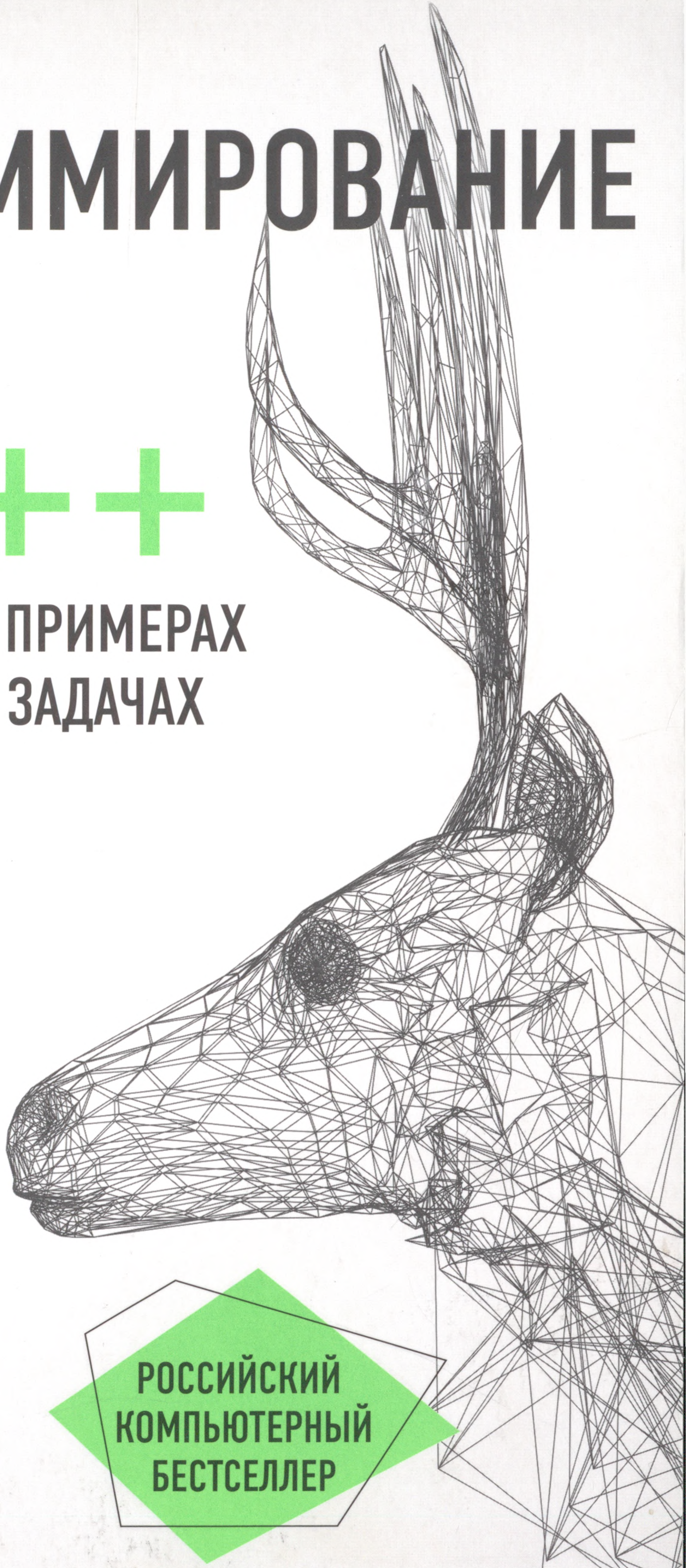
Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА C++

В ПРИМЕРАХ И ЗАДАЧАХ

- Основные сведения о синтаксисе и концепции языка
- От простых объектно-ориентированных программ до многопоточного программирования
- Примеры и задачи для самостоятельного изучения
- Для студентов и самостоятельного изучения



РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА C++ В ПРИМЕРАХ И ЗАДАЧАХ

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Москва
2018

УДК 004.43
ББК 32.973-018.1
В19

Васильев, Алексей Николаевич.

В19 Программирование на C++ в примерах и задачах / Алексей Васильев. — Москва : Эксмо, 2018. — 368 с. — (Российский компьютерный бестселлер).

ISBN 978-5-699-87445-3

Книга включает в себя полный набор сведений о языке C++, необходимых для успешного анализа и составления эффективных программных кодов. Материал излагается последовательно и дополняется большим количеством примеров, практических задач и детальным разбором их решений. К каждому разделу прилагается обширный список задач для самостоятельного решения.

УДК 004.43
ББК 32.973-018.1

ISBN 978-5-699-87445-3

ОГЛАВЛЕНИЕ

Вступление
О КНИГЕ И ЯЗЫКЕ C++

Замечательная идея! Что ж она мне самому в голову не пришла?

из к/ф «Ирония судьбы, или С легким паром»

Вниманию читателя предлагается книга о языке программирования C++. Пособие для тех, кто не знаком (или мало знаком) с языком C++ и хочет быстро научиться на нем программировать.

Собственно о книге

Издание состоит из тематически подобранных примеров и задач, которые перекрывают все основные разделы и вопросы, важные с точки зрения изучения основ языка C++. Здесь хочется отметить два важных обстоятельства.

Во-первых, опыт показывает, что наиболее успешно материал усваивается, если он иллюстрируется примерами. Более того, иногда очень сложно объяснить определенную концепцию или подход, если они не подкреплены практическим материалом. В этом смысле для книги выбран оптимальный способ представления информации.

Во-вторых, предполагается, что при работе с материалом книги читатель проявит некоторую настойчивость и готовность к углубленному анализу программных кодов. С другой стороны, автор приложил максимальные усилия для того, чтобы сделать содержимое книги простым и наглядным даже для неподготовленного читателя.

Последняя глава в определенном смысле особенная. В ней собраны некоторые математические задачи вычислительного характера. Они ориентированы на большую самостоятельность читателя в плане анализа программных кодов (по сравнению с примерами из других глав книги). Данная глава включена в книгу, поскольку представленные в ней задачи

рассматриваются в рамках университетского лекционного курса по вычислительной математике.

Для лучшего закрепления материала в конце первых четырех глав приводится список задач для самостоятельного решения. Они концептуально близки к тем задачам и примерам, что рассматривались в основной части соответствующей главы. Некоторые задачи очень просты, но встречаются и такие, для решения которых необходимо проявить смекалку. Также стоит отметить, что для части задач, предлагаемых для самостоятельного решения, в книге (обычно в следующих главах после той, где предлагается задача) приводятся решения. Возможно, контекст задачи при решении немного изменяется, но узнать знакомые «ноты» все же можно. Сделано это специально, поскольку, как показывает опыт, решить одну и ту же задачу разными методами бывает намного полезнее, чем просто решать разные задачи.

В пятой и шестой главах, где обсуждаются принципы объектно-ориентированного программирования, список задач для самостоятельного решения также есть, но он намного скромнее. Предлагаемые там задачи носят «эвристический» характер и сформулированы в общем виде. Это намеренная позиция, поскольку в данном случае важен контекст постановки задачи. Другими словами, частью задачи, которую предстоит решить читателю, является, в том числе, и ее окончательная формулировка.

В главах с пятой по восьмую выделяются некоторые «идеологические» направления на тот случай, если читатель решит самостоятельно изучать соответствующие аспекты программирования (рекомендации для самостоятельной работы). Задачи для самостоятельного решения в седьмой и восьмой главах не предлагаются, поскольку материал там сложный и некоторые темы представлены скорее в режиме ознакомления.

Наконец, в последней главе, где рассматриваются математические вычислительные задачи, нет ни заданий для самостоятельного решения, ни рекомендаций для самостоятельной работы. Причина в том, что глава основательно ориентирована на самостоятельное изучение нетривиального материала, поэтому перегружать ее нежелательно.

Язык программирования C++

C++ — один из наиболее популярных языков программирования. Невозможно представить себе профессионального программиста, который

не знал бы его. В этом смысле выбор языка C++ для изучения представляется весьма удачным.

Помимо непосредственной практической пользы от умения составлять программы на этом языке, есть еще и немаловажный методологический аспект. Связан он с исключительной гибкостью и богатством C++. После его изучения намного легче «брать на вооружение» прочие языки программирования. Но как бы там ни было, C++ на сегодня востребован, и в ближайшее время это вряд ли изменится.

Среда разработки

Нет недостатка в программном обеспечении для работы с программными кодами на C++. На данный момент существует много качественных средств разработки — как коммерческих, так и бесплатных. Обычно используют *среды разработки*, в состав которых входит редактор кодов, отладчик, обычно компилятор и ряд других вспомогательных утилит. Читатель, безусловно, волен выбрать ту среду разработки, которая ему наиболее симпатична. Рассматриваемые в основной части книги коды универсальны и не привязаны к какой-то конкретной среде. Вместе с тем важно отметить, что тестировались программы в Visual Studio Express 2013. Желающие могут загрузить (бесплатно, но с регистрацией) данную среду разработки с сайта компании Microsoft.

Об авторе

Автор книги — *Васильев Алексей Николаевич*, доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Автор книг по программированию и математическому моделированию. Сфера научных интересов: физика жидкостей, биофизика, синергетика, математические методы в экономике, моделирование социально-политических процессов, математическая лингвистика.

Обратная связь

Вы можете высказать свои замечания и предложения по поводу книги по электронной почте `alex@vasilev.kiev.ua`.

Файлы для скачивания

Дополнительные материалы можно скачать по адресу https://eksmo.ru/C++_codes.zip

Благодарности

Автор выражает искреннюю признательность своим читателям за интерес к книгам. Понимание того, что книги востребованы читателями, является наилучшим стимулом для их написания.

Глава 1
ПРОСТЫЕ ПРОГРАММЫ

Софистика, пастор, софистика!

из к/ф «Семнадцать мгновений весны»

В этой главе рассматриваются наиболее простые программы. Кроме прочего, мы узнаем:

- как организована программа в C++;
- как осуществляется ввод и вывод данных через консольное окно;
- как объявляются переменные;
- как выполняются простые арифметические операции,

ну и кое-что еще.

Первая программа

Далее рассматривается небольшая программа, которая отображает приветствие в консольном окне. Соответствующий программный код приведен в листинге 1.1.

Листинг 1.1. Первая программа

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Отображение сообщения в консоли:
   cout<<"Программируем на C++!"<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

При выполнении программы получаем такой результат:

Результат выполнения программы (из листинга 1.1)

```
Программируем на C++!
```

Соответствующее сообщение появляется в консольном окне.

Рассмотрим и проанализируем программный код. Прежде всего, выделим основные блоки в программе. Фактически он там один: это тело *главной функции* программы. Она называется `main()` и описана в соответствии с таким шаблоном:

```
int main(){
   // Код главной функции программы
}
```

Собственно тело функции выделяется с помощью пары фигурных скобок: открывающей `{` и закрывающей `}`. То, что между этими скобками — код функции `main()`. Ключевое слово `int` в описании функции означает, что функция возвращает результат, и результат функции является целым числом.

Выполнение программы в C++ — это выполнение главной функции `main()`. Проще говоря, чтобы понять, какие команды выполняются при выполнении программы, следует обратиться к функции `main()`. В данном случае в теле функции несколько (точнее, семь) строчек кода, причем три из них — *комментарии*.

Комментарии начинаются с двойной косой черты и при компиляции программы игнорируются. Назначение комментария — объяснить тому, кто анализирует программный код, назначение тех или иных команд. Комментарии предназначены для человека, а не для компьютера. Необходимости в использовании комментариев в программном коде нет, но их наличие значительно облегчает задачу программиста.

ТЕОРИЯ

В C++ используются однострочные и многострочные комментарии. Однострочный комментарий начинается с двойной косой черты //. Все, что находится справа — комментарий. Однострочный комментарий размещается в одной строке.
Многострочные комментарии, как несложно догадаться, состоят из нескольких строк. Начало выделяется комбинацией символов /*,

а конец — */. Все, что находится между инструкциями /* и */ является комментарием.

Две команды в теле функции `main()` имеют технический, «вспомогательный» характер. Их наличие в программном коде связано с особенностями функционирования консольного окна и необходимостью отображать в нем кириллический текст. Речь о командах `system("chcp 1251>nul")` и `system("pause>nul")`. Первая переводит консоль в режим использования кодировки *Windows-1251*, а вторая нужна для того, чтобы после выполнения программы консольное окно не закрывалось сразу, а задерживалось на экране до нажатия пользователем клавиши **Enter** (или какой-то другой).

ПОДРОБНОСТИ

При работе с операционной системой Windows и вводе/выводе информации через консольное окно пользователь, скорее всего, столкнется с двумя проблемами. Первая возникает при отображении кириллического текста в консольном окне.

Фундаментальная сложность связана с тем, что редактор кодов, в котором набирается код программы, имеет кодировку, отличную от кодировки консольного кона. Поэтому кириллический текст, набранный в редакторе кодов, будет нормально выглядеть в окне редактора кодов, но совершенно неприлично при отображении его в консоли. Та же проблема с вводом текста: если в консоли вводится кириллический текст, то программой он будет считан как набор странных символов.

Проблему можно решить, установив в окне консоли такую же кодировку, как в окне редактора кодов. Сделать это можно вручную или непосредственно через программный код. Для этого в начале программного кода помещается команда `system("chcp 1251>nul")`.

Еще одна проблема связана с тем, что после завершения выполнения программы консольное окно, куда осуществляется вывод, автоматически закрывается. Происходит все очень быстро — настолько, что пользователь не успевает заметить, какая информация выводилась в консольное окно при выполнении программы. Чтобы консоль не закрывалась по завершении выполнения программы, добавляем в программный код команду `system("pause>nul")`.

Команды `system("chcp 1251>nul")` и `system("pause>nul")` представляют собой вызов функции `system()`. Чтобы функция `system()` была доступна, подключаем заголовок `<cstdlib>`.

Аргументом функции `system()` передается текстовая строка, содержащая команду, которая должна быть выполнена в консоли.

В частности, команда `pause` для консоли означает задержку консольного окна, а команда `chcp 1251` используется для применения соответствующей кодировки. В принципе, в программном коде можно использовать команды `system("chcp 1251")` и `system("pause")`, однако в этом случае в консольное окно выводятся сообщения соответственно об установке кодировки и о необходимости нажать кнопку для закрытия консольного кона. Чтобы этого не было, используем инструкцию `nul`. Она указывается через символ > после собственно команды, которую необходимо выполнить в консоли.

Пожалуй, самая важная команда в теле функции `main()` — это `cout<<"Программируем на C++!"<<endl`. Именно она решает основную задачу по отображению сообщения в консольном окне, ради чего, собственно, и создавалась программа. Основу команды составляет *оператор вывода* `<<`. Оператором вывода данные, указанные справа, отображаются в устройстве, ссылка на которое указана слева от оператора. Например, команда `cout<<"Программируем на C++!"` означает, что необходимо текст `"Программируем на C++!"` вывести в устройство, «спрятанное» за идентификатором `cout`.

По умолчанию идентификатор `cout` означает консоль (*cout* от *console output*). Текст, как несложно догадаться, заключается в двойные кавычки. Операторов вывода в одной команде разрешается использовать несколько. Инструкция `endl` (от *end of line*) используется для перевода курсора в новую строку в окне вывода (консоли). Поэтому вся команда `cout<<"Программируем на C++!"<<endl` означает следующее: в консольное окно необходимо вывести текст `"Программируем на C++!"`, после чего выполняется переход к новой строке.

НА ЗАМЕТКУ

В принципе, нет крайней необходимости в использовании инструкции `endl` для перехода к новой строке. Вместо команды `cout<<"Программируем на C++!"<<endl` вполне сгодилась бы и команда `cout<<"Программируем на C++!"`.

Последняя в теле функции `main()` команда `return 0` означает завершение выполнения функции и возвращение результатом функции значения `0`. Значение `0` дает знать операционной системе, что работа функции завершилась в нормальном режиме, без ошибки. Последней командой в функции `main()` обычно является команда `return 0`.

НА ЗАМЕТКУ

Также хочется отметить, что команды в C++ заканчиваются точкой с запятой.

Мы разобрали код главной функции, но есть еще несколько инструкций в самом начале программы, до начала описания главной функции.

Инструкции `#include <iostream>` и `#include <cstdlib>` представляют собой директивы препроцессора и используются для включения в программный код заголовочных файлов, которые содержат важную информацию, необходимую для выполнения программы.

Фактически, речь идет об использовании базовых стандартных библиотек, необходимых для выполнения программы (`<iostream>` соответствует стандартной библиотеке ввода/вывода, а `<cstdlib>` соответствует стандартной общей библиотеке).

ПОДРОБНОСТИ

Стандартная библиотека ввода/вывода содержит определение таких идентификаторов, как `cout` (консольное устройство) и `cin` (клавиатура — устройство ввода). Стандартная общая библиотека содержит определение, кроме прочего, функции `system()`.

Инструкция `using namespace std` представляет собой команду для использования стандартного *пространства имен* `std`.

ТЕОРИЯ

Пространство имен — это некое абстрактное хранилище, позволяющее избегать конфликта имен. При составлении программного кода нам необходимо указать, какое именно пространство имен будет использовано для «хранения» названий создаваемых в программе утилит. Мы будем использовать стандартное пространство имен, которое называется `std`.

Собственно на этом мы заканчиваем анализ программного кода. В завершение примера отметим, что при написании программ мы в дальнейшем планируем использовать следующий шаблонный код (без учета комментариев):

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   system("chcp 1251>nul");
   // Код главной функции
   system("pause>nul");
   return 0;
}
```

Если же читатель решит проблему с кодировкой и задержкой консольного окна способом, отличным от представленного выше, то шаблонный код еще проще:

```
#include <iostream>
using namespace std;
int main(){
   // Код главной функции
   return 0;
}
```

Теперь переходим к рассмотрению следующего примера. В нем мы познакомимся с *оператором ввода* (узнаем, как вводится информация с клавиатуры) и *переменными*.

Знакомство с переменными

Далее мы рассмотрим задачу о переводе расстояния, указанного в милях, в значение, указанное в километрах. При ее решении нам предстоит вводить данные с клавиатуры и передавать их в программу для выполнения необходимых вычислений.

> **НА ЗАМЕТКУ**
>
> Британская статутная миля — мера длины, равная 1609344 миллиметрам или 1,609344 километра. Чтобы перевести значение, указанное в милях, в километры, достаточно умножить значение в милях на 1,609344 .

Программа достаточно простая: сначала выводится запрос на ввод пользователем расстояния в милях, после чего введенное значение считывается, выполняются необходимые вычисления, а результат вычислений отображается в консольном окне.

В процессе вычислений нам понадобится куда-то записывать вводимое пользователем значение, равно как и результаты вычислений. Для этого в программе используются *переменные*.

ТЕОРИЯ

Переменная — именованная область памяти, в которую можно записать значение и из которой можно прочитать значение. В C++ переменные перед использованием объявляются. Для каждой переменной указывается тип значения, которое может храниться в переменной. Тип переменной определяется через ключевое слово — идентификатор типа. Наиболее актуальные базовые типы: `int` (целые числа), `double` (действительные числа), `char` (символы).
В программе переменная объявляется в любом месте (но до первого ее использования). Область доступности переменной определяется блоком, в котором она объявлена. Блок, в свою очередь, ограничивается парой фигурных скобок. Если переменная объявлена в главной функции программы, она доступна в любом месте главной функции.
Помимо переменных, нередко используются **константы**. От переменных они отличаются тем, что значение константы изменить нельзя. Константы описываются так же, как переменные, но с идентификатором `const`.

Теперь рассмотрим программный код, представленный в листинге 1.2.

Листинг 1.2. Перевод миль в километры

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Константа определяет, сколько в одной
    // миле километров:
    const double kmInMile=1.609344;
```

```
    // Переменные для записи расстояния
    // в милях и километрах:
    double distMile,distKm;
    // Запрос на ввод расстояния в милях:
    cout<<"Укажите расстояние в милях: ";
    // Считывание значения для расстояния в милях:
    cin>>distMile;
    // Вычисление расстояния в километрах:
    distKm=distMile*kmInMile;
    // Отображение результата вычислений:
    cout<<"Расстояние (в км): "<<distKm<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В теле главной функции программы инструкцией `const double kmInMile=1.609344` объявляется константа действительного числового типа. Константа называется `kmInMile` и ее значение равно `1.609344` (километров в одной миле). Также в программе объявляются две переменные `distMile` и `distKm`, обе типа `double`. В переменную `distMile` записывается значение для расстояния в милях, которое пользователь вводит с клавиатуры. Происходит это так: сначала командой `cout<<"Укажите расстояние в милях: "` в консоли отображается сообщение с предложением ввести числовое значение. Затем пользователь с клавиатуры вводит значение и нажимает клавишу <Enter>. Такая ситуация обрабатывается командой `cin>>distMile`, основу которой составляет *оператор ввода* >>. Идентификатор `cin` (аббревиатура от *console input*) обозначает устройство ввода (по умолчанию речь идет о клавиатуре). Справа от оператора вывода указывается переменная (в данном случае `distMile`), в которую записывается значение, вводимое пользователем.

> **НА ЗАМЕТКУ**
>
> Поскольку переменная `distMile` объявлена с типом `double`, то введенное пользователем значение автоматически приводится к типу `double`.

Командой `distKm=distMile*kmInMile` вычисляется расстояние в километрах (расстояние в милях `distMile` умножается на константу `kmInMile`, определяющую количество километров в одной миле).

ТЕОРИЯ

Здесь мы сталкиваемся с оператором умножения `*`. Другие арифметические операторы: оператор сложения `+`, оператор вычитания `-`, оператор деления `/`. Оператор деления в C++ специфический: если оба операнда целочисленные, то деление выполняется нацело (целочисленное деление). Например, результатом выражения `9/4` является значение `2`. Чтобы деление выполнялось на множестве действительных чисел, перед соответствующей командой указывается инструкция `(double)` (ключевое слово `double` в круглых скобках). Например, результатом выражения `(double)9/4` является значение `2.25`. Присваивание выполняется с помощью оператора `=`. Значение, указанное справа от оператора присваивания, записывается в переменную, указанную слева от оператора присваивания.

Результат вычислений записывается в переменную `distKm`. После этого команда `cout<<"Расстояние (в км): "<<distKm<<endl` отображает вычисленное значение в консольном окне. Результат выполнения программы может быть таким (жирным шрифтом выделено значение, вводимое пользователем):

Результат выполнения программы (из листинга 1.2)

```
Укажите расстояние в милях: 3.5
Расстояние (в км): 5.6327
```

Выше мы оперировали нецелыми (действительными) числовыми значениями. Такой подход математически корректен, но с прикладной точки зрения не очень удобен, поскольку в реальности для определения расстояний используются разномасштабные метрические единицы. Подобную ситуацию рассмотрим далее.

НА ЗАМЕТКУ

Поясним, что имеется в виду. Скажем, в предыдущем примере мы при тестировании программы вводили значение `3.5` для расстояния в милях. Если это же расстояние выразить в милях и футах, то будет `3` мили и `2640` футов (в одной миле ровно `5280` футов). То же расстояние в километрах составляет величину `5.6327`, или, что то же самое, `5` километров и `632` метра (если отбросить `70` сантиметров).

Рассмотрим пример, аналогичный представленному выше, но несколько видоизменим общую постановку задачи. В частности, расстояние будет указываться в милях и футах, а пересчитываться — в километры и метры. Соответствующий программный код представлен в листинге 1.3.

Листинг 1.3. Перевод миль и футов в километры и метры

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Количество футов в миле:
   const int ftInMile=5280;
   // Количество километров в миле:
   const double kmInMile=1.609344;
   // Целочисленные переменные для записи количества
   // миль, футов, километров и метров:
   int dMile,dFt,dKm,dM;
   // Ввод расстояния в милях и футах:
   cout<<"Расстояние в милях и футах."<<endl;
   cout<<"Мили: ";
   cin>>dMile;
   cout<<"Футы: ";
   cin>>dFt;
   // Расстояние в милях:
   double distMile=dMile+(double)dFt/ftInMile;
   // Расстояние в километрах:
   double distKm=distMile*kmInMile;
   // Только километры:
   dKm=(int)distKm;
   // Метры:
   dM=(int)((distKm-dKm)*1000);
```

```
// Отображение результата вычислений:
cout<<"Расстояние в километрах и метрах."<<endl;
cout<<"Километры: "<<dKm<<endl;
cout<<"Метры: "<<dM<<endl;
// Задержка консольного окна:
system("pause>nul");
return 0;
```

Проанализируем программу. В данном случае определяется целочисленная константа `ftInMile` со значением `5280`, определяющим количество футов в миле. Действительная числовая константа `kmInMile` со значением `1.609344`, определяющим количество километров в миле. Целочисленные переменные `dMile`, `dFt`, `dKm` и `dM` используются нами для считывания количества миль, футов и записи вычисленных значений для количества километров и метров. Считывание миль и футов выполняется с помощью команд `cin>>dMile` и `cin>>dFt` соответственно. Далее в программном коде с помощью команды `double distMile=dMile+(double)dFt/ftInMile` вычисляется расстояние в милях (имеется в виду расстояние, выраженное только в милях, и поэтому такое значение в общем случае является действительным числом). Данная команда показательна во многих отношениях. Во-первых, она не только вычисляет значение, но одновременно и объявляет переменную, в которую значение записывается. Более конкретно, объявляется переменная `distMile` типа `double` и ей присваивается значение выражения `dMile+(double)dFt/ftInMile`. Данное выражение представляет собой сумму значения переменной `dMile` и выражения `(double)dFt/ftInMile`, которое, в свою очередь, является частным от деления значения переменной `dFt` на значение переменной `ftInMile`. Поскольку обе переменные целочисленные, а деление нужно выполнять на множестве действительных чисел (то есть обычное деление, не целочисленное), то перед выражением указывается инструкция `(double)`.

НА ЗАМЕТКУ

Отношение значений переменных `dFt` и `ftInMile` представляет собой часть мили, которая в исходном представлении выражена в футах.

Расстояние в километрах вычисляется командой `double distKm=distMile*kmInMile`. Здесь мы сталкиваемся с такой же ситуацией, как и выше:

переменная distKm объявляется, и ей сразу присваивается значение, причем значение вычисляется (как произведение расстояния в милях distMile на количество километров в одной миле kmInMile).

НА ЗАМЕТКУ

Ситуация, когда значение переменной присваивается при ее объявлении и присваиваемое значение определяется выражением, называется **динамической инициализацией переменной**.

В переменную distKm записано расстояние в километрах. Это действительное число. Нам необходимо извлечь из него целую часть (километры) и дробную часть, которая, умноженная на 1000 (в одном километре 1000 метров), дает значение для количества метров.

Чтобы выделить из значения переменной distKm целую часть, используем команду dKm=(int)distKm. В ней использовано явное приведение типа: за счет инструкции (int) выполняется приведение double-значения переменной distKm к целочисленному значению путем отбрасывания дробной части. Проще говоря, значением выражения (int)distKm возвращается целая часть значения переменной distKm, при этом значение самой переменной distKm не меняется.

Переменной dM присваивается значением выражение (int)((distKm-dKm)*1000). Оно вычисляется так: разность значений переменных distKm-dKm дает результатом дробную часть переменной distKm. Полученное значение умножается на 1000, и у вычисленного результата отбрасывается дробная часть (за счет инструкции (int)).

Полученные в результате вычислений значения для переменных dKm и dM отображаются в консольном окне командами cout<<"Километры: "<<dKm<<endl и cout<<"Метры: "<<dM<<endl. Ниже показано, как может выглядеть результат выполнения программы (жирным шрифтом выделены значения, которые вводит пользователь):

Результат выполнения программы (из листинга 1.3)

```
Расстояние в милях и футах.
Мили: 3
Футы: 2640
Расстояние в километрах и метрах.
```

```
Километры: 5
Метры: 632
```

Рассмотренные примеры дают некоторое представление о том, как используются переменные. Далее мы познакомимся с *функциями*.

Знакомство с функциями

Теперь мы рассмотрим ту же задачу о вычислении расстояния в километрах по значению, указанному в милях, но на этот раз прибегнем к помощи функций.

ТЕОРИЯ

Функция — именованный блок программного кода, который можно вызвать по имени. При описании функции указывается ее прототип (тип результата, название и список аргументов) и, в фигурных скобках, тело функции (программный код, который выполняется при вызове функции).

Новый способ решения уже знакомой нам задачи представлен в программном коде в листинге 1.4.

Листинг 1.4. Использование функций

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция для считывания расстояния в милях:
double getDistMile(){
   // Переменная для записи результата функции:
   double dist;
   // Запрос на ввод расстояния в милях:
   cout<<"Укажите расстояние в милях: ";
   // Считывание значения для расстояния в милях:
   cin>>dist;
   // Результат функции:
   return dist;
```

```
}
// Функция для перевода милей в километры:
double getDistKm(double dist){
   // В одной миле километров:
   double kmInMile=1.609344;
   // Результат функции:
   return dist*kmInMile;
}
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Переменная для записи расстояния в милях:
   double distMile=getDistMile();
   // Выполнение вычислений:
   cout<<"Расстояние (в км): "<<getDistKm(distMile)<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы может быть таким (жирным шрифтом выделен ввод пользователя):

Результат выполнения программы (из листинга 1.4)

```
Укажите расстояние в милях: 3.5

Расстояние (в км): 5.6327
```

Фактически, с точки зрения конечного результата программа выполняет те же операции, что и в одном из предыдущих примеров (см. листинг 1.2). Вместе с тем в данном случае код принципиально иной. Проанализируем его.

В программе кроме главной функции `main()`, описывается еще две функции: функция `getDistMile()`, предназначенная для считывания с клавиатуры значения для расстояния милях, и функция `getDistKm()`,

предназначенная для преобразования миль в километры. Объявление функции `getDistMile()` начинается с ключевого слова `double`: функция возвращает результат, и это значение типа `double`. После имени функции круглые скобки пустые — аргументов у функции нет. В теле функции командой `double dist` объявляется локальная (доступна только в теле функции) переменная, значение которой считывается с клавиатуры (команда `cin>>dist`). После этого значение переменной `dist` возвращается результатом функции (команда `return dist`).

ТЕОРИЯ

Напомним, что область доступности переменной ограничивается блоком, в котором она объявлена. Переменные, объявленные в теле функции, называются локальными и доступны только в теле функции. Место в памяти под локальные переменные выделяется при вызове функции, а по завершении выполнения функции локальные переменные удаляются из памяти. Локальная переменная, таким образом, существует, только пока выполняется функция.

Следует различать описание функции и ее вызов. Описание функции не подразумевает выполнение кода функции. Код функции выполняется при вызове функции.

Инструкция `return` приводит к завершению выполнения функции. Если после инструкции `return` указано значение, то оно возвращается результатом функции.

Функция `getDistKm()` предназначена для перевода миль в километры: у нее один аргумент (обозначен как `dist`) типа `double`, а результатом функция возвращает также значение типа `double`. Результат функции возвращается командой `return dist*kmInMile`. Это произведение аргумента `dist` и переменной `kmInMile` (значение пеерменной равно `1.609344`).

ТЕОРИЯ

Аргументы функции «имеют силу» локальной переменной. Объявление аргумента при описании функции распространяется только на тело функции.

В главной функции программы командой `double distMile=getDistMile()` объявляется переменная `distMile` и значением этой переменной присваивается результат функции `getDistMile()`. Вызов функции `getDistMile()`

приводит к отображению запроса в консольном окне и заканчивается считыванием значения с клавиатуры (и возврата этого значения функцией). Записанное в переменную `distMile` значение передается аргументом при вызове функции `getDistKm()` (имеется в виду команда `cout<<"Расстояние (в км): "<<getDistKm(distMile)<<endl`). В итоге получаем в консольном окне сообщение, содержащее значение для расстояния в километрах.

Знакомство с оператором цикла

В следующем примере вычисляется сумма квадратов натуральных чисел.

> **НА ЗАМЕТКУ**
>
> Речь идет о вычислении суммы $1^2 + 2^2 + 3^2 + \ldots + n^2$, где верхняя граница суммы n задана. В программе такая сумма вычисляется напрямую, непосредственным суммированием. Для проверки корректности результата можно воспользоваться формулой $1^2 + 2^2 + 3^2 + \ldots + n^2 = (n(n + 1)(2n + 1))/6$.

Для проведения вычислений используется оператор цикла `while`. Пример представлен в листинге 1.5.

Листинг 1.5. Вычисление суммы квадратов натуральных чисел

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Верхняя граница суммы, значение суммы и
   // индексная переменная:
   int n=10,s=0,k=1;
   // Оператор цикла для вычисления суммы:
   while(k<=n){
      // Добавление нового слагаемого к сумме:
```

```
    s=s+k*k;
    // Увеличение (на 1) значения индексной переменной:
    k++;
}
// Отображение результата вычислений:
cout<<"Сумма квадратов от 1 до "<<n<<": "<<s<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже приведен результат выполнения программы:

Результат выполнения программы (из листинга 1.5)

```
Сумма квадратов от 1 до 10: 385
```

В программе командой `int n=10,s=0,k=1` объявляются три целочисленные переменные, причем им сразу присваивается значение. Переменная `n` определяет верхнюю границу для суммы, в переменную `s` будет записываться сумма квадратов натуральных чисел, а переменная `k` понадобится в операторе цикла для «загибания пальцев» — подсчета количества слагаемых, добавленных в сумму (переменную `k` будем называть индексной переменной). Все основные вычисления выполняются в операторе цикла, описание которого начинается с ключевого слова `while`. В круглых скобках после ключевого слова `while` указано условие `k<=n`, означающее, что оператор цикла выполняется до тех пор, пока значение переменной `k` не превышает значение переменной `n`. За каждый цикл выполняется две команды: команда `s=s+k*k` добавляет к значению переменной `s` квадрат текущего значения переменной `k`, после чего команда `k++` увеличивает значение переменной `k` на единицу. Все это продолжается до тех пор, пока значение переменной `k` не станет больше значения переменной `n`.

ТЕОРИЯ

В выражении `k<=n` использован оператор сравнения `<=` (меньше или равно), а результатом выражения является значение **логического типа**. У логического типа всего два значения: `true` (истина) и `false` (ложь).

Оператор цикла `while` выполняется до тех пор, пока выражение, указанное в круглых скобках после ключевого слова `while`, имеет значение `true`. *Условие проверяется каждый раз перед началом выполнения* команд в теле оператора цикла.

Унарный оператор **инкремента** `++` позволяет увеличить на единицу значение своего оператора. Команда вида `k++` эквивалентна команде `k=k+1`.

Унарный оператор **декремента** `--` уменьшает на единицу значение своего операнда. Команда `k--` эквивалентна команде `k=k-1`.

Командой `cout<<"Сумма квадратов от 1 до "<<n<<": "<<s<<endl` результат вычислений отображается в консольном окне. Здесь мы используем значение переменной `n` для верхней границы суммы и вычисленное значение `s` для значения суммы квадратов натуральных чисел.

Несколько видоизмененная версия рассмотренного выше программного кода представлена в листинге 1.6.

Листинг 1.6. Альтернативный способ вычисления суммы квадратов

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Верхняя граница суммы и значение суммы:
   int n,s=0;
   // Ввод значения для верхней границы ряда:
   cout<<"Верхняя граница суммы: ";
   cin>>n;
   // Оператор цикла для вычисления суммы:
   while(n){
      // Добавление нового слагаемого к сумме:
      s+=n*n;
      // Уменьшение (на 1) значения переменной:
      n--;
   }
```

```
    // Отображение результата вычислений:
    cout<<"Сумма квадратов: "<<s<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В данном случае несколько изменений. Во-первых, мы воспользовались тем, что в качестве логических значений разрешается указывать числовые: отличное от нуля число интерпретируется как значение `true`, а нулевое числовое значение интерпретируется как `false`. Это позволило нам обойтись без индексной переменной `k`. Во-вторых, мы воспользовались сокращенной формой оператора присваивания.

ТЕОРИЯ

В C++ существуют **сокращенные формы оператора присваивания**. Например, вместо команды вида `x=x+y` можно использовать команду `x+=y`. Это же замечание относится и к прочим арифметическим операторам: вместо команды `x=x-y` допустимо использовать команду `x-=y`, вместо команды `x=x*y` используют команду `x*=y`, и так далее.

Значение переменной `n`, определяющей верхнюю границу суммы, теперь считывается с клавиатуры. Эта же переменная указана условием в операторе цикла `while`. Оператор цикла выполняется, пока значение переменной `n` отлично от нуля. В теле оператора цикла последовательно выполняются команды `s+=n*n` и `n--`. Первой из них к текущему значению переменной `s` прибавляется квадрат текущего значения переменной `n`, после чего командой `n--` значение переменной `n` уменьшается на единицу. Весь процесс продолжается до тех пор, пока значение переменной `n` не станет равным нулю.

НА ЗАМЕТКУ

Таким образом, в программе вычисляется сумма $n^2 + (n - 1)^2 + ... + 2^2 + 1^2$, которая, очевидно, совпадает с суммой $1^2 + 2^2 + ... + n^2$.

Ниже показано, как может выглядеть результат выполнения программы (жирным шрифтом выделен ввод пользователя):

Результат выполнения программы (из листинга 1.6)

```
Верхняя граница суммы: 10
Сумма квадратов: 385
```

У рассмотренного программного кода есть один небольшой недостаток: если пользователь введет отрицательное значение для переменной `n`, получим бесконечный цикл, поскольку при уменьшении отрицательного значения нулем оно не станет. В следующем примере мы «заблокируем» данную неприятность.

Знакомство с условным оператором

В новой версии предыдущей программы добавлена проверка значения переменной `n` после того, как оно считано с клавиатуры. Для этого использован *условный оператор* `if`.

ТЕОРИЯ

Условный оператор описывается так: после ключевого слова `if` в круглых скобках указывается условие. Далее следует блок команд в фигурных скобках, которые выполняются при истинности условия. Если условие ложно, выполняется блок команд в фигурных скобках после ключевого слова `else`.

Рассмотрим программный код примера в листинге 1.7.

Листинг 1.7. Проверка корректности введенного значения

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Верхняя граница суммы и значение суммы:
```

```
int n,s=0;
// Ввод значения для верхней границы ряда:
cout<<"Верхняя граница суммы: ";
cin>>n;
// Если введено положительное число:
if(n>0){
   // Оператор цикла для вычисления суммы:
   while(n){
      // Добавление нового слагаемого к сумме:
      s+=n*n;
      // Уменьшение значения переменной:
      n--;
   }
   // Отображение результата вычислений:
   cout<<"Сумма квадратов: "<<s<<endl;
}
   // Если введено отрицательное число или ноль:
   else{
      cout<<"Указано некорректное значение"<<endl;
   }
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

В условном операторе проверяется условие `n>0`, и если оно истинно, выполняется блок команд, которыми вычисляется сумма квадратов натуральных чисел. Если же условие `n>0` ложно, то выполняются команды в `else`-блоке. В данном случае там всего одна команда `cout<<"Указано некорректное значение"<<endl`, которой в консольное окно выводится соответствующее сообщение. Ниже показано, как может выглядеть результат выполнения программы, если пользователь вводит корректное значение для верхней границы суммы (жирным шрифтом выделен ввод пользователя):

Результат выполнения программы (из листинга 1.7)

```
Верхняя граница суммы: 10
Сумма квадратов: 385
```

Если пользователем вводится некорректное значение, результат может быть следующим (жирным шрифтом выделено веденное пользователем значение):

Результат выполнения программы (из листинга 1.7)

```
Верхняя граница суммы: -5
Указано некорректное значение
```

Еще одна полезная конструкция, с которой часто приходится иметь дело — *массивы*. Следующий пример дает некоторое представление о том, как массивы используются в программных кодах.

Знакомство с массивами

Массив (одномерный) представляет собой набор элементов, объединенных в одну общую «конструкцию». Каждый элемент массива, по сути, является отдельной переменной. Но доступ к таким переменным получают через имя массива. Поскольку в массиве элементов много, то для однозначной идентификации элемента в массиве указывается *индекс элемента*.

ТЕОРИЯ

Массив объявляется достаточно просто. Сначала указывается идентификатор типа, к которому относятся элементы массива (все элементы массива должны быть одного типа). Затем указывается имя массива и в квадратных скобках — размер массива (количество элементов в массиве). При обращении к элементу массива указывается имя массива и в квадратных скобках индекс элемента. Индексация элементов начинается с нуля, поэтому первый элемент массива имеет индекс `0`, а индекс последнего элемента массива на единицу меньше размера массива. Размер массива задается константой (это не может быть обычная переменная).

Рассмотрим пример, в котором создается числовой массив, который заполняется биномиальными коэффициентами.

НА ЗАМЕТКУ

По определению биномиальные коэффициенты вычисляются как $C_n^k = n!/(k!(n-k)!)$, где использовано обозначение для факториала числа $m! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot m$ (произведение натуральных чисел от 1 до m). Несложно заметить, что $C_n^k = C_n^{n-k}$, а также $C_n^0 = 1$, $C_n^1 = 1$, $C_n^2 = (n(n-1))/2$ и так далее. Мы в программе воспользуемся рекуррентным соотношением $C_n^{k+1} = C_n^k \cdot (n-k)/(k+1)$.

Обратимся к программному коду, представленному в листинге 1.7.

Листинг 1.8. Биномиальные коэффициенты

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Константа для определения размера массива:
   const int n=10;
   // Создание целочисленного массива:
   int bnm[n+1];
   // Индексная переменная:
   int k=0;
   // Первый элемент массива:
   bnm[0]=1;
   // Отображение первого элемента массива:
   cout<<bnm[0];
   // Оператор цикла для заполнения массива:
   while(k<n){
      // Значение элемента массива:
      bnm[k+1]=bnm[k]*(n-k)/(k+1);
     // Отображение значения элемента массива:
     cout<<" "<<bnm[k+1];
```

```
    // Изменение значения индексной переменной:
    k++;
  }
  cout<<endl;
  // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 1.8)

```
1 10 45 120 210 252 210 120 45 10 1
```

Проанализируем программный код. Сначала командой `const int n=10` объявляется целочисленная константа `n` со значением `10`. Затем создается массив `bnm`, для чего использована команда `int bnm[n+1]`. При создании массива под него выделяется место в памяти, но значениями элементы массива не заполняются. Размер созданного массива на единицу больше значения константы `n`, поэтому индексы элементов массива находятся в диапазоне значений от `0` до `n` включительно. Для перебора элементов массива в операторе цикла объявляется целочисленная индексная переменная `k` с начальным значением `0`. Также командой `bnm[0]=1` явным образом присваивается значение первому элементу массива. Оно сразу отображается в консольном окне (команда `cout<<bnm[0]`). Для заполнения прочих элементов массива запускается оператор цикла `while`. Проверяемым условием является `k<n`, поэтому, с учетом команды `k++` в теле оператора цикла, переменная `k` последовательно пробегает значения от `0` до `n-1` включительно. Значение очередного элемента вычисляется командой `bnm[k+1]=bnm[k]*(n-k)/(k+1)`. Значение нового вычисленного таким образом элемента отображается командой `cout<<" "<<bnm[k+1]` в консольном окне. Таким образом, биномиальные коэффициенты (значения элементов массива `bnm`) отображаются в одной строке, и в качестве разделителя между элементами используется пробел.

Задачи для самостоятельного решения

1. Напишите программу, которой при выполнении в консольное окно выводятся натуральные числа от 1 до 10.

2. Напишите программу, которой отображаются пять первых нечетных чисел.

3. Напишите программу для отображения в консоли чисел, которые при делении на 4 в остатке дают 3 (такие числа описываются формулой $4k + 3$, где число k принимает значения 0, 1, 2 и так далее). Можно воспользоваться тем, что остаток от целочисленного деления вычисляется оператором %. Количество чисел вводится пользователем с клавиатуры.

4. Напишите программу для отображения в консоли чисел Фибоначчи (первые два числа в последовательности Фибоначчи равны единице, а каждое следующее равняется сумме двух предыдущих). Количество выводимых чисел вводится с клавиатуры.

5. Напишите программу для отображения (без предварительной записи в массив) биномиальных коэффициентов C_n^k (параметр $k = 0, 1, 2, ..., n$). Параметр n для нижнего индекса коэффициентов вводится с клавиатуры. Предусмотреть проверку введенного пользователем значения на корректность.

6. Напишите программу для перевода километров в мили.

7. Напишите программу для перевода расстояния, указанного в километрах и метрах, в мили и футы.

8. Напишите программу для перевода расстояния, указанного *саженях* (1 сажень равняется 2,16 метра), в метры.

9. Напишите программу для перевода расстояния, указанного в саженях и аршинах (1 сажень равняется 2,16 метра, а 3 аршина равны 1 сажени), в метры и сантиметры (в 1 метре 100 сантиметров).

10. Напишите программу с функцией для перевода скорости, указанной в км/ч («километры в час») в м/с («метры в секунду»). Учесть, что в 1 километре — 1000 метров, в 1 минуте — 60 секунд, в 1 часе — 60 минут.

11. Напишите программу с функцией для перевода скорости, указанной в м/с («метры в секунду»), в км/ч («километры в час»).

12. Напишите программу с функцией для вычисления суммы натуральных чисел.

13. Напишите программу с функцией для вычисления суммы нечетных натуральных чисел.

14. Напишите программу, в которой создается массив и заполняется четными натуральными числами.

15. Напишите программу, в которой создается массив и заполняется нечетными натуральными числами.

16. Напишите программу, в которой создается массив и заполняется квадратами натуральных чисел (значения 1^2, 2^2, 3^2 и так далее).

17. Напишите программу, в которой создается массив и заполняется степенями двойки (значения 2^0, 2^1, 2^2, 2^3 и так далее).

18. Напишите программу, в которой создается массив, который заполняется числами Фибоначчи (первые два числа равны единице, а каждое следующее равно сумме двух предыдущих).

19. Напишите программу, в которой создается массив. Массив заполняется так: элементы с четными индексами значением получают индекс, а элементы с нечетными индексами значением получают квадрат индекса. Другими словами, массив заполняется числами: 0, 1^2, 2, 3^2, 4 , 5^2, 6, 7^2 и так далее.

20. Напишите программу, в которой создается массив. Массив заполняется поэлементно: пользователь для каждого элемента вводит значение с клавиатуры (организовать ввод значений для элементов масива с привлечением оператора цикла).

Глава 2
УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ

Ну зачем такие сложности?!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В этой главе рассматриваются примеры, в которых ключевое место занимают управляющие инструкции: операторы цикла, оператор выбора и условный оператор. В предыдущей главе мы познакомились с оператором цикла `while`. Далее, в частности, мы познакомимся:

- с оператором цикла `for`;
- с оператором цикла `do-while`;
- рассмотрим особенности условного оператора `if`;
- рассмотрим оператор выбора `switch`,

а также расширим познания в области некоторых других программных конструкций, операторов и приемов программирования.

Оператор цикла `for`

В предыдущей главе мы рассматривали пример, в котором вычислялась сумма квадратов натуральных чисел. Там мы использовали оператор цикла `while`. Теперь решим аналогичную задачу, но с использованием оператора цикла `for`.

ТЕОРИЯ

Оператор цикла `for` описывается по следующему шаблону: после ключевого слова `for` в круглых скобках указывается три блока инструкций (блоки разделяются точкой с запятой). В фигурных скобках размещаются команды (тело оператора цикла), выполняемые в рамках оператора цикла.
Оператор цикла `for` выполняется по следующей схеме:

- инструкции в первом блоке выполняется один раз в начале выполнения цикла;
- проверяется условие во втором блоке инструкций — для продолжения процесса оно должно быть истинным;
- выполняются команды в теле оператора цикла (команды в фигурных скобках);
- выполняются команды в третьем боке инструкций;
- проверяется условие во втором блоке инструкций, и если оно истинно — выполняются команды в теле оператора цикла, и так далее;
- если условие во втором блоке ложно, выполнение оператора цикла завершается.

Рассмотрим программный код в листинге 2.1.

Листинг 2.1. Вычисление суммы квадратов с помощью оператора цикла for

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
// Изменение кодировки консоли:
system("chcp 1251>nul");
// Переменная для записи верхней границы суммы,
// переменная для записи значения суммы и индексная
// переменная для оператора цикла:
int n,s=0,k;
cout<<"Введите верхнюю границу для суммы: ";
// Считывание верхней границы для суммы:
cin>>n;
// Оператор цикла для вычисления суммы квадратов чисел:
for(k=1;k<=n;k++){
   // Прибавление к сумме очередного слагаемого:
   s+=k*k;
}
// Отображение результата вычислений:
cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl;
// Задержка консольного окна:
```

```
    system("pause>nul");
    return 0;
}
```

Ниже представлен возможный результат выполнения программы (жирным шрифтом выделено введенное пользователем значение):

Результат выполнения программы (из листинга 2.1)

```
Введите верхнюю границу для суммы: 10
Сумма квадратов чисел от 1 до 10: 385
```

В программе объявляется три целочисленные переменные: в переменную n будет записываться значение для верхней границы суммы, переменная s с начальным нулевым значением нужна для записи значения суммы квадратов натуральных чисел, а индексная переменная k используется в операторе цикла for для подсчета количества слагаемых, прибавляемых к сумме.

После считывания значения для переменной n запускается оператор цикла for. Первый блок в круглых скобках после ключевого слова for состоит из одной команды k=1, которая выполняется один раз в начале выполнения оператора цикла. После присваивания переменной k единичного значения, проверяется условие k<=n. Если условие истинно, то выполняется команда s+=k*k в теле оператора цикла, после чего выполняется команда k++ и снова проверяется условие k<=n. Если условие остается истинным, выполняются команды s+=k*k и k++, после чего снова проверяется условие k<=n, и так далее. Процесс продолжается до тех пор, пока при проверке выражения k<=n не будет получено значение false.

После завершения выполнения оператора цикла в переменную s записана сумма квадратов натуральных чисел от 1 до n. Результат вычислений отображается командой cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl.

Стоит заметить, что программный код «отлавливает» ситуацию, когда пользователь вводит некорректное (отрицательное) значение для переменной n. В таком случае для суммы вычисляется нулевое значение:

Результат выполнения программы (из листинга 2.1)

```
Введите верхнюю границу для суммы: -5
Сумма квадратов чисел от 1 до -5: 0
```

Причина в том, что при отрицательном значении переменной n при первой проверке условия k<=n, которая имеет место сразу после выполнения команды k=1 в первом блоке инструкций, условие оказывается ложным и выполнение оператора цикла на этом завершается. Программой отображается значение переменной s, которое в таком случае равно начальному нулевому значению данной переменной.

Блоки инструкций в операторе цикла for могут быть пустыми или напротив, содержать по несколько команд. Если блок инструкций содержит несколько команд, они разделяются запятыми. Как примеры разных вариантов синтаксиса данного оператора рассмотрим все тот же пример о вычислении суммы квадратов натуральных чисел, но только каждый раз оператор цикла for будет использоваться по-особому.

В листинге 2.2 представлена версия программы, в которой блоки инструкций в операторе цикла for содержат несколько команд (для уменьшения объема кода все комментарии удалены).

Листинг 2.2. Блоки инструкций в операторе цикла for содержат несколько команд

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   system("chcp 1251>nul");
   int n,s,k;
   cout<<"Введите верхнюю границу для суммы: ";
   cin>>n;
   for(k=1,s=0;k<=n;s+=k*k,k++);
   cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl;
   system("pause>nul");
   return 0;
}
```

В данном случае присваивание нулевого начального значения переменной s вынесено в первый блок инструкций оператора for, а команда s+=k*k теперь находится в третьем блоке инструкций (перед командой k++). Тело оператора цикла, таким образом, оказалось пустым (там те-

перь нет команд), поэтому мы фигурные скобки не используем, а просто поставили после `for`-инструкции точку с запятой.

В некотором смысле противоположная ситуация представлена в листинге 2.3 – там первый и третий блоки оператора цикла `for` пустые.

