Максим Кузнецов
Игорь Симдянов

bhv®

C++ мастер-класс в задачах и примерах

158 задач и готовых решений!

Тонкости C и C++

Объектно-ориентированное программирование

Библиотека STL

Библиотека ввода-вывода

On-line поддержка

+ CD-ROM

Максим Кузнецов
Игорь Симдянов

C++ мастер-класс
в задачах и примерах

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.06
ББК 32.973.26-018.2
К89

Кузнецов, М. В.

К89 С++. Мастер-класс в задачах и примерах / М. В. Кузнецов, И. В. Симдянов. — СПб.: БХВ-Петербург, 2007. — 480 с.: ил. + CD-ROM

ISBN 978-5-94157-953-2

Книга разбита на две основные части: задачи и решения. Рассматриваются базовые конструкции языка С++, тонкие моменты низкоуровневых операций, объектно-ориентированное программирование, разработка приложений при помощи стандартной библиотеки шаблонов STL, а также прикладные задачи. Особенностью предлагаемых задач и их решений является независимость от платформы и среды программирования, поэтому книга будет интересна как UNIX-, так и Windows-программистам. Компакт-диск содержит листинги всех готовых решений, представленных в книге.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Наталья Таркова*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.12.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 38,7.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

ISBN 978-5-94157-953-2

Оглавление

Введение

Язык C++ — на сегодняшний день один из самых сложных языков программирования. Тем не менее он де-факто представляет собой промышленный стандарт. Изучение является обязательным требованием для программиста, работающего с использованием нескольких языков. Профессиональный программист, не владеющий C++, вызывает такое же удивление, как полиглот, не владеющий английским языком. Любые методологические приемы, библиотеки в первую очередь рассматриваются на примере C++.

Однако C++ ведет свою историю от C, который изначально является системным языком. C++ — это своеобразное расширение C, привносящее в язык объектно-ориентированный подход, исключения, пространство имен и т. п. Одной из целей при создании C++ было сохранение скорости выполнения конечных программ. Эта цель была успешно достигнута, однако благодаря значительной части системности языка.

Рассмотрим, чем отличается системный язык от обычного языка высокого уровня. Изначально программы создавались при помощи машинных кодов, т. е. номеров инструкций процессора. Инструкции в свою очередь осуществляли различные операции вроде перемещения значения из одного регистра в другой. Путем размещения в регистрах значений и осуществления над ними операций (вызова операции по ее номеру), за несколько вызовов (тактов) можно было выполнить определенное действие — например, сложить два числа. Из простейших арифметических операций можно было построить небольшую программу, например, деления числа (в те далекие времена еще не было сопроцессоров, где операции с плавающей точкой реализованы на аппаратном уровне). Система номеров команд, мягко говоря, неудобна для повседневного использования. Так, для копирования содержимого регистра в другой регистр необходимо было использовать один номер, для копирования в третий регистр предназначен второй номер, для копирования содержимого в ячейку оперативной памяти — третий, причем номера команд и адреса яче-

ек необходимо было писать в двоичной форме (листинг В1), что значительно затрудняло отладку. Программисты со стажем вспоминают, что приходилось помнить до 400 основных кодов команд. Самое страшное начиналось, когда нужно было переходить на процессор новой архитектуры — необходимо было как-то забыть старые команды и изучать новые, при этом не путать их. Это было достаточно сложно, т. к. и там, и там были цифры.

Листинг В1. Машинный код (шестнадцатеричный формат)

```
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
...
73 63 72 69 62 65 64 20 69 6E 20 52 46 43 20 31
33 32 31 2E 20 20 57 68 65 6E 20 53 68 65 63 6B
```

В связи с этим практически повсеместно стал вводиться язык Assembler — команды обозначались буквенными сокращениями и назывались *мнемониками*. Одна мнемоника обозначает сразу несколько машинных кодов, например перемещение значения в регистры, ячейку в Assembler можно при помощи мнемоники `MOV` (от англ. *move* — перемещать). С введением Assembler жизнь программистов значительно облегчилась, даже при смене процессора, неизбежно приводившего к изменению Assembler, основной набор команд оставался прежним. Однако архитектура процессора значительно влияла на программы, программист был вынужден помнить, в какие регистры следует поместить данные, чтобы осуществить сложение, а в какие лучше не надо. Кроме того, процессоры не сопровождались математическим сопроцессором, и алгоритмы численного вычисления синуса угла (реализованные в любом языке высокого уровня в виде стандартной функции) составляли предмет многочисленных диссертаций. Главным же недостатком Assembler оставалась значительная привязка к архитектуре процессора.

Одновременно с Assembler стали появляться языки высокого уровня — идея их заключалась в том, чтобы создать язык, который бы не зависел от архитектуры процессора. Программа, созданная на одной машине и операционной системе, будучи откомпилированной на машине с другой архитектурой и операционной системой, вела бы себя точно так же. Так как различных архитектур немного, а программистов несколько миллионов, проще для каждой из архитектур создать свой собственный компилятор, который сможет адаптировать любую программу, написанную на языке высокого уровня. Введение языков высокого уровня позволило программистам не вникать в особенности

архитектуры компьютера и не выполнять рутинные системные задачи, реализацию которых взял на себя компилятор. Теперь программист оперировал не низкоуровневыми аппаратными понятиями, такими как регистр, ячейка памяти, адрес, а более высокими абстрактными конструкциями — переменная, массив, строка, файл, функция, процедура. Однако для того чтобы имелась возможность решать системные задачи (писать компиляторы, операционные системы), ряд языков в своем синтаксисе оставили системные возможности, указатели, ссылки, битовые поля и т. п. К таким языкам относятся C и C++. Именно поэтому, когда начинающий программист, только приступивший к изучению этих языков, встречает указатель на функцию, который мало того что передается в другую функцию, но и умудряется еще и вызывать ее, это порождает слабый протест и ощущение уходящей из-под ног почвы. Именно поэтому программисты, знакомые с Assembler, изучают C/C++ гораздо быстрее, чем программисты, знающие языки высокого уровня. Знатокам Assembler проще свыкнуться с мыслью, что в C/C++ никаких строк и массивов нет, а их реализация — лишь бутафория, призванная упростить процесс создания программы человеком.

Таким образом, сложность языка C/C++ основывается на том, что язык является системным, и помимо того, что он является языком высокого уровня, его также можно назвать языком низкого уровня, позволяющим разрабатывать системные программы. Однако это еще не все. Рассмотренные системные особенности языка скорее относятся к букве C, но свои два плюса язык C++ получил благодаря дополнению C объектно-ориентированной моделью.

Переход от структурного программирования к объектно-ориентированному (ООП) связан, в первую очередь, с возрастающей сложностью создаваемого программного обеспечения. В первое время программы не превышали нескольких сотен строк. Увеличение количества кода в программах до нескольких тысяч строк привело к внедрению приемов структурного программирования (появились функции), что позволило создавать и сопровождать программы размерами до ста тысяч строк. Стремительное развитие программного обеспечения потребовало создания и сопровождения еще большего объема кода. Ответом на это было создание объектно-ориентированной технологии. Применение данной технологии дает возможность создавать еще большие по объему приложения и позволяет программисту оперировать при создании кода объектами реального мира, а не понятиями программы (переменные, функции и т. п.). ООП позволяет создавать программу на более высоком абстрактном уровне, объединяя данные и методы в классы, и является прототипом языка нового поколения. Таким образом, C++ является языком высокого уровня, имеет системные средства и обладает объектно-ориентированной технологией, занимая три уровня на шкале увеличения уровня абстрактности языков программирования (рис. В1).

Рис. В1. Шкала увеличения абстрактности языков программирования

Однако C++ — это не только основной язык, это также большое число библиотек, которые были написаны за время его существования. Сам по себе язык программирования способен не на много, сила языка C/C++ в огромном числе созданного программного обеспечения, которое можно использовать повторно. В результате сложнейшие математические задачи решаются в несколько строк. Поэтому процесс обучения языку C++ разделяется на два этапа: это изучение базовых конструкций языка и изучение библиотек. Конструкции языка играют роль грамматики — они позволяют участникам процесса программирования (программистам и компьютерам) понимать друг друга. Человек может до тонкостей знать грамматику языка, но если в жизни он прочитал лишь учебник грамматики, он вряд ли сможет написать книгу, которая заинтересует читателей. Поэтому изучение языка не ограничивается ознакомлением с базовыми основами языка, любой опытный программист должен знать библиотеки, которые позволят ему разрабатывать более функциональные программы за более короткие сроки. Библиотеки могут быть совершенно различными, от управления ресурсами операционной системы (API Windows, системные вызовы UNIX) и создания интерфейса (MFC, VCL, DirectX, OpenGL, Qt и т. д.) до математических библиотек символьной алгебры или разбора регулярных выражений.

Среди этих библиотек выделяется стандартная библиотека, которая должна быть реализована в любом компиляторе, именно она позволяет осуществлять ввод/вывод, а также реализовать конструкции, отсутствующие в языке. Одним из элементов стандартной библиотеки является стандартная библиотека шаблонов STL (Standard Template Library), которая позволяет работать с произвольными типами. По словам Н. Джосьютиса, автора книги C++ "Стан-

дартная библиотека"[1]: "Концепция STL в известном смысле противоречит исходным принципам объектно-ориентированного программирования: STL отделяет друг от друга данные и алгоритмы, вместо того чтобы объединять их".

Изучение базовых основ языка C/C++ (грамматики) строго обязательно для любого человека, претендующего на знание языка. Изучение библиотек (чтение литературы, если проводить аналогию с изучением разговорных языков) — дело индивидуальное и зависит от интересов программиста и решаемых им задач.

Замечание

Человек, интересующийся математикой, с большим интересом и эффективностью будет читать "Топологию", чем классическую литературу XIX века. Хотя для чтения книг изучать базовые основы языка все равно придется.

Поэтому в данном задачнике мы сосредоточимся в первую очередь на базовых конструкциях языка, которые актуальны для любой операционной системы, будь то UNIX или Windows, и любого компилятора. Книга заостряет внимание читателя на тонких моментах C++, заставляя глубже копнуть язык и разобраться с особенностями, которые на первый взгляд кажутся странными, но на самом деле открывают новые возможности языка и позволяют решать более широкий круг задач.

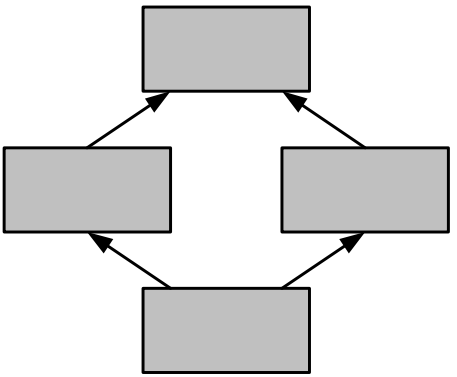
Изучение той или иной библиотеки не будет входить в круг наших задач, т. к., во-первых, почти по каждой из библиотек следует писать отдельную книгу и отдельный задачник, а во-вторых, в силу широкого распространения C++-библиотеки (да и C++-программисты тоже) слишком далеки друг от друга. Windows-программисты будут скучать при обсуждении системных вызовов UNIX, а UNIX-программистов будет только раздражать подробное описание особенностей DirectX.

Для кого предназначена книга?

Книга предназначена для читателей, которые хотят выяснить, владеют ли они языком C++ или нет. C++ интересен огромным числом библиотек, позволяющих создавать удивительные вещи: программное обеспечение для управления хитроумными приборами и роботами, игры, утилиты, языки программирования и все, с чем только можно столкнуться в мире компьютеров. Такая гибкость и мощь привлекает большое число программистов, которые стремятся побыстрее освоить ту или иную библиотеку, интерфейс или создать

[1] Джосьютис Н. C++ Стандартная библиотека. Для профессионалов. — СПб.: Питер, 2004. — 730 с.

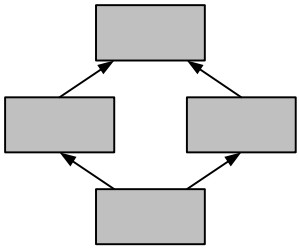
что-то свое. Если программист не обладает глубокими знаниями базовых основ языка C++, то взаимодействие с библиотекой и сам процесс создания программного обеспечения превращается в муку. Книга позволяет ответить на вопрос, готов программист к штурму сторонних библиотек или нет: если задания не вызывают трудностей или вы можете предложить более эффективные решения, чем представлены в ответах — вы знаете C++, если более 20% заданий вас ставит в тупик, то для успешного программирования вам следует еще раз проштудировать пособия по C++. Ничего страшного в том, что вы читаете книгу по программированию 5—6 раз для того, чтобы понять все тонкости языка, нет. Плотность информации, особенно в книгах по C++, очень велика, и усвоить все с первого раза, особенно, если C++ — первый язык, очень сложно. Поэтому стоит возвращаться к описанию языка снова и снова. Иногда возникает ощущение, что наступило понимание всех тонкостей языка — данная книга позволит вам проверить истинное ли это ощущение или ложное.



ЧАСТЬ I

Задачи по основам языка

ГЛАВА I.1



Базовые конструкции языка

В основе любого языка лежат типы, операторы и ключевые слова, которые позволяют формировать составные операторы. Конструкции языка появились не просто так, а в результате длительной эволюции и бурного обсуждения сообществом программистов. Если вам кажется, что какая-то конструкция языка лишняя, вы, скорее всего, ошибаетесь. Все лишние конструкции на сегодняшний день исключены из языка, и присутствие в языке конструкции говорит о том, что она необходима для элегантного решения ряда задач.

