d, double h )
: CTriangle ( a, b, c ), m_d ( c ), m_h ( h ) { }
CTrapeze :: ~CTrapeze ( ) { }
void CTrapeze ::
setFigure ( double a, double b, double c, double d, double h )
{       CTriangle :: setFigure ( a, b, c ) ; m_d = d ; m_h = h ; }
void CTrapeze ::
getFigure ( double& a, double& b, double& c, double& d, double& h ) const
{       CTriangle :: getFigure ( a, b, c ) ; d = m_d ; h = m_h ; }
// реализация чистой виртуальной функции
```

```
double CTrapeze :: getArea ( ) const
{       return 0.5 * ( m_a + m_b ) * m_h ;  }
// main.cpp - главная функция
#include <iostream>
using namespace std ;
#include "Specifications.h"
void view ( double, double ) ;
void view ( double, double, double ) ;
void view ( double, double, double, double, double ) ;
int main ( )
{
        // объявление массива указателей на базовый класс
        CFigure* pF [ 4 ] ;
        // объявление массива указателей с инициализацией названиями фигур
        char* name [ ] =
                { "Triangle", "Trapeze", "Parallelogram", "Rectangular" } ;
        // создание объектов разных типов
        CTriangle triangle ( 3.0, 4.0, 2.0 ) ;
        CTrapeze trapeze ( 5.0, 2.0, 1.9, 2.8, 1.8 ) ;
        CParallelogram parallelogram ( 15.2, 28.75, 18.3 ) ;
        CRectangular rectangular ( 100.5, 24.13 ) ;
        // инициализация массива pF адресами созданных объектов
        pF [ 0 ] = &triangle ;       pF [ 1 ] = &trapeze ;
        pF [ 2 ] = &parallelogram ;  pF [ 3 ] = &rectangular ;
        // цикл вывода состояния созданных объектов и их площадей
        double a, b, c, d, h ;
        for ( int i = 0; i < 4; i++ )
        {
                cout << '\n' << name [ i ] << " ( ";
                switch ( i )
                {
                case 0:
                        triangle.getFigure ( a, b, c ) ;
                        view ( a, b, c ) ; cout << " )\t\t\t" ; break ;
                case 1:
                        trapeze.getFigure ( a, b, c, d, h ) ;
                        view ( a, b, c, d, h ) ; cout << " )\t\t" ; break ;
                case 2:
                        parallelogram.getFigure ( a, b, h ) ;
                        view ( a, b, h ); cout << " )\t" ; break ;
```

```
            case 3:
                  rectangular.getFigure ( a, b ) ;
                  view ( a, b ) ; cout << " )\t\t" ; break ;
            }
            cout << "AREA = " << pF [ i ] -> getArea ( ) << endl ;
      }
      // цикл изменения данных объектов, вывода их состояния и площадей
      for ( int i = 0; i < 4; i++ )
      {
            cout << '\n' << name [ i ] << " ( ";
            switch ( i )
            {
            case 0:
                  triangle.setFigure ( 3, 3, 3 ) ;
                  triangle.getFigure ( a, b, c ) ;
                  view ( a, b, c ) ; cout << " )\t\t\t" ; break ;
            case 1:
                  trapeze.setFigure ( 5, 2, 1.8, 1.8, 1.9 ) ;
                  trapeze.getFigure ( a, b, c, d, h ) ;
                  view ( a, b, c, d, h ) ; cout << " )\t\t" ; break ;
            case 2:
                  parallelogram.setFigure ( 14, 15, 14 ) ;
                  parallelogram.getFigure ( a, b, h ) ;
                  view ( a, b, h ); cout << " )\t\t" ; break ;
            case 3:
                  rectangular.setFigure ( 1005, 2413 ) ;
                  rectangular.getFigure ( a, b ) ;
                  view ( a, b ) ; cout << " )\t\t" ; break ;
            }
            cout << "AREA = " << pF [ i ] -> getArea ( ) << endl ;
      }
            return 0 ;
}
// определение перегруженной функции вывода переменных
void view ( double a, double b )
{     cout << a << ", " << b ;     }
void view ( double a, double b, double c )
{     view ( a, b ) ; cout << ", " << c ; }
void view ( double a, double b, double c, double d, double h )
{     view ( a, b, c ) ; cout << ", " << d << ", " << h ;      }
```

В программе используется абстрактный базовый класс `CFigure`, который описывает геометрические фигуры и содержит чистую виртуальную функцию `getArea ( )`, предназначенную для вычисления площади геометрической фигуры. Эта функция реализована во всех производных классах, каждый из которых ориентирован на работу с разными геометрическими фигурами.

Реализация классов программы достаточна проста. Следует обратить внимание на виртуальные методы классов, названия которых начинаются с префикса `get`. Методы имеют тип `void`. Параметры методов задаются ссылками на тип с плавающей точкой. Это значит, что в вызове метода должны быть указаны константные адреса переменных, которым необходимо присвоить значения, равные значениям членов-данных объекта класса. В вызове этих методов невозможно указать константные значения.

Программа реализует перегруженную функцию `view ( )` для вывода разного числа переменных с плавающей точкой. Функция используется для вывода состояния объектов производных классов.

В главной функции создаются и модифицируются объекты классов `CTriangle`, `CTrapeze`, `CParallelogram`, `CRectangular`. Для класса `CFigure` ни один объект не может быть создан, поскольку он является абстрактным. Вызовы виртуальных методов для установки и получения значений данных объектов производных классов осуществляются по именам этих объектов. Вызов чистого виртуального метода происходит через указатель на базовый тип `pF [ i ]`, в котором записан адрес производного объекта. На рис. 16.9 показаны результаты исполнения программы с чистой виртуальной функцией.

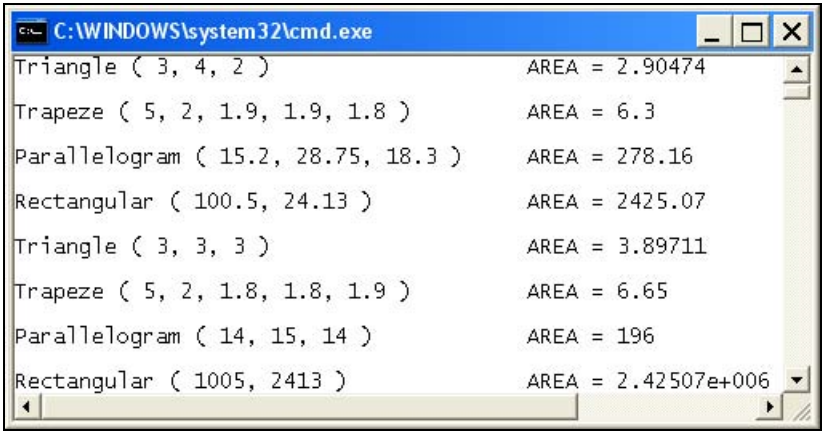
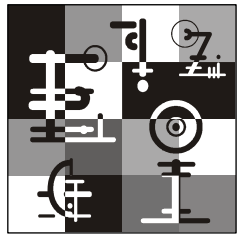


Рис. 16.9. Результаты исполнения программы с чистой виртуальной функцией



Глава 17

Перегрузка операторов

Перегрузка операторов (operator overloading) предоставляет возможность переопределить практически любой стандартный оператор C++ относительно типа данных `class` (класс). К операторам языка, которые запрещено перегружать, относятся следующие:

- оператор выбора члена класса `.` ;
- оператор селектора члена класса `.*` ;
- оператор разрешения области видимости `::` ;
- тернарный оператор `? :` ;
- статический оператор вычисления длины оператора в байтах `sizeof` .

Кроме перечисленных операторов, относящихся к синтаксису языка C++, не разрешается переопределять следующие операторы препроцессора:

- оператор превращения в строку `#` ;
- оператор конкатенации `##` .

Примечание

Операторы `#` и `##` используются в директиве препроцессора `#define`, которую современный стандарт C++ применять не рекомендует. Однако следует помнить о невозможности их перегрузки для классов.

Когда оператор перегружается, его стандартное предназначение для типов данных, отличных от типа `class`, сохраняется. В то же самое время оператор приобретает дополнительный смысл, связанный с тем классом, для которого он определен. Следовательно, перегрузка оператора расширяет набор типов, для которых он может применяться.

Перегрузка операторов осуществляется при помощи операторных функций (operator function), которые могут быть функциями-членами (принадлежать

классу) или функциями-друзьями (не принадлежать классу). В каждом случае операторная функция объявляется по-своему. В разделах главы детально рассматриваются оба случая. Далее перечислены общие правила перегрузки операторов, не зависящие от принадлежности операторной функции классу:

- Нельзя создать новый знак оператора, можно использовать лишь существующие.
- Невозможно изменить приоритет операторов.
- Нельзя изменять количество операндов в операторе. Для унарных операций (unary operation) операндом должен быть класс, для бинарных операций (binary operation) хотя бы один операнд должен быть классом.
- Оператор индексации `[ ]` рассматривается как бинарный оператор.
- Оператор вызова функции `( )` рассматривается как бинарный оператор.
- Оператор доступа к члену класса через указатель на объект этого класса `->` рассматривается как унарный оператор.
- Оператор присваивания `=` не наследуется в производных классах, но может быть повторно определен в них.
- Операторная функция, которая переопределяет постфиксную форму записи операторов инкремента и декремента, использует фиктивный параметр типа `int` как признак отличия от префиксной формы записи этих операторов.
- Идентичность оператора инкремент `++` с оператором сложения с замещением `+=` теряется, если она не задана переопределением этих операторов для класса. То же относится к операторам декремент `--` и вычитание с замещением `-=`.
- Операторная функция не может иметь параметров, заданных по умолчанию.

Примечание

Операторы индексации `[ ]`, присваивания `=`, вызова `( )` и доступа к члену класса через указатель `->` перегружаются только при помощи нестатической операторной функции-члена класса.

Операторные функции-члены класса

Операторная функция-член класса для перегрузки унарных операторов имеет следующий формат:

```
тип_функции имя_класса :: operator знак_оператора ( )
{
        // выполняемые операции
}
```

Для перегрузки бинарных операторов используется следующий формат:

```
тип_функции имя_класса :: operator знак_оператора ( описание_параметра )
{
        // выполняемые операции
}
```

Разница в форматах объясняется тем, что операнд, который стоит в левой части оператора, передается операторной функции неявно с помощью указателя `this`. Поэтому для унарных операторов список параметров функции будет пустым, а для бинарных операторов операнд, стоящий в правой части оператора, будет указан в списке параметров функции.

Как правило, операторная функция возвращает объект класса, с которым она работает. Несмотря на то, что предназначение перегружаемого оператора можно определять произвольно, следует придерживаться его стандартного предназначения и традиционных действий. Так, стандартные операторы (кроме операторов с замещением) не меняют значений своих операндов, поэтому и перегруженным операторам не стоит их модифицировать.

Перегрузка оператора ->

Оператор доступа к члену класса через указатель на объект этого класса `->`, который называют также оператором ссылки на член класса (class member access operator), при перегрузке считается унарным. Операторная функция, которая перегружает этот оператор, вызывается из объекта и должна возвращать указатель на объект класса, для которого она определена.

Для обычных указателей действие оператора доступа `->` можно выразить при помощи операторов разыменовывания `*` и индексации `[ ]`. Например, если `p` является указателем, то выполнение следующих операторов приводит к одинаковому результату:

```
p -> a ;              // -> — обычный оператор
( *p ) . a ;          // *p — объект
p [ 0 ] . a ; // p [ 0 ] — объект
```

При перегрузке оператора `->` подобная эквивалентность будет невозможна, потому что две последние строки при перегруженном операторе `->` даже не будут компилироваться. В связи с этим все указанные операторы для класса следует переопределить.

Листинг 17.1. Пример программы эквивалентной перегрузки оператора ->

```
#include <iostream>
#include <string>
using namespace std ;
```

```
class C
{
        string m_s ;
public:
        C ( ) : m_s ( "" ) { }
        C ( const string& s ) : m_s ( s ) { }
        ~ C ( ) { }
        string getS ( ) const { return m_s ; } ;
        C* operator -> ( ) { return this ; }
        C& operator * ( ) { return *this ; }
        C& operator [ ] ( const unsigned int i ) { return this [ i ] ; }
} ;
int main ( )
{
        C o ( "class member access operator" ) ;
        cout << "Overloaded operator ->\t" << o -> getS ( ) << endl ;
        cout << "Overloaded operator *\t" << ( *o ).getS ( ) << endl ;
        cout << "Overloaded operator [ ]\t" << o [ 0 ].getS ( ) << endl ;
        cout << "Standard operator .\t" << o.getS ( ) << endl ;
        return 0 ;
}
```

В листинге 17.1 для класса `C` перегружаются три оператора: оператор ссылки на член класса `->`, оператор разыменовывания `*` и оператор индексации `[ ]`. Два первых оператора унарные, поэтому список операторных функций для них пуст. Оператору индексации в качестве параметра передается целочисленная переменная без знака, которая определяет индекс. Так как перегруженный оператор `->` должен возвращать адрес, а перегруженные операторы `*` и `[ ]` должны возвращать объект, типами операторных функций являются `C*` и `C&` соответственно. Результаты исполнения программы представлены на рис. 17.1.

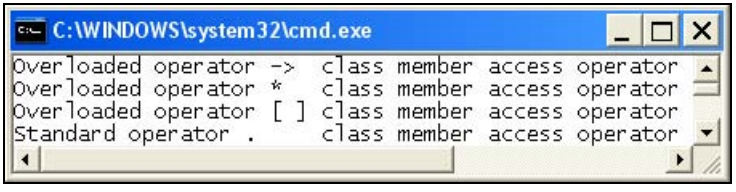


Рис. 17.1. Результаты исполнения программы эквивалентной перегрузки `->`

Перегрузка оператора *[ ]* для массива

Оператор индексации [ ] по правилам перегрузки считается бинарным. Из этого следует, что операторная функция должна иметь параметр, который задает индекс.

Чтобы использование оператора [ ] сделать возможным в левой и в правой частях оператора присваивания для одномерных массивов, операторная функция должна возвращать ссылку на элемент массива. В этом случае можно будет как передавать значение индекса в класс, так и получать значение элемента массива класса с заданным индексом.

Для двумерных массивов оператор перегружается только для первого индекса, причем в этом случае он должен возвращать адрес строки с заданным индексом. Благодаря такой перегрузке второй индекс будет использоваться так же, как и в стандартном операторе индексации.

Примечание

При помощи перегруженного оператора [ ] можно контролировать выход индекса массива за пределы допустимого диапазона, но лучше это делать при помощи механизма обработки исключений.

Листинг 17.2. Пример программы с перегрузкой оператора [ ] для одномерного массива

```
class C
{
public:
       int* m_p ; int m_n ;
       C ( int n = 10 ) : m_n ( n )
       {
              m_p = new int [ m_n ] ;
              for ( int i = 0; i < m_n; i++ ) m_p [ i ] = 0 ;
       }
       ~ C ( ) { delete [ ] m_p ; }
       int& operator [ ] ( const unsigned int i ) const
       { return m_p [ i ] ; }
} ;
#include <iostream>
using namespace std ;
int main ( )
{
```

```
        int arr [ ] = { 1, -2, -3, -4, 5 } ; C o ( 5 ) ;
        for ( int i = 0; i < 5; i++ )
                if ( arr [ i ] < 0 )
                        // оператор [ ] слева от оператора =
                        o [ i ] = -arr [ i ] ;
                else    o [ i ] = arr [ i ] ;
        // оператор [ ] справа от оператора =
        int x ; x = o [ 2 ] ;
        cout << x << endl ;                          // выводится 3
        for ( int i = 0; i < 5; i++ )                // выводится 1 2 3 4 5
                cout << o [ i ] << '\t' ;
        cout << endl ;
        return 0 ;
}
```

В листинге 17.2 для класса `C` перегружается оператор `[ ]`, который определен при помощи операторной функции-члена с типом возвращаемого значения ссылки на элемент массива.

В функции `main ( )` объявляются целочисленный массив `arr` и объект `o` класса `C`. Объект создается конструктором с параметрами, в котором выделяется память для пяти элементов, после чего им присваиваются нулевые значения. Перегрузка индексации позволяет использовать знак `[ ]` в левой части оператора присваивания в цикле `for`, где происходит присваивание значений элементам массива в объекте `o`. Использование знака `[ ]` в правой части оператора присваивания присутствует в выражении `x = o [ 2 ]`. Если в классе перегружен оператор `[ ]`, имя массива при обращении к элементам массива не указывается.

Листинг 17.3. Пример программы с перегрузкой оператора `[ ]` для двумерного массива

```
class C
{
public:
        int** m_p ; int m_m, m_n ;
        C ( int m = 10, int n = 10 ) : m_m ( m ), m_n ( n )
        {
                m_p = new int* [ m_m ] ;
                for ( int i = 0; i < m_m; i++ )
                        m_p [ i ] = new int [ m_n ] ;
```

```
                for ( int i = 0; i < m_m; i++ )
                        for ( int j = 0; j < m_n; j++ )
                                m_p [ i ] [ j ] = 0 ;
        }
        ~ C ( )
        {
                for ( int i = 0; i < m_m; i++ ) delete [ ] m_p [ i ] ;
                delete [ ] m_p ;
        }
        int* operator [ ] ( const unsigned int i ) const
        { return m_p [ i ] ; }
} ;
#include <iostream>
using namespace std ;
int main ( )
{
        int arr [ ] [ 2 ] = { 1, -2, -3, -4, 5 } ; C o ( 3, 2 ) ;
        // присваивание значений элементам матрицы в объекте
        for ( int i = 0; i < 3; i++ )
                for ( int j = 0; j < 2; j++ )
                        o [ i ] [ j ] = -arr [ i ] [ j ] ;
        // вывод состояния объекта
        for ( int i = 0; i < 3; i++ )
        {
                for ( int j = 0; j < 2; j++ )
                        cout << o [ i ] [ j ] << '\t' ;
                cout << endl ;
        }
        // использование [ ] в правой части оператора присваивания
        int x = o [ 2 ] [ 0 ] ;
        cout << x << endl ;
        return 0 ;
}
```

В листинге 17.3 перегруженный оператор `[ ]` получает в качестве параметра индекс строки матрицы и возвращает адрес строки с этим индексом. Обращение

```
o [ i ] [ j ]
```

вызывает операторную функцию-член и, получая от нее адрес `i`-ой строки, производит индексирование по индексу `j` стандартным способом. В результате выполнения программы выводится матрица

```
-1      2
3       4
-5      0
```

и первый элемент третьей строки матрицы `-5`.

Перегрузка оператора =

Для оператора `=` компилятор поддерживает скрытую функцию по умолчанию, которая копирует данные объекта, стоящего справа от знака `=`, в объект, стоящий слева. Такое исполнение оператора неверно для классов, где используется динамическое выделение памяти, поэтому для них оператор `=` всегда следует перегружать. В определении операторной функции-члена необходимо учитывать, что присваивание (как и арифметические операторы с замещением) модифицируют существующее значение операнда, который стоит слева от знака оператора (листинг 17.4).

Листинг 17.4. Пример программы с перегрузкой оператора =

```
class C
{
public:
       int* m_p ; int m_n ;
       C ( int n = 10 ) : m_n ( n )
       {
              m_p = new int [ m_n ] ;
              for ( int i = 0; i < m_n; i++ ) m_p [ i ] = 0 ;
       }
       C ( C& o ) : m_n ( o.m_n )
       {
              m_p = new int [ m_n ] ;
              for ( int i = 0; i < m_n; i++ ) m_p [ i ] = o.m_p [ i ] ;
       }
       ~ C ( ) { delete [ ] m_p ; }
       int& operator [ ] ( const unsigned int i ) const
       { return m_p [ i ] ; }
       // перегрузка оператора =
       C& operator = ( C& right )
```

```
        {
                if ( m_n != right.m_n )
                {
                        // освобождение ранее выделенной памяти
                        delete [ ] m_p ;
                        // модификация размера массива
                        m_n = right.m_n ;
                        // выделение нового блока памяти
                        m_p = new int [ m_n ] ;
                }
                // копирование элементов
                for ( int i = 0; i < m_n; i++ )
                        m_p [ i ] = right.m_p [ i ] ;
                // возврат ссылки на модифицированный массив
                return *this ;
        }
} ;
#include <iostream>
using namespace std ;
void view ( C& ) ;
int main ( )
{
        // создание объекта obj
        C obj ( 4 ) ;                   // в объекте obj массив 0 0 0 0
        // присваивание значений элементам массива в объекте obj
        for ( int i = 0; i < obj.m_n; i++ )
                obj [ i ] = ( i + 1 ) * 2 ;  // вызывается оператор [ ]
        cout << "\nobj:\t\t" ;       view ( obj ) ;
        // создание объекта через указатель
        C* p = new C ( 6 ) ;            // в объекте массив 0 0 0 0 0 0
        cout << "\n*p:\t\t" ; view ( *p ) ;
        // элементы массива объекта, на который указывает p,
        // получают значения элементов массива из объекта obj
        *p = obj ;                      // вызывается оператор =
        cout << "\n*p = obj:\t" ;    view ( *p ) ;
        // создание объекта через указатель с одновременной инициализацией
        C* pCopy = new C ( *p ) ;       // вызывается конструктор копии
        cout << "\nCOPY *p:\t" ;      view ( *pCopy ) ;
        // множественное присваивание
```

```
        C x, y ; x = y = obj ;
        cout << "\nx = y = obj:\n" ;
        cout << "\nx:\t\t" ; view ( x ) ;
        cout << "\ny:\t\t" ; view ( y ) ;
        // разрушение объектов
        delete p ; delete pCopy ;
        return 0 ;
}
// функция вывода состояния объектов типа C
void view ( C& o )
{
        for ( int i = 0; i < o.m_n; i++ ) cout << o [ i ] << '\t' ;
        cout << endl ;
}
```

Оператор присваивания вызывается только в операции присваивания. Когда объект объявляется с одновременной инициализацией (`C* pCopy = new C ( *p ) ;`), вызывается конструктор копирования `C ( *p )`, а не операторная функция.

Конструктор копирования и операторная функция-член для перегрузки оператора = реализованы по-разному. Поскольку конструктор создает новый объект, он начинается с выделения блока оперативной памяти для этого объекта. Операторная функция предназначена для замены значений существующего объекта на новые, которые принадлежат другому объекту. Если размеры этих объектов разные, функция освобождает занимаемую модифицируемым объектом память и выделяет новый блок необходимого размера.

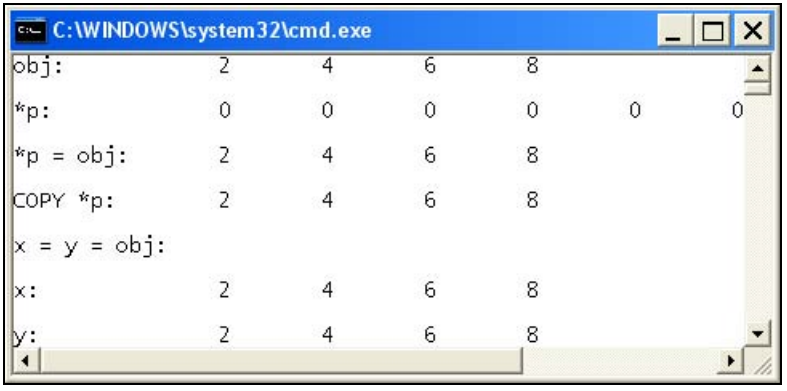


Рис. 17.2. Результаты выполнения программы с перегрузкой оператора =

Из-за того, что операторная функция `operator = ( )` возвращает ссылку на объект `*this`, в программе можно использовать множественное присваивание объектов:

```
x = y = obj;
```

Результаты выполнения программы показаны на рис. 17.2.

Перегрузка оператора ++

Оператор инкремент может находиться слева от операнда (префиксная форма оператора) и справа от него (постфиксная форма оператора). Чтобы компилятор мог различать одну форму от другой, для перегрузки постфиксного оператора задается фиктивный целочисленный параметр (листинг 17.5).

Листинг 17.5. Пример программы с перегрузкой ++

```
class CPoint
{
public:
        int m_x, m_y ;
        CPoint ( ) : m_x ( 0 ), m_y ( 0 ) { }
        CPoint ( int x, int y ) : m_x ( x ), m_y ( y ) { }
        ~CPoint ( ) { }
        CPoint& operator ++ ( )              // префиксный оператор
        { ++m_x ; ++m_y; return *this ; }
        CPoint& operator ++ ( int unused )  // постфиксный оператор
        { m_x++ ; m_y++ ; return *this ; }
} ;
#include <iostream>
using namespace std ;
int main ( )
{
        CPoint a, b ( -8, 6 ) ;
        ++a ; cout << a.m_x << '\t' << a.m_y << endl ;     // выводится 1
1
        b++ ; cout << b.m_x << '\t' << b.m_y << endl ;     // выводится -7
7
        return 0 ;
}
```

Если для класса `CPoint` перегрузить только префиксный оператор, то компилятор предупредит о возможной ошибке и выполнит операцию инкрементации.

Если перегрузить только постфиксный оператор, программа не будет компилироваться.

Перегрузка бинарных операторов + и –

Операторная функция-член при переопределении бинарных операторов должна получать второй операнд в качестве параметра. Следует помнить, что вызов операторной функции генерируется объектом, стоящим слева от знака оператора, и не должен изменять хранимых операндами значений. Если правый операнд может иметь разные типы данных, то следует перегрузить операторную функцию-член.

Листинг 17.6. Пример программы с перегрузкой бинарных операторов + и –

```
// c.h - спецификация C
#pragma once
class C
{
public:
       int* m_p ; int m_n ;
       C ( int = 10 ) ; C ( C& ) ; ~ C ( ) ;
       int& operator [ ] ( const unsigned int ) const ;
       C& operator = ( C& ) ;
       C operator + ( const C& ) const ;   // сложение объектов
       C operator + ( const int& ) const ; // сложение объекта с числом
       C operator - ( const C& ) const ;   // вычитание объектов
} ;
// c.cpp - реализация C
#include "c.h"
C :: C ( int n ) : m_n ( n )
{
       m_p = new int [ m_n ] ;
       for ( int i = 0; i < m_n; i++ ) m_p [ i ] = 0 ;
}
C :: C ( C& o ) : m_n ( o.m_n )
{
       m_p = new int [ m_n ] ;
       for ( int i = 0; i < m_n; i++ ) m_p [ i ] = o.m_p [ i ] ;
}
C :: ~ C ( ) { delete [ ] m_p ; }
// перегрузка [ ]
```

```
int& C :: operator [ ] ( const unsigned int i ) const
{       return m_p [ i ] ;    }
// перегрузка =
C& C :: operator = ( C& right )
{
        if ( m_n != right.m_n )
        { delete [ ] m_p ; m_n = right.m_n ; m_p = new int [ m_n ] ; }
        for ( int i = 0; i < m_n; i++ ) m_p [ i ] = right.m_p [ i ] ;
        return *this ;
}
// перегрузка сложения объектов
C C :: operator + ( const C& right ) const
{
        C tmp ( m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp.m_p [ i ] = m_p [ i ] + right.m_p [ i ] ;
        return tmp ;
}
// перегрузка сложения объекта с числом
C C :: operator + ( const int& right ) const
{
        C tmp ( m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp.m_p [ i ] = m_p [ i ] + right ;
        return tmp ;
}
// перегрузка вычитания объектов
C C :: operator - ( const C& right ) const
{
        C tmp ( m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp.m_p [ i ] = m_p [ i ] - right.m_p [ i ] ;
        return tmp ;
}
// main.cpp - главная функция
#include <iostream>
using namespace std ;
#include "c.h"
void view ( C& ) ;                    // вывод состояния объекта
int main ( )
```

```
{
        C o1 ( 5 ), o2 ( 5 ) ;
        for ( int i = 0; i < o1.m_n; i++ )
        {       o1 [ i ] = ( i + 1 ) * 2 ;   o2 [ i ] = i ;        }
        cout << "\no1:\t\t" ; view ( o1 ) ;
        cout << "o2:\t\t" ;  view ( o2 ) ;
        // сложение объектов o1 и o2
        C res ;
        res = o1 + o2 ;              // вызываются + (объект) и =
        cout << "\nres = o1 + o2:\t" ; view ( res ) ;
        // сложение с числом
        int x ( 5 ) ; C o3 ( 5 ) ;
        cout << "\no3:\t\t" ;  view ( o3 ) ;
        res = o3 + x ;               // вызываются + (число) и =
        cout << "res = o3 + 5:\t" ;  view ( res ) ;
        // вычитание объектов
        res = o3 - o2 ;              // вызываются - и =
        cout << "\nres = o3 - o2:\t" ; view ( res ) ;
        // вывод состояния исходных объектов
        cout << "\no1:\t\t" ; view ( o1 ) ;
        cout << "o2:\t\t" ;  view ( o2 ) ;
        cout << "o3:\t\t" ;  view ( o3 ) ;
        return 0 ;
}
// функция вывода состояния объектов типа C
void view ( C& o )
{
        for ( int i = 0; i < o.m_n; i++ ) cout << o [ i ] << '\t' ;
        cout << endl ;
}
```

В листинге 17.6 перегружена операторная функция `operator + ( )`. Первая версия с прототипом

```
C operator + ( const C& ) const ;
```

предназначена для сложения значений элементов массивов двух объектов типа `C`, один их которых передается по ссылке как параметр. Вторая версия с прототипом

```
C operator + ( const int& ) const ;
```

позволяет складывать значения элементов объекта с целым числом, заданным в качестве параметра.

Примечание

Сложить число с объектом при помощи второй версии функции нельзя. Чтобы это стало возможным, необходимо определить глобальную функцию и объявить ее другом.

Функция-член с прототипом

```
C operator + ( const C& ) const ;
```

предназначена для вычитания объектов. В реализации этой функции следует обратить внимание на выражение

```
tmp.m_p [ i ] = m_p [ i ] - right.m_p [ i ]
```

внутри цикла `for`. Если изменить операнды в операции вычитания, функция вернет неверный результат, потому что переопределенный оператор вычитания вызывается объектом, который находится с левой стороны от знака оператора (объектом `this`), а не стоящим справа объектом.

Названные операторные функции создают и возвращают объект типа `C`, в котором записан результат операции. Благодаря этому, участвующие в операции объекты не меняют свое состояние. Результаты выполнения программы показаны на рис. 17.3.

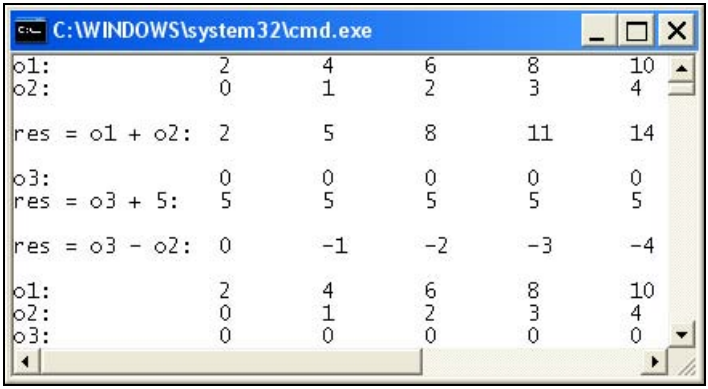


Рис. 17.3. Результаты выполнения программы с перегрузкой бинарных операторв **+** и **–**

Операторные функции-друзья класса

Обычно функции не имеют доступа к закрытым и защищенным членам классов. В C++ при помощи ключевого слова `friend` класс может предоставить функции полный доступ ко всем своим членам, причем область определения функции при этом не имеет значения. Функция-друг (friend-функция, друже-

ственная фукция) может быть глобальной, тогда ее объявление в классе имеет следующий формат:

```
friend прототип_функции-друга ;
```

Если функция-друг определена в другом классе, то для нее следует указать область видимости (имя класса, в котором она определена) согласно следующему формату:

```
friend имя_класса :: прототип_функции-друга ;
```

Прототип дружественной функции можно указывать с любым спецификатором доступа, но предпочтение остается за спецификатором `public`.

Для функций-друзей свойственны следующие правила:

- Функции-друзья получают доступ ко всем членам класса независимо от их спецификатора доступа.
- Функции-друзья не наследуются в производных классах.
- Глобальная функция-друг может вызываться как объектами класса, так и нет.
- Если функция-друг является глобальной, то для нее недоступен указатель `this`.
- Если функция-друг является глобальной, то при ее вызове не используются операторы доступа точка и стрелка.
- Если функция-друг определена в классе, то при ее вызове необходимо использовать операторы доступа точка и стрелка.

Под операторной функцией-другом понимается операторная функция, которая перегружает оператор для класса при помощи дружественной функции.

Глобальная операторная функция-друг для перегрузки унарных операторов имеет следующий формат:

```
тип_функции operator знак_оператора ( параметр1 )
{
        // выполняемые операции
}
```

Для перегрузки бинарных операторов используется следующий формат:

```
тип_функции operator знак_оператора ( параметр1, параметр2 )
{
        // выполняемые операции
}
```

Список параметров функций-друзей не может быть пустым из-за того, что указатель `this` этим функциям не передается, как это было в операторных функциях-членах класса.

Операторные функции-друзья при определения операторов, которые модифицируют левый операнд операции, являющийся объектом, в качестве параметра должны получать ссылку на объект. Если будет передаваться значение, то изменения в значениях данных объекта коснутся только копии параметра, который создается внутри функции. Следовательно, результат выполнения функции не отразится на объекте, к которому эта функция была применена.

Листинг 17.7. Пример перегрузки оператора ++ при помощи дружественных функций

```
class CPoint
{
public:
        int m_x, m_y ;
        CPoint ( ) : m_x ( 0 ), m_y ( 0 ) { }
        CPoint ( int x, int y ) : m_x ( x ), m_y ( y ) { }
        ~CPoint ( ) { }
        // объявление функций-друзей
        friend CPoint& operator ++ ( CPoint& ) ;          // префиксный
++
        friend CPoint& operator ++ ( CPoint&, int ) ;     // постфиксный
++
} ;
// главная функция
#include <iostream>
using namespace std ;
int main ( )
{
        CPoint a, b ( -8, 6 ) ;
        ++a ; cout << a.m_x << '\t' << a.m_y << endl ;    // выводится 1
1
        b++ ; cout << b.m_x << '\t' << b.m_y << endl ;    // выводится -7
7
        return 0 ;
}
// определение функций
CPoint& operator ++ ( CPoint& o )
{       ++o.m_x ; ++o.m_y; return o ;       }
CPoint& operator ++ ( CPoint& o, int unused )
{       o.m_x++ ; o.m_y++ ; return o ;      }
```

Перегруженная операторная функция `operator ++ ( )` является глобальной (листинг 17.7). Перегрузка необходима для того, чтобы в компилятор мог различать префиксную и постфиксную формы записи операции инкремент. Обе версии операторной функции объявлены в классе `CPoint` дружественными. Определение функций находится за пределами класса. Увеличение на 1 производится непосредственно в объекте, так как он передается в функцию по ссылке.

Листинг 17.8. Пример перегрузки бинарных операторов + и – при помощи дружественных функций

```
// c.h - спецификация C
#pragma once
class C
{
       int* m_p ; int m_n ;                  // закрытые данные
public:
       C ( int = 10 ) ;                      // конструктор по умолчанию
       C ( C& ) ;                            // конструктор копирования
       ~ C ( ) ;                             // деструктор
       int getN ( void ) const ;             // возврат размера массива
       // операторные функции-члены
       int& operator [ ] ( const unsigned int ) const ;
       C& operator = ( C& ) ;
       // операторные функции-друзья
       // оператор +
       friend C operator + ( const C&, const C& ) ; // объект + объект
       friend C operator + ( const C&, const int& ) ;// объект + число
       friend C operator + ( const int&, const C& ) ;// число + объект
       // оператор -
       friend C operator - ( const C&, const C& ) ; // объект - объект
       // функция-друг для вывода состояния объекта
       friend void view ( const C& ) ;
} ;
// c.cpp - реализация C
#include "c.h"
C :: C ( int n ) : m_n ( n )
{
       m_p = new int [ m_n ] ;
       for ( int i = 0; i < m_n; i++ ) m_p [ i ] = 0 ;
```

```
}
C :: C ( C& o ) : m_n ( o.m_n )
{
       m_p = new int [ m_n ] ;
       for ( int i = 0; i < m_n; i++ ) m_p [ i ] = o.m_p [ i ] ;
}
C :: ~ C ( ) { delete [ ] m_p ; }
int C :: getN ( void ) const { return m_n ; }
// перегрузка оператора [ ]
int& C :: operator [ ] ( const unsigned int i ) const
{      return m_p [ i ] ;    }
// перегрузка оператора =
C& C :: operator = ( C& right )
{
       if ( m_n != right.m_n )
       { delete [ ] m_p ; m_n = right.m_n ; m_p = new int [ m_n ] ; }
       for ( int i = 0; i < m_n; i++ ) m_p [ i ] = right.m_p [ i ] ;
       return *this ;
}
// main.cpp - главная функция
#include <iostream>
using namespace std ;
#include "c.h"
// прототипы глобальных функций
C operator + ( const C&, const C& ) ;      // сложение объектов
C operator + ( const C&, const int& ) ;    // сложение объекта с числом
C operator + ( const int&, const C& ) ;    // сложение числа с объектом
C operator - ( const C&, const C& ) ;      // вычитание объектов
void view ( const C& ) ;                   // вывод состояния объекта
int main ( )
{
       C o1 ( 5 ), o2 ( 5 ) ;
       for ( int i = 0; i < o1.getN ( ); i++ )
       {      o1 [ i ] = ( i + 1 ) * 2 ; o2 [ i ] = i ;         }
       cout << "\no1:\t\t" ; view ( o1 ) ;
       cout << "o2:\t\t" ; view ( o2 ) ;
       // сложение объектов o1 и o2
       C res ; res = o1 + o2 ;
       cout << "\nres = o1 + o2:\t" ; view ( res ) ;
```

```
        // сложение объекта o3 с числом x
        int x ( 5 ) ; C o3 ( 5 ) ;
        cout << "\no3:\t\t" ;  view ( o3 ) ;
        res = o3 + x ;                          // объект + число
        cout << "res = o3 + 5:\t" ; view ( res ) ;
        res = x + o3 ;                          // число + объект
        cout << "res = 5 + o3:\t" ; view ( res ) ;
        // вычитание объектов
        res = o3 - o2 ;                         // объект - объект
        cout << "\nres = o3 - o2:\t" ; view ( res ) ;
        res = o2 - o3 ;                         // объект - объект
        cout << "res = o2 - o3:\t" ; view ( res ) ;
        // вывод состояния исходных объектов
        cout << "\no1:\t\t" ; view ( o1 ) ;
        cout << "o2:\t\t" ;          view ( o2 ) ;
        cout << "o3:\t\t" ;          view ( o3 ) ;
        return 0 ;
}
// функция вывода состояния объектов типа C
void view ( const C& o )
{
        for ( int i = 0; i < o.m_n; i++ ) cout << o [ i ] << '\t' ;
        cout << endl ;
}
// все следующие функции используют перегруженный оператор [ ]
// перегрузка оператора + (сложение объектов)
C operator + ( const C& left, const C& right )
{
        C tmp ( left.m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp [ i ] = left [ i ] + right [ i ] ;
        return tmp ;
}
// перегрузка оператора + (сложение объекта с числом)
C operator + ( const C& left, const int& right )
{
        C tmp ( left.m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp [ i ] = left [ i ] + right ;
```

```
        return tmp ;
}
// перегрузка оператора + (сложение числа с объектом)
C operator + ( const int& left, const C& right )
{
        C tmp ( right.m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp [ i ] = left + right [ i ] ;
        return tmp ;
}
// перегрузка оператора - (вычитание объектов)
C operator - ( const C& left, const C& right )
{
        C tmp ( left.m_n ) ;
        for ( int i = 0; i < tmp.m_n ; i++ )
                tmp [ i ] = left [ i ] - right [ i ] ;
        return tmp ;
}
```

В листинге 17.8 для класса `C` перегружены операторы индексации `[ ]`, присваивания `=`, сложения `+` (объектов, объекта и числа, числа и объекта), вычитания `-` (объектов). Операторы `[ ]` и `=` могут быть перегружены только при помощи операторных функций-членов. Остальные операторы перегружаются при помощи friend-функций, определение которых является глобальным и находится за пределами класса `C` в файле с главной функцией. Чтобы глобальные функции не могли изменять состояние параметров, все параметры объявлены в прототипах с ключевым словом `const`. В теле каждой friend-функции используется перегруженный оператор индексации, который обеспечивает доступ к элементам массива объектов без указания названия массива. Результаты выполнения программы из примера показаны на рис. 17.4.

Функция-друг с прототипом

```
C operator + ( const C&, const int& ) ;
```

и функция-друг с прототипом

```
C operator + ( const int&, const C& ) ;
```

позволяют решить проблему сложения объекта и числа, которая была неразрешимой при использовании операторных функций-членов. Первая версия friend-функции реализует сложение объекта с числом, вторая — числа с объектом.

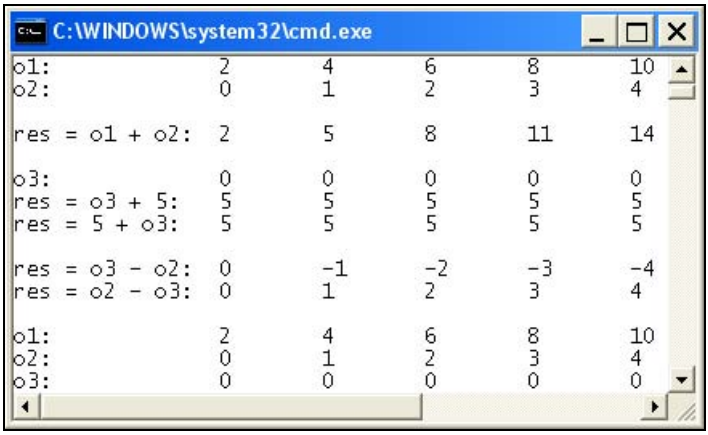


Рис. 17.4. Результаты выполнения программы с использованием дружественных функций

Операторная функция-друг `operator - ( )` гарантирует верный результат выполнения операции при любом положении операндов, так как из левого операнда всегда будет вычитаться правый операнд, ссылки на которые получает функция.

Помимо операторных функций-друзей, в классе объявлена дружественная функция `view ( )`, предназначенная для вывода состояния объекта класса. Эта функция ничего не переопределяет, просто ей разрешается доступ к закрытым членам класса.

Примечание

Главную функцию тоже можно сделать дружественной классу. Для этого необходимо в классе объявить ее как `friend int main ( ) ;`.

Перегрузка вывода для классов

Поток вывода `ostream` представляет собой механизм языка для преобразования значений различного типа в последовательность символов и является специализацией универсального шаблона `basic :: ostream`. Этот шаблон представлен в заголовке `<iostream>` и определен в пространстве имен `std`. В `<iostream>` объявлен стандартный символьный поток вывода `cout` и определен оператор << для управления выводом встроенных типов данных.

Оператор << для типов `class` можно переопределить так, что этим оператором можно пользоваться так же, как и для встроенных типов (листинг 17.9).

Листинг 17.9. Пример перегрузки оператора <<

```
#include <iostream>
using namespace std ;
class CPoint
{
       int m_x, m_y ;
public:
       CPoint ( ) : m_x ( 0 ), m_y ( 0 ) { }
       CPoint ( int x, int y ) : m_x ( x ), m_y ( y ) { }
       ~CPoint ( ) { }
       int getX ( ) { return m_x ; } int getY ( ) { return m_y ; }
} ;
ostream& operator << ( ostream& str, CPoint p )
{ return ( str << p.getX() << '\t' << p.getY() << endl ) ; }
int main ( )
{ CPoint a ( 5, 7 ) ; cout << a ; return 0 ; }    // выводится 5 7
```

Перегрузка ввода для классов

Стандартный поток ввода `cin` определяется классом `istream` универсального шаблона `basic :: istream`, который представлен в заголовке `<iostream>` пространства имен `std`. Класс `istream` обеспечивает оператор ввода `>>` для встроенных типов данных.

Оператор `>>` для типов `class` можно переопределить подобно тому, как это выполнялось для оператора вывода. Однако для ввода существенно, чтобы параметр типа `class` передавался по ссылке (листинг 17.10).

Листинг 17.10. Пример перегрузки оператора >>

```
#include <iostream>
using namespace std ;
class CPoint
{
       int m_x, m_y ;
public:
       CPoint ( ) : m_x ( 0 ), m_y ( 0 ) { }
       CPoint ( int x, int y ) : m_x ( x ), m_y ( y ) { }
       ~CPoint ( ) { }
       int getX ( ) { return m_x ; } int getY ( ) { return m_y ; }
```

```
} ;
istream& operator >> ( istream& str, CPoint& p )
{
        int x ( 0 ), y ( 0 ) ;        str >> x >> y ;
        p = CPoint ( x, y ) ;         return str ;
}
int main ( )
{
        CPoint a ; cin >> a ;
        cout << a.getX ( ) << '\t' << a.getY ( ) << endl ;
        return 0 ;
}
```

Пример перегрузки операторов с использованием дружественных функций

В листинге 17.11 рассматривается переопределение разных операторов для класса `CMatrix` (матрица) при помощи функций-друзей. Члены-данные класса (количество строк матрицы `m_m`, количество столбцов матрицы `m_n`, указатель на указатель целочисленного двумерного массива `m_p`) объявлены с закрытым спецификатором доступа `private`, но остаются доступными для функций-друзей.

Перечислим прототипы и назначение функций-друзей:

- `istream& operator >> ( istream&, CMatrix& ) ;` — вводит матрицу;
- `ostream& operator << ( ostream&, const CMatrix& ) ;` — выводит матрицу;
- `CMatrix operator * ( const CMatrix&, const CMatrix& ) ;` — умножает матрицы. Возвращает матрицу-результат перемножения;
- `CMatrix& operator – ( CMatrix& ) ;` — изменяет состояние матрицы так, что отрицательные элементы матрицы меняют свой знак на противоположный;
- `bool operator == ( const CMatrix&, const CMatrix& ) ;` — сравнивает матрицы на равенство. Возвращает истину, если значения всех элементов обеих матриц равны. Иначе возвращает ложь;
- `bool operator != ( const CMatrix&, const CMatrix& ) ;` — сравнивает матрицы на неравенство. Возвращает истину, если в матрицах встречается хотя бы одно расхождение в значении элементов с одинаковыми индексами или у матриц не совпадает размер хотя бы по одному индексу. Иначе возвращает ложь.

Для класса `CMatrix` дополнительно перегружаются оператор индексации `[ ]` и оператор присваивания `=`. Эти операторы определены при помощи операторных функций-членов, поскольку использование дружественных функций для них недопустимо. Перегрузка индексации позволяет в определении функций-друзей не указывать адрес массива для получения доступа к его элементам, а перегрузка присваивания делает безопасным копирование объектов-матриц.

Листинг 17.11. Пример перегрузки операторов для класса `CMatrix` с помощью функций-друзей