Листинг 2.3. Первый и третий блоки инструкций в операторе цикла for пустые

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   system("chcp 1251>nul");
   int n,s=0,k=1;
   cout<<"Введите верхнюю границу для суммы: ";
   cin>>n;
   for(;k<=n;){
      s+=k*k;
      k++;
   }
   cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl;
   system("pause>nul");
   return 0;
}
```

Присваивание начальных значений переменным `k` и `s` выполняется при их объявлении, поэтому первый блок инструкций в операторе цикла `for` пустой. Команды `s+=k*k` и `k++` вынесены в тело оператора цикла, поэтому пустой и третий блок инструкций в операторе цикла `for`.

Наконец, достаточно экзотическая ситуация проиллюстрирована в листинге 2.4. Там все три блока инструкций (включая второй блок с условием, регулирующим выполнение оператора цикла) пустые.

Листинг 2.4. Все блоки инструкций в операторе цикла for пустые

```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;
int main(){
   system("chcp 1251>nul");
   int n,s=0,k=1;
   cout<<"Введите верхнюю границу для суммы: ";
   cin>>n;
   for(;;){
      s+=k*k;
      k++;
      if(k>n){
         break;
      }
}
cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl;
system("pause>nul");
return 0;
}
```

Пустой второй блок в операторе цикла `for` эквивалентен условию `true`, поэтому формально получается бесконечный цикл. Чтобы выйти из такого бесконечного цикла, в тело оператора цикла `for` добавлен условный оператор `if`. В условном операторе проверяется условие `k>n`. Если оно истинно, то выполняется всего одна команда `break`, которой завершается выполнение оператора цикла.

ПОДРОБНОСТИ

Здесь использована упрощенная форма условного оператора, которая не содержит `else`-ветки. Условный оператор в такой «облегченной» версии выполняется так: проверяется условие после ключевого слова `if`, и если оно истинно, выполняются команды в фигурных скобках после `if`-инструкции. Если условие ложно, то ничего не происходит и управление передается команде после условного оператора.
Что касается инструкции `break`, то это стандартная команда для выхода из операторов цикла (таких как `while`, `do-while` и `for`), а также оператора выбора `switch`. Есть еще одна полезная инструкция `continue`, которая позволяет досрочно завершить не весь оператор цикла, а только текущую его итерацию (текущий цикл)

Оператор цикла do-while

Помимо операторов цикла while и for, в C++ есть оператор цикла do-while.

ТЕОРИЯ

Описание оператора цикла do-while начинается с ключевого слова do, после которого в фигурных скобках (тело оператора цикла) указываются команды, выполняемые в контексте оператора цикла. После блока из фигурных скобок следует инструкция while, а после нее, в круглых скобках, указывается условие. Оператор цикла выполняется до тех пор, пока истинно данное условие. Оно проверяется каждый раз по завершении выполнения команд в теле оператора цикла. Оператор цикла do-while принципиально отличается от оператора цикла while тем, что в операторе do-while сначала выполняются команды, а затем проверяется условие, в то время как в операторе while сначала проверяется условие, а после этого выполняются команды. Таким образом, команды в теле оператора цикла do-while выполняются, по меньшей мере, один раз.

Пример использования оператора цикла do-while приведен в листинге 2.5. В представленной программе вычисляется значение экспоненты для заданного аргумента. Результат проверяется с помощью встроенной математической функции exp(), предназначенной для этих же целей.

НА ЗАМЕТКУ

Для вычисления значения экспоненты от аргумента x может использоваться ряд $\exp(x) = \sum_{k=0}^{\infty}(x^k/k!) = 1 + x + (x^2/2!) + (x^3/3!) + \ldots + (x^k/k!) + \ldots$. На практике, поскольку вычислить бесконечную сумму проблематично, вычисляется ограниченная сумма $\exp(x) \approx \sum_{k=0}^{n}(x^k/k!) = 1 + x + (x^2/2!) + (x^3/3!) + \ldots + (x^n/n!)$. Таким образом, вычисление значения экспоненты сводится к вычислению соответствующей суммы. Чем больше количество слагаемых, тем точнее вычисляется значение для экспоненты.

Вычисляемую сумму удобно представить в виде $\exp(x) \approx \sum_{k=0}^{n} q^k$, где обозначено $q_k = x^k/k!$. Таким образом, имеет место соотношение $q_{k+1} = q^k \cdot (x/(k+1))$. Данным соотношением воспользуемся в программе при вычислении суммы для экспоненты.

Рассмотрим представленный ниже программный код.

Листинг 2.5. Оператор цикла do-while

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
int main(){
// Изменение кодировки консоли:
system("chcp 1251>nul");
// Индекс последнего слагаемого в сумме:
int n=100;
// Аргумент для экспоненты:
double x=1;
// Переменная для записи значения суммы, переменная
// для записи значения добавки и индексная переменная:
double s=0,q=1,k=0;
// Оператор цикла для вычисления суммы:
do{
   // Прибавление к сумме добавки:
   s+=q;
   // Новое значение индексной переменной:
   k++;
   // Новое значение для добавки:
   q*=x/k;
}while(k<=n);
// Отображение результата вычислений:
cout<<"Вычисленное значение: "<<s<<endl;
   // Использование встроенной функции для вычисления
   // значения экспоненты:
   cout<<"Контрольное значение: "<<exp(x)<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

В программе объявляется целочисленная переменная n, значение которой определяет индекс последнего слагаемого в сумме (и на единицу меньше количества слагаемых в сумме). В действительную переменную x записывается значение аргумента экспоненты. Также в программе объявляются переменные s с нулевым начальным значением (значение суммы), q с единичным начальным значением (значение добавки к сумме) и индексная переменная k с нулевым начальным значением.

Для вычисления суммы используется оператор цикла do-while. В теле оператора цикла командой s+=q к сумме прибавляется значение добавки q. Затем командой k++ на единицу увеличивается значение индексной переменной, после чего с помощью команды q*=x/k вычисляется значение добавки для следующей итерации. Оператор цикла выполняется, пока истинно условие k<=n. По завершении выполнения оператора цикла команда cout<<"Вычисленное значение: "<<s<<endl отображает вычисленное в программе значение для экспоненты, а для проверки команда cout<<"Контрольное значение: "<<exp(x)<<endl отображает значение для экспоненты, вычисленное с помощью функции exp().

> **НА ЗАМЕТКУ**
>
> Для использования функции exp() добавляем в программу инструкцию #include <cmath>, позволяющую подключить библиотеку математических функций.

Результат выполнения программы такой:

Результат выполнения программы (из листинга 2.5)

```
Вычисленное значение: 2.71828
Контрольное значение: 2.71828
```

Видим, что оба значения фактически совпадают, что свидетельствует о приемлемой точности вычислений.

Оператор выбора switch

Рассмотрим следующую простую задачу: пользователь вводит число в диапазоне значений от 1 до 3 включительно, а программой выводится название этого числа ("один", "два" или "три"). При написании программного кода воспользуемся оператором выбора switch.

ТЕОРИЯ

Оператор выбора `switch` позволяет проверять значение некоторого выражения, и в зависимости от полученного результата выполнять разные блоки команд. В некотором смысле оператор выбора напоминает условный оператор, но в данном случае альтернатив больше, чем две. Что касается значения проверяемого выражения, то это может быть либо числовое, либо символьное (тип `char`) выражение. Описывается оператор выбора следующим образом. После ключевого слова `switch` в круглых скобках указывается выражение, значение которого проверяется. Далее в теле оператора (в фигурных скобках) следуют `case`-блоки. В каждом `case`-блоке указано контрольное значение для сравнения на предмет совпадения со значением проверяемого выражения, а также команды, которые следует выполнить в случае, если совпадение имеет место. Каждый `case`-блок обычно заканчивается инструкцией `break`. Последним в операторе выбора может быть `default`-блок, команды которого выполняются в случае, если ни в одном из `case`-блоков не было совпадения контрольного значения и значения выражения.

Рассмотрим программный код в листинге 2.6. Представленная там программа организована по следующему принципу:

- Запускается оператор цикла, в теле которого выводится запрос на ввод числа (в диапазоне значений от `1` до `3`).
- Введенное пользователем значение считывается и проверяется в операторе выбора `switch`.
- В зависимости от введенного значения в консольном окне отображается сообщение с названием введенного числа.

Теперь обратимся к программному коду:

Листинг 2.6. Знакомство с оператором выбора switch

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Целочисленные переменные:
```

```
int num,k;
// Выполняется оператор цикла:
for(k=1;k<=5;k++){
   cout<<"Укажите число от 1 до 3: ";
   // Считывание значения переменной:
   cin>>num;
   // Выполняется оператор выбора:
   switch(num){
      case 1:
         cout<<"Это единица"<<endl;
         break;
      case 2:
         cout<<"Это двойка"<<endl;
         break;
      case 3:
         cout<<"Это тройка"<<endl;
         break;
      default:
         cout<<"Я не знаю такого числа"<<endl;
   }
}
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Результат выполнения программы может быть таким, как показано ниже (жирным шрифтом выделены вводимые пользователем значения):

Результат выполнения программы (из листинга 2.6)

```
Укажите число от 1 до 3: 2
Это двойка
Укажите число от 1 до 3: -2
Я не знаю такого числа
```

```
Укажите число от 1 до 3: 1
Это единица
Укажите число от 1 до 3: 3
Это тройка
Укажите число от 1 до 3: 5
Я не знаю такого числа
```

Что касается непосредственно программного кода, то в теле оператора цикла (выполняется ровно пять циклов) командой `cout<<"Укажите число от 1 до 3: "` выводится сообщение с предложением ввести число. Пользователь его вводит, и число записывается в переменную `num`, что достигается благодаря команде `cin>>num`. После этого запускается оператор выбора. Проверяемым выражением указана переменная `num`. В трех `case`-блоках указаны соответственно значения `1`, `2` и `3`. Если значение переменной `num` равно `1`, выполняется команда `cout<<"Это единица"<<endl`. Если значение переменной `num` равно `2`, выполняется команда `cout<<"Это двойка"<<endl`, а если значение переменной `num` равно `3`, то выполняется команда `cout<<"Это тройка"<<endl`. Каждый `case`-блок заканчивается инструкцией `break`.

НА ЗАМЕТКУ

Оператор выбора выполняется так: последовательно просматриваются контрольные значения в `case`-блоках до первого совпадения. Если совпадение найдено, то начинают выполняться команды после соответствующей `case`-инструкции. Команды выполняются до конца тела оператора выбора, или пока не встретится инструкция `break`. Так что если нам нужно, чтобы выполнялись команды только в одном `case`-блоке, то данный `case`-блок должен завершаться `break`-инструкцией.

В `default`-блоке выполняется команда `cout<<"Я не знаю такого числа"<<endl`. Данный блок «вступает в игру», только если при проверке контрольных значений в `case`-блока совпадений не было.

Еще одна небольшая иллюстрация к использованию оператора выбора `switch` представлена в листинге 2.7. В некотором смысле рассматриваемый здесь пример похож на предыдущий, но в нем есть и некоторые особенности. В частности, в программе генерируются случайные числа, а оператор выбора содержит пустые `case`-блоки.

ТЕОРИЯ

Для генерирования случайных чисел используется функция `rand()`, которая доступна после подключения заголовка `<cstdlib>`. Результатом функции `rand()` возвращается целое, равномерно распределенное неотрицательное случайное число от `0` до некоторой верхней границы (которая достаточно большая).

Если нам нужно получить случайное число в диапазоне значений от `a` до `b`, то можно воспользоваться выражением `a+rand()%(b-a+1)`. В этом выражении использован оператор `%` вычисления остатка от целочисленного деления.

Если `rand()` — некоторое случайное число, то результат выражения `rand()%(b-a+1)` — остаток от деления на число `b-a+1`, и это число в диапазоне от `0` до `b-a`. Следовательно, результатом выражения `a+rand()%(b-a+1)` является число (случайное) в диапазоне значений от `a` до `b`, что и требовалось получить.

При генерировании случайных (на самом деле псевдослучайных) чисел необходимо начальное «рабочее» числовое значение, чтобы запустить весь процесс. Процедура называется **инициализацией генератора случайных чисел**. Для выполнения инициализации используется функция `srand()`, аргументом которой передается числовое значение.

Конкретное числовое значение, переданное аргументом функции, особой нагрузки не несет. Важно то, что при одном и том же аргументе функции `srand()` получаем одну и ту же последовательность (псевдо)случайных чисел.

Идея, реализованная в программе, очень проста. Выполняется несколько итераций (циклов), и за каждую итерацию генерируется случайное число в диапазоне возможных значений от `2` до `8` включительно.

В зависимости от полученного значения, через оператор выбора `switch`, отображается определенное сообщение:

- для чисел `2`, `4` и `8` это сообщение со значением числа и фразой о том, что число является степенью двойки;
- для чисел `3` и `6` появляется сообщение со значением числа и фразой о том, что число делится на три;
- для чисел `5` и `7` сообщение содержит значение числа и текстовое его название.

Рассмотрим представленный ниже программный код:

Листинг 2.7. Использование оператора switch

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
// Изменение кодировки консоли:
system("chcp 1251>nul");
// Целочисленная переменная:
int num;
// Инициализация генератора случайных чисел:
srand(2);
// Оператор цикла:
for(int k=1;k<=10;k++){
   // Случайное число от 2 до 8:
   num=2+rand()%7;
   // Оператор выбора:
   switch(num){
      case 3:
      case 6:
         cout<<num<<": число делится на три"<<endl;
         break;
      case 2:
      case 4:
      case 8:
         cout<<num<<": степень двойки"<<endl;
         break;
      case 5:
         cout<<num<<": пятерка"<<endl;
         break;
      case 7:
         cout<<num<<": семерка"<<endl;
   }
```

```
    }
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы может быть таким (с поправкой на то, что используется генератор случайных чисел):

Результат выполнения программы (из листинга 2.7)

```
5: пятерка
7: семерка
8: степень двойки
3: число делится на три
2: степень двойки
3: число делится на три
4: степень двойки
8: степень двойки
4: степень двойки
6: число делится на три
```

В программе командой `srand(2)` инициализируется генератор случайных чисел и объявляется целочисленная переменная `num`. Данной переменной в теле оператора цикла командой `num=2+rand()%7` присваивается случайное значение в диапазоне значений от `2` до `8` включительно.

НА ЗАМЕТКУ

Результат выражения `rand()%7` представляет собой остаток от деления случайного числа на число `7`, то есть это число в диапазоне от `0` до `6`. Если прибавить к нему `2`, то получим число в диапазоне значений от `2` до `8`.

Также обратите внимание, что индексная переменная `k` объявляется непосредственно в `for`-операторе в первом блоке инструкций. Так можно делать, но следует помнить, что объявленная в `for`-операторе переменная доступна только в этом операторе.

Далее в операторе `switch` «перебираются» некоторые варианты. Специфика оператора выбора связана с наличием пустых `case`-блоков: после

одной case-инструкции сразу идет другая. Общий эффект такой: для всех соответствующих контрольных значений выполняется один и тот же блок команд.

> **НА ЗАМЕТКУ**
>
> В последнем case-блоке инструкция break не используется, поскольку в ней нет необходимости: это последний блок, и после его выполнения другие блоки выполняться не будут (поскольку их попросту нет).

Вложенные условные операторы

Условные операторы мы уже не раз использовали. Здесь рассмотрим ситуацию, когда в теле условного оператора используется другой условный оператор. Как иллюстрацию реализуем программный код одного из предыдущих примеров (см. листинг 2.6), но если там использовался оператор выбора switch, то здесь воспользуемся условными операторами. Программа представлена в листинге 2.8.

Листинг 2.8. Вложенные условные операторы

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Целочисленные переменные:
   int num,k;
   // Выполняется оператор цикла:
for(k=1;k<=5;k++){
   cout<<"Укажите число от 1 до 3: ";
   // Считывание значения переменной:
   cin>>num;
   // Внешний условный оператор (1-й уровень):
   if(num==1){
      cout<<"Это единица"<<endl;
```

```
}
// Блок else внешнего
// условного оператора (1-й уровень):
else{
   // Внутренний условный оператор (2-й уровень):
   if(num==2){
   cout<<"Это двойка"<<endl;
}
// Блок else внутреннего оператора (2-й уровень):
else{
   // Внутренний условный оператор (3-й уровень):
   if(num==3){
      cout<<"Это тройка"<<endl;
   }
   // Блок else внутреннего условного
         // оператора (3-й уровнеь):
         else{
            cout<<"Я не знаю такого числа"<<endl;
         }
      }
   }
}
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Ниже представлен результат выполнения программы (жирным шрифтом выделены значения, *введенные пользователем*):

Результат выполнения программы (из листинга 2.8)

```
Укажите число от 1 до 3: 2
Это двойка
Укажите число от 1 до 3: -2
```

```
Я не знаю такого числа
Укажите число от 1 до 3: 1
Это единица
Укажите число от 1 до 3: 3
Это тройка
Укажите число от 1 до 3: 5
Я не знаю такого числа
```

Фактически результат такой же, как и в примере, в котором использовался оператор `switch` (см. листинг 2.6). Здесь мы организовали проверку значения переменной `num` через вложенные условные операторы.

ТЕОРИЯ

Оператор равенства == является, наравне со знакомым нам оператором <= (меньше или равно), одним из операторов сравнения. Кроме этих операторов, полезными могут быть операторы < (меньше), > (больше), >= (больше или равно) и != (не равно).

Вложенные операторы цикла

Рассмотрим пример, в котором выполняется сортировка числового массива. Постановка задачи такова: имеется массив, заполненный числами. Необходимо расположить их (поменять местами значения элементов массива) в порядке возрастания значений.

ПОДРОБНОСТИ

Для сортировки элементов массива используем **метод пузырька**. Суть его в следующем. Последовательно перебираются элементы массива и сравниваются значения двух соседних элементов. Если значение элемента слева больше значения элемента справа, то такие элементы «обмениваются» значениями. В результате одного «прохода» по массиву самое большое значение окажется в правой крайней позиции (станет значением последнего элемента). После еще одного перебора элементов массива у предпоследнего элемента будет второе по величине значение, и так далее. Таким образом, за один перебор элементов массива один элемент оказывается с «правильным» значением.

Программный код, в котором создается целочисленный массив, заполняется случайными числами а затем сортируется *методом пузырька*, представлен в листинге 2.9.

Листинг 2.9. Сортировка массива

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Размер массива:
   const int n=10;
   // Инициализация генератора случайных чисел:
   srand(2);
   // Создание массива:
   int nums[n];
   // Целочисленные переменные:
   int i,j,k,s;
   cout<<"Массив до сортировки:\n| ";
   // Заполнение массива случайными числами:
   for(k=0;k<n;k++){
      nums[k]=rand()%10;
      cout<<nums[k]<<" | ";
   }
   cout<<"\nМассив после сортировки:\n| ";
   // Сортировка массива:
   for(i=1;i<=n-1;i++){
      // Перебор элементов массива:
      for(j=0;j<n-i;j++){
         // "Обмен" значениями:
         if(nums[j]>nums[j+1]){
            s=nums[j+1];
```

```
            nums[j+1]=nums[j];
            nums[j]=s;
        }
    }
}
// Отображение отсортированного массива:
    for(k=0;k<n;k++){
        cout<<nums[k]<<" | ";
    }
    cout<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы может быть таким:

Результат выполнения программы (из листинга 2.9)

```
Массив до сортировки:
| 5 | 6 | 8 | 5 | 4 | 0 | 0 | 1 | 5 | 6 |
Массив после сортировки:
| 0 | 0 | 1 | 4 | 5 | 5 | 5 | 6 | 6 | 8 |
```

В программном коде объявляется целочисленная константа `n`, определяющая размер массива. Сам массив создается командой `int nums[n]`. Заполнение массива выполняется с помощью оператора цикла, в котором перебираются элементы. Значения элементам массива в теле оператора цикла (при определенном значении индексной переменной `k`) присваиваются командой `nums[k]=rand()%10` (получается случайное число в диапазоне значений от `0` до `9`). Одновременно с присваиванием значений элементам массива выполняется отображение значений элементов в консольном окне.

НА ЗАМЕТКУ

Разделителем при отображении элементов массива служит вертикальная черта. В некоторых отображаемых в консоли текстовых значе-

ниях в программе использована инструкция `\n`, которая является командой перехода к новой строке при отображении соответствующего текста. Инструкция `\n` включается непосредственно в текст, а разрыв строки выполняется в том месте, где находится данная инструкция. Обратите внимание, что для массива, состоящего из `n` элементов, переменная `k`, определяющая индекс элемента, пробегает значения в диапазоне от `0` до `n-1` включительно.

После заполнения массива и отображения в консоли значений его элементов, начинается сортировка массива. Для сортировки используются вложенные операторы цикла `for`. Внешним оператором цикла (с индексной переменной `i`) отсчитываются полные переборы элементов массива. Один такой перебор в общем случае означает размещение в «правильной» позиции одного значения.

НА ЗАМЕТКУ

Если массив состоит из *n* элементов, то для сортировки всего массива необходимо последовательно выполнить *n* – 1 полный перебор элементов. Один полный перебор элементов приводит к определению «правильного» места для одного значения. Но дело в том, что когда на второй позиции оказывается «правильное» значение, на первой позиции автоматически оказывается самое маленькое значение. Поэтому количество полных переборов на единицу меньше количества элементов в массиве.

Кроме того, следует учесть, что при полном переборе элементов нет смысла просматривать те значения, которые уже находятся на «правильной» позиции. Поэтому при каждом новом полном переборе просматривается на один элемент меньше.

Внутренний оператор цикла (с индексной переменной `j`) перебирает элементы массива. Верхняя граница для диапазона изменений индекса `j` зависит от значения индекса `i` внешнего массива. Здесь учтен тот факт, что нет смысла перебирать уже отсортированные значения.

В теле внутреннего оператора цикла с помощью условного оператора сравниваются значения двух соседних элементов (условие `nums[j]>nums[j+1]`), и если значение элемента слева больше значения элемента справа, то элементы обмениваются значениями.

По завершении сортировки массива с помощью еще одного оператора цикла значения элементов уже отсортированного массива отображаются в консольном окне.

Цикл по коллекции

В рассмотренных ранее примерах, когда мы имели дело с массивами, обращение к элементам массива выполнялось через явное указание индекса элемента. Существует и иной способ перебора элементов массива, в рамках которого доступ осуществляется непосредственно к элементу массива. Реализуется такой подход с помощью специальной версии оператора цикла `for`. Соответствующую форму выполнения циклов обычно называют *циклом по коллекции*.

Цикл по коллекции реализуется через `for`-оператор, который определяется в следующем виде (жирным шрифтом выделены ключевые элементы шаблона):

```
for(тип &переменная: массив){
    // Тело оператора цикла
}
```

После ключевого слова `for` в круглых скобках указывается `тип` элемента массива и название `переменной`, которая отождествляет собой элемент массива. Перед именем переменной обычно ставится инструкция `&`. После имени `переменной` через двоеточие указывается название `массива`, элементы которого перебираются. Команды, выполняемые за каждый цикл, указываются в фигурных скобках (тело оператора цикла).

Оператор цикла выполняется так: `переменная`, указанная в `for`-инструкции, последовательно ссылается на элементы `массива`. Для каждого такого значения выполняются команды в теле цикла, после чего ссылка перебрасывается на другой элемент массива.

ТЕОРИЯ

Инструкция `&` нужна для того, чтобы переменная, указанная в `for`-инструкции, не просто принимала значения элементов массива, а была **ссылкой** на элемент массива. **Ссылка** — это как бы альтернативное название для элемента массива, поэтому через такую ссылку можно не только прочитать значение элемента массива, но и присвоить ему значение.

Далее рассматривается пример, в котором создается числовой массив, и с помощью цикла по коллекции этот массив заполняется случайными числами, а значения элементов выводятся в консольное окно. Также

с помощью цикла по коллекции выполняется подсчет количества элементов в массиве. Рассмотрим программный код, представленный в листинге 2.10.

Листинг 2.10. Цикл по коллекции

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
   // Создание массива:
   int nums[12];
   cout<<"Массив случайных чисел:\n";
   // Цикл по коллекции:
for(int &x: nums){
   x=rand()%10;   // Случайное число от 0 до 9
   cout<<x<<" ";  // Отображение значения элемента
}
cout<<endl;
// Переменная для записи размера массива:
int length=0;
// Цикл по коллекции:
for(int &x: nums){
   length++;
}
cout<<"Размер массива: "<<length<<endl;
cout<<"Проверка содержимого массива:\n";
// Обычный оператор цикла:
for(int k=0;k<length;k++){
   cout<<nums[k]<<" ";
}
```

```
    cout<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы (с учетом того, что используется генератор случайных чисел) может быть таким:

Результат выполнения программы (из листинга 2.10)

```
Массив случайных чисел:
5 6 8 5 4 0 0 1 5 6 7 9
Размер массива: 12
Проверка содержимого массива:
5 6 8 5 4 0 0 1 5 6 7 9
```

В программном коде командой `int nums[12]` создается числовой массив. Заполнение массива выполняется в операторе цикла, который описан `for`-инструкцией вида `for(int &x: nums)`. Она означает, что перебор выполняется по массиву `nums`, а через `x` обозначена ссылка на элемент данного массива. В фигурных скобках указана команда `x=rand()%10` (элементу массива присваивается случайное число в диапазоне возможных значений от `0` до `9`) и команда `cout<<x<<" "`, которой отображается значение элемента.

Еще один пример использования цикла по коллекции связан с вычислением размера массива.

ТЕОРИЯ

Размер массива в C++ обычно записывается в отдельную переменную (константу). Массив однозначно идентифицируется двумя параметрами: точкой входа (адресом начального элемента) и размером массива. В общем случае определить размер массива по одному только его имени проблематично.

В программном коде объявляется целочисленная переменная `length` с начальным нулевым значением. Затем запускается цикл по коллекции и в теле данного оператора выполняется команда `length++`. Таким образом, при получении ссылки на каждый из элементов массива `nums` значение переменной `length` увеличивается на единицу. После завершения

оператора цикла значение переменной `length` равно количеству элементов в массиве `nums`.

Для проверки содержимое массива `nums` выводится в консольное окно с помощью обычного `for`-оператора, в котором доступ к элементам осуществляется через индекс.

Генерирование и перехват исключений

В C++ есть механизм обработки *исключительных ситуаций* (ошибок), который, помимо своего непосредственного назначения, может использоваться для организации в программе точек ветвления. В таком случае используется процедура намеренного генерирования исключительной ситуации.

ТЕОРИЯ

При обработке исключительных ситуаций код, который может вызвать ошибку, помещается в `try`-блок: такой блок помечается ключевым словом `try` и заключается в фигурные скобки. После `try`-блока следует один или несколько `catch`-блоков. Каждый `catch`-блок предназначен для обработки ошибки определенного типа. Если при выполнении кода в `try`-блоке возникла ошибка, то выполнение этого блока прекращается, а управление передается `catch`-блоку, предназначенному для обработки ошибки соответствующего типа.
Для искусственного генерирования ошибки используют инструкцию `throw`, после которой указывается объект, переменная или значение, которые играют роль объекта ошибки и передаются в `catch`-блок при ее обработке.

Если в программе используется обработка исключительных ситуаций, то соответствующий фрагмент программного кода с одним блоком для перехвата ошибок может выглядеть так (жирным шрифтом выделены ключевые фрагменты кода):

```
try{
    // Код, который может вызвать ошибку
}
catch(тип_ошибки объект_ошибки){
    // Код, выполняемый при обработке ошибки
}
```

Если при выполнении кода в блоке `try` ошибок не было, то программный код в `catch`-блоке (или блоках — если их несколько) игнорируется. Если в `try`-блоке возникла ошибка, то начинает выполняться код в `catch`-блоке. Все это довольно сильно напоминает подход, реализованный в условном операторе, чем, собственно, можно воспользоваться.

Рассмотрим программу в листинге 2.11. Там, с помощью механизма искусственного генерирования исключительной ситуации с последующей ее обработкой, решается задача о поиске корня линейного уравнения вида $Ax = B$ (относительно переменной x).

> **НА ЗАМЕТКУ**
>
> Хотя формально уравнение $Ax = B$ очень простое, тут возможны некоторые специфические ситуации. Во-первых, если $A \neq 0$, то решение уравнения дается формулой $x = B/A$. Если же $A = 0$, и при этом $B \neq 0$, то решений у уравнения нет. И, наконец, если $A = 0$ и одновременно $B = 0$, то решением уравнения является любое число.

В программе для поиска решения уравнения $Ax = B$ с клавиатуры вводятся значения для параметров A и B уравнения. В зависимости от сложившейся ситуации вычисляется решение уравнение или выводится сообщение о том, что решением может быть любое число или что решений нет.

> **НА ЗАМЕТКУ**
>
> Понятно, что анализ возможных ситуаций легко организовать с помощью вложенных условных операторов. Мы используем в некотором смысле альтернативный подход, который, однозначно, не самый оптимальный с точки зрения организации программного кода, но показательный в плане использования механизма обработки исключительных ситуаций для организации точек ветвления.

Интересующий нас программный код выглядит следующим образом:

Листинг 2.11. Генерирование и перехват исключений

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
```

```
// Изменение кодировки консоли:
system("chcp 1251>nul");
cout<<"Решение уравнения Ax = B\n";
// Параметры уравнения:
double A,B;
// Считывание значений для параметров уравнения:
cout<<"A = ";
cin>>A;
cout<<"B = ";
cin>>B;
// Контролируемый блок кода:
try{
   if(A!=0){
      // Генерирование исключения:
      throw A;
   }
   if(B!=0){
      // Генерирование исключения:
      throw "Решений нет";
   }
   cout<<"Решение - любое число"<<endl;
}
// Обработка числового исключения:
catch(double e){
   cout<<"Решение уравнения: "<<B/e<<endl;
}
// Обработка текстового исключения:
   catch(char* e){
      cout<<e<<endl;
   }
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

В зависимости от вводимых значений, результаты выполнения программы качественно различны. Ниже показано, как выглядит результат выполнения программы, если для параметра *A* вводится ненулевое значение:

Результат выполнения программы (из листинга 2.11)

```
Решение уравнения Ax = B
A = 2
B = 10
Решение уравнения: 5
```

Если значение для параметра *A* нулевое, а при этом параметр *B* ненулевой, результат может быть следующим:

Результат выполнения программы (из листинга 2.11)

```
Решение уравнения Ax = B
A = 0
B = 3
Решений нет
```

Если значения для обоих параметров нулевые, результат такой:

Результат выполнения программы (из листинга 2.11)

```
Решение уравнения Ax = B
A = 0
B = 0
Решение — любое число
```

Проанализируем программный код, выполнение которого приводит к таким результатам.

В программе после считывания значений переменных `A` и `B`, начинает выполняться `try`-блок.

В этом блоке в условном операторе проверяется условие `A!=0`, и если оно истинно, командой `throw A` генерируется исключение.

Значение переменной `A` передается в `catch`-блок, предназначенный для обработки ошибки (обсуждается далее).

> **НА ЗАМЕТКУ**
>
> Генерирование исключения означает прекращение выполнения команд в `try`-блоке.

Если ошибка не была сгенерирована (что, очевидно, происходит при нулевом значении переменной `A`), в игру вступает следующий условный оператор, в котором проверяется условие `B!=0`. Если условие истинно, командой `throw "Решений нет"` генерируется ошибка с текстовым литералом, который предается в `catch`-блок для обработки ошибки. Но если и эта ошибка не генерируется, то выполняется команда `cout<<"Решение – любое число"<<endl`. Поскольку генерирование второй ошибки возможно, только если не сгенерировалась первая, то отсутствие сгенерированных ошибок означает, что обе переменные `A` и `B` имеют нулевые значения. Другими словами, команда `cout<<"Решение – любое число"<<endl` выполняется, только если для переменных `A` и `B` с клавиатуры были введены нулевые значения.

После `try`-блока описаны два `catch`-блока. В каждом `catch`-блоке в круглых скобках указывается:

- тип значения, который передается в `catch`-блок для обработки ошибки;
- обозначение для переменной, в которую записывается (или через которую передается) значение, переданное в код обработки при генерировании ошибки.

При генерировании ошибки в `try`-блоке для ее обработки используется только один из двух `catch`-блоков. Решение о том, какой именно, — принимается на основе типа значения, которое передается для обработки: тип передаваемого значения сравнивается с типом, указанным в `catch`-блоке. Блок используется для обработки ошибки, если типы значений (переданного для обработки и указанного в `catch`-блоке) совпадают. В данном случае есть `catch`-блок для обработки `double`-значения, и `catch`-блок для обработки текстового значения (тип `char*`).

> **ПОДРОБНОСТИ**
>
> По умолчанию текст в C++ реализуется в виде символьного массива — то есть массива, состоящего из символов. Символьный тип — это тип `char`. Символьный (и не только) массив передается в функции и, в частности, в `catch`-блок через указатель на свой первый элемент.

Указатель — это переменная, значением которой является адрес другой переменной. Поэтому массив фактически передается через адрес своего первого элемента (который является значением типа `char`). Чтобы показать, что в `catch`-блок передается адрес `char`-элемента, а не сам `char`-элемент, в описании `catch`-блока в идентификаторе типа используется звездочка `*`, а вся инструкция, идентифицирующая тип переданного в `catch`-блок значения, выглядит как `char*`.

В `catch`-блоке для обработки ошибки с передачей `double`-значения перехватывается первая ошибка (которая генерируется командой `throw A`). В этом блоке под `e` подразумевается переданное в блок значение — то есть значение переменной `A`. Поэтому при вычислении значения выражения `B/e` на самом деле вычисляется значение выражения `B/A`.

В `catch`-блоке для обработки исключения с передачей `char*`-значения перехватывается вторая ошибка (сгенерированная командой `throw "Решений нет"`). В этом блоке значением переменной `e` является текст `"Решений нет"` (на самом деле значением `e` является указатель на литерал `"Решений нет"`, но в данном конкретном случае это не столь важно). Поэтому при выполнении команды `cout<<e<<endl` в консольном окне отображается именно указанный текст.

Инструкция безусловного перехода

Как небольшую иллюстрацию скорее к гибкости языка C++, чем к практической стороне программирования, рассмотрим пример, в котором задача о вычислении суммы квадратов натуральных чисел решается с помощью *инструкции безусловного перехода* `goto`.

ТЕОРИЯ

Инструкция `goto` позволяет передавать управление в определенное место программного кода. Место в программном коде помечается **меткой**, и эта метка указывается после инструкции `goto`.

Метка — обычный идентификатор (название выбирается произвольно с учетом наличия зарезервированных ключевых слов), после которого ставится двоеточие.

Использование инструкции `goto` в программном коде многими программистами считается нецелесообразным.

Рассмотрим программный код, представленный в листинге 2.12.

Листинг 2.12. Инструкция безусловного перехода

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Целочисленные переменные:
   int n=10,s=0,k=1;
   start: // Метка
   s+=k*k;
   if(k<n){
      k++;
      // Переход к месту, обозначенному меткой:
      goto start;
   }
   // Отображение результата вычислений:
   cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 2.12)

```
Сумма квадратов чисел от 1 до 10: 385
```

В программе объявлены переменные `n` со значением `10` (верхняя граница суммы), `s` со значением `0` (переменная для записи суммы квадратов чисел), `k` со значением `1` (аналог индексной переменной).

Перед командой `s+=k*k` указана метка `start`. При первом выполнении команды `s+=k*k` наличие метки никак не сказывается. После выполнения команды в условном операторе проверяется условие `k<n`. Если условие истинно, командой `k++` значение переменной `k` увеличивается на

единицу, а вследствие выполнения инструкции `goto start` управление переходит к команде, помеченной меткой `start` — то есть снова выполняется команда `s+=k*k`, после чего выполняется условный оператор, и так далее. Если при проверке условия `k<n` в условном операторе оно окажется ложным, то инструкция `goto start` не выполняется, а выполняется команда `cout<<"Сумма квадратов чисел от 1 до "<<n<<": "<<s<<endl` после условного оператора.

Задачи для самостоятельного решения

1. Напишите программу для вычисления суммы нечетных натуральных чисел с использованием оператора `for`.
2. Напишите программу для вычисления суммы четных натуральных чисел с использованием оператора `do-while`.
3. Напишите такую программу: пользователем с клавиатуры вводится целое число, а программой отображаются все числа, на которые данное число делится без остатка.
4. Напишите программу для вычисления наибольшего общего делителя двух чисел.
5. Напишите программу для вычисления значения натурального логарифма $\ln(1 + x) \approx x - (x^2/2) + (x^3/3) - (x^4/4) + ... + ((-1)^{n+1}x^n)/n$.
6. Напишите программу для вычисления значения синуса $\sin(x) \approx x - (x^3/3!) + (x^5/5!) - (x^7/7!) + ... + ((-1)^n x^{2n+1})/((2n+1)!)$.
7. Напишите программу для вычисления значения косинуса $\cos(x) \approx 1 - (x^2/2!) + (x^4/4!) - (x^6/6!) + ... + ((-1)^n x^{2n})/((2n)!)$.
8. Напишите программу для вычисления значения гиперболического синуса $\mathrm{sh}(x) = (\exp(x) - \exp(-x))/2 \approx x + (x^3/3!) + (x^5/5!) + (x^7/7!) + ... + x^{2n+1}/((2n+1)!)$.
9. Напишите программу для вычисления значения гиперболического косинуса $\mathrm{ch}(x) = (\exp(x) + \exp(-x))/2 \approx 1 + (x^2/2!) + (x^4/4!) + (x^6/6!) + ... + x^{2n}/((2n)!)$.
10. Напишите программу для вычисления значения выражения $(\exp(x) - 1)/x \approx 1 + (x/2!) + (x^2/3!) + (x^3/4!) + ... + x^n/((n+1)!)$.
11. Напишите программу для решения уравнения вида $Ax = B$, используя вложенные условные операторы.
12. Напишите программу для решения уравнения вида $Ax = B$ на множестве целых чисел: параметры A и B являются целыми числами,

и решение x также должно быть целым числом (иначе полагаем, что решений нет).

13. Напишите программу для нахождения (в пределах первой сотни) таких целочисленных параметров a, b и c, что выполняется равенство $a^2 + b^2 = c^2$ (например, $3^2 + 4^2 = 5^2$). Для вычисления квадратного корня может использоваться функция `sqrt()` (доступна после подключения заголовка `<cmath>`).

14. Напишите программу для решения квадратного уравнения вида $Ax^2 + Bx + C = 0$ (предложить разные методы решения задачи).

15. Напишите программу с функцией, которая для своего целочисленного аргумента возвращает результатом сумму делителей (сумма чисел, на которые число-аргумент делится без остатка).

16. Напишите программу (с использованием оператора `switch`), в которой для введенного пользователем числа (в диапазоне от 1 до 10) проверяется, является ли оно числом Фибоначчи (одно из чисел 1, 2, 3, 5 и 8). Предложите альтернативные способы организации программного кода.

17. Напишите программу, в которой создается и заполняется случайными числами массив. Затем массив сортируется в порядке убывания значений элементов.

18. Напишите программу, в которой создается числовой массив. Массив заполняется случайными числами. Необходимо изменить порядок элементов в массиве (чтобы последний элемент стал первым, предпоследний — вторым, и так далее).

19. Напишите программу, в которой создается числовой массив и заполняется случайными целыми числами. Элементы массива должны быть упорядочены следующим образом: сначала следуют в порядке возрастания нечетные значения, а затем в порядке возрастания следуют четные значения. Например, если сначала в массиве были элементы со значениями 7, 2, 5, 1, 8, 4, то после упорядочения должны получить последовательность 1, 5, 7, 2, 4, 8.

20. Напишите программу, в которой создается числовой массив и заполняется случайными целыми числами. Элементы массива должны быть упорядочены следующим образом: сначала следуют в порядке убывания четные значения, а затем в порядке возрастания следуют нечетные значения. Например, если сначала в массиве были элементы со значениями 7, 2, 5, 1, 8, 4, то после упорядочения должны получить последовательность 8, 4, 2, 1, 5, 7.

Глава 3
УКАЗАТЕЛИ, МАССИВЫ И ССЫЛКИ

— Этот тип противен мне так же, как и тебе.
— Ну, конечно, я сам себе противен.

из к/ф «Ирония судьбы, или С легким паром»

В этой главе рассматриваются примеры и задачи, связанные в основном с использованием *указателей* и массивов. Мы, кроме прочего, узнаем:

- как создаются и где используются указатели;
- что связывает массивы и указатели;
- чем *ссылка* отличается от указателя;
- каковы особенности символьных массивов и как с их помощью реализуется текст;
- как динамически выделяется память под переменные и массивы;
- как создается двумерный массив,

а также познакомимся и некоторыми другими особенностями реализации указателей, ссылок и массивов в C++.

Знакомство с указателями

Указатель — переменная, значением которой является адрес другой переменной. Если обычная переменная позволяет получить доступ к области памяти по имени переменной, то указатель позволяет получить доступ к области памяти по адресу. Эти два способа обращения к одной и той же области памяти могут комбинироваться и использоваться одновременно. В рассматриваемом далее примере создаются две переменные, но доступ к ним для присваивания значений выполняется с помощью указателей.

ТЕОРИЯ

Указатель объявляется так: задается тип значения, на которое он может указывать, а после идентификатора типа размещается звездочка *. Далее традиционно следует имя указателя.
Чтобы получить адрес переменной (обычной, не указателя), перед именем переменной размещают &. Чтобы получить значение переменной, адрес которой записан в указателе, перед именем указателя ставят *.

В листинге 3.1 представлен небольшой программный код, в котором используются указатели.

Листинг 3.1. Знакомство с указателями

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Символьная переменная:
   char symb;
   // Целочисленная переменная:
   int num;
   // Указатель на символьное значение:
   char* p;
   // Указатель на целочисленное значение:
   int* q;
   // Значение указателя p - адрес переменной symb:
   p=&symb;
   // Значение указателя q - адрес переменной num:
   q=&num;
   // Значение переменной symb присваивается
   // через указатель на нее:
   *p='A';
```

```
    // Значение переменной num присваивается
    // через указатель на нее:
    *q=100;
    // Проверяется значение переменной symb:
    cout<<"symb = "<<symb<<endl;
    // Проверяется значение переменной num:
    cout<<"num = "<<num<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже приведен результат выполнения программы:

Результат выполнения программы (из листинга 3.1)

```
symb = A
num = 100
```

Командами `char symb` и `int num` в программе соответственно объявляются символьная переменная `symb` (значением может быть символ — одиночный символ заключается в одинарные кавычки) и целочисленная переменная `num`.

Указатель `p` на символьное значение объявляется командой `char* p`. Указатель `p`, таким образом, может содержать значением адрес переменной типа `char`. Значение указателю `p` присваивается командой `p=&symb`, так что в результате значением указателя `p` является адрес переменной `symb`. Аналогично обстоят дела с указателем `q` на целочисленное значение. Указатель объявляется командой `int* q`, а значение указателю присваивается командой `q=&num`, в результате чего значением указателя `q` является адрес переменной `num`.

НА ЗАМЕТКУ

Значения указателям `p` и `q` присваиваются до того, как переменным `symb` и `num` были присвоены значения. Проще говоря, чтобы указателю присвоить значением адрес переменной, данной переменной совсем необязательно присваивать значения. Достаточно, чтобы переменная существовала (то есть была объявлена). Причина в том,

что указателю необходим только адрес переменной, который у нее появляется при объявлении.
Если значением указателя является адрес некоторой области памяти, то будем говорить, что указатель ссылается на эту область, или что он установлен на эту область.

Значения переменным `symb` и `num` присваиваются через указатели. Значение переменной `symb` присваивается командой `*p='A'`. Значение переменной `num` присваивается командой `*q=100`. Мы учли, что для доступа к области памяти, на которую установлен указатель, перед именем указателя следует поставить звездочку `*`. После присваивания значений переменным `symb` и `num` мы проверяем значения переменных командами `cout<<"symb = "<<symb<<endl` и `cout<<"num = "<<num<<endl`, содержащими явное обращение к переменным `symb` и `num`.

Массивы и указатели

Следующий пример иллюстрирует связь между массивами и указателями.

ТЕОРИЯ

Есть несколько важных фактов по поводу массивов (мы пока обсуждаем одномерные статические массивы) и указателей. Вкратце вот они:

- При объявлении массива место под его элементы в памяти выделяется так, что элементы расположены один за другим.
- Имя массива является указателем на его первый элемент.
- К указателям можно прибавлять и отнимать от них целые числа. Результатом является указатель, смещенный от текущего указателя на соответствующее количество ячеек памяти (в направлении увеличения или уменьшения адреса).
- Разницей двух указателей (одного типа) является целое число, определяющее количество позиций (ячеек) между ячейками памяти, на которые ссылаются указатели.
- Указатели можно индексировать (индекс указывается в квадратных скобках после имени указателя и формально может быть отрицательным). Значение индексированного указателя — это значение в ячейке, которая отстоит от данного указателя на количество позиций, определяемое индексом.

Резюме достаточно простое: обращение к элементу массива представляет собой операцию индексирования указателя.

В листинге 3.2 представлена программа, в которой создается символьный массив и заполняется значениями. При этом в значительной степени используются указатели.

Листинг 3.2. Массивы и указатели

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Размер массива:
   const int size=12;
   // Индексная переменная:
   int k;
   // Создание символьного массива:
   char symbs[size];
   // Указатели на символьные значения:
   char* p;
   char* q;
   // Указатель на первый элемент массива:
   p=symbs;
   // Значение первого элемента массива:
   p[0]='A';
   // Указатель на последний элемент массива:
   q=&symbs[size-1];
   // Количество позиций между первым
   // и последним элементами массива:
   cout<<"Между первым и последним элементами "<<q-p<<" позиций\n";
   // Заполнение массива значениями:
   while(p!=q){
      // Указатель на следующий элемент:
      p++;
```

```
    // Вычисление значения элемента массива:
    *p=p[-1]+1;
}
cout<<"Элементы массива\n| ";
// Отображение содержимого массива:
for(k=0;k<size;k++){
    cout<<symbs[k]<<" | ";
}
cout<<"\nЭлементы в обратном порядке\n| ";
// Отображение содержимого массива в обратном порядке:
for(k=0;k<size;k++){
    cout<<q[-k]<<" | ";

}
cout<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 3.2)

```
Между первым и последним элементами 11 позиций
Элементы массива
| A | B | C | D | E | F | G | H | I | J | K | L |
Элементы в обратном порядке
| L | K | J | I | H | G | F | E | D | C | B | A |
```

В программе объявляется целочисленная константа `size`, определяющая размер символьного массива, который создается командой `char symbs[size]`. Также в программе объявляются два символьных указателя `p` и `q` (команды `char* p` и `char* q`). Указателю `p` значение присваивается командой `p=symbs`. Поскольку имя массива является ссылкой на первый элемент массива, то после выполнения данной команды

указатель p ссылается на первый элемент массива symbs. Командой p[0]='A' первому элементу массива symbs присваивается значение 'A'. Так происходит в силу правила индексирования указателей: p[0] представляет собой значение области памяти, которая отстоит на 0 позиций от ячейки, на которую ссылается указатель p. Поскольку на момент выполнения программы указатель p содержит значением адрес первого элемента массива, то именно этот элемент получает значение 'A'.

Значение указателю q присваивается командой q=&symbs[size-1]. Поскольку symbs[size-1] есть ни что иное, как последний элемент массива symbs, то &symbs[size-1] — адрес последнего элемента массива. Таким образом, значением указателя q является адрес последнего элемента массива. Если вычислить разность указателей q-p, то результатом получим количество позиций между последним и первым элементами массива. Данное значение на единицу меньше длины массива (и в данном конкретном случае равно 11 поскольку в массиве 12 элементов).

Для заполнения массива значениями запускается оператор цикла while. В операторе проверяется условие p!=q, которое состоит в том, что значения указателей p и q различны. В теле оператора цикла командой p++ операция инкремента применяется к указателю p. Данная команда эквивалентна команде p=p+1. Прибавление к указателю единицы дает в результате указатель, который установлен на соседнюю ячейку. Поскольку в массиве элементы в памяти расположены подряд один за другим, то после выполнения команды p++ указатель p будет ссылаться на следующий элемент в массиве. Командой *p=p[-1]+1 вычисляется значение элемента массива. В левой части команды стоит выражение *p, являющееся элементом массива, на который установлен указатель p. Выражение p[-1]+1 в правой части содержит инструкцию p[-1]. Это значение элемента, который находится на одну позицию назад (в направлении уменьшения адреса) по отношению к элементу, на который установлен указатель p. Проще говоря, для определения значения элемента, на который установлен указатель p, берется значение предыдущего элемента и к нему прибавляется единица.

НА ЗАМЕТКУ

Следует заметить, что значениями элементов массива symbs являются символы. Операция прибавления к символу числа обрабатывается так. Берется код символа в кодовой таблице и к нему прибавляется число. Если результат необходимо интерпретировать как

символ, то полученное числовое значение становится кодом в кодовой таблице, по которому и возвращается символ. В итоге прибавление единицы к символу дает результатом следующий символ в кодовой таблице (если речь идет о буквах, то получаем следующую букву в алфавите).
Также стоит напомнить, что инструкция `\n` является командой перехода к новой строке при отображении текста.

Таким образом, каждый раз при выполнении одного цикла указатель `p` смещается вправо на одну позицию, и это будет продолжаться до тех пор, пока указатель `p` не станет равным указателю `q` на последний элемент массива.

Для отображения содержимого массива (после его заполнения) используем обычный цикл, в котором обращение к элементам массива выполняется посредством индекса. Еще в одном операторе цикла значения элементов массива отображаются в обратном порядке: с конца массива в начало. В этом операторе цикла индексная переменная `k` пробегает значения от `0` до `size-1`, а обращение к элементам массива выполняется в виде `q[-k]`. Напомним, что `q` — указатель на последний элемент массива, а тогда `q[-k]` представляет собой значение элемента массива, который стоит на `k` позиций влево от последнего элемента. В результате получаем перебор элементов массива в обратном порядке.

Знакомство со ссылками

Для иллюстрации особенностей использования *ссылок* рассмотрим совсем небольшой пример, представленный в листинге 3.3.

Листинг 3.3. Знакомство со ссылками

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // целочисленная переменная:
   int num;
```

```
    // Ссылка на переменную:
    int &ref=num;
    // Присваивание значения переменной:
    num=100;
    // Проверка значения переменной и ссылки:
    cout<<"num = "<<num<<endl;
    cout<<"ref = "<<ref<<endl;
    // Присваивание значения ссылке:
    ref=200;
    // Проверка значения переменной и ссылки:
    cout<<"num = "<<num<<endl;
    cout<<"ref = "<<ref<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В программе объявляется целочисленная переменная `num`, а также еще одна переменная, которая называется `ref`. Объявление и инициализация этой переменной (команда `int &ref=num`) имеет некоторые особенности. А именно:

- при объявлении переменной `ref` перед именем переменной указан амперсанд `&`;
- переменной `ref` присваивается в качестве значения переменная `num`, причем самой переменной `num` значение еще не присвоено.

Две перечисленные особенности характерны для объявления *ссылки*.

ТЕОРИЯ

Ссылка представляет собой переменную, которая ссылается (позволяет получить доступ) на ту же область памяти, что и некоторая другая переменная. С некоторым упрощением ссылку можно считать синонимом определенной переменной.

Дело в том, что при объявлении обычной переменной для нее в памяти выделяется место. При создании ссылки память под ссылку не выделяется, а используется область памяти, выделенная под дру-

гую переменную. В результате получаем две переменные, которые имеют доступ к одной и той же области памяти.
Переменная, область памяти которой используется ссылкой, присваивается ссылке при объявлении. Перед именем ссылки при объявлении используется символ &. Ссылка создается для одной переменной и не может быть «переброшена» на другую переменную.

В программе выполняются такие операции с переменной num и ссылкой ref:

- присваивается значение переменой num, после чего проверяются значения переменной num и ссылки ref;
- присваивается значение ссылке ref и снова проверяются значения переменной num и ссылки ref.

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 3.3)

```
num = 100
ref = 100
num = 200
ref = 200
```

Видим, что изменение переменной num приводит к изменению значения ссылки ref, а изменение значения ссылки ref приводит к изменению значения переменной num. Это вполне логично, если вспомнить, что и переменная num, и ссылка ref имеют дело с одной и той же областью памяти.

Динамическое выделение памяти

Массивы, которые мы рассматривали до этого, были *статическими* — память под них выделялась в процессе компиляции программы. Существуют *динамические* массивы, память под которые выделяется в процессе выполнения программы.

ТЕОРИЯ

Динамическое выделение памяти выполняется с помощью оператора new. После оператора new указывается идентификатор типа,

определяющий, для хранения переменной какого типа выделяется память. Если память выделяется не под отдельную переменную, а под массив, то после идентификатора типа в квадратных скобках указывается размер динамического массива.

Результатом инструкции по выделению динамической памяти возвращается указатель на выделенную область памяти (для отдельной переменной) или указатель на первый элемент массива (если создается массив).

После того, как в динамически выделенной памяти необходимость отпала, ее освобождают с помощью оператора `delete`. После оператора `delete` размещается указатель на освобождаемую область памяти. Если удаляется динамический массив, то между оператором `delete` и указателем на первый элемент удаляемого массива размещаются пустые квадратные скобки `[]`.

В листинге 3.4 представлена программа, в которой создается динамический массив, заполняется значениями, после чего массив удаляется.

Листинг 3.4. Динамический массив

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Указатель на целое число:
   int* size;
   // Динамическое выделение памяти под переменную:
   size=new int;
   cout<<"Введите размер массива: ";
   // Считывание значения с клавиатуры:
   cin>>*size;
   // Указатель на символьное значение:
   char* symbs;
   // Создание символьного массива:
   symbs=new char[*size];
   // Заполнение массива:
```

```
for(int k=0;k<*size;k++){
   symbs[k]='a'+k;
   cout<<symbs[k]<<" ";
}
// Удаление массива:
delete [] symbs;
   // Удаление переменной:
   delete size;
   cout<<"\nМассив и переменная удалены\n";
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Возможный результат выполнения программы представлен ниже (жирным шрифтом выделено значение, которое вводит пользователь):

Результат выполнения программы (из листинга 3.4)

```
Введите размер массива: 12
a b c d e f g h i j k l
Массив и переменная удалены
```

В программе создается два указателя: указатель `size` на целочисленное значение и указатель `symbs` на символьное значение. Значение указателю `size` присваивается командой `size=new int`. В результате выполнения данной команды в памяти динамически выделяется место под целочисленную переменную, а адрес выделенной области памяти записывается значением в указатель `size`. Здесь следует еще раз подчеркнуть, что `size` представляет собой указатель. После динамического выделения памяти этот указатель получает значение. Но область памяти, на которую он ссылается, значение не получает. В программе оно считывается с клавиатуры, для чего используется команда `cin>>*size`. Поскольку считанное значение заносится не в указатель `size`, а в область памяти, на которую он ссылается, то справа от оператора ввода >> используем выражение `*size`.

Динамический символьный массив создается командой `symbs=new char[*size]`. В квадратных скобках указано выражение `*size`, которое

определяет размер массива и является считанным с клавиатуры значением динамически созданной переменной.

НА ЗАМЕТКУ

Мы называем динамически созданной переменной область памяти, выделенную в динамическом режиме. Это не совсем корректный термин, но он достаточно удобный, поэтому мы его будем использовать.

Также достойно внимания следующее обстоятельство: если при создании статического массива размер массива должен задаваться константой, то при создании динамического массива размер может быть задан посредством переменной (не обязательно динамической).

В результате выполнения команды `symbs=new char[*size]` создается массив размера `*size`, а адрес этого массива (адрес его первого элемента) записывается в указатель `symbs`. В статическом массиве имя массива является указателем на первый элемент массива. В данном случае идентификатор `symbs` также является указателем на первый элемент массива. Поэтому мы вполне можем отождествлять `symbs` с именем динамического массива. Мы это, собственно, и делаем, когда с помощью оператора цикла заполняем массив значениями и отображаем значения элементов массива в консольном окне. Значение элементу массива присваивается командой `symbs[k]='a'+k`. В правой части выражение `'a'+k` вычисляется так: к коду символа `'a'` прибавляется значение `k` и полученное число служит кодом, по которому определяется символ, являющийся результатом выражения. В итоге массив `symbs` заполоняется последовательностью букв, начиная с `'a'`.

Для удаления динамического массива `symbs` используется команда `delete [] symbs`, а динамическая переменная удаляется командой `delete size`.

НА ЗАМЕТКУ

Переменные-указатели `size` и `symbs` являются статическими и никуда не удаляются. Под удалением массива `symbs` подразумевается удаление того массива, на который ссылается указатель `symbs`. Про удаление динамической переменной мы говорим в том смысле, что освобождается область памяти, на которую ссылается указатель `size`.

Особенности символьных массивов

Массивы, состоящие из символов (символьные массивы) имеют некоторые особенности, связанные с тем, что символьный массив – основной способ реализации текстовых значений. Об этом следующий пример.

ТЕОРИЯ

Главная особенность символьного массива, содержащего текст — это то, что размер массива в общем случае не совпадает с размером текста, который в него записывается (массив должен быть достаточно большой, чтобы он мог содержать текст разной длины). Как индикатор завершения текста в символьном массиве используют нуль-символ `\0`, у которого нулевой код.
Оператор вывода `<<`, если справа от него указано имя символьного массива, выполняет вывод всех элементов символьного массива, вплоть до нуль-символа.

В листинге 3.5 представлена программа, в которой выполняются определенные манипуляции с текстовыми значениями. Основной для реализации текста служит символьный массив.

ТЕОРИЯ

В C++ существует два способа реализации текста: в виде символьного массива и через объект класса `string`. Однако базовым является способ реализации текста посредством символьного массива — во всяком случае, текстовые литералы реализуются именно так.

Рассмотрим представленный ниже программный код:

Листинг 3.5. Особенности символьных массивов

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Символьный массив:
```

```
    char str[100]="Программируем на C++";
    // Отображение содержимого символьного массива:
    cout<<str<<endl;
    // Поэлементное отображение содержимого
    // символьного массива:
    for(int k=0;str[k];k++){
       cout<<str[k]<<"_";
    }
    cout<<endl;
    // Отображение содержимого начиная
    // с определенной позиции:
    for(char* p=str;*p;p++){
       cout<<p<<endl;
    }
    // Изменение символа в массиве:
    str[13]='\0';
    // Отображение содержимого массива:
    cout<<str<<endl;
    // Отображение, начиная с указанной позиции:
    cout<<str+14<<endl;
    // Отображение текста, к которому прибавляется число:
    cout<<"Раз два три"+4<<endl;
    // Указатель на символьное значение:
    const char* q="Раз два три"+8;
    // Значение символа, на который ссылается указатель:
    cout<<q[0]<<endl;
    // Отображение значения указателя:
    cout<<q<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 3.5)

```
Программируем на C++
П_р_о_г_р_а_м_м_и_р_у_е_м_ _н_а_ _C_+_+_
Программируем на C++
рограммируем на C++
ограммируем на C++
граммируем на C++
раммируем на C++
аммируем на C++
ммируем на C++
мируем на C++
ируем на C++
руем на C++
уем на C++
ем на C++
м на C++
 на C++
на C++
а C++
 C++
C++
++
+
Программируем
на C++
два три
т
три
```

Обсудим основные и наиболее интересные фрагменты программы. Итак, командой `char str[100]="Программируем на C++"` создается символьный

массив `str` и в этот массив автоматически записывается (посимвольно) текст `"Программируем на C++"`. Непосредственно массив состоит из `100` элементов. Текст, который записывается в массив, очевидно, содержит намного меньше букв. Поэтому часть элементов массива остаются неиспользованными. Чтобы при обработке символьного массива было понятно, где заканчивается текст и начинаются «неиспользуемые» элементы, в массив автоматически добавляется нуль-символ `'\0'`.

НА ЗАМЕТКУ

Текст состоит из `20` символов. При их записи в массив `str` первые `20` элементов массива будут заполнены символами из текста, а `21`-й элемент массива автоматически получает значением нуль-символ `'\0'`.

Как отмечалось ранее, при реализации текста через символьный массив размер массива и размер текста отличаются. Для вычисления длины текста, записанного в массив, используют функцию `strlen()`. Аргументом функции передается имя символьного массива с текстом, а результатом возвращается количество символов в тексте, записанном в массив.

Для отображения текста из массива `str` в консольном окне используем команду `cout<<str<<endl`. Формально в данном случае отображается значение указателя (поскольку имя массива является указателем на первый элемент массива). Если бы речь шла, например, о числовом массиве, то был бы отображен числовой код адреса, являющегося значением указателя. Но символьные массивы (а если точнее, то указатели на символьные значения) обрабатываются особым образом: при попытке вывести в консольное окно с помощью оператора вывода значения указателя на символ на самом деле отображается символ из ячейки, на которую установлен указатель, а также символы из следующих ячеек. Процесс продолжается до тех пор, пока в очередной ячейке не будет прочитан нуль-символ. Итог такой: чтобы отобразить содержимое символьного массива в консоли, следует справа от оператора вывода указать имя массива.

Вместе с тем никто нам не запрещает обрабатывать и отображать содержимое символьного массива поэлементно. Ситуацию иллюстрирует оператор цикла, в котором текст из массива `str` отображается символ за символом, при этом после каждого символа добавляется символ подчеркивания.

ПОДРОБНОСТИ

Условием в операторе цикла указано выражение `str[k]`, являющееся символом, записанным в массив `str` под индексом `k`. Здесь мы воспользовались тем, что отличные от нуля числовые значения интерпретируются как значение `true`, а нулевое значение интерпретируется как `false`. Если речь идет о символе, то числовое значение — это код символа. Единственный символ, который имеет нулевой код — это нуль-символ. Поэтому если выражение вида `str[k]` указано условием, то такое условие эквивалентно `false`, только если `str[k]` является нуль-символом, и эквивалентно `true` во всех прочих случаях.

Дополнительный пример «нестандартного» отображения текста приводится еще в одном операторе цикла. В этом `for`-операторе в первом блоке инструкций объявлен указатель `p` на символьное значение, начальным значением котором присваивается имя массива `str` — то есть в начале выполнения оператора указатель `p` содержит адрес первого элемента в массиве `str`. За каждый цикл значение указателя `p` командой `p++` увеличивается таким образом, что данный указатель перебрасывается на следующий (за текущим) элемент массива. Проверяемым условие в операторе цикла является выражение `*p`. Это символ, адрес которого записан в указателе `p`. Значение условие становится равным `false`, если указатель содержит адрес элемента массива с нуль-символом. В теле оператора цикла выполняется команда `cout<<p<<endl`. Обрабатывается она так: отображаются все символы в тексте, начиная с того места, куда указывает `p`, и до конца текстового значения. Поскольку указатель `p` постепенно перемещается от начала текста в его конец, то каждое новое сообщение в консольном окне будет короче предыдущего на одну начальную букву.

Следующая часть эксперимента с текстовым массивом `str` состоит в том, что командой `str[13]='\0'` вместо пробела в текст вставляется нуль-символ. Если теперь воспользоваться командой `cout<<str<<endl`, то будет отображаться текст до первого нуль-символа. Теперь это элемент с индексом `13`. Если мы воспользуемся командой `cout<<str+14<<endl`, то в результате будет отображаться текст, начиная с символа с индексом `14` (индексация элементов в символьном массиве, как и в прочих массивах, начинается с нуля).

Несколько следующих команд иллюстрируют специфику реализации текстовых литералов.

ТЕОРИЯ

Текстовые литералы в памяти реализуются в виде символьных массивов. Сам литерал на самом деле передается в выражения и обрабатывается через неявный указатель на свой первый символ.

В результате выполнения команды `cout<<"Раз два три"+4<<endl` отображается текст `"Раз два три"` начиная с символа с индексом `4`. Причина в том, что текстовый литерал `"Раз два три"` в операции сложения с числом обрабатывается как указатель на первый символ в тексте. После прибавления к указателю числа `4` получаем указатель, смещенный по отношению к первому символу на `4` позиции.

Похожая ситуация имеет место при выполнении команды `const char* q="Раз два три"+8`. В результате ее выполнения указатель `q` содержит адрес символа с индексом `8` в тексте `"Раз два три"`.

НА ЗАМЕТКУ

Указатель, который возвращается для текстового литерала, является константой (что вполне логично). Если мы данный указатель присваиваем значением переменной (указателю), то данный указатель должен быть объявлен как константа, или, в противном случае, перед выражением в правой части ставится инструкция `(char*)` для выполнения явного приведения типа.

При выполнении команды `cout<<q[0]<<endl` отображается символ, адрес которого записан в указателе `q`. А если воспользоваться командой `cout<<q<<endl`, то будет отображен текст от текущего символа до конца строки.

Двумерные массивы

В двумерном массиве доступ к элементу выполняется с помощью двух индексов. В листинге 3.6 представлена программа, в которой создается двумерным массив. Массив заполняется случайными символами (буквами) и его содержимое выводится в консольное окно.

ТЕОРИЯ

При создании двумерного массива указывается тип его элементов, название массива и размер по каждому из индексов. Для каждого индекса используются отдельные квадратные скобки.

Двумерный массив удобно представлять в виде таблицы или матрицы. Размер по первому индексу определяет количество строк в матрице, а размер по второму индексу определяет количество столбцов. Обращение к элементу двумерного массива выполняется указанием имени массива и пары индексов. Индексация (по каждому из двух индексов) начинается с нуля.

Рассмотрим представленный ниже программный код:

Листинг 3.6. Двумерный массив из случайных букв

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
   // Количество "столбцов" в массиве:
   const int width=9;
   // Количество "строк" в массиве:
   const int height=5;
   // Создание двумерного массива:
   char Lts[height][width];
   // Заполнение двумерного массива:
   for(int i=0;i<height;i++){
      for(int j=0;j<width;j++){
         // Случайная буква от 'A' до 'Z':
         Lts[i][j]='A'+rand()%25;
         // Отображение значения элемента массива:
         cout<<Lts[i][j]<<" ";
      }
      // Переход к новой строке:
      cout<<endl;
```

```
    }
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы (с поправкой на использование генератора случайных чисел) может быть следующим:

Результат выполнения программы (из листинга 3.6)

```
U Q X U J A P G F
Q H O C G D E N R
I L O K R R U P M
Y L M X U C O Y N
X M O V Y B E M Y
```

Что касается непосредственно программного кода, то там объявляются две целочисленные константы `width` и `height`, которые определяют размеры массива. Сам массив создаем командой `char Lts[height][width]`. Для заполнения массива и вывода значений его элементов в консоль используются вложенные операторы цикла. Индексная переменная `i` во внешнем операторе цикла перебирает строки массива, а индексная переменная `j` во внутреннем операторе цикла перебирает столбцы массива. Значения элементам массива присваиваются командой `Lts[i][j]='A'+rand()%25`. Значение элемента массива получается прибавлением к коду символа `'A'` целого случайного числа в диапазоне значений от `0` до `24`, и по полученному таким образом коду восстанавливается символ. Отображается значение элемента двумерного массива командой `cout<<Lts[i][j]<<" "`. Разделителем между символами в одной строке является пробел. После завершения отображения строки командой `cout<<endl` выполняется переход к новой строке.

В качестве еще одной иллюстрации рассмотрим задачу о вычислении произведения двух квадратных матриц.

ПОДРОБНОСТИ

Квадратной называется матрица, у которой количество строк совпадает с количеством столбцов.

Если матрица A состоит из элементов a_{ij}, а матрица B состоит из элементов b_{ij} (индексы $i,j = 1,2,\ldots,n$), то элементы c_{ij} матрицы $C = AB$, равной произведению матриц A и B, определяются соотношением $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. Именно эту формулу мы используем для вычисления значений элементов матрицы-произведения.

В листинге 3.7 представлена программа, в которой определяются две квадратные матрицы, и вычисляется их произведение. Исходные матрицы и матрица-результат отображаются в консольном окне.

НА ЗАМЕТКУ

В программе для вывода информации в консольное окно вместо оператора вывода используется функция `printf()`. Для этого мы подключаем в программе заголовок `<cstdio>`. Напротив, поскольку оператор вывода не используется, заголовок `<iostream>` можно не подключать. Функция `printf()` удобна тем, что позволяет выполнять форматированный вывод в консольное окно. Эта особенность функции нам пригодится при отображении содержимого двумерных массивов.

Рассмотрим представленный ниже программный код:

Листинг 3.7. Произведение матриц

```
#include <cstdlib>
#include <cstdio>
using namespace std;
// Глобальная константа:
const int n=3;
// Функция для отображения элементов двумерного массива:
void show(int M[n][n]){
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         // Отображение элемента массива:
         printf("%4d",M[i][j]);
      }
      // Переход к новой строке:
      printf("\n");
   }
```

```
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Объявление и инициализация двумерного массива:
   int A[n][n]={{1,-2,1},{2,0,-1},{2,3,-1}};
   printf("Матрица A:\n");
   // Отображение содержимого массива:
   show(A);
   // Объявление и инициализация двумерного массива:
   int B[n][n]={{2,1,-1},{1,3,1},{-2,1,4}};
   printf("Матрица B:\n");
   // Отображение содержимого массива:
   show(B);
   // Объявление двумерного массива:
   int C[n][n];
   // Вычисление произведения матриц:
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         C[i][j]=0; // Начальное значение элемента
         for(int k=0;k<n;k++){
            C[i][j]+=A[i][k]*B[k][j];
         }
      }
   }
   printf("Матрица C=A*B:\n");
   // Отображение содержимого массива:
   show(C);
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 3.7)

```
Матрица A:
   1  -2   1
   2   0  -1
   2   3  -1
Матрица B:
   2   1  -1
   1   3   1
  -2   1   4
Матрица C=A*B:
  -2  -4   1
   6   1  -6
   9  10  -3
```

Данный программный код имеет некоторые особенности. Вкратце они сводятся к следующему.