Современное поколение программистов практически не застало конструкций и операторов, использование которых приводит к ухудшению качества и устойчивости программ. Почти все такие конструкции были исключены из ранних языков программирования высокого уровня. Например, в ранних вариантах языка FORTRAN был не только множественный выход из функции при помощи оператора `return`, но и множественный вход, т. е. можно было задать несколько точек в функции, и ее выполнение могло осуществляться с начала, с середины или с конца функции, в зависимости от формы вызова. Такие функции, возможно, и позволяли сэкономить пару тактов процессора, но их сопровождение требовало значительных усилий программиста.

Когда компьютеры стоили "миллионы долларов", время работы программиста не имело значения — стоимость работы программиста была каплей в море. С удешевлением компьютеров ситуация резко поменялась, львиную долю бюджета стало составлять не компьютерное время, а время работы программиста. Чем проще ему сопровождать программу, чем быстрее он выполнит свою работу, тем эффективнее протекает сам процесс разработки программного обеспечения. Поэтому в современных программах ценится, в первую очередь, их читабельность и лишь затем скорость выполнения и потребления памяти.

Замечание

Особняком в требованиях стоит отсутствие ошибок, которое ценится всегда. Однако согласно последним исследованиям плотность ошибок не зависит от характера программного обеспечения или языка программирования, а зависит лишь от квалификации программиста. Это означает, что чем короче будет код, тем меньше ошибок он будет содержать. Короткий код, как правило, является более читабельным — именно к нему следует стремиться.

В данной главе мы рассмотрим задачи, предусматривающие использование простых конструкций, таких как операторы цикла, ветвления, приведения типов, логические и битовые операторы.

Замечание

Решения задач данной главы можно найти в каталоге code\1 компакт-диска, прикладываемого к книге.

I.1.1. Включение заголовочных файлов

Три представленные в листинге I.1.1 записи являются правильными и позволяют включить заголовочный файл библиотеки ввода/вывода.

Листинг I.1.1. Три варианта подключения стандартной библиотеки `iostream`

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

Назовите наиболее предпочитаемый в случае библиотеки `iostream` вариант подключения и объясните, для каких целей применяется каждый из случаев.

I.1.2. Сколько байтов занимает каждый из базовых типов?

Определите, сколько байтов отводится под каждый из базовых типов:

- `char`;
- `wchar_t`;
- `unsigned char`;
- `signed char`;
- `int`;
- `unsigned int`;
- `signed int`;
- `unsigned short int`;
- `signed short int`;
- `long int`;
- `signed long int`;
- `unsigned long int`;
- `float`;
- `double`;
- `logn double`;
- `bool`.

I.1.3. Сколько байтов занимает тип *void*?

При определении размера типа `void` компилятор сообщает об ошибке. Объясните, почему это происходит, и сколько байтов отводится под тип `void`?

I.1.4. Равны ли числа?

В листинге I.1.2 между собой сравниваются два числа, каков будет результат сравнения? Числа равны между собой или нет?

Замечание

Предполагается, что выполнение программы производится на компьютере с 32-битной архитектурой.

Листинг I.1.2. Сравнение двух чисел

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

int main()
{
  unsigned short int first = 60000;
  signed short int second = 60000;
  if(second == first) cout << "Числа эквивалентны\n";
  else cout << "Числа не эквивалентны\n";

  return 0;
}
```

I.1.5. Результат сравнения?

В листинге I.1.3 между собой сравниваются два числа: 60 000 и –5536. Разумеется, программа сообщает, что они не равны. Что требуется сделать, чтобы программа посчитала эти две переменных равными, при условии, что изменять значения самих переменных нельзя (т. е. не допускается прибавлять, удалять, делить, умножать и выполнять другие математические преобразования с переменными).

Замечание

Предполагается, что выполнение программы производится на компьютере с 32-битной архитектурой.

Листинг I.1.3. Сравнение двух чисел

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

int main()
{
  unsigned short int first = 60000;
  signed short int second = -5536;
  if(second == first) cout << "Числа эквивалентны\n";
  else cout << "Числа не эквивалентны\n";

  return 0;
}
```

I.1.6. Сравнение инкремента и постинкремента

Каков результат выполнения программы, представленной в листинге I.1.4? Объясните, почему так происходит?

Замечание

Залогом успешного программирования на языке C++ (или на любом другом языке) является автоматическое применение базовых конструкций. Хороший программист заранее видит и знает результат операций, что позволяет ему без усилий создавать свои программы и читать чужие. Если базовые навыки не отработаны, то программирование превращается в эксперимент методом тыка, а сам программист напоминает музыканта, пренебрегавшего гаммами, каждая попытка сыграть какое-либо произведение приводит лишь к тому, что на четвертой ноте у него запутываются пальцы.

Листинг I.1.4. Сравнение инкремента и постинкремента

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

int main()
{
  int i = 10;

  if(++i == i++) cout << "числа равны\n";
  else cout << "числа не равны\n";

  if(i++ == ++i) cout << "числа равны\n";
  else cout << "числа не равны\n";
```

```
  if(++i == ++i) cout << "числа равны\n";
  else cout << "числа не равны\n";

  if(i++ == i++) cout << "числа равны\n";
  else cout << "числа не равны\n";

  if(i++ == --i) cout << "числа равны\n";
  else cout << "числа не равны\n";

  if(i++ == i--) cout << "числа равны\n";
  else cout << "числа не равны\n";

  if(i-- == ++i) cout << "числа равны\n";
  else cout << "числа не равны\n";

  if(i-- == i++) cout << "числа равны\n";
  else cout << "числа не равны\n";
}
```

I.1.7. Четное или нечетное?

Создайте программу, которая запрашивает целое число у пользователя и определяет, является оно четным или нет.

I.1.8. Имя программы

Создайте программу, которая выводит свое собственное имя.

I.1.9. Чем отличается *switch* от конструкции *if-else-if*?

В листингах I.1.5 и I.1.6 представлены две программы, которые в зависимости от введенного пользователем числа (от 1 до 5) выводят английские названия цифр. В листинге I.1.5 для этого используется оператор `switch`, а в листинге I.1.6 конструкция `if-else-if`. Результат работы программ идентичен. Чем отличается оператор `switch` от конструкции `if-else-if` или это одно и то же? Если операторы различны, то когда оправдано применение `switch`, а когда `if-else-if`?

Листинг I.1.5. Использование оператора `switch`

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

```
int main()
{
  int number = 0;

  cout << "Input the number ";
  cin >> number;

  switch(number)
  {
    case 1:
      cout << "\nfirst\n";
      break;
    case 2:
      cout << "\nsecond\n";
      break;
    case 3:
      cout << "\nthird\n";
      break;
    case 4:
      cout << "\nfourth\n";
      break;
    case 5:
      cout << "\nfifth\n";
      break;
    default :
      cout << "\nInput a number between 1 and 5\n";
  }

  return 0;
}
```

Листинг I.1.6. Использование оператора `if-else-if`

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

int main()
{
  int number = 0;

  cout << "Введите число ";
  cin >> number;
```

```
  if(number == 1) cout << "\nfirst\n";
  else if(number == 2) cout << "\nsecond\n";
  else if(number == 3) cout << "\nthird\n";
  else if(number == 4) cout << "\nfourth\n";
  else if(number == 5) cout << "\nfifth\n";
  else cout << "\Введите число между 1 и 5\n";

  return 0;
}
```

I.1.10. Вывод случайного числа символов

Создайте программу, которая выводит случайное число символов * от 0 до 10.

I.1.11. Вывод четных чисел

Создайте программу, которая спрашивает у пользователя число и выводит в цикле четные числа от 0 до заданного числа.

I.1.12. Вывод всех видимых ASCII-символов

Создайте цикл, выводящий все ASCII-символы, начиная с 32-го символа, с которого начинаются видимые символы.

I.1.13. Поиск простых чисел

Создайте скрипт, который запрашивает у пользователя число и ищет простые числа до введенной пользователем величины.

Замечание

Простыми называются числа, которые делятся только на 1 и на самих себя.

I.1.14. Упаковка цикла *for*

Пусть имеется программа, которая в цикле `for` увеличивает значение одной переменной `i` и уменьшает значение другой переменной `j` до тех пор, пока их величины не сравняются (листинг I.1.7).

Листинг I.1.7. Цикл `for`

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

```
int main()
{
  int number = 10;
  int j = number;

  for(int i = 0; i <= j; i++)
  {
    cout << i << " " << j << "\n";
    j--;
  }
  return 0;
}
```

Результат работы программы выглядит следующим образом:

```
0 10
1 9
2 8
3 7
4 6
5 5
```

Перепишите листинг I.1.7 таким образом, чтобы у цикла `for` отсутствовало тело цикла (листинг I.1.8).

Листинг I.1.8. Результат упаковки цикла `for`

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

int main()
{
  int number, j, i;
  for(???);

  return 0;
}
```

Вместо знаков вопроса в листинге I.1.8 подставьте решение таким образом, чтобы программа была идентична по результату программе из листинга I.1.9.

I.1.15. Преобразование десятичного числа в двоичное

Создайте программу, которая запрашивает у пользователя целое число и выводит его в двоичном представлении.

I.1.16. Преобразование двоичного числа в десятичное

Создайте программу, которая запрашивает у пользователя двоичное число и выводит его в десятичном представлении.

I.1.17. Преобразование десятичного числа в восьмеричное

Создайте программу, которая запрашивает у пользователя целое число и выводит его в восьмеричном представлении.

I.1.18. Преобразование восьмеричного числа в десятичное

Создайте программу, которая запрашивает у пользователя восьмеричное число и выводит его в десятичном представлении.

I.1.19. Преобразование десятичного числа в шестнадцатеричное

Создайте программу, которая запрашивает у пользователя десятичное число и выводит его в шестнадцатеричном представлении.

I.1.20. Преобразование шестнадцатеричного числа в десятичное

Создайте программу, которая запрашивает у пользователя шестнадцатеричное число и выводит его в десятичном представлении.

I.1.21. Исключающее ИЛИ

В C и C++ существуют три логических оператора: `&&` (И), `||` (ИЛИ) и `!` (НЕ), логика использования которых представлена в табл. I.1.1. В этой же таблице описывается логика оператора исключающего ИЛИ — `XOR`, который не пред-

ставлен в языке C++. Реализуйте логику оператора исключающего ИЛИ при помощи других логических операторов (`&&`, `||` и `!`).

Таблица I.1.1. *Логические операторы*

x	y	x && y	x || y	!x	x XOR y
false	false	false	false	true	true
false	true	false	true	true	false
true	true	true	true	false	false
true	false	false	true	false	true

I.1.22. Возведение числа в степень

Создайте программу, которая запрашивает у пользователя два целых числа и возводит одно из них в степень другого числа, не используя библиотечную функцию `pow()`.

I.1.23. Смена знака числа

Используя только побитовые операторы, измените знак числа, который вводит пользователь. Если пользователь вводит отрицательное число, измените его на положительное, если положительное — выведите отрицательное число.

I.1.24. Изменение регистра строки

Используя лишь поразрядные операторы, измените регистр каждого символа строки, введенной пользователем на строчный (на прописной). Считается, что пользователь вводит только символы английского алфавита.

I.1.25. Глобальные переменные

В листинге I.1.9 используется глобальная переменная `var`, однако программа содержит ошибку — найдите ее.

Листинг I.1.9. Использование глобальной переменной

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

extern int var = 10;
```

```
int main()
{
  cout << var << endl;
  return 0;
}

int var = 10;
```

I.1.26. Статическая глобальная переменная

В листинге I.1.10 объявлены две глобальных переменных: `var` и `var_static`, причем переменная `var_static` объявлена как статическая. Объясните, чем отличаются друг от друга эти две переменные?

Листинг I.1.10. Статическая глобальная переменная

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

int var = 10;
static int var_static = 20;

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

I.1.27. Оператор "запятая"

Какие значения получат переменные `var`, `var1` и `var2` после выполнения операторов из листингов I.1.11 и I.1.12.

Листинг I.1.11. Использование оператора "запятая"

```
int main()
{
  int var, var1, var2;

  var = (var1 = 10, var2 = 11, var1 + 1);

  return 0;
}
```

Листинг I.1.12. Альтернативное использование оператора "запятая"

```
int main()
{
  int var, var1, var2;

  var = var1 = 10, var2 = 11, var1 + 1;

  return 0;
}
```

I.1.28. Использование структур и перечислений

Объявите структуру `forum_user`, состоящую из двух полей: имени пользователя и его статуса, который может принимать три значения: `administrator`, `moderator`, `user`, обозначающих администратора, модератора и пользователя соответственно. Создайте программу, которая позволяет заполнить структуру пользователю и выводит ее содержимое в стандартный поток.

I.1.29. Объединение и битовые поля

Создайте объединение, содержащее переменную `var` типа `char` и битовое поле, каждый бит в котором соответствует биту в переменной `var`. При помощи данной структуры выведите битовое представление числа 113.

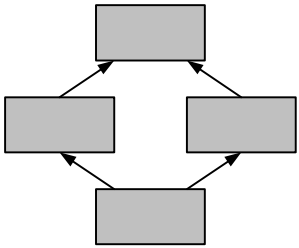
I.1.30. Преобразование арабского числа в римское

Создайте программу, которая будет преобразовывать число в арабской нотации (от 1 до 2000) в римское.

Замечание

Арабские числа соотносятся с римскими следующим образом: 1 — I, 5 — V, 10 — X, 50 — L, 100 — C, 500 — D, 1000 — M. Например, 116 = CXVI, 199 = CXCIX, 14 = XIV.