```
// Martix.h - спецификация CMatrix
#pragma once
#include <iostream>
using namespace std ;
class CMatrix
{
       int m_m ; int m_n ; int** m_p ;
public:
       int getM ( ) const { return m_m ; }
       int getN ( ) const { return m_n ; }
       CMatrix ( const int = 10, const int = 10 ) ; ~CMatrix ( ) ;
       // перегрузка оператора [ ]
       int* operator [ ] ( const unsigned int i ) const ;
       // перегрузка оператора =
       CMatrix& operator = ( const CMatrix& ) ;
       // перегрузка оператора >> (ввод матрицы)
       friend istream& operator >> ( istream&, CMatrix& ) ;
       // перегрузка оператора << (вывод матрицы)
       friend ostream& operator << ( ostream&, const CMatrix& ) ;
       // перегрузка оператора * (умножение матриц)
       friend CMatrix operator * ( const CMatrix&, const CMatrix& ) ;
       // перегрузка оператора - (изменение отрицательных элементов)
       friend CMatrix& operator - ( CMatrix& ) ;
       // перегрузка оператора == (сравнение матриц на равенство)
       friend bool operator == ( const CMatrix&, const CMatrix& ) ;
       // перегрузка оператора != (сравнение матриц на неравенство)
       friend bool operator != ( const CMatrix&, const CMatrix& ) ;
} ;
// Matrix.cpp - реализация CMatrix
```

```
#include "Matrix.h"
CMatrix :: CMatrix ( const int m, const int n ) //: m_m ( m ), m_n ( n )
{
       m_m = m ; m_n = n ; m_p = new int* [ m_m ] ;
       for ( int i = 0; i < m_m; i++ ) m_p [ i ] = new int [ m_n ] ;
       for ( int i = 0; i < m_m; i++ )
              for ( int j = 0; j < m_n; j++ ) m_p [ i ] [ j ] = 0 ;
}
CMatrix :: ~CMatrix ( )
{
       for ( int i = 0; i < m_m; i++ ) delete [ ] m_p [ i ] ;
       delete [ ] m_p ;
}
// перегрузка оператора [ ]
int* CMatrix :: operator [ ] ( const unsigned int i ) const
{      return m_p [ i ] ;    }
// перегрузка оператора =
CMatrix& CMatrix :: operator = ( const CMatrix& right )
{
       // освобождение ранее выделенной памяти для левого операнда
       for ( int i = 0; i < m_m; i++ ) delete [ ] m_p [ i ] ;
       delete [ ] m_p ;
       // выделение новой оперативной памяти для левого операнда
       m_m = right.m_m ; m_n = right.m_n ; m_p = new int* [ m_m ] ;
       for ( int i = 0; i < m_m; i++ ) m_p [ i ] = new int [ m_n ] ;
       // модификация левого операнда
       for ( int i = 0; i < m_m; i++ )
              for ( int j = 0; j < m_n; j++ )
                     m_p [ i ] [ j ] = right.m_p [ i ] [ j ] ;
       return *this ;
}
// main.cpp - главная функция
#include <iostream>
using namespace std ;
#include "Matrix.h"
// прототипы функций-друзей
istream& operator >> ( istream&, CMatrix& ) ;
ostream& operator << ( ostream&, const CMatrix& ) ;
CMatrix operator * ( const CMatrix&, const CMatrix& ) ;
```

```
CMatrix& operator - ( CMatrix& ) ;
bool operator == ( const CMatrix&, const CMatrix& ) ;
bool operator != ( const CMatrix&, const CMatrix& ) ;
int main ( )
{
        // объявление матриц
        CMatrix A ( 2, 3 ) ; CMatrix B ( 3, 3 ) ;
        // ввод матриц
        cout << "Enter A (" << A.getM ( ) * A.getN ( ) << " elements)\n" ;
        cin >> A ;
        cout << "Enter B (" << B.getM ( ) * B.getN ( ) << " elements)\n" ;
        cin >> B ;
        // вывод состояний матриц
        cout << "\nView A\n" ;       cout << A ;
        cout << "View B\n" ;         cout << B ;
        // умножение матриц A и B
        CMatrix res ;                res = A * B ;
        cout << "A * B\n" ;          cout << res ;
        if ( B.getN ( ) != A.getM ( ) )
                cout << "B * A !!! multiplying it is impossible\n" ;
        // преобразование матрицы
        cout << "\n-A\n" ;    -A ;   cout << A ;
        // сравнение матриц на равенство
        cout << "A == B\t" ;         cout << ( A == B ) ;
        // сравнение матриц на неравенство
        cout << "\nA != B\t" ;       cout << ( A != B ) << endl ;
        return 0 ;
}
// определение функций-друзей
// перегрузка оператора >> (ввод матрицы)
istream& operator >> ( istream& input, CMatrix& in )
{
        for ( int i = 0; i < in.m_m; i++ )
                for ( int j = 0; j < in.m_n; j++ )
input >> in [ i ] [ j ] ;
        return input ;
}
```

```
// перегрузка оператора << (вывод матрицы)
ostream& operator << ( ostream& output, const CMatrix& out )
{
	for ( int i = 0; i < out.m_m; i++ )
	{
		for ( int j = 0; j < out.m_n; j++ )
			output << out [ i ] [ j ] << '\t' ;
		cout << endl ;
	}
	output << endl ;
	return output ;
}
// перегрузка оператора * (умножение матриц)
CMatrix operator * ( const CMatrix& a, const CMatrix& b )
{
	CMatrix c ( a.m_m, b.m_n ) ;
	for ( int i = 0; i < c.m_m; i++ )
		for ( int k = 0; k < c.m_n; k++ )
		{
			c [ i ] [ k ] = 0 ;
			for ( int j = 0; j < a.m_n; j++ )
				c [ i ] [ k ] += a [ i ] [ j ] *
 b [ j ] [ k ] ;
		}
	return c ;
}
// перегрузка оператора - (изменение знака у отрицательных элементов)
CMatrix& operator - ( CMatrix& a )
{
	for ( int i = 0; i < a.m_m; i++ )
		for ( int j = 0; j < a.m_n; j++ )
			if ( a [ i ] [ j ] < 0 )
a [ i ] [ j ] = -a [ i ] [ j ] ;
	return a ;
}
// перегрузка оператора == (сравнение матриц на равенство)
bool operator == ( const CMatrix& a, const CMatrix& b )
{
	if ( ( a.m_m != b.m_m ) || ( a.m_n != b.m_n ) )
```

```
                return false ;
        for ( int i = 0; i < a.m_m; i++ )
                for ( int j = 0; j < a.m_n; j++ )
                        if ( a [ i ] [ j ] != b [ i ] [ j ] )
                                return false ;
        return true ;
}
// перегрузка оператора != (сравнение матриц на неравенство)
bool operator != ( const CMatrix& a, const CMatrix& b )
{
        if ( ( a.m_m != b.m_m ) || ( a.m_n != b.m_n ) )
                return true ;
        for ( int i = 0; i < a.m_m; i++ )
                for ( int j = 0; j < a.m_n; j++ )
                        if ( a [ i ] [ j ] != b [ i ] [ j ] )
                                return true ;
        return false ;
}
```

Главная функция `main ( )` начинается с объявления двух объектов-матриц `A` и `B` класса `CMatrix`. При помощи перегруженных операторов `>>` и `<<` производится ввод и вывод значений элементов массивов этих объектов.

Результат умножения матриц присваивается объекту `res` типа `CMatrix`. В процессе вычисления вызываются `friend` функция `operator * ( )` и функции-члены `operator [ ] ( )`, `operator = ( )`. Матрица `res` выводится перегруженным оператором `<<`. Оператор `if` проверяет возможность умножения матрицы `B` на матрицу `A`. Вследствие того, что количество столбцов матрицы `B` не равно количеству строк матрицы `A`, выводится сообщение о невозможности такого умножения.

Оператор, использующий выражение `-A`, вызывает friend-функцию:

`operator - ( )`,

которая меняет знак у отрицательных элементов матрицы `A`.

В конце главной функции при сравнении матриц на равенство и неравенство вызываются функции-друзья `operator == ( )` и `operator != ( )`, фактическими параметрами которых являются объекты-матрицы `A` и `B`. Результаты исполнения программы представлены на рис. 17.5.

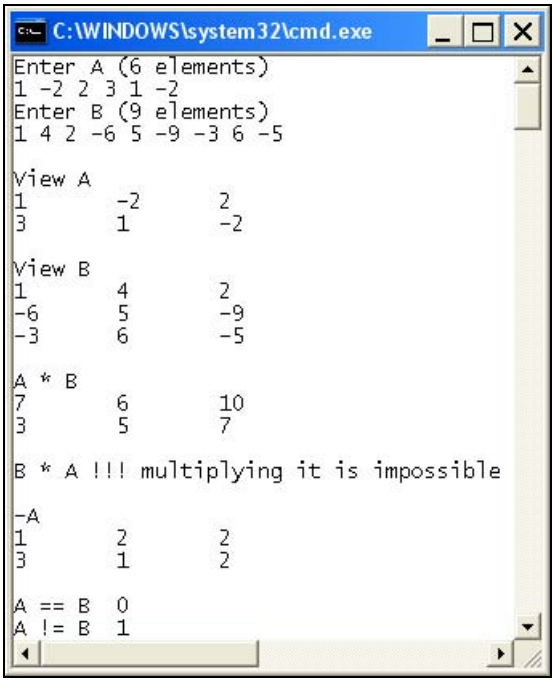


Рис. 17.5. Результаты исполнения программы с перегрузкой разных операторов

Перегрузка операторов в производных классах

Производные классы наследуют все переопределенные в базовом классе операторные функции-члены, кроме функции, которая перегружает оператор присваивания. Этот оператор должен быть определен в каждом производном классе с помощью операторной функции-члена.

При перегрузке операторов в базовом классе можно применять виртуальные операторные функции-члены. Если функция объявлена в базовом классе как чистая виртуальная, в производных классах необходимо иметь ее собственное определение с прототипом, строго соответствующим прототипу из базового класса.

Далее рассматривается пример программы, где от абстрактного класса `CPerson` (персона) наследуют два производных класса `CTeacher` (преподаватель) и `CStudent` (студент). В листинге 17.12 использована перегрузка операторов как с помощью функций-членов, так и функций-друзей.

Листинг 17.12. Пример программы с перегрузкой операторов в производных классах

```
// Person.h - спецификация CPerson (абстрактный класс)
#pragma once
#include <iostream>
#include <string>
using namespace std ;
class CPerson
{
public:
       struct
       {
              string m_fName ;      // имя человека
              string m_lName ;      // фамилия человека
       } m_data ;                   // данные о человеке
       string m_site ;              // место работы или учебы
       CPerson ( ) ;
       CPerson ( const string&, const string&, const string& ) ;
       virtual ~CPerson ( ) { }
       // оператор == для базовой части
       bool operator == ( CPerson& ) const ;
       // чистая виртуальная функция для вывода состояния объекта
       virtual ostream& out ( ostream& ) const = 0 ;
} ;
// Person.cpp - реализация CPerson
#include "Person.h"
CPerson :: CPerson ( ) : m_site ( "" )
{      m_data.m_fName = "" ; m_data.m_lName = "" ;       }
CPerson ::
CPerson ( const string& f, const string& l, const string& s )
: m_site ( s )
{      m_data.m_fName = f ; m_data.m_lName = l ;  }
// оператор == для базовой части
// Назначение: Сравнить значения данных двух подобъектов на равенство.
// Описание:  Возвращает истину, если значения равны,
//            иначе возвращает ложь.
bool CPerson :: operator == ( CPerson& r ) const
{
```

```
        if ( m_site != r.m_site ) return false ;
        if ( ( m_data.m_fName == r.m_data.m_fName )
                && ( m_data.m_lName == r.m_data.m_lName ) ) return true ;
        return false ;
}
// Teacher.h - спецификация CTeacher (Преподаватель)
#pragma once
#include "Person.h"
class CTeacher : public CPerson
{
protected:
        int m_exp ;                    // стаж работы
public:
        CTeacher ( ) ;
        CTeacher ( const string&, const string&, const string&,
        ↳const int& ) ;
        ~CTeacher ( ) { }
        CTeacher& operator = ( const CTeacher& ) ;          // оператор =
        bool operator == ( CTeacher& ) const ;              // оператор ==
        bool operator != ( CTeacher& ) const ;              // оператор !=
        // виртуальная функция вывода состояния объекта
        ostream& out ( ostream& ) const ;
        // функция-друг (оператор ==)
        friend bool operator == ( const CPerson&, const CPerson& ) ;
} ;
// Teacher.h - реализация CTeacher (Преподаватель)
#include "Teacher.h"
CTeacher :: CTeacher ( ) : CPerson ( ), m_exp ( 0 ) { }
CTeacher :: Cteacher ( const string& f, const string& l, const string& s,
↳ const int& e )
: CPerson ( f, l, s ), m_exp ( e ) { }
// оператор =
CTeacher& CTeacher :: operator = ( const CTeacher& r )
{
        m_data.m_fName = r.m_data.m_fName ;
        m_data.m_lName = r.m_data.m_lName ;
        m_site = r.m_site ; m_exp = r.m_exp ;
        return *this ;
}
```

```
// оператор ==
// Назначение: Сравнить значения данных двух объектов на равенство.
// Описание:  Возвращает истину, если значения равны,
//            иначе возвращает ложь.
bool CTeacher :: operator == ( CTeacher& r ) const
{
       if ( ! CPerson :: operator == ( r ) ) return false ;
       if ( m_exp != r.m_exp ) return false ;
       return true ;
}
// оператор !=
// Назначение: Сравнить значения данных двух объектов на неравенство.
// Описание:  Возвращает истину, если хотя бы одно значение
//            не совпадает, иначе возвращает ложь.
bool CTeacher :: operator != ( CTeacher& r ) const
{
       if ( m_site != r.m_site ) return true ;
       if ( ( m_data.m_fName != r.m_data.m_fName )
              || ( m_data.m_lName != r.m_data.m_lName ) ) return true ;
       if ( m_exp != r.m_exp ) return true ;
       return false ;
}
// вывод состояния объекта
ostream& CTeacher :: out ( ostream& str ) const
{
       str << m_site << '\t' ; str << m_data.m_fName << '\t' ;
       str << m_data.m_lName << '\t' ; str << m_exp << endl ;
       return str ;
}
// Student.h - спецификация CStudent (Студент)
#pragma once
#include "Person.h"
class CStudent : public CPerson
{
protected:
       string m_spec ;              // учебная специальность
public:
       CStudent ( ) ;
```

```
        CStudent ( const string&, const string&, const string&,
↳const string& ) ;
        ~CStudent ( ) { }
        CStudent& operator = ( const CStudent& ) ;          // оператор =
        bool operator == ( CStudent& ) const ;              // оператор ==
        bool operator != ( CStudent& ) const ;              // оператор !=
        // виртуальная функция вывода состояния объекта
        ostream& out ( ostream& ) const ;
        // функция-друг (оператор ==)
        friend bool operator == ( const CPerson&, const CPerson& ) ;
} ;
// Student.h - реализация CStudent
#include "Student.h"
CStudent :: CStudent ( ) : CPerson ( ), m_spec ( "" ) { }
CStudent :: CStudent ( const string& f, const string& l, const string& s,
↳const string& sp )
: CPerson ( f, l, s ), m_spec ( sp ) { }
// оператор =
CStudent& CStudent :: operator = ( const CStudent& r )
{
        m_data.m_fName = r.m_data.m_fName ;
        m_data.m_lName = r.m_data.m_lName ;
        m_site = r.m_site ; m_spec = r.m_spec ;
        return *this ;
}
// оператор ==
// Назначение: Сравнить значения данных двух объектов на равенство.
// Описание:  Возвращает истину, если значения равны,
//            иначе возвращает ложь.
bool CStudent :: operator == ( CStudent& r ) const
{
        if ( ! CPerson :: operator == ( r ) ) return false ;
        if ( m_spec != r.m_spec ) return false ;
        return true ;
}
// оператор !=
// Назначение: Сравнить значения данных двух объектов на неравенство.
// Описание:  Возвращает истину, если хотя бы одно значение не совпадает,
//            иначе возвращает ложь.
```

```
bool CStudent :: operator != ( CStudent& r ) const
{
       if ( m_site != r.m_site )    return true ;
       if ( ( m_data.m_fName != r.m_data.m_fName )
              || ( m_data.m_lName != r.m_data.m_lName ) ) return true ;
       if ( m_spec != r.m_spec )    return true ;
       return false ;
}
// вывод состояния объекта
ostream& CStudent :: out ( ostream& str ) const
{
       str << m_site << '\t' ; str << m_data.m_fName << '\t' ;
       str << m_data.m_lName << '\t' ; str << m_spec << endl ;
       return str ;
}
// main.cpp - главная функция
#include <iostream>
#include <string>
using namespace std ;
#include "Teacher.h"
#include "Student.h"
// прототипы глобальных функций
ostream& operator << ( ostream&, const CPerson& ) ;
bool operator == ( const CPerson&, const CPerson& ) ;
// главная функция
int main ( )
{
       // создание объектов
       CStudent s1 ("Mark", "Plotnikov", "Pedagogical U.", "050201.65")
;
       CStudent s2 ("Hellen", "Magnificent", "Academy of arts",
↳"031501.65") ;
       CTeacher* pT1 = new CTeacher ("Seda", "Cornum", "Academy of
↳arts", 4) ;
       CTeacher* pT2 = new CTeacher ("Marin", "Marinov", "Pedagogical
↳U.", 7) ;
       // вывод состояния (перегрузка <<)
       cout << "\nStudents\n" ; cout << s1 ; cout << s2 ;
       cout << "\nTeachers\n" ; cout << *pT1 ; cout << *pT2 ;
       // сравнение на неравенство (перегрузка != из CStudent)
       if ( s1 != s2 ) cout << "\nThe students completely different!\n" ;
```

```
        // сравнение на равенство (перегрузка == при помощи friend)
        if ( s1 == *pT2 )
                cout << "\nMark studies and Marin works at Pedagogical
↳university.\n\n" ;
        // копирование (перегрузка = из CTeacher )
        CTeacher* p = new CTeacher ; * p = *pT1 ;
        // вывод состояния (перегрузка <<)
        cout << * p ; cout << *pT1 ;
        // сравнение на равенство (перегрузка == из CTeacher )
        if ( *p == *pT1 ) cout << "These teachers are coincide.\n" ;
        // разрушение объектов
        delete pT1 ; delete pT2 ; delete p ;
        return 0 ;
}
// определение глобальных функций
// Назначение: перегрузка оператора <<
// Параметры: ссылка на выходной поток - ostream& str,
//            ссылка на базовый тип - const CPerson& r.
// Описание:  Если r имеет тип CTeacher, то вызывается CTeacher::out ( ).
//            Если r типа CStudent, то вызывается CStudent::out ( ).
ostream& operator << ( ostream& str, const CPerson& r )
{       return r.out ( str ) ;       }
// Назначение: перегрузка оператора ==
// Параметры: ссылки на базовый тип const CPerson& l, const CPerson& r.
// Описание:  Возвращает истину,
//      если место работы преподавателя совпадает    с местом учебы студента.
//            Иначе возвращает ложь.
bool operator == ( const CPerson& l, const CPerson& r )
{
        if ( l.m_site == r.m_site ) return true ;
        return false ;
}
```

Исходный текст включает исчерпывающие комментарии, поэтому пример не нуждается в подробных пояснениях. Есть два существенных момента реализации, на которые следует обратить внимание:

- Правильность вывода состояний объектов классов `CTeacher` и `CStudent` достигается путем ввода чистой виртуальной функции `out ( )` в базовом классе `CPerson` и определением версий этой функции в производных. Вир-

туальная функция `out ( )` объединяется с потоком вывода `ostream` и оператором `<<`, а глобальная функция `operator << ( )` использует подходящую версию `out ( )`. Выбор версии происходит во время выполнения программы в соответствии с типом второго параметра операторной функции.

- В производных классах определена операторная функция-член `operator == ( )`, которая предназначена для сравнения объектов классов на полное совпадение. В программе также определена глобальная функция `operator == ( )`, сравнивающая у объектов производных классов значение объявленного в базовом классе данного `m_site`. Эта функция объявлена для производных классов как дружественная. Параметром функции-члена в производном классе является ссылка на производный тип, а параметрами глобальной функции являются ссылки на базовый тип. Таким образом, вызов нужного оператора `==` будет продиктован типами объектов, вызывающих этот оператор. Если типы объектов совпадают, то вызывается функция-член класса, имеющего имя типа объектов. Если же типы объектов разные, вызывается дружественная функция.

Результаты исполнения программы показаны на рис. 17.6.

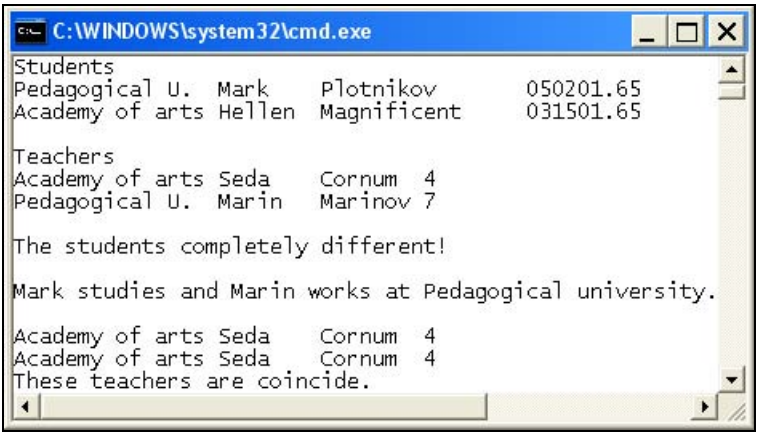
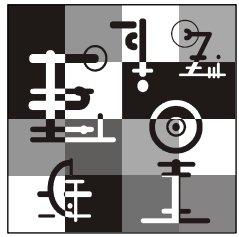


Рис. 17.6. Результаты исполнения программы с перегрузкой операторов в производных классах



Глава 18

Шаблон классов

Шаблон классов (class template) — параметризованный класс, на основе которого строятся другие классы. Шаблон определяет правила конструирования каждого отдельного класса из множества допустимых этим шаблоном классов. Классы, в свою очередь, определяют правила конструирования объектов. Графически связь между шаблоном, классами и объектами представлена на рис. 18.1.

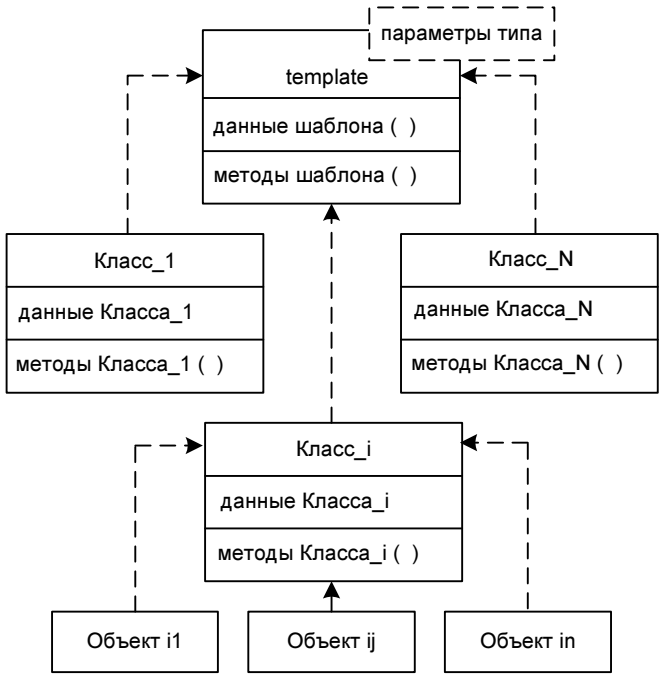


Рис. 18.1. Взаимосвязь между шаблоном, классами и объектами

Шаблон классов представляет собой класс, в котором определены данные и методы, но фактический тип данных задается в качестве параметра при создании объектов класса. Шаблоны дают возможность многократно использовать один и тот же код, который позволяет компилятору автоматизировать процесс реализации типа. Для шаблонных классов характерными являются следующие свойства:

- Шаблон позволяет передать в класс один или несколько типов в виде параметров. Передаваемым типом может быть любой базовый или производный тип, включая тип `class`.
- Параметрами шаблона могут быть не только типы, но и константные выражения.
- Объявление шаблона должно быть только глобальным.
- Каждый реализованный шаблон (созданый объект шаблона классов) требует собственного откомпилированного объектного кода. Чтобы сократить размер объектного модуля программы, рекомендуется наследовать наиболее общую часть шаблона от базового класса. Для базового и производного классов в этом случае необходим один и тот же реализованный тип.
- Базовый класс для шаблона может быть как шаблонным, так и обычным классом. Обычный класс может быть порожден от реализованного шаблона классов.
- Нельзя использовать указатель на базовый шаблонный класс для получения доступа к методам производных классов. Дело в том, что типы, полученные даже из одного и того же шаблона, всегда являются разными.
- В описание шаблона классов можно включать дружественные функции. Если функция-друг не использует спецификатор шаблона, то она считается универсальной для всех экземпляров шаблона. Если же в прототипе функции-друга содержится шаблон параметров, то эта функция будет дружественной только для того класса, экземпляр которого создается.
- Статические члены-данные специфичны для каждого реализованного шаблона.

Чаще всего шаблоны классов используются для программирования контейнерных классов (контейнеров), которые предназначены для хранения объектов других классов.

Объявление шаблона классов

Объявление шаблона классов может быть только глобальным и должно начинаться со строки, имеющей следующий формат:

```
template < class параметр_типа1, …, class параметр_типаN >
```

`Параметр_типа` замещает произвольный тип и используется в реализации класса как имя типа. Угловые скобки и ключевое слово `class` перед каждым `параметром_типа` — обязательные атрибуты объявления шаблона.

Методы должны быть объявлены как шаблоны функций, поэтому заголовок метода, определение которого находится за пределами спецификации класса, имеет следующий формат:

```
template < class параметр_типа1, …, class параметр_типаN >
тип_функции имя_шаблона < параметр_типа1, …, параметр_типаN >
:: имя_функции ( список параметров функции )
```

Спецификация и реализация шаблона классов при раздельной компиляции обязательно должны находиться в одном файле. Чтобы можно было компилировать шаблон, этот файл должен иметь расширение `.cpp`. Файл с шаблоном включается в программу с помощью директивы препроцессора `#include`.

Листинг 18.1. Пример объявления шаблона классов

```
// Container.cpp - шаблон классов TContainer
// Шаблон описывает контейнер для хранения объектов разных типов
// U - параметр типа (задает тип объектов)
// Спецификация TContainer
template < class U >
class TContainer
{
       U *m_p ;       // указатель на элементы контейнера
       size_t m_sz ; // размер контейнера
public:
       explicit TContainer ( size_t = 10 ) ;
       ~TContainer ( ) ;
       U& getElem ( int ) ; // возврат ссылки на хранимый элемент
} ;
// Реализация TContainer
// Конструктор
template < class U >
TContainer < U > :: TContainer ( size_t n )
: m_sz ( n ) { m_p = new U [ m_sz ] ; }
// Деструктор
template < class U >
TContainer < U > :: ~TContainer ( )
```

```
{       delete [ ] m_p ;      }
// Возврат ссылки на элемент контейнера
// Прототип:  U& getElem ( int ) ;
// Параметры: n - индекс элемента, хранимого в контейнере.
// Описание:  Возвращает ссылку элемента типа U с заданным индексом.
//      Если индекс находится вне допустимого диапазона,
//      вызывается функция exit ( ), и программа завершается с кодом 1.
//      Метод может использоваться в операторе присваивания слева.
template < class U >
U& TContainer < U > :: getElem ( int i )
{
       if ( i >= 0 && i < m_sz ) return * ( m_p + i ) ;
       exit ( 1 ) ;
}
```

Шаблон `TContainer` (листинг 18.1) описывает множество контейнерных классов для хранения `m_sz` объектов разного типа, который задается параметром типа `U`. Размер конкретного контейнера определяется значением членаданного `m_sz`. Объекты реализованного контейнера располагаются в оперативной памяти, начиная с адреса, на который указывает указатель `m_p`, с типом, соответствующим параметру типа `U`.

Конструктор класса выделяет блок памяти для размещения `m_sz` элементов типа `U`. Конструктор объявлен с ключевым словом `explicit` и имеет параметр по умолчанию `n`. Если при объявлении объекта класса `TContainer` не указывать размер контейнера, то по умолчанию он будет равен 10.

Примечание

Ключевое слово `explicit` применимо только для конструкторов и объявляет явный конструктор. Это значит, что вызов этого конструктора в программе не может быть автоматическим, а будет исключительно явным.

Деструктор класса освобождает выделенный конструктором блок памяти.

Метод `getElem ( )` возвращает ссылку на хранимый в контейнере элемент, имеющий заданный тип `U` и указанный индекс `i`. Чтобы возвратить ссылку, в операторе `return` указатель `(m_p + i)` разыменовывается. Возвращаемый тип ссылки позволяет использовать результат метода `getElem ( )` как с правой стороны оператора присваивания, так и с левой. Это значит, что при помощи метода можно не только получить элемент из контейнера, но и изменить его в контейнере. Если заданный параметром `i` индекс выходит за

допускаемый диапазон изменения индексов (0, m_sz), программа завершится с кодом 1, и управление будет передано операционной системе. Завершение программы производится функцией exit ( ), в вызове которой указан код ошибки 1. В случае завершения с использованием функции exit ( ) автоматического вызова деструкторов не происходит.

Объявление объектов шаблона классов

При объявлении переменных шаблона классов (объектов шаблона) создается конкретная реализация шаблона с типом, указанным в качестве параметра типа (листинг 18.2). Формат объявления имеет следующий вид:

```
имя_шаблона < параметр_типа1, …, параметр_типаN > идентификатор ;
```

Листинг 18.2. Пример объявления объектов шаблона классов

```
TContainer < int > d ( 5 ) ;        // объявление1
TContainer < char* > strP ;         // объявление2
TContainer < record > *p ;          // объявление3
TContainer < CCar* > *pp ;          // объявление4
```

Объявление1 создает реализацию типа TContainer с идентификатором d для хранения пяти целочисленных значений.

Объявление2 создает реализацию strP типа TContainer для хранения десяти указателей типа char*. Размер контейнера strP соответствует значению размера контейнера по умолчанию.

Объявление3 не создает реализацию шаблона, а создает переменную-указатель p на тип TContainer с параметром типа record. Реализованный контейнер может быть создан позже при помощи оператора динамического выделения памяти new. Размер реализуемого контейнера указывается в операторе new.

Объявление4, как и объявление3, не создает реализацию шаблона. Здесь объявляется указатель на тип TContainer указателей типа CCar*. Объявление4 представляет собой комбинацию объявления2 с объявлением3.

Примечание

Типы record и CCar должны быть известны программе.

Пример программы с простым шаблоном

В программе используется шаблон классов `TContainer`, исходный код которого приведен в примере объявления шаблонов, а также разновидности объявлений переменных типа `TContainer` из примера объявления объектов шаблона (листинг 18.3). Дополнительно представлены тип структуры `record` и тип класса `CCar`, объекты которых будут помещаться в реализованные версии класса `TContainer`.

Листинг 18.3. Пример программы с простым шаблоном классов

```
// record.h - спецификация record (мелодии)
#pragma once
struct record
{
       char name [ 100 ] ;   // название мелодии
       double time ;         // время звучания
} ;
// Car.h — Определение CCar (вагоны)
#pragma once
#include <string>
using namespace std ;
class CCar
{
protected:
       string m_model ;      // модель вагона
       double m_mass ;       // масса вагона
public:
       CCar ( ) : m_model ( "" ), m_mass ( 0.0 ) { }
       explicit CCar ( const string& md, const double m )
              : m_model ( md ), m_mass ( m ) { }
       CCar ( const CCar& c )
              : m_model ( c.m_model ), m_mass ( c.m_mass ) { }
       ~CCar ( ) { }
       string getModel ( ) const { return m_model ; }
       double getMass ( ) const { return m_mass ; }
       string getModel ( ) const { return m_model ; }
       double getMass ( ) const { return m_mass ; }
```

```
} ;
// main.cpp - главная функция
#include <iostream>
#include <typeinfo>
#include <iomanip>
using namespace std ;
#include "Container.cpp"     // включение определения шаблона классов
#include "record.h"
#include "Car.h"
int main ( )
{
        // Часть 1
        // Объявление контейнера d из пяти целых чисел
        TContainer < int > d ( 5 ) ;
        // размещение значений в контейнере d
        for ( int j = 0; j < 5; j++ )
                d.getElem ( j ) = ( j + 1 ) * ( j - 1 ) ;
        // вывод содержимого контейнера d
        cout << typeid ( d ).name ( ) << endl ;
        for ( int j = 0; j < 5; j++ )
                cout << d.getElem ( j ) << '\t' ;
        cout << endl << endl ;
        // Часть 2
        // объявление указателя на контейнер типа record
        TContainer < record > *p ;
        // массив структур для инициализации контейнера
        record r [ ] = { { "La Mer", 3.03 }, { "A Paris", 2.21 },
                { "Malaika", 2.46 }, { "Tiko Tiko", 4.42 },
                { "O Sole Mio", 4.51 }, { "Besame Mucho", 3.31 } } ;
        // создание контейнера для хранения шести структур record
        p = new TContainer < record > ( 6 ) ;
        // размещение структур в контейнере
        for ( int j = 0; j < 6; j++ )
        {
                strcpy_s ( p -> getElem ( j ).name, strlen ( r [ j ].name )
↳+ 1, r [ j ].name ) ;
                p -> getElem ( j ).time = r [ j ].time ;
        }
        // вывод содержимого контейнера *p
```

```
cout << typeid ( *p ).name ( ) << endl ;
for ( int j = 0; j < 6; j++ )
{
       cout.width ( 15 ) ; cout.setf ( ios_base::left ) ;
       cout << p -> getElem ( j ).name ;
       cout << p -> getElem ( j ).time << endl ;
}
cout << endl ;
delete p ;     // освобождение памяти
// Часть 3
// Объявления контейнера strP из четырех указателей типа char*
TContainer < char* > strP ( 4 ) ;
// массив строк для инициализации контейнера
char* str [ ] = { "\"Flowers\"", "\"Sea landscape\"",
       "\"Dawn\"", "\"Recollection\"" } ;
// размещение указателей на строки в контейнере strP
for ( int j = 0; j < 4; j++ ) strP.getElem ( j ) = & *str [ j ] ;
// вывод содержимого контейнера strP
cout << typeid ( strP ).name ( ) << endl ;
for ( int j = 0; j < 4; j++ ) cout << strP.getElem ( j ) << '\t' ;
cout << endl << endl ;
// Объявления контейнера carP из трех указателей типа CCar*
TContainer < CCar* > carP ( 3 ) ;
// размещение указателей на вагоны в контейнере carP
carP.getElem ( 0 ) = new CCar ( "15-145", 27.2 ) ;
carP.getElem ( 1 ) = new CCar ( "15-1001", 24.5 ) ;
carP.getElem ( 2 ) = new CCar ( "15-1601", 22.2 ) ;
// вывод содержимого контейнера carP
cout << typeid ( carP ).name ( ) << endl ;
for ( int j = 0; j < 3; j++ )
{
       cout.width ( 15 ) ; cout.setf ( ios_base::left ) ;
       cout << carP.getElem ( j ) -> getModel ( ) ;
       cout << carP.getElem ( j )-> getMass ( ) << endl ;
}
cout << endl ;
// освобождение памяти
for ( int j = 0; j < 3; j++ ) delete carP.getElem ( j ) ;
```

```
        // Часть 4
        // объявление указателя на контейнер указателей типа CCar*
        TContainer < CCar* > *pp ;
        // создание контейнера для хранения трех указателей на тип CCar
        pp = new TContainer < CCar* > ( 3 ) ;
        // размещение объектов в контейнере
        pp -> getElem ( 0 ) = new CCar ( "15-869", 25.3 ) ;
        pp -> getElem ( 1 ) = new CCar ( "15-C862", 24.0 ) ;
        pp -> getElem ( 2 ) = new CCar ( "15-1572", 23.5 ) ;
        // вывод содержимого контейнера
        cout << typeid ( *pp ).name ( ) << endl ;
        for ( int j = 0; j < 3; j++ )
        {
                cout.width ( 15 ) ; cout.setf ( ios_base::left ) ;
                cout << pp -> getElem ( j ) -> getModel ( ) ;
                cout << pp -> getElem ( j ) -> getMass ( ) << endl ;
        }
        // освобождение памяти
        for ( int j = 0; j < 3; j++ ) delete pp -> getElem ( j ) ;
        delete pp ;
        return 0 ;
}
```

Исходный текст файла с главной функцией `main ( )` начинается с включения заголовков стандартных библиотек и заголовочных файлов используемых в программе типов. Файл `Container` шаблона классов `TContainer` имеет расширение `.cpp`. Главная функция включает 4 независимые части, в каждой из которых производятся действия с определенной реализацией типа `TContainer`. Результаты исполнения программы показаны на рис. 18.2.

Часть 1 предназначена для работы с реализацией `d` шаблона `TContainer`, которая представляет собой контейнер для пяти целочисленных значений. Вызов метода `d.getElem ( j )` возвращает ссылку на `j`-й элемент контейнера, следовательно, результат вызова является альтернативным именем элемента. В первом цикле `for` вызов стоит слева от знака `=` и используется для присваивания значений элементам контейнера. Второй цикл `for` выводит состояние контейнера, для чего используется метод `getElem ( )`.

Часть 2 выполняет действия с реализацией шаблона `TContainer`, объявленной при помощи указателя `p`. Для создания реализованного щаблона производится динамическое выделение памяти `new TContainer < record > ( 6 )`.

В контейнер записываются экземпляры структур из массива `r`. Поскольку `p` является указателем, для вызова метода используется оператор доступа стрелка. Чтобы освободить память, занимаемую реализованным шаблоном, в конце части 2 выполняется оператор `delete`.

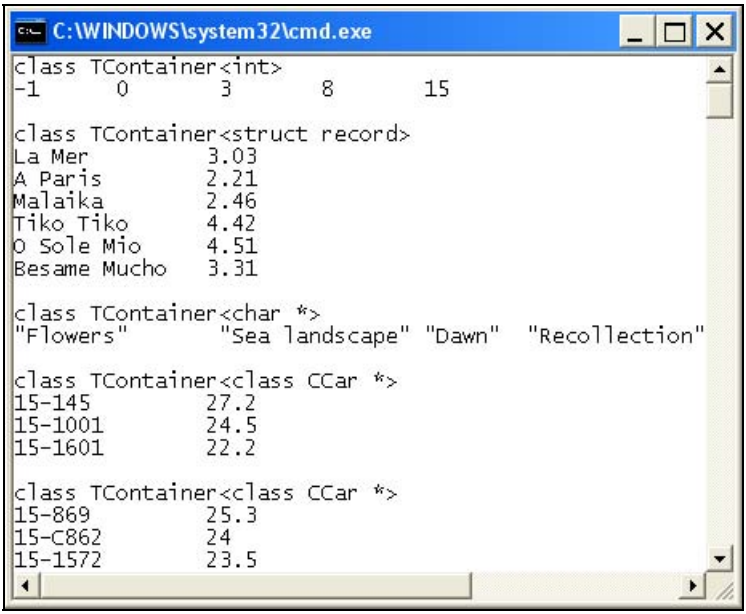


Рис. 18.2. Результаты исполнения программы с простым шаблоном классов

Часть 3 выполняет действия с двумя реализациями шаблона `Tcontainer`. Обе реализации предназначены для хранения указателей, но на объекты разного типа. Первая реализация с идентификатором `strP` использует тип `char*`, производный от стандартного типа `char`. Вторая (с идентификатором `carP`) — тип `CCar*`, производный от авторского типа `CCar`. Чтобы поместить в контейнер `strP` указатели на строки из массива `str`, к каждой строке массива применяется операция взятия адреса указателя `(& *str [ j ])`. Для размещения указателей на объекты типа `CCar` в реализацию `carP` вызывается оператор `new`, который возвращает значение адреса созданного объекта. Когда выводится состояние `carP`, для вызова методов `getModel ( )` и `getMass ( )` класса `CCar` используется оператор `->`, так как в этом случае метод `getElem ( )` возвращает указатель. Выделенная для объектов класса `CCar` память освобождается в конце части 3, где в цикле `for` к хранимым в `carP` указателям применяется оператор `delete`.

Часть 4 показывает, как можно использовать шаблон `TContainer` для создания реализации типа указателей на объекты `CCar*` через указатель `pp` на `TContainer`. В этом случае потребуется динамическое выделение памяти для реализованного шаблона `(pp = new TContainer < CCar* > ( 3 ) ;)`. При выводе состояния созданной реализации шаблона для вызова метода `getElem ( )` теперь потребуется оператор `->`. В конце программы разрушается указатель `pp`, и память, которую занимал реализованный шаблон, становится свободной.

Примечание

В программе применяется оператор `typeid`, определенный в заголовке `<typeinfo>`. Вызов `typeid ( type ).name ( )` при помощи своего операнда `name` возвращает ссылку на `type_info` (информацию о типе), который в вызове представлен параметром `type`. Это дает возможность выводить тип реализованных шаблонов класса `TContainer` в процессе выполнения программы.

Параметры по умолчанию в шаблоне классов

Параметры шаблона могут иметь значения по умолчанию. В этом случае в объявлении шаблона классов в угловых скобках после имени параметра типа, за которым ставится знак =, указывается значение по умолчанию. Например,

```
tepmlate < class U = int, int size = 100 >
class TExample { … } ;
```

Здесь шаблонный класс `TExample` имеет два параметра по умолчанию:

- Параметр типа `U` примет значение `int`, если в объявлении реализации шаблона параметр типа не будет задан.
- Значение константного параметра `size` станет равным 100, если оно не будет указано в объявлении реализации шаблона.

В листинге 18.4 рассматривается версия шаблона `TContainer`, в которой размер контейнера `m_sz` задается параметром по умолчанию.

Листинг 18.4. Пример программы с параметрами по умолчанию в шаблоне классов

```
// Container.cpp - шаблон классов TContainer
// U - параметр типа (задает тип объектов)
// m_sz — размер контейнера (по умолчанию 10)
```

```
// Спецификация TContainer
template < class U, size_t m_sz = 10 >
class TContainer
{
       U *m_p ;              // указатель на элементы контейнера
public:
       explicit TContainer ( ) ;
       ~TContainer ( ) ;
       U& getElem ( int ) ;
} ;
// Реализация TContainer
// Конструктор
template < class U, size_t m_sz >
TContainer < U, m_sz > :: TContainer ( ) { m_p = new U [ m_sz ] ; }
// Деструктор
template < class U, size_t m_sz >
TContainer < U, m_sz > :: ~TContainer ( ) { delete [ ] m_p ; }
// Возврат ссылки на элемент контейнера
template < class U, size_t m_sz >
U& TContainer < U, m_sz > :: getElem ( int n )
{
       if ( n >= 0 && n < m_sz ) return * ( m_p + n ) ;
       exit ( 1 ) ;
}
// main.cpp - главная функция
#include <iostream>
#include <typeinfo>
#include <iomanip>
using namespace std ;
#include "Container.cpp"
#include "Car.h"
int main ( )
{
       // Часть 1
       // Объявление контейнера с из десяти символов
       TContainer < char > c ;
       // размещение значений в контейнере с
       for ( int j = 0; j < 10; j++ ) c.getElem ( j ) = 'q' + j ;
       // вывод содержимого контейнера с
```

```
        cout << typeid ( c ).name ( ) << endl ;
        for ( int j = 0; j < 10; j++ ) cout << c.getElem ( j ) << " " ;
        cout << endl << endl ;
        // Часть 2
        // Объявление контейнера car из трех вагонов
        TContainer < CCar, 3 > car ;
        // массивы для инициализации контейнера
        string md [ ] = { "15-145", "15-1001", "15-1601" } ;
        double m [ ] = { 27.2, 24.5, 22.2 } ;
        // размещение значений в контейнере car
        for ( int j = 0; j < 3; j++ )
        {
                car.getElem ( j ).setModel ( md [ j ] ) ;
                car.getElem ( j ).setMass ( m [ j ] ) ;
        }
        // вывод содержимого контейнера car
        cout << typeid ( car ).name ( ) << endl ;
        for ( int j = 0; j < 3; j++ )
        {
                cout.width ( 15 ) ; cout.setf ( ios_base::left ) ;
                cout << car.getElem ( j ).getModel ( ) ;
                cout << car.getElem ( j ).getMass ( ) << endl ;
        }
        return 0 ;
}
```

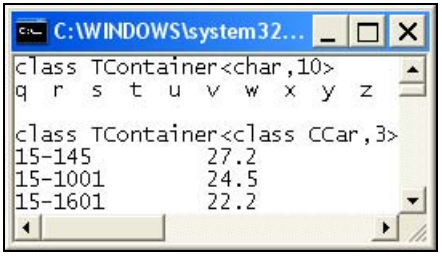


Рис. 18.3. Результаты исполнения программы с параметрами по умолчанию в шаблоне

В листинге 18.4 создается реализация шаблона `c`, размер контейнера в которой будет равен 10, так как в объявлении не указывается значение этого па-

раметра. Реализация `car` будет содержать контейнер из трех объектов типа `CCar` (исходный текст с определением класса есть в предыдущем примере). Результаты исполнения программы показаны на рис. 18.3.

Наследование и шаблоны классов

Шаблон классов может быть базовым для обычных и шаблонных классов. Производный шаблон позволяет отделить операции, производимые с типом, от описания этого типа, которое находится в базовом шаблоне. Чтобы операции правильно выполнялись над объектами типа, необходимо перегружать операторы отношения для элементов базового шаблона, если тип элементов не является встроенным.

Примечание

При раздельной компиляции файл базового класса необходимо включить в файл производного класса при помощи директивы препроцессора `#include`.

Листинг 18.5. Пример программы с производным шаблонным классом