- В программе объявляется глобальная целочисленная константа `n`, определяющая размер двумерных массивов.
- Для отображения содержимого двумерных массивов (через которые реализуются матрицы) описана функция `show()`.
- Для вывода сообщений в консольное окно используется функция `printf()` (и, соответственно, не используется оператор вывода).

Целочисленная константа `n` объявляется самым обычным образом, но вот инструкция с объявлением константы находится не в теле главной функции программы, а в самой начальной части кода, до объявления функций `show()` и функции `main()`. Это необходимо, поскольку и в функции `show()`, и в функции `main()` используется константа `n`.

ТЕОРИЯ

Область доступности переменных (и констант) ограничивается блоком, в котором они объявлены. Если объявить константу `n` в функции `main()`, то доступна она будет только в функции `main()`.

Функция `show()` предназначена для отображения содержимого двумерного массива. Она не возвращает результат, поэтому идентификатором результата функции указано ключевое слово `void`. Функция описана с одним аргументом, который задан инструкцией `int M[n][n]`, подразумевающей, что речь идет о двумерном целочисленном массиве. В теле функции запускаются вложенные операторы цикла. Во внешнем операторе цикла индексная переменная `i` перебирает строки массива, а индексная переменная `j` во внутреннем операторе цикла перебирает столбцы массива. Во внутреннем операторе цикла командой `printf("%4d",M[i][j])` отображается значение элемента с индексами `i` и `j` двумерного массива `M`. После завершения внутреннего оператора цикла для перехода к новой строке при отображении элементов выполняется команда `printf("\n")`.

ПОДРОБНОСТИ

С помощью функции `printf()` выполняется форматированный вывод в консольное окно. Если аргументом функции передать текстовое значение, то данное текстовое значение будет отображено в консоли. Пример такой ситуации дает команда `printf("\n")`, которой «печатается» инструкция перехода к новой строке `\n` — проще говоря, выполняется переход к новой строке. Но это частный случай ситуации, когда в консоли отображается текст. Примеры вызова функции `printf()` с одним текстовым аргументом встречаются в функции `main()` (например, команда `printf("Матрица A:\n")`).

Если речь идет об отображении, например, числового значения, то это числовое значение передается вторым аргументом функции `printf()`. Первым аргументом функции в таком случае передается специальная текстовая строка, которая, фактически, определяет формат отображения данных. Примером такой ситуации служит команда `printf("%4d",M[i][j])`. Здесь отображаемое числовое значение `M[i][j]` передано вторым аргументом функции `printf()`. Первым аргументом функции передается текстовая строка форматирования `"%4d"`. В этой строке символ `d` означает, что отображается целое число, а цифра `4` определяет количество позиций, которые выделяются для отображения числа (даже если число состоит из одной цифры, под него выделяется четыре позиции).

В главной функции программы командами `int A[n][n]={{1,-2,1},{2,0,-1},{2,3,-1}}` и `int B[n][n]={{2,1,-1},{1,3,1},{-2,1,4}}` объявляются и инициализируются два целочисленных двумерных массива `A` и `B`. Значения, присваиваемые элементам массива, указываются, через опера-

тор присваивания, в фигурных скобках. Для отображения содержимого массивов использованы команды `show(A)` и `show(B)` соответственно.

Двумерный массив `C` только объявляется (не инициализируется), для чего задействована команда `int C[n][n]`. Заполняется данный массив значениями в процессе вычисления произведения матриц. Для этого запускаются вложенные операторы цикла. Во внешнем операторе цикла индексная переменная `i` определяет первый индекс вычисляемого элемента. Во внутреннем операторе цикла индексная переменная `j` определяет второй индекс вычисляемого массива. При фиксированных значениях индексов `i` и `j` вычисления выполняются следующим образом. Сначала командой `C[i][j]=0` вычисляемому элементу массива присваивается нулевое значение. После этого запускается оператор цикла с индексной переменной `k`, и для каждого ее значения из диапазона от `0` до `n-1` командой `C[i][j]+=A[i][k]*B[k][j]` к текущему значению элемента `C[i][j]` прибавляется произведение `A[i][k]*B[k][j]`. Тем самым реализуется формула для вычисления элементов матрицы-произведения.

После завершения вычислений командой `show(C)` содержимое двумерного массива `C` выводится в консольное окно.

Массивы указателей

Элементами массива, кроме прочего, могут быть указатели. Другими словами, никто не запрещает нам создать массив из указателей. Более конкретно, представим, что создается массив из указателей на целочисленные значения. Элементы такого массива являются указателями. Имя данного массива является *указателем на указатель* на целое число.

ТЕОРИЯ

Чтобы объявит указатель на указатель, в команде объявления используют две звездочки `**`. Например, инструкция `int** p` объявляет указатель `p`, значением которого может быть адрес переменной, которая сама является указателем на целочисленное значение.

Далее поступаем следующим образом: создаем динамические числовые массивы и адреса этих массивов (их первых элементов) присваиваем значениями элементам массива указателей. Получается специфическая конструкция, которая по всем внешним признакам напоминает двумерный массив, только в таком массиве в разных строках может содержаться

разное количество элементов. Означенный подход иллюстрирует программа, представленная в листинге 3.8.

Листинг 3.8. Массив числовых указателей

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
   // Индексные переменные:
   int i,j;
   // Размер массива указателей:
   const int size=5;
// Массив со значениями, определяющими размеры
// числовых массивов:
const int cols[size]={3,7,6,4,2};
// Динамический массив указателей:
int** nums=new int*[size];
// Создание динамических числовых массивов
// и заполнение их значениями:
for(i=0;i<size;i++){
   // Динамический числовой массив:
   nums[i]=new int[cols[i]];
   cout<<"| ";
   // Заполнение числового массива:
   for(j=0;j<cols[i];j++){
      // Случайное число от 0 до 9:
      nums[i][j]=rand()%10;
      // Отображение значения элемента массива:
      cout<<nums[i][j]<<" | ";
   }
```

```
    cout<<endl;
}
// Удаление числовых динамических массивов:
for(i=0;i<size;i++){
    // Удаление динамического числового массива:
    delete [] nums[i];
    }
    // Удаление динамического массива указателей:
    delete [] nums;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В программе, кроме прочего, создается целочисленная константа `size`, которая определяет массив указателей. Массив создаем командой `int** nums=new int*[size]`. Это динамический массив. Тип его элементов определяется инструкцией `int*` в правой части выражения. Звездочка * после `int`-инструкции свидетельствует о том, что речь идет об указателях на целые числа. Подчеркнем, что указатели на целые числа — элементы созданного массива. Имя массива является указателем на первый элемент, который в данном случае сам является указателем. Поэтому имя массива `nums` является указателем на указатель на целое число. Поэтому переменная `nums` объявлена с идентификатором типа `int**`.

ПОДРОБНОСТИ

В инструкции объявления указателя звездочка * на самом деле относится к объявляемой переменной, а не к идентификатору типа. Например, инструкцией `int* x,y` объявляется указатель `x` на целочисленное значение и обычная (не указатель) целочисленная переменная `y`. Более того, вместо объявления `int* x,y` часто используют объявление `int *x,y`. Здесь речь идет об одной и той же инструкции, но записанной немного по-разному. Второй способ подчеркивает, что звездочка * «действует» только на одну переменную. Поэтому полагать, что `int*` является идентификатором для типа данных не совсем корректно. Вместе с тем, если инструкции вида `int*` или `int**` рассматривать как *идентификаторы типа*, то во многих случаях это способствует лучшему пониманию принципов организации программного кода.

Еще в программе командой `const int cols[size]={3,7,6,4,2}` объявляется и инициализируется константный (массив с элементами-константами) массив `cols`. Элементы данного массива определяют размер динамических числовых массивов, которые создаются далее в программе и указатели на которые записываются в массив `nums`.

Для создания динамических числовых массивов и заполнения их значениями запускается оператор цикла, в котором индексная `i` переменная пробегает значения от `0` до `size-1`. В теле оператора цикла командой `nums[i]=new int[cols[i]]` создается динамический числовой массив. Размер массива определяется элементом `cols[i]` массива `cols`. Указатель на массив присваивается значением элементу `nums[i]` массива указателей. После того, как числовой динамический массив создан, он заполняется случайными числами, а значения его элементов выводятся в консольное окно. Реализуется все перечисленное посредством оператора цикла, в котором индексная переменная `j` последовательно принимает значения в диапазоне от `0` до `cols[i]-1`. Значение элементу присваивается командой `nums[i][j]=rand()%10`, а отображается значение элемента в консоли командой `cout<<nums[i][j]<<" | "`.

ПОДРОБНОСТИ

Массив `nums` является одномерным массивом. Инструкция `nums[i]` дает значение элемента массива с индексом `i`. Но такой элемент является указателем, причем указателем на начальный элемент динамического числового массива. Указатели, как мы помним, можно индексировать. Поэтому `nums[i][j]` представляет собой элемент с индексом `j` в массиве, на который указывает элемент `nums[i]`. Другими словами, мы организовали одномерные массивы так, что создается полная иллюзия двумерного массива.

После того, как динамические массивы созданы и заполнены, потребность в них отпадает, и они удаляются из памяти. Сначала удаляются числовые массивы, для чего запускается оператор цикла и в теле цикла командой `delete [] nums[i]` удаляется каждый из числовых динамических массивов. После этого командой `delete [] nums` удаляется динамический массив указателей.

С поправкой на то, что в программе используется генератор случайных чисел (поэтому конкретные числовые значения при запуске программы на разных компьютерах могут отличаться), результат выполнения программы выглядит следующим образом:

Результат выполнения программы (из листинга 3.8)

```
| 5 | 6 | 8 |
| 5 | 4 | 0 | 0 | 1 | 5 | 6 |
| 7 | 9 | 2 | 6 | 3 | 9 |
| 3 | 7 | 8 | 1 |
| 9 | 5 |
```

Ранее мы уже имели дело с двумерным символьным массивом (см. листинг 3.6). Однако там вся «символьность» массива сводилась к тому, что элементы массива являлись символами. Массив создавался и обрабатывался поэлементно, и в этом смысле он мало отличается от двумерных массивов прочих типов. Вместе с тем символьные массивы (и это мы тоже знаем) в C++ находятся на особом счету. Некоторые особенности символьного двумерного массива проиллюстрированы в программном коде в листинге 3.9. Если быть более точным, то там создается одномерный массив из указателей на указатели на символьные значения. Ситуация немного похожа на ту, что описывалась в предыдущем примере.

Итак, рассмотрим представленный ниже программный код:

Листинг 3.9. Массив символьных указателей

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Массив символьных указателей:
   char* str[3]={{"красный"},{"желтый"},{"зеленый"}};
   // Отображение строк:
   for(int i=0;i<3;i++){
      cout<<str[i]<<endl;
   }
   // Обращение к элементам массива:
   cout<<str[0][3]<<str[1][1]<<str[0][1];
   cout<<str[2][5]<<str[1][5]<<endl;
```

```
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы следующий:

Результат выполнения программы (из листинга 3.9)

```
красный
желтый
зеленый
серый
```

Массив символьных указателей создается командой `char* str[3]={{"красный"},{"желтый"},{"зеленый"}}`. Формально данная команда означает, что создается массив из трех элементов, каждый из которых является указателем на значение типа `char`. В правой части выражения указан список значений, которыми инициализируются элементы массива `str`. Текстовые литералы `"красный"`, `"желтый"` и `"зеленый"` автоматически записываются в массивы и при обработке этих литералов возвращаются указатели на соответствующие массивы. В результате значениями элементам массива `str` присваиваются адреса первых элементов массивов, в которые были записаны текстовые литералы. Поэтому, например, при выполнении команды вида `cout<<str[i]<<endl` в операторе цикла речь идет о выводе в консоль символьного указателя, и в результате отображается соответствующий текстовый литерал. В то же время мы можем обращаться к элементам массива `str` так, как если бы это был двумерный массива. Например, `str[0][3]` представляет собой букву с индексом `3` в литерале с индексом `0` (четвертая буква в первом литерале — буква `'c'`), а `str[1][1]` — вторая буква во втором литерале (буква `'e'`), и так далее. Принцип здесь простой. Инструкция `str[i]` представляет собой указатель на символьное значение, а на самом деле указатель на первый элемент символьного массива. Тогда инструкция `str[i][j]` является буквой с индексом `j` в массиве, адрес которого (его первого элемента) содержится в элементе с индексом `i` массива `str`.

НА ЗАМЕТКУ

Если мы попытаемся присвоить новое значение элементу `str[i][j]` (скажем, добавим в программный код команду `srt[0][3]='z'`), по-

лучим ошибку времени выполнения, поскольку здесь имеет место попытка изменить значение текстового литерала, который является константой.

Задачи для самостоятельного решения

1. Напишите программу, в которой создается два числовых массива одинакового размера. Необходимо вычислить сумму попарных произведений элементов этих массивов. Так, если через a_k и b_k обозначить элементы массивов (индекс $0 \le k < n$), то необходимо вычислить сумму $\sum_{k=0}^{n-1} a_k b_k$.
2. Напишите программу, в которой создается числовой массив и для этого массива вычисляется сумма квадратов элементов массива.
3. Напишите программу, в которой создается двумерный числовой массив и для этого массива вычисляется сумма квадратов его элементов.
4. Напишите программу, в которой создается квадратная матрица (реализуется через двумерный массив). Матрицу необходимо заполнить числовыми значениями (способ заполнения выбрать самостоятельно), а затем выполнить *транспонирование* матрицы: матрица симметрично «отражается» относительно главной диагонали, в результате чего элемент матрицы a_{ij} становится элементом a_{ji}, и наоборот.
5. Напишите программу, в которой создается квадратная матрица (реализуется через двумерный массив). Матрица заполняется случайными числами, после чего выполняется «поворот по часовой стрелке»: первый столбец становится первой строкой, второй столбец становится второй строкой, и так далее.
6. Напишите программу, в которой для двумерной квадратной числовой матрицы вычисляется *след* — сумма диагональных элементов матрицы.
7. Напишите программу, в которой для одномерного числового массива вычисляется наибольший (наименьший) элемент и позиция, на которой элемент находится в массиве.
8. Напишите программу, в которой создается одномерный числовой массив. После заполнения значениями (например, случайными числами) массива в нем нужно выполнить циклическую перестановку элементов. Количество позиций для циклической перестановки вводится пользователем с клавиатуры.

9. Напишите программу, в которой создается и заполняется натуральными числами двумерный массив. Заполнение начинается с левого верхнего элемента слева направо, сверху вниз (то есть заполняется сначала первая строка, затем вторая, и так далее).

10. Напишите программу, в которой создается и заполняется натуральными числами двумерный массив. Заполнение начинается с левого верхнего элемента сверху вниз, слева направо (то есть заполняется сначала первый столбец, затем второй и так далее).

11. Напишите программу, в которой создается квадратная матрица. Элементам на обеих диагоналях присваиваются единичные значения, все остальные элементы нулевые.

12. Напишите программу для вычисления произведения прямоугольных матриц (количество строк и столбцов в матрицах различное). Перемножаются матрицы A (размерами m на n) и B (размерами n на l). Результатом является матрица C (размерами m на l). Рабочая формула для вычисления значений матрицы C имеет вид $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, где $1 \le i \le m$ и $1 \le j \le l$.

13. Напишите программу, в которой создается символьный массив для записи текста. В массив записывается текст, после чего необходимо изменить порядок следования символов в тексте на противоположный: последний символ становится первым, предпоследний символ становится вторым и так далее.

14. Напишите программу, в которой создается двумерный числовой массив и заполняется случайными числами. Для каждой строки (или столбца) двумерного массива определяется наибольший (или наименьший) элемент, и из таких элементов создается одномерный числовой массив.

15. Напишите программу, в которой создается символьный массив для записи текста. Подсчитать для записанного в массиве текста количество слов и длину каждого слова. Словами считать набор символов между пробелами.

16. Напишите программу, в которой создается динамический массив. Размер массива — случайное число (то есть генерируется случайное число, которое определяет размер массива). Заполнить массив «симметричными» значениями: первый и последний элемент получают значение 1, второй и предпоследний элемент получают значение 2, и так далее.

17. Напишите программу, в которой создается три одномерных числовых массива одинакового размера. Первые два массива заполняют-

ся случайными числами. Третий массив заполняется так: сравниваются элементы с такими же индексами в первом и втором массиве и в третий массив на такую же позицию записывается большее (или меньшее) из сравниваемых значений.

18. Напишите программу, в которой создается два динамических массива (разной длины). Массивы заполняются случайными числами. Затем создается третий динамический массив, и его размер равен сумме размеров первых двух массивов. Третий массив заполняется так: сначала в него записываются значения элементов первого массива, а затем значения элементов второго массива.

19. Напишите программу, в которой создается два динамических массива одинакового размера. Массивы заполняются случайными числами. Затем создается третий динамический массив, и его размер в два раза больше размера каждого из первых двух массивов. Третий массив заполняется поочередной записью элементов из первых двух массивов: сначала записывается значение элемента первого массива, затем значение элемента второго массива, затем снова первого и снова второго, и так далее.

20. Напишите программу, в которой создается двумерный числовой массив. Массив заполняется случайными числами. На его основе создается новый массив, который получается «вычеркиванием» из старого одной строки и одного столбца. Номера «вычеркиваемых» столбца и строки определяются вводом с клавиатуры или с помощью генератора случайных чисел.

Глава 4
ФУНКЦИИ

— Ну, вот у вас, на Земле, как вы определяете, кто перед кем сколько должен присесть?
— Ну, это на глаз...

из к/ф «Кин-Дза-Дза»

Эта глава посвящена в основном функциям. Некоторые достаточно простые примеры использования функций нам уже встречались ранее. Здесь же мы постараемся показать всю мощь и красоту применения функций в программных кодах. Предметом нашего рассмотрения станут следующие вопросы и темы:

- перегрузка функций;
- механизмы передачи аргументов функциям;
- значения аргументов функции по умолчанию;
- использование рекурсии;
- передача аргументом функции указателей, массивов и текстовых значений;
- возвращение результатом функции указателя и ссылки;
- использование указателей на функции,

и некоторые другие аспекты, связанные с функциями.

Объявление и описание функции

Очень простой пример использования функций представлен *в листинге* 4.1. Там описаны математические функции для вычисления синуса и косинуса. В программе данные функции используются для вычисления значений при заданном аргументе, а аналогичные встроенные функции вызываются для проверки результата вычислений.

ПОДРОБНОСТИ

Для вычисления синуса и косинуса (при заданном значении аргумента x) нами используются следующие приближенные выражения: $\cos(x) \approx 1 - (x^2/2!) + (x^4/4!) - (x^6/6!) + \ldots + ((-1)^n x^{2n})/((2n)!) = \sum_{k=0}^{n}(((-1)^k x^{2k})/((2k)!))$ для косинуса и $\sin(x) \approx x - (x^3/3!) + (x^5/5!) - (x^7/7!) + \ldots + ((-1)^n x^{2n+1})/((2n+1)!) = \sum_{k=0}^{n}(((-1)^k x^{2k+1})/((2k+1)!))$ для синуса. Чем больше значение верхней границы суммы n, тем выше точность вычислений.

Рассмотрим представленный ниже программный код:

Листинг 4.1. Вычисление синуса и косинуса

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Объявление функций:
double myCos(double); // Вычисление косинуса
double mySin(double); // Вычисление синуса
void show(double);    // Вспомогательная функция
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Константа для значения числа "пи":
   const double pi=3.141592;
   // Вычисление синуса и косинуса при разных
   // значениях аргумента:
   show(pi/6);
   show(pi/4);
   show(pi/3);
   show(pi/2);
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

```
// Описание функции для вычисления косинуса:
double myCos(double x){
   // Верхняя граница суммы:
   int n=100;
   // Переменные для записи значения суммы
   // и добавки к сумме:
   double s=0,q=1;
   // Вычисление суммы:
   for(int k=0;k<=n;k++){
      s+=q; // Добавка к сумме
      // Добавка для следующей итерации:
      q*=(-1)*x*x/(2*k+1)/(2*k+2);
   }
   // Результат функции:
   return s;
}
// Описание функции для вычисления синуса:
double mySin(double x){
   // Верхняя граница суммы:
   int n=100;
   // Переменные для записи значения суммы
   // и добавки к сумме:
   double s=0,q=x;
   // Вычисление суммы:
   for(int k=0;k<=n;k++){
      s+=q; // Добавка к сумме
      // Добавка для следующей итерации:
      q*=(-1)*x*x/(2*k+2)/(2*k+3);
   }
   // Результат функции:
   return s;
}
```

```
// Описание вспомогательной функции:
void show(double x){
   cout<<"Значение аргумента: "<<x<<endl;
   // Вычисление косинуса:
   cout<<"Косинус: "<<myCos(x)<<" vs. "<<cos(x)<<endl;
   // Вычисление синуса:
   cout<<"Синус: "<<mySin(x)<<" vs. "<<sin(x)<<endl;
   // Дополнительный отступ:
   cout<<endl;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 4.1)

```
Значение аргумента: 0.523599
Косинус: 0.866025 vs. 0.866025
Синус: 0.5 vs. 0.5

Значение аргумента: 0.785398
Косинус: 0.707107 vs. 0.707107
Синус: 0.707107 vs. 0.707107

Значение аргумента: 1.0472
Косинус: 0.5 vs. 0.5
Синус: 0.866025 vs. 0.866025

Значение аргумента: 1.5708
Косинус: 3.26795e-007 vs. 3.26795e-007
Синус: 1 vs. 1
```

В программе описываются функции `myCos()` для вычисления косинуса, `mySin()` для вычисления синуса и вспомогательная функция `show()`. Причем функции сначала объявляются, а непосредственно описание функций выполняется после описания главной функции `main()`.

ПОДРОБНОСТИ

При объявлении функции указывается ее прототип: идентификатор типа результата, название функции и список аргументов, причем названия аргументов можно не указывать — только их тип.
Функция должна быть объявлена до того, как она впервые вызывается.

Функция `myCos()` результатом возвращает значение типа `double` и у нее один аргумент (обозначен как `x`), тоже типа `double`. Локальная целочисленная переменная `n` объявленная и инициализированная со значением `100` в теле функции, определяет верхнюю границу суммы, возвращаемой результатом функции. Еще две переменные типа `double` предназначены для записи значения суммы (переменная `s` с начальным значением `0`) и слагаемого, которое на каждой итерации добавляется к сумме (переменная `q` с начальным значением `1`). Вычисляется сумма с помощью оператора цикла, в котором индексная переменная `k` пробегает значение от `0` до `n` включительно, и за каждый цикл выполняются команды `s+=q` (добавление нового слагаемого к сумме) и `q*=(-1)*x*x/(2*k+1)/(2*k+2)` (вычисление добавки для следующей итерации). По завершении вычислений значение переменной `s` возвращается результатом функции.

НА ЗАМЕТКУ

Для косинуса вычисляется сумма $1 - (x^2/2!) + (x^4/4!) + (x^6/6!) + \ldots + ((-1)^n x^{2n})/((2n)!) = \sum_{k=0}^{n} q_k$, где через $g_k = ((-1)^k x^{2k})/((2k)!)$ обозначена добавка к сумме (для итерации индексом k). Если q_{k+1} — добавка на следующей итерации, то легко заметить, что $q_{k+1}/q_k = ((-1)x^2)/((2k+1)(2k+2))$. Вывод такой: чтобы получить добавку для следующей итерации, текущее значение добавки следует умножить на величину $((-1)x^2)/((2k+1)(2k+2))$. Такой подход реализован в функции `myCos()`.

Для синуса вычисляется сумма $x - (x^3/3!) + (x^5/5!) - (x^7/7!) + \ldots + ((-1)^n x^{2n+1})/((2n+1)!) = \sum_{k=0}^{n} q_k$, где обозначено $q_k = ((-1)^k x^{2k+1})/((2k+1)!)$. Отношение $q_{k+1}/q_k = ((-1)x^2)/((2k+2)(2k+3))$. Поэтому при вычислении синуса (функция `mySin()`) рассчитывая добавку для следующей итерации, значение текущей добавки умножаем на $((-1)x^2)/((2k+2)(2k+3))$.

Функция для вычисления синуса `mySin()` описана аналогично к функции для вычисления косинуса `myCos()`, разве что с поправкой на измененное начальное значение для добавки (переменная `q`) и способ ее обработки (команда `q*=(-1)*x*x/(2*k+2)/(2*k+3)` в теле оператора цикла).

Функция `show()` не возвращает результат, поэтому идентификатором типа результата для нее указано ключевое слово `void`. У функции есть аргумент типа `double`, обозначенный как `x`. Это значение используется при вычислении синуса и косинуса.

Функцией `show()` отображается значение аргумента, для которого вычисляются синус и косинус, значение синуса и косинуса, вычисленные с помощью функций `mySin()` и `myCos()`, а также для сравнения приводятся значения для синуса и косинуса, вычисленные с помощью встроенных функций `sin()` и `cos()`.

> **НА ЗАМЕТКУ**
>
> Для использования встроенных функций `sin()` и `cos()` в программе подключается заголовок `<cmath>`.

Код главной функции программы достаточно простой. В теле функции `main()` определяется действительная числовая константа `pi` со значением для числа $\pi \approx 3{,}141592$, а затем путем вызова функции `show()` вычисляются синус и косинус для значений аргументов $\pi/6$, $\pi/4$, $\pi/3$ и $\pi/2$. Как видим, совпадение со значениями, вычисленными с помощью встроенных функций, более чем приемлемое.

Перегрузка функций

Решим программными методами задачу о вычислении прибыли, получаемой вкладчиком банка. Входными параметрами задачи являются:

- исходная денежная сумма, которую вкладчик размещает в банке на депозитном счету;
- годовая процентная ставка, по которой начисляется доход на вложенную сумму;
- количество лет, на которые размещается депозит;
- количество периодов в году, когда выполняется начисление процентов.

> **ПОДРОБНОСТИ**
>
> Если на депозит вносится сумма *m*, и на вклад начисляется *r* процентов годовых, то через год финальная сумма (доход) будет составлять

величину $m(1 + (r/100))$. Если речь идет о нескольких годах, и за каждый год действует ставка начисления процента r, то через y лет итоговая денежная сумма на депозите будет составлять величину $m(1 + (r/100))^y$.

Выше подразумевалось, что проценты начисляются раз в год. В реальности может использоваться схема, при которой проценты начисляются несколько раз в год. Например, если проценты начисляются n раз в год через равные промежутки времени, то на один такой интервал времени приходится ставка процента r/n, где через r обозначена годовая процентная ставка. При этом за y лет будет ny начислений со ставкой процента r/n. В результате для итоговой суммы вклада имеем $m(1 + (r/(100 \cdot n)))^{n \cdot y}$.

Мы хотим предусмотреть в программе следующие возможности:

- если при вычислении дохода указывается только начальная сумма и процентная ставка, то автоматически считается, что депозит размещается на один год и проценты начисляются тоже один раз в год;
- если при вычислении дохода указывается начальная сумма, процентная ставка и количество лет, то по умолчанию полагаем, что проценты начисляются раз в год;
- также можно при вычислении дохода указать начальную сумму, годовую процентную ставку, количество лет и количество периодов начисления процентов.

Для всех перечисленных случаев мы опишем отдельную функцию, вычисляющую итоговую сумму, причем воспользуемся таким механизмом, как *перегрузка функций*.

ТЕОРИЯ

Перегрузка функций состоит в том, что в программе создается несколько функций с одинаковыми названиями. Такие функции должны отличаться прототипом (тип результата, название функции, количество и тип аргументов) так, чтобы при вызове функции можно было однозначно определить, о какой именно версии функции идет речь. О функциях с одинаковым названием обычно говорят как о разных версиях одной функции.

Программный код с решением данной задачи представлен в листинге 4.2.

Листинг 4.2. Перегрузка функций

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция с двумя аргументами:
double getMoney(double m,double r){
   return m*(1+r/100);
}
// Функция с тремя аргументами:
double getMoney(double m,double r,int y){
   double s=m;
   for(int k=1;k<=y;k++){
      s*=(1+r/100);
   }
   return s;
}
// Функция с четырьмя аргументами:
double getMoney(double m,double r,int y,int n){
   return getMoney(m,r/n,y*n);
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Начальная сумма вклада:
   double money=1000;
   // Процентная ставка:
   double rate=5;
   cout<<"Начальная сумма: "<<money<<endl;
   cout<<"Годовая ставка: "<<rate<<"%\n";
   // Вычисление дохода за разные промежутки времени:
   cout<<"Вклад на один год: "<<getMoney(money,rate)<<endl;
```

```
cout<<"Вклад на 7 лет: "<<getMoney(money,rate,7)<<endl;
cout<<"Вклад на 7 лет\n(начисления 3 раза в год): ";
cout<<getMoney(money,rate,7,3)<<endl;
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 4.2)

```
Начальная сумма: 1000
Годовая ставка: 5%
Вклад на один год: 1050
Вклад на 7 лет: 1407.1
Вклад на 7 лет
(начисления 3 раза в год): 1414.98
```

Проанализируем программный код. Нас в основном будет интересовать описание функции `getMoney()`. В программе описаны три версии данной функции, которые отличаются лишь количеством аргументов, которые передаются функции. Рассмотрим каждый из них в отдельности.

Наиболее простой код у функции `getMoney()` с двумя аргументами. В этом случае `double`-аргумент `m` определяет начальную сумму вклада, а еще одни `double`-аргумент `r` дает значение годовой процентной ставки. Результатом функции возвращается значение выражения `m*(1+r/100)` в полном соответствии с формулой для вычисления итоговой суммы для годичного депозита.

В версии функции с тремя аргументами первые два аргумента `m` и `r` имеют такое же назначение, как и в предыдущем случае, а третий целочисленный аргумент `y` определяет количество лет, на которые размещается депозит. Результат в теле функции вычисляется с помощью оператора цикла. Локальная переменная `s` с начальным значением `m` (первый аргумент функции), которая возвращается результатом функции, за каждый цикл умножается на множитель `(1+r/100)`, и таких циклов ровно `y`. В результате получаем значение, соответствующее формуле вычисле-

ния итоговой суммы депозита, размещенного на несколько лет (с однократным начислением процентов в конце года).

В версии функции `getMoney()` с четырьмя аргументами последний целочисленный аргумент `n` определяет количество периодов в году для начисления процентов. Результатом функции возвращается значение выражение `getMoney(m,r/n,y*n)`. Здесь мы фактически в теле функции с четырьмя аргументами вызываем версию этой же самой функции с тремя аргументами.

> **НА ЗАМЕТКУ**
>
> Далее мы будем рассматривать такой механизм, как **рекурсия**. При рекурсии в теле функции вызывается эта же самая функция. Здесь, в данном примере, о рекурсии речь не идет, поскольку в одной версии функции вызывается другая версия функции. Технически это все равно, как если бы в одной функции вызывалась совершенно другая функция. Просто в данном случае у таких разных по сути функций одинаковые названия.
>
> Что касается выражения `getMoney(m,r/n,y*n)`, которое возвращается результатом для функции `getMoney()` с четырьмя аргументами, то здесь мы воспользовались тем очевидным обстоятельством, что итоговая сумма по депозиту на *y* лет с начальной суммой вклада *m* под *r* процентов годовых с *n*-кратным начислением процентов равна итоговой сумме вклада с начальной суммой *m*, размещенной под ставку процента *r/n* на *ny* лет с однократным начислением процентов.

В главной функции программы представлены примеры вызова функции `getMoney()` с разным количеством аргументов. Каждый раз, когда вызывается функция, имеющая несколько версий, решение о задействовании той или иной версии функции принимается на основе команды вызова: в основном по количеству и типу аргументов, а также по типу результата.

Значения аргументов по умолчанию

Рассмотренную выше задачу можно было решить и несколько иначе. В частности, в данном конкретном случае вместо перегрузки функций мы вполне могли ограничиться созданием всего одной функции, но с аргументами, имеющими *значения по умолчанию*.

ТЕОРИЯ

При описании функции для некоторых или всех ее аргументов можно указать **значения по умолчанию**. Данные значения используются, если при вызове функции соответствующий аргумент не указан. Аргументы, имеющие значения по умолчанию, описываются в списке аргументов функции последними. Значение по умолчанию для аргумента указывается в прототипе функции после имени аргумента через знак равенства.

В листинге 4.3 представлена программа, в которой решается поставленная ранее задача, но только теперь мы создаем всего одну функцию `getMoney()`, а два ее последних аргумента имеют значение по умолчанию.

Листинг 4.3. Значения аргументов по умолчанию

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция с аргументами, имеющими
// значения по умолчанию:
double getMoney(double m,double r,int y=1,int n=1){
   double s=m;
   double z=n*y;
   double q=r/n;
   for(int k=1;k<=z;k++){
      s*=(1+q/100);
   }
   return s;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Начальная сумма вклада:
   double money=1000;
```

```
    // Процентная ставка:
    double rate=5;
    cout<<"Начальная сумма: "<<money<<endl;
    cout<<"Годовая ставка: "<<rate<<"%\n";
    // Вычисление дохода за разные промежутки времени:
    cout<<"Вклад на один год: "<<getMoney(money,rate)<<endl;
    cout<<"Вклад на 7 лет: "<<getMoney(money,rate,7)<<endl;
    cout<<"Вклад на 7 лет\n(начисления 3 раза в год): ";
    cout<<getMoney(money,rate,7,3)<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой же, как и в предыдущем случае:

Результат выполнения программы (из листинга 4.3)

```
Начальная сумма: 1000
Годовая ставка: 5%
Вклад на один год: 1050
Вклад на 7 лет: 1407.1
Вклад на 7 лет
(начисления 3 раза в год): 1414.98
```

Когда функция `getMoney()` вызывается с тремя аргументами, это все равно, как если бы четвертый ее аргумент был равен единице (значение по умолчанию). Если функция вызывается только с двумя аргументами, то третий и четвертый ее аргументы автоматически считаются единичными (значения по умолчанию для данных аргументов). Что касается кода, определяющего функцию, то локальная переменная `s` с начальным значением `m` определяет стартовый вклад, переменная `z` со значением `n*y` определяет общее количество периодов начисления процента, переменная `q` со значением `r/n` определяет процентную ставку процента для каждого периода, за который начисляются проценты. Результат вычисляется с помощью оператора цикла, в котором `z` циклов и за каждый

цикл переменная s умножается на (1+q/100). По завершении вычислений значение переменной s возвращается результатом функции. Здесь, по большому счету, мы повторили алгоритм вычислений, использованный в предыдущем примере для случая функции с тремя аргументами. Единственное, мы сделали «поправку» на эффективную процентную ставку и количество периодов начисления процентного дохода.

(i) НА ЗАМЕТКУ

На первый взгляд может показаться, что механизм определения значений аргументов по умолчанию более прост в использовании, чем перегрузка функций. В принципе, это так. Однако следует понимать, что перегрузка функций — технология исключительно гибкая и эффективная, и не может быть заменена определением значений аргументов по умолчанию.

Также важно иметь в виду, что и перегрузка функций, и определение значений по умолчанию для аргументов могут использоваться одновременно. В таком случае функции должны перегружаться так, чтобы по контексту вызова функции всегда можно было однозначно определить вариант функции и то, какие аргументы переданы, а какие — нет.

Рекурсия

Если в описании функции вызывается эта самая функция (но, как правило, с другим аргументом или аргументами), то такая ситуация называется *рекурсией*. Пример использования рекурсии представлен в листинге 4.4. Речь идет все о той же задаче про вычисление итоговой суммы депозита, но только теперь для простоты мы рассматриваем не три, а один способ начисления процентов: депозит размещается на указанное количество лет под фиксированную годовую ставку процента, а начисления дохода по процентам производятся раз в год. В программе описана функция getMoney() с тремя аргументами, и при описании функции использована рекурсия. Рассмотрим представленный ниже программный код:

Листинг 4.4. Рекурсия

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

```
// В функции используется рекурсивный вызов:
double getMoney(double m,double r,int y){
   if(y==0){
      return m;
   }
   else{
      return (1+r/100)*getMoney(m,r,y-1);
   }
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Начальная сумма вклада:
   double money=1000;
   // Процентная ставка:
   double rate=5;
   cout<<"Начальная сумма: "<<money<<endl;
   cout<<"Годовая ставка: "<<rate<<"%\n";
   // Вычисление дохода за разные промежутки времени:
   cout<<"Вклад на 1 год: "<<getMoney(money,rate,1)<<endl;
   cout<<"Вклад на 7 лет: "<<getMoney(money,rate,7)<<endl;
   cout<<"Вклад на 10 лет: "<<getMoney(money,rate,10)<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы будет следующим:

Результат выполнения программы (из листинга 4.4)

```
Начальная сумма: 1000
Годовая ставка: 5%
```

```
Вклад на 1 год: 1050
Вклад на 7 лет: 1407.1
Вклад на 10 лет: 1628.89
```

В программе описывается функция `getMoney()` с тремя аргументами, после чего в главной функции программы функция `getMoney()` вызывается с разными значениями третьего аргумента.

Интерес представляет способ описания функции `getMoney()`. Основу кода функции составляет условный оператор, в котором проверяется условие `y==0` (через `y` обозначен третий аргумент функции `getMoney()`). Если условие истинно, результатом функции возвращается значение `m` (первый аргумент функции `getMoney()`). Здесь мы исходим из обстоятельства, что на вклад, размещенный в банке на `0` лет, проценты не начисляются. Поэтому итоговая сумма в такой ситуации равняется начальной сумме вклада. Если же условие `y==0` ложно, то результатом функции возвращается значение выражения `(1+r/100)*getMoney(m,r,y-1)`. Это и есть рекурсивный вызов функции `getMoney()`.

ПОДРОБНОСТИ

При определении функции мы исходили из следующих соображений. Допустим, что `getMoney(m,r,y)` дает значение итоговой суммы по депозиту через `y` лет. Как можно получить эту сумму? Очень просто: начислить годовые проценты на итоговую сумму вклада за период в `y-1` лет. Итоговая сумма вклада за `y-1` лет дается выражением `getMoney(m,r,y-1)`. Начисление годовых процентов означает умножение на величину `(1+r/100)`. В итоге получаем, что значения `getMoney(m,r,y)` и `(1+r/100)*getMoney(m,r,y-1)` должны совпадать.

Значение функции `getMoney()` с заданными аргументами `m`, `r` и `y` (значение выражения `getMoney(m,r,y)`) вычисляется следующим образом. Проверяется значение аргумента `y`, и если оно отлично от нуля, вычисляется значение выражения `(1+r/100)*getMoney(m,r,y-1)`. При вычислении значения выражения снова вызывается функция `getMoney()`, но на этот раз третий аргумент на единицу меньше, и так далее, пока не будет вызвана функция `getMoney()` с нулевым третьим аргументом. Тогда при вызове функции возвращается значение `m` (первый аргумент функции). Это позволяет вычислить значение предыдущего выражения, затем того, что было перед ним, и так далее, вплоть до самого первого выражения, с которого все начиналось.

Механизмы передачи аргументов функциям

Чтобы понять природу проблемы, которая будет обсуждаться, рассмотрим небольшой пример, представленный в листинге 4.5. Суть программы очень проста:

- описывается функция `swap()` с двумя символьными (тип `char`) аргументами;
- при вызове функции значения переданных аргументов выводятся в консольное окно;
- аргументы обмениваются значениями, и их новые значения также отображаются в консольном окне.

В программе объявляется две переменные, они передаются аргументами функции `swap()` при ее вызове, а затем проверяются значения переменных.

Интересующий нас программный код такой:

Листинг 4.5. Передача аргументов по значению

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Передача аргументов по значению:
void swap(char a,char b){
   cout<<"Вызывается функция swap()"<<endl;
   // Проверяются значения аргументов функции:
   cout<<"Первый аргумент: "<<a<<endl;
   cout<<"Второй аргумент: "<<b<<endl;
   // Изменение значений аргументов функции:
   char t=b;
   b=a;
   a=t;
   for(int i=1;i<=20;i++){
      cout<<"-";
   }
```

```
    cout<<endl;
    // Проверяются значения аргументов функции:
    cout<<"Первый аргумент: "<<a<<endl;
    cout<<"Второй аргумент: "<<b<<endl;
    cout<<"Выполнение функции swap() завершено"<<endl;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Переменные для передачи аргументами функции:
    char x='A',y='B';
    // Проверяются значения переменных:
    cout<<"Первая переменная: "<<x<<endl;
    cout<<"Вторая переменная: "<<y<<endl;
    // Вызов функции:
    swap(x,y);
    // Проверяются значения переменных:
    cout<<"Первая переменная: "<<x<<endl;
    cout<<"Вторая переменная: "<<y<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Наши ожидания от результатов выполнения программы могли бы быть следующими: после вызова функции `swap()` переменные, которые передаются ей аргументами, обменяются значениями. В реальности результат выполнения программы такой, как представлено ниже:

Результат выполнения программы (из листинга 4.5)

```
Первая переменная: A
Вторая переменная: B
Вызывается функция swap()
```

```
Первый аргумент: A
Второй аргумент: B
--------------------
Первый аргумент: B
Второй аргумент: A
Выполнение функции swap() завершено
Первая переменная: A
Вторая переменная: B
```

Как видим, при проверке значений аргументов в теле функции все идет по плану: аргументы обменялись значениями. Но вот когда мы проверяем значения переменных после вызова функции `swap()` оказывается, что их значения не изменились. Причина кроется в способе передачи аргументов функции. По умолчанию используется режим передачи аргументов *по значению*: передаются не сами переменные, а их копии (после завершения выполнения функции копии аргументов удаляются из памяти). Поэтому при вызове функции `swap()` все операции выполняются не с переменными `x` и `y`, которые передавались аргументами функции, а с их копиями. Именно копии обмениваются своими значениями. Значения переменных `x` и `y` остаются неизменными.

Помимо передачи аргументов по значению, аргументы передаются еще и *по ссылке*. В этом случае в функцию передаются не копии переменных, а сами переменные. Чтобы передавать аргументы по ссылке, в описании функции перед названиями аргументов указывается инструкция `&`. В листинге 4.6 функция `swap()` описана так, что аргументы ей передаются по ссылке. По сравнению с программой из листинга 4.5 изменения минимальные (для удобства в представленном далее коде удалены почти все комментарии, а инструкции `&` в описании функции `swap()` выделены жирным шрифтом):

Листинг 4.6. Передача аргументов по ссылке

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Передача аргументов по ссылке:
void swap(char &a,char &b){
   cout<<"Вызывается функция swap()"<<endl;
```

```
    cout<<"Первый аргумент: "<<a<<endl;
    cout<<"Второй аргумент: "<<b<<endl;
    char t=b;
    b=a;
    a=t;
    for(int i=1;i<=20;i++){
        cout<<"-";
    }
    cout<<endl;
    cout<<"Первый аргумент: "<<a<<endl;
    cout<<"Второй аргумент: "<<b<<endl;
    cout<<"Выполнение функции swap() завершено"<<endl;
}
int main(){
    system("chcp 1251>nul");
    char x='A',y='B';
    cout<<"Первая переменная: "<<x<<endl;
    cout<<"Вторая переменная: "<<y<<endl;
    swap(x,y);
    cout<<"Первая переменная: "<<x<<endl;
    cout<<"Вторая переменная: "<<y<<endl;
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 4.6)

```
Первая переменная: A
Вторая переменная: B
Вызывается функция swap()
Первый аргумент: A
Второй аргумент: B
--------------------
```

```
Первый аргумент: B
Второй аргумент: A
Выполнение функции swap() завершено
Первая переменная: B
Вторая переменная: A
```

Как видим, переменные на самом деле изменили свои значения после передачи аргументами в функцию `swap()`.

Передача указателя аргументом функции

Еще один способ описания функции `swap()`, но на этот раз аргументами функции передаются указатели. Рассмотрим программу, представленную в листинге 4.7.

Листинг 4.7. Передача аргументом функции указателя

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Аргументами функции передаются указатели:
void swap(char* a,char* b){
   cout<<"Вызывается функция swap()"<<endl;
   // Проверяются значения аргументов функции:
   cout<<"Первая переменная: "<<*a<<endl;
   cout<<"Вторая переменная: "<<*b<<endl;
   // Изменение значений переменных:
   char t=*b;
   *b=*a;
   *a=t;
   for(int i=1;i<=20;i++){
      cout<<"-";
   }
   cout<<endl;
   // Проверяются значения переменных:
   cout<<"Первая переменная: "<<*a<<endl;
```

```
    cout<<"Вторая переменная: "<<*b<<endl;
    cout<<"Выполнение функции swap() завершено"<<endl;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Переменные для передачи аргументами функции:
    char x='A',y='B';
    // Проверяются значения переменных:
    cout<<"Первая переменная: "<<x<<endl;
    cout<<"Вторая переменная: "<<y<<endl;
    // Вызов функции:
    swap(&x,&y);
    // Проверяются значения переменных:
    cout<<"Первая переменная: "<<x<<endl;
    cout<<"Вторая переменная: "<<y<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы приведен ниже:

Результат выполнения программы (из листинга 4.7)

```
Первая переменная: A
Вторая переменная: B
Вызывается функция swap()
Первая переменная: A
Вторая переменная: B
--------------------
Первая переменная: B
Вторая переменная: A
Выполнение функции swap() завершено
```

```
Первая переменная: B
Вторая переменная: A
```

В программе в описании функции `swap()` тип обеих аргументов указан как `char*`. Это означает, что аргументами функции передаются указатели на символьные значения. Поэтому когда функция `swap()` вызывается в главной функции программы, команды вызова имеет вид `swap(&x,&y)`. Поскольку переменные `x` и `y` объявлены как относящиеся к типу `char`, то `&x` и `&y` (адреса переменных) — это указатели на значения типа `char`. Другими словами, если в предыдущих примерах в функцию передавались переменные (или их копии), то на этот раз аргументами функции передаются указатели на переменные.

Итак, аргументами функции `swap()` передаются указатели. Но проверяются значения, которые записаны по соответствующим адресам. И обмен значениями происходит не для указателей, а для значений, которые записаны по адресам, являющимся значениями указателей. Так, командой `char t=*b` объявляется локальная переменная `t` и ей присваивается значение, записанное по адресу из указателя `b` (второй аргумент функции). Далее командой `*b=*a` по адресу из указателя `b` записывается значение, размещенное по адресу из указателя `a` (первый аргумент функции). После этого командой `*a=t` по адресу, содержащемуся в указателе `a`, записывается значение переменной `t` (именно это значение в самом начале было записано по адресу из указателя `b`).

Важно понимать, что, когда мы передаем аргументами функции указатели, они передаются *по значению*. То есть на самом деле создается копия аргумента, который передается функции. Но специфика ситуации в том, что копия указателя содержит то же значение, что и исходный указатель, а поэтому копия указателя установлена на ту же ячейку памяти, что и оригинальный указатель. В теле функции, в свою очередь, операции (по изменению значения) выполняются не с указателями, а с ячейками, адреса которых записаны в указателях. Так что механизм передачи аргументов по умолчанию в данном примере принципиальной роли не играет.

Передача массива аргументом функции

Нередко приходится передавать аргументом функции массив. Как это делается в случае одномерного массива, иллюстрирует программа из листинга 4.8.

ТЕОРИЯ

Одномерный массив передается в функцию с помощью двух параметров: указателя на первый элемент массива (имя массива) и целого числа, определяющего количество элементов в массиве.

В представленной ниже программе описывается функция `mean()`, аргументом которой передается числовой массив, а результатом возвращается среднее значение элементов этого массива. В главной функции программы функция `mean()` вызывается для вычисления среднего значения элементов для двух массивов.

Листинг 4.8. передача аргументом функции одномерного массива

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Передача функции аргументом массива:
double mean(double* m,int n){
   // Локальная переменная для записи суммы
   // значений элементов массива:
   double s=0;
   // Вычисление суммы значений элементов массива:
   for(int k=0;k<n;k++){
      s+=m[k];
   }
   // Результат функции – среднее значение:
   return s/n;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Первый массив:
   double A[]={1,3,8,-2,4};
   // Второй массив:
```

```
double B[]={4,6,2};
// Вызов функции с передачей аргументом массива:
cout<<"Среднее по массиву A: "<<mean(A,5)<<endl;
cout<<"Среднее по массиву B: "<<mean(B,3)<<endl;
// Задержка консольного окна:
system("pause>nul");
   return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 4.8)

```
Среднее по массиву A: 2.8
Среднее по массиву B: 4
```

Функция `mean()` описана с двумя аргументами. Первый аргумент `m` с идентификатором типа `double*` формально является указателем на значение типа `double`, но учитывая, что имя массива является указателем на первый элемент массива, мы вполне можем отождествлять аргумент `m` с именем массива. Именно так переменная `m` обрабатывается в теле функции. Второй целочисленный аргумент `n` функции определяет размер массива.

В теле функции вычисляется сумма значений элементов массива, и полученное значение записывается в локальную переменную `s`. Результатом функции возвращается значение `s/n` (сумма значений элементов, деленная на количество элементов — то есть среднее значение).

В главной функции программы командами `double A[]={1,3,8,-2,4}` и `double B[]={4,6,2}` создаются и инициализируются два числовых массива.

НА ЗАМЕТКУ

При объявлении массивов размер не указывается. Он определяется автоматически по количеству элементов, указанных в списке инициализации массива.

При вызове функции `mean()` аргументами передаются имя массива и размер массива: например, `mean(A,5)` или `mean(B,3)`.

Аналогичным образом в функцию передается двумерный массив. В листинге 4.9 представлен программный код, в котором описывается функция `show()`. Аргументом функции передается двумерный массив, значения элементов которого отображаются в консольном окне.

ТЕОРИЯ

Если речь идет о статическом двумерном массиве, то он обрабатывается как одномерный массив (назовем его внешним массивом), элементами которого являются одномерные массивы (назовем их внутренними массивами). Здесь есть один принципиальный момент: каждый такой внутренний массив-элемент имеет размер, который является неотъемлемой характеристикой типа для элемента внешнего массива. Проще говоря, мало указать, что внешний массив состоит из внутренних массивов. Необходимо конкретизировать размер этих внутренних массивов. При передаче двумерного массива в функцию реально аргументами передают две характеристики: название двумерного массива с явным указанием размера по второму индексу, а также целочисленное значение, определяющее размер двумерного массива по первому индексу (размер внешнего массива).

Рассмотрим программный код:

Листинг 4.9. Передача двумерного массива в функцию

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Размер массива по второму индексу:
const int n=3;
// Функция для обработки двумерного
// статического массива:
void show(int M[][n],int p){
   for(int i=0;i<p;i++){
      for(int j=0;j<n;j++){
         cout<<M[i][j]<<" ";
      }
      cout<<endl;
```

```
    }
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Первый массив:
    int A[2][n]={{1,2,3},{4,5,6}};
    // Второй массив:
    int B[][n]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
    cout<<"Первый массив:\n";
    show(A,2);
    cout<<"Второй массив:\n";
    show(B,4);
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 4.9)

```
Первый массив:
1 2 3
4 5 6
Второй массив:
11 12 13
14 15 16
17 18 19
20 21 22
```

В программе для определения размера двумерного массива по второму индексу объявляется целочисленная константа `n`. При этом, хотя данная константа доступна в функции `show()`, ее первый аргумент все равно описан выражением `int M[][n]` с явным указанием размера массива

по второму индексу. Таким образом, функция `show()` применима для отображения содержимого массивов, размер которых по второму индексу равен `n`. Второй аргумент функции `show()` отождествляет размер двумерного массива по первому индексу. В главной функции программы функция `show()` используется для отображения содержимого массивов `A` и `B`, созданных и инициализированных командами `int A[2][n]={{1,2,3},{4,5,6}}` и `int B[][n]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}}`. У этих массивов размер по второму индексу равен `n`, а размеры по первому индексу различны. Причем для массива `A` размер по первому индексу указан явно, а размер по первому индексу для массива `B` (поскольку при его описании первая квадратная скобка пустая) определяется автоматически в соответствии с содержимым списка инициализации для данного массива. При вызове функции `show()` первым аргументом передается имя массива, а второй аргумент — размер массива по первому индексу.

Несколько иначе все происходит, когда в функцию необходимо передать двумерный динамический массив.

ТЕОРИЯ

Двумерный динамический массив по факту представляет собой одномерный массив из указателей на указатели. Каждый элемент-указатель такого массива содержит адрес первого элемента другого одномерного массива. Таким образом, передача двумерного динамического массива в функцию сводится к передаче одномерного массива (два параметра: указатель на первый элемент массива и размер массива), плюс еще один аргумент, который определяет размеры внутренних массивов. В итоге обычно ограничиваются передачей трех аргументов: один является указателем на указатель, и два целочисленных аргумента.

В листинге 4.10 представлена программа, в которой, как и в предыдущем случае, описывается функция `show()`, но на этот раз она предназначена для отображения содержимого двумерного динамического массива.

НА ЗАМЕТКУ

Для выполнения форматированного вывода при работе с числовыми значениями мы используем функцию `printf()`. Соответственно, в программе подключается заголовок `<cstdio>` и не подключается заголовок `<iostream>`.