ГЛАВА I.2



Указатели, ссылки, массивы, строки

В C/C++ нет специального строкового типа, строка является массивом символов, последний символ строки в котором является нулевым `\0`. Сами массивы тоже значительно отличаются от массивов в других языка высокого уровня — в них не проверяется выход за границу массива. Это значительно повышает эффективность и скорость программ, т. к. проверка выхода за границу массива потребляет значительные ресурсы. Однако отсутствие контроля выхода за границы строки или массива требует внимания от программиста и служит причиной многочисленных ошибок.

Такое отличие от других языков программирования, где строки являются строками, а массивы — массивами, обусловлено низкоуровневой составляющей языка C++, доставшейся ему от C. В C допускается использование указателей — переменных, которые содержат адрес другой переменной. Под указатели может быть выделена память, а массивы и строки на поверку являются специально оформленными указателями. Поэтому работа с ними протекает очень быстро, но потенциально опасна хитрыми ошибками.

За годы использования C/C++ было разработано бессчетное количество библиотек эмулирующих строки и массивы, контролирующие выход за границы массива, самой известной реализации, STL, посвящена *глава 6*. Тем не менее возможности стандартных строк и массивов и тем более указателей следует осознавать очень четко. Это не тот раздел языка, который можно изучить поверхностно, т. к. объектно-ориентированное программирование, шаблоны, STL, исключения и другие высокоуровневые пласты языка ориентированы на твердое представление об указателях, ссылках и управлении памятью. Большинство неудач в изучении C++ связано с тем, что указателям уделялось недостаточное внимание.

Замечание

Решения задач данной главы можно найти в каталоге code\2 компакт-диска, поставляемого вместе с книгой.

I.2.1. Укоротить строку

Пусть имеется программа, в которой объявляется строка "Hello, world!" (листинг I.2.1). Не прибегая ни к каким библиотечным функциям, превратите строку "Hello, world!" в "Hello".

Листинг I.2.1. Объявление строки

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

int main()
{
  char str[] = "Hello, world!";
  // ?????????
  cout << str << endl;

  return 0;
}
```

Что должно быть вместо знаков вопроса в листинге I.2.1, чтобы программа вывела строку "Hello"?

I.2.2. Объявление строки

Что содержат переменные `str1`, `str2`, `str3`, `str4`, `str5` и `str6` в листинге I.2.2? Эквивалентны ли эти переменные?

Листинг I.2.2. Объявление строк

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

int main()
{
  char *str1   = "Hello";
  char str2[]  = "Hello";
  char str3[6] = "Hello";
  char *str4;
  str4 = "Hello";
```

```
  char *str5;
  str5 = str4;
  char *str6;
  str6 = str1;

  return 0;
}
```

I.2.3. Размер строки

Не прибегая к библиотечным функциям, определите размер строки, которую введет пользователь.

I.2.4. Количество элементов массива

В листинге I.2.3 объявлен целочисленный массив. Выведите его содержимое в стандартный поток вывода при помощи цикла `for`. Каким образом можно узнать количество элементов массива?

Листинг I.2.3. Объявление массива

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

int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  return 0;
}
```

I.2.5. Увеличение размера строки

Пусть имеется программа, которая объявляет строку, содержащую число 15 в бинарном представлении (листинг I.2.4).

Листинг I.2.4. Объявление строки с бинарным числом

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

int main()
{
  char *binary = "1111";
```

```
  cout << binary << endl;

  return 0;
}
```

Создайте программу, которая запрашивает у пользователя целое число, и, если оно не меньше 5 и не больше 81, программа должна создать новую строку длиной 81 символ (с учетом нулевого символа), в начало которой следует поместить значение из строки `binary`, а оставшиеся символы строки заполнить нулями. Полученный результат следует вывести в стандартный поток вывода.

Замечание

Обратите внимание на тот факт, что требуется не просто вывод результата в окно браузера, а создание переменной, способной хранить 80 символов плюс символ-признак конца строки `\0`.

I.2.6. Чередование символов строки и пробелов

Создайте программу, которая запрашивает у пользователя строку, создает переменную, в которую переносится содержимое введенной строки, однако каждый символ строки отделяется от соседнего пробелом.

I.2.7. Сравнение строк

Не прибегая к функциям сравнения строк, создайте программу, которая запрашивает у пользователя две строки и производит их сравнение, возвращая 0, если строки равны, +1, если первая строка больше второй, и –1 — в противном случае.

I.2.8. Упаковка IP-адреса

Создайте программу, которая, принимая в качестве параметра IP-адрес, возвращала бы его в виде целого числа. И наоборот, принимая целое число, возвращала бы IP-адрес. Реализуйте интерфейс программы через параметры, т. е. для упаковки IP-адреса пусть используется параметр `-p`:

```
prog -p 62.100.3.1
```

А для распаковки целого числа задается параметр `-u`:

```
prog -u 1046741761
```

Замечание

IP-адрес используется в сетях TCP/IP для идентификации узлов сети и представляет собой совокупность четырех чисел, разделенных точками. При этом каждое из чисел может принимать значения от 0 до 255.

I.2.9. Адрес переменной

Пусть имеется программа, в которой объявляется переменная `var` и массив `arr` из десяти элементов, выведите в стандартный поток вывода адреса переменной `var`, а также первого и последнего элемента массива `arr`.

Листинг I.2.5. Объявление переменной и массива

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

int main()
{
  int var = 0;
  double arr[] = {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
}
```

I.2.10. Обход массива при помощи указателей

В листинге I.2.6 представлена программа, в которой выводится содержимое массива из 10 элементов. Перепишите программу таким образом, чтобы в ней не использовалось обращение к элементам массива при помощи квадратных скобок.

Замечание

Следует использовать целочисленную арифметику указателей.

Листинг I.2.6. Обход массива в цикле

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

int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
  for(int i = 0; i < 10; i++)
  {
    cout << arr[i] << endl;
  }

  return 0;
}
```

I.2.11. Получение старшего и младшего разрядов

Пусть имеется переменная типа `int`, содержащая число 2 000 000. Известно, что переменная типа `int` занимает в памяти 4 байта. Получите содержимое первых и последних 2-х байтов, содержимое поместите в две переменные `short int` и выведите в стандартный поток вывода.

I.2.12. Новый тип

Одной из особенностей языка программирования C/C++ является возможность создания нового типа переменных. Например, в листинге I.2.7 определяется новый тип `my_long`, эквивалентный типу `unsigned long int`, объявляется переменная `var` этого типа, которой присваивается значение 10. Помимо этого объявляется тип `my_long_prt`, являющийся указателем на тип `my_long`. Переменная `prt` типа `my_long_prt` указывает на переменную `var`.

Листинг I.2.7. Объявление новых типов

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

int main()
{
  typedef unsigned long int my_long;
  my_long var = 10;
  typedef my_long *my_long_prt;
  my_long_prt prt = &var;

  cout << var << endl;
  cout << *prt << endl;
}
```

При помощи конструкции `typedef` создайте новый тип `char6`, эквивалентный указателю на строку из 6-ти символов, поместите в нее слово "Hello" и выведите в стандартный поток вывода.

I.2.13. Блочный вывод строки

Создайте программу, которая принимает у пользователя строку и выводит ее по 4 символа на одной строке. То есть строка

```
Программирование на C++ — дело не простое
```

должна быть представлена как

```
Прог
рамм
иров
ание
 на
C++
- де
ло н
е пр
осто
е
```

I.2.14. Разбивка строки по пробелу

Создайте программу, которая принимает у пользователя строку и выводит каждое слово строки на отдельной строке. То есть строка

```
Программирование на C++ — дело не простое
```

должна быть представлена как

```
Программирование
на
C++
-
дело
не
простое
```

I.2.15. Найдите ошибку

В листинге I.2.8 представлена программа, в которой переменная `var` получает целое значение и используется для инициализации указателя `prt`. Найдите ошибку в программе.

Листинг I.2.8. Ошибочная программа

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

```
int main()
{
  int var, *prt;

  var = 10;
  *prt = var;
  cout << * prt << endl;

  return 0;
}
```

I.2.16. Допустимо ли выражение ****k=56?

Допустимо ли выражение `****k=56`? Если допустимо, чем является переменная `k`, и как она объявляется? Если выражение не допустимо, то почему? Что вызывает ошибку?

I.2.17. Массив строк

Создайте программу, которая помещает в массив строк текст, введенный пользователем. При вводе пустой строки или вводе десятой строки пользователем все, что до этого пользователь ввел, должно выводиться в стандартный поток.

I.2.18. Динамический массив

Создайте динамический массив со случайным числом элементов от 1 до 100 и заполните его случайными дробными значениями от 0 до 1. Найдите среди полученного массива максимальное и минимальное значения.

I.2.19. Динамический многомерный массив

Создайте матрицу (двумерный массив) со случайным числом строк и столбцов от 1 до 10 и заполните ее случайными дробными значениями от 0 до 1. Найдите сумму всех элементов матрицы.

I.2.20. Заполнение элементов массива

Создайте матрицу, $n \times n$, где n — число от 1 до 10, которое задает пользователь программы. Заполните элементы матрицы целыми числами так, как это представлено в листинге I.2.9, и выведите ее в стандартный поток.

Листинг I.2.9. Квадратная матрица 4×4

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

Создайте альтернативную программу, выводящую квадратную матрицу, представленную в листинге I.2.10.

Листинг I.2.10. Альтернативный способ вывода квадратной матрицы 4×4

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

I.2.21. Чем отличается *int * const* от *int const **?

Модификатор `const` позволяет объявить константу, т. е. переменную, чье значение изменить невозможно. Какими свойствами обладает указатель типа `int * const` и `int const *`, и чем они различаются? Возможен ли указатель типа `int const * const`?

I.2.22. Отличие ссылки от указателя

Помимо указателей, язык C++ допускает объявление ссылок. Чем ссылки отличаются от указателей? Можно ли сказать, что это одно и то же?

I.2.23. Указатель и ссылка на структуру

Пусть имеется структура `person`, представленная в листинге I.2.11. Создайте программу, которая дает возможность пользователю заполнить поля структуры и выводит содержимое структуры в стандартный поток. Создайте два варианта программы, первый вариант для заполнения и вывода в поток должен использовать указатель на структуру, а второй ссылку — манипуляции структурой в обход указателя или ссылки производиться не должны.

Листинг I.2.11. Структура `person`

```
int main()
{
  struct person
```

```
  {
    char family[80];
    char patronymic[80];
    char name[80];
  };
}
```

I.2.24. Указатель на структуру

Пусть имеется структура `person`, представленная в листинге I.2.11. Создайте программу, которая объявляет указатель на эту структуру, выделяет для нее динамически память посредством оператора `new`. Затем объявите ссылку на полученный ранее указатель. Посредством ссылки заполните поля структуры `person` (попросив пользователя ввести значения) и выведите содержимое полей в стандартный поток вывода.

I.2.25. Использование структур для хранения строк

В листинге I.2.12 представлена программа, где в структуре, состоящей из шести переменных типа `char`, хранится строка "Hello". Указатель `char *` ссылается на первый элемент структуры. Что будет содержать строка `str`? Допустим ли такой код?

Листинг I.2.12. Использование структур для хранения строк

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

int main()
{
  struct word
  {
    char ch1;
    char ch2;
    char ch3;
    char ch4;
    char ch5;
    char ch6;
  } wrd;
```

```
  wrd.ch1 = 'H';
  wrd.ch2 = 'e';
  wrd.ch3 = 'l';
  wrd.ch4 = 'l';
  wrd.ch5 = 'o';
  wrd.ch5 = '\0';

  char *str = &wrd.ch1;

  cout << str << "\n";
}
```

I.2.26. Односвязный список

Пусть имеется односвязный список на основе массива, представленный в листинге I.2.13. Каждый элемент списка представляет собой точку (x, y) в двумерном пространстве и указывает посредством поля `next` на следующий элемент списка (индекс массива `curve[]`). Односвязный список `curve[]` образует незамкнутую ломаную кривую. Первый элемент списка является элемент массива `curve[]` с индексом 0, последний элемент списка имеет в поле `next` значение –1.

Листинг I.2.13. Односвязный список

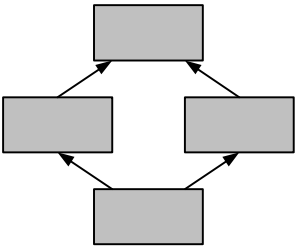
```
#include <iostream>
using namespace std;

typedef struct point
{
  double x;
  double y;
  int next;
} point;

int main()
{
  point curve[] =
  {
    {1.0, 5.0, 2},
    {3.0, 6.0, 6},
    {2.0, 5.0, 1},
    {5.0, 6.0, 4},
    {6.0, 5.0, 5},
```

```
    {7.0, 6.0, -1},
    {4.0, 5.0, 3},
  };

  return 0;
}
```

Выведите содержимое односвязного списка, начиная с первого элемента и заканчивая последним:

```
(x,y) = (1,5)
(x,y) = (2,5)
(x,y) = (3,6)
(x,y) = (4,5)
(x,y) = (5,6)
(x,y) = (6,5)
(x,y) = (7,6)
```

ГЛАВА I.3



Функции

Функции являются важнейшей структурной единицей языка C++. Программист может создавать как собственные функции, так и использовать стандартные библиотечные.

Именно с введением функций связывают эру процедурного программирования. Повторяющиеся фрагменты в программе можно оформить в виде отдельной функции, которую можно вызвать одной строкой. В результате разработчик может поддерживать гораздо больший объем кода: во-первых, многострочные последовательности заменяются вызовом одной функции, и текст программы проще воспринимать, во-вторых, монолитную программу можно разбить на отдельные блоки, которые можно разместить в разных файлах. Последнее особенно важно, т. к. программист хорошо ориентируется лишь в том коде, который занимает объем одной-двух страниц. Такую организацию кода гораздо проще выполнить при помощи функций. Таким образом, функции выполняют две важные задачи: обеспечивают повторное использование кода и улучшают читабельность программы.