```
// Container.cpp - базовый шаблон классов TContainer
// U - параметр типа (задает тип объектов)
// Спецификация TContainer
template < class U >
class TContainer
{
protected:
       U *m_p ;                    // указатель на элементы контейнера
       size_t m_sz ;               // размер контейнера
       size_t m_n ;                // количество элементов в контейнере
       enum { EMPTY = 0 } ;
private:
       bool isFull ( ) const ;     // проверить на полноту
       bool isEmpty ( ) const ;    // проверить на пустоту
public:
       explicit TContainer ( size_t = 10 ) ;
       virtual ~TContainer ( ) ;
       size_t getN ( ) const ;     // возвратить количество элементов
       U& getElem ( int ) ;        // возвратить ссылку на элемент
       bool push ( const U& ) ;    // затолкнуть элемент
```

```
        U* pop ( ) ;                    // вытолкнуть элемент
} ;
// Реализация TContainer
// Конструктор
template < class U >
TContainer < U > :: TContainer ( size_t n )
: m_sz ( n ), m_n ( EMPTY )
{       m_p = new U [ m_sz ] ;        }
// Деструктор
template < class U >
TContainer < U > :: ~TContainer ( )
{       delete [ ] m_p ;       }
// Возврат ссылки на элемент контейнера
// Прототип:  U& getElem ( int n ) ;
// Параметры: n - индекс элемента в контейнере.
// Описание:
//      Возвращает ссылку элемента типа U с заданным индексом.
//      Если индекс находится вне допустимого диапазона,
//      вызывается функция exit ( ), программа завершается с кодом 1.
template < class U >
U& TContainer < U > :: getElem ( int n )
{
        if ( n >= 0 && n < m_sz )
                return * ( m_p + n ) ;
        exit ( 1 ) ;
}
// Возврат количества элементов
// Прототип:  size_t getN ( ) const ;
// Описание: Возвращает количество хранимых в контейнере элементов.
template < class U >
size_t TContainer < U > :: getN ( ) const
{       return m_n ;   }
// Проверка контейнера на полноту
// Прототип:  bool isFull ( ) const ;
// Описание:
//      Возвращает истину, если число элементов равно размеру контейнера.
//      Иначе возвращает ложь.
template < class U >
bool TContainer < U > :: isFull ( ) const
```

```
{
        if ( m_n == m_sz ) return true ;
        return false ;
}
// Проверка контейнера на пустоту
// Прототип:  bool isEmpty ( ) const ;
// Описание:
//      Возвращает истину, если количество элементов равно 0.
//      Иначе возвращает ложь.
template < class U >
bool TContainer < U > :: isEmpty ( ) const
{
        if ( m_n == EMPTY ) return true ;
        return false ;
}
// Заталкивание элемента в контейнер
// Прототип:  bool push ( const U& elem ) ;
// Параметры: elem - ссылка на добавляемый элемент
// Описание:
//      Проверяет контейнер на полноту. Если он полон, возвращает ложь.
//      Иначе добавляет elem в конец контейнера.
//      Увеличивает количество элементов m_n на единицу.
//      Возвращает истину.
template < class U >
bool TContainer < U > :: push ( const U& elem )
{
        if ( isFull ( ) ) return false ;
        m_p [ m_n++ ] = elem ; return true ;
}
// Выталкивание последнего элемента из контейнера
// Прототип:  U* pop ( ) ;
// Описание:
//      Проверяет контейнер на пустоту. Если он пуст, возвращает 0.
//      Иначе уменьшает количество элементов m_n на единицу.
//      Возвращает адрес вытолкнутого элемента.
template < class U >
U* TContainer < U > :: pop ( )
{
        if ( m_n == EMPTY ) return reinterpret_cast < U* > ( 0 ) ;
```

```
        return & m_p [ --m_n ] ;
}
// Utility.cpp - производный шаблон классов TUtility
// U - параметр типа (задает тип объектов)
// Реализует операции над элементами контейнера
#include "Container.cpp"     // включение определения базового класса
// Спецификация TUtility
template < class U >
class TUtility : public TContainer < U >
{
public:
        U& getMin ( ) const ;         // поиск минимального элемента
        U& getMax ( ) const ;         // поиск максимального элемента
        void sort ( ) ;               // сортировка элементов
        // конструктор и деструктор производного класса
        explicit TUtility ( size_t sz ) : TContainer < U > ( sz ) { }
        ~TUtility ( ) { }
} ;
// Реализация TUtility
// Поиск минимального элемента
// Прототип:  U& getMin ( ) const ;
// Описание: Возвращает ссылку на минимальный элемент контейнера.
template < class U >
U& TUtility < U > :: getMin ( ) const
{
        int iMin = 0 ;
        for ( int i = 0; i < m_n; i++ )
                if ( m_p [ i ] < m_p [ iMin ] ) iMin = i ;
        return * ( m_p + iMin ) ;
}
// Поиск максимального элемента
// Прототип:  U& getMax ( ) const ;
// Описание: Возвращает ссылку на максимальный элемент контейнера.
template < class U >
U& TUtility < U > :: getMax ( ) const
{
        int iMax = 0 ;
        for ( int i = 0; i < m_n; i++ )
                if ( m_p [ i ] > m_p [ iMax ] ) iMax = i ;
```

```
        return * ( m_p + iMax ) ;
}
// Сортировка элементов
// Прототип:  void sort ( ) ;
// Описание:
// Упорядочивает элементы по возрастанию методом пузырьковой сортировки.
template < class U >
void TUtility < U > :: sort ( )
{
        for ( int j = 1; j < m_n; j++ )
                for ( int i = 0; i < m_n - 1; i++ )
                        if ( m_p [ i ] > m_p [ i + 1 ] )
                        {
                                U tmp = m_p [ i ] ;
                                m_p [ i ] = m_p [ i + 1 ] ;
                                m_p [ i + 1 ] = tmp ;
                        }
}
// main.cpp - главная функция
#include <iostream>
#include <ctime>
using namespace std ;
#include "Utility.cpp"
int main ( )
{
        // Объявление контейнера iCon из 5 целых чисел
        TUtility < int > iCon ( 5 ) ;
        while ( iCon.push ( rand ( ) ) )
                ;
        // вывод содержимого контейнера
        cout << "\nView iCon\n" ;
        size_t n = iCon.getN ( ) ;
        for ( int j = 0; j < n; j++ ) cout << iCon.getElem ( j ) << '\t'
;
        // вывод минимального и максимального элементов
        cout << "\n\nMin:\t" << iCon.getMin ( ) ;
        cout << "\nMax:\t" << iCon.getMax ( ) ;
        // сортировка элементов
        iCon.sort ( ) ;
```

```
        cout << "\n\nView iCon after sorting\n" ;
        for ( int j = 0; j < n; j++ ) cout << iCon.getElem ( j ) << '\t'
;
        // выталкивание элементов из отсортированного контейнера
        cout << "\n\nPop from sorted container iCon\n" ;
        for ( int j = 0; j < n; j++ ) cout << * iCon.pop ( ) << '\t' ;
        cout << endl ;
        return 0 ;
}
```

В листинге 18.5 предлагается усовершенствованная версия шаблона классов `TContainer`, который является базовым для шаблона классов `TUtility`. Деструктор базового класса `~TContainer ( )` объявлен виртуальным. Конструктор производного класса `TUtility ( )` получает необходимый для конструирования объектов параметр размера контейнера и явно вызывает конструктор базового класса. Методы классов подробно комментируются в тексте программы. Результаты выполнения показаны на рис. 18.4.

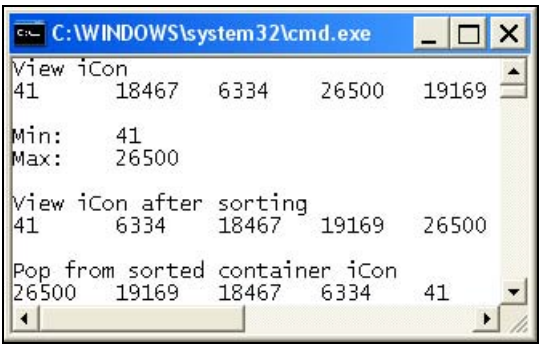


Рис. 18.4. Результаты выполнения программы с производным шаблонным классом

В главной функции повторяется код вывода состояния контейнера:

```
for ( int j = 0; j < n; j++ ) cout << iCon.getElem ( j ) << '\t' ;
```

Проблему можно решить при помощи глобальной шаблонной функции (листинг 18.6).

Листинг 18.6. Шаблонная функция просмотра состояния контейнера

```
template < class U >
void view ( TUtility < U >* p )
{
```

```
        for ( int j = 0; j < p -> getN ( ); j++ )
                cout << p -> getElem ( j ) ;
        cout << endl ;
}
```

Параметром функции `view ( )` является указатель на шаблонный тип. Чтобы получить значения защищенных членов объекта, используются соответствующие открытые методы класса: `getN ( )`, возвращающий фактическое количество элементов, и `getElem ( )` — ссылку на элемент. В вызове `view ( )` указывается адрес объекта, состояние которого необходимо просмотреть. Для примера с производным шаблонным классом вызов функции будет таким:

```
view ( &iCon ) ;
```

Использование шаблонов

В этом разделе рассматривается пример программы, в которой реализовано совместное использование шаблонов классов и шаблонов функций. Программа позволяет создавать и просматривать состояние контейнеров для элементов типа `double`, а также типа

```
class CCar { protected: string m_model ; double m_mass ; },
```

где `m_model` — модель вагона, `m_mass` — масса вагона.

К элементам контейнеров могут применяться следующие операции:

- добавление уникального элемента в конец контейнера (элементы контейнера не могут дублироваться);
- проверка по ключу. Ключом для базового типа `double` является конкретное значение; для типа `record` — название мелодии, для типа `CCar` — модель вагона;
- сортировка по возрастанию. Для типов `record` и `CCar` элементы сортируются по значению ключа.

В программе предусмотрено меню для выбора типа элементов и активизации операций над элементами контейнера. Функциональность реализуется при помощи классов, которые представлены схемой взаимосвязи классов (или диаграммой классов, в терминологии авторов книг по UML) на рис. 18.5.

На диаграмме видно, что с отношением агрегации связаны объекты классов `CMenu` и `CCtrl`. Класс `CCtrl` является производным от класса `CMaker`. Шаблон классов `TContainer < U >` является базовым для шаблона `TUtility < U >`, который имеет две специализации: `TUtility < double >` и `TUtility < CCar >`.

Специализация `TUtility < CCar >` в качестве параметра `U` получает объекты класса `CCar`. Класс `CCtrl` находится в отношении агрегации со специализациями `TUtility < double >` и `TUtility < CCar >`.

В табл. 18.1 дается описание назначения классов программы.

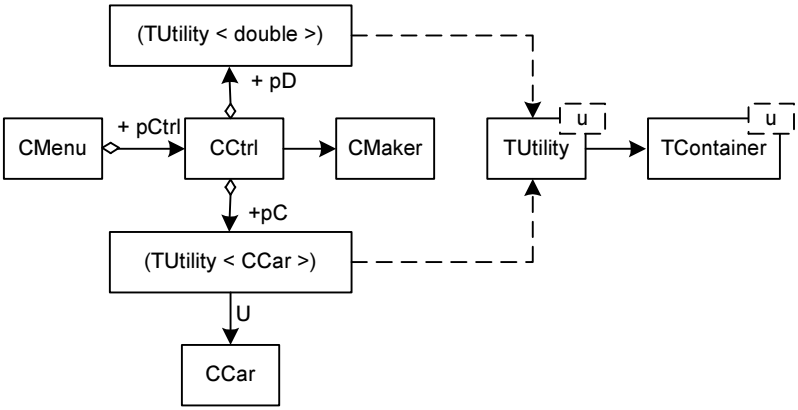


Рис. 18.5. Диаграмма классов программы с использованием шаблонов

Таблица 18.1. *Назначение классов*

Название класса	Назначение
`CMenu`	Класс пользовательского интерфейса. Организует взаимодействие с пользователем программы
`CMaker`	Абстрактный класс. Локализует место создания специализаций контейнеров
`CCtrl`	Класс управления. Координирует действия классов
`TContainer < U >`	Базовый шаблон контейнерных классов для хранения объектов разных типов. Обеспечивает основные операции с объектами контейнера
`TUtility < U >`	Производный шаблон контейнерных классов. Обеспечивает дополнительные операции с объектами контейнера
`TUtility < double >`	Специализация шаблона `TUtility < U >` для типа `double`
`TUtility < CCar >`	Специализация шаблона `TUtility < U >` для типа `CCar`
`CCar`	Класс сущностей. Описывает железнодорожные вагоны

Описание членов-данных классов представлено в табл. 18.2.

Таблица 18.2. *Описание членов-данных классов*

Название класса	Члены-данные класса		
	название	**тип**	**описание**
CMenu	type	int	Тип данных, выбранный пользователем
	pCtrl	CCtrl*	Указатель на объект управления
	{DOUBLE, CCAR}	enum	Константы, определяющие выбор пользователя (DOUBLE для типа double, CCAR для типа CCar)
CCtrl	pD	TUtility <double> *	Указатель на объект специализации контейнера с типом элементов double
	pC	TUtility <CCar> *	Указатель на объект специализации контейнера с типом элементов CCar
TContainer < U >	m_p	U *	Указатель на элементы контейнера
	m_sz	size_t	Размер контейнера
	m_n	size_t	Количество элементов в контейнере
	{EMPTY}	enum	Константа EMPTY определяет, что в контейнере нет ни одного элемента

В табл. 18.3–18.8 представлены описания методов классов программы.

Таблица 18.3. *Описание методов класса* `CMenu`

Название метода	Назначение	Тип
CMenu ( )	Создает объект класса. Выделяет память для объекта управления	нет
~CMenu ( )	Освобождает выделенную конструктором память	нет
push ( )	Позволяет добавить значение в конец контейнера	void
find ( )	Позволяет проверить наличие значения ключа в контейнере	void
sort ( )	Позволяет сортировать элементы контейнера по возрастанию	void
show ( )	Показывает состояние контейнера	void
choiceType ( )	Выбирает тип элементов контейнера	int
create ( )	Позволяет создать контейнер с инициализацией	void

Таблица 18.4. *Описание методов класса* `CMaker`

Метод			Параметры	
название	**назначение**	**тип**	**тип**	**назначение**
`create ( )`	Чистая виртуальная функция для создания с инициализацией специализации контейнера типа `double`	`TUtility <double>*`	`const double*`	Указатель на массив инициализации
			`const size_t&`	Размер контейнера
			`const size_t&`	Количество элементов инициализации
`create ( )`	Чистая виртуальная функция для создания с инициализацией специализации контейнера типа `CCar`	`TUtility <CCar>*`	`const double*`	Указатель на массив инициализации
			`const size_t&`	Размер контейнера
			`const size_t&`	Количество элементов инициализации

Таблица 18.5. *Описание методов класса* `CCtrl`

Метод			Параметры	
название	**назначение**	**тип**	**тип**	**назначение**
`CCtrl ( )`	Создает объект управления. Инициализирует нулевыми значениями указатели `pD`, `pC`	нет	–	–
`~CCtrl ( )`	Освобождает память, занятую указателями `pD`, `pC`	нет	–	–
`create ( )`	Активизирует создание специализации контейнера типа `double` и добавление полученных значений	`TUtility <double>*`	`const double*`	Указатель на массив инициализации
			`const size_t&`	Размер контейнера
			`const size_t&`	Количество элементов инициализации

Таблица 18.5 *(окончание)*

Метод			Параметры	
название	**назначение**	**тип**	**тип**	**назначение**
`create ( )`	Активизирует создание специализации контейнера типа `CCar` и добавление полученных значений	`TUtility <CCar>*`	`const double*`	Указатель на массив инициализации
			`const size_t&`	Размер контейнера
			`const size_t&`	Количество элементов инициализации
`push ( )`	Активизирует добавление указанного значения в специализацию контейнера типа `double`	`bool`	`const double&`	Добавляемое значение
`push ( )`	Активизирует добавление указанного значения в специализацию контейнера типа `CCar`	`bool`	`const string&`	Марка вагона
			`const double&`	Масса вагона
`find ( )`	Активизирует проверку значения ключа в специализации контейнера типа `double`	`bool`	`const double&`	Искомое значение
`find ( )`	Активизирует проверку значения ключа в специализации контейнера типа `CCar`	`bool`	`const string&`	Марка вагона

Таблица 18.6. *Описание методов класса* `TContainer`

Метод			Параметры	
Название	**назначение**	**тип**	**тип**	**назначение**
`TContainer ( )`	Создает контейнер указанного типа, выделяя память для заданного числа элементов	нет	`size_t`	Размер контейнера. По умолчанию равен 10
`~TContainer ( )`	Освобождает память, выделенную конструктором	нет	–	–
`getN ( )`	Возвращает фактическое количество элементов	`size_t`	–	–
`isFull ( )`	Проверяет контейнер на полноту	`bool`	–	–
`isEmpty ( )`	Проверяет контейнер на пустоту	`bool`	–	–
`getElem ( )`	Возвращает ссылку на элемент с заданным индексом	`U&`	`unsigned int`	Индекс элемента в контейнере
`push ( )`	Заталкивает указанное значение в контейнер	`bool`	`const U&`	Заталкиваемое значение
`pop ( )`	Выталкивает последний элемент из контейнера	`U*`	–	–

Таблица 18.7. *Описание методов класса* `TUtility`

Метод			Параметры	
Название	**назначение**	**тип**	**тип**	**назначение**
`TUtility ( )`	Создает контейнер заданного типа, вызывая конструктор базового класса `TContainer ( )`	нет	`size_t`	Размер контейнера
`~TUtility ( )`	Ничего не делает	нет	–	–

Таблица 18.7 *(окончание)*

Метод			Параметры	
Название	**назначение**	**тип**	**тип**	**назначение**
`sort ( )`	Упорядочивает элементы контейнера по возрастанию методом пузырьковой сортироки	`void`	–	–
`operator - ( )`	Удаляет из контейнера повторяющиеся значения	`void`	–	–
`operator + ( )`	Добавляет указанное значение в контейнер, если элемента с таким значением в нем нет	`bool`	`const U&`	Добавляемое значение
`operator == ( )`	Проверяет наличие элемента с заданным значением в контейнере	`bool`	`const U&`	Искомое значение
`delRetry ( )`	Удаляет повторения заданного значения в контейнере	`void`	`const U&`	Значение, повторения которого необходимо исключить из контейнера

Таблица 18.8. *Описание методов класса* `CCar`

Метод			Параметры	
Название	**назначение**	**тип**	**тип**	**назначение**
`CCar ( )`	Конструирует объект по умолчанию	нет	`size_t`	Размер контейнера
`CCar ( )`	Конструирует объект с заданными параметрами	нет	`const string&`	Модель вагона
			`const double`	Масса вагона
`~CCar ( )`	Ничего не делает	нет	–	–
`getModel ( )`	Возвращает модель вагона	`string`	–	–
`getMass ( )`	Возвращает массу вагона	`double`	–	–

Таблица 18.8 *(окончание)*

Метод			Параметры	
Название	**назначение**	**тип**	**тип**	**назначение**
`setModel ( )`	Устанавливает модель вагона	`void`	`const string&`	Модель вагона
`setMass ( )`	Устанавливает массу вагона	`void`	`const double&`	Масса вагона
`operator = ( )`	Перегружает присваивание	`const CCar&`	`const CCar&`	Вагон справа от знака `=`
`operator == ( )`	Сравнивает вагоны на равенство моделей	`bool`	`const CCar&`	Вагон справа от знака `==`
`operator != ( )`	Сравнивает вагоны на неравенство моделей	`bool`	`const CCar&`	Вагон справа от знака `!=`
`operator > ( )`	Перегружает отношение `>` для моделей вагонов в лексикографическом порядке	`bool`	`const CCar&`	Вагон справа от знака `>`

В листингах 18.7–18.15 приводится исходный текст программы.

Листинг 18.7. Главная программа `main.cpp`

```
#include "Menu.h"
int main ( )
{
       CMenu menu ; menu.showMenuAndSelect ( ) ;
       return 0 ;
}
```

Листинг 18.8. `Menu.h` — спецификация `CMenu`

```
#pragma once
#include "Ctrl.h"
class CCar ;
class CMenu
{
public:
```

```
        enum { DOUBLE, CCAR } ;
        int type ;
        CCtrl* pCtrl ;
        CMenu ( ) ; ~CMenu ( ) ;
        void showMenuAndSelect ( ) ;
private:
        void push ( ) const ;
        void find ( ) const ;
        void sort ( ) const ;
        void show ( ) const ;
        int choiceType ( ) ;
        void create ( ) const ;
} ;
```

Листинг 18.9. `Menu.cpp` — реализация `CMenu`

```
#include <iostream>
#include <string>
using namespace std ;
#include "Menu.h"
#include "View.cpp"
#include "Car.h"
// Конструктор и деструктор
CMenu :: CMenu ( ) { pCtrl = new CCtrl ; }
CMenu :: ~CMenu ( ) { delete pCtrl ; }
// Активизация меню
void CMenu :: showMenuAndSelect ( )
{
        char* menu [5] = {"Push", "Find", "Sort", "Show", "Other type"} ;
        while ( true )
        {
                type = choiceType ( ) ;
                if ( type < 0 ) break ;
                create ( ) ;
                char ch ;
                do
                {
                        for ( int i = 0; i < 4; i++ )
                                cout << '\n' << i + 1 << '\t' << menu [ i ] ;
```

```
                        cout << "\n0\t" << menu [ 4 ] << endl << endl ;
                        cout << "\nPlease, your choice -> " ;
                        cin >> ch ; cin.ignore ( ) ;
                        switch ( ch )
                        {
                        case '1': push ( ) ; break ;
                        case '2': find ( ) ; break ;
                        case '3': sort ( ) ; break ;
                        case '4': show ( ) ; break ;
                        }
                } while ( ch != '0' ) ;
        }
}
// Выбор типа данных
int CMenu :: choiceType ( )
{
        char* t [ 2 ] = { "double", "CCar" } ;
        type = -1 ;
        char ch ;
        do
        {
                for ( int i = 0; i < 2; i++ )
                        cout << i + 1 << " -> " << t [ i ] << '\t' ;
                cout << "0 -> Exit\n" ;
                cout << "\nPlease, your choice -> " ;
                cin >> ch ; cin.ignore ( ) ;
                switch ( ch )
                {
                // назначение типа
                case '1': type = DOUBLE ;    break ;
                case '2': type = CCAR ;      break ;
                }
        } while ( ch < '0' || ch > '2' ) ;
        return type ;
}
// Создание контейнера в соответствии с выбранным типом
void CMenu :: create ( ) const
{
        // ввод размера контейнера
```

```
        cout << "Enter size of container -> " ;
        size_t sz ; cin >> sz; cin.ignore ( ) ;
        switch ( type )       // анализ типа
        {
        case 0:
                {
                        // массив для инициализации контейнера
                        double d [ ] = { 1.1782, 4.16, -376.28, -376.28,
↳100.4 } ;
                        size_t n = sizeof ( d ) / sizeof ( d [ 0 ] ) ;
                        // создание контейнера
                        pCtrl -> create ( d, sz, n ) ;
                        break ;
                }
        case 1:
                {
                        // массив для инициализации контейнера
                        CCar car [ 3 ] ;
                        car [ 0 ] = CCar ( "15-869", 25.3 ) ;
                        car [ 1 ] = CCar ( "15-C862", 24.0 ) ;
                        car [ 2 ] = CCar ( "15-1572", 23.5 ) ;
                        // создание контейнера
                        pCtrl -> create ( car, sz, 3 ) ;
                        break ;
                }
        }
}
// Добавление элементов в контейнер
void CMenu :: push ( ) const
{
        char* err = "\nAddition is impossible!\n" ; bool res ;
        switch ( type )
        {
        case 0:
                {
                        // ввод значения
                        double key ;
                        cout << "Enter double-precision number -> " ;
                        cin >> key ; cin.ignore ( ) ;
```

```
                        // добавление в контейнер
                        res = pCtrl -> push ( key ) ;
                        break ;
                }
        case 1:
                {
                        // ввод модели вагона
                        string key ; cout << "Enter car model -> " ;
                        getline ( cin, key ) ;
                        // ввод массы вагона
                        double mass ; cout << "Enter car mass -> " ;
                        cin >> mass ; cin.ignore ( ) ;
                        // добавление в контейнер
                        res = pCtrl -> push ( key, mass ) ;
                        break ;
                }
        }
        if ( ! res ) cout << err ;
}
// Проверка наличия значения ключа в контейнере
void CMenu :: find ( ) const
{
        char* yes = "\tIs in container\n" ;
        char* not = "\tIs not in container\n" ; int res ;
        switch ( type )
        {
        case 0:
                {
                        double key ;
                        cout << "Enter double-precision number -> " ;
                        cin >> key ; cin.ignore ( ) ;
                        res = pCtrl -> find ( key ) ;
                        break ;
                }
        case 1:
                {
                        string key ; cout << "Enter car model -> " ;
                        getline ( cin, key ) ;
                        res = pCtrl -> find ( key ) ;
```

```
                    break ;
                }
        }
        if ( res ) cout << yes ;
        else cout << not ;
}
// Сортировка элементов кнтейнера
void CMenu :: sort ( ) const
{
        switch ( type )
        {
        case 0:
                if ( pCtrl -> pD ) pCtrl -> pD -> sort ( ) ;
                break ;
        case 1:
                if ( pCtrl -> pC ) pCtrl -> pC -> sort ( ) ;
                break ;
        }
}
// Вывод состояния контейнера
void CMenu :: show ( ) const
{
        switch ( type )
        {
        case 0:
                if ( pCtrl -> pD ) view ( * pCtrl -> pD ) ;
                break ;
        case 1:
                if ( pCtrl -> pC ) view ( * pCtrl -> pC ) ;
                break ;
        }
}
```

Листинг 18.10. `Maker.h` — абстрактный класс `CMaker`

```
#pragma once
#include "Utility.cpp"
class CCar ;
class CMaker
```

```
{
public:
        // чистые виртуальные функции
        virtual TUtility < double >*
        create ( const double*, const size_t&, const size_t& ) = 0 ;
        virtual TUtility < CCar >*
        create ( const CCar*, const size_t&, const size_t& ) = 0 ;
} ;
```

Листинг 18.11. `Ctrl.h` — спецификация `CCtrl`

```
#pragma once
#include <string>
using namespace std ;
#include "Maker.h"
class CCar ;
class CCtrl : public CMaker
{
public:
        TUtility < double > *pD ;
        TUtility < CCar > *pC ;
        CCtrl ( ) ;
        ~CCtrl ( ) ;
        TUtility < double >*
        create ( const double*, const size_t&, const size_t& ) ;
        TUtility < CCar >*
        create ( const CCar*, const size_t&, const size_t& ) ;
        bool push ( const double& ) ;
        bool push ( const string&, const double& ) ;
        bool find ( const double& ) ;
        bool find ( const string& ) ;
} ;
```

Листинг 18.12. `Ctrl.cpp` — реализация `CCtrl`

```
#include "Ctrl.h"
#include "Car.h"
// Конструктор
CCtrl :: CCtrl ( ) : pD ( 0 ), pC ( 0 ) { }
// Деструктор
```

```
CCtrl :: ~CCtrl ( )
{      if ( ! pD ) delete pD ; if ( ! pC ) delete pC ;   }
// Создание контейнера типа double с одновременной инициализацией
TUtility < double >*
CCtrl :: create ( const double* p, const size_t& sz, const size_t& n )
{
       pD = new TUtility < double > ( sz ) ;
       for ( unsigned int j = 0; j < n; j++ ) * pD + p [ j ] ;
       return pD ;
}
// Создание контейнера типа CCar с одновременной инициализацией
TUtility < CCar >*
CCtrl :: create ( const CCar* p, const size_t& sz, const size_t& n )
{
       pC = new TUtility < CCar > ( sz ) ;
       for ( unsigned int j = 0; j < n; j++ ) * pC + p [ j ] ;
       return pC ;
}
// Добавление значения в контейнер double
// Если контейнер полон, возвращает ложь.
// Иначе возвращает результат оператора добавления в контейнер.
bool CCtrl :: push ( const double& key )
{
       if ( pD -> isFull ( ) ) return false ;
       return * pD + key ;
}
// Добавление значения в контейнер типа CCar
// Если контейнер полон, возвращает ложь.
// Иначе создает объект типа CCar.
// Возвращает результат оператора добавления в контейнер.
bool CCtrl :: push ( const string& model, const double& mass )
{
       if ( pC -> isFull ( ) ) return false ;

       CCar car ( model, mass ) ; return * pC + car ;
}
// Проверка ключевого значения в контейнере типа double
// Возвращает результат оператора отношения равенства.
bool CCtrl :: find ( const double& key )
```

```
{       return ( *pD == key ) ;       }
// Проверка ключевого значения в контейнере типа CCar
// Создает объект типа CCar.
// Возвращает результат оператора отношения равенства.
bool CCtrl :: find ( const string& key )
{       CCar car ( key, 0.0 ) ; return ( *pC == car ) ;    }
```

Листинг 18.13. `Container.cpp` — базовый шаблон классов `TContainer`

```
// Спецификация TContainer
#pragma once
template < class U > class TContainer
{
protected:
       U *m_p ; size_t m_sz ; size_t m_n ; enum { EMPTY } ;
public:
       explicit TContainer ( size_t = 10 ) ; virtual ~TContainer ( ) ;
       size_t getN ( ) const ; U& getElem ( unsigned int ) ;
       bool isFull ( ) const ; bool isEmpty ( ) const ;
       bool push ( const U& ) ; U* pop ( ) ;
} ;
// Реализация TContainer
// Конструктор
template < class U >
TContainer < U > :: TContainer ( size_t n )
: m_sz ( n ), m_n ( EMPTY ) { m_p = new U [ m_sz ] ; }
// Деструктор
template < class U >
TContainer < U > :: ~TContainer ( ) { delete [ ] m_p ; }
// Возврат ссылки на элемент контейнера
template < class U > U& TContainer < U > :: getElem ( unsigned int n )
{
       if ( n >= 0 && n < m_sz ) return * ( m_p + n ) ;
       exit ( 1 ) ;
}
// Возврат количества элементов
template < class U > size_t TContainer < U > :: getN ( ) const
{       return m_n ;   }
// Проверка контейнера на полноту
```

```
template < class U > bool TContainer < U > :: isFull ( ) const
{
       if ( m_n == m_sz ) return true ;
       return false ;
}
// Проверка контейнера на пустоту
template < class U > bool TContainer < U > :: isEmpty ( ) const
{
       if ( m_n == EMPTY ) return true ;
       return false ;
}
// Заталкивание элемента в контейнер
template < class U > bool TContainer < U > :: push ( const U& elem )
{
       if ( isFull ( ) ) return false ;
       m_p [ m_n++ ] = elem ;
       return true ;
}
// Выталкивание последнего элемента из контейнера
template < class U > U* TContainer < U > :: pop ( )
{
       if ( m_n == EMPTY ) return reinterpret_cast < U* > ( 0 ) ;
       return & m_p [ --m_n ] ;
}
```

Листинг 18.14. `Utility.cpp` — производный шаблон классов `TUtility`

```
// Спецификация TUtility
#pragma once
#include "Container.cpp"
struct record ;
class CCar ;
template < class U > class TUtility : public TContainer < U >
{
public:
       explicit TUtility ( size_t sz ) : TContainer < U > ( sz ) { }
       ~TUtility ( ) { }
       void sort ( ) ; void operator - ( ) ;
       bool operator + ( const U& ) ; bool operator == ( const U& ) ;
```

```
private:
       void delRetry ( const U& ) ;
} ;
// Реализация TUtility
// Сортировка элементов по возрастанию
template < class U > void TUtility < U > :: sort ( )
{
       for ( unsigned int j = 1; j < m_n; j++ )
              for ( int unsigned i = 0; i < m_n - 1; i++ )
                     if ( m_p [ i ] > m_p [ i + 1 ] )
                     {
                            U tmp = m_p [ i ] ;
                            m_p [ i ] = m_p [ i + 1 ] ;
                            m_p [ i + 1 ] = tmp ;
                     }
}
// Удаление повторений элемента
// Прототип:  void delRetry ( const U& r )
// Параметр:  &r - ссылка на значение, повторения которого удаляются.
// Описание:
//     Изменяет состояние контейнера,
//     если в нем есть повторения значений, указанных параметром r.
//     Если нет повторений значения r или вообще нет такого значения,
//     состояние не изменится.
template < class U > void TUtility < U > :: delRetry ( const U& r )
{
       size_t n ( m_n ), j ( 0 ) ;
       unsigned int i ;
       for ( i = 0; i < n && ! ( m_p [ i ] == r ); i++ )
              ;
       j = ++i ;
       while ( i < m_n )
       {
              if ( ! ( m_p [ i ] == r ) )  m_p [ j++ ] = m_p [ i ] ;
              i++ ;
       }
       m_n = j ;
}
// Перегрузка оператора - (унарный минус)
```

```
// Описание:
//      Оставляет в контейнере уникальные элементы.
// Вызываемые методы: закрытый метод delRetry ( )
template < class U > void TUtility < U > :: operator - ( )
{
        unsigned int i = 0 ;
        while ( i < m_n )
        {       U elem = m_p [ i ] ;  delRetry ( elem ) ; i++ ;    }
}
// Перегрузка оператора + (сложение)
// Параметр:  r - ссылка на элемент типа U
// Описание:  Добавляет элемент, если такого нет в контейнере.
//      Если элемента нет, вызывает метод push ( ) базового класса,
//      возвращает значение, полученное от push ( ).
//      Иначе возвращает ложь.
template < class U > bool TUtility < U > :: operator + ( const U& r )
{
        unsigned int i ( 0 ) ;
        while ( i < m_n && m_p [ i ] != r ) i++ ;
        if ( i == m_n ) return push ( r ) ;
        return false ;
}
// Перегрузка оператора == (отношение равенства)
// Параметр:  key - ссылка на элемент типа U
// Описание:  Ищет индекс элемента контейнера, не равного значению key.
//      Если индекс меньше количества элементов, возвращает истину.
//      Иначе возвращает ложь.
template < class U > bool TUtility < U > :: operator == ( const U& key )
{
        unsigned int i ( 0 ) ;
        while ( i < m_n && ( m_p [ i ] != key ) ) i++ ;
        if ( i < m_n ) return true ;
        return false ;
}
```

Листинг 18.15. `View.cpp` — глобальные функции вывода состояния

```
#include <iostream>
#include <string>
#include <iomanip>
```

```
using namespace std ;
#include "Ctrl.h"
#include "Car.h"
// Вывод состояния контейнера типа TUtility < U >
template < class U >
static void view ( TUtility < U >& p )
{
       unsigned int n = 5 ;
       for ( unsigned int j = 0; j < p.getN ( ); j++ )
       {
              for ( unsigned int k = 0; k < n && ( j + k ) < p.getN ( );
↳k++ )
                     cout << p.getElem ( j + k ) << '\t' ;
              j += n - 1 ; cout << endl ;
       }
}
// Вывод состояния объекта типа CCar
static void view ( const CCar& p )
{
       cout.width ( 15 ) ; cout.setf ( ios_base::left ) ;
       cout << p.getModel ( ) ; cout << p.getMass ( ) << endl ;
}
// Вывод состояния специализации TUtility < CCar >
template < >
static void view ( TUtility < CCar >& p )
{
       int n = 5 ;
       for ( unsigned int j = 0; j < p.getN ( ); j++ )
              view ( p.getElem ( j ) ) ;
}
```

Выполнение программы начинается с создания объекта `menu` в главной функции `main ( )`. При помощи этого объекта активизируется меню программы, и ее дальнейшее поведение зависит от действий пользователя.

Сначала предлагается выбрать тип данных, которые будут размещаться в контейнере. Данное `type` объекта `menu` получает значение типа элементов контейнера в соответствии с выбором пользователя. Установка значения `type` происходит в методе `choiceType ( )`.

После выбора типа элементов контейнера пользователю предлагается выбрать пункт меню с операциями над его элементами. Поведение объектов

программы координируется объектом управления, который создан при конструировании объекта `menu` через указатель `pCtrl`. Этот объект управления в зависимости от выбора операции пользователем будет вызывать методы специализаций контейнеров `TUtility < double >` или `TUtility < CCar >`, чтобы они выполнили необходимые действия и вернули результат преобразований. Полученный результат передается объектом управления обратно объекту `menu`, который продолжает диалог с пользователем.

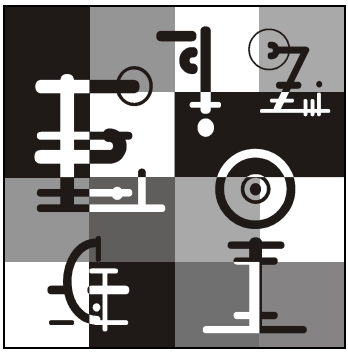
Программа завершает свою работу, когда пользователь отказывается от выбора типа элементов контейнера. Протокол работы программы приведен в листинге 18.16.

Листинг 18.16. Протокол работы программы с использованием шаблонов

```
1 -> double 2 -> CCar  0 -> Exit
Please, your choice -> 1
Enter size of container -> 10
1  Push
2  Find
3  Sort
4  Show
0  Other type
Please, your choice -> 4
1.1782 4.16 -376.28 100.4
// Меню
Please, your choice -> 1
Enter double-precision number -> 100.4
Addition is impossible!
// Меню
Please, your choice -> 1
Enter double-precision number -> -100.4
// Меню
Please, your choice -> 3
// Меню
Please, your choice -> 4
-376.28 -100.4 1.1782 4.16 100.4
// Меню
Please, your choice -> 0
1 -> double 2 -> CCar  0 -> Exit
Please, your choice -> 2
```

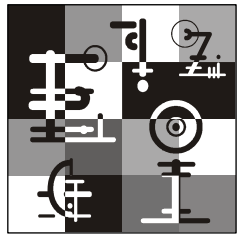
```
Enter size of container -> 25
// Меню
Please, your choice -> 1
Enter car model -> 15-1572
Enter car mass -> 23.5
Addition is impossible!
// Меню
Please, your choice -> 2
Enter car model -> 15-1572
  Is in container
// Меню
Please, your choice -> 1
Enter car model -> 15-1001
Enter car mass -> 24.5
// Меню
Please, your choice -> 3
// Меню
Please, your choice -> 4
15-1001  24.5
15-1572  23.5
15-869  25.3
15-C862  24
Please, your choice -> 0
1 -> double 2 -> CCar  0 -> Exit
Please, your choice -> 0
Для продолжения нажмите любую клавишу . . .
```

Для экономии места в листингах пункты меню показаны в развернутом виде один раз, после чего заменены в протоколе комментарием `// Меню`.



Часть IV

Ввод-вывод и исключения



Глава 19

Основы ввода-вывода

Классификация способов ввода-вывода

Мы уже касались вопросов вывода текста на экран, ввода данных с клавиатуры и вывода данных на экран. Здесь мы рассмотрим способы ввода-вывода с использованием потоков и файлов и соответствующие средства языка C++.

Система программирования MVC++ обеспечивает следующие варианты организации ввода-вывода:

- прямой небуферизованный ввод-вывод с помощью библиотеки C времени выполнения;
- с помощью потоковых средств стандарта ANSI C;
- ввод-вывод для консоли и порта;
- с помощью библиотеки MFC;
- с помощью стандартной библиотеки C++.

Перечисленные варианты организации ввода-вывода можно классифицировать по двум основным признакам:

- стилю программирования;
- уровню взаимодействия с внешними устройствами.

По стилю программирования в C++ можно выделить объектно-ориентированную систему ввода-вывода и три унаследованные процедурно-ориентированные системы ввода-вывода языка C: потоковые средства библиотеки стандарта ANSI C, систему ввода-вывода для консоли и порта, средства низкоуровневого ввода/вывода.

Базовые объектно-ориентированные средства ввода-вывода, обеспечивающие работу с потоками, объявлены в файле `iostream.h`. При подключении файла `fstream.h`, становятся доступны объектно-ориентированные средства обмена данными между программой и файлами через потоки. В файле `sstream.h`

объявлены средства для обмена данными с областью памяти, которая рассматривается как массив символов или как строка типа `string`.

Процедурно-ориентированные средства реализованы с помощью библиотеки стандартного ввода-вывода ANSI C и становятся доступными при подключении в программе заголовочного файла `stdio.h`.

По уровню взаимодействия с внешними устройствами различают ввод-вывод:

- на верхнем уровне;
- на нижнем уровне;
- на уровне консоли и порта.

При взаимодействии с внешними устройствами на верхнем уровне используется *форматированный ввод-вывод*, при этом передаваемые байты группируются в элементы данных определенного типа: целые числа, числа с плавающей точкой, отдельные символы, строки и данные определенных пользователем типов. Достоинством такого подхода является удобство передачи различных типов данных. Недостатком его является высокий уровень накладных расходов при передаче файлов больших объемов.

При работе на нижнем уровне выполняется *неформатированный ввод-вывод*. Суть его заключается просто в пересылке байтов данных между оперативной памятью и внешними устройствами. При этом каждый байт является самостоятельным элементом данных. Достоинством такого подхода является высокая скорость передачи данных. Недостаток заключается в неудобстве его использования для передачи данных разных типов.

При работе на уровне консоли и порта выполняется прямая запись и чтение данных для соответствующего устройства. Консольные средства обмена данными несовместимы со средствами потокового ввода-вывода и средствами ввода-вывода на нижнем уровне. Кроме того, для консоли и порта не требуется выполнять операции открытия и закрытия потока.

В дополнение к приведенной классификации отметим, что по типам передаваемых данных различают два варианта организации ввода-вывода:

- для базовых типов;
- для пользовательских типов.

Средства ввода-вывода в языке C++ реализованы в виде библиотечных перегруженных операций << и >> (извлечения и вставки), манипуляторов, классов потоков, констант, глобальных переменных, функций и типов данных. Рассмотрим два важнейших понятия: поток и файл — и принципы организации обмена данными с файлами через потоки.

Принципы работы с потоками и файлами

Поток (stream) можно определить как абстрактный канал связи, который создается в программе для обмена данными с файлами. Это понятие введено для того, чтобы можно было не учитывать особенности организации канала связи между источником и приемником информации.

У всех потоков имеются общие свойства. Поэтому одинаковые средства (операции и функции) могут применяться для работы с различными потоками.

Файл (file) представляет собой поименованную совокупность данных, находящуюся на внешнем устройстве и имеющую определенные атрибуты (характеристики).

Обычно файл рассматривается как последовательность байтов или символов. Файл имеет начало (перед первым байтом), конец (после последнего байта), и при продвижении от начала к концу каждый байт находится в определенной позиции. Начало файла и первый байт имеют нулевую позицию, каждый последующий байт имеет позицию на единицу больше предыдущего. Начало файла имеет позицию, равную нулю. Позиция конца файла равна размеру файла в байтах.

Высокоуровневые средства ввода-вывода C++ позволяют работать с текстовыми и бинарными файлами после того, как в программе установлена их взаимосвязь с текстовыми и бинарными потоками, соответственно.

Файл называется текстовым, если он рассматривается как последовательность строк символов, разделенных не пробельными символами. Кроме пробела, пробельными символами являются следующие специальные символы:

- \t — табуляция горизонтальная;
- \v — табуляция вертикальная;
- \n — новая строка;
- \r — возврат каретки;
- \f — перевод формата (страницы).

Среди пробельных символов выделяется символ \n (новая строка), который используется для стандартного разделения совокупностей строк на файловые строки (линии, line). Строки символов в текстовых файлах представляют собой последовательности не пробельных символов, которые могут интерпретироваться при вводе как данные определенного типа, представленные в некотором формате. Например, последовательность символов `125`, ограни-

ченная с двух сторон пробельными символами, может быть преобразована при вводе в вещественное число типа `double` или целое число типа `int`.

Файл называется бинарным, если с ним работают как с последовательностью байтов или символов. Бинарные потоки и файлы обычно используются при перегрузке операций извлечения >> и вставки << для пользовательских типов данных или реализации специфических методов ввода-вывода для пользовательских классов.

В качестве файлов рассматриваются не только файлы на дисках, но и любые устройства, с которыми можно осуществлять операции ввода-вывода. Так, файлами являются клавиатура и дисплей, а также модем, принтер и другие подключенные внешние устройства.

Не все файлы имеют одинаковые свойства в связи с различным их назначением и физической реализацией. Например, для файлов на CD ROM возможны только операции ввода. Поэтому при связывании потока с определенным файлом поток приобретает его свойства. Можно говорить о записи в поток и чтении из потока, что эквивалентно записи в файл и чтению из файла.

При работе с потоками и файлами различают буферизованный и небуферизованный ввод-вывод. *Буфер* (buffer) представляет собой область оперативной памяти, которую используют средства ввода-вывода для промежуточного хранения данных, передаваемых между программой и внешним устройством. Буфер может быть системным. Область памяти, где размещаются системные буферы, принадлежит операционной системе, а не программе.

Вывод данных в поток с буфером приводит к выводу этих данных в соответствующий файл только после заполнения буфера. Вывод данных в небуферизованный поток приводит к немедленному выводу их в файл.

Использование буферов позволяет ускорить работу с потоком путем поблочного (а не побайтного) обмена данными. Вместе с тем, наличие буфера приводит к накладным расходам. При вводе данных возможны ошибки, следовательно, требуется контролировать и обеспечивать корректность содержимого буфера перед выполнением очередной операции ввода. Далее, из-за возможности аварийного завершения программы в связи со сбоем при выводе данных необходимо обеспечивать периодическое сохранение содержимого буфера в файле. И наконец, при синхронизации работы потоков наличие буферов также должно учитываться.

Рассмотрим подробнее виды и общие свойства потоков, которые позволяют создавать стандартные средства C++.

Стандартные классы потоков

По направлению передачи данных различают следующие потоки:

- входные (input), или потоки ввода, из которых читаются данные в переменные программы;
- выходные (output), или потока вывода, в которые записываются значения из программы;
- двунаправленные (input-output), или потоки ввода-вывода, допускающие чтение и запись данных.

Входной поток не может быть связан с файлом, который предусматривает только запись. Примером такого файла является принтер или монитор. Выходной поток не может быть связан с файлом, который имеет атрибут "только для чтения".

Двунаправленные потоки имеют свойства входных и выходных потоков. Потоки такого класса имеет смысл использовать для файлов, к которым можно организовать произвольный доступ. С файлами, хранящимися на дискетах или жестких дисках и не имеющими атрибута "только для чтения", можно работать через поток следующим образом. Перед операцией записи-чтения очередной порции данных можно установить требуемую позицию в файле, начиная с которой и будет осуществляться очередной обмен данными. Такое позиционирование обеспечивает доступ к произвольному байту, а не только к очередному следующему байту, как при последовательном доступе к файлу. Поток и связанный с ним файл представляют в программе единый объект, поэтому обычно говорят о позиционировании (seek) потока.

По способу создания различают потоки следующих видов:

- автоматически создаваемые стандартные потоки, которые связаны со стандартными системными устройствами ввода-вывода;
- явно создаваемые потоки для обмена данными с файлами;
- явно создаваемые потоки для обмена данными со строкой в оперативной памяти.

При подключении заголовочного файла `iostream.h` в программе на C++ автоматически создаются 4 стандартных потока:

- `cin` — стандартного ввода данных (по умолчанию с клавиатуры);
- `cout` — стандартного вывода данных (по умолчанию на монитор);
- `cerr` — небуферизованный поток для стандартного вывода сообщений об ошибках (по умолчанию на монитор);

- `clog` — буферизованный поток для стандартного вывода сообщений об ошибках (по умолчанию на монитор).

Примечание

Потоки `cin`, `cout` и `cerr` соответствуют потокам `stdin`, `stdout` и `stderr` языка C.

Связи этих стандартных потоков с файловыми устройствами можно переназначать на уровне операционной системы или в программе. Например, при выполнении командной строки