По сравнению с предыдущим примером (см. листинг 4.9), здесь изменен способ описания аргументов функции `show()`, ну и сложнее (по сравнению со статическим массивом) выглядит процедура создания двумерного динамического массива в главной функции программы:

Листинг 4.10. Передача двумерного динамического массива в функцию

```
#include <cstdio>
#include <cstdlib>
using namespace std;
// Функции аргументом передается двумерный
// динамический массив:
void show(int** M,int p,int n){
   for(int i=0;i<p;i++){
      for(int j=0;j<n;j++){
         printf("%4d",M[i][j]);
      }
      printf("\n");
   }
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Размеры динамического массива
   // и индексные переменные:
   int a=3,b=5,i,j;
   // Создание массива указателей:
   int** A=new int*[a];
   // Создание и заполнение внутренних массивов:
   for(i=0;i<a;i++){
      A[i]=new int[b];
      for(j=0;j<b;j++){
         // Значение элемента массива:
```

```
            A[i][j]=i*b+j+1;
        }
    }
    printf("Содержимое массива:\n");
    // Отображение содержимого массива:
    show(A,a,b);
    // Удаление внутренних массивов:
    for(i=0;i<a;i++){
        delete [] A[i];
    }
    // Удаление массива указателей:
    delete [] A;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

При выполнении программы получаем следующий результат:

Результат выполнения программы (из листинга 4.10)

```
Содержимое массива:
    1    2    3    4    5
    6    7    8    9   10
   11   12   13   14   15
```

Если сравнивать различие в способах передачи в функцию двумерного статического и динамического массивов, то с динамическим массивом нет жесткой привязки к размеру массива по второму индексу. В рассматриваемом примере функция `show()` применима для отображения содержимого любых (в смысле любых размеров) двумерных динамических массивов.

Передача текста в функцию

Символьный одномерный массив с текстом в принципе передается аргументом функции так же, как и любой другой одномерный массив, с поправкой на то, что размер массива явно указывать не нужно — он

определяется автоматически по расположению нуль-символа в массиве (точнее, автоматически определяется размер текста в массиве).

ТЕОРИЯ

При передаче символьного массива с текстом в функцию аргументом функции достаточно передать указатель на первый элемент массива (имя массива).

Примеры описания функций, в которые передаются символьные одномерные массивы, приведены в программе в листинге 4.11.

Листинг 4.11. Передача в функцию символьного массива

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция для определения длины строки:
int getLength(char* str){
   int s=0;
   for(int i=0;str[i];i++){
      s++;
   }
   return s;
}
// Функция для определения количества пробелов в строке:
int getSpace(char* str){
   int s=0;
   for(int i=0;str[i];i++){
      if(str[i]==' '){
         s++;
      }
   }
   return s;
}
// Функция для отображения строки и некоторых
```

```
// дополнительных характеристик:
void show(char* str){
   cout<<"Фраза: "<<str<<endl;
   cout<<"Символов: "<<getLength(str)<<endl;
   cout<<"Пробелов: "<<getSpace(str)<<endl;
   for(int k=1;k<=35;k++){
      cout<<"-";
   }
   cout<<endl;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Символьный массив:
   char txt[100]="Изучаем язык программирования C++";
   // Передача аргументом функции символьного массива:
   show(txt);
   // Передача аргументом функции текстового литерала:
   show("В C++ есть классы и объекты");
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже представлен результат выполнения данной программы:

Результат выполнения программы (из листинга 4.11)

```
Фраза: Изучаем язык программирования C++
Символов: 33
Пробелов: 3
-----------------------------------
Фраза: В C++ есть классы и объекты
```

```
Символов: 27
Пробелов: 5
--------------------------------
```

В программе описано несколько функций, предназначенных для обработки текстов (переданных в функцию через символьный массив или текстовый литерал). В частности, мы описали функцию `getLength()`, возвращающую результатом количество символов в тексте. Функция `getSpace()` возвращает результатом количество пробелов в тексте. Функция `show()` описана так, что при вызове отображает текст, переданный ей аргументом, а также через вызов функций `getLength()` и `getSpace()` отображает информацию о длине текстовой строки и количестве пробелов в ней. Всем трем функциям аргументом передается указатель на символьное значение. Под этим указателем имеется в виду имя символьного массива. Также функциям аргументом может быть передан непосредственно текстовый литерал (поскольку текстовый литерал обрабатывается через указатель на первый символ в тексте). В главной функции программы есть команды с вызовом функции `show()`.

Указатель как результат функции

Функция результатом может возвращать практически все, что угодно — за исключением статического массива. В частности, результатом функции может быть указатель. В листинге 4.12 представлена программа, в которой описывается функция `getMax()`. В функцию передается числовой одномерный массив, а результатом функция возвращает указатель на элемент с самым большим значением в массиве. Рассмотрим представленный ниже программный код:

Листинг 4.12. Функция возвращает результатом указатель

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция результатом возвращает
// указатель на элемент массива:
int* getMax(int* nums,int n){
   int i=0,k;
```

```
    // Определение индекса наибольшего элемента:
    for(k=0;k<n;k++){
        if(nums[k]>nums[i]){
            i=k;
        }
    }
    // Результат функции – указатель на элемент:
    return nums+i;
}
// Функция для отображения содержимого массива:
void show(int* nums,int n){
    for(int i=0;i<n;i++){
        cout<<nums[i]<<" ";
    }
    cout<<endl;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Размер массива:
    const int size=10;
    // Создание массива:
    int numbers[size]={1,5,8,2,4,9,11,9,12,3};
    // Отображение содержимого массива:
    show(numbers,size);
    // Запись результат функции в указатель:
    int* maxPnt=getMax(numbers,size);
    // Отображение максимального значения:
    cout<<"Максимальное значение: "<<*maxPnt<<endl;
    // Присваивание значения элементу:
    *maxPnt=-100;
```

```
// Отображение содержимого массива:
show(numbers,size);
// Значение присваивается переменной:
int maxNum=*getMax(numbers,size);
// Проверка максимального значения:
cout<<"Максимальное значение: "<<maxNum<<endl;
// Присваивание значения переменной:
maxNum=-200;
// Проверка содержимого массива:
show(numbers,size);
cout<<"Максимальное значение: ";
// Вычисление нового наибольшего значения:
cout<<*getMax(numbers,size)<<endl;
cout<<"Индекс элемента: ";
// Вычисление индекса элемента с наибольшим значением:
   cout<<getMax(numbers,size)-numbers<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 4.12)

```
1 5 8 2 4 9 11 9 12 3
Максимальное значение: 12
1 5 8 2 4 9 11 9 -100 3
Максимальное значение: 11
1 5 8 2 4 9 11 9 -100 3
Максимальное значение: 11
Индекс элемента: 6
```

Идентификатором типа результата для функции `getMax()` указана инструкция `int*`, означающая, что результатом функции является

указатель на целочисленное значение. В теле функции целочисленная переменная `i` инициализирована с начальным нулевым значением. Далее запускается оператор цикла (индексная переменная `k` пробегает все возможные значения для индексов массива), в котором последовательно сравниваются значения элементов массива со значением элемента, имеющем индекс `i`. Если выполнено условие `nums[k]>nums[i]`, то значение индекса `k` присваивается переменной `i`. В результате после перебора всех элементов массива в переменную `i` будет записан индекс элемента с наибольшим значением (или индекс первого встретившегося элемента с наибольшим значением, если таких элементов несколько). Результатом функции возвращается значение `nums+i`. Здесь мы воспользовались правилами адресной арифметики и тем, что имя массива является указателем на его первый элемент.

В функции `main()` создается целочисленная константа `size`, определяющая размер массива, а сам массив создается командой `int numbers[size]={1,5,8,2,4,9,11,9,12,3}`. Отображение содержимого массива выполняется с помощью специально описанной для такой цели функции `show()`.

При выполнении команды `int* maxPnt=getMax(numbers,size)` в указатель `maxPnt` записывается адрес того элемента в массиве `number`, который имеет наибольшее значение. Узнать данное значение можем через выражение `*maxPnt`. При этом `maxPnt` является адресом (но не индексом!) данного элемента. Чтобы узнать индекс элемента, необходимо вычесть из указателя на этот элемент указатель на первый элемент массива (имя массива).

После выполнения команды `*maxPnt=-100` элемент массива, имевший до этого наибольшее значение, получит новое значение `-100`.

Если нас интересует значение элемента, а не его адрес, то можно при вызове функции `getMax()` указать перед ее именем звездочку `*` (что означает получение значения по адресу), как делается в инструкции `int maxNum=*getMax(numbers,size)`. В результате в переменную `maxNum` записывается значение наибольшего элемента, но при этом информация об адресе элемента не запоминается. Поэтому если мы впоследствии присвоим переменной `maxNum` новое значение, на исходном массиве это никак не скажется. Наконец, для вычисления значения индекса элемента с наибольшим значением вычисляем выражение `getMax(numbers,size)-numbers` (разница указателей дает целое число, определяющее количество ячеек между соответствующими областями памяти — для элемен-

тов массива полученное таким образом значение совпадает с индексом элемента).

Ссылка как результат функции

Функция результатом может возвращать ссылку. Если функция возвращает ссылку, то ее результатом является не просто значение некоторого выражения или переменной, а фактически сама переменная.

> **НА ЗАМЕТКУ**
>
> Понятно, что в таких случаях речь не может идти о локальной переменной — нет смысла давать доступ к переменной, которая существует, только пока выполняется функция.

Небольшая вариация на тему предыдущего примера, но на этот раз с возвращением функцией ссылки вместо указателя, приведена в листинге 4.13.

Листинг 4.13. Функция возвращает результатом ссылку

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция результатом возвращает
// ссылку на элемент массива:
int &getMax(int* nums,int n){
   int i=0,k;
   // Определение индекса наибольшего элемента:
   for(k=0;k<n;k++){
      if(nums[k]>nums[i]){
         i=k;
      }
   }
   // Результат функции – ссылка на элемент:
   return nums[i];
}
```

```
// Функция для отображения содержимого массива:
void show(int* nums,int n){
    for(int i=0;i<n;i++){
        cout<<nums[i]<<" ";
    }
    cout<<endl;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Размер массива:
    const int size=10;
    // Создание массива:
    int numbers[size]={1,5,8,2,4,9,11,9,12,3};
    // Отображение содержимого массива:
    show(numbers,size);
    // Запись результат функции в переменную:
    int maxNum=getMax(numbers,size);
    // Отображение максимального значения:
    cout<<"Максимальное значение: "<<maxNum<<endl;
    // Присваивание значения переменной:
    maxNum=-100;
    // Отображение содержимого массива:
    show(numbers,size);
    // Результат функции записывается в ссылку:
    int &maxRef=getMax(numbers,size);
    // Проверка максимального значения:
    cout<<"Максимальное значение: "<<maxRef<<endl;
    // Присваивание значения ссылке:
    maxRef=-200;
    // Проверка содержимого массива:
```

```
show(numbers,size);
cout<<"Максимальное значение: ";
// Вычисление нового наибольшего значения:
cout<<getMax(numbers,size)<<endl;
// Задержка консольного окна:
system("pause>nul");
   return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 4.13)

```
1 5 8 2 4 9 11 9 12 3
Максимальное значение: 12
1 5 8 2 4 9 11 9 12 3
Максимальное значение: 12
1 5 8 2 4 9 11 9 -200 3
Максимальное значение: 11
```

Посмотрим, что принципиально изменилось в программном коде (по сравнению с листингом 4.12.). В первую очередь, конечно, это результат функции `getMax()`: теперь функция возвращает не указатель, а ссылку, о чем свидетельствует инструкция `&` перед именем функции в ее описании. Поскольку функция описана с идентификатором `int`, то речь идет о ссылке на целочисленное значение. В теле функции, для числового массива `nums`, переданного в функцию, возвращается ссылка на элемент с наибольшим значением. Формально команда возвращения функцией результата выглядит как `return nums[i]`, что в принципе означает возвращение результатом значения элемента с индексом `i` из массива `nums`. Но поскольку в данном случае в описании функции перед ее именем стоит инструкция `&`, то возвращается не просто значение элемента, а *ссылка* на этот элемент. С прикладной точки зрения данное утверждение означает, что через результат функции `getMax()` мы можем не только прочитать значение наибольшего элемента, но и присвоить ему новое значение.

В функции `main()` создается целочисленным массив `number` размера `size`, и именно с этим массивом выполняются операции с привлечением функции `getMax()`. Так, при выполнении команды

`int maxNum=getMax(numbers,size)` в переменную `maxNum` записывается значение наибольшего элемента массива. Дальнейшее изменение значения переменной `maxNum` (как, например, при выполнении команды `maxNum=-100`) на массиве `numbers` не сказываются — значения его элементов остаются неизменными. Но если мы результат функции запишем не в обычную переменную, а в ссылку (примером может служить команда `int &maxRef=getMax(numbers,size)`), то эта ссылка станет альтернативным названием для соответствующего элемента массива (того элемента, на который функция возвращает ссылку). Поэтому, скажем, выполнение команды `maxRef=-200` означает, что соответствующий элемент массива получает новое значение.

Динамический массив как результат функции

Теоретически можно описать функцию, которая результатом возвращает динамический массив. Точнее, она возвращает указатель на первый элемент массива, а сам массив создается при вызове функции. Правда, это достаточно опасный способ программирования (почему — объясняется далее). Вместе с тем некоторые специфические моменты общего характера он позволяет продемонстрировать, поэтому рассмотрим его.

В листинге 4.14 представлена программа, в которой описаны две функции, каждая из которых возвращает результатом динамический массив: в случае функции `fibs()` это массив, заполненный числами Фибоначчи, а функцией `myrand()` возвращается целочисленный массив, заполненный случайными числами в диапазоне значений от `0` до `9`.

НА ЗАМЕТКУ

В последовательности чисел Фибоначчи первые два числа равны единице, а каждое следующее число равняется сумме двух предыдущих. В итоге получаем: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 и так далее.

Каждой из этих функций при вызове аргументом передается целое число, определяющее размер массива. Друг от друга функции принципиально отличаются только способом заполнения элементов, а в остальном они практически идентичны.

Рассмотрим следующий программный код:

Листинг 4.14. Функция возвращает результатом динамический массив

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Результат функции – динамический массив
// с числами Фибоначчи:
int* fibs(int n){
   int* nums=new int[n];
   for(int i=0;i<n;i++){
      if(i==0||i==1){
         nums[i]=1;
      }
      else{
         nums[i]=nums[i-1]+nums[i-2];
      }
   }
   return nums;
}
// Результат функции – динамический массив
// со случайными числами:
int* myrand(int m){
   int* nums=new int[m];
   for(int i=0;i<m;i++){
      nums[i]=rand()%10;
   }
   return nums;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
```

```
// Переменные:
int n=10,m=15,i;
// Указатель на целочисленное значение:
int* f;
// Создается динамический массив:
f=fibs(n);
// Отображение значений элементов
// динамического массива:
for(i=0;i<n;i++){
   cout<<f[i]<<" ";
}
cout<<endl;
// Удаление динамического массива:
delete [] f;
// Новый динамический массив:
f=myrand(m);
// Отображение содержимого массива:
for(i=0;i<m;i++){
   cout<<f[i]<<" ";
}
cout<<endl;
// Удаление динамического массива:
delete [] f;
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Результат выполнения программы, с поправкой на то, что второй массив заполняется случайными числами, может выглядеть так:

Результат выполнения программы (из листинга 4.14)

```
1 1 2 3 5 8 13 21 34 55
5 6 8 5 4 0 0 1 5 6 7 9 2 6 3
```

Проанализируем наиболее интересные места программы. Скажем, функция `fibs()` формально возвращает указатель на целое число (ее идентификатор типа определен как `int*`). В теле функции командой `int* nums=new int[n]` создается динамический массив (через `n` обозначен целочисленный аргумент функции). Массив заполняется с помощью оператора цикла. В нем, в свою очередь, есть условный оператор, в котором проверяется значение индекса элемента (в условии `i==0||i==1` использован оператор `||` *логического или*, которым возвращается значение `true`, если хотя бы один из его операндов равен `true`). Для нулевого и единичного индекса соответствующему элементу присваивается единичное значение. Для элементов с прочими индексами значение вычисляется как сумма значений двух предыдущих элементов в массиве (команда `nums[i]=nums[i-1]+nums[i-2]`). После заполнения массива значение переменной-указателя `nums` (ссылка на первый элемент динамического массива) возвращается результатом функции.

Аналогично организована функция `myrand()`, только у нее значение элементам динамического массива присваивается с использованием функции генерирования случайных чисел `rand()`.

В функции `main()` объявляется указатель `f` на целочисленное значение. Затем командой `f=fibs(n)` в указатель `f` записывается адрес первого элемента массива, создаваемого при вызове функции `fibs()`. После этого `f` можно рассматривать как массив, заполненный числами Фибоначчи. Переменной-указателю `f` можно присвоить и другое значение — например, командой `f=myrand(m)` в `f` записывается адрес первого элемента заполненного случайными числами массива, который создается при вызове функции `myrand()`. Но перед присваиванием нового значения указателю `f` необходимо выполнить команду `delete [] f`, которой удаляется массив с числами Фибоначчи. Если этого не сделать, то адрес массива «потеряется» (он не будет записан ни в один указатель), и в результате память останется неосвобожденной. Собственно, здесь кроется важный аргумент против использования подхода, при котором функцией возвращается динамический массив. Такой массив создается при вызове функции, но по завершении ее работы не удаляется из памяти. Ситуация чревата тем, что массив не будет своевременно удален из памяти.

Описанную выше процедуру (с возвращением результатом функции динамического массива), со статическим массивом проделать не получится. Дело в том, что все статические переменные (в том числе и массивы),

которые создаются при вызове функции, после завершения ее выполнения автоматически удаляются. Однако можно применить иную тактику, и вместо того, чтобы пытаться описывать функцию, возвращающую массив, передать в функцию массив через аргументы, а затем внести в этот массив нужные изменения. В листинге 4.15 представлена программа, которая формально напоминает предыдущую (из листинга 4.14). Но только теперь массивы, с которыми выполняются манипуляции, передаются в соответствующую функцию через аргументы. Рассмотрим представленный ниже программный код:

Листинг 4.15. Заполнение статического массива

```
#include <iostream>
#include <cstdlib>
using namespace std;
// В функцию передается массив для заполнения
// числами Фибоначчи:
void fibs(int* nums,int n){
    for(int i=0;i<n;i++){
        if(i==0||i==1){
            nums[i]=1;
        }
        else{
            nums[i]=nums[i-1]+nums[i-2];
        }
    }
}
// В функцию передается массив для заполнения
// случайными числами:
void myrand(int* nums,int m){
    for(int i=0;i<m;i++){
        nums[i]=rand()%10;
    }
}
// Главная функция программы:
```

```
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
   // Размер массива:
   const int n=15;
   // Массив:
   int f[n];
   // Заполнение массива числами Фибоначчи:
   fibs(f,n);
   // Отображение значений элементов массива:
   for(int i=0;i<n;i++){
      cout<<f[i]<<" ";
   }
   cout<<endl;
   // Заполнение массива случайными числами:
   myrand(f,n);
   // Отображение содержимого массива:
   for(int i=0;i<n;i++){
      cout<<f[i]<<" ";
   }
   cout<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения функции может быть следующим:

Результат выполнения программы (из листинга 4.15)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
5 6 8 5 4 0 0 1 5 6 7 9 2 6 3
```

Теперь функции `fibs()` и `myrand()` не возвращают результат, а массив, который следует заполнить соответственно числами Фибоначчи и случайными числами, передается в функцию (с помощью двух аргументов: имени массива и его размера).

В теле функции создается статический целочисленный массив `f` предопределенного размера. Затем, до присваивания значений элементам массива имя массива передается аргументом в функцию `fibs()`. Это корректная команда, поскольку `f` представляет собой указатель на первый элемент массива, и после создания массива, даже если он еще не заполнен, у указателя `f` уже есть значение. Во всем остальном команды программы должны быть понятны читателю.

Указатель на функцию

Имя функции является указателем на функцию. Это простое обстоятельство можно использовать при решении значительного класса прикладных задач. Далее рассматриваются некоторые примеры использования указателей на функции.

ТЕОРИЯ

Переменная, являющаяся **указателем на функцию**, описывается следующим образом:

- указывается ключевое слово, определяющее тип результата, который возвращает функция, на которую может ссылаться указатель;
- указывается название для переменной-указателя;
- перед именем переменной-указателя ставится звездочка *, а вся конструкция из звездочки и имени указателя заключается в круглые скобки;
- после имени указателя в круглых скобках перечисляются идентификаторы типов аргументов, которые передаются функции, на которую может ссылаться указатель.

В листинге 4.16 представлен очень простой пример использования указателей на функцию.

Листинг 4.16. Указатель на функцию

```
#include <iostream>
#include <cstdlib>
```

```
#include <cmath>
using namespace std;
// Функции с двумя аргументами (тип double и int),
// возвращающие результат типа double:
double f(double x,int n){
   double s=1;
   for(int k=1;k<=n;k++){
      s*=(1+x);
   }
   return s;
}
double g(double x,int n){
   double s=1;
   for(int k=1;k<=n;k++){
      s*=x/k;
   }
   return s;
}
// Функции с одним аргументом (тип int),
// возвращающие результат типа char:
char h(int n){
   return 'A'+n;
}
char u(int n){
   return 'Z'-n;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Переменные для передачи аргументами:
   double x=2;
   int n=3;
```

```
    // Указатели на функции:
    double (*p)(double,int);
    char (*q)(int);
    double (*r)(double);
    // Использование указателей на функции:
    p=f;
    cout<<"| "<<p(x,n)<<" | ";
    p=g;
    cout<<p(x,n)<<" | ";
    q=h;
    cout<<q(n)<<" | ";
    q=u;
    cout<<q(n)<<" | ";
    r=exp;
    cout<<r(x/2)<<" | ";
    r=log;
    cout<<r(x)<<" |\n";
       // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже показан результат выполнения программы:

Результат выполнения программы (из *листинга 4.16*)

```
| 27 | 1.33333 | D | W | 2.71828 | 0.693147 |
```

В программе описаны четыре вспомогательные функции:

- функцией `f()` для аргументов `x` и `n` возвращается значение $(1 + x)^n$;
- функцией `g()` для аргументов `x` и `n` возвращается значение $x^n/n!$;
- функцией `h()` для аргумента `n` возвращается символ, смещенный в кодовой таблице символов от символа `'A'` на `n` позиций вперед;
- функцией `u()` для аргумента `n` возвращается символ, смещенный в кодовой таблице символов от символа `'Z'` на `n` позиций назад.

В главной функции программы объявляется три указателя на функции:

- Указатель `p` объявлен командой `double (*p)(double,int)`. Значением этому указателю может присваиваться имя функции, у которой два аргумента (первый типа `double` и второй типа `int`) и которая результатом возвращает значение типа `double`. Например, после выполнения команды `p=f` указатель `p` является альтернативным способом обращения к функции `f()`. Поэтому команда `p(x,n)` означает вызов функции `f()` с аргументами `x` и `n`. После выполнения команды `p=g`, выражение `p(x,n)` означает вызов функции `g()` с аргументами `x` и `n`.
- Указатель `q` объявлен командой `char (*q)(int)`. Значением этому указателю может присваиваться имя функции, у которой один аргумент типа `int` и которая результатом возвращает значение типа `char`. Например, после выполнения команды `q=h` указатель `q` является альтернативным способом обращения к функции `h()`. Команда `q(n)` означает вызов функции `h()` с аргументом `n`. После выполнения команды `q=u`, выражение `q(n)` означает вызов функции `u()` с аргументом `n`.
- Указатель `r` объявлен командой `double (*r)(double)`. Значением этому указателю может присваиваться имя функции, у которой один аргумент типа `double` и которая результатом возвращает значение типа `double`. После выполнения команды `r=exp` (функция для вычисления экспоненты) указатель `r` является альтернативным способом обращения к встроенной математической функции `exp()`. Команда `r(x/2)` означает вызов функции `exp()` с аргументом `x/2`. После выполнения команды `r=log` (функция для вычисления натурального логарифма) выражение `r(x)` означает вызов функции `log()` с аргументом `x`.

НА ЗАМЕТКУ

Для использования встроенных математических функций в программе мы подключаем заголовок `<cmath>`.
Что касается вычисляемых значений, то имеет смысл обратить внимание на то, что при $x = 2$ и $n = 3$ имеют место такие соотношения: $(1 + x)^n = 3^3 = 27$, $x^n/n! = 2^3/3! = 8/6 = 4/3 \approx 1{,}33333$, $\exp(x/2) = \exp(1) \approx 2{,}71828$ и $\ln(x) = \ln(2) \approx 0{,}693147$. Третья буква после буквы `'A'` — буква `'D'`, а третья буква перед буквой `'Z'` — буква `'W'`.

Еще один пример использования указателей на функции представлен в листинге 4.17. Там мы имеем дело с программой, в которой в числовом

виде вычисляется определенный интеграл. Для вычисления интеграла создается специальная функция, аргументами которой при вызове передается указатель на подынтегральную функцию и числовые значения, определяющие границы области интегрирования.

ПОДРОБНОСТИ

Чтобы не усложнять себе жизнь, будем исходить из того, что определенным интегралом $\int_a^b f(x)dx$ от подынтегральной функции $f(x)$ на интервале значений от a до b является площадь под кривой, определяемой зависимостью $f(x)$. Но на самом деле формально определение интеграла в данном случае не очень важно. Важно то, что мы будем вычислять интеграл (в приближенном числовом виде) по следующей формуле (формула трапеций): $\int_a^b f(x)dx \approx ((f(a) + f(b))/2)\Delta x + \Delta x \sum_{k=1}^{n-1}(f(a + k\Delta x)$, где введено обозначение $\Delta x = (b - a)/n$, а через n обозначено некоторое целое число (чем оно больше, тем выше точность вычислений). Именно приведенная выше формула (при заданной функции $f(x)$ и границах a и b) используется в программе для вычисления интеграла.

Рассмотрим представленную ниже программу:

Листинг 4.17. Передача указателя на функцию аргументом функции при вычислении интеграла

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция для вычисления интеграла. Первым аргументом
// передается указатель на функцию:
double integrate(double (*F)(double),double a,double b){
   int n=1000;
   double dx=(b-a)/n;
   double s=(F(a)+F(b))*dx/2;
   for(int k=1;k<=n-1;k++){
      s+=F(a+dx*k)*dx;
   }
   return s;
}
```

```
// Подынтегральные функции:
double f(double x){
   return x*(1-x);
}
double g(double x){
   return 1/x;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Первый интеграл:
   cout<<integrate(f,0,1)<<endl;
   // Второй интеграл:
   cout<<integrate(g,1,2)<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 4.17)

```
0.166667
0.693147
```

Функция для вычисления интеграла называется `integrate()`. Она описана так, что результатом возвращает значение типа `double` (значение интеграла). Ее первый аргумент описан как `double (*F)(double)`. Данный аргумент — указатель на функцию, у которой один `double`-аргумент и которая возвращает значение типа `double`. Поэтому в теле функции аргумент `F` обрабатывается как имя функции. Два других `double`-аргумента `a` и `b` определяют границы области интегрирования.

Локальная переменная `n` определяет количество интервалов, на которые разбивается область интегрирования, а переменная `dx` со значением

(b-a)/n определяет длину каждого такого интервала. Интегральная сумма записывается в переменную s, начальное значение которой задано выражением (F(a)+F(b))*dx/2. Далее запускается оператор цикла, в котором за каждую очередную итерацию к значению переменной s добавляется слагаемое F(a+dx*k)*dx. После завершения оператора цикла значение переменной s возвращается результатом функции.

Также описаны две подынтегральные функции. Функция f() определяет зависимость вида $f(x) = x(1 - x)$, а функция g() определяет зависимость вида $g(x) = 1/x$. В главной функции программы командами integrate(f,0,1) и integrate(g,1,2) соответственно вычисляются интегралы $\int_0^1 x(1-x)dx = 1/6 \approx 0{,}166667$ и $\int_1^2 (dx/x) = \ln(2) \approx 0{,}693147$.

Задачи для самостоятельного решения

1. Напишите программу с функцией для вычисления факториала числа (факториал числа $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ — произведение натуральных чисел от 1 до этого числа). Предложите разные способы организации кода (например, с рекурсией и без рекурсии).

2. Напишите программу с функцией для вычисления двойного факториала числа (двойной факториал числа $n!! = n \cdot (n-2) \cdot (n-4) \cdot \ldots$ — произведение натуральных чисел «через одно число» — последний множитель 2 для четных чисел и 1 для нечетных чисел). Предложите разные способы организации кода (например, с рекурсией и без рекурсии).

3. Напишите программу функцией для вычисления чисел Фибоначчи. Аргументом функции передается номер числа в последовательности, а результатом возвращается само число. Рассмотреть способ описания функции с использованием рекурсии и без использования рекурсии.

4. Напишите программу с функцией для вычисления биномиальных коэффициентов $C_n^k = n!/(k!(n-k)!)$. Параметры k и n передаются аргументами функции.

5. Напишите программу с функцией для вычисления суммы натуральных чисел. В функции использовать рекурсию. Перегрузить функцию так, чтобы второй аргумент определял степень, в которую возводятся слагаемые при суммировании. Например, если аргументами функции переданы значения n и k, то результатом возвращает-

ся сумма $1^k + 2^k + 3^k + ... + n^k$ (при $k = 1$ получается сумма натуральных чисел).

6. Напишите программу с функцией для вычисления экспоненты $\exp(x) \approx 1 + x + (x^2/2!) + ...(x^n/n!)$. У функции должно быть два аргумента (параметры x и n), а при ее описании использовать рекурсию. Воспользоваться тем, что если $S_n(x) = 1 + x + (x^2/2!) + ...(x^n/n!)$, то имеет место рекуррентное соотношение $S_n(x) = S_{n-1}(x) + (x^n/n!)$ и кроме того $S_0(x) = 1$.

7. Напишите программу с перегруженной функцией. Если функции передается один числовой аргумент, то она возвращает результатом значение этого аргумента. Если функции передается два числовых аргумента, то она возвращает результатом сумму квадратов их значений. Если функции передается три числовых аргумента, то она возвращает результатом сумму кубов их значений.

8. Напишите программу с функцией, аргументами которой передаются: указатель на функцию и массив числовых значений. При вызове функции к каждому из элементов массива применяется функция, переданная первым аргументом (через указатель). Предусмотреть перегрузку функции таким образом, чтобы первым аргументом (вместо указателя на функцию) можно было передавать число (в таком случае все элементы массива-аргумента умножаются на данное число).

9. Напишите программу с функцией, предназначенной для вычисления производной от некоторой функции. Первым аргументом передается указатель на дифференцируемую функцию, а второй аргумент — точка, в которой вычисляется производная. Для вычисления производной от некоторой функции $f(x)$ использовать приближенную формулу $f'(x) = (f(x + h) - f(x))/h$, где через h обозначено малое приращение по аргументу (выбирается произвольно). Предусмотреть перегрузку функции такую, чтобы вместо второго числового аргумента можно было передавать массив значений. В таком случае аргументом функции передается еще один дополнительный массив, в который записываются значения производной в каждой из точек, определяемых элементами первого массива.

10. Напишите программу с функцией для вычисления значения полинома в точке. Аргументами функции передаются массив с числовыми значениями и точка, в которой вычисляется значение полинома. Если через a_k обозначить элементы массива, переданного

аргументом функции (индекс $k = 0, 1, 2, ..., n$), а через x обозначить точку, в которой вычисляется значение полинома, то результатом функции должна возвращаться сумма $a_0 + a_1x + a_2x^2 + ... + a_nx^n$. Перегрузить функцию так, чтобы при отсутствии аргумента, определяющего точку для вычисления значения полинома, функцией в консольное окно выводились коэффициенты полинома (значения элементов массива).

11. Напишите программу с функцией, аргументом которой передается числовой массив. При вызове функции элементы массива сортируются в порядке возрастания, а результатом функция возвращает указатель на последний элемент массива.

12. Напишите программу с функцией, аргументом которой передается числовой массив. При вызове функции элементы массива сортируются в порядке убывания, а результатом функция возвращает ссылка на последний элемент массива.

13. Напишите программу с функцией, аргументом которой передается символьный массив с текстом. Результатом функции возвращается длина самого длинного слова в тексте, содержащемся в массиве.

14. Напишите программу с функцией, аргументом которой передается символьный массив с текстом. При вызове функции текст, записанный в массив, отображается в обратном порядке. Попробуйте реализовать такую функцию через рекурсию.

15. Напишите программу с функцией, аргументами которой передаются символьный массив и отдельный символ. Результатом функцией возвращается количество букв, определяемых символьным аргументом, в тексте, содержащемся в символьном массиве.

16. Напишите программу с функцией, аргументом которой передается двумерный числовой массив. Результатом функцией возвращается ссылка на элемент массива с наибольшим значением.

17. Напишите программу с функцией, аргументом которой передается двумерный числовой массив. Результатом функцией возвращается указатель на элемент массива с наименьшим значением.

18. Напишите программу с функцией, аргументом которой передается двумерный числовой массив. Результатом функцией возвращается среднее арифметическое для элементов массива (сумма всех элементов, деленная на их количество).

19. Напишите программу с функцией, аргументом которой передается символ и двумерный символьный массив (предварительно заполненный случайными символами). Результатом функцией возвращается количество вхождений символа в двумерный массив.

20. Напишите программу с функцией, аргументом которой передается символ и двумерный символьный массив, в каждую строку которого записан текст. Результатом функцией возвращается количество вхождений символа в двумерный массив (*в отличие от* предыдущей задачи, в данном случае проверять нужно не все элементы двумерного символьного массива, а только те, что содержат текст).

Глава 5
КЛАССЫ И ОБЪЕКТЫ

> Узнать в ноже каретную рессору. Какое глубокое проникновение в суть вещей! Впрочем, Принц всегда тонко анализировал самую сложную ситуацию.
>
> *из к/ф «Клуб самоубийц, или Приключения титулованной особы»*

В этой главе мы познакомимся с азами объектно-ориентированного программирования (сокращенно ООП), в частности:

- научимся описывать классы и создавать объекты;
- рассмотрим режимы доступа к полям и методам класса;
- научимся использовать конструкторы и деструкторы;
- узнаем, что такое перегрузка методов;
- познакомимся с наследованием;
- узнаем, как выполняется перегрузка операторов,

а также остановимся на некоторых смежных вопросах, связанных с использованием классов и объектов.

Знакомство с классами и объектами

Рассмотрим уже знакомую нам задачу о размещении на депозите определенной денежной суммы, на которую начисляются проценты, но на этот раз прибегнем к помощи *классов и объектов*.

ПОДРОБНОСТИ

Напомним, что если на депозите размещается сумма m на время (в годах) t под r процентов годовых, то итоговая сумма по такому вкладу вычисляется по формуле $m(1 + (r/100))^t$.

Рассмотрим программный код, представленный в листинге 5.1.

Листинг 5.1. Описание класса и создание объекта

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание класса:
class MyMoney{
public:
   // Поля:
   string name;
   double money;
   double rate;
   int time;
// Методы:
double getMoney(){
   double s=money;
      for(int k=1;k<=time;k++){
         s*=(1+rate/100);
      }
   return s;
}
void showAll(){
   cout<<"Имя: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
      cout<<"Период (лет): "<<time<<endl;
      cout<<"Итоговая сумма: "<<getMoney()<<endl;
   }
};
// Главная функция программы:
```

```
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Создание объекта:
    MyMoney obj;
    // Присваивание значений полям:
    obj.name="Иванов Иван Иванович";
    obj.money=1000;
    obj.rate=8;
    obj.time=5;
    // Вызов метода:
    obj.showAll();
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Наиболее важная часть программы связана с описанием класса, который называется `MyMoney`.

ТЕОРИЯ

Класс представляет собой общую схему или шаблон, на основе которого затем создаются **объекты**. Класс содержит описание **полей** и **методов**, которые называются **членами** класса. Поле класса аналогично переменной, а метод — аналог функции. При создании на основе класса объектов, последние получают персональный набор полей и методов — строго в соответствии с описанием класса. Поля объектов имеют значения, а методы выполняют некоторые действия и могут возвращать результат. Фактически, поля и методы — это соответственно переменные и функции, но только строго «прикрепленные» к определенному объекту. Разные объекты, созданные на основе одного класса, имеют одинаковые наборы полей и методов, при этом значения полей у каждого объекта свои, а методы имеют автоматический доступ к объектам, из которых вызываются. Нет смысла говорить о значении поля или результате вызова метода, если не идентифицирован объект, к которому они принадлежат.

Описание класса начинается с ключевого слова `class`, после чего указывается имя класса, а затем в фигурных скобках описывается тело класса (в конце описания стоит точка с запятой).

В теле класса описываются *поля и методы класса*. Но перед началом описания следует ключевое слово `public` (после которого стоит двоеточие). Данное ключевое слово идентифицирует блок открытых членов класса — таких, доступ к которым есть не только в пределах класса, но и из внешнего кода.

Полей у класса `MyMoney` четыре:

- поле `name` типа `string` (текстовое значение) предназначено для записи имени вкладчика;
- поле `money` типа `double` предназначено для записи начальной суммы вклада;
- поле `rate` типа `double` предназначено для записи показателя годовой процентной ставки;
- целочисленное (тип `int`) поле `time` предназначено для записи количества лет, на которые размещается депозит.

НА ЗАМЕТКУ

Текстовое поле `name` в данном случае реализуется в виде объекта встроенного класса `string`, предназначенного для работы с текстом. Для использования класса `string` в программе подключается заголовок `<string>`.

Помимо полей, у класса `MyMoney` еще есть два метода. Метод `getMoney()` предназначен для вычисления итоговой суммы по вкладу. Метод возвращает значение типа `double` (итоговая сумма по вкладу), и у него нет аргументов.

ПОДРОБНОСТИ

Для вычисления итоговой суммы по вкладу необходимо знать три параметра: начальную сумму вклада, годовую ставку процента и количество лет, на которые размещается депозит. Значения этих параметров метод `getMoney()` получает через поля `money`, `rate` и `time` объекта, из которого вызывается. Поэтому нет необходимости передавать методу `getMoney()` аргументы.

В теле метода `getMoney()` объявляется локальная переменная `s` со значением `money` (то есть при вызове метода локальной переменной `s` присваивается значение поля `money` того объекта, из которого вызывается метод). Далее запускается оператор цикла, в котором индексная переменная пробегает значения в диапазоне от `1` до значения поля `time` (опять же, здесь имеется в виду поле объекта, из которого вызывается метод). За каждый цикл выполняется команда `s*=(1+rate/100)`, которой текущее значение переменной `s` умножается на множитель `(1+rate/100)` (здесь `rate` — поле объекта). После завершения выполнения оператора цикла значение переменной `s` возвращается результатом метода `getMoney()`.

НА ЗАМЕТКУ

Таким образом, когда метод `getMoney()` вызывается из объекта, то он обращается к полям данного объекта для получения тех значений, которые необходимы при выполнении расчетов. Поэтому если метод `getMoney()` вызывается из разных объектов, то в общем случае получаем и разные результаты.

Метод `showAll()` не возвращает результат (идентификатор типа `void` в начале описания метода), и у него нет аргументов. При вызове метода в консольном окне отображаются значения всех четырех полей объекта, из которого вызывается метод.

Собственно, на этом описание класса `MyMoney` заканчивается. Далее рассмотрим команды в главной функции программы. Там, во-первых, создается объект `obj` класса `MyMoney`. Делается это очень просто — с помощью команды `MyMoney obj`. Фактически объект создается так же, как объявляются обычные переменные, только при создании объекта в качестве идентификатора типа указывается имя класса. Но создание объекта еще не означает, что его поля получили значения. Под них только выделено место, а значения приходится присваивать вручную. Обращение к полям объекта выполняется так: указывается имя объекта, и через точку после имени объекта указывается название поля. Во всем остальном полями можно манипулировать так же, как и обычными переменными. В частности, командами `obj.name="Иванов Иван Иванович"`, `obj.money=1000`, `obj.rate=8` и `obj.time=5` полям объекта `obj` присваиваются значения. Затем командой `obj.showAll()` из объекта `obj` вызывается метод `showAll()`. Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 5.1)

```
Имя: Иванов Иван Иванович
Вклад: 1000
Ставка (%): 8
Период (лет): 5
Итоговая сумма: 1469.33
```

Как видим, использование класса и объекта во многом оказалось удобным — может быть, лишь за исключением процедуры присваивания значений полям объекта. Но, как мы увидим далее, у данной проблемы есть решение (и даже несколько).

Открытые и закрытые члены класса

Внесем некоторые изменения в программный код, рассмотренный в предыдущем примере.

В частности, для присваивания значений полям создадим специальный метод, а сами поля и метод для вычисления итоговой суммы по депозиту сделаем *закрытыми*.

ТЕОРИЯ

К закрытым полям и методами можно обращаться только в программном коде в теле класса. Закрытые поля и методы описываются в блоке, помеченном ключевым словом `private` (или вообще без ключевого слова: по умолчанию, если члены класса явно не идентифицированы ключевым словом `public` как открытые, они считаются закрытыми).

В предыдущем примере мы могли в главной функции в явном виде присваивать значения полям объекта именно потому, что поля были описаны как открытые. В следующем примере, как уже отмечалось, мы все поля сделаем закрытыми, поэтому присвоить им значения явным образом после того, как создан объект, уже не получится. Для этой цели мы описываем метод `setAll()` с четырьмя аргументами, определяющими значения, которые следует присвоить полям объекта. Понятно, что сам метод `setAll()` должен быть открытым. Рассмотрим программный код, представленный в листинге 5.2.

Листинг 5.2. Закрытые и открытые члены класса

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание класса:
class MyMoney{
private:   // Закрытые члены класса
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // Открытые члены класса
   void showAll(){
      cout<<"Имя: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
      cout<<"Период (лет): "<<time<<endl;
      cout<<"Итоговая сумма: "<<getMoney()<<endl;
   }
   void setAll(string n,double m,double r,int t){
      name=n;
      money=m;
      rate=r;
      time=t;
```

```
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Создание объектов:
MyMoney objA,objB;
    // Присваивание значений полям:
objA.setAll("Иванов Иван Иванович",1000,8,5);
objB.setAll("Петров Петр Петрович",1200,7,4);
    // Отображение значений полей:
objA.showAll();
cout<<endl;
objB.showAll();
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже приведен результат выполнения программы:

Результат выполнения программы (из листинга 5.2)

```
Имя: Иванов Иван Иванович
Вклад: 1000
Ставка (%): 8
Период (лет): 5
Итоговая сумма: 1469.33

Имя: Петров Петр Петрович
Вклад: 1200
Ставка (%): 7
Период (лет): 4
Итоговая сумма: 1572.96
```

В теле метода `setAll()` значения аргументов присваиваются значениями полям объекта, из которого метод вызывается. Поэтому для заполнения полей значениями из соответствующего объекта следует вызвать метод `setAll()` и передать ему аргументами значения для полей `name`, `money`, `rate` и `time`.

В главной функции теперь создается два объекта (`objA` и `objB`), для чего использована команда `MyMoney objA,objB`. Для присваивания значений полям объекта `objA` используем команду `objA.setAll("Иванов Иван Иванович",1000,8,5)`. Команда `objB.setAll("Петров Петр Петрович",1200,7,4)` задействуется для присваивания значений полям объекта `objB`. Проверить значение полей каждого из объектов помогают команды `objA.showAll()` и `objB.showAll()`.

НА ЗАМЕТКУ

В методе `selAll()` первый аргумент описан как такой, что относится к типу `string` (на самом деле это объект класса `string`). Текстовый литерал реализуется посредством символьного массива. То есть в принципе типы разные. Но за счет автоматического преобразования типов мы можем передавать текстовые литералы аргументом, который на самом деле должен относиться к типу `string`.

Перегрузка методов

Анализируя предыдущую программу, легко заметить, что способ присваивания значений полям с помощью специального метода хотя и упрощает процесс использования объектов, но некоторые «тонкие» места в нем все равно остаются. Скажем, вполне может понадобиться изменить значение не всех полей, а только некоторых. В этом случае полезной может быть *перегрузка методов*.

ТЕОРИЯ

При **перегрузке методов**, как и при перегрузке функций, описывается несколько версий метода с одним и тем же названием. Такие «версии» должны чем-то отличаться: количеством аргументов, их типами, типом возвращаемого результата.

В листинге 5.3 представлена версия программы, предназначенной для вычисления итоговой суммы по депозитам, в которой, как и ранее, опи-

сывается класс MyMoney, но теперь в этом классе перегружается метод setAll(), благодаря чему при изменении значений полей объекта класса MyMoney нет необходимости указывать значения всех аргументов — можно ограничиться только некоторыми из них.

Листинг 5.3. Перегрузка методов

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание класса:
class MyMoney{
private:   // Закрытые члены класса
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // Открытые члены класса
   void showAll(){
   cout<<"Имя: "<<name<<endl;
   cout<<"Вклад: "<<money<<endl;
   cout<<"Ставка (%): "<<rate<<endl;
   cout<<"Период (лет): "<<time<<endl;
   cout<<"Итоговая сумма: "<<getMoney()<<endl;
}
// Версия с четырьмя аргументами:
```

```
   void setAll(string n,double m,double r,int t){
      name=n;
      money=m;
      rate=r;
      time=t;
   }
   // Версия с тремя аргументами:
   void setAll(double m,double r,int t){
      money=m;
      rate=r;
      time=t;
   }
   // Версия с текстовым аргументом:
   void setAll(string n){
      name=n;
   }
   // Версия с целочисленным аргументом:
   void setAll(int t){
      time=t;
   }
   // Версия с двумя аргументами типа double:
   void setAll(double m,double r){
      money=m;
      rate=r;
   }
   // Версия с двумя аргументами (типа double и типа bool):
   void setAll(double x,bool s=true){
      if(s){
         money=x;
      }
      else{
         rate=x;
      }
```

```
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Создание объекта:
    MyMoney obj;
    // Присваивание значений полям:
    obj.setAll("Иванов Иван Иванович",1000,8,5);
    obj.showAll();
    cout<<endl;
    // Изменение имени:
    obj.setAll("Петров Петр Петрович");
    obj.showAll();
    cout<<endl;
    // Изменение времени вклада:
    obj.setAll(10);
    obj.showAll();
    cout<<endl;
    // Изменение начальной суммы:
    obj.setAll(1200.0);
    obj.showAll();
    cout<<endl;
    // Изменение начальной суммы:
    obj.setAll(1500,true);
    obj.showAll();
    cout<<endl;
    // Изменение процентной ставки:
    obj.setAll(6,false);
    obj.showAll();
    cout<<endl;
    // Изменение начальной суммы, процентной ставки и
```

```
// времени размещения депозита:
obj.setAll(1000,8,5);
obj.showAll();
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.3)

```
Имя: Иванов Иван Иванович
Вклад: 1000
Ставка (%): 8
Период (лет): 5
Итоговая сумма: 1469.33

Имя: Петров Петр Петрович
Вклад: 1000
Ставка (%): 8
Период (лет): 5
Итоговая сумма: 1469.33

Имя: Петров Петр Петрович
Вклад: 1000
Ставка (%): 8
Период (лет): 10
Итоговая сумма: 2158.92

Имя: Петров Петр Петрович
Вклад: 1200
Ставка (%): 8
Период (лет): 10
```

```
Итоговая сумма: 2590.71

Имя: Петров Петр Петрович
Вклад: 1500
Ставка (%): 8
Период (лет): 10
Итоговая сумма: 3238.39

Имя: Петров Петр Петрович
Вклад: 1500
Ставка (%): 6
Период (лет): 10
Итоговая сумма: 2686.27

Имя: Петров Петр Петрович
Вклад: 1000
Ставка (%): 8
Период (лет): 5
Итоговая сумма: 1469.33
```

По сравнению с предыдущим примером (см. листинг 5.2), мы добавили несколько дополнительных версий метода `setAll()`, и изменили в связи с этим код главной функции программы — таким образом, что там тестируются различные способы вызова метода `getAll()`. Более конкретно, помимо версии метода `setAll()` с четырьмя аргументами, есть еще и следующие способы передачи аргументов методу:

- Если метод вызывается с тремя числовыми аргументами, то эти аргументы определяют значение полей `money`, `rate` и `time`.
- При передаче методу аргументом текстового значения оно присваивается значением *полю* `name` *объекта*. Здесь нелишним будет заметить, что на самом деле текстовый литерал реализуется в виде символьного массива. Однако путем автоматического преобразования типа текстовый литерал «приводится» к текстовому типу `string`.
- При вызове метода с одним целочисленным аргументом он задает новое значение для поля `time`.

- При вызове метода с двумя числовыми аргументами, они определяют значения полей `money` и `rate` соответственно.
- Наконец в классе `MyMoney` описана версия метода `getAll()` с двумя аргументами: первый типа `double` и второй типа `bool` (логический тип). Причем второй аргумент имеет значение по умолчанию `true`. Если значение второго аргумента равно `true`, то тогда первый числовой аргумент служит значением для поля `money`. Если второй аргумент равен `false`, то первый числовой аргумент служит значением для поля `rate`.

НА ЗАМЕТКУ

Таким образом, если метод `setAll()` вызывается с одним целочисленным аргументом, то в таком случае значение присваивается полю `time`. Если метод `setAll()` вызывается с одним аргументом типа `double`, то значение присваивается полю `money`. Поэтому, например, в команде `obj.setAll(1200.0)` числовой аргумент `1200.0` указан с десятичной точкой, так как в противном случае он бы был целочисленным, и значение бы присваивалось полю `time`. Поскольку аргумент у метода относится к типу `double`, то вызывается версия метода с двумя аргументами, первый из которых типа `double`, а для второго логического аргумента используется значение по умолчанию `true`. А вот в инструкции `obj.setAll(1500,true)` значение второго логического аргумента указано явно (хоть оно и совпадает со значением по умолчанию), а первый аргумент является целым числом. В такой ситуации первый целочисленный аргумент автоматически приводится к типу `double`. Под это же правило подпадает команда `obj.setAll(6,false)`. Однако поскольку в ней значение второго логического аргумента равно `false`, то числовое значение присваивается полю `rate`.

Знакомство с конструкторами и деструкторами

В рассматриваемом далее программном коде класс `MyMoney` пополняет свой «арсенал» за счет *конструктора* и *деструктора*.

ТЕОРИЯ

Конструктор — метод, который автоматически вызывается при создании объекта. Название конструктора совпадает с названием класса, конструктор не возвращает результат и описывается без иден-

тификатора типа результата. У конструктора могут быть аргументы, и конструктор можно перегружать (в классе допустимо описывать несколько версий конструктора).

Деструктор — метод, который автоматически вызывается при удалении объекта из памяти. Деструктор не возвращает результат, описывается без указания идентификатора типа результата, у него нет аргументов, и он не перегружается (в классе может быть только один деструктор). Имя деструктора формируется объединением ~ и названия класса.

Интересующий нас программный код представлен в листинге 5.4. Для уменьшения объема кода мы в классе `MyMoney` описываем только одну версию метода `setAll()` (версия с четырьмя аргументами).

Листинг 5.4. Конструкторы и деструкторы

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание класса:
class MyMoney{
private:   // Закрытые члены класса
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // Открытые члены класса
   // Конструктор без аргументов:
```

```
MyMoney(){
    name="Кот Матроскин";
    money=100;
    rate=5;
    time=1;
    cout<<"Создан новый объект:\n";
    showAll();
}
// Конструктор с четырьмя аргументами:
MyMoney(string n,double m,double r,int t){
    setAll(n,m,r,t);
    cout<<"Создан новый объект:\n";
    showAll();
}
// Деструктор:
~MyMoney(){
    cout<<"Объект для \""<<name<<"\" удален\n";
    for(int k=1;k<=35;k++){
       cout<<"*";
    }
    cout<<endl;
}
// Методы класса:
void showAll(){
    cout<<"Имя: "<<name<<endl;
    cout<<"Вклад: "<<money<<endl;
    cout<<"Ставка (%): "<<rate<<endl;
    cout<<"Период (лет): "<<time<<endl;
    cout<<"Итоговая сумма: "<<getMoney()<<endl;
    for(int k=1;k<=35;k++){
       cout<<"-";
    }
    cout<<endl;
```

```
    }
void setAll(string n,double m,double r,int t){
    name=n;
    money=m;
    rate=r;
    time=t;
    }
};
// Функция:
void postman(){
    // Создание локального объекта:
    MyMoney objD("Почтальон Печкин",200,3,2);
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Создание объектов:
    MyMoney objA;
    MyMoney objB("Дядя Федор",1500,8,7);
    // Вызов функции:
    postman();
    // Создание динамического объекта:
    MyMoney* objC=new MyMoney("Пес Шарик",1200,6,9);
    cout<<"Все объекты созданы\n";
    // Удаление динамического объекта:
    delete objC;
    cout<<"Выполнение программы завершено\n";
    cout<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 5.4)

```
Создан новый объект:
Имя: Кот Матроскин
Вклад: 100
Ставка (%): 5
Период (лет): 1
Итоговая сумма: 105
-----------------------------------
Создан новый объект:
Имя: Дядя Федор
Вклад: 1500
Ставка (%): 8
Период (лет): 7
Итоговая сумма: 2570.74
-----------------------------------
Создан новый объект:
Имя: Почтальон Печкин
Вклад: 200
Ставка (%): 3
Период (лет): 2
Итоговая сумма: 212.18
-----------------------------------
Объект для "Почтальон Печкин" удален
***********************************
Создан новый объект:
Имя: Пес Шарик
Вклад: 1200
Ставка (%): 6
Период (лет): 9
Итоговая сумма: 2027.37
-----------------------------------
```

```
Все объекты созданы
Объект для "Пес Шарик" удален
**********************************
Выполнение программы завершено


Объект для "Дядя Федор" удален
**********************************
Объект для "Кот Матроскин" удален
**********************************
```

Нас в программном коде интересуют в первую очередь две версии конструктора и деструктор, описанные в классе `MyMoney`. В классе описано две версии конструктора: без аргументов и с четырьмя аргументами. В версии конструктора без аргументов командами `name="Кот Матроскин"`, `money=100`, `rate=5` и `time=1` полям объекта присваиваются значения. Затем командой `cout<<"Создан новый объект:\n"` в консольное окно выводится соответствующее сообщение и, наконец, командой `showAll()` из создаваемого объекта вызывается метод, благодаря чему сведения о значениях полей объекта появляются в консольном окне. Еще раз подчеркнем, что все эти действия выполняются при создании объекта, если для его создания используется конструктор без аргументов.

НА ЗАМЕТКУ

Мы несколько усовершенствовали метод `showAll()`, так что теперь при его вызове, после того, как отображена информация об объекте, появляется сформированная из дефисов импровизированная горизонтальная черта. Предполагается, что это позволит лучше визуализировать результат выполнения программы.

Также в классе `MyMoney` описан конструктор с четырьмя аргументами. В таком случае вызывается метод `setAll()` и ему передаются те же аргументы, что были переданы конструктору. Таким образом, аргументы конструктора в данном случае определяют значения, которые присваиваются полям создаваемого объекта.

Далее, после заполнения полей, командой `cout<<"Создан новый объект:\n"` отображается сообщение о создании объекта, после чего вызывается метод `showAll()`.

В деструкторе командой `cout<<"Объект для \""<<name<<"\" удален\n"` в консоль выводится сообщение об удалении объекта (в сообщении указывается, в двойных кавычках, значение поля `name` удаляемого объекта). После этого «подводится черта» из символов * (для чего использован оператор цикла).

ПОДРОБНОСТИ

Чтобы отобразить двойные кавычки в тексте, перед ними ставится косая черта. Проще говоря, если в текст необходимо вставить двойные кавычки, то используется инструкция `\"`. Использование в такой ситуации одних лишь кавычек (без косой черты) приведет к ошибке, поскольку двойные кавычки используются для идентификации текстовых литералов.

Кроме класса `MyMoney` в программе описана функция `postman()`. Функция не возвращает результат, и у нее нет аргументов. Код функции очень простой: в теле функции командой `MyMoney objD("Почтальон Печкин", 200,3,2)` создается объект класса `MyMoney`.

В функции `main()` создается несколько объектов класса `MyMoney` и вызывается функция `postman()`. Правда, объекты создаются по-разному. Один из объектов создается командой `MyMoney objA`. Здесь речь идет о статическом объекте `objA`, при создании которого вызывается конструктор без аргументов. А вот при создании объекта `objB` использована команда `MyMoney objB("Дядя Федор",1500,8,7)`, в которой после имени создаваемого объекта в круглых скобках указаны значения, передаваемые конструктору. Как несложно догадаться, объект `objB` создается с вызовом конструктора с четырьмя аргументами.

ПОДРОБНОСТИ

Решение о том, какую именно версию конструктора следует вызывать при создании объекта, принимается исходя из списка аргументов, которые передаются конструктору (аргументы, указанные после имени объекта в команде, которой объект создается).

Также нужно иметь в виду, что пока в классе не описан ни один конструктор, то при создании используется так называемый **конструктор по умолчанию**, у которого нет аргументов и который никаких дополнительных действий с объектом не выполняет. Как только в классе описан хотя бы один конструктор, конструктор по умолчанию больше не доступен.

Стоит заметить, что как при создании объекта `objA`, так и при создании объекта `objB` вследствие вызова конструктора (разного для разных объектов) в самом конструкторе вызывается метод `showAll()`. Как следствие само только создание объекта приводит к появлению в консольном окне сообщения соответствующего содержания с информацией о значениях полей созданного объекта. Это же замечание относится к объекту, который создается при вызове функции `postman()`. Однако теперь речь идет о локальном объекте. Локальный объект, как и локальная переменная, существует, пока выполняется функция. Поэтому когда вызывается функция `postman()` и начинает выполняться ее код, создается объект `objD`, на чем выполнение функции завершается. Но при завершении выполнения функции из памяти удаляется локальный объект `objD`. Как следствие, вызов функции `postman()` приводит к вызову конструктора класса `MyMoney` (с четырьмя аргументами) и практически сразу к вызову деструктора. При вызове деструктора появляется сообщение об удалении объекта.

Также в программе создается динамический объект, для чего мы используем команду `MyMoney* objC=new MyMoney("Пес Шарик",1200,6,9)`. Здесь инструкцией `MyMoney* objC` объявляется указатель `objC` на объект класса `MyMoney`. Инструкцией `new MyMoney("Пес Шарик",1200,6,9)` создается динамический объект (создается объект путем динамического выделения памяти под него). Адрес объекта записывается в указатель `objC`. При создании динамического объекта вызывается конструктор с четырьмя аргументами.

После создания динамического объекта командой `cout<<"Все объекты созданы\n"` отображается сообщение о завершении процесса создания объектов и командой `delete objC` динамический объект удаляется. При удалении объекта вызывается деструктор. Затем командой `cout<<"Выполнение программы завершено\n"` отображается сообщение о завершении выполнения программы (и выполняется дополнительный пропуск строки с помощью команды `cout<<endl`).

Однако на самом деле на этом выполнение программы не заканчивается. После того, как пользователь нажмет клавишу <Enter> (или другую клавишу), в консольном окне появятся сообщения об удалении объектов со значениями `"Дядя Федор"` и `"Кот Матроскин"` поля `name`. Правда, произойдет это очень быстро и консольное окно сразу после этого закроется.