Замечание

Решения задач данной главы можно найти в каталоге code\3 компакт-диска, поставляемого вместе с книгой.

I.3.1. Подсчет числа вызовов функции

Создайте функцию, которая будет подсчитывать число собственных вызовов и возвращать это значение.

I.3.2. Подсчет среднего значения

Создайте программу, которая будет принимать у пользователя число и выводить среднее арифметическое всех введенных за сеанс чисел. Программа

должна запрашивать у пользователя числа до тех пор, пока пользователь не введет 0. Процедуру подсчета среднего арифметического реализуйте в виде отдельной функции.

Подсказка

Для реализации функции подсчета среднего арифметического следует использовать статические переменные.

I.3.3. Обработка одномерного массива в функции

Пусть имеется массив `arr`, объявление которого представлено в листинге I.3.1. Необходимо создать функцию `squared()`, которая примет массив `arr`, возведет значения его элементов в квадрат. Вторая функция `print_arr()` должна вывести элементы массива в стандартный поток.

Листинг I.3.1. Одномерный массив `arr`

```
int main()
{
  int arr[] = {3, 4, 5, 6, 7, 8, 9, 10};
  return 0;
}
```

I.3.4. Указатель на последний элемент массива

Создайте функцию, которая принимает в качестве аргумента массив `arr` (см. листинг I.3.1) и возвращает указатель на последний элемент массива — выведите содержимое последнего элемента в стандартный поток вывода.

I.3.5. Функция обмена значений двух переменных

Создайте функцию `swap()`, которая будет принимать два параметра и менять местами их значения. Следует реализовать два варианта: с использованием указателей и ссылок. Как правило, для создания такой функции применяется временная переменная или указатель, попробуйте создать функцию, которая использует только два переданных параметра, не привлекая временных переменных.

Замечание

Предполагается, что функция обменивает значениями две переменные типа `int`.

I.3.6. Рекурсивный вызов

Создайте рекурсивную функцию, которая, принимая в качестве значения целое положительное число `number`, выводила бы его и, если оно не равно нулю, вызывала бы сама себя, передавая в качестве параметра число на единицу меньше: `number − 1`.

I.3.7. Переменная сумма

В математике имеется понятие суммы, обозначаемой символом Σ. Пусть имеется переменное число вложенных сумм:

Σ Σ Σ Σ ... Σ Σ Σ 1

Первая сумма (крайняя левая) пробегает значение от 0 до 1, т. е. она суммирует свой аргумент один раз, вторая сумма от 0 до 2, т. е. суммируем свой аргумент два раза, последняя сумма (крайняя правая) суммирует единицу n раз, где n — количество сумм в последовательности. Напишите программу, которая будет запрашивать у пользователя число вложенных сумм и возвращать их значение согласно представленной выше формуле. Решите задачу рекурсивно и итеративно (без использования рекурсии).

I.3.8. Допустимо ли выражение *f() = 10.0* ?

Допускается ли использование функции в левой части равенства, если нет, то почему? Если допускается, то при каких условиях?

I.3.9. Предотвращение выхода за границы массива

Создайте функции `put()` и `get()`, которые обращались бы к глобальному массиву `int arr[]` и позволяли присваивать значения его элементам и читать соответственно. Функции не должны допускать выхода за границы массива `arr[]`, в котором должно быть 100 элементов.

I.3.10. Вывод строки в стандартный поток

Создайте функцию `print_str()`, которая выводит в стандартный поток строку. Пусть функция может принимать два аргумента, первый из которых является строкой, а второй необязательный аргумент является количеством символов в строке, которые нужно вывести в стандартный поток. Если второй параметр не передается, в стандартный поток должно выводиться не более 30 символов.

I.3.11. Функции *abs()*, *labs()* и *fabs()*

Функции `abs()`, `labs()` и `fabs()` стандартной библиотеки возвращают абсолютное значение целого (`int`), длинного целого (`long int`) и числа с плавающей точкой (`double`). Почему они не реализованы как одна функция `abs()`, которая принимает аргументы всех трех типов? Реализуйте такую функцию.

I.3.12. Ошибка в перегрузке функции

В листинге I.3.2 представлен код, в котором перегружается функция `low()`, для аргумента типа `double` возвращает косинус, а для аргумента типа `float` — синус. Код содержит ошибку, найдите и устраните ее.

Замечание

Для вызова библиотечных функций `sin()` и `cos()` необходимо подключать заголовочный файл `<math>`.

Листинг I.3.2. Перегрузка функции `low()`

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

float low(float var);
double low(double var);

int main()
{
  cout << low(20) << "\n";

  return 0;
}

float low(float var)
{
  return sin(var);
}

double low(double var)
{
  return cos(var);
}
```

I.3.13. Функция с переменным количеством параметров

Создайте функцию, которая будет принимать произвольное количество аргументов и возвращать в качестве результатов их сумму.

Замечание

В качестве первого аргумента допускается передать количество последующих аргументов.

I.3.14. Указатель на функцию

В листинге I.3.3 представлены две однотипные функции — `f1()` и `f2()`. Создайте указатель на эти функции и вызовите их посредством указателя.

Листинг I.3.3. Функции `f1()` и `f2()`

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

void f1(void)
{
  cout << "Выполняется функция f1()\n";
}
void f2(void)
{
  cout << "Выполняется функция f2()\n";
}

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

I.3.15. Обработка функцией элементов массива

Создайте функцию `trig()`, которая принимает массив типа `double` в качестве первого параметра и указатель на библиотечную функцию `sin()` и `cos()` из стандартной библиотеки в качестве второго параметра. В результате работы функции каждый элемент массива должен принять синус или косинус своего старого значения, в зависимости от того указатель на функцию синуса или косинуса передано в качестве второго аргумента функции `trig()`.

Замечание

Для вызова библиотечных функций `sin()` и `cos()` необходимо подключать заголовочный файл `<math>`.

I.3.16. Односвязный список

Используя конструкцию `struct`, создайте односвязный список, содержащий целое число и ссылку на следующий элемент списка (рис. I.3.1). Функция `add_element()` должна добавлять новый элемент в список, функция `delete_element()` удалять элемент из начала списка, а функция `delete_list()` удалять все элементы списка.

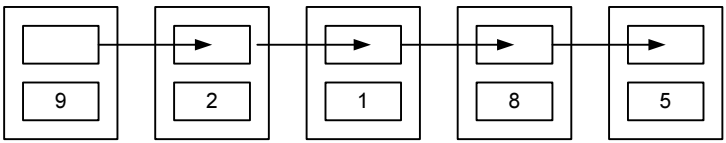


Рис. I.3.1. Односвязный список

Для контроля состояния создайте функцию `print_list()`, которая должна выводить значения элементов с первого элемента до последнего. Так для списка, изображенного на рис. I.3.1, программа должна вывести следующую последовательность цифр:

```
9
2
1
8
5
```

I.3.17. Двухсвязный список

Используя конструкцию `struct`, создайте двухсвязный список, содержащий целое число и ссылку на следующий и предыдущий элемент списка (рис. I.3.2). Создайте набор функций, позволяющих вставлять элемент в начало списка `add_first()`, в конец списка `add_last()`, слева от текущего элемента `add_prev()`, справа от текущего элемента `add_next()`. Кроме того, создайте функции, удаляющие текущий элемент `delete_element()` и весь список `delete_list()`. Для того чтобы двухсвязный список считался полноценным, создайте функцию, которая возвращает указатель на первый `get_first()` и последний `get_last()` элементы списка, а также функцию `get_count()`, подсчитывающую количество элементов в списке.

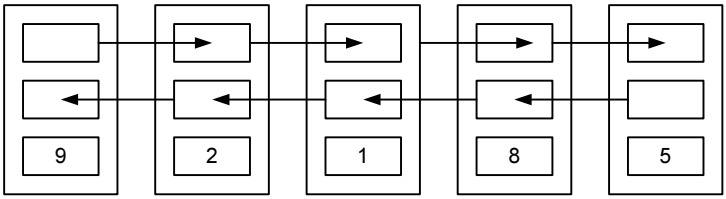


Рис. I.3.2. Двухсвязный список

Для контроля состояния создайте функцию `print_list()`, которая должна выводить значения элементов, начиная с первого элемента и заканчивая последним. Так для списка, изображенного на рис. I.3.2, программа должна вывести следующую последовательность цифр:

```
9
2
1
8
5
```

I.3.18. Создание файла с уникальным именем

Напишите программу, которая будет создавать в текущем каталоге файл с уникальным именем и записывать введенные пользователем строки в файл до тех пор, пока пользователь не введет пустую строку.

I.3.19. Количество строк в файле

Пусть имеется текстовый файл, например, файл созданный при помощи программы из *разд. I.3.18*. Создайте программу, подсчитывающую количество строк в файле.

I.3.20. Вывод случайной строки из файла

Пусть имеется текстовый файл, например, файл созданный при помощи программы из *разд. I.3.18*. Создайте программу, которая будет выводить из файла случайную строку.

I.3.21. Вывод трех случайных строк файла

Создайте программу, которая выводила бы из текстового файла три случайных строки, при этом строки не должны повторяться.

I.3.22. Последние три строки файла

Напишите программу, которая выводила бы из текстового файла три последних строки.

I.3.23. Поиск строки в файле

Создайте программу, которая станет выводить номера строк файла, в которых будет найдено слово, введенное пользователем.

I.3.24. Самая длинная и самая короткая строка в файле

На компакт-диске, поставляемом вместе с книгой, в каталоге code\3 находится стандартный словарь Linux — linux.words. Создайте программу, которая последовательно читает данный файл и выводит самое длинное и самое короткое слово словаря, а также количество символов в нем.

I.3.25. Список слов заданной длины

На компакт-диске, поставляемом вместе с книгой, в каталоге code\3 находится стандартный словарь Linux — linux.words. Создайте программу, которая принимает у пользователя целое число `num` и выводит слова, число символов в которых не превышает, меньше или равно `num`. Подсчитайте количество найденных соответствий.

I.3.26. Поиск слов по первым символам

На компакт-диске, поставляемом вместе с книгой, в каталоге code\3 находится стандартный словарь Linux — linux.words. Создайте программу, которая принимает у пользователя первые символы слова и выводит все найденные соответствия и количество найденных соответствий.

I.3.27. Изменение порядка следования строк в файле

На компакт-диске, поставляемом вместе с книгой, в каталоге code\3 находится стандартный словарь Linux — linux.words. Создайте программу, которая изменяет порядок следования строк в файле, размещая последнюю строку файла первой, предпоследнюю — второй, пред-предпоследнюю — третьей и т. д.

I.3.28. Разбиение файла на части

На компакт-диске, поставляемом вместе с книгой, в каталоге code\3 находится стандартный словарь Linux — linux.words. Создайте программу, которая разбивает содержимое файла linux.words на несколько частей по 1000 слов в каждой и сохраняет каждый фрагмент linux.words в отдельных файлах с именем part и расширением, равным текущему номеру файла, т. е. part.1, part.2, part3, ..., part.100.

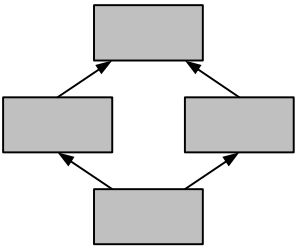
I.3.29. Шаблоны функций

При помощи шаблона функции реализуйте функцию `swap()`, которая будет принимать два параметра произвольного типа и менять местами их значения.

I.3.30. Перегрузка шаблона функций

Создайте два варианта функций `mmin()` и `mmax()`, которые возвращают минимальное и максимальное значения из ряда чисел. Первый вариант должен принимать указатель на массив элементов и количество элементов в массиве, а второй должен принимать лишь два значения. Оба варианта должны использовать шаблоны так, чтобы для аргументов допускалось использование любого базового типа.

ГЛАВА I.4



Объекты и классы

Классы позволяют объединять переменные и обрабатывающий их код в единую структуру. Это поднимает процесс разработки программы на новый уровень абстракции, программист получает возможность программировать не в терминах функций, переменных, файлов, а при помощи абстрактных понятий: можно отправить объект "договор" объекту "принтер", а к объекту "словарь" применить метод поиска. Библиотеки классов позволяют сосредоточиться на решении прикладных задач и думать в терминах прикладной задачи, не увязая в реализации, алгоритмах и машинном представлении программы.

Объектно-ориентированная технология в C++ является чрезвычайно мощным инструментом, сродни бензопилы, однако при неумелом использовании этот инструмент может нанести вред и тому, кто им пользуется. Язык C++, начиная с указателей и заканчивая объектно-ориентированной технологией и библиотекой STL, построен так, чтобы выжать максимум из компьютера — поэтому здесь нет даже намека на "защиту от дурака". Чтобы создавать эффективные программы, следует от начала до конца понимать, что делаешь, зачем и к каким последствиям это приведет.

Замечание

Решения задач данной главы можно найти в каталоге code\4 компакт-диска, поставляемого вместе с книгой.

I.4.1. Чем отличается структура *struct* от класса *class*?

Отличается ли чем-то структура `struct` от класса `class` или данные ключевые слова дублируют друг друга?

I.4.2. Чем отличается объединение *union* от класса *class*?

Назовите сходства и различия объединения `union` и класса `class`.

I.4.3. Константы в классах

Создайте класс `cls`, который бы в качестве одного из значений содержал константу `COUNT`. Объявите два объекта класса `cls`. Поле `COUNT` первого объекта `fst` должно иметь значение 100, а поле `COUNT` второго объекта `snd` — значение 200.

I.4.4. Подсчет количества созданных объектов

Создайте класс, который бы подсчитывал, сколько объектов данного класса создано в настоящий момент.

I.4.5. Найдите ошибку

В листинге I.4.1 приведен код, реализующий класс `cls` и объявляющий объект `obj` данного класса. Однако класс реализован ошибочно. Найдите и исправьте ошибку.

Листинг I.4.1. Определение класса

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

class cls
{
  public:
    void cls(int fst, int snd);
  private:
    int first;
    int second;
};
// Конструктор
void cls::cls(int fst, int snd)
{
  first = fst;
  second = snd;
}
```

```
int main()
{
  cls obj(1,2);

  return 0;
}
```

I.4.6. Использование объекта в нескольких файлах

Определение класса `number` находится в файле myclass.h (листинг I.4.2), реализация методов находится в файле myclass.cpp (листинг I.4.3).