```
tstpgm < tstpgm.in > tstpgm.out
```

будут переопределены связи стандартных потоков ввода-вывода. И во время выполнения программы из файла `tstpgm.exe` ввод через `cin` будет производиться из файла `tstpgm.in`, а вывод через `cout` — в файл `tstpgm.out`.

Использование потоков `cerr` и `clog` упрощает создание программ на языке C++ для сред, в которых имеются специальные внешние устройства для немедленного вывода сообщений об ошибках через `cerr` и сохранения этих сообщений в файле регистрации ошибок `errors log` .

Потоки для работы с другими файловыми устройствами нужно в программе создавать явно с требуемыми свойствами потоков. Для большинства задач эти потоки могут быть объектами стандартных классов, объявленных в файлах `fstream.h` и `sstream.h` (`strstrea.h`).

Примечание

Базовые свойства всех потоков содержит класс `ios`, объявленный в файле `iostream.h`.

Если в программе требуется работать с файлами через потоки, то необходимо подключать файл `fstream.h`. Тогда можно создавать потоки следующих трех классов:

- `ifstream` — для ввода из файлов;
- `ofstream` — для вывода в файл;
- `fstream` — для обмена с файлом в двух направлениях.

В файле `sstream.h` (`strstrea.h`) объявлены классы потоков для обмена данными с оперативной памятью, в которой выделена специальная область. Эта область рассматривается как массив символов или строка типа `string`. При подключении файла `sstream.h` (`strstrea.h`) в программе можно создавать потоки шести аналогичных классов:

- `istrstream`, `istringstream`

- `ostrstream`, `ostringstream`
- `strstream`, `stringstream`

для работы с массивом символов и строкой типа `string`, соответственно.

На рис. 19.1 приведен фрагмент иерархии стандартных классов потоков.

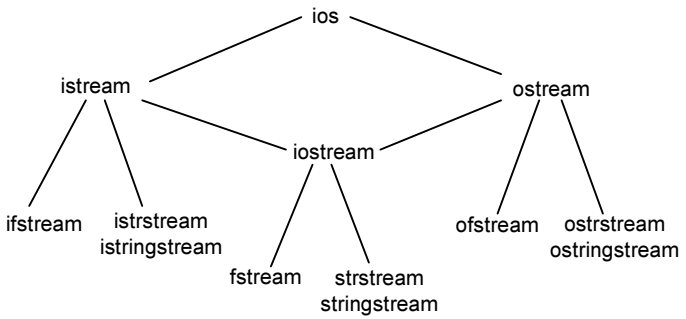


Рис. 19.1. Стандартные классы потоков

В последнем стандарте языка C++ возможности библиотеки классов потоков расширены, имеются отличия в названиях и реализации классов. Приведенные имена классов используются в MVC++.

Процесс работы с файлом через потоки включает 4 этапа:

1. Создание потока.
2. Связывание потока с файлом и открытие файла в определенном режиме.
3. Обмен данными с файлом через поток.
4. Разрыв связи потока с файлом.

При работе со стандартными потоками действия этапов 1, 2 и 4 выполняются автоматически. Стандартные потоки `cin`, `cout`, `cerr` и `clog` называют также предопределенными потоками, поскольку связь их с файловыми устройствами определена до начала выполнения первого оператора программы. Поток `cin` является объектом класса `istream`. Соответственно, потоки `cout`, `cerr` и `clog` — это объекты класса `ostream`. Классы стандартных потоков являются наследниками классов `istream` и `ostream` соответственно.

По умолчанию при корректном завершении программы или при выходе из области видимости потока освобождение буфера и закрытие файла осуществляется автоматически. Тем не менее, этап 4 повышает надежность программ при работе с файлами.

При выполнении этапов 2, 3, 4 следует контролировать наличие ошибок ввода-вывода.

Каждый поток, как и всякий объект, в любой момент времени характеризуется состоянием, определяющим свойства потока и зависящим от работы с потоком. Состояние представляется набором трех групп переменных:

- флагов (признаков) состояния потока и указателя, определяющего связь потока с другим потоком;
- флагов и переменных форматирования, выполняемого в потоке;
- переменной, хранящей текущую позицию потока и флаги режимов работы с файлами.

Состояние потока определяется также содержимым буфера потока.

Анализ флагов состояния позволяет установить наличие ошибок ввода-вывода и возможность дальнейшей работы с потоком. Для этого используются функции-члены класса `ios`.

При анализе состояния потока важным понятием является событие "конец файла" (End Of File, EOF). Это событие возникает при попытке чтения из потока, в то время как текущая позиция потока равна размеру файла. То есть операция ввода выполняется после того, как уже был считан последний байт файла или файл пуст. При этом устанавливается соответствующий флаг состояния. Функции ввода-вывода могут возвращать значение EOF не только при возникновении события "конец файла", но и при возникновении ошибок ввода-вывода. Данное событие используют для завершения считывания данных из файлов.

В зависимости от наличия или отсутствия форматирования различают два варианта ввода-вывода:

- форматированный;
- неформатированный.

Форматированный ввод-вывод, как отмечалось, предполагает преобразование последовательности байтов потока в соответствии с установленными правилами. При выводе в поток форматирование позволяет получать в файле выводимое значение в определенном формате. При вводе форматирование может позволить считывать из потока последовательность байтов, как значение определенного типа.

Флаги форматирования определяют поведение потока при форматированном вводе-выводе с помощью операций >> и << (извлечения и вставки). Флаги и переменные форматирования задают установленные в потоке правила форматирования, которые действуют при выполнении операций >> и <<. Управлять флагами и устанавливать значения переменных форматирования можно

с помощью соответствующих методов класса `ios` или с помощью специальных функций — манипуляторов, часть из которых объявлена в файле `iomanip.h`, а остальные — в файле `iostream.h`. При создании потоков действуют единые правила форматирования, установленные по умолчанию.

Неформатированный ввод-вывод осуществляется с помощью функций-членов `get()`, `put()`, `read()` и `write()` соответствующих классов `istream`, `ostream`, `iostream`. При этом можно выделить два варианта организации: строко-ориентированный и символьный. Первый вариант удобен для работы с текстовыми файлами. Символьный ввод-вывод используется, в первую очередь, для работы с бинарными файлами.

Ранее мы рассматривали форматированный ввод-вывод через стандартные потоки `cin` и `cout`, при котором использовались параметры форматирования по умолчанию. Рассмотрим возможности форматирования для стандартных потоков и другие варианты организации ввода-вывода.

Форматированный ввод-вывод базовых типов

Для форматированного ввода-вывода используются перегруженные операции `>>` и `<<` (правого и левого сдвига), называемые операциями извлечения (extraction) и вставки (insertion), соответственно. Перегрузка проведена в файле `iostream.h` так, чтобы облегчить форматированный ввод-вывод значений переменных базовых типов. Операции вставки и извлечения распознают тип правого операнда и в соответствии с установленными правилами форматирования в потоке преобразуют значение этого операнда (листинг 19.1).

Листинг 19.1. Пример форматированного ввода-вывода базовых типов

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{       char c = 't';
        char s[7] = "string";
        int i = 26;
        double df = 12.3456789;
        ostream *pcout = &cout;
        cout << c <<'\n'<<s<<'\n'<<i<<'\n'<<df<<'\n'<<pcout<<'\n';
        return 0;
}
```

Программа показывает свойства форматирования при работе со стандартными потоками `cin` и `cout`. В результате ее выполнения на экран будут выведены следующие 5 строк:

```
t
string
26
12.34567
10572AC8
```

Последняя строка вывода может отличаться, поскольку шестнадцатеричный адрес размещения области памяти для потока `cout` может быть другим.

Для базовых типов и символьных массивов по умолчанию установлены следующие стандартные правила форматирования:

- символы и строки символов выводятся в привычном виде;
- числа выводятся в десятичной системе счисления;
- знак у положительных чисел не выводится;
- у целых чисел выводятся только значащие цифры (незначащие старшие нули отбрасываются);
- вещественные числа выводятся с сохранением до 6 значащих цифр и указанием положения десятичной точки, если дробная часть не нулевая; при отбрасывании младших значащих цифр производится округление.

Можно управлять форматом ввода-вывода с помощью флагов форматирования потока, которые объявлены в классе `ios`. Для установки флагов форматирования используются символические константы, ряд из которых приведен в табл. 19.1.

Таблица 19.1. *Константы флагов форматирования*

Константа	Способ форматирования
`dec`	Использовать десятичное представление
`fixed`	Для вещественных чисел использовать представление с фиксированной точкой
`hex`	Использовать шестнадцатеричное представление
`internal`	Помещать символ символ-заполнитель (по умолчанию пробел) после знака числа или символа-признака основания системы счисления
`left`	Прижимать к левому краю при выводе
`oct`	Использовать восьмеричное представление

Таблица 19.1 *(окончание)*

Константа	Способ форматирования
`scientific`	Для вещественных чисел использовать "научное" представление: мантисса и порядок, разделенные символом e или E
`skipws`	Пропускать пробельные символы при вводе
`stdio`	Освобождать стандартные потоки `stdout`, `stderr` языка C после каждого вывода в поток
`unitbuf`	Освобождать буферы (выводить содержимое) всех потоков после каждого включения (вывода) в поток

Флаги форматирования хранятся в защищенном члене класса `ios` — переменной `x_flags` типа `long`. Устанавливать флаги форматирования можно с помощью функции `flags()` — члена класса `ios`. Эта функция объявлена в двух вариантах:

```
long flags();
long flags(long new_fmtflags);
```

Новая совокупность флагов `new_fmtflags` может быть получена с помощью операции `|` (побитного ИЛИ) и констант, приведенных в табл. 19.1. Функции возвращают прежний набор флагов.

Листинг 19.2. Пример использования флагов форматирования

```
long oldfmtflags = cout.flags();
// задание новых флагов
cout.flags(ios::scientific|ios::hex);
cout  <<'\n' <<1.2 << "; (dec) 110 = (hex) " << 110;
// восстановление флагов
cout.flags(oldfmtflags);
cout  <<'\n' <<1.2 << "; (dec) 110 = (hex) " << 110;
```

В листинге 19.2 выполняется сохранение флагов, установленных по умолчанию, и установка флагов для вывода вещественных чисел в научном формате и целых чисел в шестнадцатеричной системе счисления. Пояснения приведены в комментариях.

В результате выполнения этого фрагмента на экран будут выведены строки:

```
1.200000e+00; (dec) 110 = (hex) 6e
1.2; (dec) 110 = (hex) 110
```

Сохранять значения флагов, установленных по умолчанию, необязательно. Для восстановления начальных установок можно воспользоваться вызовом `flags(0)`.

С помощью функции `setf()` можно устанавливать отдельные флаги без использования переменной флагов форматирования.

Не все флаги форматирования можно установить одновременно. Информацию о взаимосвязи флагов дают константы, используемые при образовании маски для функции `setf()`. В классе `ios` объявлено три таких константы:

- `basefield` — поле флагов `dec`, `oct`, `hex`;
- `adjustfield` — поле флагов `left`, `right`, `internal`;
- `floatfield` — поле флагов `scientific` и `fixed`.

Эти константы следует использовать в том случае, когда перед установкой флага требуется сбросить все альтернативные флаги.

С помощью функции `unsetf()` сбрасываются все флаги, которые помечены в параметре. Функция возвращает предыдущее значение переменной `x_flags`.

Отметим, что не все флаги действуют на входные и на выходные потоки. Учет свойств флагов может облегчить ввод некоторых специфических значений.

При выводе данных в виде таблиц часто требуется определять фиксированные размеры строки или символьного представления числа. В этом случае говорят о поле вывода или о числе символов, которые должны быть выведены. Если поле вывода больше числа символов в выводимой строке или числе, то незанятые места заполняются определенными символами. В качестве символа-заполнителя по умолчанию устанавливается пробел. Флаги `left`, `right`, `internal` задают правило размещения числа или строки в этом поле.

Для вещественных чисел часто требуется управлять и точностью представления выводимых чисел.

Значения ширины поля вывода, точности представления и символа заполнителя хранятся в следующих защищенных переменных класса `ios`:

```
int x_width;          // задает минимальную ширину поля вывода
int x_precision;      // определяет максимальное количество значащих
                         цифр вещественного числа
int x_fill;           // задает символ-заполнитель поля вывода
                         до минимальной ширины,
```

Доступ к переменным форматирования обеспечивают функции-члены того же класса, имеющие следующие заголовки:

```
char fill();     // возвращает используемый символ-заполнитель
// устанавливает новый символ-заполнитель и возвращает прежний символ
char fill(char cf);
```

```
// возвращает значение переменной x_precision
int precision();
// устанавливает новое значение и возвращает прежнее значение
int precision(int p);
int width();          // возвращает используемый размер поля вывода
// устанавливает ширину поля вывода и возвращает прежнее значение
int width(int w);
```

Примечание

Новое значение ширины поля вывода действует только на очередную операцию вывода, после которой восстанавливается прежнее значение величины `x_width = 0`.

Листинг 19.3. Пример управления параметрами форматирования

```
cout.flags(0);
cout.width(12);
cout.precision(8);
cout.fill('%');
cout.setf(ios::showpoint|ios::showpos|ios::left);
cout<<'\n'<<17.77777<< " = " << 1.77777e1 << " > " << 17<<'\n';
cout.setf(ios::internal);
cout.precision(6);
cout.width(12);
cout.fill('*');
cout<<17.77777<< " != " << 1.77777e1 << " > " << 17<<'\n';
...
```

В результате выполнения указанного фрагмента программы (листинг 19.3) на экране появятся строки:

```
%%%%%%%%%%+17.777770 = +17.777700 > +17
****+17.7778 != +17.7777 > +17
```

В приведенных примерах можно выделить два стиля работы с потоками:

- функциональный, когда для управления состоянием потока используются вызовы вида

```
имя_потока.функция();
```

- операциональный, который наглядно отражает порядок обмена данными, в виде

```
имя_потока << выводимое_1_значение << ...<< n_значение;
имя_потока >> вводимое_1_значение >> ...>> n_значение;
```

Второй из них представляется более наглядным и более привлекательным. В качестве средства для широкого применения этого стиля используются манипуляторы. Рассмотрим их подробнее.

Манипуляторы

Манипуляторами называют специальные функции, позволяющие изменять состояние потока и использующиеся совместно с операциями извлечения и вставки в одном операторе ввода или вывода данных. Отличие манипуляторов от обычных функций состоит в том, что их имена можно использовать в качестве правого операнда при выполнении форматированного обмена с помощью операций << и >>. В качестве самого левого операнда выражения с манипуляторами и операторами обмена всегда используется имя потока.

Например, используя манипуляторы `hex`, `oct` и `endl`, можно написать следующий оператор:

```
cout <<"(dec) 1023 = (hex) " << hex << 1023 << " = (oct) " << oct <<
1023 << endl;
```

В результате выполнения на экране появится строка:

```
(dec) 1023 = (hex) 3ff = (oct) 1777
```

Стандартные манипуляторы ввода-вывода делятся на две группы: манипуляторы с параметрами и манипуляторы без параметров. Манипуляторы с параметрами объявлены в файле `iomanip.h`, а без параметров — в файле `iostream.h`. В табл. 19.2 приведены стандартные манипуляторы без параметров.

Таблица 19.2. *Манипуляторы без параметров*

Манипулятор	Действие в потоке
`dec`	Преобразование в десятичное представление
`hex`	Преобразование в шестнадцатеричное представление
`oct`	Преобразование в восьмеричное представление
`endl`	Вставка символа новой строки и выгрузка буфера
`ends`	Вставка в поток нулевого признака конца строки
`flush`	Выгрузка буфера выходного потока
`ws`	Извлечение и игнорирование пробельных символов
`showbase`	Вставка признака системы счисления
`noshowbase`	Изъятие признака системы счисления
`skipws`	Пропуск пробельных символов при вводе

Таблица 19.2 *(окончание)*

Манипулятор	Действие в потоке
noskipws	Отмена пропуска пробельных символов при вводе
uppercase	Использование символов верхнего регистра при выводе чисел
nouppercase	Отмена использования символов верхнего регистра при выводе чисел
internal	Вставка заполнителей между знаком и модулем выводимого числа
left	Вставка заполнителей после значения в поле вывода
right	Вставка заполнителей перед значением в поле вывода
fixed	Использование для вещественных чисел формата dddd .dd
scientific	Использование для вещественных чисел формата 1.ddddd e dd
boolalpha	Вывод данных типа bool в символическом виде
noboolalfa	Вывод данных типа bool как целых чисел

Манипуляторы без параметров управляют флагами форматирования потока и буфером потока.

Манипулятор endl удобно использовать для немедленной выдачи сообщений программы.

Например:

```
cout << "\nModeling is going." << endl;
```

Вывод с таким использованием манипулятора endl избавляет от необходимости ожидания заполнения буфера до момента окончания процесса моделирования.

Чтобы избежать подобных ситуаций при выводе сообщения об ошибках, целесообразно направлять в небуферизованный поток cerr. При этом именем потока отмечается место обработки ошибок в программе (листинг 19.4).

Листинг 19.4. Пример вывода сообщений об ошибках в поток cerr

```
if (SizeFile > MaxSizeFile)
{
       cerr << "\nSize of file is very large.";
...    // операторы обработки ошибки
}
```

Если при выводе не требуется начинать новую строку и нужно обеспечить немедленное освобождение буфера, то следует использовать манипулятор `flush`.

Например,

```
clog << "\nWork is finishing." << flush;
```

Для управления точностью, шириной поля вывода и другими параметрами форматирования используются манипуляторы с параметрами, приведенные в табл. 19.3.

Таблица 19.3. *Манипуляторы с параметрами*

Манипулятор с параметром	Действие в потоке
`resetiosflags (long fflags)`	Сброс флагов форматирования
`setiosflags (long fflags)`	Установка флагов форматирования
`setbase (int b)`	Установка системы счисления с заданным основанием
`setfill (int cf)`	Установка символа-заполнителя
`setpresision (int p)`	Установка точности вывода вещественных чисел
`setw (int w)`	Установка ширины поля для очередной операции вывода

Отметим, что манипуляторы обеспечивают все возможности методов класса `ios` для управления форматированием.

Листинг 19.5. Использование манипуляторов с параметрами

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   double darray[]={6.7891, -0.0089, 5.6789};
   char col_name[]="Rounded values:";
   cout << setw(sizeof(col_name))<<col_name<<endl;
```

```
    cout << setprecision(3) << setiosflags(ios::fixed | ios::showpoint);
    cout << setfill('x')    << setiosflags(ios::showpos | ios::left);
    for (int i=0;i<sizeof(darray)/sizeof(darray[0]);i++)
    cout << setw(sizeof(col_name))<<darray[i]<<endl;
    return 0;
}
```

В результате выполнения программы (листинг 19.5) на экран будут выведены строки с заголовком столбцов и округленными значениями элементов массива:

```
Rounded values:
+6.7891xxxxxxxxxx
-0.0089xxxxxxxxxx
+5.6789xxxxxxxxxx
```

Анализ состояния потока

При вводе данных с клавиатуры очень легко ошибиться. Поскольку оператор `>>` предназначен для считывания данных ожидаемого типа в ожидаемом формате, то очередная операция ввода может выполниться как нулевая (безрезультатно). Чтобы контролировать подобные ситуации, класс `ios` имеет защищенную переменную `state` типа `int`, в которой хранятся флаги состояния потока.

Для анализа состояния потока используются следующие четыре константы — флагов состояния:

- `badbit` — поток испорчен;
- `eofbit` — достигнут конец файла;
- `failbit` — следующая операция не выполнится;
- `goodbit` — поток не испорчен (флаг установлен в ноль), следующая операция может выполниться.

Для анализа состояния потока можно использовать методы `good()`, `eof()`, `fail()`, `bad()` и `rdstate()` без параметров и две перегруженные унарные операции `( )` и `!( )`.

Метод `good()` и операция `( )` возвращают не ноль, если установлен флаг `goodbit`. Метод `fail()` и операция `!( )` возвращают не ноль, если установлен флаг `failbit`. Метод `rdstate()` возвращает значение переменной `state`.

Листинг 19.6. Пример анализа состояния потока

```
int d;
cout<<"Enter value of d" <<flush;
cin>>d;
if (cin.fail())
{
        cerr<<"Error at entering of d";
        exit(1);
}
return 0;
```

В листинге 19.6 анализ состояния потока при вводе выполняется с помощью метода `fail()` объекта `cin` стандартного потока ввода. С помощью приведенного кода может быть отслежена, к примеру, ошибка переполнения при вводе значения переменной.

Флаги состояния можно переустанавливать с помощью метода `clear()` или второй формы метода `rdstate()`, указав в качестве фактического параметра сбрасываемые или устанавливаемые флаги, соответственно. Здесь есть аналогия с установкой и сбросом флагов форматирования. Вызов `clear()` без параметров эквивалентен вызову `rdstate(ios::goodbit)`.

Листинг 19.7. Пример изменения флагов состояния потока

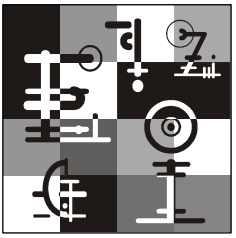
```
void TestFlags( ios& x )
{
   cout << ( x.rdstate( ) ) << endl;
   cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
   fstream x( "c:\test.txt", ios::out );
   x.clear( );
   TestFlags( x );
   x.clear( ios::badbit | ios::failbit  );
   TestFlags( x );
   x.clear( ios::eofbit  );
```

```
    TestFlags( x );

    return 0;
}
```

В листинге 19.7 изменение флагов состояния потока `x` типа `fstream` выполняется с помощью метода `clear()` без параметров и с параметрами. В результате выполнения программы на экран будут выведены следующие строки:

```
0
6
1
```

Например, вызов метода `clear()` с параметром `ios::eofbit` приводит к установке в 1 самого правого бита флага состояния (значение 1).



Глава 20

Дополнительные возможности ввода-вывода

Форматированный ввод-вывод пользовательских типов

В случае создания нового типа данных для удобства работы с ним целесообразно перегрузить операции << и >> (вставки и извлечения). Рассмотрим вариант перегрузки в листинге 20.1.

Листинг 20.1. Пример перегрузки операций << и >>

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class cid_point
 {
    char c_p;
    int i_p;
    double d_p;
   public:
    cid_point(char c= ' ', int i= 0, double d = 0)
                                  { c_p = c;  i_p = i; d_p = d; }
    char cp()                                   { return c_p; }
    int ip()                                    { return i_p; }
    double dp()                                 { return d_p; }
    void cp(char c)                             { c_p=c; }
    void ip(int i)                              { i_p=i; }
    void dp(double d)                           { d_p=d; }

};
```

```
// cid_point insertion
ostream &operator << (ostream &stream, cid_point &p)
{
    stream << '(';
    stream <<p.cp()<<',';
    stream <<p.ip()<<',';
    stream <<p.dp();
    stream << ')' ;
    return stream;
};
// cid_point extraction
istream &operator >> (istream &stream, cid_point &p)
{
    char cr=' ';int ir=0; double dr=0;
    stream.setf(ios::skipws);
    stream >> cr;  //   '('
    stream >> cr; p.cp(cr);
    stream >> cr;  //   ','
    stream >> ir; p.ip(ir);
    stream >> cr;  //   ','
    stream >> dr; p.dp(dr);
    stream >> cr;  //   ')'
    return stream;
};

int _tmain(int argc, _TCHAR* argv[])
{
    cid_point ok('a',1,2.3);
    cout << "ok = " << ok << endl;
    return 0;
}
```

Здесь в качестве возвращаемого значения операций << и >> объявлена ссылка на объект соответствующего типа (`ostream` и `istream`). Это необходимо для того, чтобы работать с классом `cid_point` как с базовым типом. Использование флага `skipws` позволяет снять ограничение на число пробелов между элементами (скобками и запятыми) формата внешнего представления объектов класса `cid_point`.

В результате выполнения программы получим строку:

```
ok = (a,1,2.3)
```

Допустимыми значениями при вводе будут являться не только строки вида `(j,7,9.0)`,

но и такие как

```
( j  ,  7 , 9.0 )
```

или

```
( <Enter> j <Enter> , <Enter> 7 <Enter> , <Enter> 9.0<Enter> )<Enter>.
```

Если при перегрузке указать конкретный поток, например, `cout`, то этим мы уменьшим область применения перегруженного оператора вставки.

Перегружаемый оператор не должен быть членом класса `cid_point`, потому что в противном случае правым операндом будет объект этого класса, а не поток. Возникает вопрос: как обеспечить вывод в поток защищенных и личных членов класса. В рассмотренном примере это достигнуто с помощью соответствующих методов класса. Другой вариант — объявить в классе операцию `>>` с атрибутом `friend`, а затем определить ее. Если все элементы-данные класса имеют спецификатор доступа `public`, то перегрузка самая простая.

Перегрузка оператора извлечения из потока более хлопотная, из-за достаточно жестких ограничений на единый вид формата при вводе и выводе. Приведенное решение на основе форматированного ввода не универсально, но показывает возможное направление действий. В более сложных случаях на помощь здесь могут прийти средства неформатированного ввода-вывода.

Файловый ввод-вывод

Для обмена данными с файлами, размещенными на дисках, используются три стандартных класса `ifstream`, `ofstream` и `iofstream` файловых потоков. Эти классы являются производными от классов `istream`, `ostream` и `iostream`, соответственно. Поэтому они наследуют все свойства родительских потоков по форматированному вводу-выводу, связыванию потоков и управлению состоянием.

Чтобы начать работу с файлами, необходимо "создать" соответствующие потоки. Для этого можно использовать конструкторы соответствующих классов файловых потоков ввода-вывода в самой простой форме — без параметров. Листинг 20.2 демонстрирует выполнение всех четырех основных этапов работы с файлами через потоки.

Листинг 20.2. Пример копирования файлов

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;

// вспомогательная функция
void error (const char * message)
  {
          cerr<<'\n'<<message<<'\n';
          exit(1);
  }
int _tmain(int argc, _TCHAR* argv[])
   {
       char buf[64]; // вспомогательный буфер
       // 1. Создание потоков
       ifstream fin;
       ofstream fout;
       // 2. Открытие файлов и связывание с потоками
       fin.open("C:\\Temp\\in.txt");
       if (!fin.good()) error("Can't open file for reading.");
       fout.open("C:\\Temp\\out.txt");
       if (fout.fail()) error("Can't open file for writing.");
       // 3. Выполнение операций обмена
       while (fin.getline(buf,sizeof(buf))) fout<<buf<<endl;
       if ( !fin.eof() || !fout.good() )
          error("Error during coping of files");
       // 4. Закрытие файлов, разрыв связей потоков с файлами
       fin.close();
       fout.close();
       return 0;
}
```

В программе копирования файла `in.txt` в файл `out.txt` есть все четыре основных этапа работы с файлами. Предполагается, что файлы находятся в каталоге C:\\Temp. Файл `in.txt` содержит строки символов, разделенные парами символов `\r` и `\n` (символами-ограничителями "конец файловой строки"). Длина файловой строки не превосходит 64 символов.

Число символов, извлекаемых из потока методом `getline()` и помещаемых в буфер, задается вторым параметром. При этом считывание происходит до первого символа-ограничителя "конец файловой строки". Метод `getline()` извлекает символ-ограничитель из потока, но не помещает его в буфер. Поэтому после каждого вывода содержимого буфера в поток `fout` вставляется и символ-ограничитель. При использовании потока `cin` символ-ограничитель появляется в потоке при нажатии клавиши <Enter>.

Для примера причинами выдачи сообщений `"\nCan't open file for..."` могут быть следующие:

- отсутствие файла `in.txt` в указанном каталоге или его особые атрибуты;
- неготовность устройства;
- у существующего файла `out.txt` установлен атрибут "только для чтения" или другие особые атрибуты.

В примере используется режим работы с файлом, устанавливаемый по умолчанию. Достаточно часто по умолчанию предполагается работа с текстовыми файлами.

Возможно совмещение первых двух этапов с помощью второй формы конструкторов потоков:

```
ifstream(const char *fname, int mode = ios::in);
ofstream(const char *fname, int mode = ios::out);
ifstream(const char *fname, int mode);
```

Например:

```
ifstream in("C:\\autoexec.bat");
```

Режим работы можно явно указывать и во втором параметре метода `open()`. Для задания режимов используются символические константы, определенные в классе `ios` (табл. 20.1).

Таблица 20.1. *Константы режимов работы с файлами*

Константа	Режим	Позиция ввода-вывода
`in`	Открытие файла для чтения	начало файла
`out`	Открытие файла для записи	начало файла
`ate`	При открытии файла искать конец файла	конец файла
`app`	Открытие существующего файла для обновления (записи и чтения)	конец файла
`trunc`	Усечение размера файла до нулевой длины	начало и конец файла
`binary`	Открыть для символьного (бинарного) обмена	–

При связывании файла с потоком соответствующим образом инициализируется и внутренняя переменная потока "позиция ввода-вывода". Значение этой переменной указывает на место записи/чтения очередной порции данных в файле и приведено в табл. 20.1.

Для одного и того же файла можно создать несколько потоков, чтобы работать с файлом в различных режимах. Перед использованием другого потока нужно разорвать его связь с файлом и установить связь с этим файлом для другого потока. В частности, метод `close()` необходимо явно вызывать при перенаправлении потока и изменении режима его работы.

Для ввода и вывода числовых значений можно пользоваться операциями извлечения и вставки. При этом следует помнить, что при вводе пробельные символы извлекаются и игнорируются, а при выводе необходимо между числами вставлять пробельные символы явно.

Неформатированный ввод-вывод

Неформатированный ввод-вывод обычно подразумевает работу с бинарными файлами или с потоками символов (байтов). Среди методов классов `istream`, `ostream`, `iostream` имеются такие, которые ориентированы на работу со строками символов и, соответственно, позволяют работать с текстовыми файлами. Поэтому можно выделить два варианта неформатированного ввода-вывода: символьный (бинарный) и строко-ориентированный.

Символьный ввод-вывод

При символьном обмене данными символы (байты) передаются через поток без преобразования. Обычно для этого используются методы `read()` и `write()`, заголовки которых имеют вид:

```
istream& read(char* pstring, int n);
ostream& write(const char* pstring, int n);
```

Метод `read()` извлекает `n` символов из потока ввода и помещает их по указателю `pstring`. Метод `write()` по указателю `pstring` из области памяти вставляет в поток вывода `n` символов. Пример использования этих функций приведен ранее.

Для ввода одного символа можно использовать метод `get()` без параметра. А для посимвольного вывода используется метод `put()` с единственным параметром типа `char`.

Метод `get()` с параметром типа `char` при символьном вводе работает так же, как и без параметра.

Листинг 20.3. Пример работы с файлами при символьном вводе-выводе

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;

        int _tmain(int argc, _TCHAR* argv[])

       { ifstream fin;
         fstream fout;
         char ch;
         char fni[80];
         char fno[80];
         cout<<"Enter the name of input file: ";
         cin.getline(fni, sizeof(fni));
         fin.open(fni,ios::binary);
         if(!fin.good()) { cerr<<"Can't open input file."; exit(1);}
         cout<<"Enter the name of output file: ";
         cin.getline(fno, sizeof(fno));
         fout.open(fno, ios::app|ios::binary);
         if(!fout.good()){ cerr<<"Can't open output file."; exit(1);};
         while(!fin.eof())
         {
           fin.get(ch);
           fout.put(ch);
         }
        cout<<"Contents of file'" << fni
          ↳<< "' has been added to the end of file '" << fno << "'";
        return 0;
}
```

Эта программа (листинг 20.3) добавляет содержимое входного файла в конец выходного файла.

Если предполагается, что в файле хранятся строки, то имеет смысл работать с такими файлами, не используя символьный ввод-вывод.

Строко-ориентированный ввод-вывод

Классы потоков ввода-вывода имеют средства для чтения и записи строк. При работе с файлами, содержащими строки символов, наряду с рассмотренными методами `getline()` и `write()`, следует использовать метод `get()`, который имеет следующую форму записи заголовка:

```
istream& get(char* pstring, int n, char = '\n');
```

В качестве третьего параметра в `get()` можно задавать другой символ, используемый для разделения строк.

Например, при вызове

```
mstream.get(istr, n);
```

из потока `mstream` будут извлечены `n` символов, если не встретился символ-разделитель `\n`, и помещены в массив символов `istr`. В противном случае из потока будут извлечены все символы до разделителя и помещены в массив `istr` с добавленным в конец нуль-символом. Символ-разделитель из потока не извлекается.

Напомним следующие особенности при работе с файлами:

- при выводе символ `\n` заменяется последовательностью символов `\r` и `\n`;
- при вводе последовательность из этих двух символов рассматривается как один символ `\n`.

Предположим, что в файле `par.dat` хранятся значения радиуса основания и высоты цилиндра в виде следующих двух строк:

```
r =  5.6
h = 19.2
```

Программа, выполняющая считывание значения `r` и `h` из файла и запись в новый файл вычисленной площади поверхности цилиндра, представлена в листинге 20.4.

Листинг 20.4. Работа с текстовым файлом

```
#include "stdafx.h"
#define M_PI       3.14159265358979323846
#include <fstream>
#include <iostream>
using namespace std;
       int main()
       {
         ifstream fin;
         char rstr[80];
```

```
    double r,h;
    fin.open("C:\\Temp\\par.dat");
    if(!fin.good())
     { cerr <<"Can't open input file"; exit(1);
     }
    fin.get(rstr,sizeof(rstr),'=');
    fin.get(); // считывание разделителя
    fin>>r;
    fin.get(rstr,sizeof(rstr),'=');
    fin.get();
    fin>>h;
    ofstream fout("C:\\Temp\\res.txt",ios::out);
    if(!fout.good())
     {  cout <<"Can't open output file"; exit(1);
     }

    fout << "s = " << 2*M_PI*r*(h+r) << endl;
    fout.close();
    return 0;
}
```

В результате выполнения программы в текстовом файле C:\\Temp\\res.txt окажется строка вида `s = 872.609`.

Вместо двух вызовов `get()` для считывания строки символов, находящихся до строки с символьным представлением числа, можно однократно использовать метод `ignore()`. Заголовок этого метода имеет вид:

```
istream& ignore(int n = 1, int delimiter = EOF);
```

Здесь `n` — число символов, извлекаемых из потока до символа-разделителя `delimiter`, заданного вторым параметром; `EOF` — символическая константа, означающая признак конца файла. Из потока извлекается и символ-разделитель.

Например, вызов метода:

```
fin.ignore(80,'=');
```

позволяет считать из потока символы до символа, разделяющего строку файла `par.dat` на две части, и сам символ-разделитель.

Из приведенных примеров ясно, что для успешной работы должна быть априорная информация о структуре входных файлов.

Еще одним полезным методом потокового ввода является метод `gcount()`, который возвращает число успешно считанных символов в предшествующей

операции неформатированного ввода. Неформатированное извлечение символов из входного потока осуществляется с помощью одного из методов `get()`, `getline()` и `read()`.

Строко-ориентированный ввод-вывод полезен не только при работе с текстовыми файлами. Можно поток связать со строкой в памяти и производить обмены в памяти потоковыми методами. Рассмотрим этот вопрос подробнее.

Обмены со строкой в памяти

Стандартная библиотека C++ позволяет организовать обмен данными между различными модулями программы через общий массив символов или строку типа `string`. При обменах через массив символов обычно используют С-строки и следующие три класса, объявленные в файле `sstream.h (strstrea.h)`: `istrstream`, `ostrstream` и `strstream`. Для обменов со строкой типа `string` используются также три класса: `istringstream`, `ostringstream` и `stringstream`. С данными классами работают аналогично как и с соответствующими классами файловых потоков. Небольшие отличия связаны, в первую очередь, с упрощением интерфейса работы с памятью. Классы строковых потоков хорошо обеспечивают форматированный обмен данными с символьными массивами и строками типа `string`.

Мы рассмотрим обмены в памяти с использованием первых трех классов строковых потоков. Организация обмена со строкой типа `string` производится аналогично.

Рассмотрим обмен данными в памяти через массив символов (строку) (листинг 20.5). Для простоты обмены проводятся в пределах одного модуля.

Листинг 20.5. Основные возможности строковых потоков

```
#include "stdafx.h"
#include <strstream>
#define M_PI       3.14159265358979323846
#include <fstream>
#include <iostream>
using namespace std;



   char str[180];  // глобальная переменная
   int main()
     {
```

```
        char rstr[80];
        double r = 2., h = 5., p = 1.;
        // запись параметров в память
        ostrstream sout(str,sizeof(str));
        // у выходного строкового потока указывают размер
        if(!sout.good())
          { cerr << "Can't write in to array str"; exit(1);}
        sout << " For cilinder with parameters: r = " << r;
        sout << "; h = " << h;
        sout << " and p = " << p;
        // чтение параметров из памяти
        istrstream s1in(str);
        s1in >> rstr >> r >> rstr >> h;
        // запись результата
        sout << " we have mass m = " << M_PI * r * r * h * p;
        // передача из памяти в cout
        istrstream s2in(str);
        while (!s2in.eof())
        {
             s2in.get(rstr,sizeof(rstr));
             s2in >> r;
             cout << rstr << r;
        }
    return 0;
}
```

В программе формируется массив комментариев и значений радиуса основания, высоты и плотности для вычисления массы цилиндра. Затем этот массив считывается. Вычисленная масса с комментарием дописывается в массив. В конце содержимое массива выводится на экран. Из приведенной программы нетрудно заметить схожесть работы с файловыми и строковыми потоками.

Ввод-вывод с помощью библиотеки ANSI C

Для совместимости с предыдущими версиями язык C++ поддерживает стандартные средства языка ANSI C для работы с потоками в процедурно-ориентированном стиле, которые объявлены в заголовочном файле `stdio.h`. Имеется много общего при работе со средствами работы с файлами через

потоки в C и C++. Поэтому сначала кратко рассмотрим работу со стандартными потоками, а затем файловый ввод-вывод.

Характеристика стандартных потоков

При подключении файла `stdio.h` автоматически создаются следующие пять потоков:

- `stdin` — для ввода со стандартного устройства (обычно клавиатура);
- `stdout` — для вывода на стандартное устройство (обычно монитор);
- `stderr` — для стандартного вывода сообщений об ошибках (обычно на монитор);
- `stdaux` — для вывода на стандартное дополнительное устройство (обычно на монитор);
- `stdprn` — для вывода на стандартное устройство печати (принтер).

Первые три потока имеют прямые аналоги в C++: `cin`, `cout` и `cerr`. Их аналогия заключается в устанавливаемой по умолчанию связи с соответствующим файловым устройством и наличии или отсутствии буферизации. Но изначально все эти потоки разные, поскольку за их работу отвечают разные средства и используются различные системные буферы.

Форматированный ввод-вывод через стандартные потоки

Операции форматированного обмена данными с консолью через стандартные потоки `stdin` и `stdout` выполняются с помощью двух функций: `printf()` и `scanf()`. Прототипы функций имеют следующий вид.

```
int printf(const char *format[, argument, ...]);
int scanf(const char *format[, address, ...]);
```

Обязательным аргументом функций является строка форматирования. Необязательные аргументы задают имена (в `printf`) или адреса (в `scanf`) переменных, значения которых выводятся или вводятся соответственно.

При успешном выводе метод `printf()` возвращает число переданных байтов. В случае ошибки возвращается значение EOF.

При успешном форматированном вводе метод `scanf()` возвращает число запомненных полей. В случае ошибки ввода возвращается 0, если не произведено ни одного запоминания по указанным адресам. Если при выполнении `scanf()` происходит событие "конец файла" или делается попытка прочитать пустую строку, то возвращается значение EOF.

Рассмотрим программу, в которой выполняется форматированный ввод-вывод через стандартные потоки языка C. Она осуществляет считывание с клавиатуры трех чисел и вывод полученной суммы на экран (листинг 20.6).

Листинг 20.6. Использование стандартных потоков языка C

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main(void)
{
    int n;
    float d1,d2,d3, dsum;
    printf("\nInput 3 real numbers, devided blanks \n");
    n=scanf("%f %f %f",&d1,&d2,&d3);
    printf("Read fields: %d", n);   // при успешном чтении n=3
    dsum = d1+d2+d3;
    printf("\n%f + %f + %f = %f",d1,d2,d3,dsum);
}
```

При выводе с помощью `printf()` строка форматирования выводится посимвольно на экран, пока не встретится символ начала спецификатора формата `%`. После чего начинается вывод преобразованного (форматированного) значения первой переменной, следующей в списке аргументов за строкой форматирования. Вывод строки форматирования продолжается до очередного спецификатора и выводится значение второй переменной и т. д.

Рекомендуется, чтобы число аргументов за строкой форматирования совпадало с числом спецификаторов, и тогда соответствие между ними определяется очередностью следования в `printf()`.

Ввод с помощью `scanf()` осуществляется аналогично с одним существенным отличием. В качестве аргументов, следующих за строкой форматирования, необходимо передавать адреса переменных, значения которых вводятся с клавиатуры.

Общая форма спецификатора формата имеет следующий вид:

```
%[*][width][.prec][F|N][h|l|L]type_char
```

В квадратных скобках указаны необязательные составляющие спецификатора, а в табл. 20.2 даны необходимые пояснения.

Таблица 20.2. *Спецификаторы формата*

Компонент	Назначение
*	Здесь указывается один из следующих символов: • минус — выровнять по левому краю поля вывода; • пробел — если число положительное, то вместо знака выводить пробел; • + — значения знаковых типов всегда выводятся со знаком; • 0 — заполнить лишнее пространство нулями вместо пробелов; • # — выводить 0 перед восьмеричным или 0x перед шестнадцатеричным значением
width	Указывается целое положительное число, задающее максимальную ширину поля вывода в символах
prec	Указывается целое положительное число, задающее максимальное число цифр дробной части, для целых — минимальное число выводимых цифр
F|N	Один из модификаторов размера адреса (указателя) аргумента: • F — дальний (длинный); • N — ближний (короткий)
h|l|L	Один из модификаторов типа аргумента: • h = short int; • l = long int, если type_char определяет целочисленное преобразование; • l = double, если type_char определяет вещественное преобразование; • L = long double
type_char	Указывается один из символов, приведенных в табл. 13.3, который задает соответствующее преобразование типа при вводе-выводе

Таблица 20.3. *Символы преобразования при вводе-выводе*

Значение type_char	Тип выводимого значения аргумента
числовые	
d	Десятичное целое со знаком типа int
D	Десятичное целое со знаком типа long

Таблица 20.3 *(окончание)*

Значение type_char	Тип выводимого значения аргумента
числовые	
e, E	Вещественное типа `float` в форме `[-]d.dddde[-]ddd` или `[-]d.ddddE[-]ddd`
f	Вещественное типа `float` в форме `dddd.dddd`
g, G	Вещественное типа `float`
o	Восьмеричное целое типа `int`
O	Восьмеричное целое типа `long`
i	Десятичное, восьмеричное или шестнадцатеричное целое типа `int`
I	Десятичное, восьмеричное или шестнадцатеричное целое типа `int`
u	Беззнаковое десятичное целое типа `unsigned int`
U	Беззнаковое десятичное целое типа `unsigned long`
x, X	Шестнадцатеричное типа `int`
символьные	
s	Строка символов
c	Символ
%	Символ `%`
адресные	
n	Указатель на целое
p	Указатель в шестнадцатеричной форме `YYYY:ZZZZ` или `ZZZZ`

В листинге 20.7 представлены некоторые возможности функции `printf()`.

Листинг 20.7. Некоторые возможности функции `printf()`

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main(void)
{
```

```
 printf("%s begin %s!\n","Programme","work");
 int i = 98;
 printf("%d = %o <8> = %x <16>.\n",i,i,i);
 double f1=345.54321;
 printf("Different width and precision: \n%f\n%6.2f\n%10.5f",f1,f1,f1);
 printf("\nAlign left f1 = %-15.2f = %.2f;",f1,f1);
 printf("\nFill by zero f1 = %015.2f = %.2f;",f1,f1);
 double f2 = 3.4554321;
 printf ("\n (%f) / (%f) * 100%% = %2.0f%%",f2,f1,(f2/f1*100.));
 return 0;
}
```

Для функции `scanf()` обычно используют следующие спецификаторы формата:

- одиночный символ `%c`;
- массив символов `%[size]c`, где `size` — размер массива (константа);
- строка `%s`;
- вещественное число `%e`, `%E`, `%f`, `%g` или `%G`;
- беззнаковые `%d`, `%i`, `%o`, `%x`, `%D`, `%I`, `%O`, `%X`, `%c`, `%n`.

Использование мощных функций `printf()` и `scanf()` требует большой внимательности. Поэтому проще реализовывать так называемый строко-ориентированный ввод-вывод, к рассмотрению которого мы и переходим.