НА ЗАМЕТКУ

Чтобы увидеть результат выполнения программы «во всей красе» можно порекомендовать (при работе с операционной системой

Windows) воспользоваться командой **Выполнить** в главном меню и запустить через эту утилиту откомпилированный `exe`-файл проекта.

Причина в том, что при завершении выполнения программы (то есть при завершении выполнения функции `main()`) из памяти удаляются объекты, которые были созданы в функции `main()`. Объекты удаляются в порядке, обратном к тому, как они создавались. Поэтому сначала удаляется объект `objB`, а после этого удаляется объект `objA`. Понятно, что каждый раз вызывается деструктор.

Принципы перегрузки операторов

Приведенный далее пример является очередной вариацией на финансовую тему, дополненную на сей раз *перегрузкой операторов.*

ТЕОРИЯ

Для пользовательских классов можно переопределять (или, точнее, задавать) правила вычисления выражений с основными операторами. Это называется **перегрузкой операторов**. Для перегрузки определенного оператора описывается специальная **операторная функция** или **операторный метод**. Операторная функция или метод описываются, как и обычная функция или метод, но с некоторыми особенностями. Так, название операторной функции или метода получается объединением ключевого слова `operator` и символа оператора. Тип результата операторной функции или метода — тип значения, которое возвращается при применении оператора к объекту класса. Аргументы операторной функции определяются операндами выражения, для которого задается действие оператора. Если речь идет об операторном методе, то объект вызова метода отождествляется с первым операндом выражения.

В листинге 5.5 представлена программа, основу которой составляет класс `MyMoney`, для которого определено несколько операторных методов и несколько операторных функций. В частности, с помощью операторных функций определяются такие действия с объектами класса `MyMoney`:

- Вычитание объектов: если из одного объекта класса `MyMoney` вычесть другой объект класса `MyMoney`, в результате получим разницу итоговых сумм по соответствующим вкладам.

- Префиксная форма оператора декремента: при применении оператора декремента к объекту значение поля `money` объекта уменьшается на величину `1000` (но остается со значением не меньше нуля).
- Постфиксная форма оператора декремента: при применении оператора декремента к объекту значение поля `time` объекта уменьшается на `1` (но не может быть меньше нуля).

С помощью операторных методов задаются следующие действия с объектами класса `MyMoney`:

- При сложении двух объектов класса `MyMoney` получаем новый объект этого же класса, у которого значения полей вычисляются на основе значений полей исходных (суммируемых) объектов.
- Префиксная форма оператора инкремента: при применении оператора инкремента к объекту значение поля `money` объекта увеличивается на величину `1000`.
- Постфиксная форма оператора инкремента: при применении оператора инкремента к объекту значение поля `time` объекта увеличивается на `1`.

ПОДРОБНОСТИ

У операторов инкремента `++` и декремента `--` есть префиксная и постфиксная формы. В префиксной форме оператор указывается перед операндом. В постфиксной форме оператор указывается после операнда. Префиксную и постфиксную формы операторов инкремента и декремента можно перегружать по-разному. Действует такое правило: при перегрузке постфиксной формы оператора в операторном методе или функции объявляется формальный не используемый дополнительный целочисленный аргумент.

Для упрощения ситуации и сокращения объема программного кода в программный код класса `MyMoney` внесены некоторые изменения. В частности, все поля и методы в классе объявлены открытыми. Не используется деструктор. Упрощен код конструкторов (теперь при создании объектов сообщения не отображаются). В итоге получился следующий программный код:

Листинг 5.5. Операторные методы и функции

```
#include <iostream>
#include <cstdlib>
```

```
#include <string>
using namespace std;
// Описание класса:
class MyMoney{
public:
   string name;
   double money;
   double rate;
   int time;
   // Конструктор без аргументов:
   MyMoney(){
      name="";
      money=0;
      rate=0;
      time=0;
   }
   // Конструктор с четырьмя аргументами:
   MyMoney(string n,double m,double r,int t){
      setAll(n,m,r,t);
   }
   // Методы класса:
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
   void showAll(){
      cout<<"Имя: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
```

```
    cout<<"Период (лет): "<<time<<endl;
    cout<<"Итоговая сумма: "<<getMoney()<<endl;
    for(int k=1;k<=35;k++){
       cout<<"-";
    }
    cout<<endl;
}
void setAll(string n,double m,double r,int t){
    name=n;
    money=m;
    rate=r;
    time=t;
}
// Префиксная форма оператора инкремента:
MyMoney operator++(){
    money=money+1000;
    return *this;
}
// Постфиксная форма оператора инкремента:
MyMoney operator++(int){
    time++;
    return *this;
}
// Операторный метод для сложения двух объектов:
MyMoney operator+(MyMoney obj){
    MyMoney tmp;
    tmp.name="Почтальон Печкин";
    tmp.money=money+obj.money;
    tmp.rate=(rate>obj.rate)?rate:obj.rate;
    tmp.time=(time+obj.time)/2;
    return tmp;
}
```

```
};
// Операторная функция для вычитания объектов:
double operator-(MyMoney objX,MyMoney objY){
   return objX.getMoney()-objY.getMoney();
}
// Префиксная форма оператора декремента:
MyMoney operator--(MyMoney &obj){
   if(obj.money>1000){
      obj.money-=1000;
   }
   else{
      obj.money=0;
   }
   return obj;
}
// Постфиксная форма оператора декремента:
MyMoney operator--(MyMoney &obj,int){
   if(obj.time>0){
      obj.time--;
   }
   else{
      obj.time=0;
   }
   return obj;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Создание объекта:
   MyMoney objA("Кот Матроскин",1200,7,1);
   objA.showAll();
```

```
// Уменьшение значения поля time:
objA--;
objA.showAll();
// Уменьшение значения поля time:
objA--;
objA.showAll();
// Увеличение значения поля time:
objA++;
objA.showAll();
// Уменьшение значения поля money:
--objA;
objA.showAll();
// Уменьшение значения поля money:
--objA;
objA.showAll();
// Увеличение значения поля money:
++objA;
objA.showAll();
// Создание объекта:
MyMoney objB("Пес Шарик",1100,8,5);
objB.showAll();
// Создание объекта:
MyMoney objC;
// Вычисление суммы объектов:
objC=objA+objB;
objC.showAll();
// Вычисление разности объектов:
cout<<"Разница в доходах: "<<objC-objB<<endl;
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 5.5)

```
Имя: Кот Матроскин
Вклад: 1200
Ставка (%): 7
Период (лет): 1
Итоговая сумма: 1284
-----------------------------------
Имя: Кот Матроскин
Вклад: 1200
Ставка (%): 7
Период (лет): 0
Итоговая сумма: 1200
-----------------------------------
Имя: Кот Матроскин
Вклад: 1200
Ставка (%): 7
Период (лет): 0
Итоговая сумма: 1200
-----------------------------------
Имя: Кот Матроскин
Вклад: 1200
Ставка (%): 7
Период (лет): 1
Итоговая сумма: 1284
-----------------------------------
Имя: Кот Матроскин
Вклад: 200
Ставка (%): 7
Период (лет): 1
Итоговая сумма: 214
```

```
-----------------------------------

Имя: Кот Матроскин
Вклад: 0
Ставка (%): 7
Период (лет): 1
Итоговая сумма: 0
-----------------------------------
Имя: Кот Матроскин
Вклад: 1000
Ставка (%): 7
Период (лет): 1
Итоговая сумма: 1070
-----------------------------------
Имя: Пес Шарик
Вклад: 1100
Ставка (%): 8
Период (лет): 5
Итоговая сумма: 1616.26
-----------------------------------
Имя: Почтальон Печкин
Вклад: 2100
Ставка (%): 8
Период (лет): 3
Итоговая сумма: 2645.4
-----------------------------------
Разница в доходах: 1029.13
```

Проанализируем ту часть программы, которая связана с описанием и использованием операторных методов и операторных функций. Начнем с анализа описания операторных функций.

Операторная функция для вычитания объектов называется `operator-()`, возвращает результат типа `double`, и у нее объявлено два аргумента `objX` и `objY` — оба объекты класса `MyMoney`. Данное объявление означает, что

если мы от одного объекта класса MyMoney отнимем другой объект класса MyMoney, то результатом будет число типа double. Первый аргумент objX соответствует первому операнду (объект, из которого вычитается значение), а второй аргумент objY соответствует второму операнду (объект, который вычитается). В теле функции всего одна команда return objX.getMoney()-objY.getMoney(), которой результатом функции возвращается разность значений, возвращаемых методом getMoney() для каждого из объектов. Очевидно, что здесь идет речь о вычислении разности значений итоговых сумм по депозитам.

Операторная функция для префиксной формы оператора декремента называется operator--(). У функции один аргумент (обозначен как obj) типа MyMoney, причем этот аргумент передается по ссылке. Результатом функция возвращает также объект класса MyMoney. В теле функции в условном операторе проверяется условие obj.money>1000. Условие состоит в том, что значение поля money того объекта, к которому применяется операция инкремента, превышает значение 1000. Если это так, то командой obj.money-=1000 значение данного поля уменьшается на величину 1000. В противном случае командой obj.money=0 полю присваивается нулевое значение. Результатом функции возвращается объект obj.

> **НА ЗАМЕТКУ**
>
> В операторной функции operator--() изменяется аргумент функции (объект obj). Именно поэтому аргумент в функцию передается по ссылке, а не по значению.

Функция для постфиксной формы оператора декремента, по сравнению с префиксной формой, объявлена с дополнительным целочисленным аргументом. Причем поскольку роль аргумента чисто формальная, то имя аргумента можно не указывать, а только его тип. В теле функции запускается условный оператор с условием obj.time>0. Если целочисленное поле time неотрицательно, то командой obj.time-- его значение уменьшается на единицу. Если условие obj.time>0 не выполнено, то командой obj.time=0 полю присваивается нулевое значение.

Как отмечалось ранее, при перегрузке операторов обычно есть альтернатива: описывать операторные функции или операторные методы. В данном примере часть операций реализуется через операторные методы. Так, операторный метод operator++() для оператора инкремента описан без аргументов и как такой, что возвращает значением объект

класса MyMoney. Отсутствие аргументов у операторного метода объясняется просто: единственный операнд в данном случае отождествляется с объектом, из которого вызывается метод. Можно сказать и иначе: объект, к которому применяется операнд, является тем объектом, из которого вызывается операторный метод для обработки соответствующей команды.

В теле метода командой money=money+1000 изменяется значение поля money, после чего командой return *this объект, из которого вызывается метод, возвращается результатом выражения. Здесь мы воспользовались стандартным ключевым словом this, которое является указателем на объект, из которого вызван метод. Соответственно, *this обозначает сам объект.

ПОДРОБНОСТИ

Нелишним будет уточнить способ, которым функции и методы (операторные и не только) возвращают результат. Итак, вкратце ситуация следующая. Если метод или функция возвращает результат, то под возвращаемое значение выделяется область памяти. В эту область памяти записывается копия того значения, которое возвращается результатом метода или функции. Поэтому если метод или функция возвращает результатом некоторый объект, то на самом деле возвращается копия этого объекта.

Аналогичным образом описывается операторный метод для постфиксной формы оператора инкремента. Только теперь у метода есть формальный целочисленный аргумент (точнее, указан только идентификатор типа аргумента) — признак того, что речь идет о постфиксной форме оператора. В теле оператора командой time++ на единицу увеличивается значение поля time, после чего командой return *this возвращается объект, из которого вызывается операторный метод (объект, к которому применяется операция инкремента).

Операторный метод operator+() для сложения двух объектов описан с аргументом obj — объектом класса MyMoney, и результатом метод возвращает объект того же класса. Таким образом, первый операнд в соответствующем выражении отождествляется с объектом, из которого вызывается метод, а второй операнд в выражении отождествляется с объектом obj, переданным аргументом операторному методу. В теле метода командой MyMoney tmp создается локальный объект tmp класса

MyMoney. После создания объекта его полям присваиваются значения. Так, командой tmp.name="Почтальон Печкин" присваивается значение полю name (очевидно, значение не зависит от значений полей объектов, задействованных в операции сложения). Значение поля money для объекта tmp вычисляется командой tmp.money=money+obj.money. Таким образом, поле money объекта tmp представляет собой сумму значений полей money суммируемых объектов. Значение поля rate объекта tmp вычисляется командой tmp.rate=(rate>obj.rate)?rate:obj.rate. Здесь мы воспользовались тернарным оператором ?:. Результат выражения (rate>obj.rate)?rate:obj.rate вычисляется так: проверяется условие rate>obj.rate и если оно истинно, то результатом возвращается значение rate (поле объекта для первого операнда). Если условие rate>obj.rate ложно, результатом возвращается obj.rate (поле объекта obj для второго операнда). Таким образом, значением полю rate объекта tmp присваивается наибольшее из значений полей rate суммируемых объектов.

Наконец, командой tmp.time=(time+obj.time)/2 значение поля time вычисляется как среднеарифметическое для значений полей time суммируемых объектов. Следует учесть, что поскольку поле time целочисленное, то операция деления в приведенной выше команде выполняется на множестве целых чисел (деление нацело — дробная часть отбрасывается).

После присваивания значений полям объекта tmp данный объект возвращается результатом операторного метода.

В функции main() проверяется возможности описанных в программе операторных функций и методов. Для начала командой MyMoney objA("Кот Матроскин",1200,7,1) создается объект objA класса MyMoney. На этом объекте тестируется выполнение унарных операторов инкремента ++ и декремента --. Каждый раз после изменения параметров объекта командой objA.showAll() выполняется проверка значений его полей.

Для тестирования бинарных операций (сложение и вычитание объектов) командой MyMoney objB("Пес Шарик",1100,8,5) создается еще один объект (второй операнд), а для записи результат вычисления суммы создаем объект командой MyMoney objC. После этого выполняется команда objC=objA+objB, а значения полей объекта-результата проверяются с помощью инструкции objC.showAll(). Также в программе вычисляется разность объектов objC-objB (результат — число типа double).

Знакомство с наследованием

В следующем примере на основе класса `MyMoney` путем *наследования* создается класс `BigMoney`.

ТЕОРИЯ

Наследование — механизм, позволяющий создавать новые классы на основе уже существующих. Класс, служащий основной для создания другого класса, называется **базовым**. Класс, который создается на основе базового класса, называется **производным** классом. Чтобы создать производный класс на основе базового класса в описании производного класса после его имени указывается базовый класс. Между именем производного и базового класса ставится двоеточие и идентификатор, определяющий характер (тип) наследования. В качестве последнего могут использоваться ключевые слова `public` (открытое наследование), `private` (закрытое наследование) и `protected` (защищенное наследование). Независимо от типа наследования, в производном классе закрытые члены базового класса не наследуются.

Главное отличие класса `BigMoney` от класса `MyMoney` состоит в том, что в классе `BigMoney` появляется целочисленное поле `periods`, определяющее количество годовых начислений процентов по депозиту. Соответственно, коррективы вносятся в способ вычисления итоговой суммы на депозите и способ отображения информации об объекте.

Далее представлена программа, в которой использовано наследование классов:

Листинг 5.6. Наследование классов

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание базового класса:
class MyMoney{
public:
   // Поля базового класса:
```

```
    string name;
    double money;
    double rate;
    int time;
    // Методы базового класса:
    double getMoney(){
        double s=money;
            for(int k=1;k<=time;k++){
                s*=(1+rate/100);
            }
        return s;
    }
    void showAll(){
        cout<<"Имя: "<<name<<endl;
        cout<<"Вклад: "<<money<<endl;
        cout<<"Ставка (%): "<<rate<<endl;
        cout<<"Период (лет): "<<time<<endl;
        cout<<"Итоговая сумма: "<<getMoney()<<endl;
    }
    void setAll(string n,double m,double r,int t){
        name=n;
        money=m;
        rate=r;
        time=t;
    }
    // Конструктор базового класса (четыре аргумента):
    MyMoney(string n,double m,double r,int t){
        setAll(n,m,r,t);
    }
    // Конструктор базового класса (без аргументов):
    MyMoney(){
        setAll("",0,0,0);
    }
```

```
};
// Производный класс:
class BigMoney: public MyMoney{
public:
   // Поля производного класса:
   int periods;
   // Переопределение методов:
   double getMoney(){
      double s=money;
         for(int k=1;k<=time*periods;k++){
            s*=(1+rate/100/periods);
         }
      return s;
   }
   void showAll(){
      cout<<"Имя: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
      cout<<"Период (лет): "<<time<<endl;
      cout<<"Начислений в год: "<<periods<<endl;
      cout<<"Итоговая сумма: "<<getMoney()<<endl;
   }
   void setAll(string n,double m,double r,int t,int p){
      MyMoney::setAll(n,m,r,t);
      periods=p;
   }
   // Конструктор производного класса (пять аргументов):
   BigMoney(string n,double m,double r,int t,int p=1):
MyMoney(n,m,r,t){
      periods=p;
   }
   // Конструктор производного класса (без аргументов):
   BigMoney(): MyMoney(){
```

```
        periods=1;
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Создание объекта класса MyMoney:
    MyMoney objA("Кот Матроскин",1200,8,5);
    // Создание объектов класса BigMoney:
    BigMoney objB("Дядя Федор",1000,7,6,2);
    BigMoney objC("Пес Шарик",1500,6,8);
    BigMoney objD;
    objD.setAll("Почтальон Печкин",800,10,3,4);
    // Проверка характеристик объектов:
    objA.showAll();
    cout<<endl;
    objB.showAll();
    cout<<endl;
    objC.showAll();
    cout<<endl;
    objD.showAll();
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 5.6)

```
Имя: Кот Матроскин
Вклад: 1200
Ставка (%): 8
```

```
Период (лет): 5
Итоговая сумма: 1763.19

Имя: Дядя Федор
Вклад: 1000
Ставка (%): 7
Период (лет): 6
Начислений в год: 2
Итоговая сумма: 1511.07

Имя: Пес Шарик
Вклад: 1500
Ставка (%): 6
Период (лет): 8
Начислений в год: 1
Итоговая сумма: 2390.77

Имя: Почтальон Печкин
Вклад: 800
Ставка (%): 10
Период (лет): 3
Начислений в год: 4
Итоговая сумма: 1075.91
```

Базовый класс `MyMoney` описан достаточно просто: у него есть четыре поля (`name`, `money`, `rate` и `time`), три метода (`getMoney()`, `setAll()` и `showAll()`), а также две версии конструктора — без аргументов и с четырьмя аргументами. Нас он интересует не сам по себе, а как основа для создания, путем наследования, класса `BigMoney`. В описании класса `BigMoney` после имени класса (через двоеточие) указано ключевое слово `public` и имя класса `MyMoney`. Это означает, что класс `BigMoney` создается путем открытого наследования на основе класса `MyMoney` и поэтому все открытые члены класса `MyMoney` автоматически становятся членами класса `BigMoney`. Следовательно, непосредственно в классе `BigMoney` имеет смысл описывать только те члены, которые «добавляются» к унаследованным из

базового класса. По части полей в классе `BigMoney` добавляется только целочисленное поле `periods`. Также мы используем *переопределение методов*, унаследованных из класса `MyMoney`.

ТЕОРИЯ

При наследовании в производном классе можно изменить, или **переопределить**, код метода, который наследуется из базового класса. Для этого в производном классе метод (несмотря на то, что он наследуется), описывается заново с нужным кодом.
Не следует путать **перегрузку** и **переопределение** метода. При перегрузке создается дополнительная версия метода с таким же названием, но измененным прототипом. Переопределение метода может иметь место только при наследовании и состоит в том, что создается новая версия метода вместо той, что наследуется.

Несмотря на то, что в классе `BigMoney` наследуются методы `getMoney()`, `showAll()` и `setAll()`, нас они не очень устраивают, поскольку у класса теперь есть дополнительное поле, что, кроме прочего, влечет за собой изменение способа вычисления итоговой суммы по депозиту. Другими словами, нужно чтобы такие методы у класса `BigMoney` были, но необходимо внести изменения в их программный код. Мы поступаем просто и незатейливо — в теле класса `BigMoney` описываем данные методы заново.

Программный код метода `getMoney()` изменяется так, чтобы в учет принималось наличие нескольких выплат в году. Основных изменений (по сравнению с кодом метода из класса `MyMoney`) два: количество периодов начисления процентов умножается на значение поля `periods`, а процентная ставка делится на значение поля `periods`.

НА ЗАМЕТКУ

Фактически, если раньше вычисления производились в соответствии с формулой $m(1 + (r/100))^t$, то теперь рабочая формула имеет вид $m(1 + r/(100p))^{pt}$, где параметры m, r, t и p обозначают соответственно начальную сумму вклада, годовую процентную ставку, время, на которое размещается вклад, и количество периодов в году для начисления процентов.

Программный код метода `showAll()` в производном классе отличается от кода аналогичного метода в базовом классе только командой `cout<<"Начислений в год: "<<periods<<endl`, которой в консольное окно выводится значение поля `periods`.

ПОДРОБНОСТИ

В теле метода `showAll()` в производном классе есть команда вызова метода `getMoney()`. Поскольку метод `getMoney()` переопределен в производном классе, то будет вызываться именно эта переопределенная версия метода.

Метод `setAll()` описан в производном классе с пятью, а не с четырьмя аргументами, как было в базовом классе. Дополнительный пятый аргумент нужен для передачи значения полю `periods`. В теле метода `setAll()` всего две команды. Командой `MyMoney::setAll(n,m,r,t)` вызывается старая, унаследованная из базового класса `MyMoney` версия метода `setAll()` с четырьмя аргументами. В результате получают значения поля `name`, `money`, `rate` и `time`. А значение полю `periods` присваивается в явном виде командой `periods=p`.

ПОДРОБНОСТИ

При переопределении методов в производном классе «старые» версии, которые наследуются из базового класса, на самом деле никуда не деваются. Технически они существуют. Просто «перекрываются» новыми версиями методов. При необходимости эти «спрятанные» версии методов из базовых классов можно «извлечь» и использовать по назначению. Однако для этого необходимо явно указать, что подразумевается та версия метода, которая наследуется из базового класса. Для этого перед именем метода указывается имя базового класса, из которого метод наследуется. Между именем класса и именем метода ставится оператор расширения контекста `::` (два двоеточия).

Достойно внимания и то обстоятельство, что в классе `BigMoney` метод `setAll()` описывается с пятью аргументами, а в классе `MyMoney` у метода `setAll()` четыре аргумента. Может сложиться впечатление, что в производном классе выполняется перегрузка метода `setAll()`, но это не так. В производном классе версия метода `setAll()` с пятью аргументами «перекрывает» наследуемую из базового класса версию метода `setAll()` с четырьмя аргументами. Так что прямой (без указания базового класса) доступ в производном классе есть только к той версии метода, что описана в производном классе.

В производном классе описаны две версии конструктора: без аргументов и с пятью аргументами (один из аргументов имеет значение по умолчанию). Специфика описания конструктора в производном классе

такова, что необходимо явно указывать вызов конструктора базового класса. Например, при описании конструктора класса `BigMoney` с пятью аргументами после названия конструктора и списка аргументов через двоеточие указана инструкция `MyMoney(n,m,r,t)`, которая означает вызов конструктора базового класса с соответствующими аргументами. После того, как выполнение конструктора базового класса завершается, командой `periods=p` присваивается значение полю `periods`. Проще говоря, в теле конструктора производного класса программируются те дополнительные действия, которые следует выполнить после вызова конструктора базового класса.

> **НА ЗАМЕТКУ**
>
> В теле конструктора базового класса `MyMoney` вызывается метод `setAll()` с четырьмя аргументами. Поскольку вызов происходит в конструкторе именно базового класса, то и вызывается версия метода из базового класса.

Аналогичным образом описывается версия конструктора производного класса без аргументов. Только теперь конструктор базового класса также вызывается без аргументов, а полю `periods` присваивается единичное значение.

В функции `main()` создаются несколько объектов, и затем проверяются их характеристики. Так, командой `MyMoney objA("Кот Матроскин", 1200,8,5)` создается объект базового класса `MyMoney`. Командами `BigMoney objB("Дядя Федор",1000,7,6,2)`, `BigMoney objC("Пес Шарик",1500,6,8)` и `BigMoney objD` создаются объекты производного класса `BigMoney`. Поскольку в производном классе описан конструктор без аргументов и с пятью аргументами (при том, что один из аргументов имеет значение по умолчанию), то при создании объекта в конструктор можно передавать четыре или пять аргументов, или не передавать их совсем. Командой `objD.setAll("Почтальон Печкин",800,10,3,4)` задаются значения полей объекта производного класса. Проверка значений полей объектов выполняется с помощью метода `showAll()`.

Задачи для самостоятельного решения

1. Напишите программу, в которой создается класс для описания такого физического объекта, как «параллелепипед». У класса три поля: ширина, глубина и высота, а также метод для вычисления объема

(произведение значений полей объекта). Предусмотреть наличие конструктора и метода для отображения характеристик объекта. Рассмотрите механизм наследования, создав на основе класса производный с дополнительным полем, определяющим массу «параллелепипеда». Добавить в производном классе метод для вычисления плотности материала, из которого сделан «параллелепипед» (массу нужно поделить на объем). Подумайте над тем, как добавить в класс операторные методы или описать операторные функции для выполнения некоторых простых операций с объектами. Например, деление объекта на число может приводить к тому, что масса «параллелепипеда» уменьшается в определенное количество раз. Или: в результате сложения двух объектов получаем новый объект. Его объем равен сумме объемов исходных объектов, его масса равна сумме масс исходных объектов, а все три линейных размера одинаковые.

2. Напишите программу с классом для реализации комплексных чисел. У класса два поля (действительная и мнимая части комплексного числа). Описать для класса операторные функции и методы, позволяющие выполнять базовые математические операции с объектами класса. Создать на основе базового класса производный класс, в котором реализовать представление комплексного числа в тригонометрической форме. Напомним, что комплексное число $z = x + iy$ имеет действительную x и мнимую y части, а мнимая единица i такая, что $i^2 = -1$. Основные математические операции с комплексными числами выполняются так же, как и с обычными числами, с учетом лишь того, что $i^2 = -1$. В тригонометрической форме комплексное число представимо в виде $z = |z|\exp(i\varphi)$, где модуль числа $|z| = \sqrt{x^2 + y^2}$, а аргумент φ такой, что $\sin(\varphi) = y/(\sqrt{x^2 + y^2})$ и $\cos(\varphi) = x/(\sqrt{x^2 + y^2})$. Учесть, что имеет место тождество $\exp(i\varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$.

3. Напишите программу с классом для реализации векторов в двумерном пространстве. У класса два поля (координаты вектора). Предусмотреть выполнение базовых операций с векторами (таких, например, как сложение и вычитание векторов, умножение и деление векторов на число, вычисление скалярного произведения векторов). Путем наследования создать класс для реализации векторов в трехмерном пространстве (у такого вектора три координаты). Напомним, что вектор в трехмерном пространстве однозначно описывается тремя координатами. Операции с векторами выполняются так: если $\vec{a} = (a_1,a_2,a_3)$ и $\vec{b} = (b_1,b_2,b_3)$ векторы, а λ и ξ — числа, то $\lambda\vec{a} + \xi\vec{b} = (\lambda a_1 + \xi b_1, \lambda a_2 + \xi b_2, \lambda a_3 + \xi b_3)$, а скалярное произведение $\vec{a} \cdot \vec{b} = a_1b_1 + a_2b_2 + a_3b_3$.

4. Напишите программу с классом для описания «легкового автомобиля». Через поля класса реализуются такие характеристики, как марка, цвет, объем топливного бака (в литрах), норма потребления бензина (литров на 100 километров). Предусмотреть, кроме прочего, наличие метода, позволяющего вычислить расстояние, которое проедет автомобиль, израсходовав полный бак бензина (вычисляется как умноженное на 100 отношение объема топливного бака к норме потребления бензина). На основе класса путем наследования создать производный класс для описания «грузового автомобиля». Добавить поле, описывающее грузоподъемность автомобиля (максимальная масса груза в тоннах, которую может перевозить автомобиль) и метод, позволяющий вычислить себестоимость транспортировки одной тонный груза на один километр. Результат метода рассчитывается так: норма потребления бензина делится на 100, умножается на цену литра бензина и делится на грузоподъемность. Цена литра бензина передается аргументом методу. Рассмотреть возможность определения для класса операторных методов и операторных функций.

5. Напишите программу с классом для описания ситуации, когда денежный вклад, размещаемый на депозите, состоит из двух частей. Время размещения депозита одно и то же для обеих частей вклада, а процентная ставка для каждой части вклада своя. Фактически речь идет о расширенном варианте задачи, рассмотренной выше в данной главе. Предлагается реализовать те же операции, что описывались для объектов класса `MyMoney`, но только с учетом новой постановки задачи.

Рекомендации для самостоятельной работы

- Предложите объекты и явления из реальной жизни, которые можно было бы описать на программном уровне в рамках концепции классов и объектов. Составьте соответствующие программы.
- Какие математические объекты и операции могут быть запрограммированы с применением объектно-ориентированного подхода? Предложите свои программы.
- Предложите варианты решения тех задач, которые рассматривались в предыдущих главах (или предлагались там для самостоятельного решения), с использованием классов и объектов.

Глава 6

ИСПОЛЬЗОВАНИЕ КЛАССОВ И ОБЪЕКТОВ

— А почему он роет на дороге?
— Да потому, что в других местах все уже перерыто и пересеяно.

из к/ф «31 июня»

В этой главе мы рассмотрим ряд специфических вопросов, связанных с использованием классов и объектов. Кроме прочего, мы остановимся на таких темах как:

- создание массива объектов;
- создание объекта с полем-массивом;
- работа с указателями на объект;
- создание функторов;
- индексирование объектов;
- реализация конструктора создания копии объекта;
- использование виртуальных методов;
- применение множественного наследования;
- особенности работы с закрытыми членами базового класса при наследовании;
- доступ к объектам производного класса через объектные переменные базового класса.

Рассматриваемые далее примеры и задачи призваны проиллюстрировать наиболее важные механизмы ООП, реализуемые в C++.

Указатель на объект

Для объекта, как и для значения простого (базового типа) можно создать указатель. Указатель на объект — переменная, значением которой является адрес объекта.

ТЕОРИЯ

Получить адрес объекта можно с помощью инструкции &. Если перед указателем поставить *, получим значение — то есть сам объект. Если имеется указатель на объект, то для получения доступа к полям и методам объекта через указатель используют оператор ->.

Небольшой пример использования указателей на объекты представлен в листинге 6.1.

Листинг 6.1. Доступ к объектам через указатель

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание класса:
class MyClass{
public:
// Текстовое поле:
string name;
// Целочисленное поле:
int number;
// Метод для отображения значения полей:
void show(){
   cout<<"Поле name: "<<name<<endl;
   cout<<"Поле number: "<<number<<endl;
}
};
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Создание объектов:
   MyClass objA,objB;
   // Объявление указателя на объект:
```

```
MyClass* p;
// Адрес объекта записывается в указатель:
p=&objA;
// Присваивание значений полям объекта
// через указатель:
p->name="Объект objA";
p->number=111;
// Вызов метода через указатель:
p->show();
// Новое значение указателя:
p=&objB;
// Присваивание значений полям объекта
// через указатель:
p->name="Объект objB";
p->number=222;
// Вызов метода через указатель:
   p->show();
   cout<<"Проверяем объекты\n";
   // Проверка значений полей объектов:
   objA.show();
   objB.show();
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 6.1)

```
Поле name: Объект objA
Поле number: 111
Поле name: Объект objB
Поле number: 222
Проверяем объекты
```

```
class Alpha{
public:
   // Виртуальный метод:
   virtual void show(){
      cout<<"Класс Alpha"<<endl;
   }
   void showAll(){
      show();
   }
};
class Bravo: public Alpha{
public:
   void show(){
      cout<<"Класс Bravo"<<endl;
   }
};
int main(){
   system("chcp 1251>nul");
   Bravo obj;
   obj.show();
   obj.showAll();
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы теперь следующий:

Результат выполнения программы (из листинга 6.9)

```
Класс Bravo
Класс Bravo
```

Таким образом, благодаря тому, что метод `show()` теперь виртуальный, при вызове метода `showAll()` из объекта производного класса вызывается переопределенная в классе `Bravo` версия метода `show()`.

НА ЗАМЕТКУ

Переопределяемые методы обычно объявляются виртуальными.

Множественное наследование

В C++ производный класс может создаваться на основе сразу нескольких базовых классов. Об этом следующий пример, представленный в листинге 6.10.

Листинг 6.10. Множественное наследование

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Первый базовый класс:
class Alpha{
public:
   // Поле:
   int alpha;
   // Конструктор:
   Alpha(int a){
      alpha=a;
   }
   // Метод:
   void show(){
      cout<<"Класс Alpha: "<<alpha<<endl;
   }
};
// Второй базовый класс:
class Bravo{
public:
   // Поле:
   int bravo;
   // Конструктор:
```

```
// Целочисленная переменная:
int n=10;
// Динамическое создание объекта:
MyClass* pnt=new MyClass;
// Доступ к полю объекта через указатель на него:
pnt->code='A';
// Указатель на объект:
MyClass *p;
// Начальное значение указателя:
p=pnt;
// Создание цепочки объектов:
for(int k=1;k<=n;k++){
   // Создание очередного объекта:
   p->next=new MyClass;
   // Значение поля code для нового объекта:
   p->next->code=p->code+1;
   // Новое значение указателя – адрес
   // вновь созданного объекта:
   p=p->next;
}
// Нулевая ссылка в поле next для последнего
// объекта в цепочке:
p->next=0;
// Вызов метода для первого объекта в цепочке:
pnt->show();
   cout<<endl;
   // Удаление цепочки объектов:
   deleteAll(pnt);
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 6.2)

```
A B C D E F G H I J K
Объект с полем K удален
Объект с полем J удален
Объект с полем I удален
Объект с полем H удален
Объект с полем G удален
Объект с полем F удален
Объект с полем E удален
Объект с полем D удален
Объект с полем C удален
Объект с полем B удален
Объект с полем A удален
```

Проанализируем программный код примера. В классе `MyClass` описано два поля: символьное поле `code` и поле `next`, которое является указателем на объект класса `MyClass` (поле объявлено с помощью инструкции `MyClass* next`). Также в классе описан деструктор. В теле деструктора командой `cout<<"Объект с полем "<<code<<" удален\n"` отображается сообщение об удалении объекта и указывается, какое значение поля `code` удаляемого объекта. Деструктор нам нужен исключительно для того, чтобы сделать наглядным процесс удаления объекта. Проще говоря, когда будет удаляться тот или иной объект, мы это увидим (практически в буквальном смысле).

Еще в классе `MyClass` описан метод `show()`, код которого достоин отдельного внимания. Метод не возвращает результат, и у него нет аргументов. При вызове метода сначала командой `cout<<code<<" "` отображается значение поля `code` объекта, из которого вызывается метод. Но это еще не все. После отображения значения поля `code` в условном операторе проверяется значение поля `next`. Если значение поля отлично от нуля, то командой `next->show()` метод `show()` вызывается для следующего объекта в цепочке.

ПОДРОБНОСТИ

В поле `next` объекта записан адрес следующего элемента. Для последнего объекта в цепочке следующего объекта нет, поэтому

последний объект не ссылается на другой объект. Предполагается, что поле next последнего в цепочке объекта содержит нулевое значение. Таким образом, равенство нулю поля next интерпретируется как признак последнего объекта в цепочке.

Здесь также уместно напомнить, что отличные от нуля значения интерпретируются как логическое значение true, а нулевое значение интерпретируется как логическое значение false. Поэтому отличный от нуля адрес в поле next (переданный условием в условный оператор) интерпретируется как логическое значение true, а поле next с нулевым значением в последнем объекте интерпретируется как значение false.

Следовательно, при вызове метода show() из объекта отображается значение code объекта, после чего автоматически вызывается метод show() для следующего объекта, и так далее. В результате при вызове метода show() последовательно отображаются значения полей code всех объектов в цепочке, начиная с того, с которого вызывается метод.

Поскольку объекты в программе создаются динамические, то их следует удалять. Проблема связана с тем, что речь идет о цепочке объектов и прямой доступ имеется только к первому объекту. Поэтому при удалении объектов нужно «пройти» по цепочке и удалить каждый объект. Для решения задачи описывается функция deleteAll(). Функция не возвращает результат, и у нее один аргумент (обозначен как q), который является указателем на объект класса MyClass. Код тела функции начинается с условного оператора (в упрощенной форме), в котором проверяется условие q->next. Условие q->next состоит в том, что адрес в поле next объекта, переданного через указатель q аргументом функции, не равен нулю. Проще говоря, данное условие истинно, если объект, адрес которого записан в указатель q, не является последним объектом в цепочке. Если так, то командой deleteAll(q->next) функция deleteAll() вызывается для следующего объекта в цепочке. После условного оператора командой delete q удаляется объект, адрес которого содержится в указателе q.

НА ЗАМЕТКУ

То есть общая схема такая: функция deleteAll() вызывается с указанием адреса некоторого объекта, который должен быть удален. Но перед его удалением автоматически вызывается функция deleteAll() для удаления следующего объекта. Перед удалением следующего объекта опять вызывается функция deleteAll() для уда-

ления еще одного объекта в цепочке, и так далее — пока очередь не дойдет до последнего объекта. Он удаляется, после чего удаляется предыдущий объект, затем объект перед ним, и так далее. Таким образом, объекты будут удаляться с конца цепочки в ее начало. При вызове функции `deleteAll()` аргументом ей передается первый объект в цепочке.

В главной функции программы объявляется целочисленная переменная `n`. Количество объектов в цепочке на единицу больше значения этой переменной. Командой `MyClass* pnt=new MyClass` создается динамический объект и его адрес записывается в указатель `pnt`. Командой `pnt->code='A'` полю `code` этого объекта присваивается значение `'A'`. Командой `MyClass *p` создается указатель `p` на объект класса `MyClass` и значение указателю присваивается командой `p=pnt`. Поэтому в начальный момент указатель `p` содержит адрес того же объекта, что и указатель `pnt`. Далее запускается оператор цикла, в котором индексная переменная `k` принимает значения от `1` до `n` включительно. За каждый цикл последовательно выполняются три команды. Сначала командой `p->next=new MyClass` создается очередной динамический объект и его адрес записывается в поле `next` объекта, определяемого указателем `p` (следующий объект в цепочке). Командой `p->next->code=p->code+1` полю `code` вновь созданного объекта присваивается увеличенное на единицу значение поля `code` текущего (адрес которого записан в указателе `p`) объекта.

ПОДРОБНОСТИ

Обращение к полю `code` объекта, адрес которого записан в поле `next` (а поле `next`, в свою очередь, относится к объекту, определяемому указателем `p`), выглядит как `p->next->code`. Такая инструкция находится в левой части команды присваивания значения `p->next->code=p->code+1`. С другой стороны, у объекта, определяемого указателем `p`, помимо поля `next` имеется еще и поле `code`. Обращение к этому полю выполняется с помощью инструкции `p->code`. Поле символьное, поэтому если к нему прибавить единицу (выражение `p->code+1`), получим следующую букву по отношению к той, что записана в поле `code` объекта, определяемого указателем `p`. Именно такое значение (имеется в виду выражение `p->code+1`) присваивается полю `p->next->code`.

Наконец, командой `p=p->next` в указатель `p` записывается значение поля `next` объекта, адрес которого до этого был записан в данном указателе.

Таким образом, указатель «перебрасывается» на следующий объект в цепочке.

После выполнения оператора цикла цепочка объектов создана. Осталось только командой `p->next=0` присвоить нулевое значение полю `next` последнего объекта в цепочке. Здесь мы воспользовались тем, что по завершении оператора цикла указатель `p` будет содержать адрес последнего объекта в цепочке объектов.

Командой `pnt->show()` из первого объекта в цепочке вызывается метод `show()`. Благодаря автоматическому вызову этого метода для всех прочих объектов, в консольном окне в строчку через пробел отображаются значения символьных полей объектов для всей цепочки. После этого командой `deleteAll(pnt)` цепочка объектов удаляется из памяти.

Создание массива объектов

Нередко приходится сталкиваться с ситуацией, когда нужно создать массив, элементами которого являются объекты. Именно такой пример рассмотрим далее. В некотором смысле он напоминает предыдущий.

ТЕОРИЯ

Массив объектов создается точно так же, как и массив элементов других типов, с той лишь разницей, что в качестве типа элемента указывается название класса.

Важное условие состоит в том, что класс, на основе которого создаются элементы массива, должен иметь конструктор без аргументов, поскольку именно такой конструктор вызывается при создании элементов массива.

Основу представленного в листинге 6.3 программного кода составляет описание класса `MyWords`. У класса есть текстовое (тип `string`) поле `word`, логическое (тип `bool`) поле `state`, описан конструктор без аргументов и метод `read()`. В конструкторе командами `word=""` и `state=true` полям присваиваются значения. Метод `read()` (метод не возвращает результат, и у него нет аргументов) состоит из команды `cout<<word<<" "` (командой отображается значение поля `word` объекта, из которого вызывается метод) и условного оператора в упрощенной форме. В условном операторе проверяемым условием выступает значение поля `state`. Если значение

поля равно `true`, то командой `(this+1)->read()` вызывается метод `read()` для следующего объекта в массиве. Здесь мы воспользовались двумя обстоятельствами:

- элементы массива в памяти расположены подряд;
- при прибавлении к указателю целого числа получаем указатель на область памяти, отстоящую от исходной на количество ячеек, определяемое прибавляемым числом.

Ключевое слово `this` в теле метода является указателем на объект, из которого вызывается метод. Значением выражения `this+1` является указатель на объект, находящийся в памяти за данным объектом. Поскольку здесь речь идет об объектах, которые являются элементами массива, то значение выражения `this+1` — указатель на следующий элемент (он же объект) массива. Поэтому команда `(this+1)->read()` означает вызов метода `read()` для следующего (по сравнению с текущим) объекта в массиве.

НА ЗАМЕТКУ

Получается так, что при вызове метода `read()` из объекта в массиве данный метод начнет вызываться из каждого следующего элемента в массиве. Так будет продолжаться до тех пор, пока в массиве не встретится объект со значением поля `state` равным `false`. Как мы увидим далее, поле с таким значением будет у последнего объекта в массиве.

Весь программный код примера представлен ниже:

Листинг 6.3. Массив объектов

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание класса:
class MyWords{
public:
   // Текстовое поле:
   string word;
```

```
    // Поле логического типа:
    bool state;
    // Конструктор без аргументов:
    MyWords(){
        word="";
        state=true;
    }
    // Метод для считывания значения текстового поля:
    void read(){
        // Отображение значения поля объекта:
        cout<<word<<" ";
        // Проверка значения поля state:
        if(state){
            // Вызов метода для соседнего объекта:
            (this+1)->read();
        }
    };
    // Главная функция программы:
    int main(){
        // Изменение кодировки консоли:
        system("chcp 1251>nul");
        // Размер массива:
        const int n=5;
        // Текстовый массив:
        string numbers[n]={"один","два","три","четыре","пять"};
        // Массив объектов:
        MyWords words[n];
        // Значение для поля state последнего
        // объекта в массиве:
        words[n-1].state=false;
        // Присваивание значения полю word для
```

```
// объектов в массиве:
for(int k=0;k<n;k++){
   words[k].word=numbers[k];
}
// Вызов метода:
words[0].read();
   cout<<endl;
   words[2].read();
   cout<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

В главной функции программы объявляется целочисленная константа `n` со значением `5` (размер массива). Вспомогательный массив с текстовыми значениями создаем и инициализируем командой `string numbers[n]={"один","два","три","четыре","пять"}`.

Данный массив содержит значения, которые будут присваиваться текстовым полям объектов из массива, объявляемого инструкцией `MyWords words[n]`. Конструктор без аргументов, который автоматически вызывается для создания каждого из объектов в массиве, присваивает полю `state` каждого из объектов значение `true`. Командой `words[n-1].state=false` последнему объекту в массиве `words` для поля `state` задается значение `false`. После этого запускается оператор цикла, в котором перебираются элементы массива `words` и за каждый цикл командой `words[k].word=numbers[k]` полю `word` очередного объекта в массиве `words` присваивается значение соответствующего элемента из текстового массива `numbers`.

После того, как поля объектов в массиве `words` заполнены, из первого объекта в массиве вызывается метод `read()` (команда `words[0].read()`). В результате для каждого из объектов в массиве `words` отображается значение поля `word`. Если же воспользоваться командой `words[2].read()`, то значение поля `word` будет отображаться для всех объектов в массиве, начиная с третьего (индекс `2`) и до конца массива. Результат выполнения программы такой:

Результат выполнения программы (из листинга 6.3)

```
один два три четыре пять
три четыре пять
```

Первая строка в консольном окне появляется вследствие выполнения команды `words[0].read()`, вторая — при выполнении команды `words[2].read()`.

Массив как поле класса

В некоторых случаях имеет смысл или просто необходимо, чтобы в классе полем был массив. Рассмотрим такой случай. В представленном далее примере описывается класс `Taylor`, предназначенный для реализации ряда Тейлора для различных функций.

ПОДРОБНОСТИ

Для функции $f(x)$ рядом Тейлора в окрестности точки x_0 называется представление функции в виде $f(x) = \sum_{k=0}^{\infty} c_k(x - x_0)^k$, где коэффициенты $c_k = (f^{(k)}(x_0))/k!$. Ряд Тейлора можно использовать для аппроксимирования функций. В этом случае ряд обрывается, и имеет место оценка $f(x) \approx \sum_{k=0}^{n} (c_k(x - x_0)^k)$. Если речь идет о разложении в окрестности нуля (то есть при $x_0 = 0$), то для функции $f(x)$ получаем оценку в виде полинома $f(x) \approx \sum_{k=0}^{n} (c_k x^k)$, которая дает приближенное значение функции в точке x. В таком случае представление функции $f(x)$ в виде ряда сводится к определению коэффициентов c_k (индекс $k = 0, 1, \ldots, n$).
В рассматриваемой далее программе описывается класс с полем-массивом, и в этот массив заносятся коэффициенты, определяющие ряд (ограниченный) Тейлора для некоторой функции. Каждый объект такого класса соответствует определенной функциональной зависимости.

В листинге 6.4 представлена программа, в которой описывается класс `Taylor`. У класса есть поле-массив. Массив предназначен для запоминания коэффициентов, которые мы отождествляем с коэффициентами разложения функции в ряд Тейлора (в окрестности нуля).

У класса есть две версии конструктора и метод, позволяющий вычислять значение ряда Тейлора в указанной точке. Рассмотрим код программы:

Листинг 6.4. Ряд Тейлора для функции

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Размер массива:
const int n=10;
// Класс для реализации ряда Тейлора:
class Taylor{
public:
   // Поле – массив:
   double a[n];
   // Конструктор с одним числовым аргументом:
   Taylor(double p=0){
      for(int k=0;k<n;k++){
         a[k]=p;
      }
   }
   // Конструктор с аргументом – указателем:
   Taylor(double* b){
      for(int k=0;k<n;k++){
         a[k]=b[k];
      }
   }
   // Метод для вычисления значения ряда:
   double value(double x){
      double s=0,q=1;
      for(int k=0;k<n;k++){
         s+=a[k]*q;
         q*=x;
      }
      return s;
```

```
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Числовой массив:
    double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835};
    // Создание объектов:
    Taylor myexp,f(1),mytan(b);
    // Обращение к элементу массива — поля объекта:
    myexp.a[0]=1;
    // Заполнение массива — поля объекта:
    for(int k=1;k<n;k++){
        myexp.a[k]=myexp.a[k-1]/k;
    }
    // Аргумент для вычисления значения функции и ряда:
    double x=1.0;
    // Вычисление значений для ряда и функции:
    cout<<myexp.value(x)<<" vs. "<<exp(x)<<endl;
    cout<<mytan.value(x)<<" vs. "<<tan(x)<<endl;
    cout<<f.value(x/2)<<" vs. "<<1/(1-x/2)<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Размер массива `a` (который описан как поле в классе `Taylor`) определяется глобальной целочисленной константой `n`. Конструктор класса `Taylor` описан с одним `double`-аргументом, который имеет значение по умолчанию. Аргумент конструктора определяет значения элементов массива `a`, которые им присваиваются при создании объекта. Если при вызове конструктора аргумент не указан, то у всех элементов массива будут нулевые значения. Также есть версия конструктора, аргументом

которой передается массив с коэффициентами, которые присваиваются значениями элементам массива a.

НА ЗАМЕТКУ

В общем случае массив в функцию или метод передается с помощью двух параметров: указателя на первый элемент массива и количества элементов в массиве. В данном же случае размеры всех массивов, с которыми нам предстоит иметь дело, имеют одинаковый размер, который определяется глобальной константой n. Поэтому нет необходимости передавать в конструктор второй параметр, определяющий размер массива. В результате у конструктора класса Taylor один аргумент, который является указателем на значение типа double (указатель на первый элемент массива).

В классе Taylor описан метод value(), предназначенный для вычисления значения ряда Тейлора в точке (для указанного аргумента). Метод возвращает значение типа double и у него один аргумент (обозначен как x) типа double. В теле метода локальной переменной s присваивается начальное значение 0, а локальная переменная q получает начальное значение 1.

Далее запускается оператор цикла, в котором с помощью индексной переменной k перебираются элементы массива a. За каждый цикл командой s+=a[k]*q значение переменной s увеличивается на очередное слагаемое, а командой q*=x вычисляется значение для переменной q на следующей итерации. По завершении вычислений значение переменной s возвращается результатом метода.

ПОДРОБНОСТИ

Нам фактически необходимо вычислить сумму $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$, где через $a_0, a_1, \ldots, a_{n-1}$ обозначены элементы массива a, а через x обозначен аргумент метода value(). Сумму можно записать в виде $a_0q_0 + a_1q_1 + \ldots + a_{n-1}q_{n-1}$. Здесь мы обозначили $q_k = x^k$ (аналог локальной переменной q в теле метода value()). Начальное значение $q_0 = 1$, а для прочих итераций несложно заметить, что $q_{k+1} = q_kx$. Именно этим соотношением мы воспользовались, когда вычисляли значение переменной q для следующей итерации.

В главной функции программы командой double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835} создается и инициализируется числовой

массив, который мы используем для передачи аргументом конструктору класса `Taylor`.

ПОДРОБНОСТИ

Ряд Тейлора для тангенса имеет вид $tg(x) \approx x + (1/3)x^3 + (2/15)x^5 + (17/315)x^7 + (62/2835)x^9$. Отсутствие слагаемых с четными показателями степени аргумента означает, что числовые коэффициенты для таких слагаемых равны нулю. При определении коэффициентов массива `b` числа в числителе указаны с десятичной точкой, чтобы избежать целочисленного деления.

Также в программе использованы следующие приближенные представления для функций: $1/(1-x) \approx 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9$ (оценка справедлива только при $|x| < 1$) и $\exp(x) \approx 1 + x + (x^2/2!) + (x^3/3!) + (x^4/4!) + (x^5/5!) + (x^6/6!) + (x^7/7!) + (x^8/8!) + (x^9/9!)$.

Командой `Taylor myexp,f(1),mytan(b)` создаются три объекта класса `Taylor`. При создании объекта `myexp()` вызывается конструктор с нулевым (значение по умолчанию) числовым аргументом. В результате все элементы массива `a` в объекте `myexp` получат нулевые значения. При создании объекта `f` вызывается конструктор с числовым аргументом, значение которого равно `1`. Такое значение будет у всех элементов массива `a` в объекте `f` (получаем ряд для функции $f(x) = 1/(1 - x)$). Наконец, при создании объекта `mytan` аргументом конструктору передается имя массива `b`, в результате чего у элементов массива `a` в объекте `mytan` будут такие же значения, как и у элементов в массиве `b` (ряд для тангенса).

Для объекта `myexp` значения коэффициентов массива вычисляются с помощью оператора цикла. Перед вызовом оператора цикла командой `myexp.a[0]=1` первому элементу присваивается единичное значение, после чего запускается оператор цикла, в котором индексная переменная `k` перебирает элементы (индексы элементов), начиная со второго.

За каждый цикл выполняется команда `myexp.a[k]=myexp.a[k-1]/k`, которой значение элемента массива вычисляется на основе значения предыдущего элемента в том же массиве.

ПОДРОБНОСТИ

Если через a_k обозначить элементы массива `a`, то тогда $a_k = 1/k!$. Легко заметить, что $a_k/a_{k-1} = 1/k$, или $a_k = a_{k-1}/k$.

Командой `double x=1.0` объявляется переменная `x`, используемая для передачи аргументом при вычислении значения ряда. Делается это с помощью инструкций `myexp.value(x)` (экспонента в точке `x`), `mytan.value(x)` (тангенс в точке `x`) и `f.value(x/2)` (значение ряда для функции $f(x) = 1/(1 - x)$ в точке `x/2`).

Для сравнения приводятся точные значения, вычисленные с помощью встроенных математических функций `exp()` (экспонента), `tan()` (тангенс) и непосредственно вычисляется значение выражения `1/(1-x/2)`.

> **НА ЗАМЕТКУ**
>
> Для использования математических функций в программе подключается заголовок `<cmath>`.

Результат выполнения программы такой:

Результат выполнения программы (из листинга 6.4)

```
2.71828 vs. 2.71828
1.5425 vs. 1.55741
1.99805 vs. 2
```

Видим, что совпадение достаточно неплохое. Для увеличения точности необходимо взять больше слагаемых в ряде.

Функторы и индексация объектов

Рассмотренную выше задачу можно реализовать в виде программного кода более эффектно. Для этого следует в классе `Taylor` описать два операторных метода:

- метод для оператора «квадратные скобки» (метод `operator[]()`), позволяющий индексировать объекты (обращаться к элементам массива `a` в объекте указывая индекс непосредственно после имени объекта);
- метод для оператора «круглые скобки» (метод `operator()()`), позволяющий «вызывать» объект как функцию.

Новая версия программы представлена в листинге 6.5.

Листинг 6.5. Поле-массив и перегрузка операторов

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

// Размер массива:
const int n=10;
// Класс для реализации ряда Тейлора:
class Taylor{
private:
   // Поле – массив:
   double a[n];
public:
   // Операторный метод для индексирования объекта:
   double &operator[](int k){
      return a[k];
   }
   // Конструктор с одним числовым аргументом:
   Taylor(double p=0){
      for(int k=0;k<n;k++){
         (*this)[k]=p; // Индексирование объекта
      }
   }
   // Конструктор с аргументом – указателем:
   Taylor(double* b){
      for(int k=0;k<n;k++){
         (*this)[k]=b[k]; // Индексирование объекта
      }
   }
   // Операторный метод для вычисления значения ряда:
   double operator()(double x){
```

```
    double s=0,q=1;
    for(int k=0;k<n;k++){
        s+=(*this)[k]*q; // Индексирование объекта
        q*=x;
    }
    return s;
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Числовой массив:
    double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835};
    // Создание объектов:
    Taylor myexp,f(1),mytan(b);
// Индексирование объекта:
myexp[0]=1;
// Заполнение массива – поля объекта:
for(int k=1;k<n;k++){
    myexp[k]=myexp[k-1]/k; // Индексирование объекта
}
// Аргумент для вычисления значения функции и ряда:
double x=1.0;
// Вычисление значений для ряда и функции:
    cout<<myexp(x)<<" vs. "<<exp(x)<<endl;
    cout<<mytan(x)<<" vs. "<<tan(x)<<endl;
    cout<<f(x/2)<<" vs. "<<1/(1-x/2)<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Код похож на рассмотренный ранее, но есть некоторые особенности. В первую очередь это касается операторного метода для квадратных скобок. Метод описан очень просто:

```
double &operator[](int k){
      return a[k];
   }
```

Результатом метода возвращается *ссылка* на элемент массива `a` с индексом, который определяется аргументом метода. Результатом возвращается именно ссылка на элемент (о чем свидетельствует инструкция `&` в прототипе метода), поскольку мы хотим не просто считывать значение элемента массива, но и иметь возможность присваивать значение элементу.

НА ЗАМЕТКУ

Массив `a` в классе `Taylor` теперь объявлен как закрытый. Поэтому вне тела класса прямое обращение к массиву `a` невозможно.

Операторный метод для вычисления значения ряда `operator()()` описан практически так же, как метод `value()` в предыдущем примере (см. листинг 6.4).

Единственное принципиальное отличие состоит в том, что вместо инструкции `a[k]` для обращения к элементу массива использована инструкция `(*this)[k]`, в которой принята во внимание возможность индексировать объекты класса `Taylor`. Подобная «замена» выполнена не только в методе, но и в конструкторе класса.

ПОДРОБНОСТИ

Необходимости отказываться от инструкций вида `a[k]` не было — можно было, как и раньше, использовать данную инструкцию.
Выражение `(*this)[k]` представляет собой применение операторного метода `operator[]()` к объекту, из которого вызывается операторный метод `operator()()`. Поскольку `this` — это указатель на объект, то `*this` — сам объект. Если дописать квадратные скобки с индексом, получаем индексированный объект. Обработка такого выражения выполняется с помощью операторного метода `operator[]()`. Круглые скобки используются с учетом приоритета операторов `*` и `[]` (чтобы сначала «применялась» звездочка `*`, а затем квадратные скобки `[]`).

В главной функции программы изменились команды, которыми выполняется обращение к элементам массива в объекте и способ вычисления значения ряда. Так, инструкция `myexp[k]` эквивалентна обращению к элементу с индексом `k` массива `a` в объекте `myexp`. Для вычисления значений рядов используются выражения `myexp(x)`, `mytan(x)` и `f(x/2)`. Здесь объекты вызываются как функции, и все благодаря тому, что в классе `Taylor` описан операторный метод `operator()()`.

НА ЗАМЕТКУ

Объекты, которые можно вызывать наподобие функции, называются **функторами**.

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.5)

```
2.71828 vs. 2.71828
1.5425 vs. 1.55741
1.99805 vs. 2
```

Видим, что результат выполнения программы точно такой же, как и в предыдущем случае.

Конструктор создания копии

Обычно при описании класса среди его конструкторов описывают, как минимум, конструктор без аргументов и конструктор создания копии. В последнем случае речь не обязательно идет о создании копии объекта, а скорее о процедуре создания объекта на основе уже существующего.

ТЕОРИЯ

При создании объекта на основе уже существующего объекта (того же класса), аргументом соответствующему конструктору передается исходный объект. Важное правило состоит в том, что аргумент конструктору создания копии передается **по ссылке**. По значению аргумент передавать нельзя. При попытке передать аргумент по значению для объекта-аргумента создается копия. При создании копии вызывается конструктор создания копии. Ему передается

копия для копии объекта-аргумента. Поэтому снова вызывается конструктор создания копии, и так до бесконечности. Получается такой своеобразный рекурсивный вызов конструктора создания копии, что неприемлемо.