Листинг I.4.2. Файл myclass.h

```
#ifndef MyclassH
#define MyclassH
class number
{
  public:
    number(int value);
    int get_number();
  private:
    int var;
};
#endif
```

Листинг I.4.3. Файл myclass.cpp

```
#include "myclass.h"

number::number(int value)
{
  var = value;
}
int number::get_number()
{
  return var;
}
```

В файлах main.cpp и utils.cpp используется один и тот же объект `num` класса `number`. Как его следует объявить, чтобы в файлах main.cpp и utils.cpp использовался один и тот же объект?

I.4.7. Инициализация объекта при помощи =

В листинге I.4.4 используется объект `obj`, при этом его инициализация производится не посредством передачи параметров конструктору, а при помощи оператора `=`. Какова должна быть реализация класса `cls`, чтобы код в листинге I.4.4 был рабочим?

Листинг I.4.4. Инициализация объекта при помощи оператора =

```
#include <iostream>
#include "cls.h";
using namespace std;

int main()
{
  cls obj = 12;
  cout << obj.number << "\n";

  return 0;
}
```

I.4.8. Класс с динамическим массивом

Создайте класс, который бы при создании объекта в конструкторе выделял память под целочисленный массив, объем которого задается параметром конструктора, а при уничтожении в деструкторе освобождал бы память. Создайте интерфейс для доступа к членам массива (чтение и запись) с контролем выхода за границы массива.

I.4.9. Класс-интерфейс к файлу

На компакт-диске, поставляемом вместе с книгой, в каталоге code\4 находится стандартный словарь Linux — linux.words. Создайте класс, объект которого в конструкторе открывает этот файл, читает его содержимое в массив, а в деструкторе закрывает файл. Класс должен содержать перегруженный метод `print()`, который должен реализовывать функциональность, представленную в табл. I.4.1.

Таблица I.4.1. *Варианты использования метода* `print()`

Синтаксис метода `print()`	Описание
`print()`	Без параметров метод `print()` должен выводить все содержимое файла linux.words

Таблица I.4.1 *(окончание)*

Синтаксис метода `print()`	Описание
`print(char *str)`	Если метод `print()` принимает в качестве параметра единственную строку `str`, в стандартный поток должны выводиться строки файла linux.words, которые начинаются со строки `str`
`print(int min, int max)`	Если метод принимает два целочисленных параметра, то в стандартный поток должны выводиться слова из файла linux.words, состоящие из количества символов больше или равных `min` и меньше или равных `max`
`print(int count)`	Если метод `print()` принимает единственный целочисленный параметр `count`, в стандартный поток должны выводиться слова, содержащие не больше `count` символов

I.4.10. Постраничная навигация

В листинге I.4.5 представлен класс `part`, позволяющий хранить в каждом объекте класса 10 строк длиной 80 символов. На компакт-диске, поставляемом вместе с книгой, в каталоге code\4 находится стандартный словарь Linux — linux.words. Создайте класс `dict`, который бы читал файл linux.words, разбивал его на части по 10 слов, создавая под каждую из таких частей объект `part`, реализуя массив объектов `part`. На основании этих двух классов создайте программу, которая бы загружала словарь и позволяла пользователю выводить части словаря по индексу массива объектов `part`.

Листинг I.4.5. Класс `part`

```
class part
{
  public:
    static const int NUMBER_STRINGS = 10;
    static const int NUMBER_CHARS   = 80;
    char str[NUMBER_STRINGS][NUMBER_CHARS];
    part()
    {
      for(int i = 0; i < NUMBER_STRINGS; i++)
      {
        str[i][0] = '\0';
      }
    }
```

```
    void print()
    {
      for(int i = 0; i < NUMBER_STRINGS; i++)
      {
        cout << str[i];
      }
    }
};
```

I.4.11. Алфавитная навигация

На компакт-диске, поставляемом вместе с книгой, в каталоге code\4 находится стандартный словарь Linux — linux.words, слова в котором отсортированы по алфавиту. Создайте два класса: `symbol` и `dict`. Объекты первого класса, `symbol`, должны содержать слова словаря, начинающиеся на одну из букв алфавита. Второй класс, `dict`, должен содержать 26 объектов (по числу символов английского алфавита) класса `symbol`.

I.4.12. Дружественная функция

Создайте дружественную функцию `print()`, которая бы выводила в стандартный поток содержимое закрытого массива `array` в классе `arr` (листинг I.4.6).

Листинг I.4.6. Класс `arr`

```
class arr
{
  public:
    arr(int num)
    {
      count = num;
      array = new int[count];
      for(int i = 0; i < count; i++) array[i] = i*i;
    }
    ~arr()
    {
      delete [] array;
    }
  private:
    int count;
    int *array;
};
```

I.4.13. Блокировка файла по статическому члену класса

Создайте класс `put_file`, который при помощи метода `add()` дописывает в конец файла text.txt строку. При этом класс должен содержать статический член `open_file`, который принимает значение `true`, если файл text.txt в данный момент открыт для записи одним из объектов класса `put_file`, и `false`, если файл не занят ни одним объектом. Пока статический член `open_file` равен `true`, все остальные объекты класса `put_file` должны ожидать освобождения файла. Таким образом, необходимо реализовать блокировку файла по статической переменной `open_file` (рис. I.4.1).

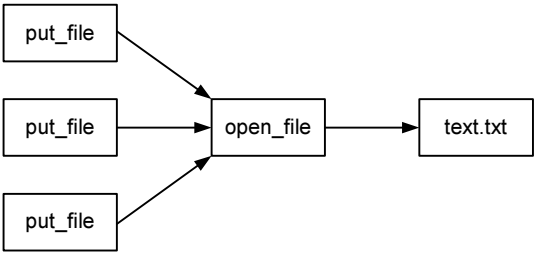


Рис. I.4.1. Блокировка файла text.txt по статической переменной `open_file`

I.4.14. Блокировка файла двумя классами

Создайте два класса: `fst` и `snd`, которые при помощи метода `add()` дописывают в конец файла text.txt строку. Каждый из двух классов должен иметь закрытый статический член `open_file`, который принимает значение `true`, если в файл text.txt в настоящий момент записывается строка, и значение `false`, если файл свободен. Создайте дружественную функцию `is_free_file()`, которая бы возвращала `true`, если файл не используется ни одним из объектов классов `fst` и `snd`, и `false` — в противном случае (рис. I.4.2).

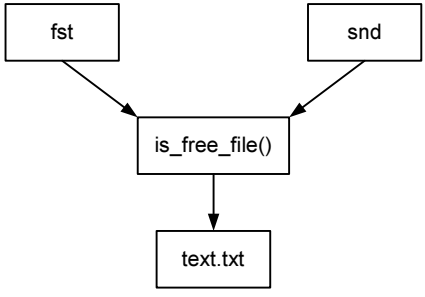


Рис. I.4.2. Проверка блокировки файла при помощи дружественной функции `is_free_file()`

Замечание

В программе будет использоваться только по одному объекту классов `fst` и `snd`.

I.4.15. Копирующий конструктор

Как объявляется копирующий конструктор и для каких целей он применяется? В листинге I.4.7 приводится класс `arr`, создайте для него копирующий конструктор и перечислите основные случаи, когда используется копирующий конструктор.

Листинг I.4.7. Класс `arr`

```
class arr
{
  public:
    arr(int number);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);
  private:
    int count;
    int *array;
};
arr::arr(int number)
{
  count = number;
  array = new int[number];
  for(int i = 0; i < number; i++) array[i] = i*i;
}
arr::~arr()
{
  delete [] array;
}
int arr::get_count()
{
  return count;
}
int arr::get_arr(int index)
{
  if(index >= 0 && index < count) return array[index];
  else return -1;
}
```

```
int &arr::set_arr(int index)
{
  if(index >= 0 && index < count) return array[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}
```

I.4.16. Перегрузка оператора =

Для класса `arr` (см. листинг I.4.7) перегрузите оператор `=`, создайте программу, демонстрирующую работу оператора `=` в классе `arr`.

I.4.17. Перегрузка логических операторов

Для класса `arr` (см. листинг I.4.7) перегрузите операторы `>`, `<`, `>=`, `<=`, `==` и `!=`. Сравнению должны подвергаться элементы массивов. Если один массив имеет больше элементов, чем другой, сравнению подлежит количество элементов, которые содержит короткий массив. Например, для оператора `==` два массива будут равны, если их соответствующие элементы равны друг другу. Оператор `>` должен возвращать `true`, если все элементы левого массива больше элементов правого массива, в противном случае оператор должен возвращать `false`.

I.4.18. Перегрузка операторов +, –, / и *

Для класса `arr` (см. листинг I.4.7) перегрузите операторы `+`, `-`, `/` и `*`. При сложении, вычитании, делении и умножении массивов данным операторам должны подвергаться соответствующие элементы массивов. Если в одном из массивов меньше элементов, чем в другом, возвращаться должен массив с минимальным количеством элементов. Для операторов `+` и `-`, помимо бинарной версии (`a - b`), перегрузите унарный оператор (`-a`). Оператор `+` должен возвращать массив, каждый из элементов которого будет взят по абсолютному значению, а оператор `-` должен менять значение каждого элемента на противоположный. То есть при применении унарного оператора `-` отрицательные элементы должны становиться положительными, а положительные — отрицательными.

I.4.19. Перегрузка операторов ++ и --

Для класса `arr` (см. листинг I.4.7) перегрузите операторы `++` и `--`. Оператор `++` должен увеличивать на единицу каждый элемент массива, а оператор `--` дол-

жен уменьшать на единицу каждый элемент массива. Следует учитывать, что операторы ++ и -- имеют префиксную и постфиксную формы, необходимо реализовать оба варианта.

I.4.20. Перегрузка оператора *[]*

Для класса arr (см. листинг I.4.7) перегрузите оператор [] так, чтобы его можно было использовать и для получения элементов массива по индексу, и для операций присваивания новых значений отдельному элементу массива. В качестве проверки можно использовать код из листинга I.4.8.

Листинг I.4.8. Использование оператора [] для объектов класса arr

```
int main()
{
  arr fst(5);
  fst[0] = 4;
  fst[1] = 1;
  fst[2] = 7;
  fst[3] = 2;
  fst[4] = 9;

  for(int i = 0; i < fst.get_count(); i++)
  {
    cout << fst[i] << "\n";
  }

  return 0;
}
```

Программа из листинга I.4.8 должна вывести содержимое объекта fst:

```
4
1
7
2
9
```

I.4.21. Перегрузка оператора *()*

Для класса arr (см. листинг I.4.7) перегрузите оператор (), который должен принимать одно целое значение и вставлять его в конец массива array, увеличивая количество его элементов на единицу.

I.4.22. Наследование одного класса другим

В листинге I.4.9 представлено определение класса `base`, унаследуйте от него класс `child` с использованием спецификаторов `public`, `private` и `protected`. Какой статус будут иметь члены `first`, `second` и `third` в новом классе `child`?

Листинг I.4.9. Класс `base`

```
class base
{
  public:
    int first;
    base()
    {
      first = 1;
      second = 2;
      third = 3;
    }
  private:
    int second;
  protected:
    int third;
};
```

I.4.23. Расширение функциональности класса

От класса `arr` (см. листинг I.4.7) унаследуйте класс `file_arr`, конструктор которого принимает количество элементов в массиве, имя файла, сохраняет массив в файл, оставляя его открытым на все время существования объекта. В деструкторе класса `file_arr` файл должен закрываться. Следует ли в деструкторе освобождать память, выделенную под массив `array` в конструкторе класса `arr`?

I.4.24. Перегрузка метода базового класса

В листинге I.4.10 представлен класс `base`, который содержит метод `print()`. Унаследуйте от него производный класс `child()`, также содержащий метод `print()`, который выводит в стандартный поток сообщение о вызове метода `print()` производного класса, и вызывающий метод `print()` базового класса.

Листинг I.4.10. Класс base

```
class base
{
  public:
    void print()
    {
      cout << "Вызов метода print() базового класса base\n";
    }
};
```

I.4.25. Виртуальный класс

Что такое виртуальный класс и для каких целей он применяется? Приведите пример использования виртуального класса.

I.4.26. Указатель на объект базового типа

В листинге I.4.11 приводится пример, в котором от базового класса `base` наследуется производный класс `child`.

Листинг I.4.11. Наследование производного класса child от базового класса base

```
class base
{
  public:
    void print()
    {
      cout << "Вызов метода базового класса base\n";
    }
};

class child : public base
{
  public:
    void print()
    {
      cout << "Вызов метода производного класса child\n";
    }
};
```

Создайте указатель `p` на базовый класс `base`, который бы указывал на объект `child`. При помощи указателя `p` вызовете методы `print()` базового и производного классов.

I.4.27. Чем отличается виртуальная функция от чисто виртуальной функции?