Строко-ориентированный ввод-вывод через стандартные потоки

Для ввода символов из потока `stdin` используется функция с прототипом вида:

```
int getchar(void);
```

Кроме того, можно использовать функцию с прототипом:

```
int getc(FILE *stream);
```

Функция `getc()`, в отличие от `getchar()`, позволяет выбирать символ из заданного потока. Поэтому для ввода из потока `stdin` его нужно указать в качестве аргумента. Функции считывают символ из буфера, после того как нажата клавиша <Enter>. В случае ошибки ввода-вывода, функции возвращают значение EOF.

Прототипы аналогичных функций вывода символов имеют вид:

```
int putchar(int c);
int putc(int c, FILE *stream);
```

Для ввода символа на экран с помощью `putc()` в качестве второго аргумента нужно указать `stdout` или `stderr`. При возникновении ошибки ввода-вывода функции возвращают значение EOF, в противном случае — значение аргумента (первого — для `putc()`).

Рассмотрим программу, в которой четырьмя способами выводится введенный символ (листинг 20.8).

Листинг 20.8. Варианты вывода символов

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main(void)
{
  char c1;
  printf("Input symbol: ");
  c1 = getc(stdin);
  printf("\nWas innput symbol: %c",c1);          // 1
  putchar('\n');
  putchar(c1);                                   // 2
  putc('\n',stdout);
  putc(c1,stdout);                               // 3
  if(putc('\n',stderr)==EOF) printf("EOF in stderr!");
  putc(c1,stderr);                               // 4
  return 0;
}
```

Для вывода строк используются функции, имеющие прототипы:

```
int puts(const char *s);
int fputs(const char *s, FILE *stream);
```

Например, если в программе определена строка вида

```
char str[]="Проверка вывода строки";
```

то вызов функции `puts(str)`, как и вызов `fputs(str,stdout)`, приводит к выводу в стандартный поток `stdout` строки символов `str`, оканчивающуюся нулем, заменяя последний нулевой байт символом "новая строка". В случае ошибки вывода возвращается значение EOF, в противном случае возвращается неотрицательное число.

Чтобы вывести вещественное число с помощью `puts()`, необходимо воспользоваться функцией с прототипом

```
char *gcvt(double value, int ndec, char *buf);
```

которая объявлена в `stdlib.h`. Функция преобразует значение переменной value в ASCIIZ-строку, содержащую символьное представление числа с плавающей запятой, `buf` — указатель на буфер, в котором хранится полученная строка из `ndec` значащих цифр (листинг 20.9).

Листинг 20.9. Использование функции `puts()`

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main(void)
{
   double value = 1.234;
   char buf[19];
   int ndec = 5;
   gcvt(value, ndec, buf);
   puts(buf);
   return 0;
}
```

Для ввода строк используется функция `gets()`, прототип которой аналогичен `puts()`. Для ввода чисел с помощью `gets()` необходимо воспользоваться функциями `atoi()`, `atol()` и `atof()` для преобразования строки, содержащей символьное представление числа в целое, длинное целое и вещественное число соответственно (листинг 20.10).

Листинг 20.10. Использование функции `gets()`

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(void)
{
   int vi;
   float vf;
   char buf[19];
   puts("Input real number: ");
   gets(buf);
   vf = atof(buf);
```

```
   printf("Was input vf: %f\n", vf);
   vi = atol(buf);
   printf("Was input vi: %d\n", vi);
   printf("Was input vi: %.3d \n", vi);
   return 0;
}
```

При вводе с помощью функции `gets()` значения `5.234` в результате выполнения программы на экран будут выданы строки:

```
Input real number:
Was input vf: 5.234000
Was input vi: 5
Was input vi: 005
```

Отличия последних двух строк обусловлены использованием спецификатора %.3d.

Создание потоков для файлового ввода-вывода

Для того чтобы начать работу с файлами, расположенными на логических дисках, необходимо "создать" соответствующие потоки. Для этого используются переменные типа указатель на структуру `FILE`, например:

```
using namespace std;
...
FILE *fin, *fout, *fio;
```

Можно считать, что после объявления этих трех переменных в программе созданы три потока для ввода из файла `fin`, для вывода в файл `fout` и для обмена данными с файлом в обоих направлениях `fio`. Имена этим переменным можно давать любые, но лучше отражать в них функциональное назначение потока.

Для дальнейшей работы после создания потока его нужно связать с необходимым файлом, при этом указав режим работы. Для этого используется функция `fopen()`, которая в случае ошибки ввода-вывода возвращает нулевое значение указателя на `FILE`. Прототип функции `fopen()` имеет вид:

```
FILE *fopen(const char *filename, const char *mode);
```

Например:

```
if ((fin=fopen("A:\DAT.IN", "rt")) = = NULL)
{
```

```
  fprintf(stderr,"\nCan't open file!");
}
```

При связывании файла с потоком соответствующим образом инициализируется и внутренняя переменная потока "позиция ввода-вывода". Значение этой переменной соответствует номеру позиции в потоке, с которой начинают записываться или считываться символы.

В табл. 20.4 приведены значения, с помощью которых задают режим работы с текстовым файлом и соответствующие значения переменной "позиция ввода-вывода".

Таблица 20.4. *Режим работы с текстовым файлом*

Значение режима	Описание режима	Позиция ввода-вывода
`rt`	Открытие файла только для чтения	Начало файла
`wt`	Создание файла для записи. Если файл с указанным именем существовал, то его содержимое теряется после вызова `fopen(...,"wt");`	Начало файла
`at`	Открытие файла для добавления в конец файла; если файл с указанным именем не существует, то он создается	Конец файла
`rt+`	Открытие существующего файла для обновления	Конец файла
`wt+`	Создание нового файла для обновления. Если файл с указанным именем существовал, то его содержимое теряется	Начало файла
`at+`	Открытие файла для обновления, начиная с конца; если файл не существовал, то он создается	Конец файла

Для задания режима работы с бинарными файлами вместо символа `t` в значении режима следует использовать символ `b`. Если явно тип файла не задан, то значение по умолчанию определяется глобальной переменной `_fmode`, объявленной в файле `fcntl.h`.

Перед обменом данными с файлом можно изменять значение внутренней переменной "позиция ввода-вывода" — позиционировать поток, связанный с определенным файлом. Позиционирование осуществляется с помощью функции `fseek()`, имеющей следующий прототип:

```
int fseek(FILE *stream, long offset, int whence);
```

Аргумент `offset` задает смещение в байтах относительно точки отсчета, положение которой определяется третьим аргументом `whence`. Параметр `whence` может принимать 3 значения, которые приведены в табл. 20.5.

Таблица 20.5. *Значение параметра* `whence`

Константа	Значение	Положение точки отсчета
`SEEK_SET`	0	Начало файла
`SEEK_CUR`	1	Текущее значение переменной "позиция ввода-вывода"
`SEEK_END`	2	Конец файла

После вызова функции `fseek()` следующая операция обмена с файлом при обновлении его содержимого (режимы `a+` и `w+`) может быть вводом и выводом. Функция возвращает ненулевое значение при ошибке позиционирования. Рассмотрим варианты организации третьего этапа работы с текстовыми файлами.

Файловый форматированный ввод-вывод

Для обмена данными между программой и файлом со строкой форматирования используются две функции `fprintf()` и `fscanf()`, которые аналогичны функциям `printf()` и `scanf()` соответственно. Первое отличие заключается в том, что перед строкой форматирования нужно указывать имя потока.

Например:

```
fprintf(fout,"First line of file\n");
fscanf(fin,"%s",stringbuf);
```

Второе отличие связано с функцией `fscanf()`, которая в указанном выше примере будет считывать не всю строку, а только подстроку (слово) до первого пробельного символа.

Для организации цикла чтения из файла можно воспользоваться функцией `feof()`, которая сообщает о возникновении события "конец файла" в потоке, связанном с определенным файлом. Прототип данной функции имеет вид

```
int feof(FILE *stream);
```

Функция возвращает ненулевое значение, если произошло событие "конец файла" при выполнении предшествующей операции чтения из файла.

При завершении работы с файлом целесообразно разрывать связь потока с файлом с помощью функции `fclose()`, у которой в качестве фактического параметра следует использовать имя соответствующего потока.

Можно использовать функцию `fcloseall()` для закрытия всех открытых в программе файлов. У данной функции пустой список аргументов.

При выполнении закрытия файлов буфер потока освобождается: содержимое выталкивается в файл, в который производилась запись, или теряется, если последней производилась операция чтения.

Если операция чтения выполнилась с ошибкой, то для корректной дальнейшей работы с файлом необходимо прежде всего явно очистить буфер с помощью функции `fflash()`. Для анализа наличия ошибки в потоке используется функция `ferror()`. Данные функции возвращают ненулевое значение, если произошла ошибка при выполнении последней операции записи/чтения или освобождении буфера. Единственный аргумент функций — имя потока.

Например:

```
fprintf(fin,"%s",stringbuf);
if (ferror(fin))
{
  printf("\nError of file reading!");
  if (fflash(fin))
     printf("\nError of work with stream fin!");
}
```

При таком флэшировании файл остается открытым, и информация в нем не искажается.

Функция `fflash()` не влияет на потоки без буферов.

Завершить выполнение программы (листинг 20.11) можно принудительно при чтении файла `tstfile.txt` путем нажатия клавиши <Esc>. Для чтения очередного слова из файла требуется на клавиатуре нажимать на символьную клавишу.

Листинг 20.11. Работа с файлом в режимах записи, добавления и чтения через один поток

```
#include "stdafx.h"
#include <conio.h>                    //kbhit()
#include <iostream>
using namespace std;
int main(void)
{
   FILE * fsio;              // FileStreamInputOutput
   char fname[13] = "tstfile.txt";
   char rbuf[81] = "";
   fsio = fopen(fname, "wt");
```

```
    fprintf(stdout,"Open file %s in regime 'wt'\n", fname);
    fprintf(fsio,"Example of work with file;\n");
    for (int i = 2; i<17; i++)
        fprintf(fsio,"\n%i line.", i);
    fclose(fsio);
    fsio = fopen(fname, "at");
    fprintf(stdout,"Open file %s in regime 'at'\n", fname);
    for (int i = 1; i<17; i++)
        fprintf(fsio,"\n%i added line.", i);
    fclose(fsio);
    fsio = fopen(fname, "rt");
    fprintf(stdout,"Open file %s in regime 'at'\n", fname);
    printf("\nContent of file %s:\n", fname);
    while(!feof(fsio))
    {
     fscanf(fsio,"%s",rbuf);//считывается только очередное слово
     fprintf(stdout,"next word: %s\n",rbuf);
     while(!kbhit())
      {
        if (getch()!=27) break;
           else
            {
              puts("\nQuit by ESC...");
              fclose(fsio);
              return 0;
            }
      }
    }
    fclose(fsio);
    puts("\nProgram done...");
    return 0;
}
```

В примере используется функция `kbhit()`, определяющая состояние буфера клавиатуры. Функция возвращает нулевое значение, если буфер клавиатуры пуст, и ненулевое в противном случае.

Файловый строко-ориентированный ввод-вывод

Для ввода-вывода строк через потоки используются функции, прототипы которых имеют вид:

```
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
```

Данные функции имеют ряд отличий от аналогичных функций `puts()` и `gets()` (листинг 20.12):

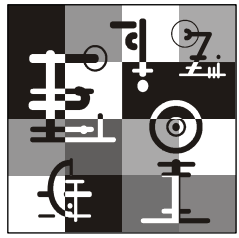
- в качестве последнего фактического параметра нужно задавать имя потока;
- при использовании `fputs()` строка выводится без замены терминального символа `0x00` символами `\r\n` (терминальный символ не выводится);
- второй параметр в функции `fgets()` задает размер массива, где будет хранится считанная строка; поэтому если длина файловой строки меньше величины `n-1`, то она считывается полностью с добавлением нулевого терминального символа, в противном случае считывается только `n` символов.

Листинг 20.12. Работа с текстовым файлом

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main(void)
{
   FILE *pfs; //PointerFileStream,
   char rbuf[127] = "";
   pfs = fopen("dat.txt", "wt");
   fputs(strcpy(rbuf,"Example of work with file.\n"), pfs);
   fputs(strcpy(rbuf,"Last line of file."), pfs);
   puts("Two lines was writed in file dat.txt.");
   fclose(pfs);
   pfs = fopen("dat.txt", "rt");
   puts("\nContent of file 'dat.txt':");
   while(!feof(pfs))
   {
    fgets(rbuf, sizeof(rbuf)-1, pfs);
```

```
    printf("next line: %s", rbuf);
   }
   fclose(pfs);
   puts("\nProgram done...");
   return 0;
}
```

Здесь с помощью указателя `pfs` выполняется открытие файла с именем `dat.txt` в режиме `wt` и в него записываются две строки. Затем файл закрывается, открывается в режиме `rt`, и из него считываются строки с помощью функции `fgets()`.



Глава 21

Обработка исключений

Основы обработки исключений

Исключения, или исключительные ситуации (exception handling) представляют собой события, возникающие внутри программы и приводящие к ненормальной ее работе или ошибкам времени исполнения.

Исключения встроены в язык C++ для обработки ошибочных ситуаций, которые иным образом обработаны быть не могут, таких как ошибки конструктора или перегруженной операции. Обработка исключений позволяет автоматизировать большую часть кода для обработки ошибок, для чего раньше требовалось ручное кодирование. Обслуживаются только так называемые *синхронные исключения*, которые возникают внутри программы. Внешние события, такие как нажатие клавиш <Ctrl>+<C>, исключениями не являются.

Когда программа встречает ненормальную ситуацию, на которую она не рассчитана, можно передать управление другой части программы, способной справиться с этой проблемой, и либо продолжить выполнение программы, либо завершить работу.

Выброс исключения (exception throwing) представляет собой событие, приводящее к передаче управления в секцию кода, содержащую функцию-обработчик данного исключения. Выброс исключения позволяет собрать в точке выброса информацию, которая может оказаться полезной для диагностики причин, приведших к нарушению нормального процесса функционирования программы.

Как правило, исключение должно выбрасываться в следующих случаях:

- нет другого способа сообщить об ошибке (например, в конструкторах, перегруженных операциях и т. д.);
- ошибка неисправимая (например, недостаток памяти);

- ошибка непонятна или неожиданна, в связи с чем ее тестирование весьма затруднено.

В C++ предусмотрено три ключевых слова для обработки исключительных ситуаций: `try` (контролировать), `catch` (ловить) и `throw` (генерировать, порождать, бросать, посылать, формировать). Код, способный сгенерировать исключение, должен исполняться внутри фигурных скобок, следующих за ключевым словом `try`. Если блок `try` обнаруживает исключение внутри этого блока кода, происходит программное прерывание и выполняется такая последовательность действий:

1. Программа ищет подходящий обработчик исключения.
2. Если обработчик найден, стек очищается и управление передается обработчику исключения.
3. Если обработчик не найден, вызывается функция `terminate` для завершения программы.

Возникшая исключительная ситуация обрабатывается внутри блока `catch`, который должен следовать непосредственно за блоком `try`. Для каждого исключения необходимо предусмотреть свой обработчик, иначе может возникнуть ненормальное завершение программы. Обработчики исключений просматриваются по порядку и управление передается тому обработчику, тип аргумента которого совпадает с типом вызвавшего этот обработчик исключения. Если в теле обработчика исключения не содержится операторов перехода, то выполнение программы продолжается с точки, следующей непосредственно за последним блоком `catch`.

Блоки `try` и `catch` имеют следующие форматы:

```
try
  {
// Код, генерирующий исключение
  }
catch ( Type   X )
  {
// Обработчик исключения X типа Type, которое могло быть
// ранее сгенерировано внутри предыдущего блока try
  }

// Обработчики других исключений предыдущего блока try
catch(…)
  {
// Обработчик любого исключения предыдущего блока try
  }
```

Размеры блока `try` могут колебаться в больших пределах: от нескольких инструкций до всей функции `main()`. В последнем случае вся программа будет охвачена обработкой исключений.

Инструкция `throw` имеет следующую форму записи:

```
throw исключение;
```

В этой инструкции исключение означает сгенерированное значение. Выполнение этой инструкции должно происходить либо внутри блока `try`, либо в функции, вызванной из блока `try`. В случае генерации исключения, которому не соответствует ни одна инструкция `catch`, может произойти аварийное завершение программы. При возникновении такого необработанного исключения вызывается функция `terminate()`, которая по умолчанию вызывает функцию `abort()`, приводящую к завершению выполнения программы.

Рассмотрим листинг 21.1, поясняющий принципы обработки исключительных ситуаций.

Листинг 21.1. Пример обработки исключений

```
#include "stdafx.h"
#include <iostream>
using namespace std ;

int _tmain(int argc, _TCHAR* argv[])
{
   cout << "Programme start\n";
   try   // start of try block
     {
         cout << "Inside of try block\n";
         throw 13;  // error generation
         cout << "This line will not receive the control ";
     }
   catch (int exception)
     {
         cout << "Catched error with value ";
         cout << exception << "\n";
     }
   cout << "End of programme";
   return 0;
}
```

В результате работы программы на экране можно наблюдать следующее:

```
Programme start
Inside of try block
Catched error with value 13
End of programme
```

Как отмечалось, тип исключения должен соответствовать типу, указанному в инструкции `catch`. Если, например, исправить вторую строку исходного кода в инструкции `try` на

```
throw 13.0 ;
```

то программа выдаст на экран следующее:

```
Programme start
Inside of try block
Abnormal program termination
```

Исключение может быть сгенерировано из функции, вызванной из блока `try` (листинг 21.2).

Листинг 21.2. Пример генерирования исключения из функции

```
#include "stdafx.h"
#include <iostream>
using namespace std ;

void test_exception(int t)
{
   cout << "Inside of test_exception\n";
   if (!t) throw t;
}

int _tmain(int argc, _TCHAR* argv[])
{
   cout << "Programme start\n";
   try   // start of try block
     {
         cout << "Inside of try block\n";
         test_exception(2);
         test_exception(1);
         test_exception(0);
     }
   catch (int j)
```

```
    {
      cout << "Error: j=" << j;
    }
  cout << "\nEnd of programme";
  return 0;
}
```

В результате получим:

```
Programme start
Inside of try block
Inside of test_exception
Inside of test_exception
Inside of test_exception
Error: j=0
End of programme
```

Как видим из текста программы, внутри функции `test_exception(int t)` выполняется генерация исключения, если значение аргумента при обращении к этой функции равно нулю.

Управление обработкой исключений

В C++ имеется ряд дополнительных возможностей для управления обработкой исключительных ситуаций. Они позволяют сделать ее более гибкой и эффективной.

В некоторых случаях возникает необходимость перехватывать все исключения. Для реализации этой возможности используется инструкция вида:

```
catch(...)
{
    // обработка исключения любого типа
}
```

Эта инструкция должна описываться в программе последней из операторов `catch`, следующих за блоком `try`, иначе не будут выполняться обработчики исключений, находящиеся за этой инструкцией (листинг 21.3).

Листинг 21.3. Пример использования `catch(…)`

```
#include "stdafx.h"
#include <iostream>
using namespace std ;
```

```
void handle_exception(int t)
{
   try
      {   // Begin of try block
          if (!t)   throw t;          // generation int
          if (t==1) throw 2.06;       // generation double
          if (t==2) throw " char*";  // generation char *
      }
   catch(char *s)                     // handler char *
      {
          cout << "Exception throw " << s << " (t=2)\n";
      }
   catch(double d)                      // handler double
      {
          cout << "Exception throw  " << d << " (t=1)\n";
      }
   catch(...)                            // handler of any exception
      {
          cout << "Error in programme\n";
      }

}
int _tmain(int argc, _TCHAR* argv[])
{
   cout << "Programme start\n";
   cout << "Inside of main() function\n";
   handle_exception(2);
   handle_exception(1);
   handle_exception(0);
   cout << "End of programme\n";
   return 0;
}
```

В результате выполнения программы на экран будут выведены следующие строки:

```
Programme start
Inside of main() function
Exception throw char* (t=2)
```

```
Exception throw 2.06  (t=1)
Error in programme
End of programme
```

Как видим, обработчик `catch(...)` вызывается в случаях, когда не найдется другого обработчика соответствующего типа исключения.

Может возникнуть необходимость повторной генерации исключения из блока, который обрабатывает исключение. Это достигается путем указания оператора `throw` без параметров (листинг 21.4). В этом случае текущее исключение будет передано для обработки во внешнюю последовательность `try/catch`.

Листинг 21.4. Пример повторной генерации исключений

```
#include <iostream>
#include "stdafx.h"
#include <iostream>
using namespace std ;

void hand_exception(int x)
{
   try
      {
         if(!x) throw x;      // generation int
      }
   catch(int)                        // catch int
      {
          cout << "Catch exception inside of hand_exception\n";
          throw;    // repeated generation of  exception
      }
}

int main()
{
   cout << "Programme start\n";
   try              // begin of try block
      {
         hand_exception(0);
      }
   catch(int)
```

```
    {
        cout << "Catch exception inside of main()\n";
    }
  cout << "End of programme";
  return 0;
}
```

В результате выполнения программы на экране получим следующее:

```
Programme start
Catch exception inside of hand_exception
Catch exception inside of main()
End of programme
```

Как видим, повторно сгенерированное исключение обрабатывается в функции `main()` просто путем выдачи соответствующего сообщения.

Обычно в обработчике `catch()` первоначального исключения предусматривается проверка возможности его полной обработки, в этом случае оно обрабатывается и управление передается в точку вызова, в противном случае (если полная обработка невозможна) выполняется некоторая частичная его обработка и повторная генерация исключения с помощью `throw;`.

Рассмотрим общий пример использования обработки исключительных ситуаций (листинг 21.5).

Листинг 21.5. Пример обработки исключений

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std ;

int main()
{
  double a,b,c,num_a = 200.55, num_b=20.0;
  char str1[80], str2[] = "Checking";
  try
   {
    // запись данных в файл
    ofstream fout("test", ios::out | ios::binary);
    if(!fout)
       throw "Can't open file\n";
```

```
    fout.write((char *) &num_a, sizeof(double));
    fout.write((char *) &num_b, sizeof(double));
    fout.write(str2, strlen(str2));
    fout.close();
    // считывание данных из файла
    ifstream fin("test", ios::in | ios::binary);
    if (!fin)
         throw "Can't open file\n";
    fin.read((char *) &a, sizeof(double));
    fin.read((char *) &b, sizeof(double));
    fin.read(str1, 9);
    fin.close();
    str1[8]='\0';
    cout << "a= " << a << "\n" << "b= " << b << "\n";
       cout << "str1= " << str1 << "\n";
       if(!b)
      throw "Zero devide!";
    c = a/b;
    if(printf("Result of division:%f",c)  <= 0)
      throw "Output error!";
   }
   // обработчик исключений
 catch(char * s)
   {
    puts(s);
   }

  return 0;
}
```

Программа выполняет запись в файл двух чисел и символьной строки, затем считывание из файла двух чисел и строки и деление первого числа на второе. При выполнении программы на экран будут выведены следующие строки:

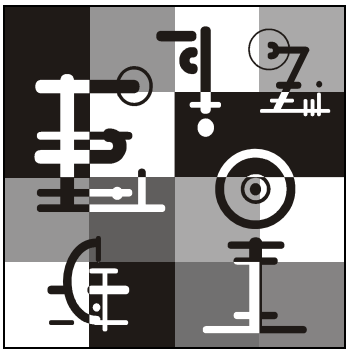
```
a=200.55
b=0.0
str1= Checking
Result of division: 10.027500
```

Ошибки в программе могут возникнуть в следующих случаях: ошибка открытия файла в режиме для записи и в режиме для чтения; ошибка при делении на ноль и ошибка при выводе результата деления на экран. Например, если

в приведенном тексте программы переменной num_b присвоить значение 0.0, а не 20.0, то при выполнении программы на экран будут выведены следующие строки:

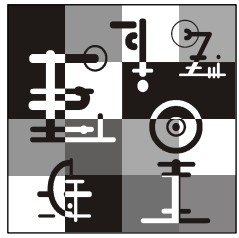
```
a=200.55
b=0.0
str1= Checking
Zero devide!
```

Обработка исключительных ситуаций в C++ представляет собой хорошо структурированное средство для обнаружения ошибок времени исполнения. Используя этот подход, можно обезопасить себя от всякого рода неожиданностей и повысить надежность программ. Следует иметь в виду, что интенсивное применение обработки исключений приводит к большим непроизводительным затратам по размеру кода и по времени выполнения.



Часть V

Приложения API



Глава 22

Характеристика приложений API Windows

Варианты приложений Windows

Под приложениями API обычно понимают приложения, интерфейс которых основан на использовании функций API (Application Programming Interface) операционной системы. Естественно, мы рассматриваем создание приложений с использованием функций API семейства операционных систем Windows, поэтому они получили название приложений API Windows.

Отметим, что с помощью системы программирования MVC++ приложения могут создаваться с применением процедурно-ориентированного и объектно-ориентированного подходов. Приложения API Windows создаются на основе процедурно-ориентированного подхода с помощью функций API. При создании приложений на основе объектно-ориентированного подхода используются компоненты библиотеки MFC (Microsoft Foundation Classes).

При применении любого из названных подходов, в отличие от консольных приложений, можно использовать всевозможные элементы управления графического интерфейса пользователя Windows: диалоговые окна, меню, кнопки, переключатели и др. Вид интерфейса простейшего приложения API Windows, созданного с помощью Мастера приложений, приведен на рис. 22.1. Как видим, интерфейс такого приложения составляет диалоговое окно с кнопками управления и меню.

API Windows обеспечивает взаимодействие между приложениями и операционной системой. В состав API Windows входит множество функций, сообщений, макросов и предопределенных констант. С его помощью обеспечивается управление окнами и приложениями, а также поддержка графики.

Рис. 22.1. Вид интерфейса простейшего приложения API Windows

Графический интерфейс приложений Windows

Для приложений Windows (API и MFC) обеспечивается универсальное графическое представление информации на экране и на других устройствах ввода-вывода (таких как принтер). В качестве основного устройства вывода в приложениях Windows используется диалоговое окно. Операционная система обеспечивает приложения набором функций ввода-вывода на экран и вывода на печать, а также управляет всем выводом.

В частности, Windows обеспечивает приложения набором графических объектов, предназначенных для управления выводом. К числу важнейших объектов можно отнести следующие объекты:

- карандаши (pens) — задают параметры рисования линий (цвет, толщина, стиль и др.);
- кисти (brushes) — задают параметры заливки замкнутых контуров, например, цвета и стиля;
- шрифты (fonts) — задают параметры вывода текста, в частности, гарнитуру (название) шрифта, размер символов и т. п.;
- битовые массивы (bitmaps) — задают массивы точек, формирующих растровые изображения;
- логические палитры (logical palettes) — содержат список цветов приложения и обеспечивают интерфейс между приложением и цветным устройством вывода, таким как дисплей.

Важным достоинством API Windows является независимость графического интерфейса вывода от устройства. Достигается это с помощью двух библио-

тек динамической компоновки (DLL, Dynamic Link Library): одна обеспечивает графический интерфейс устройства (GDI, Graphics Device Interface), вторая служит драйвером устройства соответствующего типа (например, для вывода в диалоговое окно или на принтер). Перед операцией вывода на внешнее устройство приложение запрашивает графический интерфейс устройства о загрузке драйвера. После загрузки драйвера приложение Windows может настроить ряд параметров вывода, которые мы приводили при указании графических объектов, используя так называемый контекст устройства.

Контекст устройства

Контекст устройства (device context) представляет собой структуру, в которой определяется набор графических объектов, связанные с ними параметры вывода и графические режимы. Приложение Windows не имеет прямого доступа к контексту устройства, для настройки параметров вывода осуществляется вызов соответствующих функций API.

API Windows задает 4 типа контекстов устройств:

- ❒ экрана (display) — для рисования на экране;
- ❒ принтера (printer) — для вывода на принтер или плоттер;
- ❒ памяти (memory) — для рисования без вывода на экран;
- ❒ информации (information) — для получения данных об устройстве.

С помощью контекста устройства приложение Windows может осуществлять различные операции с графическими объектами: перечислять, устанавливать новые и удалять графические объекты; сохранять и восстанавливать графические объекты с их параметрами и графическими режимами.

Для экрана Windows API обеспечивает 3 типа контекста устройства:

- ❒ контекст класса;
- ❒ общий контекст устройства;
- ❒ частные контексты устройства.

Контекст класса служит для обеспечения совместимости с предшествующими версиями Windows. Частные контексты устройства используются в приложениях, ориентированных на работу с графикой, например, в настольных издательских системах или системах автоматизированного проектирования. Общий контекст устройства используется в остальных приложениях.

Общий контекст устройства для экрана приложение получает с помощью вызова функций `BeginPaint`, `GetDC` или `GetDCEx`. Контекст устройства инициализируется значениями по умолчанию, при необходимости их можно изменить

с помощью специальных функций. После завершения операций вывода нужно сообщить об этом операционной системе с помощью функции `EndPaint` или `ReleaseDC`. При этом изменения, внесенные приложением в контекст устройства, отменяются.

Для принтера (лазерного, струйного или матричного) и для графопостроителя Windows обеспечивает один контекст устройства. Для принтера контекст устройства приложение получает с помощью функции `CreateDC`. При этом задаются параметры: имя драйвера, имя принтера, файла или другого устройства вывода. После завершения печати контекст устройства должен быть удален с помощью функции `ReleaseDC`.

Состав приложения. Функция *WinMain*

В состав приложения API Windows входят две основные части: функция `WinMain` и функция окна, или оконная процедура `WndProc`. Рассмотрим, какую роль в составе приложения играют эти части.

Функция `WinMain` включает три основные составляющие:

- функцию `ATOM MyRegisterClass(HINSTANCE hInstance)` регистрации класса окна;
- функцию `BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)`, в которой выполняется инициализация приложения и создание главного окна приложения;
- стандартный цикл обработки сообщений.

Примечание

В приложениях API Windows, создаваемых в среде MVC++ 2005, в качестве точки входа в приложения используется функция `int APIENTRY _tWinMain()`. Это связано с ограничениями, накладываемыми компоновщиком.

Рассмотрим названные составляющие функции `WinMain`.

Регистрация класса окна

В приложениях Windows используются диалоговые окна, которые очень часто имеют много одинаковых параметров. В связи с этим для экономии программного кода диалоговоые окна рассматриваются как объекты, которые при создании имеют класс окна с общими для всего класса параметрами. После создания конкретного окна параметры соответствующего ему объекта при необходимости можно настраивать.

Чтобы иметь возможность использовать класс окна, его нужно зарегистрировать. Ниже (листинг 22.1) приводится пример функции регистрации класса окна.

Листинг 22.1. Пример функции регистрации класса окна

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
       WNDCLASSEX wcex;

       wcex.cbSize = sizeof(WNDCLASSEX);

       wcex.style            = CS_HREDRAW | CS_VREDRAW;
       wcex.lpfnWndProc      = WndProc;
       wcex.cbClsExtra       = 0;
       wcex.cbWndExtra       = 0;
       wcex.hInstance        = hInstance;
       wcex.hIcon            = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_API_2006));
       wcex.hCursor          = LoadCursor(NULL, IDC_ARROW);
       wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
       wcex.lpszMenuName     = MAKEINTRESOURCE(IDC_API_2006);
       wcex.lpszClassName    = szWindowClass;
       wcex.hIconSm          = LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_SMALL));

       return RegisterClassEx(&wcex);
}
```

При регистрации класса окна используется структура `WNDCLASSEX`, определенная следующим образом:

```
typedef struct {
    UINT style;             // стиль
    WNDPROC lpfnWndProc;    // указатель на оконную процедуру
    int cbClsExtra;         // размер дополнительной памяти для WNDCLASS
    int cbWndExtra;         // размер дополнительной памяти для окна
    HINSTANCE hInstance;    // дескриптор приложения
    HICON hIcon;            // дескриптор пиктограммы окна
    HCURSOR hCursor;        // дескриптор курсора класса
    HBRUSH hbrBackground;   // дескриптор кисти фона класса
```

```
    LPCTSTR lpszMenuName;    // указатель на меню окна
    LPCTSTR lpszClassName;   // указатель на строку с именем класса окна
} WNDCLASS, *PWNDCLASS;
```

В комментариях указано назначение каждого члена-данного этой структуры.

Инициализация приложения и создание главного окна

Инициализация приложения выполняется с помощью функции `BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)`. Первый параметр этой функции задает дескриптор экземпляра приложения, второй служит для управления способом отображения окна.

Для создания главного окна приложения используется функция `CreateWindow` с прототипом вида:

```
HWND CreateWindow(
    LPCTSTR lpClassName,     // имя класса
    LPCTSTR lpWindowName,    // имя окна
    DWORD dwStyle,           // стиль окна
    int x,                   // начальная горизонтальная координата окна
    int y,                   // начальная вертикальная координата окна
    int nWidth,              // ширина окна
    int nHeight,             // высота окна
    HWND hWndParent,  // дескриптор родительского окна или владельца
    HMENU hMenu,             // дескриптор меню
    HINSTANCE hInstance,     // дескриптор приложения
    LPVOID lpParam           // указатель на передаваемое значение
);
```

В комментариях к приведенной структуре указано назначение ее членов-данных. В листинге 22.2 приводится пример инициализации приложения и создания главного окна.

Листинг 22.2. Пример инициализации приложения и создания главного окна

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // дескриптор приложения

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
```

```
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}
```

Как видим, при инициализации приложения сначала создается главное окно приложения с помощью функции `CreateWindow()`. При этом третий аргумент `WS_OVERLAPPEDWINDOW` указывает на стиль окна с перекрытием, значение `CW_USEDEFAULT` четвертого аргумента указывает, что *X*-координата окна выбирается по умолчанию, *Y*-координата игнорируется.

Аналогично значение `CW_USEDEFAULT` шестого аргумента указывает, что ширина окна устанавливается по умолчанию, а аргумент, указывающий высоту окна, не учитывается.

Отображение диалогового окна обеспечивается путем вызова функции `ShowWindow()`. При этом первый аргумент `hWnd` задает дескриптор окна, второй аргумент `nCmdShow` (передается как параметр функции `InitInstance()`) определяет способ отображения окна.

С помощью функции `UpdateWindow()` выполняется обновление содержимого окна для заданного в качестве аргумента функции дескриптора окна `hWnd`.

Цикл обработки сообщений

Управление выполнением приложения Windows пользователем основано на обработке сообщений, которые могут поступать от клавиатуры и мыши. Операционная система Windows является многозадачной, т. е. допускающей выполнение нескольких приложений одновременно. В связи с этим в ней все сообщения поступают не приложению, а помещаются в очередь сообщений.

Для обработки сообщений из очереди обычно используется стандартный цикл обработки, который приводится в листинге 22.3.

Листинг 22.3. Пример стандартного цикла обработки сообщений

```
BOOL bRet;
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0 )
{
   if (bRet == -1 )
   {
      // handle the error and possibly exit
   }
   else
   {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
   }
}
```

Функция `GetMessage()` выполняет копирование сообщения из очереди в структуру, указываемую с помощью первого фактического параметра `&msg`. Значение `NULL` второго фактического параметра означает возможность передачи сообщения из потока вызовов любому окну. Третий и четвертый параметры позволяют установить соответственно минимальное и максимальное значения для кодов сообщений (фильтр). При нулевых значениях параметров фильтр не устанавливается.

Функция `TranslateMessage()` транслирует виртуальные коды клавиш в ASCII-значения символьных сообщений `WM_CHAR`.

Функция `DispatchMessage()` посылает каждое сообщение соответствующей оконной процедуре.

Схематично процесс обработки сообщений в цикле показан на рис. 22.2.

Сообщение представляет собой следующую структуру данных:

```
typedef struct tagMSG
{
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSGMSG;
```

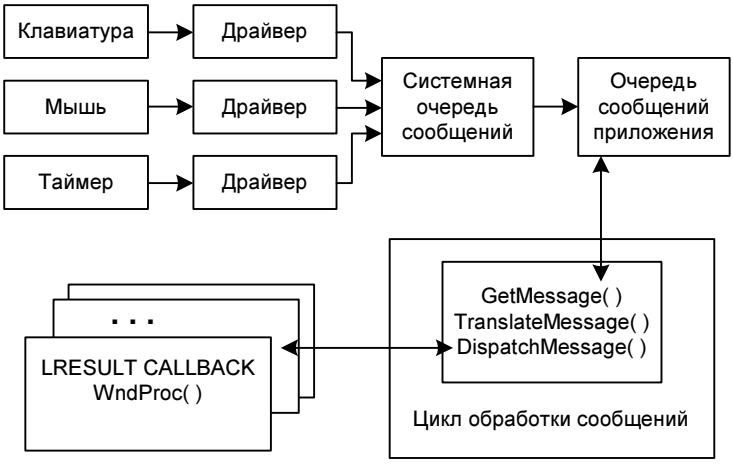


Рис. 22.2. Схема цикла обработки сообщений

Эта структура содержит уникальный для Windows код сообщения `message` и другие параметры, отражающие: адресат сообщения `hwnd`, содержимое сообщения `wParam` и `lParam`, время отправления `time` и информацию о координатах `pt`.

Оконная процедура обработки сообщений

Оконная процедура обратного вызова (описатель `CALLBACK`) создается для каждого окна приложения API Windows и предназначена для обработки сообщений, поступающих этому окну. Для главного окна приложения соответствующая процедура формируется автоматически при создании приложения. Пример такой процедуры с именем `WndProc` приведен в листинге 22.4. После создания заготовки приложения оконную процедуру главного окна дополняют и настраивают с учетом функциональности приложения.

Листинг 22.4. Пример оконной процедуры для главного окна приложения

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
        int wmId, wmEvent;
        PAINTSTRUCT ps;
        HDC hdc;
```

```
        switch (message)
        {
        case WM_COMMAND:
                wmId    = LOWORD(wParam);
                wmEvent = HIWORD(wParam);
                // Разбор выбора команд меню:
                switch (wmId)
                {
                case IDM_ABOUT:
                        DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX),
↳hWnd, About);
                        break;
                case IDM_EXIT:
                        DestroyWindow(hWnd);
                        break;
                default:
                        return DefWindowProc(hWnd, message, wParam,
↳lParam);
                }
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                // Место для кода рисования в окне ...
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

С помощью оконной процедуры чаще всего обрабатываются следующие типы оконных сообщений:

- `WM_COMMAND` — команды меню приложения;
- `WM_PAINT` — рисования в окне;
- `WM_DESTROY` — фиксация сообщения выхода и возврата.

Оконные сообщения имеют префикс `WM_` (Window Message) и рассматриваются как символьные константы.

Функция `DefWindowProc()` в оконной процедуре вызывается в блоке `default` при поступлении сообщений, для которых обработка не предусмотрена. При этом происходит освобождение очереди от таких сообщений. Тем самым обеспечивается корректная обработка остальных сообщений.

Как видим из текста приведенной оконной процедуры, при поступлении сообщения типа `WM_COMMAND` с идентификатором команды `IDM_ABOUT` происходит вызов соответствующего диалогового окна:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
```

Пример оконной процедуры обратного вызова для этого диалогового окна приводится в листинге 22.5.

Листинг 22.5. Пример оконной процедуры для диалогового окна

```
// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
        UNREFERENCED_PARAMETER(lParam);
        switch (message)
        {
        case WM_INITDIALOG:
                return (INT_PTR)TRUE;

        case WM_COMMAND:
                if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
                {
                        EndDialog(hDlg, LOWORD(wParam));
                        return (INT_PTR)TRUE;
                }
                break;
        }
        return (INT_PTR)FALSE;
}
```

Эта процедура выполняет обработку сообщений, поступающих к его элементам управления (командам меню и кнопкам).

Пример заготовки приложения

Для полноты картины приведем пример заготовки приложения API Windows, генерируемой Мастером приложений Win32 (листинг 22.6).

Листинг 22.6. Пример заготовки приложения API Windows

```
// API_2006.cpp : Точка входа приложения.
//

#include "stdafx.h"
#include "API_2006.h"

#define MAX_LOADSTRING 100

// Глобальные переменные:
HINSTANCE hInst;                    // текущий экземпляр
TCHAR szTitle[MAX_LOADSTRING]; // строка заголовка
// имя класса основного окна
TCHAR szWindowClass[MAX_LOADSTRING];

// Опережающие объявления функций модуля:
ATOM                        MyRegisterClass(HINSTANCE hInstance);
BOOL                        InitInstance(HINSTANCE, int);
LRESULT CALLBACK      WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK      About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
       UNREFERENCED_PARAMETER(hPrevInstance);
       UNREFERENCED_PARAMETER(lpCmdLine);

       // TODO: Place code here.
       MSG msg;
       HACCEL hAccelTable;

       // Инициализация глобальных строк
```

```
        LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
        LoadString(hInstance, IDC_API_2006, szWindowClass,
MAX_LOADSTRING);
        MyRegisterClass(hInstance);

        // Инициализация приложения:
        if (!InitInstance (hInstance, nCmdShow))
        {
                return FALSE;
        }

        hAccelTable = LoadAccelerators(hInstance,
MAKEINTRESOURCE(IDC_API_2006));

        // Основной цикл обработки сообщений:
        while (GetMessage(&msg, NULL, 0, 0))
        {
                if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }

        return (int) msg.wParam;
}



//
//  Функция регистрации класса окна:
ATOM MyRegisterClass(HINSTANCE hInstance)
{
        WNDCLASSEX wcex;

        wcex.cbSize = sizeof(WNDCLASSEX);

        wcex.style                      = CS_HREDRAW | CS_VREDRAW;
        wcex.lpfnWndProc      = WndProc;
```

```
        wcex.cbClsExtra             = 0;
        wcex.cbWndExtra             = 0;
        wcex.hInstance        = hInstance;
        wcex.hIcon                  = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_API_2006));
        wcex.hCursor          = LoadCursor(NULL, IDC_ARROW);
        wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
        wcex.lpszMenuName     = MAKEINTRESOURCE(IDC_API_2006);
        wcex.lpszClassName    = szWindowClass;
        wcex.hIconSm          = LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_SMALL));

        return RegisterClassEx(&wcex);
}

//
//   Функция инициализации экземпляра приложения и создания
//   основного окна
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Сохранение дескриптора приложения в
                      // глобальной переменной
   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
```

```
//  Оконная процедура обработки сообщений
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
        int wmId, wmEvent;
        PAINTSTRUCT ps;
        HDC hdc;

        switch (message)
        {
        case WM_COMMAND:
                wmId    = LOWORD(wParam);
                wmEvent = HIWORD(wParam);
                // Разбор команд меню:
                switch (wmId)
                {
                case IDM_ABOUT:
                        DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX),
↳hWnd, About);
                        break;
                case IDM_EXIT:
                        DestroyWindow(hWnd);
                        break;
                default:
                        return DefWindowProc(hWnd, message, wParam,
↳lParam);
                }
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                // место для кода рисования...
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
```

```
        return 0;
}


// Обработчик сообщений для диалогового окна about.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
        UNREFERENCED_PARAMETER(lParam);
        switch (message)
        {
        case WM_INITDIALOG:
                return (INT_PTR)TRUE;

        case WM_COMMAND:
                if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
                {
                        EndDialog(hDlg, LOWORD(wParam));
                        return (INT_PTR)TRUE;
                }
                break;
        }
        return (INT_PTR)FALSE;
}
```

Как видим из представленного листинга, в нем присутствуют все составляющие приложения API Windows, которые мы рассматривали ранее.

Шаги создания приложения API

Для создания приложения API Windows (Win32) в среде MVC++ нужно выполнить следующие шаги.

1. Задать команду **File/New/Project** основного меню MVC++.
2. В открывшемся диалоговом окне **New Project** указать имя рабочей области и имя проекта (на рис. 22.3 **APISolution** и **APIProject**), выбрать тип проекта **Win32** и шаблон приложения Win32 **Project**.
3. Мастером приложений Win32 Application Wizard при необходимости внести уточняющие настройки. Нажать **Finish**.

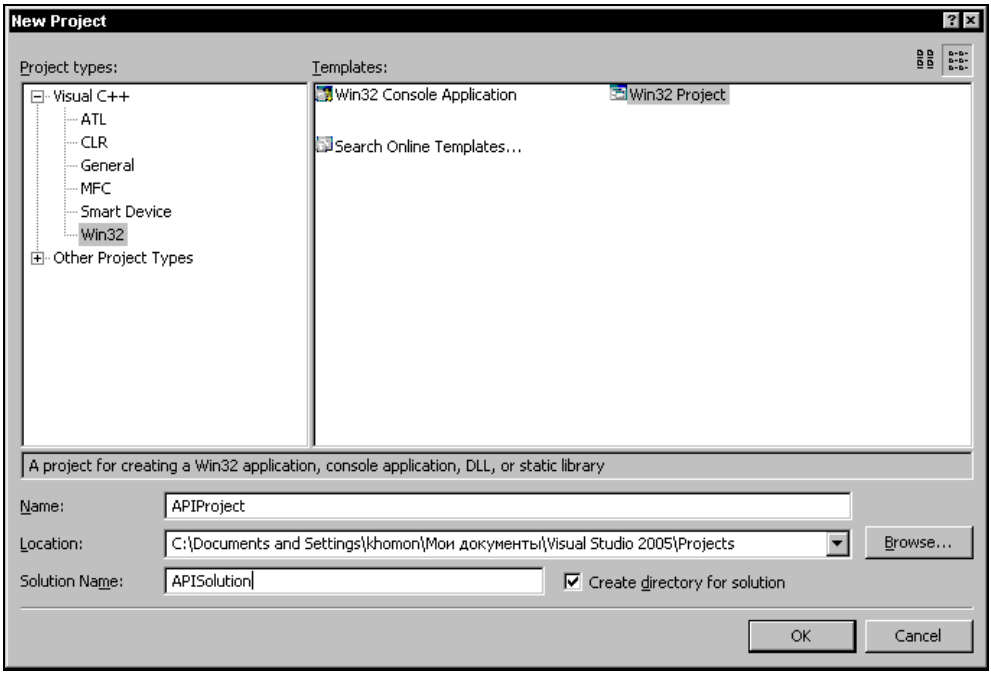
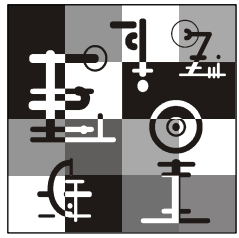


Рис. 22.3. Диалог задания типа приложения Win32

4. Дополнить созданный проект приложения нужными ресурсами: меню, диалоговыми окнами и другими элементами управления.
5. Задать обработчики сообщений и настроить функциональность приложения.