Как иллюстрацию рассмотрим задачу, которая решалась в листинге 6.2, но теперь идеологический подход будет базироваться на использовании конструктора создания копии. Рассмотрим программный код, представленный в листинге 6.6.

Листинг 6.6. Конструктор создания копии

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Описание класса:
class MyClass{
public:
// Символьное поле:
char code;
// Поле – указатель на объект:
MyClass* next;
// Конструктор создания копии:
MyClass(MyClass &obj){
   // Значение поля next исходного объекта:
   obj.next=this;
   // Значение поля code создаваемого объекта:
   code=obj.code+1;
   }
   // Конструктор с символьным аргументом:
   MyClass(char s){
      code=s;
   }
   // Деструктор:
   ~MyClass(){
```

```
    if(next){ // Если не последний объект
       // Удаление следующего объекта:
       delete next;
    }
    cout<<"Объект с полем "<<code<<" удален\n";
}
// Метод для отображения значения символьного поля:
void show(){
    // Отображение значения поля:
    cout<<code<<" ";
    if(next){
       // Вызов метода для следующего объекта:
       next->show();
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Целочисленная переменная:
    int n=10;
    // Динамическое создание объекта:
    MyClass* pnt=new MyClass('A');
    // Указатель на объект:
    MyClass *p;
    // Начальное значение указателя:
    p=pnt;
    // Создание цепочки объектов:
    for(int k=1;k<=n;k++){
       // Создание очередного объекта:
       p=new MyClass(*p);
```

```
}
// Нулевая ссылка в поле next для последнего
// объекта в цепочке:
p->next=0;
// Вызов метода для первого объекта в цепочке:
pnt->show();
cout<<endl;
// Удаление первого объекта (с автоматическим
// удалением всей цепочки объектов):
delete pnt;
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 6.6)

```
A B C D E F G H I J K
Объект с полем K удален
Объект с полем J удален
Объект с полем I удален
Объект с полем H удален
Объект с полем G удален
Объект с полем F удален
Объект с полем E удален
Объект с полем D удален
Объект с полем C удален
Объект с полем B удален
Объект с полем A удален
```

Видим, что результат выполнения программы (по сравнению с примером из листинга 6.2) не изменился. Программный код, тем не менее, претерпел значительные изменения. Проанализируем их.

В классе MyClass теперь есть конструктор с символьным аргументом, определяющим значение поля code создаваемого объекта. Еще в классе MyClass описан конструктор создания копии. У этой версии конструктора один аргумент (обозначен как obj), и это объект класса MyClass. Аргумент передается по ссылке (инструкция & в описании аргумента конструктора). В теле конструктора командой obj.next=this значению поля next объекта, переданного конструктору, присваивается адрес создаваемого объекта. Мы исходим из того, что в цепочке объектов новый объект создается на основе предыдущего. Именно «предыдущий» объект передается аргументом конструктору при создании нового объекта. Поэтому логично, если в поле next объекта, на основе которого создается новый, записывается адрес вновь созданного объекта.

Для поля code создаваемого объекта значение присваивается командой code=obj.code+1. Таким образом, в новом объекте значение поля code на единицу «больше» значения поля code в объекте, переданном конструктору. Поскольку речь идет о символьных полях, то на самом деле получаем следующий символ в кодовой таблице символов.

В предыдущей версии программы для удаления цепочки объектов описывалась специальная функция. В данном случае аналогичная задача решается с помощью деструктора. Идея состоит в том, что при удалении объекта удаляется и вся цепочка объектов, с ним связанная. Реализуется все в деструкторе с помощью условного оператора, условием в котором проверяется значение выражения next. Значение данного выражения равно нулю и интерпретируется как false, если поле next текущего объекта равно нулю, и, значит, объект последний в цепочке. В этом случае команда в условном операторе игнорируется и просто выводится сообщение командой cout<<"Объект с полем "<<code<<"удален\n". Таким образом, если деструктор вызывается для последнего объекта в цепочке, то появляется сообщение, и объект будет удален (поскольку для него вызван деструктор, который выходит на сцену при удалении объекта). Если же значение поля next отлично от нуля, то в условном операторе командой delete next выполняется попытка удалить объект, адрес которого записан в поле next. Для этого объекта вызывается деструктор, и если объект в цепочке не последний, то вызванный деструктор вызовет деструктор для следующего объекта, и так вплоть до последнего объекта в цепочке. Когда удаляется последний объект, будет удален предпоследний объект, затем объект перед ним, и так до самого первого объекта, для которого вызывался самый первый деструктор.

В функции `main()` первый динамический объект создается командой `MyClass* pnt=new MyClass('A')`. Начальное значение указателя `p` совпадает со значением указателя `pnt`. Далее в операторе цикла при создании цепочки объектов выполняется одна и та же команда `p=new MyClass(*p)`, которой на основе текущего объекта (доступ к нему получаем с помощью инструкции `*p`) создается новый объект в цепочке, и указатель `p` «перебрасывается» на этот новый объект. Полю `next` последнего объекта в цепочке командой `p->next=0` присваиваем нулевое значение. Командой `pnt->show()` отображаются значения полей `code` всех объектов в цепочке, а удаление первого объекта с помощью команды `delete pnt` приводит к автоматическому удалению всех объектов в цепочке.

Наследование и закрытые поля базового класса

Следующий небольшой пример иллюстрирует ситуацию, когда в базовом классе есть закрытые члены, которые не наследуются в производном классе, но при этом используются в наследуемых открытых методах.

ТЕОРИЯ

Утверждается, что закрытые члены базового класса в производном классе не наследуются. Это так: в производном классе невозможно напрямую (то есть явно) обратиться к членам класса, которые в базовом классе являются закрытыми. При этом «технически» данные члены класса существуют и в производном классе. Поэтому, например, если производный класс наследует метод из базового класса, а этот метод в базовом классе, в свою очередь, обращается к закрытым членам, то ситуация останется корректной и в производном классе.

Рассмотрим программный код, представленный в листинге 6.7.

Листинг 6.7. Закрытые члены базового класса и наследование

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Базовый класс:
class Alpha{
```

```
private:
   // Закрытое поле:
   char symb;
public:
   // Конструктор класса:
   Alpha(char s){
      symb=s;
   }
   // Метод для отображения значения поля:
   void show(){
      cout<<"Класс Alpha: "<<symb<<endl;
   }
   // Метод для присваивания значения полю:
   void set(char s){
      symb=s;
   }
};
// Производный класс:
class Bravo: public Alpha{
public:
   // Открытое поле:
   int num;
   // Конструктор класса:
   Bravo(char s,int n):Alpha(s){
      num=n;
   }
   // В методе вызывается унаследованный метод:
   void showAll(){
      show();
      cout<<"Класс Bravo: "<<num<<endl;
   }
};
```

```
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Создание объекта производного класса:
   Bravo obj('A',100);
   // Отображение параметров объекта:
   obj.showAll();
   // Вызов унаследованного метода:
   obj.set('Z');
   // Отображение параметров объекта:
   obj.showAll();
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 6.7)

```
Класс Alpha: A
Класс Bravo: 100
Класс Alpha: Z
Класс Bravo: 100
```

Программа достаточно простая, но некоторые моменты все же прокомментировать стоит. Итак, в классе `Alpha` описано закрытое символьное поле `symb`. В классе есть конструктор с одним аргументом (значение для поля) и два открытых метода. Методом `show()` отображается значение поля `symb`, а метод `set()` предназначен для присваивания значения полю. Класс `Bravo` создается путем наследования класса `Alpha`. В классе `Bravo` наследуются методы `set()` и `show()`, но не наследуется поле `symb`. Также в классе `Bravo` описано открытое целочисленное поле `num`. У конструктора класса `Bravo` два аргумента: второй аргумент определяет значение поля `num`, а первый символьный аргумент передается конструктору базового класса. Здесь нужно учесть, что хотя поле `symb` в производном

классе `Bravo` не наследуется, память под поле выделяется и те методы (в том числе и конструктор) из базового класса, которые обращаются к данному полю и наследуются в производном классе, могут обращаться к соответствующей области памяти.

> **НА ЗАМЕТКУ**
>
> Проще говоря, «ненаследуемость» поля (или метода) означает, что в производном классе к нему нельзя обратиться по имени, но можно обратиться не напрямую.

В классе `Bravo` описан метод `showAll()`, в котором, кроме прочего, вызывается унаследованный из класса `Alpha` метод `show()`, а он, в свою очередь, обращается к полю `symb` (которое не наследуется).

В функции `main()` командой `Bravo obj('A',100)` создается объект класса `Bravo`. Хотя поля `symb` у объекта нет, при выполнении команды `obj.showAll()` для этого поля отображается значение `'A'`. Более того, после присваивания значения командой `obj.set('Z')` «несуществующему» полю проверка (с помощью команды `obj.showAll()`) показывает, что значение «несуществующего» поля изменилось. Но ничего удивительного здесь на самом деле нет. Как отмечалось выше, место в памяти под поле `symb` при создании объекта производного класса выделяется, просто мы к этой области памяти не имеем прямого доступа.

Виртуальные методы и наследование

Прежде чем приступить к обсуждению сути следующей проблемы, рассмотрим исключительно простой программный код, представленный в листинге 6.8.

Листинг 6.8. Невиртуальный метод

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Базовый класс:
class Alpha{
```

```
public:
   // Обычный (невиртуальный) метод:
   void show(){
      cout<<"Класс Alpha"<<endl;
   }
   // В методе вызывается метод:
   void showAll(){
      show();
   }
};
// Производный класс:
class Bravo: public Alpha{
public:
   // Переопределение метода:
   void show(){
      cout<<"Класс Bravo"<<endl;
   }
};
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Объект производного класса:
   Bravo obj;
   // Вызов методов:
   obj.show();
   obj.showAll();
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 6.8)

```
Класс Bravo
Класс Alpha
```

Ситуация исключительно простая: в классе `Alpha` описан метод `show()`, которым выводится сообщение в консольное окно. В этом же классе описан метод `showAll()`, в теле которого вызывается метод `show()`.

На основе класса `Alpha` путем наследования создается класс `Bravo`. Понятно, что в классе `Bravo` наследуются методы `show()` и `showAll()`. Тем не менее метод `show()` в классе `Bravo` переопределяется — меняется содержание сообщения, отображаемого в консольном окне.

В главной функции программы создается объект `obj` производного класса `Bravo`, и из него последовательно вызываются методы `show()` и `showAll()`. Результат выполнения команды `obj.show()` ожидаем — в консоль выводится сообщение `Класс Bravo`, в соответствии с тем, как переопределен метод `show()` в классе `Bravo`. Но вот при выполнении команды `obj.showAll()` появляется сообщение `Класс Alpha`. Причина очевидна: метод `showAll()` вызывает ту версию метода `show()`, что описана в базовом классе, а не переопределенную версию из производного класса. Чтобы изменить ситуацию, метод `show()` в базовом классе нужно сделать *виртуальным*.

Описание виртуального метода начинается с ключевого слова `virtual`. Например, чтобы описать в базовом классе `Alpha` метод `show()` как виртуальный, используем такой код:

```
virtual void show(){
   cout<<"Класс Alpha"<<endl;
}
```

С учетом изменений, программа будет выглядеть так (практически все комментарии удалены, а важные места выделены жирным шрифтом):

Листинг 6.9. Виртуальный метод

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

```
class Alpha{
public:
   // Виртуальный метод:
   virtual void show(){
      cout<<"Класс Alpha"<<endl;
   }
   void showAll(){
      show();
   }
};
class Bravo: public Alpha{
public:
   void show(){
      cout<<"Класс Bravo"<<endl;
   }
};
int main(){
   system("chcp 1251>nul");
   Bravo obj;
   obj.show();
   obj.showAll();
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы теперь следующий:

Результат выполнения программы (из листинга 6.9)

```
Класс Bravo
Класс Bravo
```

Таким образом, благодаря тому, что метод `show()` теперь виртуальный, при вызове метода `showAll()` из объекта производного класса вызывается переопределенная в классе `Bravo` версия метода `show()`.

НА ЗАМЕТКУ

Переопределяемые методы обычно объявляются виртуальными.

Множественное наследование

В C++ производный класс может создаваться на основе сразу нескольких базовых классов. Об этом следующий пример, представленный в листинге 6.10.

Листинг 6.10. Множественное наследование

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Первый базовый класс:
class Alpha{
public:
// Поле:
int alpha;
// Конструктор:
Alpha(int a){
   alpha=a;
}
// Метод:
void show(){
   cout<<"Класс Alpha: "<<alpha<<endl;
}
};
// Второй базовый класс:
class Bravo{
public:
   // Поле:
   int bravo;
   // Конструктор:
```

```
    Bravo(int b){
        bravo=b;
    }
    // Метод:
    void show(){
        cout<<"Класс Bravo: "<<bravo<<endl;
    }
};
// Производный класс:
class Charlie:public Alpha,public Bravo{
public:
    // Поле:
    int charlie;
    // Конструктор:
    Charlie(int a,int b,int c):Alpha(a),Bravo(b){
        charlie=c;
    }
    // Переопределение метода:
    void show(){
        // Вызов версии метода из класса Alpha:
        Alpha::show();
        // Вызов версии метода из класса Bravo:
        Bravo::show();
        // Отображение значения поля:
        cout<<"Класс Charlie: "<<charlie<<endl;
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Создание объекта производного класса:
```

```
Charlie obj(10,20,30);
// Вызов метода:
obj.show();
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 6.10)

```
Класс Alpha: 10
Класс Bravo: 20
Класс Charlie: 30
```

Теперь проанализируем программный код. В программе описываются классы `Alpha` и `Bravo`. Они становятся базовыми для класса `Charlie`, который создается на их основе. В частности, в классе `Alpha` есть целочисленное поле `alpha`, конструктор с одним аргументом и метод `show()`, которым при вызове отображается значение поля `alpha` объекта, из которого вызывается метод. Аналогичная структура и у класса `Bravo`, с поправкой на то, что его целочисленное поле называется `bravo`. У класса `Bravo` также имеется метод `show()`, выполняющий аналогичную миссию (как и метод `show()` в классе `Alpha`).

Класс `Charlie` создается путем открытого наследования на основе классов `Alpha` и `Bravo`. Поэтому в описании класса `Charlie` после имени класса через двоеточие указываются названия базовых классов и тип наследования (ключевое слово `public`) для каждого класса. Непосредственно в классе `Charlie` описано целочисленное поле `charlie` и естественно, наследуются поля `alpha` и `bravo`.

У конструктора класса `Charlie` три целочисленных аргумента: два передаются в конструкторы базовых классов, а еще один аргумент задает значение поля `charlie`. В описании производного класса после имени конструктора через двоеточие указываются инструкции вызова конструкторов базовых классов (с передачей им аргументов).

Метод `show()` в классе `Charlie` переопределяется. В теле метода `show()` в классе `Charlie` командами `Alpha::show()` и `Bravo::show()`

последовательно вызываются версии метода из класса `Alpha` и класса `Bravo`. В данном случае для идентификации версии метода используется название класса, в котором определена соответствующая версия метода. Между именем класса и именем метода размещается оператор расширения контекста `::` (двойное двоеточие). После вызова версий метода `show()` из классов `Alpha` и `Bravo` (в результате чего в консоли отображаются значения полей `alpha` и `bravo`) командой `cout<<"Класс Charlie: "<<charlie<<endl` отображается значение поля `charlie`. В главной функции программы создается объект производного класса и для проверки значений его полей вызывается метод `show()`.

Доступ к объектам через переменную базового класса

Есть важное свойство, которое «связывает» объекты базовых классов и объекты производных классов: переменной для объекта базового класса значением можно присвоить объект производного класса.

Эта важная и фундаментальная возможность проиллюстрирована в программном коде, представленном в листинге 6.11.

Листинг 6.11. Объекты базовых и производных классов

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Первый базовый класс:
class Alpha{
public:
   // Поле:
   char codeA;
   // Конструктор:
   Alpha(char a){
      codeA=a;
   }
   // Виртуальный метод:
   virtual void show(){
```

```
        cout<<"Метод из класса Alpha: "<<codeA<<endl;
    }
};
// Второй базовый класс:
class Bravo{
public:
// Поле:
char codeB;
// Конструктор:
Bravo(char b){
    codeB=b;
}
// Виртуальный метод:
virtual void show(){
    cout<<"Метод из класса Bravo: "<<codeB<<endl;
}
};
// Производный класс:
class Charlie:public Alpha,public Bravo{
public:
    // Конструктор с пустым телом:
    Charlie(char a,char b):Alpha(a),Bravo(b){}
    // Переопределение метода:
    void show(){
        cout<<"Метод из класса Charlie: ";
        cout<<codeA<<codeB<<endl;
    }
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
```

```
    cout<<"Используем переменные\n";
    // Создание объектов:
    Alpha objA('A');
    objA. show();
    Bravo objB('B');
    objB. show();
    Charlie objC('C','D');
    objC. show();
    // Присваивание объектным переменным базовых классов
    // значением объекта производного класса:
    objA=objC;
    objB=objC;
    objA.show();
    objB.show();
    cout<<"Используем указатели\n";
    // Объявление указателей:
    Alpha* pntA=&objC;
    Bravo* pntB=&objC;
    Charlie* pntC=&objC;
    // Метод вызывается через указатели:
    pntA->show();
    pntB->show();
    pntC->show();
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 6.11)

```
Используем переменные
Метод из класса Alpha: A
```

```
Метод из класса Bravo: B
Метод из класса Charlie: CD
Метод из класса Alpha: C
Метод из класса Bravo: D
Используем указатели
Метод из класса Charlie: CD
Метод из класса Charlie: CD
Метод из класса Charlie: CD
```

Программный код заслуживает некоторых пояснений. Классы `Alpha` и `Bravo` являются базовыми для класса `Charlie`. В классе `Alpha` есть символьное поле `codeA`, конструктор с одним аргументом и метод `show()` который описан как виртуальный (с ключевым словом `virtual`). Аналогичный метод есть в классе `Bravo`. Еще там есть символьное поле `codeB` и конструктор с одним аргументом. Класс `Charlie` наследует классы `Alpha` и `Bravo`. У конструктора класса `Charlie` два аргумента, которые передаются конструкторам базовых классов. Поскольку другие действия в конструкторе не выполняются, то тело конструктора пустое (в фигурных скобках нет команд). Метод `show()` в классе `Charlie` переопределяется так, что последовательно отображаются значения двух полей: `codeA` и `codeB`.

Все самое интересное происходит в функции `main()`. Там сначала создается три объекта `objA`, `objB` и `objC` соответственно классов `Alpha`, `Bravo` и `Charlie`. Для каждого из этих объектов вызывается метод `show()`. На данном этапе все происходит «штатно". Затем выполняются команды `objA=objC` и `objB=objC`, которыми объектным переменным `objA` и `objB` базовых классов `Alpha` и `Bravo` значением присваивается объект `objC` производного класса `Charlie`. Поскольку класс `Charlie` является производным от классов `Alpha` и `Bravo`, такие присваивания законны. Однако есть важное ограничение: через переменную базового класса можно получить доступ к только тем полям и методам, которые объявлены в данном базовом классе. Поэтому при вызове через переменную `objA` получаем доступ только к полю `codeA` и версии метода `show()`, объявленной в классе `Alpha`. Через переменную `objB` получаем доступ к полю `codeB` и версии метода `show()` из класса `Bravo`.

Иначе дело обстоит, если мы используем указатели. В программе объявлены три указателя: указатель `pntA` на объект класса `Alpha`, указатель

pntB на объект класса Bravo и указатель pntC на объект класса Charlie. Каждому из указателей значением присваивается адрес объекта objC производного класса Charlie (инструкция &objC). Такая операция тоже законная, и все благодаря тому, что классы Alpha и Bravo являются базовыми для класса Charlie. Пикантность ситуации в том, что если мы вызываем метод show() через указатели pntA и pntB, то вызывается версия метода, описанная в классе Charlie.

ПОДРОБНОСТИ

Как и в случае с объектными переменными, через указатели на объекты базового класса есть доступ только к тем полям и методам объекта производного класса, которые описаны в базовом классе. Проще говоря, через указатель pntA получаем доступ только к полю codeA, а через указатель pntB есть доступ только к полю codeB. Но если через такой указатель вызывается переопределенный метод, а исходная версия метода описана как виртуальная, то будет вызвана именно переопределенная версия метода.

Задачи для самостоятельного решения

1. Напишите программу, в которой создается массив объектов. У каждого объекта есть закрытое числовое поле и метод, возвращающий значение данного поля. При создании объекта полю присваивается случайное целое число.

2. Напишите программу с классом, у которого есть поле, представляющее собой числовой массив. Предусмотреть разные способы создания объектов класса (в том числе на основе уже существующего объекта). Описать операторные функции и методы для выполнения различных операций с объектами (например, сложение объектов, индексирование объектов, умножение и деление на число). Например, при сложении объектов результатом может быт объект с массивом, элементы которого определяются как среднее арифметическое соответствующих элементов в массивах складываемых объектов. При вызове объекта результатом может возвращаться среднее арифметическое значение элементов массива из данного объекта, и так далее.

3. Напишите программу, в которой создается «кольцо» из объектов: «цепочка» объектов, в которой один объект ссылается на другой, а самый последний объект ссылается на первый. Каждый объект имеет, кроме прочего, закрытое числовое поле со случайным значением,

и закрытое символьное поле тоже со случайным значением. Предложить функции и методы для работы с таким «кольцом»: например, метод для отображения значений полей объектов, которые входят в «кольцо», или функцию для удаления объектов в «кольце». Рекомендуется решить задачу разными способами.

4. Напишите программу, в которой создается «бинарное дерево» объектов: каждый объект ссылается на два других объекта. Предложить алгоритм создания такого «дерева», методы для идентификации объектов в нем и способы удаления «дерева».

5. Напишите программу с классом, у которого полем является массив указателей на объекты того же класса. Предложите методы и функции для работы с такими объектами.

Рекомендации для самостоятельной работы

- Попробуйте решить задачи, рассмотренные в главе, используя вместо статических массивов динамические массивы и динамический способ выделения памяти (разумеется, там, где это уместно).
- Попробуйте решить задачи, рассмотренные в предыдущих главах (и предлагавшиеся там для самостоятельного решения), используя вместо массивов индексируемые объекты.
- В примерах с использованием функций, которые рассматривались в предыдущих главах, попробуйте заменить функции на функторы.
- Известно, что функция не может результатом возвращать статический массив. Вместе с тем результатом функции может быть объект, допускающий индексирование. В таком случае создается иллюзия, что функция возвращает массив. Также индексируемый объект может быть результатом, возвращаемым функтором. Подумайте и предложите способы решения некоторых из рассмотренных ранее задач с использованием данной технологии.

Глава 7

ОБОБЩЕННЫЕ ФУНКЦИИ И КЛАССЫ

— Скажите, доктор Ватсон, Вы понимаете всю важность моего открытия?
— Да, как эксперимент это интересно. Но какое практическое применение?
— Господи, именно практическое!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Список тем, обсуждаемых в этой главе, достаточно краток. Однако сами по себе темы очень объемные и интересные. Итак, далее мы обсудим и рассмотрим:

- обобщенные функции;
- обобщенные классы,

ну и еще некоторые сопутствующие подходы и механизмы.

Обобщенные функции

Идея *обобщенных функций* состоит в том, что в них тип данных выступает параметром.

ТЕОРИЯ

Описание **обобщенной функции** начинается с ключевого слова `template`, после которого в угловых скобках указывается ключевое слово `class` и формальное обозначение для типа данных (параметр типа или обобщенный параметр). Во всем остальном обобщенная функция описывается как обычная (не обобщенная), с поправкой на то, что в описании функции в качестве идентификатора типа можно использовать параметр.

В листинге 7.1 представлена программа, содержащая обобщенную функцию `show()`. Этой функцией в консольном окне отображается значение аргумента, который передается функции при вызове.

Листинг 7.1. Обобщенная функция с одним параметром типа

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Обобщенная функция:
template<class X> void show(X arg){
   cout<<arg<<endl;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Вызов обобщенной функции с символьным аргументом:
   show('A');
   // Вызов обобщенной функции с числовым аргументом:
   show(123);
   // Вызов обобщенной функции с текстовым аргументом:
   show("Текст");
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения функции такой:

Результат выполнения программы (из листинга 7.1)

```
A
123
Текст
```

Самое интересное здесь, безусловно — это описание обобщенной функции. Вот так выглядит интересующий нас код:

```
template<class X> void show(X arg){
   cout<<arg<<endl;
}
```

Описание обобщенной функции начинается с ключевого слова `template`. Инструкция `<class X>`, следующая далее, означает, что идентификатор `X` в описании функции обозначает некоторый тип данных. Какой именно — становится ясно при вызове функции. Функция, как видим, не возвращает результат (ключевое слово `void` перед именем функции), и у нее один аргумент (обозначен как `arg`). Тип аргумента определяется идентификатором `X`. В теле функции всего одна команда `cout<<arg<<endl`, в соответствии с которой значение аргумента отображается в консольном окне (и выполняется переход к новой строке).

В главной функции программы функция `show()` вызывается трижды: речь о командах `show('A')`, `show(123)` и `show("Текст")`.

Каждый раз при вызове функции по типу переданного ей аргумента определяется значение для обобщенного параметра `X` и код функции выполняется при соответствующем значении данного параметра. Например, в команде `show('A')` функция `show()` получает аргументом значение типа `char`. Поэтому код функции выполняется так, как если бы параметр `X` был идентификатором `char`. При выполнении команды `show(123)` параметр `X` «становится» идентификатором `int`, а при выполнении команды `show("Текст")` параметр `X` интерпретируется как указатель на символьное значение (условно говоря, тип `char*`).

В следующей программе использование обобщенных функций носит более утонченный характер. В частности, в ней описываются две обобщенные функции: функция `show()` предназначена для отображения содержимого массива, а функция `sort()` используется для сортировки массива методом пузырька. Причем в теле функции `sort()` вызывается функция `show()`.

НА ЗАМЕТКУ

При сортировке массива (в порядке возрастания значений элементов) методом пузырька сравниваются последовательно значения соседних элементов массива, и если значение элемента слева больше

значения элемента справа, то они обмениваются значениями. Количество полных «проходов» массива с «перебором» элементов на единицу меньше количества элементов в массиве. За каждый следующий проход количество проверяемых элементов уменьшается на единицу.

Рассмотрим программный код, представленный в листинге 7.2.

Листинг 7.2. Обобщенная функция для сортировки массива

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Обобщенная функция для отображения массива:
template<class T> void show(T* m,int n){
   for(int i=0;i<n;i++){
      cout<<m[i]<<" ";
   }
   cout<<endl;
}
// Обобщенная функция для сортировки массива:
template<class X> void sort(X* m,int n){
   // Отображение несортированного массива:
   show(m,n);
   // Локальная переменная обобщенного типа:
   X s;
   // Вложенные операторы цикла:
   for(int i=1;i<=n-1;i++){
      for(int j=0;j<n-i;j++){
         if(m[j]>m[j+1]){
            // Обмен значениями соседних элементов:
            s=m[j+1];
            m[j+1]=m[j];
            m[j]=s;
         }
```

```
        }
    }
    // Отображение отсортированного массива:
    show(m,n);
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Числовой массив:
    int A[5]={3,2,8,1,0};
    // Символьный массив:
    char B[7]={'Z','B','Y','A','D','C','X'};
    // Сортировка массивов:
    sort(A,5);
    sort(B,7);
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 7.2)

```
3 2 8 1 0
0 1 2 3 8
Z B Y A D C X
A B C D X Y Z
```

В описании обобщенной функции `show()` параметр типа обозначен как `T`. Аргументов у функции два: первый аргумент `m` описан с идентификатором типа `T*`. Это означает, что аргумент `m` является указателем на значение того типа, который «скрывается» за идентификатором `T`. Несложно догадаться, что под `m` подразумевается имя массива, элементы которого относятся к типу `T`. Второй аргумент `n` функции `show()` цело-

численный. Он определяет размер массива, обрабатываемого функцией. В теле функции запускается оператор цикла, в котором аргумент `m` обрабатывается как массив: в консольное окно выводятся значения его элементов.

Обобщенная функция `sort()` для сортировки массива описана с параметром типа `X`, она не возвращает результат, и у нее, как и у функции `show()`, два аргумента. Первый (обозначен как `m`) формально является указателем на значение типа `X`, но мы предполагаем на самом деле, что это имя массива, элементы которого относятся к типу `X`. Второй целочисленный аргумент `n` является размером массива. В теле функции сначала командой `show(m,n)` вызывается обобщенная функция `show()`, в результате чего в консольном окне отображается содержимое неотсортированного еще массива. Затем объявляется локальная переменная `s`, которая относится к обобщенному типу `X`. Переменная нам понадобится, когда мы во вложенных операторах цикла при переборе и сравнении элементов будем производить «обмен» значениями. После того, как массив отсортирован, командой `show(m,n)` отображаем новый порядок элементов в массиве (точнее, новый порядок значений элементов в массиве).

В функции `main()` командами `int A[5]={3,2,8,1,0}` и `char B[7]={'Z','B','Y','A','D','C','X'}` создаются целочисленный и символьный массивы. Для их сортировки используются соответственно команды `sort(A,5)` и `sort(B,7)`.

> **НА ЗАМЕТКУ**
>
> При сравнении значений элементов символьных массивов сравниваются коды соответствующих символов.

Обобщенная функция с несколькими параметрами

У обобщенной функции может быть несколько параметров типа. Как иллюстрацию рассмотрим программу, представленную в листинге 7.3.

Листинг 7.3. Обобщенная функция с несколькими параметрами

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

```
// Обобщенная функция с двумя параметрами:
template<class X,class R> R apply(R (*fun)(X),X arg){
   return fun(arg);
}
// Обычная функция (аргумент типа double
// и результат типа double):
double f(double x){
   return x*(1-x);
}
// Обычная функция (аргумент типа int
// и результат типа int):
int factorial(int n){
   if(n==0){
      return 1;
   }
   else{
      return n*factorial(n-1);
   }
}
// Обычная функция (аргумент типа int
// и результат типа char):
char symb(int n){
   return 'A'+n;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Вызов обобщенной функции:
   cout<<apply(f,0.5)<<endl;
   cout<<apply(factorial,5)<<endl;
   cout<<apply(symb,3)<<endl;
```

```
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В программе описана обобщенная функция `apply()`, предназначенная для применения некоторой (обычной) функции, переданной первым аргументом функции `apply()`, к аргументу, который передан вторым аргументом функции `apply()`. Описание обобщенной функции выглядит так:

```
template<class X,class R> R apply(R (*fun)(X),X arg){
    return fun(arg);
}
```

Функция объявлена с двумя параметрами типа `X` и `R`. Результатом функции возвращается значение типа `R`. Аргументов у нее два: второй аргумент `arg` относится к типу `X`, а первый аргумент функции `apply()` является указателем на функцию, у которой один аргумент типа `X` и которая возвращает результатом значение типа `R`. Инструкция, описывающая первый аргумент, выглядит как `R (*fun)(X)`.

НА ЗАМЕТКУ

Напомним, что указатель на функцию описывается следующим образом:

- указывается идентификатор типа результата функции;
- в круглых скобках указывается название переменной-указателя, перед названием — звездочка `*`;
- в круглых скобках через запятую перечисляются идентификаторы типа для аргументов функции.

Результат обобщенной функции `apply()` возвращается командой `return fun(arg)`. Фактически, здесь результатом обобщенной функции возвращается значение, вычисляемое при применении функции, имя которой передано первым аргументом обобщенной функции `apply()`, к значению, переданному вторым аргументом функции `apply()`.

Для проверки работы функции `apply()` в программе объявлено несколько обычных функций (главное условие — чтобы они возвращали результат и чтобы у них был один аргумент). В частности:

- у функции `f()` аргумент типа `double` и результат типа `double`;
- функция для вычисления факториала числа `factorial()` возвращает значение типа `int` и аргумент функции также относится к типу `int`;
- функция `symb()` возвращает результатом символ (тип `char`), а аргумент функции имеет тип `int`.

В главной функции программы инструкциями `apply(f,0.5)`, `apply(factorial,5)` и `apply(symb,3)` вызывается обобщенная функция `apply()`, аргументом которой передается имя обычной (необобщенной функции) и значение, используемое при ее вызове. Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 7.3)

```
0.25
120
D
```

Таким образом, результатом функция `apply()` возвращает то значение, которое является результатом переданной аргументом функции.

Перегрузка обобщенной функции

Истина проста — можно создать несколько версий обобщенной функции. Пример такой ситуации представлен в листинге 7.4. Там описано две версии обобщенной функции `show()`: с одним аргументом и с двумя аргументами. Сама функция очень простая: при ее вызове отображаются значения аргументов. Благодаря тому, что типы аргументов идентифицируются через обобщенные параметры, в программе функцию `show()` можно вызывать с одним или двумя аргументами произвольного типа. Однако при этом должна иметь смысл команда выведения значения аргумента в консольное окно. Рассмотрим представленный ниже программный код:

Листинг 7.4. Перегрузка обобщенной функции

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

```
// Версия обобщенной функции с одним аргументом:
template<class X> void show(X x){
   cout<<"Функция с одним аргументом\n";
   cout<<"Значение аргумента: "<<x<<endl;
}
// Версия обобщенной функции с двумя аргументами:
template<class X,class Y> void show(X x,Y y){
   cout<<"Функция с двумя аргументами\n";
   cout<<"Первый аргумент: "<<x<<endl;
   cout<<"Второй аргумент: "<<y<<endl;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Вызов обобщенной функции с одним аргументом:
   show('A');
   show(123);
   show("Текст");
   // Вызов обобщенной функции с двумя аргументами:
   show(321,"Текст");
   show('B',456);
   show('C','D');
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

При выполнении программы получаем такой результат:

Результат выполнения программы (из листинга 7.4)

```
Функция с одним аргументом
Значение аргумента: A
```

```
Функция с одним аргументом
Значение аргумента: 123
Функция с одним аргументом
Значение аргумента: Текст
Функция с двумя аргументами
Первый аргумент: 321
Второй аргумент: Текст
Функция с двумя аргументами
Первый аргумент: B
Второй аргумент: 456
Функция с двумя аргументами
Первый аргумент: C
Второй аргумент: D
```

Явная специализация обобщенной функции

При работе с обобщенными функциями можно создавать специальные версии функции для определенных значений обобщенных параметров.

В подобном случае говорят о *явной специализации* обобщенной функции. Пример такой ситуации представлен в листинге 7.5.

Листинг 7.5. Явная специализация обобщенной функции

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Описание класса:
class MyClass{
public:
   int number;
   MyClass(int n){
```

```
        number=n;
    }
    void show(){
        cout<<"Объект класса MyClass: "<<number<<endl;
    }
};
// Обобщенная функция:
template<class X> void show(X arg){
    cout<<"Значение аргумента функции: "<<arg<<endl;
}
// Явная специализация обобщенной функции:
template<> void show<int>(int arg){
    cout<<"Целочисленный аргумент: "<<arg<<endl;
}
template<> void show<MyClass>(MyClass obj){
    obj.show();
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    MyClass obj(300);
    // Вызов обобщенной функции:
    show('A');
    show(100.0);
    show(200);
    show(obj);
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 7.5)

```
Значение аргумента функции: A
Значение аргумента функции: 100
Целочисленный аргумент: 200
Объект класса MyClass: 300
```

В программе описана обобщенная функция `show()` с одним аргументом, тип которого задан через обобщенный параметр. Функцией при вызове отображается значение аргумента.

Вместе с тем в программе также в явном виде описаны две специализации функции для случая, когда аргумент является целым числом (значение типа `int`) и когда аргумент является объектом класса `MyClass`. Последний описан в программе таким образом, что у объектов класса `MyClass` есть поле `number` и метод `show()`, при вызове которого отображается значения поля `number`. Явная специализация функции `show()` для аргумента, являющегося объектом класса `MyClass`, предусматривает, что из объекта, переданного аргументом функции `show()` вызывается одноименный метод.

Что касается технического вопроса по явной специализации обобщенной функции, то начинается описание явной специализации с ключевого слова `template`, после которого следуют пустые угловые скобки <>. Далее идет обычное описание функции, за тем исключением, что после имени функции в угловых скобках указывается идентификатор типа, для которого выполняется явная специализация функции.

Обобщенные классы

Аналогично обобщенным функциям можно создавать классы, в которых идентификаторы типов данных выступают параметрами.

ТЕОРИЯ

Описание **обобщенного класса** начинается так же, как и описание обобщенной функции: с ключевого слова `template`, после которого в угловых скобках перечисляются параметры для обобщенных типов (каждый параметр указывается с ключевым словом `class`). Затем идет описание класса, в котором можно использовать параметры обобщенных типов.

При создании объекта на основе обобщенного класса после имени класса в угловых скобках указываются идентификаторы типа, которые следует использовать вместо обобщенных параметров.

В листинге 7.6 приведен пример описания и использования обобщенного класса.

Листинг 7.6. Обобщенный класс

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Обобщенный класс:
template<class A,class B> class MyClass{
public:
   // Поля класса:
   A first;
   B second;
   // Конструктор класса:
   MyClass(A f,B s){
      first=f;
      second=s;
   }
   // Метод для отображения значения полей:
   void show(){
      cout<<"Первое поле: "<<first<<endl;
      cout<<"Второе поле: "<<second<<endl;
   }
};
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
```

```
    // Создание объектов на основе обобщенного класса:
    MyClass<int,char> objA(100,'A');
    MyClass<string,double> objB("текст",10.5);
    MyClass<char*,string> objC("первый","второй");
    MyClass<int,int> objD(1,2);
    // Проверка значений полей объектов:
    objA.show();
    objB.show();
    objC.show();
    objD.show();
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 7.6)

```
Первое поле: 100
Второе поле: A
Первое поле: текст
Второе поле: 10.5
Первое поле: первый
Второе поле: второй
Первое поле: 1
Второе поле: 2
```

Описание обобщенного класса, как и следовало ожидать, начинается с ключевого слова `template`. Инструкция `<class A,class B>` означает, что в классе используются два обобщенных параметра `A` и `B` для идентификации типов.

Далее идет непосредственно описание класса `MyClass`. В частности, в классе объявлено поле `first` типа `A`, и поле `second` типа `B`. Что скрывается за параметрами `A` и `B`, станет понятно после создания объекта класса `MyClass`.

В классе описан конструктор с двумя аргументами (типа `A` первый и типа `B` второй), которые определяют значения полей `first` и `second`. Также в классе есть метод `show()`, предназначенный для отображения значений полей.

В главной функции программы создается ряд объектов. В частности, для этого используются команды `MyClass<int,char> objA(100,'A')`, `MyClass<string,double> objB("текст",10.5)`, `MyClass<char*,string> objC("первый","второй")` и `MyClass<int,int> objD(1,2)`.

В каждой команде после имени класса `MyClass` в угловых скобках указаны идентификаторы типов, которые следует использовать вместо обобщенных параметров при создании объекта класса. Скажем, инструкция `<int,char>` в команде создания объекта `objA` означает, что в качестве параметра `A` нужно использовать `int`, а в качестве параметра `B` — `char`.

В команде `MyClass<string,double> objB("текст",10.5)` для первого параметра указано значение `string` и, соответственно, первым аргументом конструктору передается текстовый литерал `"текст"`. Как неоднократно отмечалось, текстовые литералы реализуются в виде символьных массивов, поэтому на самом деле речь идет о формальном типе `char*`. Но здесь имеет место автоматическое приведение типов и литерал без нашего участия приводится к типу `string`. Вместе с тем наравне с типом `string` допустимо использовать и формальный тип `char*` для обозначения «текстового» типа.

Примером служит команда `MyClass<char*,string> objC("первый", "второй")`.

Наконец, оба обобщенных параметра могут принимать одинаковые значения — как, например, имеет место при создании объекта командой `MyClass<int,int> objD(1,2)`. Здесь оба параметра `A` и `B` принимают значение `int`.

НА ЗАМЕТКУ

Для использования в программе класса `string` подключается заголовок `<string>`.

Для проверки значения полей созданных объектов из каждого объекта вызывается метод `show()`.

Явная специализация обобщенного класса

Как и в случае с обобщенными функциями, при описании обобщенного класса можно задавать его *явную специализацию* для некоторых частных случаев. Рассмотрим пример, представленный в листинге 7.7.

Листинг 7.7. Явная специализация обобщенного класса

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Обобщенный класс:
template<class T> class MyClass{
private:
   // Закрытое поле:
   T value;
public:
   // Конструктор:
   MyClass(T v){
      value=v;
   }
   // Метод для отображения значения поля:
   void show(){
      cout<<"Значение поля: "<<value<<endl;
   }
};
// Явная специализация класса:
template<> class MyClass<string>{
private:
   // Закрытое поле – массив:
   char value[100];
public:
```

```
// Конструктор:
MyClass(char* str){
   int k;
   // Первый символ в поле - массиве:
   value[0]='|';
   // Заполнение поля - массива:
   for(k=0;str[k];k++){
      value[2*k+1]=str[k];
      value[2*k+2]='|';
   }
   // Добавление нуль-символа в конец массива:
   value[2*k+1]='\0';
}
void show(){
   cout<<value<<endl;
}
};
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   MyClass<int> objA(100);
   // Создание объектов на основе обобщенного класса:
   MyClass<char> objB('A');
   MyClass<char*> objC("текст");
   MyClass<string> objD("текст");
   // Вызов метода из объекта, созданного на
   // основе обобщенного класса:
   objA.show();
   objB.show();
   objC.show();
   objD.show();
```

```
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы будет таким, как показано ниже:

Результат выполнения программы (из листинга 7.7)

```
Значение поля: 100
Значение поля: A
Значение поля: текст
|т|е|к|с|т|
```

В программе описывается обобщенный класс `MyClass`, в котором через `T` обозначен обобщенный тип. У класса есть закрытое поле `value`, относящееся к обобщенному типу `T`.

Конструктор класса получает один аргумент, который и задает значение поля `value`. Для получения информации о значении поля `value` предназначен метод `show()`.

Явная специализация класса `MyClass` описана для случая, когда в качестве обобщенного параметра используется тип `string`.

НА ЗАМЕТКУ

Поскольку в программе используется класс `string`, подключаем заголовок `<string>`. Что касается явной специализации для обобщенного класса, то после ключевого слова `template` указываются пустые угловые скобки `<>`, а после имени класса в угловых скобках указывается идентификатор типа (в данном случае `string`), для которого выполняется специализация класса.

Специальная версия класса `MyClass` содержит, как и «общая» версия класса, закрытое поле с названием `value`, которое теперь является символьным массивом (из `100` элементов). Он заполняется в конструкторе. Аргументом конструктору (обозначен как `str`) формально передается указатель на значение типа `char`, что позволяет передать аргументом конструктору текстовый литерал. Принцип заполнения массива `value`

такой: в него копируются символы из текстового литерала, переданного аргументом конструктору, но между символами добавляется вертикальная черта. Такая же черта добавляется в начало массива `value`. Поскольку вертикальная черта — это отдельный символ, то заполнение массива `value` выглядит нетривиально.

В самом начале командой `value[0]='|'` первым символом в массив `value` записывается вертикальная черта. Затем запускается оператор цикла, в котором индексная переменная `k` пробегает значения от `0` и до конца текстового литерала.

ПОДРОБНОСТИ

Условие в условном операторе выглядит как `str[k]`. Оно будет интерпретироваться как `false`, только если элемент с индексом `k` является нуль-символом окончания строки.

Индексная переменная `k` объявлена до запуска оператора цикла, поскольку значение, которое она будет иметь по завершении оператора цикла, нами используется при записи нуль-символа в массив `value`.

При фиксированном значении индексной переменной командой `value[2*k+1]=str[k]` в массив `value` записывается очередная буква из аргумента `str` конструктора, а после нее добавляется вертикальная черта (команда `value[2*k+2]='|'`).

ПОДРОБНОСТИ

Индекс `k` определяет букву в текстовом литерале `str` (который на самом деле является символьным массивом). Массив `value` заполняется по два элемента. На позициях с четными индексами в массиве `value` размещаются символы вертикальной черты. На нечетных позициях в массиве `value` находятся буквы из массива `str`. Формула для нечетных индексов в массиве `value` дается выражением `2*k+1`, а значения для четных индексов вычисляются как `2*k+2`. Здесь мы учли, что при нулевом значении индекса `k` (первая буква в массиве `str`) в массиве `value` следует заполнять элемент с индексом `1`, а символ вертикальной черты присваивается элементу с индексом `2` в массиве `value`.

После завершения оператора цикла массив `value` заполнен буквами и вертикальными черточками, но нужно еще добавить нуль-символ.

Поскольку в операторе цикла переменная k уже получила очередное значение, то добавление нуль-символа выполняем командой value[2*k+1]='\0'.

ПОДРОБНОСТИ

После завершения оператора цикла индексная переменная k получает значение элемента в массиве str, который соответствует нуль-символу. В массиве value этой позиции соответствует элемент с индексом 2*k+1.

Метод show() в данной версии класса MyClass определен так, что отображается содержимое массива value (без дополнительного поясняющего текста.

НА ЗАМЕТКУ

Таким образом, в явной специализации класса MyClass для типа string этот самый тип string фактически не используется.

В главной функции программы на основе обобщенного класса MyClass создается несколько объектов. Особого внимания стоят объекты, которые создаются с идентификаторами типов char* и string для обобщенных параметров.

Аргументом конструктору в обоих случаях передается один и тот же текст, но результат получаем разный, поскольку в одном случае (формальный тип char*) используется «общая» версия класса MyClass, а во втором — (тип string) используется явная спецификация данного класса.

НА ЗАМЕТКУ

Когда используется идентификатор типа char*, то в конструкторе полю value в качестве значения присваивается адрес того текстового литерала, который передан аргументом конструктору. Если передать аргументом конструктору имя символьного массива, в поле value запишется адрес этого массива. Если создать несколько объектов на основе класса MyClass с идентификатором типа char* и каждый раз передавать аргументом конструктору один и тот же символьный массив (его имя), то поля value всех созданных объектов будут ссылаться на один и тот же массив.

Значения параметров по умолчанию

При описании обобщенного класса для параметров типа можно задать значения по умолчанию. Значения по умолчанию для обобщенных параметров указываются через знак равенства после названия параметра и представляют собой идентификаторы типов.

В листинге 7.8 по этому поводу представлена программа.

Листинг 7.8. Значение параметров по умолчанию

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Обобщенный класс:
template<class A=int,class B=char> class MyClass{
public:
   // Поля:
   A first;
   B second;
   // Конструктор:
   MyClass(A f,B s){
      first=f;
      second=s;
   }
   // Метод для отображения значения полей:
   void show(){
      cout<<"Значения "<<first<<" и "<<second<<endl;
   }
};
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
```

```
    // Создание объектов:
    MyClass<double,int> objA(3.5,100);
    MyClass<string> objB("текст",'A');
    MyClass<> objC(200,'B');
    // Вызов метода:
    objA.show();
    objB.show();
    objC.show();
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 7.8)

```
Значения 3.5 и 100

Значения текст и A

Значения 200 и B
```

В основном код читателю должен быть понятен, обратим внимание лишь на принципиальные и новые моменты. В первую очередь они касаются описания шаблонного класса `MyClass`: в угловых скобках после названий обобщенных параметров указаны значения по умолчанию. В частности инструкция `<class A=int,class B=char>` означает, что если при создании объекта на основе обобщенного класса явно не указать идентификаторы типов для обобщенных параметров, то для первого параметра будет использован идентификатор `int`, а для второго параметра будет использован идентификатор `char`. Главная функция программы содержит примеры создания объектов на основе обобщенного класса. Так, команда `MyClass<double,int> objA(3.5, 100)` создает объект с явным указанием используемых идентификаторов для обобщенных параметров (`double` и `int`). В команде `MyClass<string> objB("текст",'A')` явно указан только один идентификатор типа `string`, поэтому для второго обобщенного параметра будет использовано значение по умолчанию `char`. Наконец, в коман-

де `MyClass<> objC(200,'B')` явно не указан ни один идентификатор, поэтому для обобщенных типов используются значения по умолчанию (`int` и `char`).

> **НА ЗАМЕТКУ**
>
> Даже если идентификаторы типов в команде создания объекта явно не указываются, угловые скобки (пустые) все равно присутствуют.

Наследование обобщенных классов

В следующем примере показано, как на основе одного обобщенного класса путем наследования можно создать другой обобщенный класс. Рассмотрим программный код в листинге 7.9.

Листинг 7.9. Наследование обобщенного класса

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Первый базовый обобщенный класс:
template<class A> class Alpha{
public:
   // Поле:
   A alpha;
   // Конструктор:
   Alpha(A a){
      alpha=a;
   }
   // Метод:
   void show(){
      cout<<"Поле alpha: "<<alpha<<endl;
   }
};
// Второй базовый обобщенный класс:
```

```
template<class B> class Bravo{
public:
   // Поле:
   B bravo;
   // Конструктор:
   Bravo(B b){
      bravo=b;
   }
   // Метод:
   void show(){
      cout<<"Поле bravo: "<<bravo<<endl;
   }
};
// Производный обобщенный класс:
template<class A,class B,class C> class Charlie: public
Alpha<A>,public Bravo<B>{
public:
   // Поле:
   C charlie;
   // Конструктор:
   Charlie(A a,B b,C c):Alpha<A>(a),Bravo<B>(b){
      charlie=c;
   }
   // Переопределение метода:
   void show(){
      // Вызов версии метода из первого базового класса:
      Alpha<A>::show();
      // Вызов метода из второго базового класса:
      Bravo<B>::show();
      cout<<"Поле charlie: "<<charlie<<endl;
   }
};
```

```
// Главная функция программа:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Создание объектов:
   Charlie<string,char,int> objA("текст",'A',100);
   Charlie<int,double,char> objB(200,5.5,'B');
   // Вызов метода из объектов:
   objA.show();
   objB.show();
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 7.9)

```
Поле alpha: текст
Поле bravo: A
Поле charlie: 100
Поле alpha: 200
Поле bravo: 5.5
Поле charlie: B
```

Программа достаточно простая. В ней описывается два однотипных обобщенных класса `Alpha` и `Bravo`, а затем на основе этих классов путем наследования создается обобщенный класс `Charlie`. Вообще, схема наследования обобщенных классов напоминает наследование обычных классов. Поправку нужно делать только на то, что в случае с обобщенными классами важно не только название класса, но и обобщенные параметры, которые используются. Скажем, в описании класса `Charlie` после ключевого слова `template` указано выражение `<class A,class B,class C>`, означающее, что в классе `Charlie` используется три обобщенных параметра (`A`, `B` и `C`). Далее в описании как наследуемые указаны классы `Alpha<A>`

и Bravo<B>. Здесь мы не ограничиваемся только названиями базовых классов, но еще и указываем, какие обобщенные параметры следует использовать.

НА ЗАМЕТКУ

Все это означает, что когда создается объект класса Charlie и заданы значения для обобщенных параметров, то первый параметр передается в обобщенный класс Alpha, а второй параметр передается в обобщенный класс Bravo.

Конструктор класса Charlie описан с тремя аргументами. Первый передается аргументом при вызове конструктора базового класса Alpha<A>, а второй аргумент передается конструктору базового класса Bravo<B>. Здесь (имеются в виду инструкции Alpha<A> и Bravo<B>) мы явно подчеркиваем, какие значения для обобщенных параметров необходимо использовать, когда вызывается конструктор базового класса. Аналогичная ситуация имеет место при переопределении в производном классе Charlie метода show(). В теле этого метода командой Alpha<A>::show() вызывается версия метода из первого базового класса, а версия метода из второго базового класса вызывается командой Bravo<B>::show().

В главной функции программы показано как на основе обобщенного производного класса Charlie создаются объекты.

В рассмотренном примере на основе двух обобщенных классов создавался производный обобщенный класс. Можно поступить и несколько иначе: взять обобщенный базовый класс и на его основе наследованием создать обычный класс (или классы). Программа, в которой реализуется данный подход, представлена в листинге 7.10.

Листинг 7.10. Создание обычного класса на основе обобщенного

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Базовый обобщенный класс:
template<class X> class BaseClass{
private:
```

```
    // Закрытое поле:
    X value;
public:
    // Конструктор:
    BaseClass(X val){
        set(val);
    }
    // Метод для присваивания значения полю:
    void set(X val){
        value=val;
    }
    // Метод для получения значения поля:
    X get(){
        return value;
    }
};
// Первый производный класс:
class Alpha: public BaseClass<int>{
public:
    // Конструктор:
    Alpha():BaseClass<int>(0){}
};
// Второй производный класс:
class Bravo: public BaseClass<char>{
public:
    // Конструктор:
    Bravo(char s):BaseClass<char>(s){}
};
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
```

```
    // Объект первого производного класса:
    Alpha objA;
    // Выполнение операций с объектом:
    cout<<"Объект objA: "<<objA.get()<<endl;
    objA.set(100);
    cout<<"Объект objA: "<<objA.get()<<endl;
    // Объект второго производного класса:
    Bravo objB('A');
    // Выполнение операций с объектом:
    cout<<"Объект objB: "<<objB.get()<<endl;
    objB.set('B');
    cout<<"Объект objB: "<<objB.get()<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Как выглядит результат выполнения программы, показано ниже:

Результат выполнения программы (из листинга 7.10)

```
Объект objA: 0
Объект objA: 100
Объект objB: A
Объект objB: B
```

В программе описывается обобщенный класс с названием `BaseClass`. Для класса объявлен один обобщенный параметр, обозначенный как `X`. В самом классе описано закрытое поле `value` относящееся к обобщенному типу `X`. Открытый метод `set()` предназначен для присваивания значения полю `value`, а открытый метод `get()` результатом возвращает значение данного поля. Также в классе описан конструктор с одним аргументом. Значение, переданное аргументом конструктору, через вызов метода `set()` присваивается полю `value`.

На основе класса `BaseClass` наследованием создаются классы `Alpha` и `Bravo`. При создании класса `Alpha` в базовый обобщенный класс

BaseClass в качестве значения обобщенного параметра передается int. При создании класса Bravo класс BaseClass используется с параметром char. В классах Alpha и Bravo описаны конструкторы. Для класса Alpha выполнение конструктора сводится к вызову конструктора класса BaseClass<int> с нулевым аргументом. При выполнении конструктора класса Bravo вызывается конструктор класса BaseClass<char> с одним символьным аргументом.

В функции main() показано, как создаются и используются объекты классов Alpha и Bravo.

Целочисленные обобщенные параметры

Параметры в обобщенном классе служат обозначением для идентификаторов типа. Но есть в этом правиле исключение — в обобщенном классе могут также использоваться целочисленные параметры. Описываются такие параметры подобно стандартным обобщенным параметрам и играют роль, подобную аргументу, который передается в класс. Как пример такой ситуации рассмотрим программу в листинге 7.11. В ней описан обобщенный класс Polynom, предназначенный для реализации полиномов и выполнения некоторых базовых операций, таких, например, как умножение полиномов, сложение и вычитание полиномов, а также вычисление производной для полинома.

ПОДРОБНОСТИ

Для понимания принципов организации программы имеет смысл напомнить особенности таких математических объектов, как полиномы.

Полиномом степени n называется функциональная зависимость вида $P_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Полином как функция однозначно определяется набором коэффициентов $a_0, a_1, \ldots, a_n$. Количество коэффициентов полинома на единицу больше степени полинома. Чтобы узнать значение полинома в точке x, следует вычислить сумму $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$.

Производная $P_n'(x)$ для полинома $P_n(x)$ является полиномом степени $n - 1$ и определяется выражением $P_n'(x) = a_1 + 2a_2x + 3a_3x^2 + \ldots + na_nx^{n-1}$.

Если полином умножается на некоторое число k, то в результате получаем также полином. По сравнению с исходным (умножаемым) полиномом, коэффициенты полинома-результата умножаются на соответствующе число: $kP_n(x) = ka_0x + ka_1x + ka_2x^2 + \ldots + ka_nx^n$.

При вычислении суммы и разности полиномов складываются (вычитаются) значения коэффициентов полиномов, соответствующие одинаковым индексам (одинаковым показателям степени аргумента *x*).

Основу программы составляет обобщенный класс `Polynom`. В нем в качестве обобщенного параметра выступает степень полинома, реализуемого через объект класса. В классе есть обычные и обобщенные операторные методы. Также в программе описаны обобщенные операторные функции. Теперь рассмотрим представленный далее программный код:

Листинг 7.11. Обобщенный класс для реализации полиномов

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Обобщенный класс для реализации полиномов:
template<int power> class Polynom{
private:
   // Закрытое поле – массив с коэффициентами.
   // Размер массива на единицу больше степени полинома:
   double a[power+1];
public:
   // Конструктор без аргументов:
   Polynom(){
      // Заполнение массива нулями:
      for(int k=0;k<=power;k++){
         a[k]=0;
      }
   }
   // Конструктор с аргументом – именем числового массива:
   Polynom(double* nums){
      // Копирование элементов из одного массива в другой:
      for(int k=0;k<=power;k++){
```

```
        a[k]=nums[k];
    }
}
// Метод для отображения значений
// элементов поля – массива:
void getAll(){
    cout<<"| ";
    for(int k=0;k<=power;k++){
        cout<<a[k]<<" | ";
    }
    cout<<endl;
}
// Операторный метод позволяет вызвать объект.
// Результатом возвращается значение полинома:
double operator()(double x){
    double s=0,q=1;
    for(int k=0;k<=power;k++){
        s+=a[k]*q;
        q*=x;
    }
    return s;
}
// Операторный метод для индексирования объектов:
double &operator[](int k){
    return a[k];
}
// Обобщенный операторный метод для вычисления
// произведения полиномов:
template<int n> Polynom<power+n> operator*(Polynom<n> pol){
    // Локальный объект обобщенного класса:
    Polynom<power+n> tmp;
    // Вычисление коэффициентов для
```

```
        // полинома-результата:
        for(int i=0;i<=power;i++){
            for(int j=0;j<=n;j++){
                tmp[i+j]+=a[i]*pol[j];
            }
        }
        // Результат метода:
        return tmp;
    }
    // Обобщенный операторный метод для вычисления
    // суммы полиномов:
     template<int n> Polynom<(n>power?n:power)>
operator+(Polynom<n> pol){
        int i;
        // Локальный объект обобщенного класса:
        Polynom<(n>power?n:power)> tmp;
        // Вычисление коэффициентов для
        // полинома-результата:
        for(i=0;i<=power;i++){
            tmp[i]+=a[i];
        }
        for(i=0;i<=n;i++){
            tmp[i]+=pol[i];
        }
        // Результат метода:
        return tmp;
    }
};
// Обобщенная операторная функция для вычисления
// результата умножения полинома на число:
template<int power> Polynom<power> operator*(Polynom<power>
pol,double r){
```

```
// Локальный объект обобщенного класса:
Polynom<power> tmp;
// Вычисление коэффициентов для полинома-результата:
for(int k=0;k<=power;k++){
   tmp[k]=pol[k]*r;
}
   // Результат функции:
   return tmp;
}
// Обобщенная операторная функция для вычисления
// результата умножения числа на полином:
template<int power> Polynom<power> operator*(double
r,Polynom<power> pol){
   // Полином умножается на число:
   return pol*r;
}
// Обобщенная операторная функция для вычисления
// разности двух полиномов:
template<int m,int n> Polynom<(m>n?m:n)> operator-(Polynom<m>
x,Polynom<n> y){
   // К первому полиному прибавляется второй, умноженный
   // на число -1:
   return x+(-1)*y;
}
// Обобщенная функция для вычисления
// производной от полинома:
template<int power> Polynom<power-1> Diff(Polynom<power> pol){
   // Локальный объект обобщенного класса:
   Polynom<power-1> tmp;
   // Вычисление коэффициентов для полинома-результата:
   for(int k=0;k<=power-1;k++){
      tmp[k]=pol[k+1]*(k+1);
```

```
    }
    // Результат функции:
    return tmp;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Массивы с коэффициентами:
    double A[]={1,2,-1,1};
    double B[]={-1,3,0,2,-1,1};
    // Аргумент для полиномов:
    double x=2;
    // Первый полином:
    Polynom<3> P(A);
    cout<<"Полином P:\t";
    // Коэффициенты первого полинома:
    P.getAll();
    cout<<"Значение P("<<x<<") = ";
    // Значение первого полинома в точке:
    cout<<P(x)<<endl;
    cout<<"Полином P':\t";
    // Коэффициенты для производной от полинома:
    Diff(P).getAll();
    cout<<"Значение P'("<<x<<") = ";
    // Значение производной в точке:
    cout<<Diff(P)(x)<<endl;
    // Второй полином:
    Polynom<5> Q(B);
    cout<<"Полином Q:\t";
    // Коэффициенты второго полинома:
    Q.getAll();
```

```
cout<<"Значение Q("<<x<<") = ";
// Значение второго полинома в точке:
cout<<Q(x)<<endl;
cout<<"Полином P*Q:\t";
// Коэффициенты для произведения полиномов:
(P*Q).getAll();
cout<<"Значение (P*Q)("<<x<<") = ";
// Значение произведения полиномов в точке:
cout<<(P*Q)(x)<<endl;
cout<<"Полином P+Q:\t";
// Коэффициенты для суммы полиномов:
(P+Q).getAll();
cout<<"Значение (P+Q)("<<x<<") = ";
// Значение суммы полиномов в точке:
cout<<(P+Q)(x)<<endl;
    cout<<"Полином Q-P:\t";
    // Коэффициенты для разности полиномов:
    (Q-P).getAll();
    cout<<"Значение (Q-P)("<<x<<") = ";
    // Значение разности полиномов в точке:
    cout<<(Q-P)(x)<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

В результате выполнения программы получаем следующее:

Результат выполнения программы (из листинга 7.11)