Чем отличается виртуальная функция от чисто виртуальной функции? Продемонстрируйте различие на примере базового класса `base`, от которого наследуются два производных класса: `first` и `second` (листинг I.4.12), создав виртуальную функцию `print()`.

Листинг I.4.12. Наследование производного класса `child` от базового класса `base`

```
class base {};
class first : public base {};
class second : public base {};
```

I.4.28. Динамическая идентификация типов

Что такое динамическая идентификация типов? Каковы различия применения оператора `typeid()` для указателей на полиморфные (содержащие виртуальные функции) и обычный классы?

I.4.29. Приведение типов

В языке C++ существует пять операторов приведения типов. Это традиционные, доставшиеся от C круглые скобки `()`, а также четыре новых оператора: `dynamic_cast`, `const_cast`, `reinterpret_cast`, `static_cast`. Чем отличаются друг от друга эти пять операторов?

I.4.30. Обобщенный класс безопасного массива

Создайте шаблон для класса безопасного массива `arr` (см. листинг I.4.7), который бы поддерживал базовые типы данных.

I.4.31. Использование параметров в шаблонах классов

Создайте шаблон для класса безопасного массива `arr` (см. листинг I.4.7), который бы имел два параметра: первый определял тип элементов массива, а второй — количество элементов в массиве. Объявление целочисленного массива из 10-ти элементов должно выглядеть так, как это представлено в листинге I.4.13.

Листинг I.4.13. Использование параметров в шаблонах классов

```
int main()
{
  arr <int, 10> obj;
  for(int i = 0; i < 10; i++)
  {
    cout << obj.get_arr(i) << "\n";
  }
  return 0;
}
```

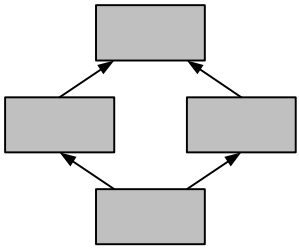
I.4.32. Перегрузка шаблонов

Создайте шаблон класса `value`, который бы хранил единственный член `var` произвольного типа, значение которого задавалось бы параметром конструктора. Метод `get()` класса `value` должен возвращать значение члена `var`. Шаблон должен реализовывать одно исключение: если в качестве параметра шаблона передается тип `int`, то метод `get()` должен возвращать строку "Передано целочисленное значение".

I.4.33. Обобщенный двухсвязный список

Используя шаблоны, создайте двухсвязный список, который бы хранил произвольный тип данных. В качестве демонстрации создайте программу, которая организует список из значений `double`.

ГЛАВА I.5



Исключения

Обработка ошибок зачастую занимает львиную долю программы, особенно в таком капризном языке, как C++. Еще в C программисты часто приходили к выводу, что им приходится слишком часто дублировать код, обрабатывающий ошибки. С введением классов ситуация только усугубилась, поэтому был введен механизм исключений, позволяющий обрабатывать ошибки оптом в одном обработчике.

Замечание

Решения задач данной главы можно найти в каталоге code\5 компакт-диска, поставляемого вместе с книгой.

I.5.1. Генерация исключений

Исключения генерируются при помощи оператора `throw`, а обрабатываются при помощи конструкции `try...catch`. Создайте функцию, принимающую единственный целочисленный параметр и генерирующую исключение, если переданное ей значение не лежит в диапазоне от 0 до 100. Какой тип должно иметь исключение, какие ограничения накладываются на данный тип?

I.5.2. Перехват исключений в иерархии классов

Создайте класс исключений `base_exception` и унаследуйте от него производный класс `child_exception`. Будет ли обработчик для `base_exception` перехватывать исключения `child_exception` и наоборот, будет ли обработчик для `child_exception` перехватывать исключения для `base_exception`?

I.5.3. Перехват всех исключений

Как известно, исключения могут принадлежать различным классам. Каким образом можно перехватить все исключения вне зависимости от их типов?

I.5.4. Функция, генерирующая исключение

Создайте функцию `print_array()`, которая выводит в стандартный поток содержимое целочисленного массива, указатель на который передается через первый параметр, а количество элементов в массиве — через второй. При этом генерирует исключение типа `int` (`-1`), если массив имеет меньше 10 элементов, исключение типа `char *` (`"too_long"`), если количество элементов в массиве больше 100, и исключение типа `float` (`1.0`), если хоть один из элементов массива окажется отрицательным. Функция должна обрабатывать каждый из типов исключений и передавать их в основную программу.

В основной программе в блок `try` поместите код выделения динамической памяти для массива `arr` и его инициализацию, а также вызов функции `print_array()`, которая выведет в стандартный поток массив `arr`. Создайте обработчик, который будет отлавливать каждый из типов исключительной ситуации и освобождать ранее выделенную динамическую память под массив `arr`.

I.5.5. Выделение динамической памяти

Как правильно следует обрабатывать выделение динамической памяти? В листинге I.5.1 приводится пример выделения динамической памяти под массив `int`, количество элементов в котором задается пользователем. Обработайте исключительную ситуацию нехватки памяти.

Листинг I.5.1. Выделение динамической памяти

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

int main()
{
  int number = 0;
  int *arr;
  // Запрашиваем у пользователя количество элементов в массиве
  cout << "Введите количество элементов в массиве ";
  cin >> number;

  arr = new int[number];
```

```
  delete [] arr;

  return 0;
}
```

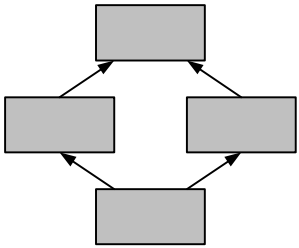
I.5.6. Перегрузка операторов *new* и *delete*

Пусть имеется класс `var_class` (листинг I.5.2), перегрузите для него операторы `new` и `delete`, так чтобы они использовали С-функции `malloc()` и `free()` для выделения и освобождения динамической памяти.

Листинг I.5.2. Класс `var_class`

```
class var_class
{
  public:
  int var;
}
```

ГЛАВА I.6



Стандартная библиотека

Существуют операции, которые программисты вынуждены реализовывать практически в каждом более или менее крупном проекте: выделение памяти, организация списков, их сортировка и поиск отдельных элементов. Для решения этих проблем стандарт языка C++ предусматривает реализацию стандартной библиотеки, позволяющей решать наиболее распространенные задачи схожими средствами.

Самой мощной частью стандартной библиотеки является библиотека STL (Standard Template Library), представляющая собой библиотеку шаблонов классов, которые позволяют обрабатывать наиболее типичные массивы данных. Вместо того чтобы всякий раз разрабатывать собственную библиотеку или разбираться в чужих библиотеках хранения данных (классы строк, безопасных массивов, списков и т. п.), было принято решение включить в стандарт языка C++ библиотеку STL, обязательную к реализации в любом компиляторе.

Замечание

Для освоения библиотеки STL необходимо четкое понимание шаблонов, которые составляют основу этой библиотеки.

Замечание

Решения задач данной главы можно найти в каталоге code\6 компакт-диска, поставляемого вместе с книгой.

I.6.1. Стандартное пространство имен

Что такое пространство имен? Для чего предназначено пространство имен `std`?

I.6.2. Класс *auto_ptr*

Пусть имеется класс `positive`, который хранит целое число `variable` и помимо конструктора имеет один-единственный метод `get_variable()`, который возвращает значение `variable`, если оно положительно и генерирует исключение, если оно отрицательно (листинг I.6.1).

Листинг I.6.1. Класс `positive`

```
class positive
{
  public:
    positive(int var)
    {
      variable = var;
    }
    int get_variable()
    {
      if(variable <  0) throw -1;
      else return variable;
    };
  private:
    int variable;
};
```

Функция `print_obj()` динамически создает объект `obj` класса `positive`, инициируя его переданным ей в качестве параметра значением `number`. После чего при помощи метода `get_variable()` в стандартный поток выводится значение переменной (листинг I.6.2).

Замечание

Для работы с классом `auto_ptr` необходимо подключить библиотеку `<memory>`.

Листинг I.6.2. Функция `print_obj()`

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

void print_obj(int number);

int main()
```

```
{
  int number = 0;
  cout << "Введите положительное число ";
  cin >> number;

  // Выводим объект в стандартный поток
  print_obj(number);

  return 0;
}

void print_obj(int number)
{
  try
  {
    positive *obj = new positive(number);

    try
    {
      cout << obj->get_variable();
    }
    catch(...)
    {
      delete obj;
      throw;
    }
  }
  catch(bad_alloc expt)
  {
    cout << "Не удалось выделить память под объект arr\n";
  }
}
```

Если пользователь вводит положительное число, оно выводится в стандартный поток, если отрицательное — генерируется исключение.

Перепишите представленную в листинге I.6.2 функцию `print_obj()` с использованием объекта класса `auto_ptr`.

I.6.3. Присваивание и класс *auto_ptr*

Можно ли избежать перегрузки оператора "равно" = в классе, если память под объект будет выделяться при помощи класса `auto_ptr`?

I.6.4. Какие типы контейнеров поддерживаются в STL?

На рис. I.6.1 схематически приведены пять основных контейнеров STL. Назовите их и охарактеризуйте.

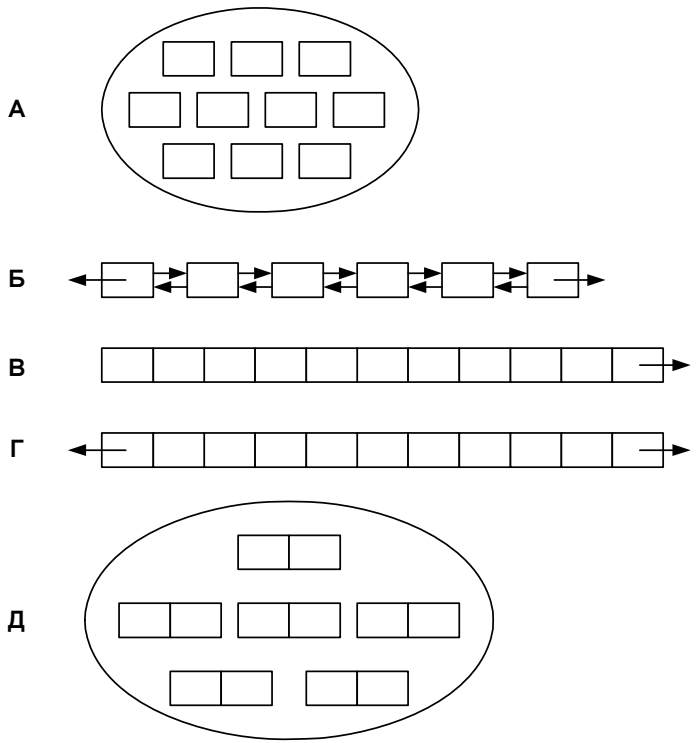


Рис. I.6.1. Схематичное изображение контейнеров STL

I.6.5. Работа с вектором

Создайте программу, которая запрашивала бы у пользователя число `count` и создавала бы вектор (`vector`), который был бы заполнен целочисленными элементами, начиная с 1 и заканчивая `count`. Выведите элементы коллекции в стандартный поток через пробел.

I.6.6. Работа с деком

Создайте программу, которая запрашивала бы у пользователя число `count` и создавала дек (`deque`), который был бы заполнен целочисленными элементами

от -count до count. Выведите элементы коллекции в стандартный поток через пробел. То есть, если пользователь вводит цифру 5, программа должна вывести строку:

```
-5 -4 -3 -2 -1 0 1 2 3 4 5
```

I.6.7. Работа со списком

Создайте программу, которая бы заполняла коллекцию-список буквами английского алфавита в нижнем регистре и выводила бы содержимое коллекции в стандартный поток через пробел. Далее программа должна переводить в верхний регистр каждый символ в коллекции и выводить содержимое списка повторно.

I.6.8. Работа с множеством

Создайте программу, которая бы запрашивала у пользователя целые числа (до тех пор, пока не будет введено отрицательное число) и размещала их в коллекции типа множество. Реализуйте два варианта программы: первый вариант должен выводить содержимое коллекции по возрастанию, а второй — по убыванию.

I.6.9. Работа с отображением

Создайте программу, которая объявляет отображение арабских цифр (от 1 до 10) и их английских названий (табл. I.6.1).

Таблица I.6.1. *Соответствие цифр и их названий*

Цифра	Название
1	one
2	two
3	three
4	four
5	five
6	six
7	seven
8	eight
9	nine
10	ten

Пусть у пользователя запрашивается строка, и если она совпадает с одним из английских названий, выводится соответствующая ему цифра.

I.6.10. Преобразование одной коллекции в другую

Выполните преобразование коллекции `vector<int>` в коллекцию `list<float>`.

I.6.11. Допускается ли сравнение коллекций друг с другом?

Допускается ли сравнение коллекций друг с другом? Другими словами, можно ли применять операторы ==, !=, <, >, >=, <= к коллекциям, а не только их элементам?

I.6.12. Сортировка строк

Создайте программу, которая запрашивает у пользователя строки и размещает их в коллекции до тех пор, пока пользователь не введет пустую строку. Выведите строки в порядке убывания.

I.6.13. Поиск максимального и минимального значений коллекции

Создайте программу, которая бы запрашивала у пользователя целые числа (до тех пор, пока не будет введено отрицательное число). Как результат, программа должна выводить в стандартный поток максимальное и минимальное из введенных чисел.

I.6.14. Обращение порядка следования элементов

В листинге I.6.3 представлена программа, которая объявляет коллекцию-вектор и заполняет ее элементами.