Глава 23

Разработка интерфейса приложения

Интерфейс приложений Windows составляют меню, диалоговые окна и размещаемые на них элементы управления. Все они являются ресурсами, для разработки и настройки которых применяются соответствующие редакторы ресурсов.

Создание меню

Меню представляет собой сгруппированный набор команд, с помощью которых можно выполнять требуемые действия при выполнении приложения. При разработке приложения API Win32 с помощью Мастера приложений Win32 в нем автоматически создается меню с двумя пунктами: **File** и **Help**. Фактически требуемое меню нужно создавать заново: дополнить названное меню набором команд с учетом функциональности приложения.

Для создания меню в среде MVC++ нужно выполнить следующие действия.

1. При открытом проекте приложения Win32 задать команду **View/Resource View** основного меню MVC++.
2. В открывшемся окне **Resource View** двойным щелчком мыши раскрыть папку **<Имя проекта>.rc**, раскрыть папку **Menu** и выбрать в ней ресурс меню.
3. В окне Конструктора меню (рис. 23.1) с помощью мыши дополнить имеющееся меню нужными пунктами и командами.
4. Для каждой команды созданного меню нужно включить соответствующий код в состав оконной процедуры обработки сообщений, чтобы обеспечить нужное действие при задании этой команды.

Рассмотрим несколько подробнее детали указанных действий.

Имя команды вводится с помощью клавиатуры непосредственно в поле команды, оно присваивается как значение свойству **Caption** команды меню (в правой

части окна). Если перед некоторой буквой в названии команды указать символ &, то эта буква получит начертание с подчеркиванием, как на рис. 23.1, и будет играть роль "горячей" клавиши для вызова команды. В частности, это означает, что команду с именем **Exit** при раскрытом пункте меню **File** можно вызвать с помощью "горячей" клавиши <x>.

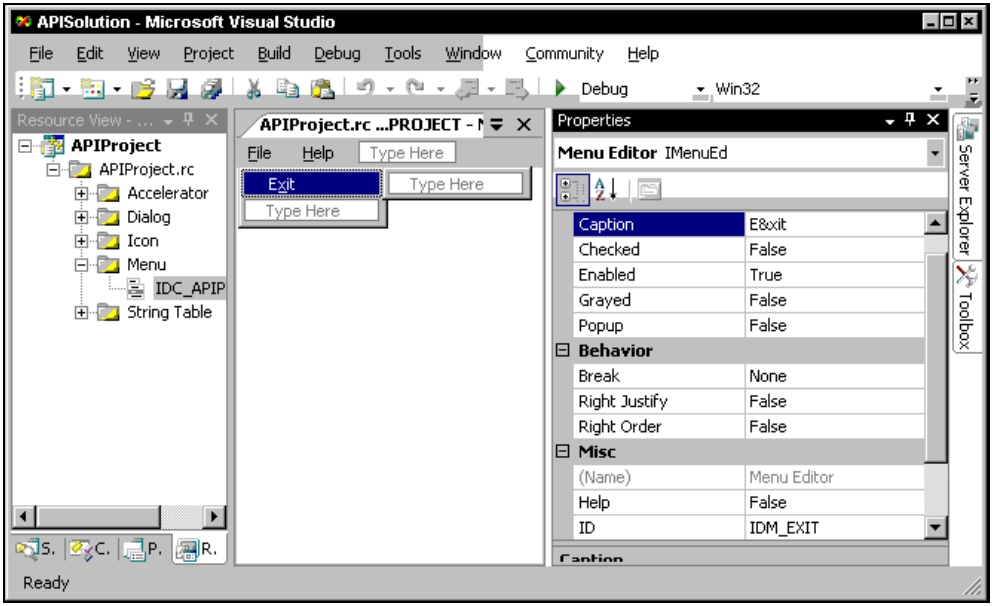


Рис. 23.1. Вид Конструктора при разработке меню

В состав оконной процедуры обработки сообщений включается идентификатор команды, который является значением свойства `ID` этой команды. Например, как показано на рисунке, для команды с именем **Exit** имеет место `ID=IDM_EXIT`.

Примечание

У идентификаторов в ресурсах назначают следующие префиксы: `IDM` — для идентификаторов команд меню; `IDD` — для идентификаторов диалоговых окон.

Для каждой команды меню предусматривается свое действие, указываемое в оконной процедуре обработки сообщений. Например, чтобы обеспечить нужное действие при задании команды **Exit**, мы должны поместить в состав оконной процедуры обработки сообщений код, приведенный в листинге 23.1.

Листинг 23.1. Пример задания реакции на выполнение команды

```
switch (message)
        {
        case WM_COMMAND:
                wmId    = LOWORD(wParam);
                wmEvent = HIWORD(wParam);
                // Разбор команд меню:
                switch (wmId)
                {
                . . .
                case IDM_EXIT:
                        DestroyWindow(hWnd);
                        break;
                default:
                        return DefWindowProc(hWnd, message, wParam, lParam);
                }
                break;
                . . .
```

Теперь при задании команды **Exit** будет происходить обращение к функции `DestroyWindow(hWnd)`, вызывающей закрытие главного диалогового окна приложения. Эта функция разрушает оконное меню, освобождает системную очередь, удаляет содержимое буфера обмена. Таким образом, происходит завершение работы приложения.

Создание диалогового окна

Диалоговые окна чаще всего служат для организации обмена информацией между пользователем и приложением. В диалоговых окнах размещаются элементы управления, такие как кнопки, флажки, переключатели, поля и др. С помощью диалоговых окон обычно пользователю выдаются запросы на подтверждение или отмену некоторых действий, а также осуществляется ввод данных для приложения.

Важной характеристикой окна является стиль окна. Стиль класса окна определяет внешний вид и поведение всех окон, созданных на базе данного класса. Напомним, что он определяется при регистрации класса:

```
wc.style = CS_HREDRAW | CS_VREDRAW;
```

Стиль окна служит для уточнения внешнего вида и поведения окна. Различают три основных стиля окон:

- перекрывающиеся окна `WS_OVERLAPPED`;
- всплывающие, или временные окна `WS_POPUP`;
- дочерние окна `WS_CHILD`.

Перекрывающиеся окна используются в качестве главного окна приложения. Окно имеет заголовок, рамку и внутреннюю часть окна. Дополнительно окно может иметь системное меню, кнопки для максимального увеличения размера окна и для сворачивания окна в пиктограмму, вертикальную и горизонтальную полосу просмотра и меню.

При создании перекрывающегося окна при помощи функции `CreateWindow()` в качестве восьмого параметра можно указать идентификатор окна-владельца. Окно-владелец должно существовать на момент создания подчиненного окна.

При сворачивании в пиктограмму окна-владельца все окна, которыми оно владеет, становятся невидимыми. Если свернуть в пиктограмму окно, которым владеет другое окно, а затем и окно-владельца, пиктограмма подчиненного окна исчезает.

Если окно уничтожили, автоматически уничтожаются и все принадлежащие ему окна. Обычное перекрывающееся окно, не имеющее окна-владельца, может располагаться в любом месте экрана и принимать любые размеры. Подчиненные окна располагаются всегда над поверхностью окна-владельца, загораживая его. Приведем пример создания перекрывающегося окна с помощью функции `CreateWindow()` (листинг 23.2).

Листинг 23.2. Пример создания перекрывающегося окна

```
OwnedHwnd = CreateWindow(
   szMainClassName,        // имя класса окна
   "Перекрывающееся окно", // заголовок окна
   WS_OVERLAPPEDWINDOW,    // стиль
   100, 100, 200, 100,     // расположение и размеры
   MainHwnd,               // идентификатор родительского окна
   0,                      // идентификатор меню
   hInstance,              // идентификатор приложения
   NULL);                  // указатель на дополнительные параметры
```

Временные окна используются для вывода информационных сообщений и остаются на экране непродолжительное время. Временные окна могут не иметь заголовок (title bar). Временные окна могут иметь окно-владельца

и могут сами владеть другими окнами. В общем случае временные окна можно рассматривать как разновидность перекрывающихся окон. Приведем пример создания временного окна (листинг 23.3).

Листинг 23.3. Пример создания временного окна

```
PopUpHwnd = CreateWindow(
   szPopUpClassName,        // имя класса окна
   "Временное окно",        // заголовок окна
   WS_POPUPWINDOW | WS_CAPTION | WS_VISIBLE,  // стиль
   100, 100, 200, 100,      // расположение и размеры окна
   MainHwnd,                // идентификатор родительского окна
   0,                       // идентификатор меню
   hInstance,               // идентификатор приложения
   NULL);                   // указатель на дополнительные параметры
```

В листинге 23.3 мы установили стиль окна как временное, с заголовком и видимое сразу после создания.

Дочерние окна ограничиваются клиентской областью родительского окна. Дочернее окно должно иметь родительское окно. Дочернее окно не может выходить за пределы клиентской области родительского окна и перемещается только вместе с ним. Обычно приложение использует дочерние окна для разделения клиентской области родительского окна на функциональные области.

В дочернем окне могут быть строка заголовка, оконное меню, кнопки минимизации и максимизации, рамка и полосы прокрутки. В нем не допускается наличие меню. Дочернее окно создается с помощью функции `CreateWindow()` так же, как и временное окно, но со стилем `WS_CHILD`.

По способу поведения по отношению к другим окнам приложения различают модальные и немодальные диалоговые окна. Если окно является модальным, то после установления его отображения доступ к другим окнам становится невозможен, вплоть до его закрытия. То есть модальное окно находится все время поверх других окон приложения. В отличие от модальных окон, при открытии немодального диалогового окна доступ к другим диалоговым окнам не ограничивается. Программирование немодальных окон заметно сложнее, чем модальных.

При создании новых и модификации имеющихся диалоговых окон проекта приложения удобно пользоваться редактором ресурсов. Для создания нового диалогового окна в среде MVC++ нужно выполнить следующие действия.

1. При открытом проекте приложения Win32 задать команду **View/Resource View** основного меню MVC++.

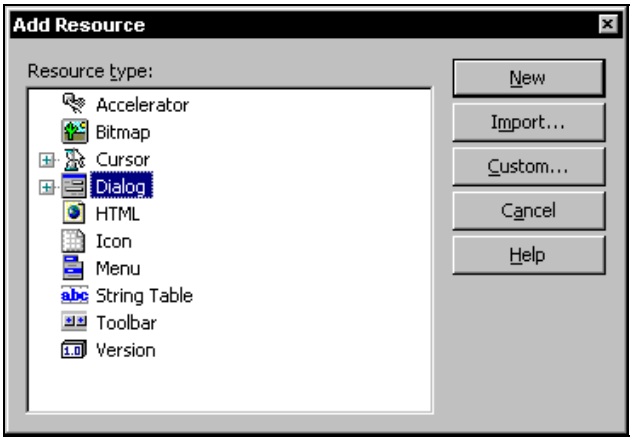


Рис. 23.2. Диалоговое окно **Add Resource**

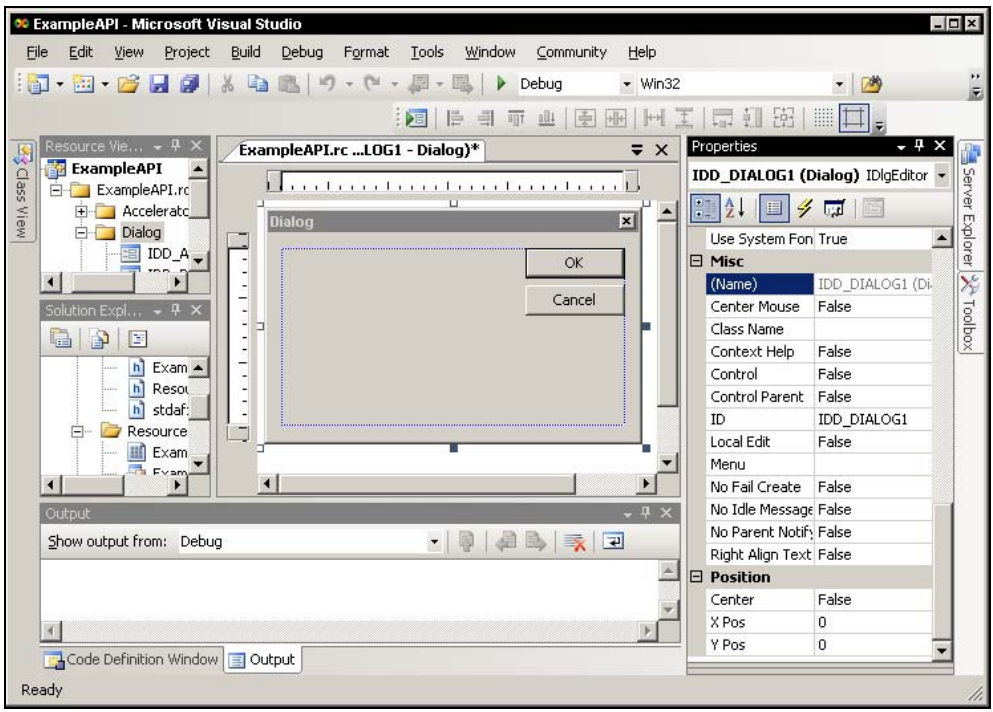


Рис. 23.3. Заготовка диалогового окна в окне Редактора ресурсов

2. В открывшемся окне **Resource View** (Просмотр ресурса) двойным щелчком мыши раскрыть папку имени проекта и выбрать папку **Dialog** (рис. 23.2).
3. Щелчком правой кнопки мыши вызвать контекстное меню и выбрать команду **Add Resource** (Добавить ресурс).
4. В открывшемся диалоговом окне **Add Resource** выбрать вариант **Dialog** (Диалог) и нажать кнопку **New** (Новый).

В результате будет создан ресурс с заготовкой диалогового окна. При этом откроется окно редактора ресурсов (рис. 23.3), в котором можно выполнить конструирование нужного диалогового окна, наполняя его элементами управления.

Теперь нам остается наполнить заготовку диалогового окна нужными элементами управления и задать для них обработку сообщений.

Элементы управления

В составе диалоговых окон можно использовать разнообразные элементы управления. В процессе создания диалоговых окон с помощью Конструктора элементы управления размещены на Панели инструментов (рис. 23.4). С помощью мыши элементы управления можно перетаскивать на создаваемое диалоговое окно. Раскрытие Панели инструментов выполняется щелчком мыши на ярлыке **Toolbox**, расположенном на правом краю окна среды или с помощью одноименной команды **Toolbox** меню **View**.

Назначение распространенных элементов управления, расположенных на Панели инструментов:

- **Pointer** (Указатель) — служит для выделения нескольких элементов управления, расположенных в диалоговом окне;
- **Button** (Кнопка) — используется для задания кнопок управления приложением, таких как **ОК**, **Cancel** и др.;
- **Check Box** (Флажок) — обычно используется в роли независимого переключателя, анализируя состояние которого можно управлять приложением;
- **Edit Control** (Простой редактор) — служит для задания однострочного или многострочного текста, который можно редактировать при выполнении приложения;
- **Combo Box** (Поле со списком) — представляет собой комбинацию поля и списка, позволяет выбирать элементы из имеющегося списка и набирать текст в поле ввода;

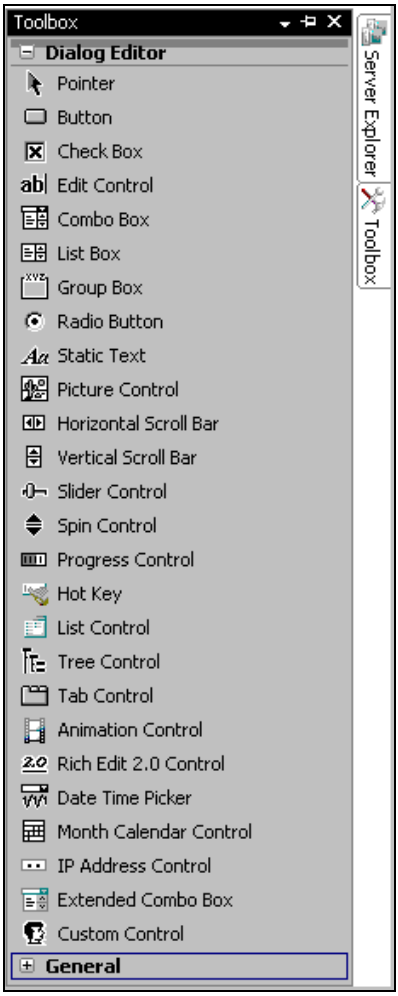


Рис. 23.4. Вид Панели инструментов

- **List Box** (Список) — служит для выбора пользователем одного элемента из имеющегося списка, который при выполнении приложения может изменяться программно;
- **Group Box** (Группа) — служит для объединения в группу нескольких зависимых переключателей;
- **Radio Button** (Переключатель) — служит для задания зависимого переключателя в составе группы;
- **Static Text** (Надпись) — служит для отображения нередактируемого текста, используемого в качестве надписи для другого элемента управления.

Элементам управления (так же как командам меню и диалоговым окнам) назначаются идентификаторы. Напомним, что идентификаторы диалоговых окон имеют префикс `IDD`, а идентификаторы элементов управления — префикс `IDC`.

Перечень идентификаторов элементов управления и других ресурсов, используемых в приложении, содержится в файле `resource.h`. В листинге 23.4 приводится содержимое такого файла для заготовки приложения API Windows, созданной с помощью Мастера приложений Win32.

Листинг 23.4. Пример содержимого файла `resource.h`

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by ExampleAPI.rc
//

#define IDS_APP_TITLE               103

#define IDR_MAINFRAME               128
#define IDD_EXAMPLEAPI_DIALOG       102
#define IDD_ABOUTBOX                103
#define IDM_ABOUT                   104
#define IDM_EXIT                    105
#define IDI_EXAMPLEAPI              107
#define IDI_SMALL                   108
#define IDC_EXAMPLEAPI              109
#define IDC_MYICON                  2
#ifndef IDC_STATIC
#define IDC_STATIC                          -1
#endif
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS

#define _APS_NO_MFC                         130
#define _APS_NEXT_RESOURCE_VALUE            129
#define _APS_NEXT_COMMAND_VALUE             32771
#define _APS_NEXT_CONTROL_VALUE             1000
#define _APS_NEXT_SYMED_VALUE               110
#endif
#endif
```

Как видим, в файле содержатся идентификаторы элементов управления и других ресурсов, справа указываются соответствующие им числовые константы, которые можно использовать в программах вместо идентификаторов.

Задание реакции на те или иные действии с элементами управления выполняется с помощью оконной процедуры обработки сообщений того диалогового окна, на котором эти элементы размещены (как в примере задания реакции на выполнение команды **Exit** в разделе, посвященном созданию и настройке меню).

Для наглядности, поясним технику работы с двумя важнейшими элементами управления: Edit Control и Combo Box.

При работе с элементом управления Edit Control никаких действий по инициализации его содержимого в ходе обработки сообщения WM_INITDIALOG (возникает непосредственно перед отображением диалогового окна) выполнять не обязательно, хотя и возможно: тем самым можно задать начальное содержимое редактора по умолчанию. Содержимое простого редактора Edit Control обычно заполняется и редактируется пользователем при выполнении приложения. При обработке сообщения `WM_COMMAND` в оконной процедуре соответствующего диалогового окна содержимое простого редактора Edit Control помещается с помощью функции `GetDlgItemText` в символьную строку. Например, так:

```
GetDlgItemText(hDlg, IDC_EDIT1, str1, 20);
```

Здесь `hDlg` — дескриптор диалогового окна, содержащего элемент управления, `IDC_EDIT1` — идентификатор нашего элемента управления Edit Control, `str1` — символьная строка, в которую помещается содержимое редактора, `20` — длина считываемой строки. Полученную символьную строку можно использовать, например, при обработке того же сообщения `WM_COMMAND` в оконной процедуре диалогового окна.

При работе с элементом управления Combo Box, наоборот, требуется выполнить инициализацию его содержимого в ходе обработки названного нами сообщения `WM_INITDIALOG`. Сделать это можно с помощью функций `SendMessage` и `GetDlgItem`, например, так:

```
int year = 1985;
char* str = new char[10]; *str = 0;
for ( int i = 0; i < 10; i++ )
{ SendMessage(GetDlgItem(hDlg,IDC_COMBO1),CB_ADDSTRING,true,↳
     (LPARAM)(_itoa(i+year,str,10)));
}
SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,true,↳
```

```
    (LPARAM)("red"));
SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,true,↳
    (LPARAM)("blue"));
SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,true,↳
    (LPARAM)("green"));
```

Здесь для элемента управления Combo Box с идентификатором `IDC_COMBO1` с помощью цикла `for` выполнено заполнение его десятью строками:

`1985, 1986, 1987, …, 1994.`

При этом функция `_itoa` используется для преобразования целочисленного значения в строку. Для элемента управления Combo Box с идентификатором `IDC_COMBO3` выполнено заполнение его тремя строками: `red`, `blue` и `green`.

В процессе выполнения приложения при открытом диалоговом окне пользователь может выбирать в каждом элементе управления Combo Box любую из сформированных указанным способом строк. Выбранные символьные строки, как и в случае простого редактора Edit Control, можно использовать при обработке сообщения `WM_COMMAND` в оконной процедуре диалогового окна. Например, так:

```
GetDlgItemText(hDlg, IDC_COMBO3, str2, 20);
GetDlgItemText(hDlg, IDC_COMBO1, str3, 20);
```

Как видим, здесь также используется функция `GetDlgItemText`.

Пример задания оконных процедур

Предположим, что в приложении API Windows имеется главное диалоговое окно и несколько дочерних диалоговых окон с именами: **About**, **Add**, **Create**, **Delete**, **Find**, **Modification** и **DlgProc**. Отсюда следует, что в главном файле проекта приложения с расширением cpp должны быть размещены следующие прототипы соответствующих оконных процедур:

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Add(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Create(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Delete(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Find(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Modification(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);
```

Кроме того, в этом же файле помещаются описания всех этих оконных процедур. Для примера приведем код оконной процедуры обработки сообщений диалогового окна **Delete** (листинг 23.5).

Листинг 23.5. Оконная процедура обработки сообщений диалогового окна

```
LRESULT CALLBACK Delete(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        char* str = new char[20];
        switch (message)
        {
        case WM_INITDIALOG:
                for (int i = 0; i < Control.Massiv->Q; i ++ ){

        SendMessage(GetDlgItem(hDlg,IDC_LIST1),LB_ADDSTRING,true,(LPARAM)C
ontrol.Massiv->Key[i]);
                }
                return TRUE;

        case WM_COMMAND:
                if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
                {
             GetDlgItemText(hDlg, IDC_LIST1, str, 20);
                        Control.Delete(str);
                        EndDialog(hDlg, LOWORD(wParam));
                        return TRUE;
                }
                break;
        }
        return FALSE;
}
```

Как видим из приведенного кода процедуры, в ней задается реакция на получение оконного сообщения `WM_INITDIALOG`, возникающего непосредственно перед открытием диалогового окна **Delete**.

Кроме того, в состав главной оконной процедуры включается код обработки сообщения (листинг 23.6), с помощью которого выполняется вызов подчиненного диалогового окна с указанием соответствующей процедуры.

Листинг 23.6. Код обработки сообщения для вызова диалогового окна

```
case IDM_DELETE:
        DialogBox(hInst, (LPCTSTR)IDD_DELETE, hWnd, (DLGPROC)Delete);
        InvalidateRect( hWnd,NULL,true);
        UpdateWindow(hWnd);
        break;
```

Здесь приведен код для вызова диалогового окна **Delete** (идентификатор окна `IDD_DELETE`) по команде меню с идентификатором `IDM_DELETE`. В качестве последнего параметра при вызове функции `DialogBox()` указано имя `(DLGPROC)Delete` ее оконной процедуры.

Приведем еще один пример оконной процедуры для диалогового окна **Add new item**, вид которого показан на рис. 23.5. На нем расположен ряд элементов управления, в том числе один простой редактор Edit Control и три поля со списком Combo Box.

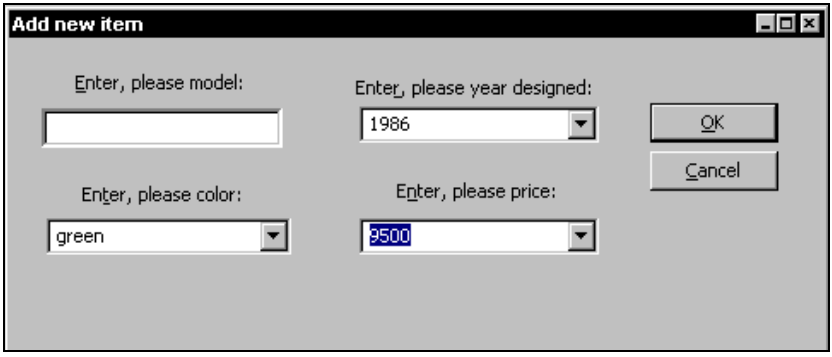


Рис. 23.5. Вид диалогового окна **Add new item**

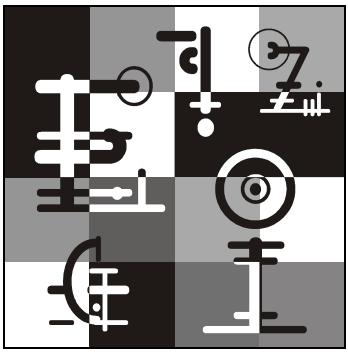
Возможный вариант оконной процедуры обработки сообщений для этого диалогового окна приведен в листинге 23.7.

Листинг 23.7. Пример оконной процедуры `Add`

```
LRESULT CALLBACK Add(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        int k = 20; int year = 1985;
        int price = 9000;
        char* str = new char[10]; *str = 0;
        switch (message)
        {
        case WM_INITDIALOG:
         for ( int i = 0; i < k; i++ )
           {
             SendMessage(GetDlgItem(hDlg,IDC_COMBO1),CB_ADDSTRING,↳
               true,(LPARAM)(_itoa(i+year,str,10)));
             SendMessage(GetDlgItem(hDlg,IDC_COMBO2),CB_ADDSTRING,↳
```

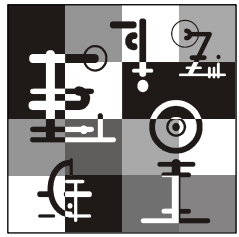
```
            true,(LPARAM)(_itoa(i*500+price,str,10)));
        }
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("red"));
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("blue"));
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("yellow"));
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("green"));
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("white"));
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("black"));
      SendMessage(GetDlgItem(hDlg,IDC_COMBO3),CB_ADDSTRING,↳
         true,(LPARAM)("gray"));
      return TRUE;

    case WM_COMMAND:
           if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
           {
                  char* str1 = new char[20]; *str1 = 0;
        char* str2 = new char[20]; *str2 = 0;
        char* str3 = new char[20]; *str3 = 0;
        char* str4 = new char[20]; *str4 = 0;
                  GetDlgItemText(hDlg, IDC_EDIT1, str1, 20);
                  GetDlgItemText(hDlg, IDC_COMBO3, str2, 20);
                  GetDlgItemText(hDlg, IDC_COMBO1, str3, 20);
                  GetDlgItemText(hDlg, IDC_COMBO2, str4, 20);
                  Control.Add( str1, str2, atoi(str3), atoi(str4));
                  EndDialog(hDlg, LOWORD(wParam));
                  return TRUE;
           }
           break;
    }
    return FALSE;
}
```

Как видим, с помощью названных элементов управления получаются значения четырех символьных строк `str1`, `str2`, `str3` и `str4`. Они используются в обработчике сообщения `WM_COMMAND` и при вызове функции `Control.Add`.



Часть VI

Приложения MFC



Глава 24

Характеристика приложений MFC

Библиотека MFC

При создании приложений на основе объектно-ориентированного подхода используются компоненты библиотеки MFC (Microsoft Foundation Classes). Соответствующие приложения называют приложениями MFC. Как отмечалось, при создании приложений MFC можно использовать всевозможные элементы управления графического интерфейса пользователя Windows: кнопки, переключатели, меню и др. Дадим краткую характеристику названной библиотеке.

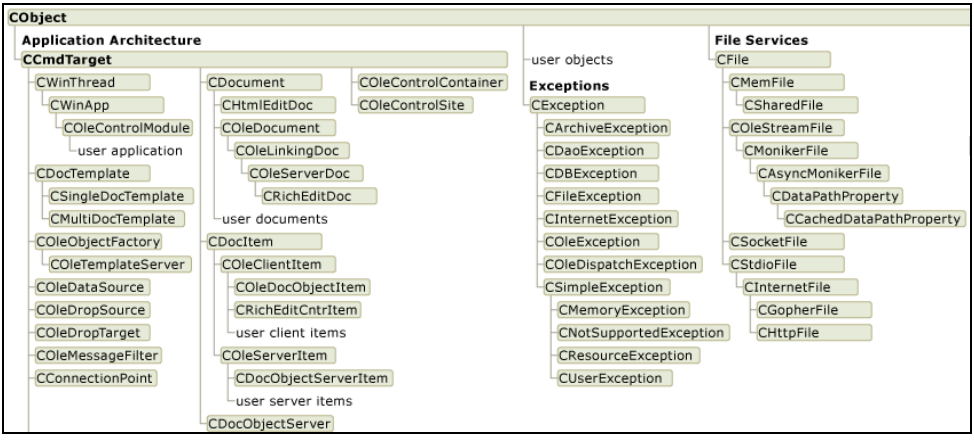


Рис. 24.1. Фрагмент иерархии классов библиотеки MFC

Библиотека MFC содержит набор классов, созданных для упрощения программирования приложений. При разработке библиотеки ставилась задача обеспечить минимальный объем дополнительного кода и скорость выполнения

приложений, использующих MFC, сопоставимую со скоростью выполнения приложений, использующих API Win32. Кроме того, подразумевалась возможность удобного вызова любых функций C API Win32. Фрагмент иерархии классов библиотеки MFC приведен на рис. 24.1.

Основным классом является класс CObject, от него наследуются все остальные классы. Все они разбиты на группы, в частности в группе классов обслуживания файлов File Services основным является класс CFile.

Этапы создания приложения MFC

Создание приложения MFC с помощью MVC++ не очень сильно отличается от создания консольного приложения (*см. главу 3*). Для создания нового приложения MFC достаточно выполнить следующие шаги:

1. Задать команду **File/New/Project** основного меню MVC++.
2. В открывшемся диалоговом окне **New Project** указать имя рабочей области и имя проекта (на рис. 24.2 это **MFCSolution1** и **MFCProject1**), выбрать тип проекта (**MFC**) и шаблон приложения (**MFC Application**).

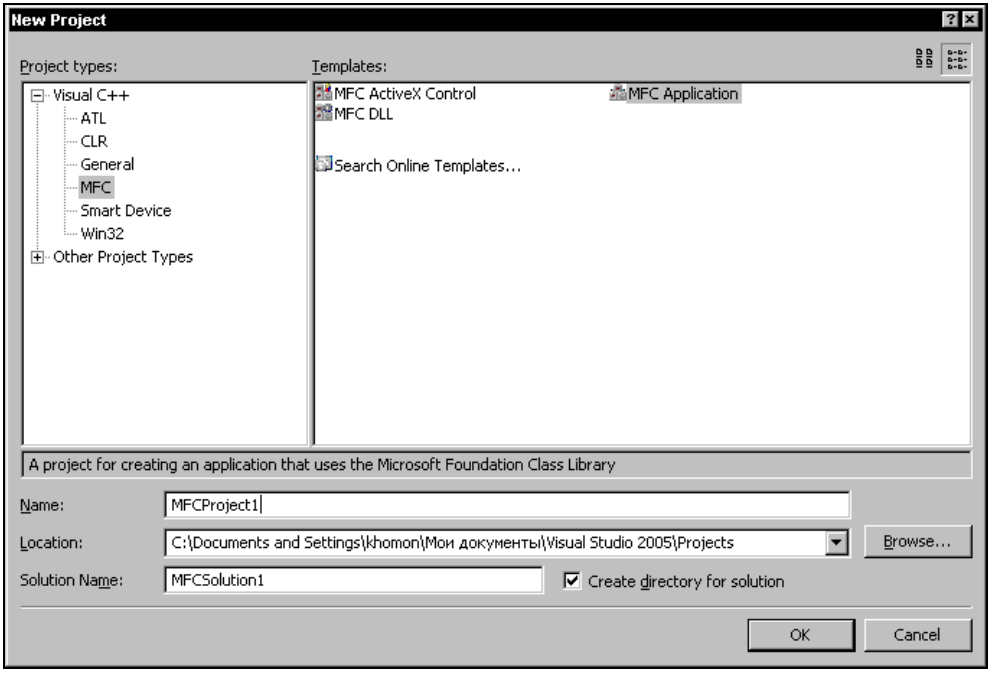


Рис. 24.2. Диалог задания типа приложения MFC

3. На последующих шагах взаимодействия Мастером приложений MFC Application Wizard следует при необходимости внести уточняющие настройки, например, выбрать тип приложения MFC с однодокументным интерфейсом. Нажать **ОК**.
4. Дополнить созданный проект приложения нужными классами, диалоговыми окнами, меню, настроить функциональность приложения.

Типы и состав приложений MFC

Приложения MFC могут создаваться с однодокументным (SDI, Single Document Interface) и многодокументным (MDI, Multiple Document Interface) интерфейсом. Они могут обеспечивать или не обеспечивать работу с базами данных и работу с составными документами. Кроме того, приложения могут использовать разделяемые библиотеки DLL или статические библиотеки. Мы рассмотрим состав простейшего приложения MFC с однодокументным интерфейсом.

Приложение SDI имеет меню, с помощью которого пользователь может открыть один документ и с ним работать. Рассмотрим код, сгенерированный для таких приложений Мастером приложений MFC при настройках по умолчанию: нет поддержки операций с базами данных и составными документами, есть панель инструментов, строка состояния, и справка; библиотека MFC используется в качестве разделяемой библиотеки DLL.

Мастер приложений MFC создает пять классов. Для нашего приложения MFCSDI эти классы имеют следующие имена:

- CAboutDlg — класс диалогового окна **About**;
- CMFCSDIApp — класс для приложения, производный от класса CWinApp;
- CMFCSDIDoc — класс документа, производный от класса CDocument;
- CMFCSDIView — класс представления, производный от класса CView;
- CMainFrame — класс фрейма.

Файл заголовка для класса CMFCSDIApp приведен в листинге 24.1. Для просмотра содержимого этого файла достаточно выполнить двойной щелчок на имени этого класса в окне **Class View**. После этого текст файла заголовка появится в окне редактора кода.

Листинг 24.1. Код MFCSDI.h — главного файла заголовка приложения MFCSDI

```
// MFCSDI.h : главный файл заголовка для приложения MFCSDI
//
#pragma once

#ifndef __AFXWIN_H__
	#error подключите stdafx.h' перед подключением этого файла для PCH
#endif

#include "resource.h"		// main symbols


// CMFCSDIApp:
// Реализацию этого класса см. в файле MFCSDI.cpp
//

class CMFCSDIApp : public CWinApp
{
public:
	CMFCSDIApp();


// Перегрузка
public:
	virtual BOOL InitInstance();

// Реализация
	afx_msg void OnAppAbout();
	DECLARE_MESSAGE_MAP()
};

extern CMFCSDIApp theApp;
```

Здесь директива препроцессора `#pragma` служит для защиты заголовка. Она гарантирует, что файл заголовка не будет откомпилирован в процессе сборки проекта дважды. Двойное включение файла заголовка достаточно характерно для программ на языке C++.

В файле заголовка важную роль играет объявление класса CMFCSDIApp, наследуемого от класса CWinApp. Последний класс включает в себя большинство функциональных возможностей приложения. Мастер приложений MFC сгенерировал несколько функций для класса-наследника, которые перегружают соответствующие функции базового класса. Фрагмент текста, который начинается с комментария

```
// Перегрузка,
```

представляет собой перегрузку виртуальной функции. Следующий фрагмент программного кода с комментария

```
// Реализация
```

представляет часть карты сообщений, которая содержит объявление функции-обработчика `OnAppAbout ()` и предложение

```
DECLARE_MESSAGE_MAP(),
```

представляющее макроопределение препроцессора. С помощью этого макроса определяются переменные и функции, используемые для обработки сообщений. Вторая часть карты сообщения размещается в файле с исходным кодом программы.

Мастер приложений MFC генерирует текст методов класса CMFCSDIApp: конструктора, `InitInstance ()` и `OnAppAbout ()` — и размещает его в файле MFCSDI.cpp. Приведем текст конструктора, который инициализирует объект класса:

```
CMFCSDIApp::CMFCSDIApp()
// TODO: добавьте текст конструктора и
// выполните инициализацию
// объекта в методе InitInstance.
```

Этот конструктор ничего не возвращает и, по сути, не инициализирует. Чтобы сообщить программе о возникших при инициализации объекта проблемах, применяется так называемая двухэтапная инициализация. Создается функция инициализации со стандартным именем `InitInstance ()`, код которой приводится в листинге 24.2.

Листинг 24.2. Код функции инициализации

```
// CMFCSDIApp инициализация

BOOL CMFCSDIApp::InitInstance()
{
        // Вызов требуется для приложений Windows XP,
```

```
    // использующих стили библиотеки ComCtl32.dll
    // версии 6 или ниже, иначе создать окно нельзя.
    InitCommonControls();

    CWinApp::InitInstance();

    // Инициализация OLE-библиотек
    if (!AfxOleInit())
    {
           AfxMessageBox(IDP_OLE_INIT_FAILED);
           return FALSE;
    }
    AfxEnableControlContainer();
    // Стандартная инициализация
    // удалите те из программ инициализации,
    // в которых нет необходимости
    // Измените ключ реестра, в котором хранятся установки
    // TODO: измените строку в соответствии с названием
    // фирмы и прочее
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));
    LoadStdProfileSettings(4);  // Загрузка стандартных параметров
                                // файла INI
    // Регистрация шаблонов документа для связи
    // между документами, фреймами и представлениями
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
           IDR_MAINFRAME,
           RUNTIME_CLASS(CMFCSDIDoc),
           RUNTIME_CLASS(CMainFrame),       // основной фрейм
                                          // приложения SDI
           RUNTIME_CLASS(CMFCSDIView));
    AddDocTemplate(pDocTemplate);
    // Разбор командной строки
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);
    // Обработка команд из командной строки,
   // возврат значения FALSE, если приложение было запущено
    // с ключами /RegServer, /Register, /Unregserver или /Unregister.
    if (!ProcessShellCommand(cmdInfo))
```

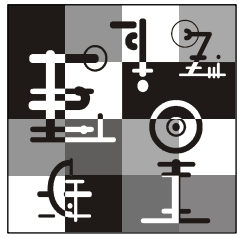
```
            return FALSE;
        // Инициализировано одно окно,
        // оно отображается и обновляется
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();
        // Вызов DragAcceptFiles, если  передан аргумент
        // Строка имеется в приложении SDI после
        // вызова функции ProcessShellCommand
        return TRUE;
}
```

Функция `InitInstance ()` готовит приложение к работе. Проводится инициализация библиотек OLE и разрешается приложению содержать элементы управления ActiveX, для чего вызывается функция `AfxEnableControl Container()`. Затем происходит регистрация приложения в системном реестре.

Далее `InitInstance ()` выполняется регистрация шаблонов документа, который будет создан SDI-приложением.

Для анализа командной строки `InitInstance ()` организует пустой объект класса `CCommandLineInfo`. Функция `ParseCommandLine ()` помещает в этот объект параметры, заданные в командной строке при запуске приложения. Функция `ProcessShellCommand ()` вызывается для выполнения операций, заданных этими параметрами. Это означает, что приложение сможет поддерживать параметры командной строки.

Последний оператор в `InitInstance ()` возвращает значение `TRUE`, извещая о том, что инициализация завершена.



Глава 25

Обработка сообщений

Так же, как и приложения API Windows, выполнение приложения MFC основано на использовании цикла обработки сообщений. Однако в тексте программ, генерируемых с помощью Мастера приложений MFC, соответствующий код цикла скрыт от разработчика. Но в организации обработки сообщений разработчик приложения участвует непосредственно при определении функциональности приложения.

Карты сообщений

Одно из важнейших отличий в составе и организации приложений API Windows и MFC связано с тем, как организуется обработка сообщений. В приложении API Windows для каждого окна разработчик создает оконную процедуру обработки сообщений, в которой с помощью оператора CASE задает нужную обработку каждого вида сообщений. В приложении MFC для этих целей используются так называемые карты сообщений.

Для каждого класса, который может получать сообщения, должна быть определена своя карта сообщений. Причем определение карты сообщений должно размещаться вне функции или объявления класса. Карта сообщений состоит из двух частей.

Первая часть карты сообщений помещается в заголовочном файле с расширением `.h`, в котором содержится определение класса. Например, в файле `MFCSDI.h` содержится первая часть карты сообщений с кодом:

```
afx_msg void OnAppAbout();
DECLARE_MESSAGE_MAP()
```

Здесь макрос `DECLARE_MESSAGE_MAP()` объявляет, что в классе будет использоваться карта сообщений, перед ним указан прототип функции `OnAppAbout ()`,

которая обрабатывает соответствующие сообщения. Перед приведенным макросом могут быть заданы прототипы нескольких функций, которые участвуют в обработке сообщений.

Вторая часть карты сообщений размещается в файле с исходным кодом и расширением `.cpp` (листинг 25.1), она указывает, какие именно сообщения обрабатываются и с помощью каких методов выполняется их обработка.