```
Полином P:        | 1 | 2 | -1 | 1 |
Значение P(2) = 9
Полином P':       | 2 | -2 | 3 |
```

```
Значение P'(2) = 10
Полином Q:          | -1 | 3 | 0 | 2 | -1 | 1 |
Значение Q(2) = 37
Полином P*Q:        | -1 | 1 | 7 | -2 | 6 | -3 | 5 | -2 | 1 |
Значение (P*Q)(2) = 333
Полином P+Q:        | 0 | 5 | -1 | 3 | -1 | 1 |
Значение (P+Q)(2) = 46
Полином Q-P:        | -2 | 1 | 1 | 1 | -1 | 1 |
Значение (Q-P)(2) = 28
```

В обобщенном классе `Polynom` через `power` обозначен целочисленный параметр. Данный параметр, как отмечалось выше, при создании объекта определяет степень полинома, который отождествляется с объектом класса. Количество коэффициентов в полиноме тогда равно `power+1`. Для запоминания коэффициентов полинома в классе `Polynom` объявляется закрытое числовое поле `a`, являющееся массивом размера `power+1`. Индексы элементов массива `a`, таким образом, изменяются в пределах от `0` до `power` включительно.

В классе описано два конструктора. Конструктор без аргументов все элементы массива заполняет нулями. Есть еще одна версия конструктора, позволяющая передавать аргументом имя массива с коэффициентами полинома. В таком случае значения элементов из массива, переданного аргументом конструктору, копируются в массив `a`. Понятно, что размеры обоих массивов должны совпадать.

В целях удобства и наглядности в классе описан метод `getAll()`, которым в консольном окне отображаются значения коэффициентов полинома.

Чтобы объект класса `Polynom` можно было вызывать как функцию, описывается операторный метод `operator()()`. В теле метода с помощью оператора цикла вычисляется сумма, дающая значение для полинома в точке, определяемой аргументом операторного метода.

Операторный метод `operator[]()` позволяет индексировать объекты класса `Polynom`. Аргументом методу передается целочисленный индекс, а результатом метод возвращает ссылку (инструкция `&` в описании метода) на элемент в массиве `a` с индексом, определяемым

аргументом операторного метода. На этом простой код, в общем-то, заканчивается.

Для вычисления произведения полиномов описан обобщенный операторный метод `operator*()`. Аргумент метода обозначен как `pol` и он описан с типом `Polynom<n>` — то есть является объектом обобщенного класса. Через `n` здесь обозначен целочисленный параметр обобщенного метода. Тип результат метода определяется идентификатором `Polynom<power+n>`, то есть тоже объект обобщенного класса. Выражение `power+n` в угловых скобках равно сумме показателей степеней перемножаемых полиномов.

НА ЗАМЕТКУ

В операторном методе аргумент метода отождествляется со вторым операндом. Первым операндом выступает объект, из которого вызывается метод. Объекту, из которого вызывается метод, соответствует полином степени `power`. Степень второго полинома равна `n`. Поэтому степень полинома, равного произведению полиномов со степенями `power` и `n`, равна `power+n`.

В теле метода командой `Polynom<power+n> tmp` создается локальный объект `tmp` обобщенного класса. Он будет возвращаться результатом метода. Вначале все коэффициенты объекта `tmp` нулевые (благодаря конструктору класса без аргументов). Их нужно вычислить. Для этого запускаются вложенные операторы цикла, в котором индексные переменные `i` и `j` пробегают значение индексов коэффициентов в перемножаемых объектах (полиномах). При фиксированных индексах `i` и `j` выполняется команда `tmp[i+j]+=a[i]*pol[j]`. Здесь, мы, во-первых, воспользовались возможностью индексировать объект `pol` (откровенно говоря, другого способа получить доступ к закрытому массиву данного объекта просто нет). Во-вторых, приняли во внимание, что при перемножении слагаемых со степенями аргумента `i` и `j` получается слагаемое с аргументом в степени `i+j`.

ПОДРОБНОСТИ

Следует учесть, что если $P_n(x) = \sum_{i=0}^{n} a_i x^i$ и $Q_m(x) = \sum_{j=0}^{m} b_j x^j$, то $P_n(x)Q_m(x) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}$. Таким образом, в полиноме-результате коэффициент при слагаемом с x^k равен сумме всех возможных произведений вида $a_i b_j$, для которых $i + j = k$.

После завершения вычислений объект `tmp` возвращается результатом метода.

Похожим образом описывается обобщенный операторный метод для вычисления суммы полиномов. Тип результата определен выражением `Polynom<(n>power?n:power)>`.

Через `n` обозначен целочисленный параметр обобщенного метода. Результатом выражения `n>power?n:power` возвращается большее из значений параметров `n` и `power`.

ПОДРОБНОСТИ

Тернарным оператором `?:` проверяется условие, указанное первым операндом (для выражения `n>power?n:power` это условие `n>power`). Если условие истинно, возвращается значение после вопросительного знака (значение `n`). Если условие ложно, то возвращается значение после двоеточия (значение `power`).

Результатом суммы двух полиномов является полином. Степени суммируемых полиномов в общем случае разные. Если у одного из них степень больше, то такая же степень будет и у полинома-результата. Поэтому в операторном методе необходимо определить, у какого из суммируемых полиномов степень больше. Поэтому и сравниваются значения параметров `power` и `n`. Большее из них определяет степень полинома-результата.

В теле метода локальный объект с нулевыми коэффициентами создается командой `Polynom<(n>power?n:power)> tmp`. К нулевым значениям коэффициентов объекта `tmp` необходимо последовательно добавить значения соответствующих коэффициентов из объекта `pol` (аргумент метода) и массива `a`. Для этого последовательно запускается два оператора цикла. В итоге «заполненный» объект `tmp` возвращается результатом метода.

Кроме класса `Polynom`, в программе описано несколько обобщенных функций. Так, обобщенная операторная функция `operator*()` для вычисления результата умножения полинома на число результатом возвращает объект класса `Polynom<power>` (через `power` обозначен параметр обобщенной функции).

Аргументов у операторной функции два. Первый аргумент `pol` является объектом класса `Polynom<power>`. Второй аргумент `r` является числом типа `double`. В теле функции командой `Polynom<power> tmp` создается локальный объект, и с помощью оператора цикла каждому коэффици-

ент (при фиксированном индексе k) присваивается значение командой tmp[k]=pol[k]*r (произведение коэффициента исходного полинома на число). Затем объект tmp возвращается результатом функции. Следует отметить, что данная функция обрабатывает ситуацию, когда объект (полином) умножается на число.

Чтобы можно было умножать число на объект, необходимо описать еще одну версию операторной функции. Такая функция описана. От первой версии она отличается порядком следования аргументов (сначала указан числовой аргумент r, а затем объект pol).

Чтобы не дублировать программный код в теле функции, мы прибегли к помощи инструкции return pol*r. Проще говоря, для вычисления результата функции, обрабатывающей операцию умножения числа на объект, мы вызываем функцию, обрабатывающую умножение объекта на число.

К похожей хитрости мы прибегли при описании обобщенной операторной функции operator-(), которой вычисляется разность двух объектов (полиномов).

ПОДРОБНОСТИ

Мы исходим из того, что вычитание из полинома $Q_m(x)$ полинома $P_n(x)$ эквивалентно прибавлению к полиному $Q_m(x)$ полинома $P_n(x)$, умноженного на -1. Другими словами, мы воспользовались тождеством $Q_m(x) - P_n(x) = Q_m(x) + (-1)P_n(x)$.

У функции два целочисленных обобщенных параметра m и n (степени вычитаемых полиномов). Результатом возвращается объект класса Polynom<(m>n?m:n)> (степень полинома-результата определяется большим из двух значений параметров m и n). Аргументы функции x и y являются объектами соответственно классов Polynom<m> и Polynom<n>. Результатом функции возвращается значение выражения x+(-1)*y. Оно имеет смысл, поскольку определены операции умножения объектов на число и сложения объектов.

Обобщенная функция Diff() с обобщенным целочисленным параметром power (степень дифференцируемого полинома) для вычисления производной от полинома результатом возвращает объект класса Polynom<power-1> (степень полинома-производной на единицу меньше степени дифференцируемого полинома). Аргумент функции pol (дифференцируемый полином) является объектом класса Polynom<power>.

В теле функции создается локальный объект `tmp` класса `Polynom<power-1>`, и в операторе цикла заполняются значения коэффициентов для данного объекта. Объект `tmp` возвращается результатом функции.

В главной функции программы объявляются и инициализируются два числовых массива `A` и `B`, на основе которых командами `Polynom<3> P(A)` и `Polynom<5> Q(B)` создаются объекты, отождествляемые с полиномами. Далее с объектами выполняются некоторые операции, такие как вычисление производной, умножение, сложение и вычитание.

НА ЗАМЕТКУ

Инструкция `\t` является символом табуляции.

Желающие могут самостоятельно поупражняться в проверке результатов этих вычислений.

НА ЗАМЕТКУ

В рассмотренном примере объекты для полиномов создаются на основе обобщенного класса. Однако каждый раз обобщенному классу передается целочисленный параметр. Формально объекты, созданные на основе одного и того же обобщенного класса, но с разными значениями целочисленного параметра, имеют разные типы. Это накладывает определенные ограничения. Например, при вычислении произведения, суммы, разности полиномов или производной от полинома характеристики класса (имеется в виду значение числового параметра) вычисляются автоматически. Все это выполняется в операторных методах или функциях. Но чтобы присвоить результат вызова такой функции или метода объектной переменной, эта объектная переменная должна быть предварительно объявлена на основе обобщенного класса с таким же целочисленным параметром, как для объекта — результата функции или метода. Поэтому, например, мы пользовались командами `(P*Q).getAll()`, `(P+Q).getAll()`, `(Q-P).getAll()`, `(P*Q)(x)`, `(P+Q)(x)` или `(Q-P)(x)`, а не записывали результат произведения объектов в отдельную переменную.

Рекомендации для самостоятельной работы

- Подумайте, какие универсальные алгоритмы могут быть реализованы с помощью обобщенных функций? Предложите программные коды таких функций.

- Попытайтесь создать обобщенный контейнерный класс, предназначенный для «хранения» упорядоченного списка элементов, в который, кроме прочего, можно было бы уже после создания объекта добавлять элементы и удалять элементы. Другими словами, предлагается создать аналог массива, в который можно добавлять новые элементы (изменяя тем самым размер «массива») и удалять элементы из «массива».
- Попытайтесь создать обобщенный класс, предназначенный для создания «цепочки» объектов или «бинарного дерева». Такой класс должен содержать поле (или поля), являющееся указателем на объект того же обобщенного класса.

Глава 8
РАЗНЫЕ ЗАДАЧИ

Очень убедительно. Мы подумаем, к кому это применить.

из к/ф «31 июня»

В этой главе рассматриваются различные задачи. В частности, мы узнаем:

- что такое структуры и как они используются;
- как в программе реализуются комплексные числа;
- какие обобщенные контейнерные классы могут использоваться для создания массивов, множеств и ассоциативных контейнеров;
- как выполняется обработка исключительных ситуаций;
- что такое многопоточное программирование,

ну и еще кое-что.

Знакомство со структурами

В представленной далее программе задача о вычислении итоговой суммы депозита решается с использованием *структур*.

ТЕОРИЯ

Структура представляет собой набор переменных (разного типа в общем случае), объединенных общим названием. Описание структуры начинается с ключевого слова `struct`, после которого указывается название структуры, и в фигурных скобках — типы и названия переменных, которые входят в структуру (после фигурных скобок ставится точка с запятой). Переменные, входящие в структуру, будем называть полями структуры.

В некотором смысле структура напоминает класс, но только без методов. Структура, подобно классу, определяет своеобразный тип данных. Аналогом объекта класса выступает экземпляр структуры. Для создания экземпляра структуры объявляется переменная, типом которой указывается имя структуры. Если необходимо инициализировать экземпляр структуры, то соответствующей переменной в команде объявления экземпляра структуры присваивается список со значениями полей.

Обращение к полям экземпляра структуры выполняется в «точечной» нотации: после имени экземпляра структуры через точку указывается название поля.

Рассмотрим программный код, представленный в листинге 8.1.

Листинг 8.1. Знакомство со структурами

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Описание структуры:
struct MyMoney{
   // Поля структуры:
   string name;
   double money;
   double rate;
   int time;
};
// Функция для вычисления (на основе экземпляра
// структуры) итоговой суммы:
double getMoney(MyMoney str){
   double s=str.money;
   for(int k=1;k<=str.time;k++){
      s*=(1+str.rate/100);
   }
   return s;
```

```
}
// Функция для отображения полной информации
// для экземпляра структуры:
void show(MyMoney str){
   cout<<"Имя вкладчика: "<<str.name<<endl;
   cout<<"Начальная сумма: "<<str.money<<endl;
   cout<<"Процентная ставка: "<<str.rate<<endl;
   cout<<"Время (лет): "<<str.time<<endl;
   cout<<"Итоговая сумма: "<<getMoney(str)<<endl;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Первый экземпляр структуры:
   MyMoney cat={"Кот Матроскин",1000,8,5};
   // Второй экземпляр структуры:
   MyMoney dog={"Пес Шарик",1200,7,4};
   // Отображение информации:
   show(cat);
   show(dog);
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 8.1)

```
Имя вкладчика: Кот Матроскин
Начальная сумма: 1000
Процентная ставка: 8
Время (лет): 5
```

```
Итоговая сумма: 1469.33
Имя вкладчика: Пес Шарик
Начальная сумма: 1200
Процентная ставка: 7
Время (лет): 4
Итоговая сумма: 1572.96
```

В программе описывается структура с названием `MyMoney`. У структуры несколько полей:

- поле `name` типа `string` (для использования данного класса подключается заголовок `<string>`) — предназначено для записи имени вкладчика;
- поля `money` и `rate` типа `double` — предназначены соответственно для записи начальной суммы вклада и годовой процентной ставки;
- целочисленное поле `time` — предназначено для записи времени размещения вклада (в годах).

В программе для вычисления по значениям полей экземпляра структуры `MyMoney` описана функция `getMoney()`. Аргументом функции передается экземпляр структуры `MyMoney` (обозначен как `str`). Результатом функция возвращает значение типа `double` (итоговая сумма по депозиту).

В теле функции объявляется локальная переменная `s` с начальным значением `str.money` (значение поля `money` экземпляра `str`). Далее запускается оператор цикла. Количество циклов определяется значением поля `time` экземпляра `str`. За каждый цикл выполняется команда `s*=(1+str.rate/100)`, которой текущее значение переменной `s` умножается на величину `(1+str.rate/100)`. По завершении оператора цикла значение переменной `s` возвращается результатом функции.

ПОДРОБНОСТИ

Напомним, что если начальная сумма равна m, годовая процентная ставка равна r, а вклад размещается на время t, то итоговая сумма по депозиту вычисляется по формуле $m(1 + (r/100))^t$.

Функция `show()` не возвращает результат. Аргументом функции передается экземпляр структуры `MyMoney` (обозначен как `str`). При вызове

функции в консольном окне отображаются значения полей name, money, rate и time экземпляра str. Для вычисления итоговой суммы по депозиту вызывается функция getMoney(), а аргументом ей передается экземпляр str.

В главной функции программы командами MyMoney cat={"Кот Матроскин", 1000,8,5} и MyMoney dog={"Пес Шарик",1200,7,4} создаются и инициализируются (полям присваиваются значения) два экземпляра (cat и dog) структуры MyMoney. Командами show(cat) и show(dog) в консольное окно выводится информация по каждому из экземпляров.

Обобщенные структуры

В структуру разрешается передавать обобщенные параметры так, как это делается в обобщенных классах. В листинге 8.2 представлена программа, в которой создается обобщенная структура с двумя полями. Типы полей определяются обобщенными параметрами. Кроме структуры, в программе описана обобщенная функция, предназначенная для работы с экземплярами обобщенной структуры.

Листинг 8.2. Обобщенные структуры

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Обобщенная структура:
template<class A,class B> struct MyStruct{
   A first;
   B second;
};
// Обобщенная функция для работы с экземплярами
// обобщенной структуры:
template<class A,class B> void show(MyStruct<A,B> str){
   cout<<"Первое поле: "<<str.first<<endl;
   cout<<"Второе поле: "<<str.second<<endl;
}
```

```
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Создание экземпляров обобщенной структуры:
   MyStruct<int,char> strA={100,'A'};
   MyStruct<double,string> strB={2.5,"текст"};
   // Вызов обобщенной функции:
   show(strA);
   show(strB);
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 8.2)

```
Первое поле: 100
Второе поле: A
Первое поле: 2.5
Второе поле: текст
```

Описание обобщенной структуры начинается с ключевого слова `template`. Инструкция `<class A,class B>` определяет обобщенные параметры `A` и `B`. В теле структуры поля `first` и `second` описаны с идентификаторами типа `A` и `B` соответственно. В главной функции программы экземпляры `strA` и `strB` обобщенной структуры `MyStruct` создаются командами `MyStruct<int,char> strA={100,'A'}` и `MyStruct<double,string> strB={2.5,"текст"}`. Здесь прослеживается явная аналогия с классами и объектами. Для отображения значений полей экземпляров обобщенной структуры используем функцию `show()`, аргументом которой передается экземпляр структуры (команды `show(strA)` и `show(strB)`).

Функция `show()` описана как обобщенная. У нее в описании два параметра типа (`A` и `B`) и она не возвращает результат. Аргументом функции

передается экземпляр `str` структуры `MyStruct<A,B>`. Идентификация типов (значения параметров `A` и `B`) при вызове функции `show()` выполняется на основании типа аргумента, переданного функции.

Работа с комплексными числами

В библиотеке стандартных классов есть класс `complex`, предназначенный для работы с комплексными числами.

ТЕОРИЯ

Для использования класса `complex` в программе подключается заголовок `<complex>`. Поскольку класс `complex` является обобщенным, то в угловых скобках указывается базовый тип (обычно `double`) для действительной и мнимой частей комплексного числа. Значения действительной и мнимой частей комплексного числа передаются аргументами конструктору.

В представленной далее программе создаются комплексные числа и с ними выполняются базовые вычислительные операции.

ПОДРОБНОСТИ

Любое комплексное число z может быть представлено в виде $z = x + iy$ (алгебраическая форма представления комплексного числа), где действительные числа x и y являются соответственно действительной и мнимой частями комплексного числа, а мнимая единица i такая, что $i^2 = -1$.

Операции сложения, вычитания, умножения и деления выполняются, как и с обычными числами, но с поправкой на то, что $i^2 = -1$. Например, если $z_1 = x_1 + iy_1$ и $z_2 = x_2 + iy_2$, то $z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$, $z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$, $z_1 \cdot z_2 = (x_1 + iy_1) \cdot (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$, а $z_1/z_2 = (x_1 + iy_1)/(x_2 + iy_2) = ((x_1 + iy_1)(x_2 - iy_2))/((x_2 + iy_2)(x_2 - iy_2)) = (x_1x_2 + y_1y_2) + i(x_2y_1 - x_1y_2)/(x_2^2 + y_2^2)$.

Комплексно-спряженным к числу $z = x + iy$ называется число $z^* = x - iy$ (получается из исходного заменой i на $-i$).

Модуль комплексного числа $z = x + iy$ вычисляется как $|z| = \sqrt{x^2 + y^2}$.

Помимо алгебраической формы представления комплексного числа $z = x + iy$, существует тригонометрическая форма представления числа в виде $z = |z|\exp(i\varphi)$, где через φ обозначен аргумент комплексного числа такой, что $\cos(\varphi) = x/(\sqrt{x^2 + y^2})$ и $\sin(\varphi) = y/(\sqrt{x^2 + y^2})$ (здесь нужно учесть, что по определению $\exp(i\varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$).

Рассмотрим программный код, представленный в листинге 8.3.

Листинг 8.3. Работа с комплексными числами

```
#include <iostream>
#include <cstdlib>
#include <complex>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Действительные числа:
   double x=2,y=3;
   // Комплексные числа:
   complex<double> A(3,4),B(2,-1);
   // Сумма комплексных чисел:
   cout<<"Сумма: ";
   cout<<A<<" + "<<B<<" = "<<A+B<<endl;
   // Разность комплексных чисел:
   cout<<"Разность: ";
   cout<<A<<" - "<<B<<" = "<<A-B<<endl;
   // Произведение комплексных чисел:
   cout<<"Произведение: ";
   cout<<A<<" * "<<B<<" = "<<A*B<<endl;
   // Частное комплексных чисел:
   cout<<"Частное: ";
   cout<<A<<" / "<<B<<" = "<<A/B<<endl;
   // Сумма комплексного и действительного чисел:
   cout<<"Сумма: ";
   cout<<A<<" + "<<x<<" = "<<A+x<<endl;
   // Разность комплексного и действительного чисел:
   cout<<"Разность: ";
   cout<<A<<" - "<<x<<" = "<<A-x<<endl;
```

```
// Произведение комплексного и действительного чисел:
cout<<"Произведение: ";
cout<<A<<" * "<<x<<" = "<<A*x<<endl;
// Частное комплексного и действительного чисел:
cout<<"Частное: ";
cout<<A<<" / "<<x<<" = "<<A/x<<endl;
// Сумма действительного и комплексного чисел:
cout<<"Сумма: ";
cout<<y<<" + "<<B<<" = "<<y+B<<endl;
// Разность действительного и комплексного чисел:
cout<<"Разность: ";
cout<<y<<" - "<<B<<" = "<<y-B<<endl;
// Произведение действительного и комплексного чисел:
cout<<"Произведение: ";
cout<<y<<" * "<<B<<" = "<<y*B<<endl;
// Частное действительного и комплексного чисел:
cout<<"Частное: ";
cout<<y<<" / "<<B<<" = "<<y/B<<endl;
// Действительная часть комплексного числа:
cout<<"Действительная часть: ";
cout<<"Re"<<A<<" = "<<A.real()<<endl;
// Мнимая часть комплексного числа:
cout<<"Мнимая часть: ";
cout<<"Im"<<A<<" = "<<A.imag()<<endl;
// Модуль комплексного числа:
cout<<"Модуль: ";
cout<<"abs"<<A<<" = "<<abs(A)<<endl;
// Аргумент комплексного числа:
cout<<"Аргумент: ";
cout<<"arg"<<A<<" = "<<arg(A)<<endl;
// Комплексно-сопряженное число:
cout<<"Комплексно-сопряженное: ";
```

```
cout<<A<<"* = "<<conj(A)<<endl;
// Определение числа на основе модуля и аргумента:
cout<<"Определение числа: ";
cout<<polar(abs(A),arg(A))<<endl;
// Задержка консольного окна:
system("pause>nul");
   return 0;
}
```

Ниже приведен пример выполнения программы:

Результат выполнения программы (из листинга 8.3)

```
Сумма: (3,4) + (2,-1) = (5,3)
Разность: (3,4) - (2,-1) = (1,5)
Произведение: (3,4) * (2,-1) = (10,5)
Частное: (3,4) / (2,-1) = (0.4,2.2)
Сумма: (3,4) + 2 = (5,4)
Разность: (3,4) - 2 = (1,4)
Произведение: (3,4) * 2 = (6,8)
Частное: (3,4) / 2 = (1.5,2)
Сумма: 3 + (2,-1) = (5,-1)
Разность: 3 - (2,-1) = (1,1)
Произведение: 3 * (2,-1) = (6,-3)
Частное: 3 / (2,-1) = (1.2,0.6)
Действительная часть: Re(3,4) = 3
Мнимая часть: Im(3,4) = 4
Модуль: abs(3,4) = 5
Аргумент: arg(3,4) = 0.927295
Комплексно-сопряженное: (3,4)* = (3,-4)
Определение числа: (3,4)
```

Кратко резюмируя, отметим, что объект класса `complex` отождествляется с комплексным числом. С объектами можно выполнять те же операции,

что с обычными числами: складывать, вычитать, умножать и делить. Причем одним из операндов может быть число. При отображении объекта класса `complex` в консольном окне с помощью оператора вывода появляются круглые скобки с парой числовых значений в них — действительная и мнимая части комплексного числа.

Некоторые операции выполняются с помощью специальных функций: модуль числа можно вычислить функцией `abs()`, аргумент числа вычисляется функцией `arg()`, комплексно-сопряженное число вычисляем функцией `conj()`.

Функция `polar()` позволяет создать комплексное число на основе значений его модуля и аргумента. Поэтому, например, если `A` — объект соответствующий комплексному числу, то результатом выражения `polar(abs(A),arg(A))` будет такое же число (проще говоря, если мы ранее создали объект `A` для комплексного числа, то при выполнении команды `polar(abs(A),arg(A))` получим такое же число, но реализованное через другой объект).

Методы `real()` и `imag()` (вызываются из объекта комплексного числа) возвращают результатом соответственно действительную и мнимую части комплексного числа.

Класс для реализации числовых массивов

Обобщенный класс `valarray` библиотеки стандартных классов предназначен для реализации числовых массивов. При работе с классом подключается заголовок `<valarray>`. Далее представлена программа, в которой класс `valarray` используется для создания и заполнения числового массива числами Фибоначчи. Специфика задачи в том, что размер массива определяется в процессе выполнения программы.

> **НА ЗАМЕТКУ**
>
> Здесь и далее мы говорим о массиве, хотя на самом деле имеем дело с объектом класса `valarray`, который по свойствам похож на массив. Использование термина «массив» является удобным и интуитивно понятным, но все же следует понимать, что на самом деле речь идет об объекте.

Рассмотрим программный код, представленный в листинге 8.4.

Листинг 8.4. Размер массива определяется в процессе выполнения программы

```
#include <iostream>
#include <cstdlib>
#include <valarray>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Переменная для записи размера массива:
   int n;
   cout<<"Введите размер массива: ";
   // Считывание размера массива:
   cin>>n;
   // Создание числового массива:
   valarray<int> fibs(n);
   // Значения первых двух элементов:
   fibs[0]=1;
   fibs[1]=1;
   cout<<fibs[0]<<" "<<fibs[1];
   // Заполнение массива и отображение значений элементов:
   for(int k=2;k<n;k++){
      fibs[k]=fibs[k-1]+fibs[k-2];
      cout<<" "<<fibs[k];
   }
   cout<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Возможный результат выполнения программы представлен ниже (жирным шрифтом выделено вводимое пользователем значение):

Результат выполнения программы (из листинга 8.4)

```
Введите размер массива: 15
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

В программе объявляется целочисленная переменная n, значение которой пользователем вводится с клавиатуры. Переменная определяет количество элементов в массиве. Массив реализуется через объект обобщенного класса valarray. Объект, отождествляемый нами с массивом, создается командой valarray<int> fibs(n). Инструкция <int> означает, что элементы массива fibs (будем для простоты называть данный объект массивом) относятся к типу int. Аргумент, указанный в круглых скобках, определяет количество элементов в массиве.

Хотя fibs является объектом, базовые операции с ним выполняются так, как если бы это был обычный массив. В частности данное замечание касается обращения к элементам массива. Скажем, первым двум элементам единичные значения присваиваются командами fibs[0]=1 и fibs[1]=1 — то есть так, как если бы мы имели дело с настоящим массивом. После присваивания значений элементам они отображаются в консоли (команда cout<<fibs[0]<<" "<<fibs[1]). Затем запускается оператор цикла, в котором индексная переменная k пробегает значения от 2 до n-1 включительно. За каждый цикл командой fibs[k]=fibs[k-1]+fibs[k-2] вычисляется значение для текущего элемента, а командой cout<<" "<<fibs[k] это значение отображается в консольном окне.

НА ЗАМЕТКУ

Напомним, что числа Фибоначчи получаются так: первые два числа равны единице, а каждое следующее равно сумме двух предыдущих.

Выше мы использовали объект класса valarray фактически как обычный массив. Однако использование объекта, по сравнению с массивом, имеет неоспоримые преимущества. Достаточно отметить, что размер массива, «спрятанного» в объекте класса valarray, можно не только узнать с помощью метода size(), но и изменить с помощью метода resize(). Таких возможностей при работе с обычными массивами нет.

Еще один небольшой пример использования обобщенного стандартного класса valarray представлен в листинге 8.5. Там представлена програм-

ма, в которой решается уже знакомая нам задача о представлении полинома в виде объекта некоторого класса (такая задача рассматривалась в предыдущей главе — там использовались обобщенные классы с передачей целочисленного значения в качестве параметра). Идея и алгоритм реализованы практически те же, что и в программе в листинге 7.11. Однако за счет использования класса `valarray` код во многих моментах упростился.

НА ЗАМЕТКУ

Напомним, что речь идет о классе для реализации полинома. Полиномиальная функция однозначно определяется набором коэффициентов, которые записываются в массив, являющийся полем класса. Для полиномов предусматриваются такие операции, как вычисление производной, умножение полиномов, сумма и разность полиномов, а также ряд других. Во всех перечисленных случаях результатом является полином. Более подробно операции с полиномами описаны в пояснениях к примеру из листинга 7.11 в предыдущей главе.

Рассмотрим представленную ниже программу:

Листинг 8.5. Класс для реализации полинома

```
#include <iostream>
#include <cstdlib>
#include <valarray>
using namespace std;
// Класс для реализации полиномов:
class Polynom{
private:
   // Закрытое поле – массив, реализованный через
   // объект класса valarray:
   valarray<double> a;
public:
   // Конструктор с целочисленным аргументом:
   Polynom(int n=0){
      // Определение размера массива
```

```
        // и заполнение нулями:
        a.resize(n+1,0);
    }
    // Конструктор с аргументом — именем числового массива
    // и степенью полинома:
    Polynom(double* nums,int n){
        // Создание нового объекта для массива:
        valarray<double> b(nums,n+1);
        // Присваивание полю нового значения:
        a=b;
    }
    // Результатом метода возвращается степень полинома:
    int power(){
        return a.size()-1;
    }
    // Метод для отображения значений
    // коэффициентов полинома:
    void getAll(){
        cout<<"| ";
        for(int k=0;k<=power();k++){
            cout<<a[k]<<" | ";
        }
        cout<<endl;
    }
    // Операторный метод позволяет вызвать объект.
    // Результатом возвращается значение полинома:
    double operator()(double x){
        // Копия массива с коэффициентами полинома:
        valarray<double> b(a);
        double q=1;
        // Вычисление элементов массива:
        for(int k=0;k<b.size();k++){
```

```
        b[k]*=q;
        q*=x;
    }
    // Результат метода - сумма элементов массива:
    return b.sum();
}
// Операторный метод для индексирования объектов:
double &operator[](int k){
    return a[k];
}
// Операторный метод для вычисления
// произведения полиномов:
Polynom operator*(Polynom pol){
    // Локальный объект:
    Polynom tmp(pol.power()+power());
    // Вычисление коэффициентов для
    // полинома-результата:
    for(int i=0;i<=power();i++){
        for(int j=0;j<=pol.power();j++){
            tmp[i+j]+=a[i]*pol[j];
        }
    }
    // Результат метода:
    return tmp;
}
// Операторный метод для вычисления
// суммы полиномов:
Polynom operator+(Polynom pol){
    int i;
    int length=pol.power()>power()?pol.power():power();
    // Локальный объект:
    Polynom tmp(length);
```

```
        // Вычисление коэффициентов для
        // полинома-результата:
        for(i=0;i<=power();i++){
           tmp[i]+=a[i];
        }
        for(i=0;i<=pol.power();i++){
           tmp[i]+=pol[i];
        }
        // Результат метода:
        return tmp;
     }
};
// Операторная функция для вычисления
// результата умножения полинома на число:
Polynom operator*(Polynom pol,double r){
   // Локальный объект:
   Polynom tmp(pol.power());
   // Вычисление коэффициентов для полинома-результата:
   for(int k=0;k<=pol.power();k++){
      tmp[k]=pol[k]*r;
   }
   // Результат функции:
   return tmp;
}
// Операторная функция для вычисления
// результат умножения числа на полином:
Polynom operator*(double r,Polynom pol){
   // Полином умножается на число:
   return pol*r;
}
// Операторная функция для вычисления разности
// двух полиномов:
```

```
Polynom operator-(Polynom x,Polynom y){
   // К первому полиному прибавляется второй, умноженный
   // на число -1:
   return x+(-1)*y;
}
// Обобщенная функция для вычисления
// производной от полинома:
Polynom Diff(Polynom pol){
   // Локальный объект:
   Polynom tmp(pol.power()-1);
   // Вычисление коэффициентов для полинома-результата:
   for(int k=0;k<=tmp.power();k++){
      tmp[k]=pol[k+1]*(k+1);
   }
   // Результат функции:
   return tmp;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Массивы с коэффициентами:
   double A[]={1,2,-1,1};
   double B[]={-1,3,0,2,-1,1};
   // Аргумент для полиномов:
   double x=2;
   // Объект для записи результатов вычислений:
   Polynom res;
   // Первый полином:
   Polynom P(A,3);
   cout<<"Полином P:\t";
   // Коэффициенты первого полинома:
```

```
P.getAll();
cout<<"Значение P("<<x<<") = ";
// Значение первого полинома в точке:
cout<<P(x)<<endl;
// Производная:
res=Diff(P);
cout<<"Полином P':\t";
// Коэффициенты для производной от полинома:
res.getAll();
cout<<"Значение P'("<<x<<") = ";
// Значение производной в точке:
cout<<res(x)<<endl;
// Второй полином:
Polynom Q(B,5);
cout<<"Полином Q:\t";
// Коэффициенты второго полинома:
Q.getAll();
cout<<"Значение Q("<<x<<") = ";
// Значение второго полинома в точке:
cout<<Q(x)<<endl;
// Произведение полиномов:
res=P*Q;
cout<<"Полином P*Q:\t";
// Коэффициенты для произведения полиномов:
res.getAll();
cout<<"Значение (P*Q)("<<x<<") = ";
// Значение произведения полиномов в точке:
cout<<res(x)<<endl;
// Сумма полиномов:
res=P+Q;
cout<<"Полином P+Q:\t";
// Коэффициенты для суммы полиномов:
```

```
    res.getAll();
    cout<<"Значение (P+Q)("<<x<<") = ";
    // Значение суммы полиномов в точке:
    cout<<res(x)<<endl;
    // Разность полиномов:
    res=Q-P;
    cout<<"Полином Q-P:\t";
    // Коэффициенты для разности полиномов:
    res.getAll();
    cout<<"Значение (Q-P)("<<x<<") = ";
    // Значение разности полиномов в точке:
    cout<<res(x)<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 8.5)

```
Полином P:        | 1 | 2 | -1 | 1 |
Значение P(2) = 9
Полином P':       | 2 | -2 | 3 |
Значение P'(2) = 10
Полином Q:        | -1 | 3 | 0 | 2 | -1 | 1 |
Значение Q(2) = 37
Полином P*Q:      | -1 | 1 | 7 | -2 | 6 | -3 | 5 | -2 | 1 |
Значение (P*Q)(2) = 333
Полином P+Q:      | 0 | 5 | -1 | 3 | -1 | 1 |
Значение (P+Q)(2) = 46
Полином Q-P:      | -2 | 1 | 1 | 1 | -1 | 1 |
Значение (Q-P)(2) = 28
```

Во многих местах код должен быть знаком и понятен читателю. Мы выделим лишь то новое, что появилось в программе (по сравнению с программой из листинга 7.11).

Итак, использование объекта класса `valarray` в качестве контейнера для массива позволяет отказаться от описания класса `Polynom` как обобщенного. Теперь это обычный класс. Закрытое поле `a`, через которое реализуется массив с коэффициентами полинома, в классе объявляется инструкцией `valarray<double> a`. В данном случае объявляется пустой объект (не содержит массива) класса `valarray<double>`. Параметры объекта изменяются при вызове конструктора (напомним, что размер содержащегося в объекте класса `valarray` массива можно изменять). Предусмотрены две версии конструктора: с одним аргументом (со значением аргумента по умолчанию) и с двумя аргументами. Если конструктор вызывается с одним аргументом, то аргумент определяет размер массива, а сам массив заполняется нулями. В частности, эти две операции выполняются одной командой `a.resize(n+1,0)` (через `n` обозначен целочисленный аргумент конструктора). Первый аргумент метода `resize()` определяет размер массива, а второй аргумент метода определяет значение, которое присваивается элементам массива.

> **НА ЗАМЕТКУ**
>
> Количество элементов в массиве на единицу больше значения аргумента, переданного конструктору. Аргумент конструктора интерпретируется как степень полинома, который реализуется через объект класса `Polynom`.

Если аргументов у конструктора два, то первый (обозначен как `nums`) является именем числового массива, а второй (обозначен как `n`) определяет степень полинома, реализуемого через данный массив (на единицу меньше размера массива). В теле такого конструктора командой `valarray<double> b(nums,n+1)` создается объект `b` класса `valarray<double>`, в котором содержится массив с такими же элементами, как и массив `nums`. Командой `a=b` объект `b` присваивается значением объекту `a`.

В классе `Polynom` описан метод `power()`, которым в качестве значения возвращается степень полинома, реализуемого через объект. Данное значение на единицу меньше количества элементов в массиве `a` и вычисляется с помощью выражения `a.size()-1`.

> **НА ЗАМЕТКУ**
>
> Функция `power()` используется в программе там, где в примере из листинга 7.11 использовался параметр `power`.

В операторном методе `operator()()` претерпел изменения алгоритм вычисления значения полинома в точке. В теле метода командой `valarray<double> b(a)` создается копия (объект `b`) объекта `a`. Затем в теле оператора цикла, в котором перебираются элементы в массиве `b`, командой `b[k]*=q` очередной элемент умножается на значение переменной `q`, а затем командой `q*=x` переменная `q` умножается на значение аргумента метода `x`. Поскольку начальное значение переменной `q` равно единице, то в итоге получается, что в массиве объекта `b` значения элементов совпадают со слагаемыми в выражении для полинома (коэффициент, умноженный на аргумент `x` в соответствующей степени). Поэтому все, что остается — вычислить сумму элементов в массиве объекта `b`. Сделать это можно с помощью метода `sum()`, который вызывается из объекта `b` (команда `b.sum()`).

Во всем остальном код должен быть понятен читателю. Единственное, на что еще хочется обратить внимание — объявление объекта `res` класса `Polynom`. Значением переменной `res` последовательно присваивается результат таких выражений, как произведение полиномов (объектов класса `Polynom`), их сумма, разность и результат вызова функции `Diff()`. Отображение коэффициентов и вычисление значения полиномов в точке выполняется через обращение к объектной переменной `res`.

Контейнер для динамического массива

В стандартной библиотеке есть класс `vector`, по свойствам напоминающий динамический массив, но по сравнению с массивом функциональные возможности класса намного шире.

> **НА ЗАМЕТКУ**
>
> Для использования класса `vector` подключают заголовок `<vector>`.

В листинге 8.6 предлагается программа, в которой создается массив и заполняется случайными символами. Массив реализуется через объект

класса vector (ситуация похожа на случай, когда массив реализовался через объект класса valarray). Размер массива определяется в процессе выполнения программы.

Рассмотрим представленный далее программный код:

Листинг 8.6. Случайные символы и класс vector

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
   // Количество символов:
   int n;
   cout<<"Количество символов: ";
   // Считывание значения переменной:
   cin>>n;
   // Объект с символьным массивом:
   vector<char> symbs(n);
   cout<<"|";
   // Вычисление значений элементов и их
   // отображение в консольном окне:
   for(int k=0;k<symbs.size();k++){
      // генерирование случайного символа:
      symbs[k]='A'+rand()%(n+5);
      // Отображение значения элемента:
      cout<<" "<<symbs[k]<<" |";
   }
   cout<<endl;
```

```
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Возможный результат (возможный — потому что используется генератор случайных чисел) выполнения программы представлен ниже (жирным шрифтом выделено введенное пользователем значение):

Результат выполнения программы (из листинга 8.6)

```
Количество символов: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Хотя код простой, проанализируем его. А именно, стоит обратить внимание, что объект с массивом создается командой `vector<char> symbs(n)`. Здесь речь идет об объекте `symbs`, который создается на основе обобщенного класса `vector` и содержит массив из `n` элементов типа `char`.

Обращаться с объектом `symbs` можно как с обычным массивом. В частности, объект индексируется.

Метод `size()`, вызываемый из объекта, позволяет определить размер массива. Поэтому в операторе цикла значение элементам массива в объекте `symbs` присваивается командой `symbs[k]='A'+rand()%(n+5)`, при этом индексная переменная `k` пробегает значения от `0` до `symbs.size()-1` включительно. Отображается элемент в консольном окне командой `cout<<" "<<symbs[k]<<" |"`.

Эта же задача, но реализованная с помощью *итераторов*, представлена в программном коде в листинге 8.7.

ТЕОРИЯ

Итератор — объект, имеющий свойства указателя. Другими словами, итератор представляет собой объект, с которым можно выполнять операции так, как если бы объект был указателем. Итераторы создаются для работы с контейнерными классами — такими, например, как обобщенный класс `vector`.

Принципы использования итераторов обсудим на примере, представленном ниже:

Листинг 8.7. Использование итераторов

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
// Изменение кодировки консоли:
system("chcp 1251>nul");
// Инициализация генератора случайных чисел:
srand(2);
// Количество символов:
int n;
cout<<"Количество символов: ";
// Считывание значения:
cin>>n;
// Объект с символьным массивом:
vector<char> symbs(n,'A');
// Итератор:
vector<char>::iterator p;
cout<<"|";
// Вычисление значений элементов массива
// и отображение этих значений в консоли:
for(p=symbs.begin();p!=symbs.end();p++){
   // Вычисление значения элемента
   // (доступ через итератор):
   *p+=rand()%(n+5);
   // Отображение значения элемента:
   cout<<" "<<*p<<" |";
}
cout<<endl;
// Задержка консольного окна:
```

```
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой же, как и в предыдущем случае (жирный шрифт — то, что вводит пользователь):

Результат выполнения программы (из листинга 8.7)

```
Количество символов: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Контейнерный объект для реализации массива создается командой `vector<char> symbs(n,'A')`. Второй аргумент `'A'`, который передается конструктору, служит значением для элементов массива. Проще говоря, в созданном массиве у всех элементов значение будет `'A'`. Для работы с данным контейнерным объектом создается итератор `p`. Объявляется итератор командой `vector<char>::iterator p`. В соответствии с ней итератор `p` является объектом класса `iterator`, который является внутренним классом класса `vector<char>`.

НА ЗАМЕТКУ

Внутренний класс — класс, описанный в другом (внешнем) классе.

Метод `begin()` объекта `symbs` результатом возвращает итератор, указывающий на начальный элемент в массиве объекта `symbs`. В операторе цикла в первом блоке командой `p=symbs.begin()` переменной `p` значением присваивается объект-итератор, указывающий на начальный элемент массива. В третьем блоке размещена команда `p++`, которой значение итератора «увеличивается на единицу»: в результате новое значение переменной `p` будет итератором, указывающим на следующий элемент массива.

НА ЗАМЕТКУ

Вообще, чтобы понять принцип работы с итератором, разумно интерпретировать итератор как указатель.

Проверяемым условием в операторе цикла указано выражение `p!=symbs.end()`. В нем использован результат вызова метода `end()`

из объекта symbs. Результатом метод end() возвращает итератор, указывающий на область памяти, расположенную сразу за последним элементом. Условие p!=symbs.end() истинно, пока итератор p указывает на элемент массива, и становится ложным, когда итератор «выйдет» за пределы массива. В таком случае выполнение оператора цикла завершается.

По аналогии с указателем, чтобы получить доступ к элементу, на который указывает итератор, пред итератором следует поставить звездочку *. Поэтому в результате выполнения команды *p+=rand()%(n+5) к начальному значению элемента (а это значение 'A'), на который указывает итератор p, прибавляется значение rand()%(n+5) (и через механизм автоматического приведения типов полученный в итоге код преобразуется в символ). Командой cout<<" "<<*p<<" |" значение элемента, на который указывает итератор p, отображается в консольном окне.

В рассмотренных примерах контейнерный объект нужного размера создавался сразу. Подход может быть иным, когда размер массива, «спрятанного» в контейнерном объекте, увеличивается по мере добавления элементов в массив. Подобный способ организации кода проиллюстрирован в программе из листинга 8.8.

Листинг 8.8. Изменение размеров контейнерного объекта

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Инициализация генератора случайных чисел:
   srand(2);
   // Количество элементов:
   int n;
   cout<<"Количество символов: ";
   // Считывание значения для переменной:
   cin>>n;
   // Создание пустого контейнерного объекта:
```

```
vector<char> symbs;
cout<<"|";
// Добавление элементов в контейнерный объект:
while(symbs.size()<n){
   // В конец массива добавляется новый элемент:
   symbs.push_back('A'+rand()%(n+5));
   // Отображение значения
   // последнего элемента в массиве:
   cout<<" "<<symbs[symbs.size()-1]<<" |";
}
cout<<endl;
// Задержка консольного окна:
system("pause>nul");
return 0;
}
```

Ниже представлен результат выполнения программы (жирным шрифтом выделено вводимое пользователем значение):

Результат выполнения программы (из листинга 8.8)

```
Количество символов: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Видим, что результат такой же, как в предыдущих случаях. Теперь постараемся понять, почему все происходит именно так, а не иначе.

Командой `vector<char> symbs` в программе создается пустой (без массива или с массивом нулевой длины) контейнерный объект `symbs` (размер внутреннего массива в этом объекте изменяется в процессе выполнения программы). Для добавления элементов в контейнерный объект запускается оператор цикла `while`. Условие в операторе цикла `symbs.size()<n` состоит в том, что размер массива в объекте `symbs` меньше значения `n`, которое пользователь ввел с клавиатуры.

В операторе цикла командой `symbs.push_back('A'+rand()%(n+5))` в конец массива в объекте `symbs` добавляется новый элемент, а значение этого элемента определяется аргументом метода `push_back()`. Далее, поскольку методом `size()` возвращается количество элементов в массиве

объекта, из которого вызывается метод, то индекс последнего элемента на единицу меньше результата метода `size()`. Таким образом, командой `cout<<" "<<symbs[symbs.size()-1]<<" |"` отображается значение последнего (на данный момент) элемента в массиве. Несложно догадаться, что речь идет о том элементе, который был добавлен в массив предыдущей командой.

Контейнерный класс для реализации множества

Модифицируем рассмотренную выше задачу о заполнении массива случайными символами, наложим дополнительное ограничение: символы должны быть разными. Также нас будет интересовать только набор символов, безотносительно к порядку, в котором они были сгенерированы.

Для решения задачи воспользуемся контейнерным классом `set`, позволяющим реализовать *множества*.

ПОДРОБНОСТИ

Множество (с математической точки зрения) представляет собой неупорядоченный набор разных элементов. Таким образом, у множества не может быть двух идентичных элементов, а порядок следования элементов в множестве не фиксирован — имеет значение лишь входит элемент в множество или нет. Добавление в множество элемента, который там уже есть, на самом деле не приводит к появлению в множестве нового элемента. Проще говоря, элемент добавляется в множество, если там такого элемента еще нет.

В листинге 8.9 представлена программа, в которой на основе класса `set` создается и заполняется множество, состоящее из случайных символов.