Листинг I.6.3. Заполнение коллекции `vector`

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

int main()
{
  vector<int> coll;
```

```
  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Модифицируйте программу таким образом, чтобы, начиная с элемента 4 и до последнего элемента коллекции (6), менялся порядок на обратный. То есть в результате должен получиться следующий ряд:

```
1 2 3 6 5 4
```

I.6.15. Сортировка содержимого файла

На компакт-диске, поставляемом вместе с книгой, в каталоге code\4 находится стандартный словарь Linux — linux.words. Создайте программу, которая читает содержимое файла, сортирует слова, начиная со строки "deep" по подстроку "submit" в соответствии со следующим критерием сортировки: слово считается меньше чем другое слово, если оно содержит меньшее количество символов. Слова до строки "deep" и после строки "submit" должны оставаться неизменными.

I.6.16. Создание копии коллекции

В листинге I.6.3 создается коллекция-вектор `coll`. Средствами библиотеки STL создайте копию объекта `coll` — `copy_coll`.

I.6.17. Удаление элементов коллекции

В листинге I.6.4 представлена программа, которая удаляет из коллекции `coll` при помощи глобального алгоритма `remove()` элементы со значением 3.

Листинг I.6.4. Удаление элементов коллекции со значением 3

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

int main()
{
  deque<int> coll;
  deque<int>::const_iterator pos;

  try
  {
    // Заполняем коллекцию
    for(int i = 1; i < 5; i++)
    {
      coll.push_front(i);
      coll.push_back(i);
    }

    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    cout << *pos << ' ';
    cout << "\n";

    // Удаляем все элементы со значением 3
    remove(coll.begin(), coll.end(), 3);

    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    cout << *pos << ' ';
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Результат работы программы из листинга I.6.4 выглядит следующим образом:

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

Причем если воспользоваться методом `size()`, то выяснится, что количество элементов коллекции осталось неизменным. Объясните результат работы программы и добейтесь того, чтобы в коллекции `coll` элементы со значением 3 уничтожались окончательно.

I.6.18. Вывод содержимого произвольной коллекции

Создайте функцию `print_collection()`, которая принимает ссылку на произвольную коллекцию (вектор, дек, список, множество, мультимножество, отображение или мультиотображение) и выводит в стандартный поток ее содержимое. Создайте альтернативную функцию `print_back_collection()`, которая выводит содержимое коллекции в обратном порядке. Кроме этого, по аналогии с алгоритмами библиотеки STL создайте функцию `print_interval()`, которая бы принимала начало и конец интервала и выводила содержимое коллекции из этого интервала.

I.6.19. Преобразование коллекции при копировании

Пусть имеется коллекция-вектор `coll`, создание которой обеспечивает код в листинге I.6.3. Создайте на основе коллекции `coll` новую коллекцию `coll_list` типа список, элементы которой будут соответствовать квадрату соответствующих элементов коллекции `coll`.

I.6.20. Что такое предикат?

Что такое предикат? Продемонстрируйте использование предиката.

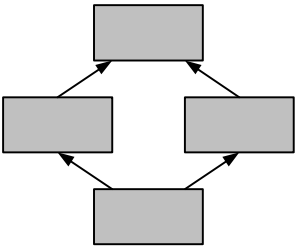
I.6.21. Что такое объект-функция?

Что представляет собой объект-функция? Какие стандартные объекты-функции вам известны?

I.6.22. В чем особенность контейнера *vector<bool>*?

Чем контейнер `vector<bool>` отличается от контейнеров других типов, например, `vector<int>`?

ГЛАВА I.7



Ввод/вывод

Проблема ввода/вывода в языке C++ достаточно многогранна. Это обусловлено двумя факторами: во-первых, функция ввода/вывода очень тесно связана с операционной системой (операционных систем много, и они, в отличие от языков программирования, нестандартны, если не принимать во внимание стандарт POSIX), а во-вторых, язык C++ включает в себя функции ввода/вывода C и объектно-ориентированную библиотеку потоковых классов `IOStream`, введенную в C++. Библиотеки C и C++ используют разную философию ввода/вывода для достижения одного и того же результата — оба подхода оттачивались в течение длительного времени, и любой программист должен владеть ими обоими.

Замечание

Решения задач данной главы можно найти в каталоге code\7 компакт-диска, поставляемого вместе с книгой.

I.7.1. Что такое поток?

В системе ввода/вывода фигурирует понятие потока. Что это такое и какие виды потоков вам известны?

I.7.2. Выравнивание строк по правому краю

Создайте программу, которая бы запрашивала у пользователя строки до тех пор, пока он не введет пустую строку, после чего программа должна вывести список введенных строк, отсортированных по алфавиту и выровненных по правому краю. Например, если пользователь вводит последовательность строк

```
Hello
world
C++
program
```

программа должна вывести данные строки в формате

```
     C++
   Hello
programm
   world
```

Замечание

Решите программу в С-стиле, т. е. без использования библиотеки <iostream>, используя С-функции ввода/вывода.

I.7.3. Выравнивание строк по правому и левому краям

Создайте программу, которая бы запрашивала у пользователя строки до тех пор, пока он не введет пустую строку, после чего программа должна вывести список введенных строк в два столбца, первый из которых выровнен по левому краю, а второй — по правому краю. То есть результат работы программы может выглядеть примерно следующим образом:

```
Aarhus            Aaron
Ababa              aback
abaft           abandon
abandoned    abandoning
abandonment    abandons
abase            abased
abasement    abasements
```

I.7.4. Ввод строк пользователем

Прочитайте из стандартного потока ввода в строку, длиной 80 символов, строку, введенную пользователем в стиле языков С и С++. Проследите за тем, чтобы отсутствовал выход за границу массива. Полученную строку выведете в стандартный поток вывода.

I.7.5. Перегрузка операторов >> и <<

Операторы >> и << перегружены для всех базовых типов, однако для того чтобы выводить состояние пользовательских классов, необходимо их пере-

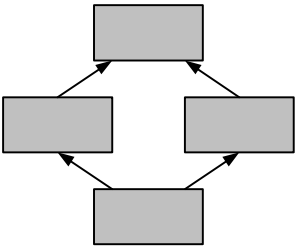
гружать. В листинге I.7.1 представлен класс `cls`, содержащий в своем составе два открытых члена: строковое `key` и числовое `value`. Перегрузите методы `>>` и `<<` таким образом, чтобы можно было вводить и выводить объекты класса `cls` в стандартные потоки.

Листинг I.7.1. Класс `cls`

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

class cls
{
  public:
    cls(string str, int val)
    {
      key = str;
      value = val;
    }
    cls()
    {
      key = "";
      value = 0;
    }
    string key;
    int value;
};
```

I.7.6. Собственный манипулятор

Создайте собственный манипулятор `ednll` для стандартного потока вывода, который выводит два перевода строки и сбрасывает буфер.



ГЛАВА I.8

Разное

Язык C++ позволяет решать практически любые задачи, которые встают перед современным программистом. Так как программы, созданные с помощью C++, обладают высоким быстродействием, язык применяется для решения широкого круга математических задач. Современные научные и инженерные вычисления, несмотря на постоянно увеличивающуюся мощность вычислительных средств, зачастую требуют многодневных и многомесячных вычислений. Различная реализация того или иного блока может приводить к многократному увеличению скорости решения задачи.

С другой стороны различные математические приемы используются для решения повседневных задач — сглаживание графиков, вычисление площади перекрывания графических элементов и т. п.

Данная глава, с одной стороны, позволяет опробовать язык C++ в решении конкретных задач с использованием всех элементов языка, с другой стороны, помогает потренироваться в решении несложных математических задач, которые встречаются в повседневной практике программиста.

Замечание

Решения задач данной главы можно найти в каталоге code\8 компакт-диска, поставляемого вместе с книгой.

I.8.1. Кривая Безье

Кривая Безье — это система параметрических уравнений, которая позволяет по координатам (x, y) четырех точек построить сглаженную кривую (рис. I.8.1). Система уравнений выглядит следующим образом:

$$x(t) = (1-t)^3 x_0 + 3t(1-t)^2 x_1 + 3t^2(1-t)x_2 + t^3 x_3,$$
$$y(t) = (1-t)^3 y_0 + 3t(1-t)^2 y_1 + 3t^2(1-t)y_2 + t^3 y_3.$$

Параметр t принимает произвольное значение от 0 до 1, что позволяет получить координаты любой точки (x, y) сглаженной кривой.

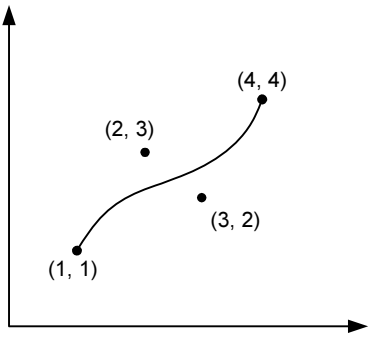


Рис. I.8.1. Построение сглаженной кривой по четырем точкам

Создайте программу, которая запрашивает у пользователя четыре точки с координатами (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) и формирует файл text.out, куда выводит координаты сглаженной кривой (1000 точек).

I.8.2. Преобразование строк в массив

Одной из наиболее часто встречающихся в программировании задач является преобразование строки в массив и обратное преобразование массива в строку.

Создайте функцию `explode()`, которая принимает в качестве первого аргумента символ-разделитель строки `separator`, в качестве второго аргумента саму строку `str`. Через третий аргумент должна передаваться коллекция "вектор" `vec` библиотеки STL, в элементы этого вектора функция должна помещать отдельные фрагменты строки `str`, разделенные символом-разделителем `separator`.

Кроме этого, необходимо реализовать обратную функцию `implode()`, которая объединяет элементы коллекции "вектор" `vec` (содержащих строки) в единую строку `str`. Функции должны иметь следующие прототипы:

```
void explode(char separator, string &str, vector<string> &vec)
void implode(char separator, vector<string> &vec, string &str)
```

В качестве демонстрации работы функций следует создать программу, которая читает файл linux.words, разбивает его содержимое при помощи функции `explode()` по символу перевода строки `\n` и перезаписывает файл, объединяя фрагменты функцией `implode()` с использованием пробела в качестве разделителя.

Замечание

Файл linux.words можно найти на компакт-диске, поставляемом вместе с книгой, в каталоге code\8.

I.8.3. Разгрузка баржи

Баржа с гравием объемом 2000 тонн должна быть разгружена максимально быстро. Баржа обычно разгружается при помощи одного или двух плавучих кранов, которые располагаются между баржей и берегом. При этом один кран может разгружать баржу с разворотом стрелы на 120° (рис. I.8.2), а два крана — с разворотом на 240°, в противном случае стрелы будут сталкиваться (рис. I.8.3).

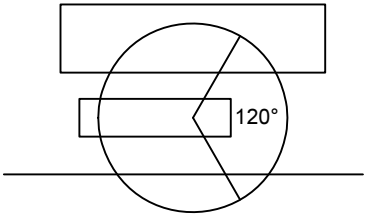


Рис. I.8.2. Разгрузка баржи одним краном

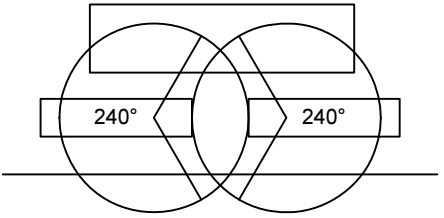


Рис. I.8.3. Разгрузка баржи двумя кранами

Доступны три вида плавучих кранов, грейферы которых могут захватывать за один раз 1,5, 3,5 и 8,5 тонн соответственно. При этом полный круг в 360° (с учетом захвата и сброса груза) краны осуществляют за 1, 2 и 5 минут соответственно (табл. I.8.1).

Таблица I.8.1. *Характеристики плавучих кранов*

Кран	Объем грейфера, тонн	Время разворота на 360°, мин
I кран	1,5	1
II кран	3,5	2
III кран	8,5	5

Подсчитайте, сколько времени потребуется на разгрузку баржи для различных комбинаций кранов: когда работает один плавучий кран каждого вида и два плавучих крана в различных комбинациях. Найдите минимальное время разгрузки баржи, если доступны все виды кранов.

I.8.4. Длительность жизни ученого

Средняя продолжительность жизни ученого составляет 70 лет. В университете работает молодой ученый в возрасте 30 лет. Ученый совет, который проходит раз в два месяца, уменьшает срок жизни ученого в среднем на 2 недели. Рыбалка, на которую ученый выбирается раз в два месяца, увеличивает срок жизни ученого на 10 дней. Подсчитайте, сколько лет проживет ученый?

I.8.5. Выгода предпринимателя

В городе Н.Н маршрутные такси часто падают с моста в реку. Однако другого транспорта для перемещения граждан на работу нет. После того, как 10-я маршрутка упала в реку, местное телевидение выяснило, что среди пострадавших не было ни одного пассажира, чей бы билет был счастливым.

Замечание

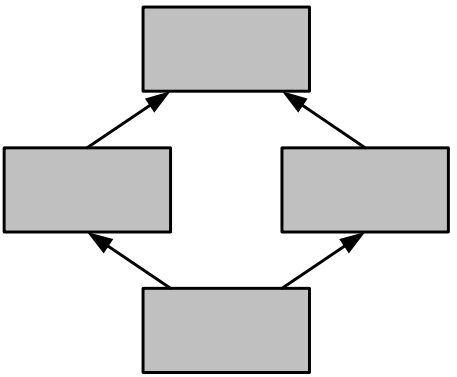
Счастливым билет считается, если сумма первых трех его номеров совпадает с суммой последних трех номеров, т. е. для билета с номером XXXYYY справедливо соотношение X + X + X = Y + Y + Y. То есть номер 361424 является счастливым.

Предприниматели решили предоставлять услугу по продаже счастливых билетов гражданам. Обязательным условиям бизнеса является отсутствие фальсификацией, за это лишают лицензии. То есть специальных допечаток счастливых билетов не производится — они изымаются из рулона с билетами естественным путем.

Общие расходы на извлечение и продажу счастливых билетов составляют 20 000 рублей в месяц. Доход с одного обычного счастливого билета — 3 рубля, с симметричного — 10 рублей. Под симметричными имеются в виду билеты