Листинг 25.1. Пример 2-й части карты сообщений из файла `MFCSDI.cpp`

```
BEGIN_MESSAGE_MAP(CMFCSDIApp, CWinApp)
       // Стандартная команда справочной помощи
       ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
       // Стандартные команды для файлов документов
       ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
       ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
       // Стандартная команда настройки принтера
       ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Эта карта сообщений обеспечивает обработку команд меню главного окна приложения. Так при выполнении команды **Help/About** (в ресурсах эта команда имеет идентификатор `ID_APP_ABOUT`), вызывается метод `CMFCSDIApp :: OnAppAbout ()`. Если выполнить команду **File/New**, **File/Open** или **File/Print Setup**, то будут выполнены соответствующие методы класса `CWinApp`. Эти функции можно перегрузить функциями собственной разработки, если требуется, чтобы они выполняли что-то нестандартное при вызове указанных команд меню. Текст функции `OnAppAbout ()` приведен в листинге 25.2.

Листинг 25.2. Пример текста функции OnAppAbout ()

```
void CMFCSDIApp: :OnAppAbout ()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal();
}
```

Здесь объявляется объект, который является экземпляром класса `CAboutDlg`, а также вызывается функция `DoModal ()`, которая выводит на экран диалоговое окно. Изменять что-либо в стандартной процедуре обработки щелчков на кнопках **ОК** и **Cancel** не требуется.

Внутренний механизм работы карты сообщений в приложениях MFC можно описать следующим образом. Каждое приложение имеет объект, наследуемый от класса `CWinApp`, и метод `Run ()`, который вызывает одноименную функцию `Run ()` класса `CWinThread`, обеспечивающего работу с потоками. В составе этой функции имеется привычный для нас цикл обработки сообщений с помощью вызова функций `GetMessage ()`, `TranslateMessage ()` и `DispatchMessage ()`.

Каждый объект-окно использует класс окна и оконную процедуру обработки сообщений с именем `AfxWndProc ()`. Напомним, что в приложениях API Windows сообщения назначаются оконным процедурам обработки сообщений с помощью дескрипторов окон `hWnd`. В библиотеке MFC-приложения имеется так называемая карта дескрипторов, состоящая из таблицы дескрипторов окон и указателей объектов. Таким образом, главная программа приложения с помощью названной информации находит указатель на объект-окно и вызывает соответствующую ей виртуальную процедуру этого объекта. Нужная реализация названной процедуры для разных элементов управления или окон обеспечивается на основе полиморфизма.

Макросы карт сообщений

В картах сообщений используется достаточно большое количество макросов, с помощью которых приложение может перемещаться по карте сообщений и распознавать объекты и методы обработки сообщений. Для наглядности мы приведем описание ряда важнейших, на наш взгляд, макросов карт сообщений.

- `DECLARE_MESSAGE_MAP()` — объявляет, что карта сообщений будет использоваться в классе для передачи сообщений функциям обработки;
- `BEGIN_MESSAGE_MAP` — определяет начало карты обработки сообщений;
- `END_MESSAGE_MAP()` — определяет конец карты обработки сообщений;
- `ON_COMMAND` — указывает на метод для обработки сообщения от команды меню;
- `ON_CONTROL` — указывает на метод для обработки сообщения от элемента управления;
- `ON_MESSAGE` — указывает на метод для обработки сообщения, созданного пользователем.

Подчеркнем, что основное назначение макросов карт сообщений заключается в определении методов, используемых для обработки соответствующих видов сообщений.

Типы передаваемых сообщений

Для распознавания типов передаваемых сообщений им назначаются предопределенные префиксы. Элементы управления и диалоговые окна имеют свои префиксы сообщений, источниками которых они являются или которые могут принимать. К примеру, для элемента управления стандартной кнопки **OK** диалогового окна **About** может иметь место следующий список уведомляющих сообщений:

- `BN_CLICKED` — нажатие кнопки;
- `BN_DOUBLECLICKED` — двойное нажатие кнопки;
- `BN_KILLFOCUS` — потеря фокуса;
- `BN_SETFOCUS` — приобретение фокуса.

При необходимости можно просмотреть имена сообщений, используемые всеми элементами управления некоторого диалогового окна. Для этого при открытом ресурсе нужного диалогового окна в окне **Properties** (Свойства) достаточно щелкнуть пиктограмму  **Control Events** (События элементов управления). При этом в окне **Properties** (рис. 25.1) будет отображен список всех оконных событий для каждого из элементов управления нашего ресурса диалогового окна.

Некоторые возможные варианты префиксов в сообщениях от элементов управления приведены в табл. 25.1.

Таблица 25.1. *Варианты префиксов в сообщениях от элементов управления*

Префиксы	Элемент управления или окно
BN	**Button** (Кнопка)
BCN, BN	**Check Box** (Флажок)
CBN	**Combo Box** (Поле со списком)
EN	**Edit Control** (Простой редактор)
LBN	**List Box** (Список)
–	**Static Text** (Надпись)
–	**Group Box** (Группа)
BCN, BN	**Radio Button** (Переключатель)

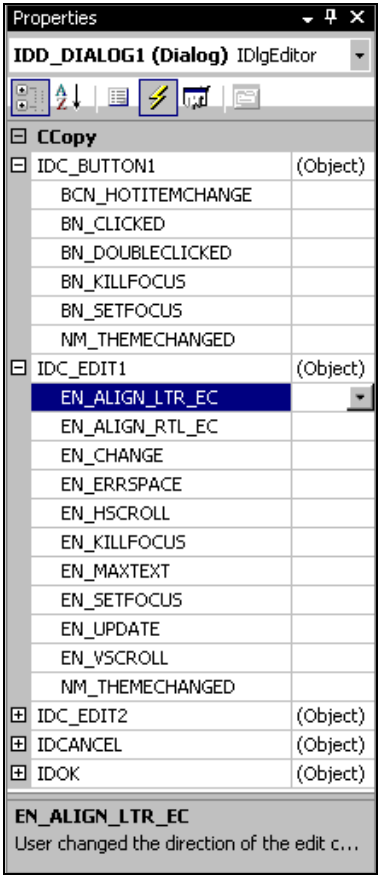
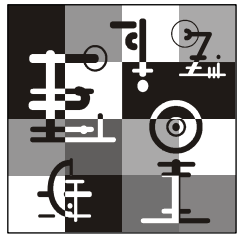


Рис. 25.1. Список всех оконных событий для элементов управления

Для каждого из указанных в табл. 25.1 элементов управления, имеющих префиксы, существует также сообщение `NM_THEMECHANGED`. Это уведомительное сообщение, которое передается элементу управления от родительского окна.



Глава 26

Разработка интерфейса приложения

Общая характеристика интерфейса приложения

Интерфейс приложения MFC образуют диалоговые окна и меню, которые, как и в случае приложений Windows API, представляют собой ресурсы. Они служат приложению для обеспечения отображения на экране самого диалогового окна и меню, а также элементов управления, размещаемых на поверхности диалоговых окон.

Создание диалогового окна

После создания заготовки приложения MFC с помощью Мастера в нем имеется главное диалоговое окно с меню. В общем случае приложение может иметь множество диалоговых окон. Для создания нового диалогового окна в приложении MFC, как и в приложении API Windows, достаточно выполнить следующие шаги:

1. В окне браузера системы программирования выбрать нужный проект создаваемого приложения.
2. Выполнить команду меню MVC++ **Project/Add Resource** (Проект/Добавить ресурс).
3. В открывшемся диалоговом окне **Add Resource** (см. рис. 23.2) выбрать вариант **Dialog** и нажать кнопку **New**.

Напомним, что окно Windows принадлежит одному из существующих в системе классов, который должен быть определен до отображения окна на экране. Обычно класс конкретного диалогового окна наследуется от базового класса `CDialog` библиотеки MFC. После создания ресурса диалогового окна

можно выполнить создание класса окна и связать его с нашим диалоговым окном. Это позволяет нам создать карту сообщений для диалогового окна и обеспечить нужную нам обработку сообщений, генерируемых с помощью элементов управления, расположенных на этом окне.

Создание класса окна

Для создания класса окна и связи его с нашим диалоговым окном нужно выполнить следующие шаги:

1. При открытом окне ресурса созданного диалогового окна выполнить команду меню MVC++ **Project/Add Class** (Проект/Добавить класс) или одноименную команду контекстного меню окна редактирования ресурса.
2. В открывшемся окне Мастера классов MFC Class Wizard в поле **Class name** (Имя класса) указать имя класса диалогового окна (рис. 26.1). Нажать кнопку **Finish**.

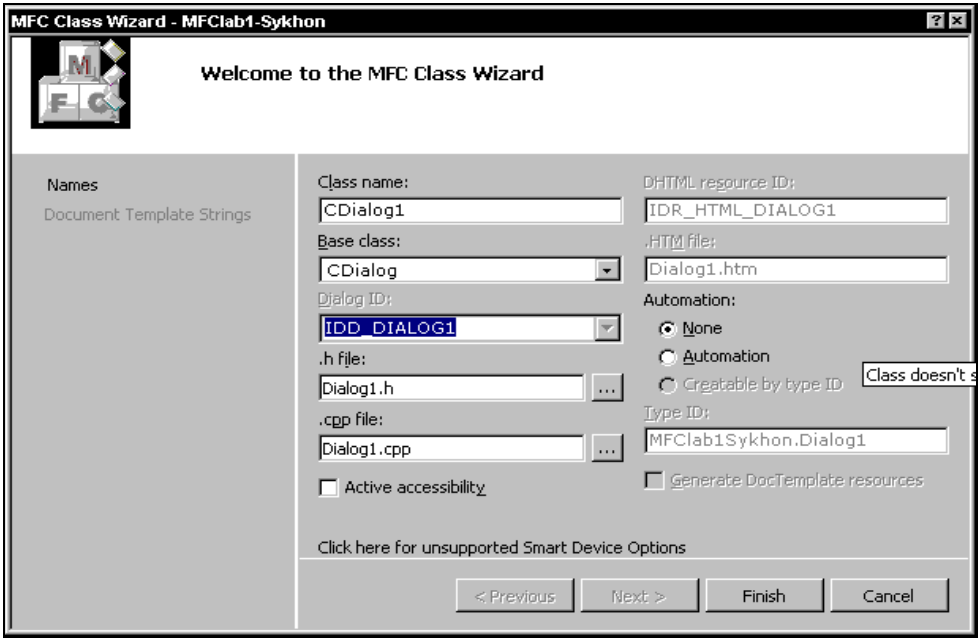


Рис. 26.1. Диалоговое окно Мастера классов MFC

При этом в полях **.h file** и **.cpp file** будут автоматически сформированы имена файлов заголовка **CDialog1.h** и исходного файла **CDialog1.cpp**, а сами эти файлы будут автоматически добавлены в проект. Кроме того, как видим

на рис. 26.1, в поле **Dialog ID** автоматически помещен идентификатор ресурса созданного нами диалогового окна `IDD_DIALOG1`. Это означает установление связи созданного класса с ресурсом нашего диалогового окна.

Доступ к элементам управления окна

После создания класса окна и установления связи его с ресурсом окна можно обеспечить доступ к элементам управления диалогового окна путем добавления в его класс атрибутов. Для этого достаточно выполнить такую последовательность действий.

1. При открытом проекте приложения Win32 задать команду **View/Resource View** основного меню MVC++.
2. В открывшемся окне **Resource View** двойным щелчком мыши раскрыть папку **<Имя проекта>.rc** и выбрать папку **Dialog**.
3. Выбрать требуемое диалоговое окно и щелчком мыши выделить нужный элемент управления, для которого требуется обеспечить доступ и задать реакцию.
4. Щелчком правой кнопкой мыши вызвать контекстное меню и выбрать команду **Add Variable**.
5. В открывшемся диалоговом окне **Add Member Variable Wizard** (рис. 26.2) выбрать тип доступа, идентификатор элемента управления и категорию.
6. Установить флажок в поле **Control variable** (Переменная элемента управления), чтобы связать атрибут с элементом управления диалогового окна.
7. Ввести имя переменной, например `m_check`, в поле **Variable Name** и нажать **Finish**.

В результате произойдет создание переменной с заданным именем. Кроме того, в функцию `DoDataExchange()` (о ней речь пойдет далее) для класса диалогового окна будет добавлена соответствующая строка. Для приведенного на рис. 26.2 примера будет добавлена следующая строка:

```
DDX_Control(pDX, IDC_CHECK1, m_check);
```

Чтобы задать обработчик события для некоторого элемента управления диалогового окна достаточно выполнить такую последовательность действий.

1. При открытом проекте приложения Win32 задать команду **View/Resource View** основного меню MVC++.
2. В открывшемся окне **Resource View** двойным щелчком мыши раскрыть папку **<Имя проекта>.rc** и выбрать папку **Dialog**.

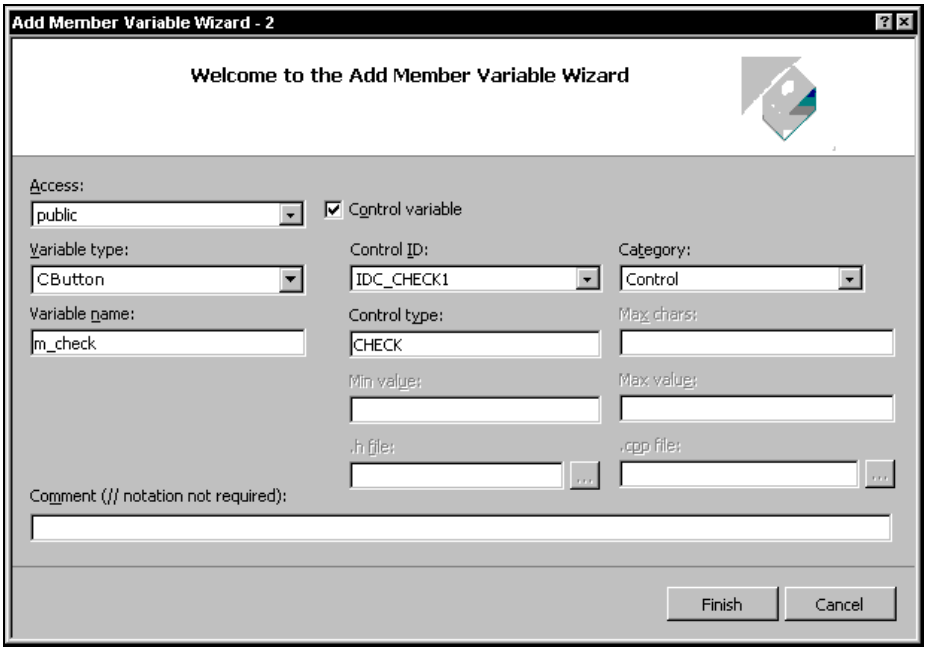


Рис. 26.2. Диалоговое окно **Add Member Variable Wizard**

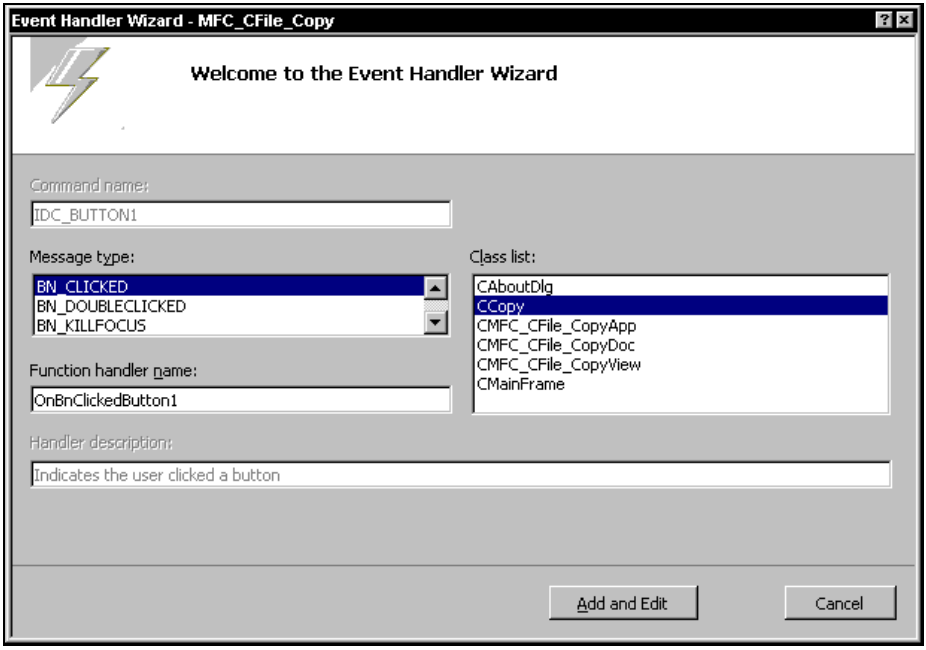


Рис. 26.3. Диалоговое окно **Event Handler Wizard**

3. Выбрать требуемое диалоговое окно и щелчком мыши выделить нужный элемент управления, для которого требуется обеспечить доступ и задать реакцию.
4. Щелчком правой кнопкой мыши вызвать контекстное меню и выбрать команду **Add Event Handler**.
5. В открывшемся диалоговом окне **Event Handler Wizard** (рис. 26.3) выбрать класс, событие для задания реакции и обработчик соответствующего события.

В результате будет автоматически сформирована заготовка процедуры обработчика события доступа к выбранному нами элементу управления (щелчок по кнопке) вида:

```
void CCopy::OnBnClickedButton1()
{
        // Место для кода обработчика
}
```

Сюда остается поместить нужный нам код обработки события щелчка по кнопке.

Заметим, что в отличие от обработчиков событий остальных кнопок, обработчик события для кнопки **OK** любого диалогового окна обязательно содержит автоматически генерируемый Мастером MFC вызов функции `OnOk()`:

```
void CAboutDlg::OnBnClickedOk()
{
        // Место для кода обработчика
        OnOK();
}
```

Важно, что функция `OnOk()` содержит вызов функции `DoDataExchange()`, сгенерированной Мастером MFC Class Wizard. Она служит для внесения выполненных в элементы управления диалогового окна изменений перед его закрытием путем щелчка по кнопке. Функция `DoDataExchange()` может содержать, к примеру, следующий код:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
    // Вызов базовой версии класса
    CDialog::DoDataExchange(pDX);
    DDX_Check(pDX, IDC_MY_CHECKBOX, m_bVar);
    DDX_Text(pDX, IDC_MY_TEXTBOX, m_strName);
    DDX_Control(pDX, IDC_LIST1, m_listbox);
    DDV_MaxChars(pDX, m_strName, 20);
}
```

В функции `DoDataExchange()` параметр `pDX` определяет указатель на объект класса `CDataExchange`. Функции, имена которых начинаются с префикса `DDX`, осуществляют обмен данными. У них при вызове второй аргумент указывает идентификатор элемента управления, а третий аргумент — идентификатор атрибута класса (его мы задавали с помощью Мастера Add Member Variables Wizard). Например, в строке кода

```
DDX_Control(pDX, IDC_LIST1, m_listbox);
```

выполняется обмен данными элемента управления **List Box** с идентификатором `IDC_LIST1` и идентификатором атрибута класса `m_listbox`. Как видим, таким образом устанавливается соответствие между элементами управления и членами класса диалогового окна.

Как следует из приведенных примеров, имена функций с префиксами зависят также от типа данных, которыми могут обмениваться диалоговое окно и объект соответствующего класса. А именно каждая такая функция включает в свой состав имя элемента управления. К примеру, функция `DDX_Text()` служит для обеспечения связи между простым редактором Edit Control и атрибутом типа `CString`.

Функции, имена которых начинаются с префикса `DDV`, служат для подтверждения внесенных изменений при работе с диалоговыми окнами.

Разработчик приложения при необходимости самостоятельно должен вносить изменения в состав функции `DoDataExchange()`, дополняя ее необходимыми вызовами функции с префиксами `DDX` и `DDV`.

Для примера приведем коды (листинг 26.1) обработчиков событий и функций для класса `CFindDlg` диалогового окна с заголовком **Find by key**, вид которого приведен на рис. 26.4.

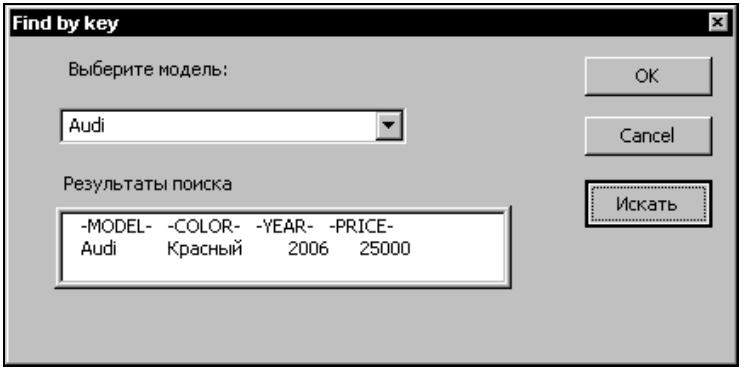


Рис. 26.4. Диалоговое окно **Find by key**

Листинг 26.1. Коды обработчиков событий и функций для класса `CFindDlg`

```
// обработчик для кнопки OK
void CFindDlg::OnBnClickedOk()
{
        theApp.m_pMainWnd->Invalidate(TRUE);
        OnOK();
}

// обработчик для кнопки IDC_BUTTON1
void CFindDlg::OnBnClickedButton(){
    char* str1 = new char[20]; *str1 = 0;
    char* str2 = new char[20]; *str2 = 0;
    char* str3 = new char[20]; *str3 = 0;
    char* str4 = new char[20]; *str4 = 0;
    char* String1 = new char[100]; *String1 = 0;
    char* String2 = new char[100]; *String2 = 0;
        POSITION pos = theApp.m_pDocManager->
↳GetFirstDocTemplatePosition();
        CSingleDocTemplate* Doc = (CSingleDocTemplate*)
                ↳theApp.m_pDocManager->GetNextDocTemplate(pos);
        pos = Doc->GetFirstDocPosition();
        CMy2Doc* pDoc = (CMy2Doc*)Doc->GetNextDoc(pos);
        for ( int i = 0; i < pDoc->Control.Massiv->GetQ(); i++ )
                if (i == combo.GetCurSel())
                        pDoc->Control.Find(pDoc->Control.Massiv->Key[i]);
        str1 = pDoc->Control.Auto->Model;
        str2 = pDoc->Control.Auto->Color;
        _itoa(pDoc->Control.Auto->YearDesigned,str3,10);
        _itoa(pDoc->Control.Auto->Price,str4,10);
        strcat(String2,"   ");strcat(String2,"-MODEL-");
        strcat(String2,"   ");strcat(String2,"-COLOR-");
        strcat(String2,"   ");strcat(String2,"-YEAR-");
        strcat(String2,"   ");strcat(String2,"-PRICE-");
        strcat(String1,"   ");strcat(String1,str1);
        strcat(String1,"          ");strcat(String1,str2);
        strcat(String1,"         ");strcat(String1,str3);
        strcat(String1,"      ");strcat(String1,str4);
        list.ResetContent () ;
```

```
    list.AddString((LPCTSTR)String2);
    list.AddString((LPCTSTR)String1);

}

// код функции Initial()
void CFindDlg::Initial (){
POSITION pos = theApp.m_pDocManager->GetFirstDocTemplatePosition();
       CSingleDocTemplate* Doc = (CSingleDocTemplate*)
              ↳theApp.m_pDocManager->GetNextDocTemplate(pos);
       pos = Doc->GetFirstDocPosition();
       CMy2Doc* pDoc = (CMy2Doc*)Doc->GetNextDoc(pos);
       int k = pDoc->Control.Massiv->GetQ();
       combo.ResetContent () ;
       for ( int i = 0; i < k; i++ )
              combo.AddString ((LPCTSTR)pDoc->Control.Massiv->Key[i]);
}

// код функции DoDataExchange()
void CFindDlg::DoDataExchange(CDataExchange* pDX)
{
       CDialog::DoDataExchange(pDX);
       DDX_Control(pDX, IDC_COMBO1, combo);
       DDX_Control(pDX, IDC_LIST1, list);
}
```

Здесь в обработчике `OnBnClickedButton` для кнопки с идентификатором `IDC_BUTTON1` и заголовком **Найти** выполняется поиск строки, номер которой определяется с помощью проверки условия

```
if (i == combo.GetCurSel())
```

где с помощью функции `GetCurSel()` определяется номер строки, выбранной в поле со списком Combo Box. По найденному номеру выполняется поиск самой строки с помощью вызова функции `Control.Find()`. Затем с помощью соответствующих методов класса `Control.Auto` осуществляется считывание параметров автомобиля (модель, цвет, год и стоимость) и запись их в строковые переменные. Далее сбрасывается содержимое элемента управления List Box (имя переменной `list`) и с помощью строк:

```
list.AddString((LPCTSTR)String2);
list.AddString((LPCTSTR)String1);
```

выполняется добавление символьных строк в элемент управления List Box.

В функции `Initial()`сбрасывается содержимое и выполняется заполнение элемента управления поле со списком Combo Box (имя переменной `combo`) с помощью следующего цикла:

```
for ( int i = 0; i < k; i++ )
                .AddString ((LPCTSTR)pDoc->Control.Massiv->Key[i]);
```

В функции `DoDataExchange()` выполняется обмен данными для двух элементов управления: списка List Box и поля со списком Combo Box. Напомним, что это код формируется с помощью Мастера Add Member Variable Wizard.

Вывод текста в диалоговое окно

Как отмечалось, Windows обеспечивает широкий набор графических средств для использования с помощью контекстов устройств. Можно рисовать линии, закрашивать области кистью, использовать шрифты для вывода текста. MFC содержит классы графических объектов, эквивалентных средствам рисования в Windows. В табл. 26.1 дается соответствие между доступными классами графических объектов MFC и эквивалентными им типами дескрипторов графического интерфейса Windows.

Таблица 26.1. *Соответствие классов графических объектов*

Классы MFC	**Типы дескрипторов Windows**
CPen	HPEN
CBrush	HBRUSH
CFont	HFONT
CBitmap	HBITMAP
CPalette	HPALETTE
CRgn	HRGN

Примечание

Класс CImage служит для обеспечения расширенной поддержки битовых массивов.

Мы не будем касаться всех возможностей графического интерфейса приложений MFC. Остановимся на возможностях вывода текста в диалоговое окно.

Прежде всего, отметим, что вывод текста в диалоговое окно связан с перерисовкой изображения в этом окне. Перерисовка изображения связывается с сообщением `WM_PAINT`, посылаемым операционной системой объекту окна

приложения при выполнении любой из операций, требующих перерисовки. К числу таких операций относятся, например, запуск приложения, изменение размеров окна или открытие ранее невидимой его части.

В приложениях MFC для сообщения `WM_PAINT` в карте обработки сообщений соответствует макрос `ON_WM_PAINT()`. Для обработки этого сообщения используется функция с именем `OnPaint()`, которая формируется автоматически. В теле этой функции содержится вызов функции `OnDraw()`, которая выполняет основные управляющие действия по обновлению представления интерфейса приложения на экране. Обычно для приложения нужно самостоятельно разрабатывать эту функцию на основе заготовки. К примеру, для приложения с именем проекта `CMFCTextOut` исходный текст функции `CMFCTextOutView::OnDraw()` содержит код:

```
void CMFCTextOutView::OnDraw(CDC* pDC)
{
        CMFCTextOutDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if (!pDoc)
                return;
        // место размещения вызовов для перерисовки
}
```

В теле этой функции могут размещаться разветвления или переключатели для вызова нужных функций перерисовки текста или графики в зависимости от условий.

Для использования функций перерисовки, например, функции вывода текста с именем `PrintText()`, ее нужно включить в класс `CMFCTextOutView`, выполнив следующие шаги:

1. Вызвать контекстное меню класса `CMFCTextOut` в диалоговом окне **Class View** и задать в нем команду **Add/Add Function**.
2. В открывшемся диалоговом окне (см. рис. 26.5) в списке поля **Return type** (Тип возвращаемого значения) выбрать вариант **void**.
3. В поле **Function name** ввести имя добавляемой функции `PrintText`.
4. В поле комбинированного списка **Parameter type** ввести значение **CDC***.
5. В поле комбинированного списка **Parameter name** ввести значение **pDC**.
6. Нажав кнопку **Add** (Добавить), поместить сформированную таким образом пару значений в поле **Parameter list** (см. рис. 26.5).
7. В поле комбинированного списка выбрать вариант **protected**.
8. Нажать кнопку **Finish**.

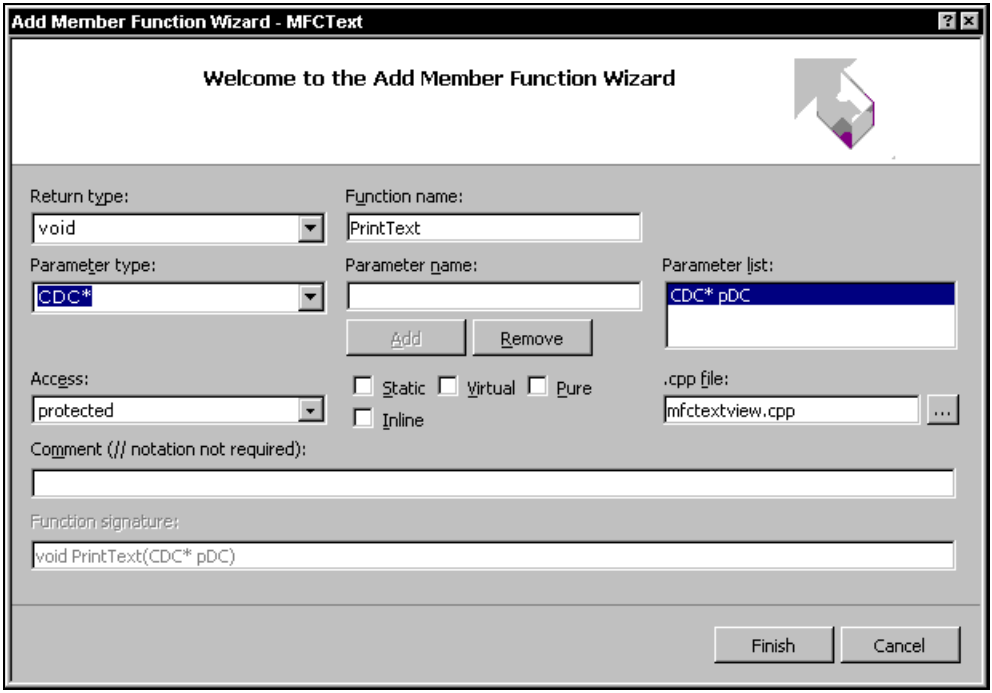


Рис. 26.5. Диалоговое окно **Add Member Function Wizard**

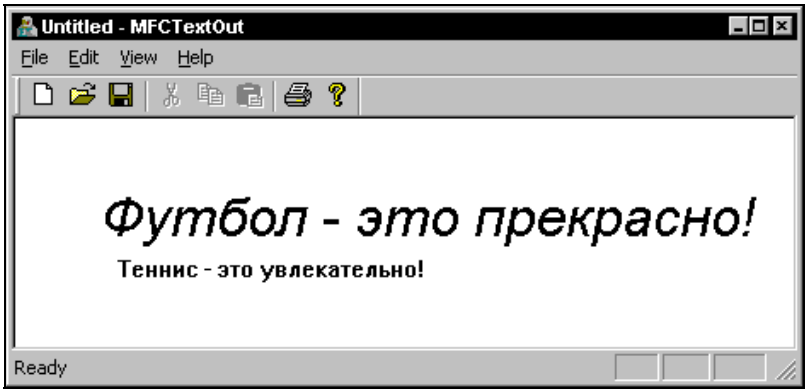


Рис. 26.6. Вывод текста в диалоговом окне

В результате будет сформирована заготовка функции `PrintText()`, в которую нам нужно наполнить кодом, обеспечивающим вывод текста. Вызов этой функции нужно разместить в теле функции `OnDraw()` с помощью строки вида:

```
PrintText(pDC);
```

Для примера приведем код функции `PrintText()` (листинг 26.2), которая обеспечивает вывод текста в диалоговом окне приложения MFC, как показано на рис. 26.6.

Листинг 26.2. Пример функции для вывода текста в диалоговое окно

```
void CMFCTextOutView::PrintText(CDC* pDC)
{
   // структура для задания параметров шрифта
   LOGFONT logFont;
   // обнуление всех параметров структуры
   memset(&logFont, 0, sizeof(logFont));
   // задание значений нужных параметров
   logFont.lfHeight = -MulDiv(24, pDC->GetDeviceCaps(LOGPIXELSY), 72);
   logFont.lfWeight = FW_LIGHT;
   logFont.lfOutPrecision = OUT_TT_ONLY_PRECIS;
   logFont.lfItalic = 1;
   // создание нового шрифта
   CFont newFont;
   if (!newFont.CreateFontIndirect(&logFont))
      return;
   // внесение нового шрифта и сохранение старого
   CFont* pOldFont = pDC->SelectObject(&newFont);
   // вывод текста
   pDC->TextOut(50,40,_T("Футбол - это прекрасно!"));
   // восстановление старого шрифта
   pDC->SelectObject(pOldFont);
   pDC->TextOut(60,80,_T("Теннис - это увлекательно!"));
}
```

Вывод текста в диалоговое окно выполняется с помощью метода `TextOut`. Первые два параметра метода определяют начальные координаты выводимого текста. Функция `_T()` обеспечивает преобразование третьего параметра к нужному типу.

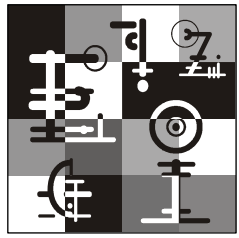
Структура `LOGFONT` служит для хранения информации о параметрах шрифта. Эта структура определена следующим образом:

```
typedef struct tagLOGFONT {
  LONG lfHeight;           // высота шрифта
  LONG lfWidth;            // ширина шрифта
```

```
  LONG lfEscapement;        // угол наклона текста
  LONG lfOrientation;       // угол наклона символов
  LONG lfWeight;            // толщина линий символов
  BYTE lfItalic;            // курсивное начертание
  BYTE lfUnderline;         // подчеркнутое начертание
  BYTE lfStrikeOut;         // перечеркивание текста
  BYTE lfCharSet;           // номер набора символов
  BYTE lfOutPrecision;      // соответствие имеющемуся шрифту
  BYTE lfClipPrecision;     // способ усечения букв
  BYTE lfQuality;           // качество при печати
  // фиксирование ширины и принадлежность семейству
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE]; // гарнитура
} LOGFONT, *PLOGFONT;
```

Большинство из параметров приведенной структуры может иметь нулевое значение или значение по умолчанию. Например, при установке параметру `lfItalic` единичного значения мы обеспечили курсивное начертание выводимого текста.

Функция `SelectObject()` служит для привнесения в контекст устройства различных объектов — шрифтов (в нашем случае), перьев, кистей и др. Кроме того, после привнесения объекта шрифта в контекст устройства его можно использовать для вывода текста в диалоговое окно. Причем при вызове эта функция возвращает указатель на объект, который использовался в контексте до внесения нового объекта. Это позволяет нам восстанавливать старый контекст после завершения работы с новым контекстом (см. листинг 26.2).



Глава 27

Ввод-вывод с помощью класса *CFile*

В приложениях MFC для ввода-вывода файлов предназначен класс `CFile`. Этот класс наследуется непосредственно от класса `CObject` и позволяет упростить использование файлов, представляя файл как объект, который можно создать, читать, записывать и т. д.

Создание объекта класса *CFile*

Класс `CFile` инкапсулирует все функции по обработке файлов любого типа. Для получения доступа к файлу нужно объявить и создать объект класса `CFile`. Например:

```
// объявление объекта класса CFile
public:
       CFile* pFile;
```

Конструктор класса `CFile` позволяет сразу после создания объекта открыть файл. Можно также открыть файл позже, вызвав метод `Open ()`.

Открытие и создание файлов

После создания объекта класса `CFile` можно открыть файл, вызвав метод `Open()`. Для него требуется указать путь к открываемому файлу и режим использования. Прототип метода `Open()` имеет вид:

```
virtual BOOL
Open(LPCTSTR lpszFileName, UINT nOpenFlags,
              CFileException* pError = NULL);
```

Параметр `lpszFileName` задает имя открываемого файла. Можно указывать только имя файла или полное имя файла, включающее полный путь к нему.

Второй параметр `nOpenFlags` определяет действие, выполняемое методом `Open()`, и атрибуты файла. Для полноты картины ниже приведен список ряда возможных значений этого параметра:

- `CFile::modeCreate` — создается новый файл; если указанный файл существует, то его содержимое стирается и длина устанавливается равной нулю;
- `CFile::modeRead` — файл открывается только для чтения;
- `CFile::modeReadWrite` — файл открывается для чтения и записи;
- `CFile::modeWrite` — файл открывается только для записи;
- `CFile::shareCompat` — файл открывается в режиме совместимости; любой другой процесс может открыть его несколько раз;
- `CFile::shareExclusive` — после открытия другим процессам запрещается запись и чтение этого файла;
- `CFile::typeText` — файл открывается в текстовом режиме;
- `CFile::typeBinary` — файл открывается в двоичном (бинарном) режиме.

Текстовый и двоичный режимы используются классами, порожденными от класса `CFile`, например `CStdioFile`. Напомним, что текстовый режим работы с файлами обеспечивает преобразование комбинации символа возврата каретки и символа перевода строки. Рассмотрим пример объявления объекта класса `CFile` и создания файла (листинг 27.1).

Листинг 27.1. Пример объявления объекта класса `CFile` и создания файла

```
CFile f;
CFileException ex;

if( !f.Open( "pFileName.txt", CFile::modeCreate | CFile::modeWrite, &ex ) )
     {
#ifdef _DEBUG
   afxDump << "File could not be opened " << ex.m_cause << "\n";
#endif
}
```

Здесь создается файл с именем `pFileName.txt` в режиме только для чтения. Если файл создать не удается, выдается сообщение об ошибке.

Объявление и создание объекта класса `CFile` можно выполнить с помощью указателя, как показано в следующем примере (листинг 27.2).

Листинг 27.2. Пример объявления и создания объекта класса `CFile` с помощью указателя

```
// объявление указателя на объект класса CFile
public:
       CFile* pFile;
       . . .
// создание объекта класса CFile
pFile = new CFile(lpszPathName,CFile::
                     ↳modeReadWrite|CFile::typeBinary);
```

Здесь выполняется создание двоичного файла в режиме для чтения/записи.

Чтение и запись файлов

Для обмена данными с файлами с помощью класса `CFile` служат следующие методы: `Read()`, `Write()` и `Flush()`.

Чтение данных из открытого файла выполняет метод `Read()`, он имеет следующий прототип:

```
virtual UINT Read (void* lpBuf, UINT nCount);
```

Параметр `lpBuf` указывает буфер, в который записываются прочитанные из файла данные. Параметр `nCount` определяет количество байтов для считывания из файла. Фактически из файла может быть считано меньше байтов, чем запрошено этим параметром, если во время чтения достигнут конец файла. Метод `Read()` возвращает количество байтов, прочитанных из файла.

Запись данных в открытый файл выполняет метод `Write()`, он имеет следующий прототип:

```
virtual void Write(const void* lpBuf, UINT nCount);
```

Метод `Write()` записывает в открытый файл `nCount` байт из буфера `lpBuf`. В случае возникновения ошибки записи, например, переполнения диска, метод `Write()` вызывает исключение.

При использовании метода `Write()` для записи данных в файл они некоторое время могут находиться во временном буфере, ожидая полного заполнения временного буфера. Немедленную запись изменений в файл на диске (не зависимо от заполнения временного буфера) обеспечивает метод `Flush()`. Он имеет следующий прототип:

```
virtual void Flush();
```

Рассмотрим комплексный пример (листинг 27.3) использования методов класса `CFile` для решения прикладной задачи файлового ввода-вывода, постановка которой в более общем приведена в приложении. Пусть требуется выполнить запись данных в бинарный файл на диске, структура которых имеет следующий формат:

Длина	Строковое поле

При таком формате данных можно выполнить чтение строковых данных переменной длины из файла.

Листинг 27.3. Пример файлового ввода-вывода данных переменной длины

```
// CAD.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "CAD.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// The one and only application object

CWinApp theApp;

using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
        int nRetCode = 0;

        // initialize MFC and print and error on failure
        if (!AfxWinInit(::GetModuleHandle(NULL), NULL,
↳::GetCommandLine(), 0))
        {
          // TODO: change error code to suit your needs
          _tprintf(_T("Fatal Error: MFC initialization failed\n"));
          nRetCode = 1;
```

```
    }
    else
    {
      CFile myFile;
      char* fileName = "TEST.DAT";
      BOOL bOpenOK;
      CFileStatus status;
      if( CFile::GetStatus( (LPCTSTR)fileName, status ) )
      {
       // Открываем файл без создания
       bOpenOK = myFile.Open((LPCTSTR)fileName,CFile::modeReadWrite );
       cout << "Open";
      }
      else
      {
       // Создаем файл и открываем
       bOpenOK = myFile.Open( (LPCTSTR)fileName,CFile::modeCreate
       ↳| CFile::modeReadWrite );
       cout << "Create + open";
      }
      // символьная строка для записи
      char buf1[128]="Example - CFile";
      // длина символьной строки
      int buflength = strlen(buf1);
      cout << endl<< buflength;
     myFile.Write(&buflength, sizeof(int));
      myFile.Write(&buf1, buflength+1);
      char buf[128];
      myFile.Seek( 0, CFile::begin );
      // длина символьной строки при чтении
      int len;
      myFile.Read( &len, sizeof(int) );
      cout << endl << len;
      myFile.Read( &buf, len+1);
      cout<< endl << buf << endl;
      myFile.Close();

    }

    return nRetCode;
}
```

Здесь создали консольное приложение, в котором обеспечивается использование классов библиотеки MFC. С помощью метода `GetStatus()` определяется наличие открываемого файла на диске, этот метод возвращает 0, если файл не существует. Если файл имеется, то выполняется его открытие, в противном случается файл создается и открывается.

В результате при повторном выполнении приведенной программы (когда файл уже имеется) получим следующий вывод на консоли:

```
Open
15
15
Example - CFile
```

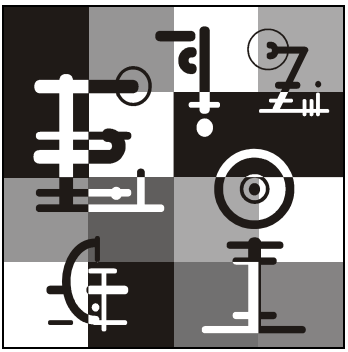
При записи в файл и чтении из файла символьной строки с помощью методов `Write()` и `Read()` класса `CFile` указывается значение buflength+1 и len+1, т. е. на единицу большее, чем длина этой символьной строки. Это обусловлено необходимостью учесть нуль-терминальный символ, находящийся в конце строки.

Примечание

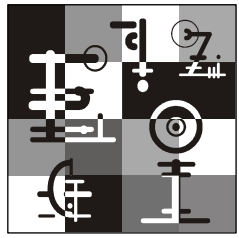
При использовании объектов класса `CFile` в консольных приложениях нужно обеспечить следующие настройки: в диалоговом окне свойств проекта нужно выбрать в группе **Configuration Properties** вкладку **General**, на ней в группе **Project Defaults** для параметра **Character Set** выбрать из списка вариант **No Set**. В случае выбора варианта **Use Unicode Character Set** при создании новых файлов возникают проблемы с кодировкой имени файла. Напомним, что для вызова указанного диалогового окна задают команду **Properties** меню **Project**.

Список литературы

1. Дейтел Х., Дейтел П. Как программировать на C++. — М.: Бином, 2002.
2. Довбуш Г. Ф., Каялайнен О. В., Свистунов С. Г. Лабораторные работы по программированию на языке высокого уровня C++. // Методические указания. — ПГУПС, 2003.
3. Довбуш Г. Ф., Кожомбердиева Г. И., Куранова О. Н. Примеры лабораторных работ на языке C++. // Методические указания. — ПГУПС, 2001.
4. Кейт Г. Использование Microsoft Visual C++.NET. Специальное издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2003.
5. Мешков А. В., Тихомиров Ю. В. Visual C++ и MFC.: Пер. с англ. — СПб.: БХВ-Петербург, 2004.
6. Паппас К., Мюррей У. Эффективная работа: Visual C++.NET. — СПб.: Питер, 2002.
7. Пирогов В. Программирование на Visual C++ .NET. — СПб.: БХВ-Петербург, 2003.
8. Программирование на C++: Учебное пособие/ В. П. Аверкин, А. И. Бобровский, В. В. Веснич и др.; Под ред. проф. А. Д. Хомоненко, 2-е изд. — СПб.: КОРОНА принт, М.: Альтекс-А, 2003.
9. Страуструп Б. Язык программирования C++. — М.: Бином, 2005.
10. Шилдт Г. Самоучитель C++. — СПб.: БХВ-Петербург, 2003.



ПРИЛОЖЕНИЯ



Приложение 1

Контрольные вопросы и задания

Дорогой читатель, надеемся, вы изучили и усвоили весь материал книги. Теперь полезно будет проверить свои знания с помощью предлагаемых контрольных вопросов и заданий. Мы не приводим правильные варианты ответов, потому что, в действительности, все ответы имеются в самой книге.

Вопросы и задания к первой части

1. Укажите этапы подготовки программы к исполнению.
2. Что представляет собой файл спецификации, или заголовочный файл для C++?
3. Приведите схему процесса подготовки программы к исполнению.
4. Перечислите средства, входящие в состав среды программирования.
5. Что представляет собой препроцессор?
6. Для чего предназначен компилятор?
7. Каково назначение компоновщика?
8. Укажите основные функции отладчика среды программирования MVC++.
9. Как соотносятся понятия проект и рабочая область?
10. Укажите действия, требуемые для создания нового проекта в рабочей области.
11. Для чего предназначена папка \Debug в рабочей области?
12. Какие действия нужно выполнить для создания нового проекта консольного приложения внутри рабочей области?
13. Как добавляются в проект консольного приложения исходные файлы: заголовочные файлы (Header Files) и файлы реализации (Source Files)?

14. Какие имена в случае проектов, состоящих из нескольких файлов, рекомендуется использовать для заголовочного файла и файла реализации главной функции?
15. Какие имена в случае проектов, состоящих из нескольких файлов, рекомендуется использовать для файлов с описанием и реализацией используемых в программе классов?
16. Каким образом выполняется пошаговая трассировка?
17. Опишите действия по установке точки останова в программе.
18. Как можно создать собственный набор просматриваемых переменных при трассировке?
19. Укажите назначение функции `main()` в консольной программе.
20. Что делают следующие строки кода:

```
cout << "Beginning of the program\n" ;
return 0 ; ?
```

21. Каково назначение знака `\n`?

Вопросы и задания ко второй части

1. Приведите схему классификации простых типов данных.
2. Что представляет собой размер элемента данных?
3. Укажите форматы объявления констант.
4. Что задают следующие строки кода:

```
const char COPYA = A ;
const bool T = true ;
const char TRUE [ ] = "true" ;?
```

5. Укажите форматы объявления переменных.
6. Охарактеризуйте локальные переменные.
7. Укажите отличия в объявлении глобальных переменных.
8. Приведите пример объявления локальных и глобальных переменных.
9. Укажите назначение и формат оператора разрешения области видимости.
10. Каково назначение объектов `cin` и `cout`?
11. Приведите пример манипулятора, используемого при вводе-выводе.

12. Что задают следующие строки кода:

```
int i ; double x ; char a ;
   cout << "Enter an integer ->\t" ;   cin >> i ;
   cout << "Enter any letter ->\t" ;   cin >> a ;
   cout << "Enter real value ->\t" ;   cin >> x ;
```
?

13. Поясните понятия: операция, операнд, выражение.
14. Укажите назначение арифметических операций инкремента и декремента, приведите примеры их использования.
15. Какие значения будут выведены при выполнении следующих строк кода:

```
int i = 2 ;
cout << ++i << endl ;
cout << i++ << endl ;
cout << i << endl ;
cout << i-- << endl ;
cout << i << endl ;
cout << --i << endl ;
```
?

16. Приведите пример записи операции вычитания с замещением.
17. Поясните смысл логических операций конъюнкция и дизъюнкция.
18. Какие значения будут выведены при выполнении следующих строк кода:

```
int n = 5, j = 0  ;
cout << ! ( n == 5 ) << endl ;
cout << ( n >= 5 && n < 100 ) << endl ;
cout << ( n > 5 || n < 7 ) << endl ;
```
?

19. Укажите назначение и формат операторов присваивания и вызова функции.
20. Что представляет собой составной оператор?
21. Приведите схему условного оператора `if`.
22. Приведите пример использования оператора `if else`.
23. Приведите схему оператора `while`.
24. Укажите формат оператора цикла `for`.
25. Приведите схему и укажите отличия оператора `do while` от других операторов цикла.
26. Для чего и где может использоваться оператор `continue`?
27. Приведите пример использования оператора передачи управления `break`.
28. Укажите назначение и приведите схему оператора `switch`.

29. Каково назначение метки `default`?
30. Приведите пример тернарного оператора.
31. Как рассчитать необходимое для хранения массива количество байтов оперативной памяти?
32. Какие элементы массивов индексируются в следующих строках кода:

```
array [ 2 ] ;
matrix [ 1 ] [ 4 ] ;
b [ i ] [ j ] ;
a [ i + 2 ] ;
```
?

33. Приведите пример объявления, инициализации и вывода элементов одномерного массива.
34. Составьте программу вычисления значения максимального элемента в одномерном массиве.
35. Приведите формат оператора объявления многомерного массива.
36. Приведите пример инициализации матрицы.
37. Приведите пример объявления строки `s`, содержащей не более 50 символов.
38. Составьте программу реверса символьной строки.
39. Приведите форматы объявления указателя.
40. Что делают следующие строки кода:

```
char* ps ;
float *ptr ;
```
?

41. Перечислите допустимые операции для указателей.
42. Укажите действие следующих строк кода:

```
 int x = 5 ;
int * px = & x ;
cout << "Value x\t\t" << x << endl ;
cout << "Address x\t" << & x << endl ;
cout << "Pointer px\t" << px << endl ;.
```

43. Приведите пример операции разыменования адреса.
44. Укажите, что делает следующий фрагмент кода:

```
int array [ ] = { 10, 20, 30, 40, 50 } ;
int* p = array ;
int count = 0 ;
do
{
```

```
        cout << * p++ << '\t' ;
        count++ ;
    } while ( count < size ) ;.
```

45. Укажите назначение и форматы операторов `new` и `delete`.
46. Приведите форматы операторов `new` и `delete`, используемые при работе с динамическими массивами.
47. Как объявляется константный указатель и какие ограничения накладываются при его использовании?
48. Укажите действие следующего кода:

```
char const* pS [ 4 ] =
 { "1. New array", "2. View", "3. Search", "0. Exit" } ;.
```

49. Укажите действие следующего кода:

```
int* p [ SIZE ], x ;
 for ( int i = 0; i < SIZE; i++ )
 {
        cout << "-> " ;       cin >> x ;
        p [ i ] = new int ( x ) ;
 }
```

50. Приведите пример состояния памяти при использовании косвенной адресации.
51. Приведите формат объявления структуры.
52. Назовите варианты доступа к элементам структуры.
53. Укажите действие следующего кода:

```
struct Time {  int hour ; int minute ; int second ; } ;
Time top = { 5, 30, 0 } ; Time end = { 15, 30, 45 } ;.
```

54. Укажите действие следующего кода:

```
struct Road { char name [ 30 ] ; double length ; } ;
Road roadArray [ 20 ] ;
int i = 0 ;
while ( i < size )
{
        strcpy_s ( roadArray [ i ] . name, 30, pName [ i ] ) ;
        roadArray [ i ] . length = len [ i++ ] ;
}
```

55. Какой формат имеет прототип функции?