Листинг 8.9. Реализация множества на основе класса map

```
#include <iostream>
#include <cstdlib>
#include <set>
using namespace std;
int main(){
   // Изменение кодировки консоли:
```

```
system("chcp 1251>nul");
// Инициализация генератора случайных чисел:
srand(2);
// Количество разных символов:
int n;
cout<<"Количество разных символов: ";
// Считывание значения:
cin>>n;
// Создание пустого контейнерного объекта:
set<char> symbs;
// Символьная переменная:
char s;
// Счетчик символов:
int counter=0;
// Заполнение множества:
while(symbs.size()<n){
   // Случайный символ:
   s='A'+rand()%(n+5);
   // Изменение значения счетчика символов:
   counter++;
   // Отображение сгенерированного символа:
   cout<<s<<" ";
   // Добавление элемента в множество:
   symbs.insert(s);
}
cout<<"\nВсего сгенерировано "<<counter<<" символов\n";
cout<<"Разные символы:\n";
// Итератор для работы с элементами множества:
set<char>::iterator p;
// Итератор установлен на начальный
// элемент множества:
p=symbs.begin();
```

```
    cout<<"|";
    // Отображается содержимое множества:
    while(p!=symbs.end()){
        // Текущий элемент:
        cout<<" "<<*p<<" |";
        // Итератор на следующий элемент:
        p++;
    }
    cout<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже показано, как может выглядеть результат выполнения программы (жирным шрифтом выделено вводимое пользователем значение):

Результат выполнения программы (из листинга 8.9)

```
Количество разных символов: 12
L K H N G P E N M H P D E L G N M F L K G M N K D O M K M P Q
Всего сгенерировано 31 символов
Разные символы:
| D | E | F | G | H | K | L | M | N | O | P | Q |
```

Рассмотрим более детально программный код. Командой `set<char> symbs` создается контейнерный объект для реализации множества, в котором на момент создания нет элементов. Символьная переменная `s` предназначена для запоминания сгенерированных символов, а в целочисленную переменную `counter` (с нулевым начальным значением) записывается количество *всех* сгенерированных символов. При этом значение переменной `n`, которое считывается с клавиатуры, определяет количество *разных* символов. Заполнение множества выполняется в операторе цикла `while`, в котором проверяется условие `symbs.size()<n` (количество элементов в множестве меньше значения переменной `n`).

В теле оператора цикла командой `s='A'+rand()%(n+5)` генерируется случайный символ и записывается в переменную `s`. После этого значение

переменной `counter` увеличивается на единицу, а сгенерированный символ отображается в консольном окне. Далее командой `symbs.insert(s)` элемент добавляется в множество. Но в отличие от случая с массивом, при добавлении элемента в множество элемент на самом деле добавляется, только если в множестве такого элемента нет. Оператор цикла завершается, когда размер множества станет равным `n` — причем по определению все элементы в множестве имеют разные значения.

Для отображения элементов множества в программе командой `set<char>::iterator p` объявляется итератор для работы с элементами множества. Итератор установлен на начальный элемент множества с помощью команды `p=symbs.begin()`. В операторе цикла `while` командой `cout<<" "<<*p<<" |"` отображается значение текущего (на который указывает итератор `p`) элемента множества, после чего командой `p++` итератор перебрасывается на следующий элемент. Процесс продолжается, пока выполняется условие `p!=symbs.end()` — то есть пока итератор не станет указывать на область памяти за последним элементов в множестве.

НА ЗАМЕТКУ

Поскольку элементы в множестве не упорядочены, то сказать однозначно, какой элемент первый, а какой последний весьма проблематично. Важно то, что какой-то из них первый, а какой-то последний. Если посмотреть на результат выполнения программы, то легко заметить, что элементы множества отображаются в алфавитном порядке а не в том, как они добавлялись в множество.

Ассоциативный контейнер

Контейнерный класс `map` позволяет создавать *ассоциативные контейнеры* — аналог массива, в котором роль индексов могут играть значения нечисловых типов. Аналог индекса называется *ключом* элемента. Таким образом, у каждого элемента в ассоциативном массиве есть значение и есть ключ.

ТЕОРИЯ

Для использования контейнерного класса `map` в программе подключается заголовок `<map>`. У обобщенного класса `map` два обобщенных параметра: тип ключа и тип значения (указываются в угловых скобках после имени класса).

Доступ к элементу контейнера осуществляется, кроме прочего, через ключ: ключ указывается (как индекс) в квадратных скобках после имени объекта или передается аргументом методу `at()`, который вызывается из объекта контейнера.

Вставка элемента в контейнерный объект может выполняться с помощью метода `insert()`. Аргументом методу передается объект типа `pair` (название обобщенной структуры, для использования которой подключается заголовок `<utility>`). У экземпляра структуры `pair` имеются два поля: поле `first` соответствует ключу элемента, а поле `second` соответствует значению элемента.

Пример использования ассоциативного массива, созданного на основе контейнерного класса `map`, представлен в листинге 8.10.

Листинг 8.10. Ассоциативный контейнер

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <map>
#include <utility>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Объект для ассоциативного контейнера:
   map<string,int> numbers;
   // Размер массива:
   const int n=5;
   // Текстовый массив с ключами:
   string names[n]={"один","два","три","четыре","пять"};
   // Числовой массив со значениями элементов:
   int nms[n]={1,2,3,4,5};
   // Добавление элементов в контейнер:
   for(int k=0;k<n;k++){
      numbers.insert(pair<string,int>(names[k],nms[k]));
```

```
}
// Добавляется еще один элемент:
numbers.insert(pair<string,int>("шесть",6));
// Удаление элемента из массива:
numbers.erase("три");
// Итератор для работы с ассоциативным
// контейнерным объектом:
map<string,int>::iterator p;
// Итератор установлен на начальный элемент:
p=numbers.begin();
// Отображение содержимого контейнера:
while(p!=numbers.end()){
   cout<<(*p).first<<"\t- "<<(*p).second<<endl;
   p++;
}
// Явное обращение к элементам по ключу:
   cout<<"Единица\t - это "<<numbers["один"]<<endl;
   cout<<"Двойка\t - это "<<numbers.at("два")<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 8.10)

```
два     - 2
один    - 1
пять    - 5
четыре  - 4
шесть   - 6
Единица  - это 1
Двойка   - это 2
```

Прокомментируем программный код. В программе командой `map<string,int> numbers` создается объект `numbers`, который является ассоциативным контейнером (пока что без элементов). Ключами элементов в контейнере могут быть значения типа `string`, а значения элементов относятся к типу `int`.

Для автоматического заполнения контейнера объявляется два массива: текстовый `names` с текстовыми названиями цифр (значения для ключей элементов контейнера) и числовой массив `nms` с числовыми значениями (значения для элементов контейнера). Заполняется контейнер `numbers` с помощью оператора цикла, в котором индексная переменная `k` перебирает индексы в массивах `names` и `nms` (массивы одинакового размера). Добавление элементов в контейнер выполняется с помощью команды `numbers.insert(pair<string,int>(names[k],nms[k]))`. Для вставки элемента в контейнер использован метод `insert()`, аргументом которому передается добавляемый элемент. Элемент в данном случае — это два параметра (ключ элемента и значение элемента). Элемент формируется инструкцией `pair<string,int>(names[k],nms[k])`. Здесь речь идет о создании анонимного экземпляра структуры обобщенного типа `pair` (с использованием типов `string` и `int`). Два значения (`names[k]` и `nms[k]`), указанные в круглых скобках — соответственно ключ и значение элемента.

Команда `numbers.insert(pair<string,int>("шесть",6))` после оператора цикла дает еще один пример вставки элемента в контейнер, но теперь ключ и значение элемента указаны в виде литералов (текстового и целочисленного). После добавления элементов в контейнер, один из них удаляется из контейнера — используем для этого команду `numbers.erase("три")`, в которой из объекта контейнера вызывается метод `erase()`, а аргументом ему передается ключ удаляемого элемента.

Отображение содержимого контейнера реализуем через итератор. Итератор для работы с ассоциативным контейнерным объектом объявляется командой `map<string,int>::iterator p`. Командой `p=numbers.begin()` итератор устанавливается на начальный элемент контейнера.

> **НА ЗАМЕТКУ**
>
> Поскольку в контейнерном объекте элементы не упорядочены, то понятие «первый» и «последний» в отношении элементов достаточно условное. Понятно, что как-то элементы в памяти распределены, и при переборе какой-то из них просматривается первым, а какой-

то последним. Но все равно как такового порядка следования элементов в контейнере нет.

Отображается содержимое контейнерного объекта с помощью оператора цикла `while`. Цикл выполняется, пока истинно условие `p!=numbers.end()` (то есть пока итератор установлен на элемент контейнера). В теле оператора цикла отображаются значения `(*p).first` для ключа элемента и `(*p).second` для значения элемента, на который установлен итератор `p`. После этого командой `p++` итератор «перебрасывается» на следующий элемент. Другой пример обращения к элементам контейнера по ключу дается инструкциями `numbers["один"]` и `numbers.at("два")`.

Обработка ошибок

Ранее мы уже встречались с обработкой исключений — правда, мы их генерировали вручную для создания точек ветвления в программе. Теперь рассмотрим пример, в котором механизм обработки исключений используется, так сказать, «по прямому назначению».

В задаче о создании массива с числами Фибоначчи (из листинга 8.4) размер массива вводится с клавиатуры. Теоретически возможна ситуация, когда пользователь вводит некорректное значение для размера массива (например, отрицательное число). В таком случае при попытке создать массив возникнет ошибка. Чтобы обработать подобную ошибку, дополним программный код блоком `try-catch`, благодаря чему при вводе отрицательного значения для размера массива аварийного завершения программы не будет. Рассмотрим программный код в листинге 8.11.

Листинг 8.11. Обработка ошибок

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Переменная для записи размера массива:
   int n;
```

```
    cout<<"Числа Фибоначчи\n";
    cout<<"Введите размер массива: ";
    // Считывание размера массива:
    cin>>n;
    // Контролируемый код:
    try{
    // Создание числового массива:
    vector<int> fibs(n,1);
    // Отображение значений первых двух элементов:
    cout<<fibs[0]<<" "<<fibs[1];
    // Заполнение массива и отображение
    // значений элементов массива:
    for(int k=2;k<n;k++){
       fibs[k]=fibs[k-1]+fibs[k-2];
       cout<<" "<<fibs[k];
    }
    }
    // Блок перехвата ошибки:
    catch(...){
       cout<<"Произошла ошибка!";
    }
    cout<<"\nПрограмма завершила выполнение...\n";
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы может быть таким (если пользователем введено корректное значение для размера массива):

Результат выполнения программы (из листинга 8.11)

```
Числа Фибоначчи
Введите размер массива: 10
1 1 2 3 5 8 13 21 34 55
Программа завершила выполнение...
```

Или таким (если введено некорректное значение):

Результат выполнения программы (из листинга 8.11)

```
Числа Фибоначчи
Введите размер массива: -3
Произошла ошибка!
Программа завершила выполнение...
```

В данной программе мы для создания массива с числами Фибоначчи используем контейнерный класс `vector`. Программный код с командой `vector<int> fibs(n,1)` для создания объекта `fibs` с внутренним массивом размещен в `try`-блоке.

НА ЗАМЕТКУ

Массив состоит из `n` элементов с единичными значениями, поэтому отпадает необходимость в явном виде присваивать единичные значения первым двум элементам.

Если при выполнении команды по созданию массива и последующих команд ошибок не возникает, то `catch`-блок игнорируется. Если же возникает ошибка, то код в `try`-блоке прекращает выполняться, и будет выполнен код в `catch`-блоке.

НА ЗАМЕТКУ

Троеточие в круглых скобках после инструкции `catch` означает, что данный блок перехватывает ошибки всех типов, которые могут возникнуть в `try`-блоке.

Знакомство с многопоточным программированием

В следующем примере в программе выполняется несколько *потоков*.

ТЕОРИЯ

Потоки — разные части программы, которые выполняются одновременно. Общая схема такая: в главном потоке, который отождествляется с выполнением функции `main()`, могут запускаться **дочерние**

потоки. Код, который выполняется в рамках дочерних потоков, обычно оформляется в виде функции или функций, которые вызываются в главном потоке в режиме параллельного выполнения кода. Для реализации в программе многопоточного подхода подключается заголовок <thread>. Для запуска дочернего потока в главном потоке создается объект класса thread, а аргументом конструктору при создании объекта передается имя функции, которая выполняется в рамках дочернего потока.

В листинге 8.12 вниманию читателя предлагается достаточно простой пример программы, в которой помимо главного потока выполняется еще и два дочерних потока. В главном потоке и двух дочерних потоках в консольное окно выводятся сообщения. Специфика ситуации в том, что вывод сообщений выполняется одновременно всеми потоками. Для начала имеет смысл взглянуть на представленный далее программный код:

Листинг 8.12. Многопоточное программирование

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <thread>
#include <chrono>
#include <mutex>
using namespace std;
mutex m;
// Функция для создания потоков:
void mythread(string name,int time,int steps){
for(int k=1;k<=steps;k++){
   // Временная задержка в выполнении команд:
   this_thread::sleep_for(chrono::seconds(time));
   // Блокировка доступа к ресурсу (консоли):
   m.lock();
   // Отображение сообщения в консоли:
   cout<<"Поток "<<name<<":\tсообщение "<<k<<endl;
```

```
        // Разблокировка ресурса (консоли):
        m.unlock();
    }
}
// Главная функция программы:
int main(){
// Изменение кодировки консоли:
system("chcp 1251>nul");
// Количество циклов в потоках:
int n=5;
cout<<"Запускаются потоки...\n";
// Первый дочерний поток:
thread A(mythread,"Alpha",4,n);
// Второй дочерний поток:
thread B(mythread,"Bravo",3,n);
// Вызов функции в главном потоке:
mythread("Main",2,n);
// Ожидание завершения первого дочернего потока:
if(A.joinable()){
   A.join();
}
// Ожидание завершения второго дочернего потока:
if(B.joinable()){
   B.join();
}
   cout<<"Выполнение программы завершено...\n";
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Возможный результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 8.12)

```
Запускаются потоки...
Поток Main:     сообщение 1
Поток Bravo:    сообщение 1
Поток Alpha:    сообщение 1
Поток Main:     сообщение 2
Поток Bravo:    сообщение 2
Поток Main:     сообщение 3
Поток Alpha:    сообщение 2
Поток Main:     сообщение 4
Поток Bravo:    сообщение 3
Поток Main:     сообщение 5
Поток Alpha:    сообщение 3
Поток Bravo:    сообщение 4
Поток Bravo:    сообщение 5
Поток Alpha:    сообщение 4
Поток Alpha:    сообщение 5
Выполнение программы завершено...
```

Не лишним будет проанализировать и пояснить ключевые элементы программы. В первую очередь следует охарактеризовать ее общую структуру.

В программе кроме функции `main()` описана еще и пользовательская функция `mythread()` с тремя аргументами. В главной функции `main()` функция `mythread()` используется трижды: функция вызывается в двух дочерних потоках, а также есть команда вызова функции непосредственно в функции `main()`.

Общий алгоритм выполнения функции `mythread()` следующий. При вызове функции запускается оператор цикла, в котором определенное количество раз в консольное окно выводится сообщение. В сообщении (для идентификации потока) содержится значение первого текстового аргумента функции `mythread()` и номер сообщения. Количество выводимых сообщений определяется третьим аргументом функции `mythread()`. Между выводом сообщений делается пауза. Длительность паузы (в се-

кундах) определяется вторым аргументом функции mythread(). Как это все организовано в теле функции мы рассмотрим далее, а сейчас проанализируем код функции main().

Итак, в теле главной функции объявляется переменная n со значением 5. Данное значение определяет количество сообщений, которое выводится в консоль каждым из потоков. Перед запуском дочерних потоков командой cout<<"Запускаются потоки...\n" в консоль выводится сообщение, служащее индикатором того, что начинается запуск дочерних потоков. Первый дочерний поток запускается благодаря команде thread A(mythread,"Alpha",4,n). В результате выполнения команды создается объект A класса thread (для использования класса thread подключается заголовок <thread>). Объект A отождествляется с первым дочерним потоком. Доступ к этому потоку осуществляется через объект A. Аргументы, переданные конструктору при создании объекта A, означают, что в рамках данного потока будет выполняться код функции mythread() с аргументами "Alpha", 4 и n (всего будет n сообщений с текстом "Alpha" и интервалом между сообщениями в 4 секунды). Как только поток создан, не дожидаясь его завершения и параллельно с ним, выполняется команда thread B(mythread,"Bravo",3,n), означающая создание второго дочернего потока. Доступ к потоку осуществляется через объект B, а в рамках потока выполняется функция mythread() с аргументами "Bravo", 3 и n (всего будет n сообщений с текстом "Bravo" и интервалом между сообщениями в 3 секунды). В результате, наравне с главным потоком (код функции main()) будет выполняться уже два дочерних потока. Далее, командой mythread("Main",2,n) функция mythread() запускается и в главном потоке (появится n сообщений с текстом "Main" и интервалом между сообщениями в 2 секунды).

Получаем три параллельных процесса (главный и два дочерних потока), и у них разное время выполнения. Нам необходимо обеспечить ситуацию, когда до завершения главного потока закончатся все дочерние потоки. Для этого используются команды вида A.join() и B.join(). Команды означают, что перед выполнением следующей инструкции нужно дождаться завершения соответственно первого потока и второго потока. Однако перед тем как вызывать метод join() из объекта соответствующего потока, необходимо проверить, что поток еще выполняется. Проверить данное обстоятельство можно с помощью метода joinable(), который вызывается из объекта потока, а сама инструкция передается в качестве условия в условный оператор. После завершения двух дочерних потоков командой cout<<"Выполнение программы

завершено...\n" в главном потоке выводится сообщение о завершении программы.

Теперь разберемся с программным кодом функции mythread(). Код функции простой и небольшой, но содержит несколько новых для нас команд и незнакомых (пока еще) синтаксических конструкций. В частности, командой this_thread::sleep_for(chrono::seconds(time)) выполняется блокировка выполнения команд на время (в секундах), определяемое аргументом time функции. Использование данной команды возможно благодаря подключению заголовка <chrono> в программе. Команда означает следующее:

- Вызывается функция sleep_for(), описанная в пространстве имен this_thread (данное пространство имен доступно после подключения заголовка <thread>). Пространство имен, в котором определена функция, указывается явно перед именем функции и отделяется от имени функции оператором расширения контекста ::.
- Аргументом функции sleep_for() передается результат вызова функции seconds() (результатом функции является объект, определяющих характеристики интервала для задержки в выполнении кода) из пространства имен chrono. Пространство имен chrono доступно после подключения заголовка <chrono>.

Перед командой cout<<"Поток "<<name<<":\tсообщение "<<k<<endl, которой в консольном окне отображается сообщение, выполняется команда m.lock(), а после отображения сообщения выполняется команда m.unlock(). Если кратко, то выполнение команды m.lock() приводит к блокировке консольного окна, а выполнение команды m.unlock() такую блокировку снимает. Если более детально, то ситуация следующая: поскольку несколько потоков выполняются одновременно и все они обращаются к консольному окну, то теоретически возможно, что в процесс вывода сообщения одним потоком может «вклиниться» другой поток, так что внутри одного сообщения появляется другое. Мы хотим перестраховаться и добиться того, что вывод сообщения в консоль одним потоком приводит к автоматической блокировке консоли для других потоков. Как только сообщение выведено, консоль разблокируется для использования другими потоками. Блокировка и разблокировка ресурсов (синхронизация потоков) выполняется посредством мьютексного объекта. Мьютекстный объект — объект класса mutex. Класс доступен после подключения заголовка <mutex>. Объект должен быть доступен во всех потоках, поэтому создается как глобальный командой mutex m.

Когда каким-то из потоков командой `m.lock()` из мьютексного объекта `m` вызывается метод блокировки `lock()`, то это является индикатором для прочих потоков о блокировке ресурса. Ресурс снова доступен после того, как в заблокировавшем ресурс потоке из мьютексного объекта вызывается метод `unlock()`.

Рекомендации для самостоятельной работы

- В задачах из предыдущих глав, в которых использовались классы (включая обобщенные) и объекты, рассмотреть возможность использования структур и внешних функций.
- В задачах из предыдущих глав, в которых использовались статические и динамические массивы, рассмотреть возможность использования контейнерных классов (таких, как `vector`, `valarray`, `set` и `map`).
- Для рассмотренных ранее задач предложить механизм перехвата и обработки возможных ошибок.

Глава 9
МАТЕМАТИЧЕСКИЕ ЗАДАЧИ

— Какая гадость.
— Это не гадость. Это последние достижения современной науки.

из к/ф «31 июня»

В этой главе мы рассмотрим некоторые математические задачи, которые традиционно входят в университетский курс по вычислительным методам и подразумевают написание специальных программ. В частности, мы сосредоточим внимание на:

- решении алгебраических уравнений;
- построении интерполяционных полиномов;
- вычислении интегралов;
- решении дифференциальных уравнений,

причем практически для всех из перечисленных типов задач предлагается несколько способов решения. Эта глава — особая.

> **НА ЗАМЕТКУ**
>
> Во-первых, здесь рассматриваются исключительно математические задачи. А во-вторых, разбор программных кодов требует от читателя некоторой самостоятельности, поскольку комментарии в книге касаются в первую очередь математической стороны вопроса и общего алгоритма решения задачи. Относительно программных кодов приводятся лишь пояснения общего характера. Такой подход должен способствовать лучшему закреплению рассмотренного ранее материала и позволит проверить полученные знания и навыки.

Метод последовательных приближений

При решении уравнения вида $x = \varphi(x)$ при некоторых дополнительных условиях может использоваться метод последовательных приближений. Суть его сводится к тому, что задается начальное приближение x_0

для корня уравнения, после чего, используя итерационную процедуру, каждое следующее приближение вычисляется на основе предыдущего в соответствии с формулой $x_{n+1} = \varphi(x_n)$. В листинге 9.1 представлена программа, с помощью которой описанным способом решаются алгебраические уравнения $x = 0{,}5 \cdot \cos(x)$, $x = \exp(-x)$ и $x = (x^2 + 6)/5$.

Листинг 9.1. Метод последовательных приближений

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>
using namespace std;
// Функция для решения уравнения методом
// последовательных приближений:
double findRoot(double (*f)(double),double x0,int n){
   // Начальное приближение для корня уравнения:
   double x=x0;
   // Последовательные итерации:
   for(int k=1;k<=n;k++){
      x=f(x);
   }
   // Результат функции:
   return x;
}
// Функции для определения правой
// части решаемых уравнений:
double f(double x){
   return 0.5*cos(x);
}
double g(double x){
   return exp(-x);
}
double h(double x){
   return (x*x+6)/5;
```

```
}
// Вспомогательная функция для решения уравнений
// и тестирования найденных решений:
void test(double (*f)(double),double x0,string eq){
   // Количество итераций:
   int n=100;
   // Переменная для записи значения корня уравнения:
   double z;
   cout<<"Решение уравнения "<<eq<<":\t";
   // Решение уравнения:
   z=findRoot(f,x0,n);
   // Отображение результата:
   cout<<z<<endl;
   cout<<"Проверка найденного решения:\t";
   // Проверка решения:
   cout<<z<<" = "<<f(z)<<endl;
   for(int k=1;k<=50;k++){
      cout<<"-";
   }
   cout<<endl;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Примеры решения уравнений:
   test(f,0,"x=0.5cos(x)");
   test(g,0,"x=exp(-x)");
   test(h,1,"x=(x*x+6)/5");
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 9.1)

```
Решение уравнения x=0.5cos(x):  0.450184
Проверка найденного решения:    0.450184 = 0.450184
-----------------------------------------------

Решение уравнения x=exp(-x):    0.567143
Проверка найденного решения:    0.567143 = 0.567143
-----------------------------------------------

Решение уравнения x=(x*x+6)/5:  2
Проверка найденного решения:    2 = 2
-----------------------------------------------
```

В программе для решения уравнений описана функция `findRoot()`. Ее первый аргумент — указатель на функцию, определяющую правую часть решаемого уравнения. Это должна быть функция с одним `double`-аргументом, возвращающая `double`-результат. Второй аргумент функции `findRoot()` определяет начальное приближение для корня уравнения. Третий целочисленный аргумент задает количество выполняемых при поиске корня итераций.

В программе решаются три уравнения. Их правые части определяются с помощью функций `f()`, `g()` и `h()`.

Названия этих функций передаются первым аргументом при вызове функции `findRoot()`. Последняя в свою очередь вызывается в теле функции `test()`.

После вычисления корня уравнения выполняется проверка. Для этого в консольное окно выводится вычисленное значение для корня уравнения (значение x) и значение в соответствующей точке функции для правой части уравнения (значение $\varphi(x)$). В идеале, если найдено точное решение, оба значения должны совпадать.

НА ЗАМЕТКУ

Метод последовательных итераций позволяет найти решение далеко не для каждого уравнения. Здесь важно все: и какая функция в правой части уравнения, и какое начальное приближение выбрано для поиска корня.

Метод половинного деления

Метод половинного деления для поиска решения алгебраического уравнения вида $f(x) = 0$ подразумевает, что искомый корень локализован в интервале $a \le x \le b$, причем на границах интервала поиска корня функция $f(x)$ должна принимать значения разных знаков (что формально может быть записано в виде условия $f(a)f(b) < 0$).

Суть метода состоит в таких действиях. Вычисляется значение функции $f(x)$ в центральной точке интервала поиска решения, которая определяется как x = $(a + b)/2$. Интерес представляет знак функции в этой точке: в нее сдвигается одна из границ интервала поиска корня. А именно: если в центральной точке интервала значение функции положительно, то в центр смещается та граница, где функция положительна. Если в центральной точке функция отрицательна, то сдвигается та граница, где функция отрицательна.

В результате после каждой такой операции интервал поиска корня уравнения уменьшается ровно в два раза, и при этом функция $f(x)$ на границах нового интервала поиска корня принимает значения разных знаков. Задача, таким образом, сводится к предыдущей.

Пример программы, в которой поиск корня уравнения выполняется методом половинного деления, представлен в листинге 9.2.

Листинг 9.2. Решение уравнения методом половинного деления

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>
using namespace std;
// Функция для решения уравнения методом
// половинного деления:
double findRoot(double (*f)(double),double a,double b,double dx)
{
   // Переменная для записи значения корня:
   double x=(a+b)/2;
   // Вычисление корня:
```

```
while((b-a)/2>dx){
   // Если корень попал на левую границу:
   if(f(a)==0){
      return a;
   }
   // Если корень попал на правую границу:
   if(f(b)==0){
      return b;
   }
   // Если корень посредине интервала:
   if(f(x)==0){
      return x;
   }
   // Смещение границы в центр интервала:
   if(f(a)*f(x)>0){
      a=x;
      }
      else{
         b=x;
      }
      // Новое значение для центральной точки интервала:
      x=(a+b)/2;
   }
   // Результат функции:
   return x;
}
// Функции для определения решаемых уравнений:
double f(double x){
   return 0.5*cos(x)-x;
}
double g(double x){
   return exp(-x)-x;
```

```
}
double h(double x){
   return x*x-5*x+6;
}
// Вспомогательная функция для решения уравнений
// и тестирования найденных решений:
void test(double (*f)(double),double a,double b,string eq){
// Контролируемый код:
try{
   // Если на границах интервала функция уравнения
   // принимает значения одинаковых знаков:
   if(f(a)*f(b)>0){
      // Генерирование ошибки с текстовым объектом:
      throw "Указан неверный диапазон!";
   }
   // Точность вычисления корня:
   double dx=0.001;
   // Переменная для записи значения корня уравнения:
   double z;
   cout<<"Решение уравнения "<<eq<<":\t";
   // Решение уравнения:
   z=findRoot(f,a,b,dx);
   // Отображение результата:
   cout<<z<<endl;
   cout<<"Проверка найденного решения:\t";
      // Проверка решения:
      cout<<f(z)<<" = 0"<<endl;
}
// Обработка ошибки:
catch(char* e){
      // Сообщение о некорректном интервале
      // для поиска корня уравнения:
```

```
        cout<<e<<endl;
    }
    for(int k=1;k<=50;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Примеры решения уравнений:
    test(f,0,1,"0.5cos(x)-x=0");
    test(g,0,2,"exp(-x)-x");
    test(h,0,5,"x*x-5*x+6=0");
    test(h,0,2,"x*x-5*x+6=0");
    test(h,1,3,"x*x-5*x+6=0");
    test(h,2.5,4.5,"x*x-5*x+6=0");
    test(h,2.5,10,"x*x-5*x+6=0");
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже представлен результат выполнения программы:

Результат выполнения программы (из листинга 9.2)

```
Решение уравнения 0.5cos(x)-x=0:          0.450195
Проверка найденного решения:     -1.4247e-005 = 0
----------------------------------------------------
Решение уравнения exp(-x)-x:     0.567383
Проверка найденного решения:     -0.000375349 = 0
----------------------------------------------------
```

```
Указан неверный диапазон!
-----------------------------------------------
Решение уравнения x*x-5*x+6=0:  2
Проверка найденного решения:    0 = 0
-----------------------------------------------
Решение уравнения x*x-5*x+6=0:  3
Проверка найденного решения:    0 = 0
-----------------------------------------------
Решение уравнения x*x-5*x+6=0:  3
Проверка найденного решения:    0 = 0
-----------------------------------------------
Решение уравнения x*x-5*x+6=0:  3.00079
Проверка найденного решения:    0.000794087 = 0
-----------------------------------------------
```

Метод половинного деления реализован в программе через функцию `findRoot()`. У нее такие аргументы:

- указатель на функцию, определяющую решаемое уравнение (для уравнения вида $f(x) = 0$ имеется в виду функция $f(x)$);
- левая и правая границы интервала, на котором выполняется поиск корня уравнения;
- параметр, определяющий точность, с которой вычисляется корень.

НА ЗАМЕТКУ

Если корень уравнения локализован в интервале от *a* до *b*, то он отстоит (отличается) от центральной точки интервала на величину, не большую чем половина длины интервала. Данный критерий использован в функции `findRoot()` при вычислении корня уравнения. Вычисления продолжаются до тех пор, пока половина длины интервала локализации корня не станет меньше параметра, который передается последним аргументом функции `findRoot()` и определяет точность вычислений.

Алгоритм выполнения функции `findRoot()` следующий.

- Последовательно проверяется на предмет равенства нулю значение функции, переданной первым аргументом (функция уравнения)

на границах интервала поиска решения и его в центре. Если корень уравнения попадает в одну из указанных точек, соответствующее значение возвращается результатом функции `findRoot()`. Такая проверка выполняется для того, чтобы избежать ненужной итерационной процедуры в случае, если очередная пробная точка для решения уравнения случайно попала в точное значение для корня.

- Если значения функции уравнения на границах интервала и в его центре отличны от нуля, то одна из границ смещается в центр, и процесс повторяется.

Функция для решения уравнений `findRoot()` вызывается в функции `test()`. В последней перед началом процесса вычислений проверяется, корректно ли указан интервал для поиска корня. Если оказывается, что функция уравнения на границах интервала имеет значения одинаковых знаков, то генерируется ошибка (объектом ошибки служит текстовый литерал). При перехвате ошибки выводится сообщение, что указан некорректный интервал.

(i) НА ЗАМЕТКУ

Функции `f()`, `g()` и `h()` в предыдущем примере (листинг 9.1) определяли правые части для уравнения вида $x = \varphi(x)$. Здесь же решается уравнение вида $f(x) = 0$. Несложно понять, что если уравнение $x = \varphi(x)$ записать в виде $\varphi(x) - x = 0$, то имеем дело с функцией $f(x) = \varphi(x) - x$. Проще говоря, теперь функции для тех же решаемых уравнений определены по-другому.

В главной функции программы функция `test()` вызывается с разными аргументами. В частности, там реализуются ситуации, когда интервал указан некорректно, когда корень попадает в одну из граничных точек интервала или когда через несколько итераций центральная точка интервала попадает в точное решение для уравнения.

Метод касательных

Еще один итерационный метод решения уравнения, который называется методом касательных, подразумевает следующее.

- Для поиска корня уравнения $f(x) = 0$ задается начальное приближение x_0.

- В точке x_0 строится касательная к кривой, определяемой зависимостью $f(x)$.
- Точка пересечения касательной с координатной осью абсцисс служит новым приближением для корня уравнения x_1.
- В точке x_1 строится касательная, и ее пересечение с горизонтальной координатной осью определяет новое приближение для корня x_2, и так далее.

Описанная схема сводится к итерационной формуле $x_{n+1} = x_n - f(x_n)/f'(x_n)$ для вычисления нового приближения для корня уравнения x_{n+1} по известному предыдущему приближению x_n. Здесь через $f'(x)$ обозначена производная от функции уравнения $f(x)$. В листинге 9.3 вниманию читателя предлагается программа, в которой реализован процесс поиска корня уравнения методом касательных.

Листинг 9.3. Метод касательных

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функция, определяющая решаемое уравнение:
double f(double x){
   return 2*exp(-x)-1;
}
// Главная функция программы:
int main(){
   // Изменение кодировки консоли:
   system("chcp 1251>nul");
   // Количество итераций:
   int n=10;
   // Приращение аргумента для вычисления производной:
   double dx=0.00001;
   // Начальное приближение для корня уравнения:
   double x=0;
   // Вычисление корня уравнения:
```

```
for(int k=1;k<=n;k++){
   x=x-f(x)/((f(x+dx)-f(x))/dx);
}
// Отображение результата:
cout<<"Найдено такое решение:\t"<<x<<endl;
cout<<"Контрольное значение:\t"<<log(2)<<endl;
   // Задержка консольного окна:
   system("pause>nul");
   return 0;
}
```

Ниже показан результат вычисления программы:

Результат выполнения программы (из листинга 9.3)

```
Найдено такое решение:  0.693147
Контрольное значение:   0.693147
```

Программа совсем маленькая и простая. В ней решается уравнение, решением которого на самом деле является значение $x = \ln(2) \approx 0{,}693147$. Уравнение задается через функцию `f()`. Итерационный процесс реализуется непосредственно в главной функции программы. Там запускается оператор цикла, в котором выполняется всего `10` итераций (значение переменной `n`). Производная, значение которой нужно использовать в рекуррентной формуле для вычисления корня уравнения, рассчитывается в приближенном виде $f'(x) \approx (f(x + dx) - f(x))/dx$ как отношение приращения функции к приращению аргумента. Приращение для аргумента в программе задается переменной `dx`. В качестве начального приближения для корня уравнения использовано нулевое значение. В программе помимо вычисленного корня для сравнения приводится и «точное» значение для решения уравнения.

НА ЗАМЕТКУ

Как и в случае с методом последовательных итераций, сходимость метода касательных требует отдельного исследования. Другими словами, метод касательных применим далеко не каждому уравнению. Читатели, интересующиеся вопросом применимости этого и других методов, могут обратиться к специальной литературе.

Интерполяционный полином Лагранжа

Задача по построению интерполяционного полинома вкратце может быть сформулирована следующим образом. Имеется набор узловых точек $x_0, x_1, x_2, ..., x_n$ (всего $n + 1$ точка) и в этих точках заданы значения $y_0, y_1, y_2, ..., y_n$ некоторой функции. Необходимо построить полином степени n (степень полинома на единицу меньше количества точек), такой, чтобы в узловых точках $\{x_k\}$ он принимал значения $\{y_k\}$ (индекс $k = 0, 1, 2, ..., n$).

Решать задачу можно разными способами. Здесь мы рассмотрим метод построения интерполяционного полинома по схеме Лагранжа. В таком случае искомый полином представляется в виде $L_n(x) = y_0\varphi_0(x) + y_1\varphi_1(x) + y_2\varphi_2(x) + ... + y_n\varphi_n(x)$. Мы ввели обозначение для функций

$$\varphi_k(x) = \frac{(x-x_0)(x-x_1)...(x-x_{k-1})(x-x_{k+1})...(x-x_n)}{(x_k-x_0)(x_k-x_1)...(x_k-x_{k-1})(x_k-x_{k+1})...(x_k-x_n)}$$

(индекс $k = 0, 1, 2, ..., n$). Легко проверить, что $\varphi_k(x_m) = 0$ если $k \neq m$ и $\varphi_k(x_k) = 1$, поэтому условия $L_n(x_k) = y_k$, накладываемые на интерполяционный полином, удовлетворяются автоматически.

В программе из листинга 9.4 реализован описанный выше способ построения интерполяционного полинома на основе массива со значениями узловых точек и массива со значениями функции в узловых точках.

Листинг 9.4. Интерполяционный полином Лагранжа

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Вспомогательная функция:
double phi(int k,double z,double* x,int n){
   // Индексная переменная:
   int i;
   // Переменная для записи результата:
   double res=1;
   // Вычисление произведения:
   for(i=0;i<k;i++){
      res*=(z-x[i])/(x[k]-x[i]);
   }
```

```
    for(i=k+1;i<n;i++){
        res*=(z-x[i])/(x[k]-x[i]);
    }
    // Результат функции:
    return res;
}
// Полином Лагранжа:
double L(double z,double* x,double* y,int n){
    // Переменная для записи результата:
    double s=0;
    // Вычисление полиномиальной суммы:
    for(int k=0;k<n;k++){
        s+=y[k]*phi(k,z,x,n);
    }
    // Результат функции:
    return s;
}
// Функция для отображения "линии":
void line(int m){
    for(int k=1;k<=m;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Главная функция программы:
int main(){
// Изменение кодировки консоли:
system("chcp 1251>nul");
// Индексная переменная и длина "линии":
int k,m=20;
// Размер массивов:
const int n=5;
// Узловые точки:
```

```
    double x[n]={1,3,5,7,9};
    // Значения функции в узловых точках:
    double y[n]={0,2,-1,1,3};
    line(m);
    cout<<"x\t| L(x)\n";
    line(m);
    // Отображение узловых точек и значений
    // интерполяционного полинома в этих точках:
    for(k=0;k<n;k++){
       cout<<x[k]<<"\t| "<<L(x[k],x,y,n)<<endl;
    }
    // Сдвиг для аргумента:
    double dx=1;
    line(m);
    cout<<"x\t| L(x)\n";
    line(m);
    // Значения аргумента и значения полинома:
    for(k=0;k<n;k++){
       cout<<x[k]+dx<<"\t| "<<L(x[k]+dx,x,y,n)<<endl;
    }
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже показано, как выглядит результат выполнения программы:

Результат выполнения программы (из листинга 9.4)

```
--------------------
x       | L(x)
--------------------
1       | 0
3       | 2
5       | -1
```

```
7          | 1
9          | 3
---------------------
x          | L(x)
---------------------
2          | 2.83594
4          | 0.148438
6          | -0.664063
8          | 2.89844
10         | -1.66406
```

Для реализации функции $\varphi_k(x)$ в программе описана функция `phi()`. Первый аргумент функции соответствует индексу k. Второй аргумент функции `phi()` соответствует точке x, в которой вычисляется значение $\varphi_k(x)$. Третьим аргументом функции `phi()` передается имя массива, в который записаны значения узловых точек. Наконец, последний аргумент функции `phi()` определяет размер массива узловых точек.

Результатом функции возвращается произведение множителей вида $(x - x_i)/(x_k - x_i)$, среди которых отсутствует множитель с индексом $i = k$.

> **НА ЗАМЕТКУ**
>
> Константа `n` в программе, определяющая размер массивов с узловыми точками и значениями функции в узловых точках, на единицу меньше параметра n, который использовался в формулах, приводившихся для интерполяционного полинома.

Значение интерполяционного полинома вычисляется функцией `L()`. Аргументами функции передаются:

- точка, в которой следует вычислить значение полинома;
- массив с узловыми точками;
- массив со значениями функции в узловых точках;
- целочисленный аргумент, определяющий размер массивов.

В теле функции `L()` при вычислении полиномиальной суммы вызывается функция `phi()`.

В главной функции программы как иллюстрация вычисляются значения интерполяционного полинома в узловых точках и в точках, сдвинутых на единичное расстояние по отношению к узловым точкам.

Интерполяционный полином Ньютона

При создании интерполяционного полинома методом Ньютона для полинома используется такое выражение: $P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + ... + a_n(x - x_0)(x - x_1)...(x - x_{n-1})$. Задача фактически сводится к вычислению коэффициентов a_k (индекс $k = 0, 1, 2, ..., n$). Коэффициент определяются исходя из накладываемых на полином условий $P_n(x_k) = y_k$. В частности, из соотношения $P_n(x_0) = y_0$ следует, что $a_0 = y_0$. Из условия $P_n(x_1) = y_1$ получаем $a_1 = (y_1 - a_0)/(x_1 - x_0)$. Условие $P_n(x_2) = y_2$ дает $a_2 = (y_2 - a_0 - a_1(x_2 - x_0))/((x_2 - x_0)(x_2 - x_1))$ и так далее. Рекуррентное соотношение для вычисления коэффициентов имеет вид

$$a_k = \frac{y_k - a_0 - a_1(x_k - x_0) - a_2(x_k - x_0)(x_k - x_1) - ... - a_{k-1}(x_k - x_0)...(x_k - x_{k-2})}{(x_k - x_0)(x_k - x_1)...(x_k - x_{k-1})}.$$

Программа, в которой на основе массивов с узловыми точками и значениями функции в узловых точках вычисляется интерполяционный полином Ньютона, представлена в листинге 9.5.

Листинг 9.5. Интерполяционный полином Ньютона

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функция вычисления коэффициентов для полинома:
void findA(double* a,double* x,double* y,int n){
  // Переменная для записи произведения
  // разностей узловых точек:
  double q;
  // Значение первого коэффициента для полинома:
  a[0]=y[0];
  // Вычисление прочих коэффициентов для полинома:
  for(int k=1;k<n;k++){
```

```
        // Начальное значение для коэффициента:
        a[k]=y[k];
        // Начальное значение для произведения
        // разностей значений узловых точек:
        q=1;
        // Уточнение значения коэффициента и вычисление
        // произведения разностей значений узловых точек:
        for(int m=0;m<k;m++){
            a[k]-=a[m]*q;
            q*=(x[k]-x[m]);
        }
        // Окончательное значение коэффициента:
        a[k]/=q;
    }
}
// Функция для вычисления значения полинома Ньютона:
double P(double* a,double z,double* x,int n){
    // Переменная для записи результата функции:
    double s=0;
    // Локальная переменная для записи произведения
    // разностей значений аргумента и узловых точек:
    double q;
    // Вычисление полиномиальной суммы:
    for(int k=0;k<n;k++){
    // Начальное значение для произведения:
    q=1;
    // Вычисление произведения:
    for(int m=0;m<k;m++){
        q*=(z-x[m]);
    }
    s+=a[k]*q;
```

```
    }
    // Результат функции:
    return s;
}
// Функция для отображения "линии":
void line(int m){
    for(int k=1;k<=m;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Индексная переменная и длина "линии":
    int k,m=20;
    // Размер массивов:
    const int n=5;
    // Узловые точки:
    double x[n]={1,3,5,7,9};
    // Значения функции в узловых точках:
    double y[n]={0,2,-1,1,3};
    // Коэффициенты для полинома Ньютона:
    double a[n];
    // Вычисление значений для коэффициентов:
    findA(a,x,y,n);
    line(m);
    cout<<"x\t| P(x)\n";
    line(m);
    // Отображение узловых точек и значений
```

```
// интерполяционного полинома в этих точках:
for(k=0;k<n;k++){
   cout<<x[k]<<"\t| "<<P(a,x[k],x,n)<<endl;
}
// Сдвиг для аргумента:
double dx=1;
line(m);
cout<<"x\t| P(x)\n";
line(m);
// Значения аргумента и значения полинома:
for(k=0;k<n;k++){
   cout<<x[k]+dx<<"\t| "<<P(a,x[k]+dx,x,n)<<endl;
}
// Задержка консольного окна:
system("pause>nul");
   return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.5)

```
--------------------
x       | P(x)
--------------------
1       | 0
3       | 2
5       | -1
7       | 1
9       | 3
--------------------
x       | P(x)
--------------------
2       | 2.83594
```

```
4          | 0.148438
6          | -0.664063
8          | 2.89844
10         | -1.66406
```

Несложно заметить, что при выполнении данной программы получаем такой же результат (такие же значения полинома), как и в предыдущем примере. Ничего удивительного в этом нет, так как на самом деле в обоих случаях создается один и тот же полином, просто мы использовали разные алгоритмы его вычисления.

Что касается непосредственно программы, то мы использовали общий подход, который состоит в том, что на основе значений узловых точек и значений интерполируемой функции в этих узловых точках вычисляются коэффициенты для интерполяционного полинома. Коэффициенты записываются в массив, и затем он используется при вычислении значения интерполяционного полинома.

Коэффициент для полинома вычисляются при вызове функции `findA()`. Функция не возвращает результат, а аргументами ей передаются:

- имя массива, в который записываются значения коэффициентов интерполяционного полинома;
- массив со значениями узловых точек;
- массив со значениями интерполируемой функции в узловых точках;
- целочисленное значение, определяющее размер массивов.

В теле функции `findA()` с использованием итерационной процедуры вычисляются значения коэффициентов для полинома, и эти вычисленные значения записываются в массив, переданный первым аргументом функции.

Функция `P()` предназначена для вычисления значения интерполяционного полинома в точке. Точка (аргумент полинома), в которой вычисляется значение, передается вторым аргументом функции `P()`. Первый ее аргумент — массив с коэффициентами. Третий — массив со значениями узловых точек, а четвертый аргумент функции `P()` задает размеры используемых массивов. Передавать в функцию `P()` массив со значениями интерполируемой функции в узловых точках в данном случае не нужно, поскольку соответствующий массив уже был «учтен» при вычислении коэффициентов для интерполяционного полинома.

Вычисление интеграла методом Симпсона

Далее рассмотрим задачу о вычислении определенного интеграла вида $\int_a^b f(x)dx$. Существуют разные способы ее решения (в числовом виде). Метод Симпсона — один из наиболее простых методов вычисления интеграла. Кратко его суть состоит в том, что интервал, на котором вычисляется интеграл, разбивается на большое и, что важно, четное, количество интервалов равной длины. Обозначим количество интервалов, на которые разбивается интервал интегрирования, как $2m$ (тогда длина одного интервала равна $h = (b - a)/2m$). Узловые точки определяются как $x_k = a + kh$, где индекс $k = 0, 1, 2, ..., 2m$. Понятно, что $x_0 = a$ и $x_{2m} = b$ по определению. Обозначим также как $f_k = f(x_k)$ значения подынтегральной функции $f(x)$ в узловых точках. Далее на каждых двух смежных интервалах (то есть по трем соседним узловым точкам) строятся интерполяционные полиномы (второй степени). Преимущество подхода в том, что вместо функции $f(x)$ под интегралом оказываются относительно простые полиномиальные выражения, которые можно проинтегрировать в явном виде. В итоге для интеграла получаем такое приближенное выражение: $\int_a^b f(x)dx \approx h/3(f_0 + f_{2m} + 4\sum_{k=1}^{m} f_{2k-1} + 2\sum_{k=1}^{m-1} f_{2k})$. Данную формулу, записанную в виде $\int_a^b f(x)dx \approx h/3(f_0 + f_{2m} + 4f_{2m-1}) + (h/3)\sum_{k=1}^{m-1}(4f_{2k-1} + 2f_{2k})$, мы и используем в программе из листинга 9.6 для вычисления определенных интегралов.

Листинг 9.6. Вычисление интегралов методом Симпсона

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функция для вычисления интегралов методом Симпсона:
double integrate(double (*f)(double),double a,double b,int
m=1000){
   // Длина внутреннего интервала:
   double h=(b-a)/2/m;
   // Переменная для записи интегральной суммы:
   double s=0;
   // Вычисление интегральной суммы:
   for(int k=1;k<=m-1;k++){
```

```
        s+=4*f(a+(2*k-1)*h)+2*f(a+2*k*h);
    }
    s+=f(a)+f(b)+4*f(a+(2*m-1)*h);
    s*=h/3;
    // Результат функции:
    return s;
}
// Подынтегральные функции:
double F1(double x){
    return x*(1-x);
}
double F2(double x){
    double pi=3.141592;
    return pi/2*tan(pi*x/4);
}
double F3(double x){
    return exp(-x)*cos(x);
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    cout<<"Вычисление интегралов\n";
    cout<<integrate(F1,0,1)<<" vs. "<<(double)1/6<<endl;
    cout<<integrate(F2,0,1)<<" vs. "<<log(2)<<endl;
    cout<<integrate(F3,0,100,1e5)<<" vs. "<<0.5<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы такой:

Результат выполнения программы (из листинга 9.1)

```
Вычисление интегралов
0.166667 vs. 0.166667
0.693147 vs. 0.693147
0.5 vs. 0.5
```

В программе для вычисления интегралов описана функция `integrate()`. Первым аргументом ей передается указатель на функцию, определяющую подынтегральное выражение. Два следующих аргумента определяют границы области интегрирования. Четвертый необязательный аргумент (у него есть значение по умолчанию) определяет количество интервалов, на которые разбивается область интегрирования (количество таких интервалов в два раза больше значения четвертого аргумента).

В главной функции программы вычисляется три интеграла: интеграл $\int_0^1 x(1-x)dx = 1/6$, интеграл $(\pi/2)\int_0^1 \mathrm{tg}(\pi x/4)dx = \ln(2)$, а также интеграл $\int_0^\infty \exp(-x)\cos(x)dx = 1/2$. В последнем случае имеем дело с несобственным интегралом. На практике мы верхнюю бесконечную границу заменяем на большое, но конечное значение 100 (при этом функции `integrate()` в явном виде передается значение 10^5 для последнего аргумента). Подынтегральные функции, от которых вычисляются интегралы, задаются с помощью программных функций `F1()`, `F2()` и `F3()`. Для сравнения в программе наряду с вычисленным значением для интеграла отображается и контрольное значение. Несложно заметить, что результаты вычислений более чем приемлемые даже для несобственного интеграла (хотя в общем случае для вычисления несобственных интегралов используются специально разработанные для этого методы).

Вычисление интегралов методом Монте-Карло

Сразу следует отметить, что вычисление интегралов методом Монте-Карло не относится ни к точным, ни к быстрым, хотя алгоритм не очень сложный. Обычно к методу Монте-Карло прибегают, когда другие методы неприменимы.

Суть метода поясним на примере вычисления интеграла $\int_a^b f(x)dx$ от функции $f(x)$, которая на интервале интегрирования неотрицательна

и ограничена, так что имеет место соотношение $0 \le f(x) \le f_{max}$, где через f_{max} обозначена некоторая оценка для значения функции на интервале интегрирования. Если рассмотреть в координатной плоскости квадрат $a \le x \le b$ и $0 \le y \le f_{max}$, то площадь такого квадрата равна, очевидно, величине $(b - a)f_{max}$. Далее выбираем случайным образом точку внутри данного квадрата. Теория вероятностей утверждает, что вероятность P для случайно выбранной точки оказаться под кривой, определяемой функцией $f(x)$, равна отношению площади S области, ограниченной данной кривой, и площади прямоугольника, то есть $P = S/[(b - a)f_{max}]$. Тогда $S = (b - a)f_{max}P$. С другой стороны, площадь области под кривой и есть интеграл. Поэтому для вычисления интеграла достаточно вычислить вероятность P события, состоящего в том, что случайно выбранная в прямоугольнике точка окажется под кривой $f(x)$. Если умножить такую вероятность на площадь квадрата, получим искомый интеграл.

Для вычисления вероятности P можно последовательно генерировать случайные точки и проверять, попадают ли они под кривую. Отношение числа «внутренних» (попавших под кривую) точек к общему числу точек дает оценку для вероятности P. При этом важно, чтобы точек было много, а распределялись они по области квадрата равномерно. Здесь многое зависит от генератора случайных чисел.

На практике вместо генерирования случайных точек можно воспользоваться сеткой из равномерно распределенных точек. В остальном алгоритм остается неизменным: подсчитываются точки под кривой, вычисляется отношение их количества к общему количеству точек в сетке, результат умножается на площадь прямоугольника — так получаем значение интеграла. Программа с реализацией такого подхода представлена в листинге 9.7.

Листинг 9.7. Вычисление интегралов методом Монте-Карло

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функция для вычисления интегралов методом Монте-Карло:
double integrate(double (*f)(double),double a,double b,double Fmax){
   // Переменные для записи координат точки:
   double x,y;
```

```
// Количество интервалов по каждой из осей:
int m=10000;
// Приращение по первой координате:
double dx=(b-a)/m;
// Приращение по второй координате:
double dy=Fmax/m;
// Переменная для подсчета внутренних точек:
int count=0;
// Перебор точек на плоскости:
for(int i=0;i<=m;i++){
   for(int j=0;j<=m;j++){
      // Первая координата точки:
      x=a+i*dx;
      // Вторая координата точки:
      y=j*dy;
      // Проверка точки:
      if(y<=f(x)){
         // Если точка под графиком функции:
         count++;
      }
   }
}
// Доля внутренних точек:
double z=(double)count/(m+1)/(m+1);
// Результат функции:
return Fmax*(b-a)*z;
}
// Подынтегральные функции:
double F1(double x){
   return x*(1-x);
}
double F2(double x){
```

```
    double pi=3.141592;
    return pi/2*tan(pi*x/4);
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    cout<<"Вычисление интегралов\n";
    cout<<integrate(F1,0,1,0.25)<<" vs. "<<(double)1/6<<endl;
    cout<<integrate(F2,0,1,1.6)<<" vs. "<<log(2)<<endl;
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

У функции `integrate()` четыре аргумента: указатель на подынтегральную функцию, нижняя и верхняя границы для интервала интегрирования, а также числовая оценка для максимального значения подынтегральной функции на интервале интегрирования. Результат выполнения программы представлен ниже:

Результат выполнения программы (из листинга 9.7)

```
Вычисление интегралов
0.166646 vs. 0.166667
0.693167 vs. 0.693147
```

Хотя результат вычислений неплох, но на его получение уходит все же значительное время. Так что данный метод лучше приберечь для особых случаев.

Решение дифференциального уравнения методом Эйлера

Решим дифференциальное уравнение вида $y'(x) = f(x,y(x))$ с начальным условием $y(x_0) = y_0$ (в такой постановке речь идет о задаче Коши). Здесь функция двух аргументов $f(x,y)$ задана, через x обозначен незави-

симый аргумент, $y(x)$ является искомой неизвестной функцией, а через $y'(x)$ обозначена производная от функции $y(x)$. Задача состоит в том, чтобы найти такую функцию $y(x)$, которая бы удовлетворяла начальному условию и, будучи подставлена в уравнение $y'(x) = f(x,y(x))$, превращала бы его в тождество.

Наиболее простой (но не самый надежный в плане точности) метод числового решения дифференциального уравнения состоит в том, что значение функции $y(x)$ вычисляется в узловых точках $x_k = x_0 + kh$, где индекс $k = 0,1,2,...$, шаг приращения по аргументу h по возможности должен быть минимальным, а через x_0 обозначена точка, в которой задано начальное условие.

Решение задачи базируется на том, что для производной от функции $y(x)$ в узловой точке x_k используется приближенное выражение $y'(x_k) \approx (y(x_{k+1}) - y(x_k))/h$. В рамках такого подхода легко получаем рекуррентное соотношение $y(x_{k=1}) = y(x_k) + hf(y_k,y(x_k))$, позволяющее получить значение функции $y(x)$ в точке x_{k+1} если известно значение функции в узловой точке x_k. Отправной является точка x_0, в которой задано значение функции y_0.

В листинге 9.8 представлена программа, в которой методом Эйлера решается дифференциальное уравнение $y'(x) = x^2\exp(-x) - y(x)$ с начальным условием $y(0) = 1$. У этой задачи есть точное (аналитическое) решение $y(x) = (x^3/3 + 1)\exp(-x)$. В программе числовое решение сравнивается с аналитическим решением.

Листинг 9.8. Решение дифференциального уравнения методом Эйлера

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функция, определяющая дифференциальное уравнение:
double f(double x,double y){
   return x*x*exp(-x)-y;
}
// Функция для решения дифференциального
// уравнения методом Эйлера:
double dsolve(double (*f)(double,double),double x0,double y0,double x){
```

```
    // Количество интервалов:
    int n=1000;
    // Приращение по аргументу:
    double h=(x-x0)/n;
    // Начальное значение искомой функции:
    double y=y0;
    // Вычисление значения функции в точке:
    for(int k=0;k<n;k++){
        y=y+h*f(x0+k*h,y);
    }
    // Значение искомой функции в точке:
    return y;
}
// Функция, определяющая точное решение
// дифференциального уравнения:
double Y(double x){
    return (x*x*x/3+1)*exp(-x);
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Массив значений аргумента, для которых
    // вычисляется решение дифференциального уравнения:
    double x[]={0,0.5,1,3,10};
    cout<<"Решение дифференциального уравнения:\n";
    for(int k=0;k<5;k++){
        cout<<dsolve(f,0,1,x[k])<<" vs. "<<Y(x[k])<<endl;
    }
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Результат выполнения программы будет таким:

Результат выполнения программы (из листинга 9.8)

```
Решение дифференциального уравнения:
1 vs. 1
0.631701 vs. 0.631803
0.490245 vs. 0.490506
0.497815 vs. 0.497871
0.015116 vs. 0.0151787
```

Поиск решения дифференциального уравнения выполняется с помощью функции `dsolve()`. Аргументом функции передается указатель на функцию, определяющую дифференциальное уравнение, начальную точку по аргументу и значение искомой функции в начальной точке, а также значение аргумента, для которого вычисляется значение искомой функции.

НА ЗАМЕТКУ

В теле функции `dsolve()` диапазон от начальной точки `x0` до точки `x` разбивается на `n` интервалов. Поэтому расстояние между соседними узловыми точками изменяется с изменением аргумента `x`. Чем больше расстояние между узловыми точками, тем ниже точность вычислений.

Решение дифференциального уравнения методом Рунге — Кутты

По сравнению с методом Эйлера, метод Рунге — Кутты для решения дифференциальных уравнений имеет большую надежность. Так, в рамках методы Рунге — Кутты четвертого порядка значение искомой функции $y(x_{k+1})$ в узловой точке x_{k+1} вычисляется по формуле $y(x_{k+1}) = y(x_k) + (h/6)(p_1(h) + 2p_2(h) + 2p_3(h) + p_4(h))$, где введены такие обозначения: $p_1(h) = f(x_k,y(x_k))$, $p_2(h) = f(x_k + (h/2),y(x_k) + (h/2)p_1(h))$, $p_3(h) = f(x_k + (h/2),y(x_k) + (h/2)p_2(h))$ и $p_4(h) = f(x_k + h,y(x_k) + hp_3(h))$. Реализация данной схемы представлена в программном коде в листинге 9.9. Код выдержан в духе предыдущего примера, но только при вычислении решения использован более «изощренный» базовый алгоритм.

Листинг 9.9. Решение дифференциального уравнения методом Рунге — Кутты

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функция, определяющая дифференциальное уравнение:
double f(double x,double y){
   return x*x*exp(-x)-y;
}
// Функция для решения дифференциального
// уравнения методом Рунге - Кутты:
double dsolve(double (*f)(double,double),double x0,double
y0,double x){
   // Количество интервалов:
   int n=1000;
   // Переменные для записи промежуточных
  // значений при вычислениях:
   double p1,p2,p3,p4;
   // Приращение по аргументу:
   double h=(x-x0)/n;
   // Начальное значение искомой функции:
   double y=y0;
   // Вычисление значения функции в точке:
   for(int k=0;k<n;k++){
      p1=f(x0+k*h,y);
      p2=f(x0+k*h+h/2,y+h*p1/2);
      p3=f(x0+k*h+h/2,y+h*p2/2);
      p4=f(x0+k*h+h,y+h*p3);
      y=y+(h/6)*(p1+2*p2+2*p3+p4);
   }
   // Значение искомой функции в точке:
```

```
    return y;
}
// Функция, определяющая точное решение
// дифференциального уравнения:
double Y(double x){
    return (x*x*x/3+1)*exp(-x);
}
// Главная функция программы:
int main(){
    // Изменение кодировки консоли:
    system("chcp 1251>nul");
    // Массив значений аргумента, для которых
    // вычисляется решение дифференциального уравнения:
    double x[]={0,0.5,1,3,10};
    cout<<"Решение дифференциального уравнения:\n";
    for(int k=0;k<5;k++){
        cout<<dsolve(f,0,1,x[k])<<" vs. "<<Y(x[k])<<endl;
    }
    // Задержка консольного окна:
    system("pause>nul");
    return 0;
}
```

Ниже можно видеть результат выполнения программы:

Результат выполнения программы (из листинга 9.1)

```
Решение дифференциального уравнения:
1 vs. 1
0.631803 vs. 0.631803
0.490506 vs. 0.490506
0.497871 vs. 0.497871
0.0151787 vs. 0.0151787
```

Видим, что точность вычислений более чем приемлемая.

Оптовая торговля книгами «Эксмо»:
ООО «ТД «Эксмо». 123308, г. Москва, ул.Зорге, д.1, многоканальный тел.: 411-50-74.
E-mail: **reception@eksmo-sale.ru**

По вопросам приобретения книг «Эксмо» зарубежными оптовыми покупателями *обращаться в отдел зарубежных продаж ТД «Эксмо»*
E-mail: **international@eksmo-sale.ru**

International Sales: International wholesale customers should contact Foreign Sales Department of Trading House «Eksmo» for their orders.
international@eksmo-sale.ru

По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении, *обращаться по тел.: +7 (495) 411-68-59, доб. 2261.*
E-mail: **ivanova.ey@eksmo.ru**

Оптовая торговля бумажно-беловыми и канцелярскими товарами для школы и офиса «Канц-Эксмо»:
Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2, Белокаменное ш., д. 1, а/я 5. Тел.:/факс +7 (495) 745-28-87 (многоканальный).
e-mail: **kanc@eksmo-sale.ru**, сайт: www.**kanc-eksmo.ru**

В Санкт-Петербурге: в магазине «Парк Культуры и Чтения БУКВОЕД», Невский пр-т, д. 46.
Тел.: +7(812)601-0-601, **www.bookvoed.ru**

Полный ассортимент книг издательства «Эксмо» для оптовых покупателей:
Москва. ООО «Торговый Дом «Эксмо». Адрес: 123308, г. Москва, ул.Зорге, д.1.
Телефон: +7 (495) 411-50-74. **E-mail:** reception@eksmo-sale.ru
Нижний Новгород. Филиал «Торгового Дома «Эксмо» в Нижнем Новгороде. Адрес: 603094, г. Нижний Новгород, ул. Карпинского, д. 29, бизнес-парк «Грин Плаза».
Телефон: +7 (831) 216-15-91 (92, 93, 94). **E-mail:** reception@eksmonn.ru
Санкт-Петербург. ООО «СЗКО». Адрес: 192029, г. Санкт-Петербург, пр. Обуховской Обороны, д. 84, лит. «Е». Телефон: +7 (812) 365-46-03 / 04. **E-mail:** server@szko.ru
Екатеринбург. Филиал ООО «Издательство Эксмо» в г. Екатеринбурге. Адрес: 620024, г. Екатеринбург, ул. Новинская, д. 2щ. Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08).
E-mail: petrova.ea@ekat.eksmo.ru
Самара. Филиал ООО «Издательство «Эксмо» в г. Самаре.
Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е».
Телефон: +7(846)207-55-50. **E-mail:** RDC-samara@mail.ru
Ростов-на-Дону. Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону. Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, д. 44 А. Телефон: +7(863) 303-62-10. **E-mail:** info@rnd.eksmo.ru
Центр оптово-розничных продаж Cash&Carry в г. Ростове-на-Дону. Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, д. 44 В. Телефон: (863) 303-62-10.
Режим работы: с 9-00 до 19-00. **E-mail:** rostov.mag@rnd.eksmo.ru
Новосибирск. Филиал ООО «Издательство «Эксмо» в г. Новосибирске. Адрес: 630015, г. Новосибирск, Комбинатский пер., д. 3. Телефон: +7(383) 289-91-42. **E-mail:** eksmo-nsk@yandex.ru
Хабаровск. Обособленное подразделение в г. Хабаровске. Адрес: 680000, г. Хабаровск, пер. Дзержинского, д. 24, литера Б, офис 1. Телефон: +7(4212) 910-120. **E-mail:** eksmo-khv@mail.ru
Тюмень. Филиал ООО «Издательство «Эксмо» в г. Тюмени.
Центр оптово-розничных продаж Cash&Carry в г. Тюмени.
Адрес: 625022, г. Тюмень, ул. Алебашевская, д. 9А (ТЦ Перестройка+).
Телефон: +7 (3452) 21-53-96/ 97/ 98. **E-mail:** eksmo-tumen@mail.ru
Краснодар. ООО «Издательство «Эксмо» Обособленное подразделение в г. Краснодаре
Центр оптово-розничных продаж Cash&Carry в г. Краснодаре
Адрес: 350018, г. Краснодар, ул. Сормовская, д. 7, лит. «Г». Телефон: (861) 234-43-01(02).
Республика Беларусь. ООО «ЭКСМО АСТ Си энд Си». Центр оптово-розничных продаж Cash&Carry в г.Минске. Адрес: 220014, Республика Беларусь, г. Минск, пр-т Жукова, д. 44, пом. 1-17, ТЦ «Outleto». Телефон: +375 17 251-40-23; +375 44 581-81-92.
Режим работы: с 10-00 до 22-00. **E-mail:** exmoast@yandex.by
Казахстан. РДЦ Алматы. Адрес: 050039, г. Алматы, ул. Домбровского, д. 3 «А».
Телефон: +7 (727) 251-59-90 (91,92). **E-mail:** RDC-Almaty@eksmo.kz
Интернет-магазин: www.book24.kz
Украина. ООО «Форс Украина». Адрес: 04073 г. Киев, ул. Вербовая, д. 17а.
Телефон: +38 (044) 290-99-44. **E-mail:** sales@forsukraine.com

Полный ассортимент продукции Издательства «Эксмо» можно приобрести в книжных магазинах «Читай-город» и заказать в интернет-магазине www.chitai-gorod.ru.
Телефон единой справочной службы 8 (800) 444 8 444. Звонок по России бесплатный.

Интернет-магазин ООО «Издательство «Эксмо»
www.book24.ru
Розничная продажа книг с доставкой по всему миру.
Тел.: +7 (495) 745-89-14. E-mail: **imarket@eksmo-sale.ru**

BOOK24.RU
ИНТЕРНЕТ-МАГАЗИН
ИНТЕРНЕТ-МАГАЗИН
BOOK24.RU

EKSMO.RU
новинки издательства