56. Как осуществляется включение функций в проект приложения?
57. Охарактеризуйте способы передачи параметров функции.
58. Как передается массив в качестве параметра функции?
59. Что обеспечивают приводимые функции и какой способ передачи параметров они используют:

```
 void swapV ( int x, int y )
{ int t = x ; x = y ; y = t ; }

void swapP ( int* px, int* py )
{ int t = * px ; * px = * py ; * py = t ;  }

void swapR ( int& rx, int& ry )
{ int t = rx ; rx = ry ; ry = t ; }
```

?

60. Приведите примеры функций преобразования символов.
61. Укажите основные строковые функции.
62. Что делает функция с прототипом:

```
int strcmp ( const char *s1, const char *s2 ) ;
```

?

63. Приведите примеры функций преобразования строк.
64. В чем суть и что дает использование механизма перегрузки функций?
65. Какие функции указывают следующие прототипы:

```
void swapR ( int&, int& ) ;
void swapR ( char&, char& ) ;
```

?

66. Что представляет собой шаблонная функция?
67. Какое отличие имеет объявление шаблона функции?

Вопросы и задания к третьей части

1. Как соотносятся между собой понятия класса и объекта?
2. Что представляют собой член-данное и член-функция класса?
3. Охарактеризуйте спецификаторы доступа к членам класса.
4. В каком случае указывается заголовок функции вида:

```
тип_функции-члена имя_класса : : имя_функции-члена ( список параметров )
```

?

5. Каким образом осуществляется доступ к объекту по его имени?
6. Каково назначение и формат объявления конструктора по умолчанию?

7. Каково назначение и формат объявления конструктора с параметрами?
8. Каково назначение и формат объявления конструктора копирования?
9. Что представляет собой указатель, обозначаемый ключевым словом `this`?
10. Каким образом можно выделить общую память для всех объектов одного класса?
11. Укажите основные свойства статических данных класса.
12. Охарактеризуйте данные и методы класса, имеющего следующее описание:

```
class C
{
public:
        static double A ;
        int m_x, m_y ;
private:
        static int B ;
public:
        C ( ) : m_x ( 0 ), m_y ( 0 ) {  }
        C ( int x, int y ) : m_x ( x ), m_y ( y ) {   }
        ~C ( ) {  }
        void setB ( int _b ) { B = _b ; }
        int getB ( ) { return B ; }
} ;
```

13. Охарактеризуйте константные методы класса.
14. Каково назначение класса `string`?
15. Каково назначение метода с прототипом

```
basic_string& erase ( size_type _Pos = 0, size_type _Count = npos ) ;
```

?

16. Каковы результаты выполнения кода:

```
string s = "My ram", ss ( "prog" ) ;
cout << s << endl ;
s.insert ( 3, ss ) ;
cout << s << endl ;
```

?

17. Что представляет собой агрегация отношений?
18. Как устанавливаются спецификаторы доступа в производном классе к членам базового класса?
19. Какой формат имеет объявление производного класса при одиночном наследовании?

20. Что обеспечивает следующий код:

```
class CCargo : public CCar
 {
    protected:
    int m_carCapacity ;  // грузоподъемность
    // описание методов класса
  } ; ?
```

21. В каком порядке выполняется конструирование объекта производного класса?
22. Что собой представляет виртуальная функция класса?
23. Укажите схему вызова виртуальной функции через указатель на базовый тип.
24. Что представляет собой множественное наследование?
25. Какой формат имеет объявление производного класса?
26. Что дает использование ключевого слова `virtual` в следующем коде объявления класса `CDerived`:

```
class CBase { … } ;
class CBase_1 : virtual public CBase { … } ;
class CBase_2 : virtual public CBase { … } ;
class CDerived : public CBase_1, public CBase_2 { … } ; ?
```

27. Что представляет собой чистая виртуальная функция?
28. Какие возможности предоставляет перегрузка операторов?
29. Укажите операторы языка, которые запрещено перегружать.
30. Приведите форматы перегрузки унарных и бинарных операторов?
31. Поясните, что выполняется при следующем описании класса:

```
class C
{
        string m_s ;
public:
        C ( ) : m_s ( "" ) { }
        C ( const string& s ) : m_s ( s ) { }
        ~ C ( ) { }
        string getS ( ) const { return m_s ; } ;
        C* operator -> ( ) { return this ; }
        C& operator * ( ) { return *this ; }
```

```
        C& operator [ ] ( const unsigned int i ) { return this [ i ]
; }
} ;
```

32. В чем состоит особенность перегрузки оператора =?
33. Каково назначение дружественных функций?
34. Если дружественная функция определена в другом классе, как для нее следует указать область видимости?
35. Укажите формат глобальной операторной дружественной функции для перегрузки унарных операторов.
36. Укажите действие следующих строк кода:

```
ostream& operator << ( ostream& str, CPoint p )
{ return ( str << p.getX() << '\t' << p.getY() << endl ) ; }
```

37. Укажите особенности перегрузки операторов в производных классах.
38. Что такое шаблон классов?
39. Приведите схему взаимосвязи между шаблоном, классами и объектами.
40. Назовите характерные свойства шаблонных классов.
41. Укажите формат объявления шаблона классов.
42. Приведите формат объявления объектов шаблона классов.
43. Каково назначение следующих строк кода:

```
tepmlate < class U = int, int size = 100 >
class TExample { … } ;
```

?

44. Что представляет собой контейнерный класс?

Вопросы и задания к четвертой части

1. Перегрузите операции << и >> для форматированного ввода объектов класса "матрица вещественных чисел".
2. В каком файле объявлены классы файловых потоков?
3. Назовите основные методы файловых потоков.
4. Когда требуется явно разрывать связь потока с файлом?
5. Какие методы используются для неформатированного ввода-вывода?
6. Как задать бинарный режим работы с файлом?
7. Когда целесообразно использовать средства строко-ориентированного ввода-вывода?

8. Когда целесообразно использовать строковые потоки?
9. Какие классы используются для создания строковых потоков?
10. В чем заключается особенность использования строковых потоков, для которых не указан размер массива символов?
11. Что общего и в чем отличие стандартных потоков библиотек `iostream` и `stdio`?
12. Какие функции библиотеки `stdio` используются для форматированного ввода-вывода?
13. Что такое строка форматирования и спецификатор формата?
14. Напишите программу, в которой стандартный поток `cerr` перенаправлен в файл `C:\error.log` в режиме добавления.
15. Напишите программу с перенаправлением потока `clog` в файл `error.log`.
16. Что называется исключительной ситуацией?
17. Поясните смысл выражения "выброс исключения".
18. В каких случаях генерируется исключение?
19. Какие средства предусмотрены в C++ для обработки исключительных ситуаций?

Вопросы и задания к пятой части

1. Что представляют собой приложения API?
2. Приведите примеры графических объектов.
3. Чем достигается независимость графического интерфейса вывода от устройства?
4. Что представляет собой контекст устройства?
5. Назовите и охарактеризуйте типы контекстов устройств API Windows.
6. Укажите назначение общего контекста устройства для экрана.
7. Назовите составляющие функции `WinMain`.
8. С какой целью выполняется регистрация класса окна?
9. Каково назначение следующих строк кода:

```
wcex.style = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc = WndProc;
```

10. Каково назначение структуры `WNDCLASSEX`?
11. С помощью какой функции выполняется инициализация приложения?

12. Укажите назначение и прототип функции `CreateWindow`.
13. Как устанавливается отображение диалогового окна?
14. Приведите код стандартного цикла обработки сообщений.
15. Укажите назначение функции `GetMessage()`.
16. Приведите схему цикла обработки сообщений.
17. Приведите структуру данных сообщения и укажите назначение его параметров.
18. Поясните назначение следующих строк кода в составе оконной процедуры:

```
case WM_COMMAND:
         wmId    = LOWORD(wParam);
         wmEvent = HIWORD(wParam);
         // Разбор выбора команд меню:
         switch (wmId)
         {
         case IDM_ABOUT:
           DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
           break;
         case IDM_EXIT:
           DestroyWindow(hWnd);
           break;
         default:
           return DefWindowProc(hWnd, message, wParam, lParam);
           }
           break;
```

20. Укажите основные типы оконных сообщений.
21. Поясните роль функции `DefWindowProc()` в оконной процедуре обработки сообщений.
22. Назовите шаги создания приложения API Windows.
23. Каким образом задается реакция приложения API Windows на выполнение команд меню?
24. Укажите действия, выполняемые при создании меню в среде MVC++.
25. Перечислите и охарактеризуйте стили диалоговых окон.
26. Поясните действие следующих строк кода:

```
PopUpHwnd = CreateWindow(
   szPopUpClassName,
   "Временное окно",        //
```

```
    WS_POPUPWINDOW | WS_CAPTION | WS_VISIBLE,
    100, 100, 200, 100,
    MainHwnd,
    0,
    hInstance,
      NULL);
```

27. Укажите действия при создании нового диалогового окна в среде MVC++.
28. Каким образом осуществляется помещение элементов управления на диалоговое окно?
29. Укажите назначение наиболее распространенных элементов управления.
30. Поясните назначение следующих строк кода в файле `resource.h`:

```
#define IDD_ABOUTBOX                    103
#define IDM_ABOUT               104
#define IDM_EXIT                105
#define IDI_EXAMPLEAPI          107
#define IDI_SMALL               108
#define IDC_EXAMPLEAPI          109
#define IDC_MYICON                      2
```

31. Приведите прототипы оконных процедур для дочерних диалоговых окон с именами: **About**, **Add**, **Create**, **Delete**.
32. Укажите назначение следующих строк кода в состав главной оконной процедуры:

```
case IDM_DELETE:
                   DialogBox(hInst, (LPCTSTR)IDD_DELETE, hWnd,
(DLGPROC)Delete);
                   InvalidateRect( hWnd,NULL,true);
                   UpdateWindow(hWnd);
                   break;
```

33. В каком файле помещаются описания всех оконных процедур для диалоговых окон?

Вопросы и задания к шестой части

1. Какую роль играет класс `CObject` в составе библиотеки MFC?
2. Приведите примеры соответствия классов графических объектов MFC и типов дескрипторов Windows.

3. Укажите этапы создания приложения MFC.
4. Охарактеризуйте типы приложений MFC.
5. Какие классы создает Мастер приложений MFC?
6. Каково назначение класса `CMFCSDIApp` в составе приложения с именем MFCSDI?
7. Укажите назначение директивы препроцессора `#pragma`.
8. Поясните назначение функции `InitInstance ()`.
9. Какую роль играют функции `ParseCommandLine ()` и `ProcessShellCommand ()`?
10. Что представляет собой карта сообщений?
11. Приведите пример первой части карты сообщений.
12. Поясните назначение следующих строк кода в составе второй части карты сообщений:

```
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
```

13. Запишите код строки для второй части карты сообщений, которая обеспечивает вызов метода `OnFileOpen` класса `CWinApp` при выполнении команды меню с идентификатором `ID_FILE_OPEN`.
14. Поясните действия кода в составе следующей функции:

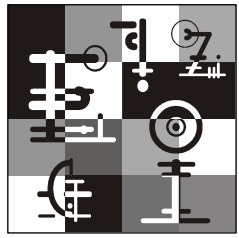
```
void CMFCSDIApp: :OnAppAbout ()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal();
}
```

15. Опишите внутренний механизм работы карты сообщений.
16. Приведите примеры макросов карт сообщений.
17. Укажите назначение макросов `ON_COMMAND` и `ON_CONTROL`.
18. Приведите примеры префиксов, используемых для распознавания типов передаваемых сообщений.
19. Для чего служит сообщение `NM_THEMECHANGED`?
20. Укажите шаги создания класса окна для приложения MFC.
21. Как осуществляется доступ к элементам управления диалогового окна приложения MFC?
22. Каково назначение класса `CFile`?

23. Приведите пример кода создания объекта класса `CFile` .
24. Поясните действие следующего кода:

```
CFile f;
CFileException ex;

if( !f.Open( "pFileName.txt", CFile::modeCreate | CFile::modeWrite,
&ex ) )
     {
#ifdef _DEBUG
   afxDump << "File could not be opened " << ex.m_cause << "\n";
#endif
}
```

25. Приведите пример объявления и создания объекта класса `CFile` с помощью указателя.



Приложение 2

Пример разработки консольного приложения MVC++

Методические указания для разработки

В приложении описана работа, которая посвящена изучению ввода-вывода в MVC++ с помощью функций.

Предлагается выбрать предметную область и представить класс, описывающий эту область (необходимо указать, какие поля и атрибуты будут в классе, их названия и типы данных).

Разрабатываемая программа должна обеспечивать следующую функциональность:

- ❒ просмотр пользователем записей в двоичном файле;
- ❒ добавление записей в двоичный файл (ввод осуществляется с клавиатуры);
- ❒ удаление записей;
- ❒ изменение записей;
- ❒ поиск записей;
- ❒ хранение записей в двоичном файле на диске;
- ❒ чтение записей из текстового файла и сохранение их в двоичный файл.

Примерный интерфейс пользователя приведен на рис. П2.1.

Реализация должна отвечать следующим требованиям:

- ❒ Не допускается ввод дублирующих записей, т. е. записей, имеющих одинаковые ключевые поля. Это должно проверяться как при вводе с клавиатуры, так и при чтении из текстового файла.
- ❒ Во время исполнения в программе может существовать только один объект класса предметной области и только в то время, когда он необходим. После использования он должен уничтожаться.

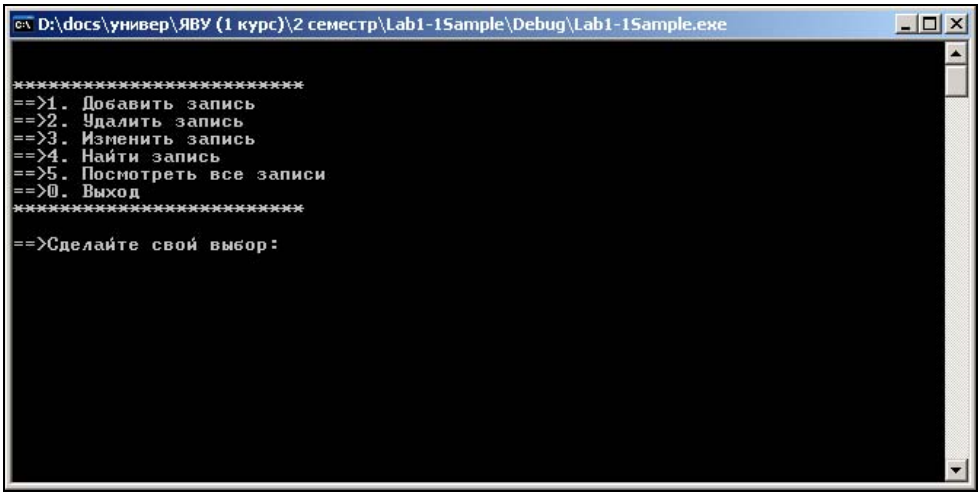


Рис. П2.1. Интерфейс пользователя

- Поиск должен осуществляться по индексу, для чего после запуска программы должен формироваться индексный массив, каждый элемент которого должен содержать в себе пары значений: ключевое поле записи — позиция этой записи в двоичном файле данных.
- При изменении записей, удалении, добавлении новых записей должен изменяться индексный массив. Удаление данных из двоичного файла можно реализовать как при каждом удалении или изменении записи, так и при выходе из программы.
- Запись одного объекта в двоичный файл должна происходить одним вызовом функции записи, обращающейся к диску, для чего в классе предметной области должен быть метод для создания буфера записи.
- Чтение объекта из двоичного файла необходимо продумать и реализовать так, чтобы минимизировать количество вызовов функции чтения диска.

В листинге П2.1 класс описывает автомобили, приведена полная спецификация класса.

Листинг П2.1. Полная спецификация класса, описывающего автомобиль

```
class CAuto
{
public:
        CAuto(void);
        ~CAuto(void);
```

```
private:
        char* model;
        char* color;
        int   year;
        float price;
public:
//методы setXXX – для установления значений полям класса
        void setModel(char* model);
        void setColor(char* color);
        void setYear(int year);
        void setPrice(float price);

//методы getXXX – для получения значений полей класса
        char* getModel(void);
        char* getColor(void);
        float getPrice(void);
        int   getYear(void);

//метод toBinString – преобразует объект в строку, соответствующую
//формату файла
        char* toBinString(void);

//метод toObject – преобразует полученную строку в объект
        static CAuto* toObject(char* binStr);

//метод binaryLength возвращает количество байтов, необходимых
//для формирования буфера
        int binaryLength(void);
};
```

В примере используется структура файла следующего вида:

- `L` — суммарная длина всех полей класса `CAuto`;
- `model`, `color`, `year`, `price` — поля класса `CAuto`;
- `\0` — разделитель, необходим для возможности отделения полей `model` и `color` при чтении.

Далее будут рассмотрены примеры реализации некоторых методов следующих классов:

- `CAuto` — класс предметной области, описывает автомобиль;
- `CCmdMenu` — класс, отвечающий за взаимодействие с пользователем;

- `CControl` — класс для управления работой других классов;
- `CIndex` — класс для работы с индексом;
- `CKey` — класс, описывающий элемент индекса;
- `CBinaryFile` — класс для работы с двоичным файлом.

Общая структура приложения

Все классы, их основные методы, атрибуты, а также связи между классами представлены на диаграмме классов (рис. П2.2).

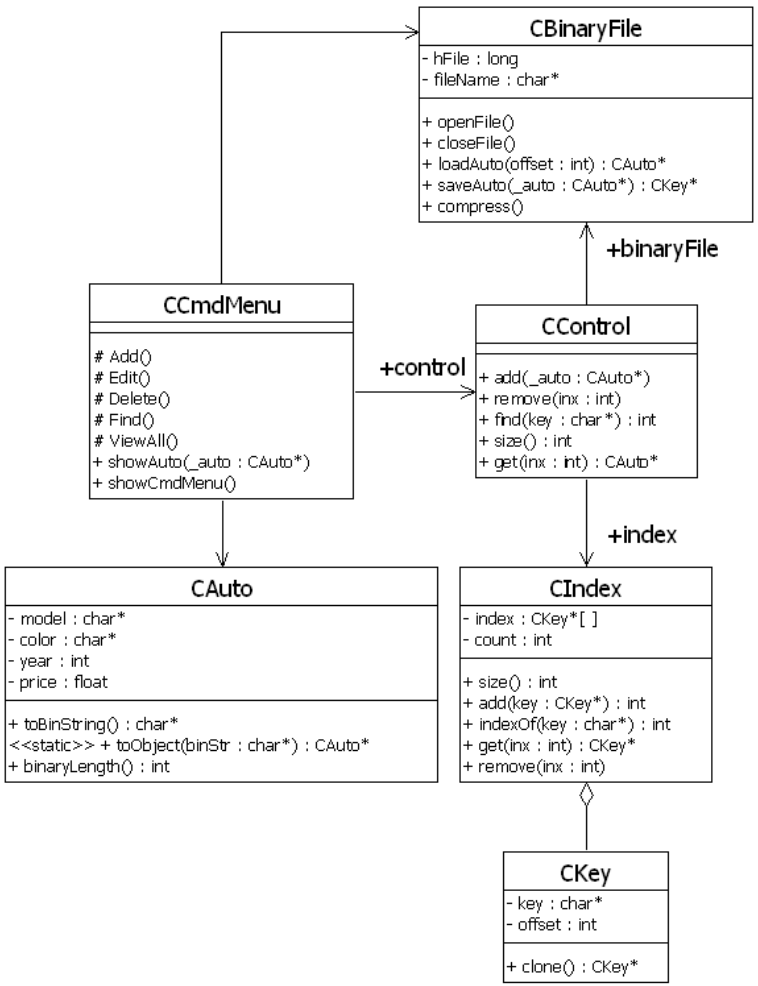


Рис. П2.2. Диаграмма классов

Особенности реализации класса *CAuto*

Все атрибуты класса объявлены в разделе `private`, для работы с ними должны быть предусмотрены стереотипные методы `setXXX()` и `getXXX()`.

Для стандартного чтения и записи объектов в бинарный файл необходимо реализовать методы, преобразующие объект в строку, и наоборот — формирующие из строки объект. Прежде чем преобразовывать объект в строку, нужно выделить память под буфер, для этого подсчитывается необходимое количество байтов — метод `binaryLength()`. Этот метод суммирует размеры символьных массивов, размеры числовых полей, кроме того, он учитывает 1 байт для записи суммы длин символьных массивов и 1 байт для символа-разделителя `\0`. Для этого понадобятся следующие функции и операторы:

- `size_t strlen(const char *string)` — определяет количество символов в строке;
- `sizeof(type-name)` — определяет необходимое количество байтов для представления типа данных `type-name`.

Пример реализации метода `binaryLength()` приведен в листинге П2.2.

Листинг П2.2. Вычисление размера буфера записи

```
int CAuto::binaryLength(void)
{
       return strlen(model)+strlen(color)+sizeof(int)+sizeof(float)+2;
}
```

После определения размера буфера можно выделить для него память (оператор `new`) и приступить к его формированию. Для этого понадобятся следующие функции и операторы:

- `char *strcpy(char *strDestination, const char *strSource)` — копирует строку `strSource` в `strDestination`;
- `void *memcpy(void *dest, const void *src, size_t count)` — копирует из области памяти `src` в `dest`.

Пример фрагмента реализации метода `toBinString()` приведен в листинге П2.3.

Листинг П2.3. Формирование буфера записи (фрагмент)

```
//определяем размер необходимого буфера и выделяем для него память
int buffSize =  binaryLength();
char* bBuff = new char[buffSize];
```

```
int bInx = 0;
//копируем в буфер длину всей записи (1 байт) и смещаем индекс на единицу
memcpy(&bBuff[bInx++], &buffSize,1);
//копируем model в буфер и смещаем индекс на количество скопированных
символов
strcpy(&bBuff[bInx], model);
bInx+=strlen(model);
//копируем в буфер символ-разделитель и смещаем индекс
bBuff[bInx++]='\0';
strcpy(&bBuff[bInx], color);
bInx+=strlen(color);
//копируем в буфер поле year типа int, копируется sizeof(int) байт,
//и смещаем индекс
memcpy(&bBuff[bInx], &year, sizeof(int));
bInx+=sizeof(int);
```

Примечание

При выделении памяти возможна ситуация, когда оператор `new` не сможет выделить память и вернет `NULL`. Это необходимо предусмотреть.

Работу функции можно проконтролировать в отладчике. Например, возьмем объект, содержащий следующую информацию:

- `model = z4;`
- `color = silver;`
- `year = 2002;`
- `price = 45000.5.`

Для заданного формата файла должен получиться буфер, как на рис. П2.3, сформированный буфер выделен рамкой.

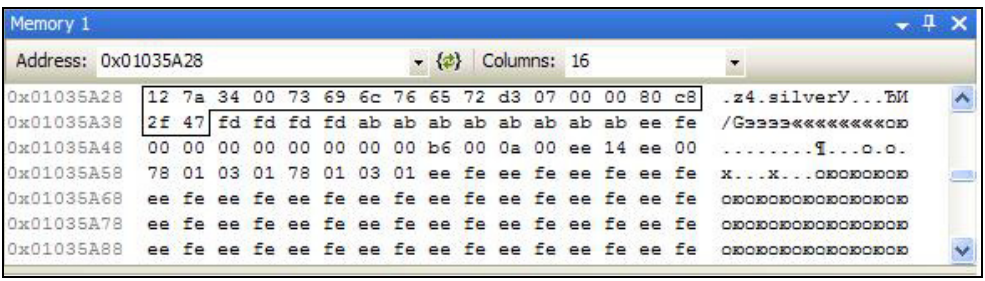


Рис. П2.3. Просмотр буфера в отладчике

Метод `toObject(char* binStr)` должен реализовать обратный процесс. Фрагмент реализации этого метода приведен в листинге П2.4.

Листинг П2.4. Формирование объекта из строки (фрагмент)

```
CAuto* newAuto = new CAuto();
char* cBuff = new char[BUFF_SIZE];
//из буфера выделяем длину всей записи, явно приводя тип данных
short length = (short)binStr[0];
//копируем из входного буфера модель (символы до разделителя \0) и
//заносим модель в объект
strcpy(cBuff, binStr+1);
newAuto->setModel(cBuff);
int buffLen = strlen(cBuff);
//копируем из входного буфера цвет, вычисляя длину этого поля
// зная формат данных, записываем в конец строки символ завершения
//строки (\0), после чего заносим цвет в объект
memcpy(cBuff, &binStr[buffLen+2], length-sizeof(int)-sizeof(float)-buffLen-1);
cBuff[length-sizeof(int)-sizeof(float)-buffLen-2]='\0';
newAuto->setColor(cBuff);
//аналогично предыдущему копируется и устанавливается год (поле year)
int year;
memcpy(&year, &binStr[length-sizeof(int)-sizeof(float)], sizeof(int));
newAuto->setYear(year);
```

Класс *CCmdMenu*

Его задача вывести пользователю возможные варианты действий и определить выбор пользователя, после чего дать возможность вводить информацию либо сразу передать управление классу `CControl`.

Интерфейс пользователя лучше делать на русском языке. Вывод русского текста в консоли осложнен тем, что в среде разработки и консоли используются различные кодировки, в VS могут быть разные кодировки, а в консоли используется DOS-CP866. Существует несколько вариантов решения этой проблемы, один из самых простых методов представлен в листинге П2.5.

Листинг П2.5. Пример вывода русского текста в консольном приложении

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

```
int main(int argc, char* argv[])
{
    char s[]="Привет всем!";
    CharToOem(s,s);
    cout << s;
    return 0;
}
```

Функция `BOOL CharToOem(LPCTSTR lpszSrc, LPSTR lpszDst)` переводит кодировку текста в необходимую.

Классы для организации работы с индексом *CIndex* и *CKey*

Класс `CKey` содержит в себе ключевое поле из класса предметной области и смещение в двоичном файле данных. Метод `clone()` класса `CKey` создает копию объекта и возвращает его.

Класс `CIndex` содержит в себе массив указателей на объекты класса `CKey` и позволяет добавлять, удалять, искать и получать объекты.

Класс *CBinaryFile*

Класс `CBinaryFile` предназначен для работы с двоичным файлом, только в методах этого класса могут вызываться функции для работы с файлом (открытие, запись, чтение, позиционирование и др.).

Для выполнения первой части работы необходимо использовать следующие функции:

- `int _access(const char *path, int mode)` — позволяет узнать права доступа к файлу (можно проверить, существует ли файл на диске);
- `int _creat(const char *filename, int pmode)` — создает файл с указанным именем и устанавливает уровень доступа;
- `int _open(const char *filename, int oflag[, int pmode])` — открывает указанный файл для операций, указанных параметром `oflag` (последний параметр, указывающий права доступа, опционален);
- `int _close(int fd)` — закрывает файл;
- `int _chsize(int fd, long size)` — устанавливает для файла указанный размер;

- `long _lseek(int fd, long offset, int origin)` — перемещает позицию в файле на `offset` байт от `origin`;
- `int _read(int fd, void *buffer, unsigned int count)` — читает из файла `fd count` байтов в память по адресу `buffer`, возвращает количество реально прочитанных байтов;
- `int _write(int fd, const void *buffer, unsigned int count)` — записывает в файл `fd count` байт из буфера `buffer`, возвращает количество реально записанных байтов;
- `long _tell(int fd)` — возвращает текущую позицию в файле;
- `int _eof(int fd)` — позволяет определить, достигнут ли конец файла.

В листинге П2.6 приведен пример работы с файлом.

Листинг П2.6. Пример загрузки объекта из файла

```
CAuto* CBinaryFile::loadAuto(int offset)
{
//смещаем позицию в файле на offset байт от начала файла
       _lseek(hFile, offset, SEEK_SET);
       int binaryLength=0;
//чтение первого байта, в соответствии с форматом это длина всей записи
       _read(hFile, &binaryLength, 1);
//выделение памяти под буфер и чтение в него всей записи
       char* cBuff = new char[binaryLength];
       cBuff[0]=(char)binaryLength;
       _read(hFile, &cBuff[1], binaryLength-1);
//из прочитанной записи формируется объект, освобождается память
//под буфер и возвращается указатель на сформированный объект
       CAuto* lAuto = CAuto::toObject(cBuff);
       delete [] cBuff;
       return lAuto;
}
```

Класс `CBinaryFile` не работает с текстовым файлом, для этого необходимо создать специальный класс и реализовать в нем открытие.

Класс управления *CControl*

Класс управления используется для координации действий других классов, для примера приведем листинг П2.7 метода добавления нового объекта.

Листинг П2.7. Добавление нового объекта

```
void CControl::add(CAuto* auto_)
{
//параметр метода — указатель на новый объект, его сохраняем в файл, в
//результате получаем указатель на объект класса CKey, он содержит ключ
//добавляемого объекта и позицию в файле, куда он был записан
       CKey* key = binaryFile->saveAuto(auto_);
//добавляем полученные данные в индекс
       index->add(key);
//удаляем объект класса CKey (при добавлении в индекс была создана копия,
//поэтому данные не будут потеряны)
       delete key;
}
```

Пример консольного приложения MVC++ по файловому вводу-выводу

Рассмотрим пример одного из возможных вариантов реализации работы (листинг П2.8).

Листинг П2.8. Пример разработки консольного приложения

```
/* Файл - Auto.h
*   Класс - модель (Model)
*   В простейшем случае представляет объект предметной области
*   (C) Сухоногов А. М., 2005.
*/
#pragma once

class CAuto
{
public:
       CAuto(void);
       ~CAuto(void);
private:
       char* model;
       char* color;
```

```
        int year;
        float price;
public:
        void setModel(char* model);
        void setColor(char* color);
        void setYear(int year);
        void setPrice(float price);
        char* getModel(void);
        char* getColor(void);
        float getPrice(void);
        int getYear(void);
        char* toBinString(void);
        static CAuto* toObject(char* binStr);
        int binaryLength(void);
};
/* Файл - Auto.cpp
*   Класс - модель (Model)
*   В простейшем случае представляет объект предметной области
*   (C) Сухоногов А. М., 2005.
*/
#include "StdAfx.h"
#include ".\auto.h"
#include "auto.h"

CAuto::CAuto(void)
{
}

CAuto::~CAuto(void)
{
        delete [] model;
        delete [] color;
}

void CAuto::setModel(char* model)
{
        this->model = new char[strlen(model)+1];
        strcpy(this->model, model);
}
```

```
void CAuto::setColor(char* color)
{
       this->color = new char[strlen(color)+1];
       strcpy(this->color, color);
}

void CAuto::setYear(int year)
{
       this->year = year;
}

void CAuto::setPrice(float price)
{
       this->price = price;
}

char* CAuto::getModel(void)
{
       return model;
}

char* CAuto::getColor(void)
{
       return color;
}

float CAuto::getPrice(void)
{
       return price;
}

int CAuto::getYear(void)
{
       return year;
}

char* CAuto::toBinString(void)
{
```

```
        int buffSize =  binaryLength();
        char* bBuff = new char[buffSize];
        int bInx = 0;
        memcpy(&bBuff[bInx++], &buffSize,1);
        strcpy(&bBuff[bInx], model);
        bInx+=strlen(model);
        bBuff[bInx++]='\0';
        strcpy(&bBuff[bInx], color);
        bInx+=strlen(color);
        memcpy(&bBuff[bInx], &year, sizeof(int));
        bInx+=sizeof(int);
        memcpy(&bBuff[bInx], &price, sizeof(float));
        return bBuff;
}

CAuto* CAuto::toObject(char* binStr)
{
    CAuto* newAuto = new CAuto();
        char* cBuff = new char[BUFF_SIZE];
        short length = (short)binStr[0];
        strcpy(cBuff, binStr+1);
        newAuto->setModel(cBuff);
        int buffLen = strlen(cBuff);
        memcpy(cBuff, &binStr[buffLen+2], length-sizeof(int)- ↳si-
zeof(float)-buffLen-1);
        cBuff[length-sizeof(int)-sizeof(float)-buffLen-2]='\0';
        newAuto->setColor(cBuff);
    int year;
        memcpy(&year, &binStr[length-sizeof(int)-sizeof(float)], ↳si-
zeof(int));
        newAuto->setYear(year);
        float price;
        memcpy(&price, &binStr[length-sizeof(float)], sizeof(float));
        newAuto->setPrice(price);
        delete [] cBuff;
        return newAuto;
}

int CAuto::binaryLength(void)
```

```
{
       return strlen(model)+strlen(color)+sizeof(int)+sizeof(float)+2;
}
/* Файл BinaryFile.h
*   Класс обеспечения постоянства (персистентности) объектов
*   Реализует хранение объектов предметной области в бинарном файле
*   (C) Сухоногов А. М., 2005.
*/
#pragma once
#include ".\Index.h"
#include ".\Auto.h"
#include ".\Key.h"

class CBinaryFile
{
private:
       long hFile;
       char* fileName;
       CIndex* index;
       void buildIndex(void);
public:
       CBinaryFile(char* fileName, CIndex* index);
       ~CBinaryFile(void);
       void openFile();
       void closeFile(void);
       CAuto* loadAuto(int offset);
    CKey* saveAuto(CAuto* auto_);
    void compress(void);
};
/* Файл BinaryFile.cpp
*   Класс обеспечения постоянства (персистентности) объектов
*   Реализует хранение объектов предметной области в бинарном файле
*   (C) Сухоногов А. М., 2005.
*/
#include "StdAfx.h"
#include ".\binaryfile.h"
#include ".\Auto.h"
CBinaryFile::CBinaryFile(char* fileName, CIndex* index)
{
```

```
        this->fileName = fileName;
        this->index = index;
}

CBinaryFile::~CBinaryFile(void)
{
}

void CBinaryFile::openFile()
{
   if(_access( fileName, 0 ) == -1)
     hFile = _creat( fileName, _S_IREAD | _S_IWRITE );
   else
   {
        hFile = _open(fileName,_O_RDWR);
     buildIndex();
   }
}

void CBinaryFile::closeFile(void)
{
       _close(hFile);
}

void CBinaryFile::compress(void)
{
       int offset = 0;
       for(int i=0; i<index->size();i++)
       {
              CAuto* cAuto = loadAuto(index->get(i)->getOffset());
        char* cBuff = cAuto->toBinString();
              _lseek(hFile, offset, SEEK_SET);
        _write(hFile, cBuff, cAuto->binaryLength());
              offset+=cAuto->binaryLength();
        delete [] cBuff;
              delete cAuto;
       }
       _chsize(hFile, offset);
}
```

```
CAuto* CBinaryFile::loadAuto(int offset)
{
        _lseek(hFile, offset, SEEK_SET);
        int binaryLength=0;
        _read(hFile, &binaryLength, 1);
        char* cBuff = new char[binaryLength];
        cBuff[0]=(char)binaryLength;
        _read(hFile, &cBuff[1], binaryLength-1);
        CAuto* lAuto = CAuto::toObject(cBuff);
        delete [] cBuff;
        return lAuto;
}

CKey* CBinaryFile::saveAuto(CAuto* auto_)
{
        CKey* key = new CKey(auto_->getModel());
        char* bBuff = auto_->toBinString();
        key->setOffset(_lseek(hFile, 0L, SEEK_END));
        _write(hFile, bBuff, auto_->binaryLength());
        delete [] bBuff;
        return key;
}

void CBinaryFile::buildIndex(void)
{
    int offset = 0;
       while (!_eof(hFile))
       {
              CAuto* iAuto = loadAuto(offset);
              CKey* key = new CKey(iAuto->getModel());
              key->setOffset(offset);
              index->add(key);
              offset+=iAuto->binaryLength();
        delete key;
              delete iAuto;
       }
}
/* Файл CmdMenu.h
```

```
*   Класс - представление (View)
*   Реализует интерактивное взаимодействие с пользователем
*   (C) Сухоногов А. М., 2005.
*/
#pragma once
#include ".\Control.h"
#include ".\Auto.h"

class CCmdMenu
{
private:
       CControl* control;
protected:
       void Add(void);
       void Delete(void);
       void Edit(void);
       void Find(void);
       void ViewAll(void);
public:
       CCmdMenu(void);
       ~CCmdMenu(void);
       void showCmdMenu(void);
       void showAuto(CAuto* auto_);
};
/* Файл CmdMenu.cpp
*   Класс - представление (View)
*   Реализует интерактивное взаимодействие с пользователем
*   (C) Сухоногов А. М., 2005.
*/
#include "StdAfx.h"
#include ".\cmdmenu.h"
#include <Windows.h>

using namespace std;

CCmdMenu::CCmdMenu(void)
{
       control = new CControl();
}
```

```
CCmdMenu::~CCmdMenu(void)
{
    delete control;
}

void CCmdMenu::Add(void)
{
       CAuto* auto_ = new CAuto();
    char* cBuff = new char[BUFF_SIZE];
       int year;
       float price;

       cout << "\nEnter model of auto ->  ";
       cin >> cBuff;
       auto_->setModel(cBuff);

       cout << "\nEnter color of auto ->  ";
       cin >> cBuff;
       auto_->setColor(cBuff);

       cout << "\nEnter year designed of auto ->  ";
       cin >> year;
       auto_->setYear(year);

       cout << "\nEnter price of auto ->  ";
       cin >> price;
       auto_->setPrice(price);

       cout << endl;

       if (control->find(auto_->getModel())==NULL_OFFSET)
              control->add(auto_);

       delete auto_;
}

void CCmdMenu::Delete(void)
{
```

```
    char* model = new char[BUFF_SIZE];
       cout << "\nEnter model of auto ->  ";
       cin >> model;
       int inx = control->find(model);
       if (inx>NULL_OFFSET)
         control->remove(inx);
       delete [] model;
}

void CCmdMenu::Edit(void)
{
       CAuto* eAuto = new CAuto();
       char* cBuff = new char[BUFF_SIZE];

       cout << "\nEnter exist model of car ->  ";
       cin >> cBuff;
       int existInx = control->find(cBuff);
       if (existInx>NULL_OFFSET)
       {
          cout << "\nEnter model of car ->  ";
          cin >> cBuff;
          eAuto->setModel(cBuff);

          cout << "\nEnter color ->  ";
          cin >> cBuff;
          eAuto->setColor(cBuff);

       int year;
          cout << "\nEnter year of designed ->  ";
          cin >> year;
          eAuto->setYear(year);

          float price;
          cout << "\nEnter price ->  ";
          cin >> price;
          eAuto->setPrice(price);

          int inx = control->find(eAuto->getModel());
          if (inx==NULL_OFFSET||inx==existInx)
```

```
          {
                 control->remove(existInx);
                 control->add(eAuto);
          }
       }
       delete [] cBuff;
       delete eAuto;
}

void CCmdMenu::Find(void)
{
    char* model = new char[BUFF_SIZE];
       cout << "\nEnter model of auto ->  ";
       cin >> model;
       cout << endl;
       int inx = control->find(model);
       if (inx > NULL_OFFSET)
       {
              CAuto* fAuto = control->get(inx);
              showAuto(fAuto);
              delete fAuto;
       }
    delete [] model;
}

void CCmdMenu::ViewAll(void)
{
       for(int i=0;i<control->size();i++)
       {
              CAuto* auto_ = control->get(i);
        showAuto(auto_);
              delete auto_;
       }
}

void CCmdMenu::showCmdMenu(void)
{

       char s1[]="\n==>1. Добавить запись    ";
```

```
char s2[]="\n==>2. Удалить запись     ";
char s3[]="\n==>3. Изменить запись    ";
char s4[]="\n==>4. Найти запись       ";
char s5[]="\n==>5. Посмотреть все записи";
char s6[]="\n==>0. Выход         ";
char s7[]="\n==>Сделайте свой выбор:  ";
CharToOem(s1,s1);
CharToOem(s2,s2);
CharToOem(s3,s3);
CharToOem(s4,s4);
CharToOem(s5,s5);
CharToOem(s6,s6);
CharToOem(s7,s7);

char Choise;
do {
       cout << "\n                          ";
       cout << "\n*************************";
       cout << s1;
       cout << s2;
       cout << s3;
       cout << s4;
       cout << s5;
       cout << s6;
 cout << "\n*************************";
       cout << "\n                          ";
       cout << s7;
       cin >> Choise;
       cout << endl;
       switch(Choise)
       {
              case '1':
                     Add();
                     break;
              case '2':
                     Delete();
                     break;
              case '3':
                     Edit();
```

```
                        break;
                case '4':
                        Find();
                        break;
                case '5':
                        ViewAll();
                        break;
            }
        }
        while (Choise!='0');
}

void CCmdMenu::showAuto(CAuto* auto_)
{
        cout << "Model: " << auto_->getModel() << endl;
        cout << "Color: " << auto_->getColor() << endl;
        cout << "Year: "  << auto_->getYear() << endl;
        cout << "Price: "  << auto_->getPrice() << endl;
        cout << endl;
}
/* Файл Control.h
*   Класс - контроллер (Controller)
*   Реализует основную функциональность приложения
*   (C) Сухоногов А. М., 2005.
*/

#pragma once
#include ".\BinaryFile.h"
#include ".\Auto.h"

class CControl
{
public:
        CControl(void);
        ~CControl(void);
        void add(CAuto* auto_);
private:
        CIndex* index;
        CBinaryFile* binaryFile;
```

```
public:
        void remove(int inx);
        int find(char* key);
        int size(void);
        CAuto* get(int inx);
};
/* Файл Control.cpp
*   Класс - контроллер (Controller)
*   Реализует основную функциональность приложения
*   (C) Сухоногов А. М., 2005.
*/
#include "StdAfx.h"
#include ".\control.h"

CControl::CControl(void)
{
        index = new CIndex();
        binaryFile = new CBinaryFile(binaryFileName, index);
        binaryFile->openFile();
}

CControl::~CControl(void)
{
        binaryFile->compress();
        binaryFile->closeFile();
        delete index;
        delete binaryFile;
}

void CControl::add(CAuto* auto_)
{
        CKey* key = binaryFile->saveAuto(auto_);
        index->add(key);
        delete key;
}

void CControl::remove(int inx)
{
        if (inx > NULL_OFFSET)
```

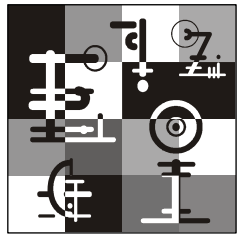
```
                index->remove(inx);
}

int CControl::find(char* key)
{
        return index->indexOf(key);
}

int CControl::size(void)
{
        return index->size();
}

CAuto* CControl::get(int inx)
{
        CKey* key = index->get(inx);
        return binaryFile->loadAuto(key->getOffset());
}
/* Файл Index.h
*   Класс индекса
*   Представляет массив ключей для коллекции объектов предметной области.
*   (C) Сухоногов А. М., 2005.
*/
#pragma once
#include ".\Key.h"

class CIndex
{
public:
        CIndex(void);
        ~CIndex(void);
private:
    CKey* index[INDEX_SIZE];
        int count;
public:
        int size(void);
        int add(CKey* key);
        int indexOf(char* key);
        CKey* get(int inx);
```

```
        void remove(int inx);
};
/* Файл Index.cpp
*   Класс индекса
*   Представляет массив ключей для коллекции объектов предметной области.
*   (C) Сухоногов А. М., 2005.
*/
#include "StdAfx.h"
#include ".\index.h"

CIndex::CIndex(void)
{
        count=0;
}

CIndex::~CIndex(void)
{
        for (int i=0; i<count; i++)
           delete index[i];
}

int CIndex::size(void)
{
        return count;
}

int CIndex::add(CKey* key)
{
        index[count++]=key->clone();
        return count-1;
}



int CIndex::indexOf(char* key)
{
        for (int i=0; i<count; i++)
                if (strcmp(key, index[i]->getKey())==0)
                        return i;
        return NULL_OFFSET;
```

```
}

CKey* CIndex::get(int inx)
{
        return (inx<count)?index[inx]:NULL;
}

void CIndex::remove(int inx)
{
        delete index[inx];
        for (int i=inx; i<count-1; i++)
                index[i] = index[i+1];
        count--;
}
/*  Файл Key.h
*   Класс ключа
*   Представляет объект предметной области в индексе.
*   (C) Сухоногов А. М., 2005.
*/
#pragma once

const int NULL_OFFSET = -1;

class CKey
{
public:
        CKey(char* key);
        ~CKey(void);
private:
        char* key;
        int offset;
public:
        void setOffset(int offset);
        int getOffset(void);
        CKey* clone(void);
        char* getKey(void);
};
/* Файл CKey.cpp
*   Класс ключа
```

```
*   Представляет объект предметной области в индексе.
*   (C) Сухоногов А. М., 2005.
*/

#include "StdAfx.h"
#include ".\key.h"

CKey::CKey(char* key)
{
       this->key = new char[strlen(key)+1];
       strcpy(this->key, key);
       offset = NULL_OFFSET;
}

CKey::~CKey(void)
{
    delete [] key;
}

void CKey::setOffset(int offset)
{
       this->offset = offset;
}

int CKey::getOffset(void)
{
       return offset;
}

CKey* CKey::clone(void)
{
       CKey* clone = new CKey(this->key);
       clone->offset = offset;
       return clone;
}

char* CKey::getKey(void)
{
       return key;
```

```
}
// Lab1-1Sample.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include ".\CmdMenu.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
        CCmdMenu cmdMenu;
        cmdMenu.showCmdMenu();
        return 0;
}
```



Приложение 3

Описание компакт-диска

На компакт-диске в папках с именами соответствующих глав содержатся листинги примеров из книги. Все листинги представляют либо фрагменты исходных кодов, либо полные исходные коды приложений, которые отлаживались в среде Visual Studio 2005. На их основе нужно создать приложение с помощью Мастера и выполнить заполнение соответствующих заголовочных файлов и файлов реализации согласно рекомендациям из раздела третьей главы книги о добавлении исходных файлов в проект.

Предметный указатель

(окончание рубрики см. на стр. 512)