```
123 321
123 231
312 123
```

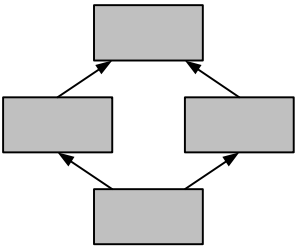
В городе в месяц продается 100 000 билетов (номера идут от 000000 до 999999). Выясните, будет ли доходным бизнес предпринимателей или они понесут убытки?



ЧАСТЬ II

Ответы

ГЛАВА II.1



Базовые конструкции языка

II.1.1. Включение заголовочных файлов

Наиболее предпочитаемым в случае библиотеки `IOstream` является первый случай, т. е. заголовочный файл стандартной библиотеки iostream.h следует заключать в угловые скобки и при этом не указывать расширение файла.

Практически с самого начала при включении заголовочных файлов при помощи конструкции препроцессора `include` действовало правило: заголовочные файлы стандартных библиотек, поставляемых вместе с компилятором, включаются при помощи угловых скобок (листинг II.1.1), а заголовочные файлы, которые создает программист, — при помощи двойных кавычек (листинг II.1.2).

Листинг II.1.1. Подключение заголовочного файла библиотеки компилятора

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

Листинг II.1.2. Подключение заголовочного файла пользовательской библиотеки

```
#include "my_const.h"
```

Большинство компиляторов даже поиск заголовочных файлов производят в зависимости от того, какие обрамляющие символы используются. Так, если используются угловые скобки, то поиск заголовочного файла сначала производится в каталогах библиотек, и лишь затем в текущем каталоге, если заголовочный файл включен при помощи обычных кавычек, то поиск производится в обратной последовательности — сначала в текущем каталоге и лишь затем в каталоге библиотек.

За годы развития компиляторов скопилось большое число заголовочных файлов, которые в различных системах имели разные расширения. Так, помимо традиционного расширения h, для системных библиотек часто применялись расширения hpp, hxx и т. п. Поэтому, когда пришло время стандартизации, Комитет по стандартизации принял решение, что для тех библиотек, которые включаются в стандарт (т. е. обязательны для реализации в любом компиляторе), при включении заголовочного файла расширение не указывать. Поэтому правильным решением будет включение стандартной библиотеки ввода/вывода `iostream` так, как это представлено в листинге II.1.3. То есть в угловых скобках без расширения.

Замечание

Для сохранения обратной совместимости большинство компиляторов наряду с новым правилом поддерживают и старую возможность включения стандартных библиотек с расширением, однако лучше придерживаться рекомендованной стандартом практики.

Листинг II.1.3. Подключение стандартной библиотеки

```
#include <iostream>
```

Это вовсе не означает, что заголовочный файл стандартной библиотеки не имеет расширения, просто добавление расширения к имени заголовочного файла теперь лежит на совести компилятора.

II.1.2. Сколько байтов занимает каждый из базовых типов?

В стандарте C++ не определяются объемы памяти, отводимой под базовые типы. Выяснить, сколько байтов отводится под тот или иной тип, можно при помощи конструкции `sizeof()`, которая принимает в качестве аргумента тип. В листинге II.1.4 последовательно перебираются все базовые типы, для каждого из которых выводится объем (в байтах), занимаемый переменными этого типа.

Замечание

При помощи `sizeof()` можно выяснять размер не только базовых типов, но и определенных программистом.

Листинг II.1.4. Определение объема памяти, занимаемой базовыми типами

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

```
int main()
{
  cout << "char "               << sizeof(char) << "\n";
  cout << "wchar_t "            << sizeof(wchar_t) << "\n";
  cout << "unsigned char "      << sizeof(unsigned char) << "\n";
  cout << "signed char "        << sizeof(signed char) << "\n";
  cout << "int "                << sizeof(int) << "\n";
  cout << "unsigned int "       << sizeof(unsigned int) << "\n";
  cout << "signed int "         << sizeof(signed int) << "\n";
  cout << "unsigned short int " << sizeof(unsigned short int) << "\n";
  cout << "signed short int "   << sizeof(signed short int) << "\n";
  cout << "long int "           << sizeof(long int) << "\n";
  cout << "signed long int "    << sizeof(signed long int) << "\n";
  cout << "unsigned long int "  << sizeof(unsigned long int) << "\n";
  cout << "float "              << sizeof(float) << "\n";
  cout << "double "             << sizeof(double) << "\n";
  cout << "logn double "        << sizeof(long double) << "\n";

  return 0;
}
```

Как видно из листинга II.1.4, для вывода результата используется вывод в поток `cout` стандартной библиотеки ввода/вывода.

Результат работы программы, представленной в листинге II.1.4, может выглядеть следующим образом:

```
char 1
wchar_t 2
unsigned char 1
signed char 1
int 4
unsigned int 4
signed int 4
unsigned short int 2
signed short int 2
long int 4
signed long int 4
unsigned long int 4
float 4
double 8
long double 10
```

Замечание

Любопытно, что размер типа `long double` в Windows составляет 10 байтов, а в Linux — 12 байтов. Это лишний раз свидетельствует, что не стоит доверять

"общепризнанным" величинам базовых типов и для создания переносимого кода лучше использовать конструкцию `sizeof()`.

Для перевода строки в стандартный поток отправляется последовательность `\n`, которая заставляет выводить следующую подстроку с новой строки. Вместо этой последовательности можно также использовать манипулятор `endl`, который включает в выходной поток символ конца строки и сбрасывает буфер потока (листинг II.1.5). Манипулятор `flush` позволяет сбросить буфер без помещения в поток символа конца строки.

Замечание

Дело в том, что символы, которые выводит программа в стандартный поток (в нашем случае стандартный поток связан с консолью), выводятся не сразу после того, как выполняется соответствующий оператор, а спустя некоторое время, когда буфер вывода будет заполнен. Это приводит к тому, что программа, выведя в консоль часть строки и начав длительные вычисления, не выведет оставшуюся часть строки, т. к. буфер не достигнет критического объема заполнения — это особенно неприятно, когда вычисления производятся часами. Манипуляторы `endl` и `flush` позволяют сбросить буфер в поток, не дожидаясь, когда это произойдет "естественным" путем. Подробнее манипуляторы рассматриваются в *главе II.7*.

Листинг II.1.5. Использование манипулятора `endl`

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

int main()
{
  cout << "sizeof(char) " << sizeof(char) << endl;
  return 0;
}
```

II.1.3. Сколько байтов занимает тип *void*?

При попытке определить тип `void` при помощи конструкции `sizeof()` компилятор возвращает ошибку "Not an allowed type" ("Неразрешенный тип") и отказывается компилировать программу. Дело в том, что компилятор не знает и не может знать, сколько байтов занимает тип `void`, т. к. ключевое слово `void` характеризует отсутствие типа.

Переменной типа `void` не существует, а вот указатель типа `void *` вполне допустим и занимает 4 байта. Тип переменной, функции и указателя сообщает компилятору и программисту, что ожидается объем памяти определенного размера, который определяется типом. Тип `void`, например, функции сообща-

ет, что никакого объема памяти выделять не предвидится, т. к. функция просто не возвращает значения. Переменная не может иметь тип `void`, т. к. переменная — это именованная область памяти. Если память отсутствует — именовать нечего. Однако тип `void *` уже вполне допустим и означает, что указатель ссылается на неизвестный объем памяти.

Замечание

Пользоваться указателями типа `void *` не рекомендуется, т. к. компилятор "не знает", на какой объем памяти ссылается указатель, и не сможет корректно удалить ее, если об этом не позаботится сам программист, осуществив привидение типа.

II.1.4. Равны ли числа?

Программа из листинга I.1.2 сообщит, что числа не равны, несмотря на то, что обоим числам присваиваются одинаковые значения 60 000. Дело в том, что под оба числа — и беззнаковое короткое целое `unsigned short int`, и знаковое короткое целое `signed short int` — отводятся по два байта, в чем можно легко убедиться при помощи программы из листинга II.1.6.

Замечание

В C++ помимо базовых типов существуют также модификаторы типов. Так, модификаторы `signed` и `unsigned` указывают, обладают переменные знаком или нет. Если переменная не обладает знаком (`unsigned`), то она может принимать только положительные значения. По умолчанию, если модификатор не указывается, считается, что объявляется переменная со знаком (`signed`). Последнее означает, что вместо `signed short int` допускается использовать краткую форму `short int`.

Листинг II.1.6. Определение размера переменных

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

int main()
{
  cout << "unsigned short int " << sizeof(unsigned short int) << "\n";
  cout << "signed short int " << sizeof(signed short int) << "\n";

  return 0;
}
```

Дело в том, что у переменных со знаком один разряд отводится для обозначения знака (положительная переменная или отрицательная), в то время как

у беззнаковых переменных все разряды отводятся под число. Таким образом, максимальные значения по абсолютной величине у знаковых и беззнаковых переменных отличаются в два раза. То есть если максимальное значение типа `unsigned short int` — 2^{16} (65 536), то у типа — 2^{16-1} (32 768). Число 16 здесь означает количество битов, которых в 2-х байтах как раз убирается 16. Таким образом, короткий целочисленный тип без знака `unsigned short int` может принимать значения от 0 до 65 636, в то время как короткий целочисленный тип со знаком может принимать значения от –32 768 до 32 767. При попытке присвоить ему значения 60 000 в листинге I.1.2 происходит переполнение и выход за границу числа. Остается выяснить, что при этом происходит. Для этого выведем содержимое переменных в консоль при помощи программы, представленной в листинге II.1.7.

Листинг II.1.7. Вывод содержимого переменных в консоль

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

int main()
{
  unsigned short int first = 60000;
  signed short int second = 60000;

  cout << "first "  << first << "\n";  // 60000
  cout << "second " << second << "\n"; // -5536

  return 0;
}
```

При выполнении кода из листинга II.1.7 выясняется, что в случае переменной со знаком (`signed short int`) выводится число –5536. Бинарное представление числа 5536 эквивалентно 00010101 10100000, в то время как бинарное представление числа 60 000 равно 11101010 01100000.

Отрицательные числа в компьютерах представлены достаточно своеобразно, для их хранения используется так называемый *дополнительный код*. То есть значение переменной инвертируется (0 заменяются на 1, а 1 на 0), и к нему добавляется единица:

`00010101 10100000` (5536)

`11101010 01011111` (инвертированное число 5536)

`00000000 00000001` (1)

`11101010 01100000` (–5536)

Как видно, последнее бинарное представление числа –5536 полностью совпадает с бинарным представлением числа 60 000. Таким образом, в бинарном представлении числа полностью эквивалентны, но C++ считает их различными, т. к. они принадлежат разным типам и имеют в связи с этим разное значение.

II.1.5. Результат сравнения

Ситуация аналогична задаче из *разд. I.1.4*: бинарное представление переменных эквивалентно, но C++ считает их значения различными из-за того, что они принадлежат разным типам. Впрочем, можно произвести расширение типа `signed short int` за счет операции приведения типа. Так программа из листинга II.1.8 сообщит о том, что переменные эквивалентны, т. к. компилятору C++ сообщается, что переменную `second` следует рассматривать как беззнаковую переменную, вполне способную вместить значение 60 000. Достигается приведение типа при помощи указания имени типа в круглых скобках перед именем переменной.

Листинг II.1.8. Приведение типа

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

int main()
{
  unsigned short first = 60000;
  short int second = -5536;
  if((unsigned short int)second == first)
  {
    cout << "Values are equvalent\n";
  }
  else cout << "Values are not equvalent\n";

  return 0;
}
```

II.1.6. Сравнение инкремента и постинкремента

При использовании операторов инкремента и постинкремента следует помнить, что оператор инкремента `++i` сначала увеличивает значение переменной `i` на единицу, а затем возвращает значение, а оператор постинкремента `i++` сначала возвращает значение переменной, а потом увеличивает значение переменной `i`. Точно так же ведут себя операторы декремента `--i` и постдекре-

мента `i--`, только вместо увеличения значения происходит его уменьшение на единицу.

Замечание

Зачастую операторы инкремента и декремента более эффективны, чем обычное сложение `i = i + 1` или вычитание `i = i - 1`. Это связано с тем, что для данных операций в процессорах часто реализуют отдельную команду. Впрочем, беспокоиться об использовании операторов инкремента и декремента вместо обычного сложения и вычитания не обязательно — современные компиляторы достаточно развиты, чтобы распознавать такие ситуации и генерировать более эффективный код.

Ответ к задаче из *разд. I.1.6* представлен в табл. II.1.1.

Таблица II.1.1. *Сравнение инкремента и постинкремента*

Сравнение	Результат
`++i == i++`	Числа равны
`i++ == ++i`	Числа не равны
`++i == ++i`	Числа не равны
`i++ == i++`	Числа не равны
`i++ == --i`	Числа равны
`i++ == i--`	Числа не равны
`i-- == ++i`	Числа равны
`i-- == i++`	Числа не равны

II.1.7. Четное или нечетное?

Самый простой способ определить четность числа — это проверить остаток деления его на 2 при помощи оператора `%`. Ответ может выглядеть следующим образом (листинг II.1.9).

Листинг II.1.9. Проверка четности числа

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

int main()
{
  int number = 0;
```

```
  cout << "Введите число ";
  cin >> number;
  // Проверяем четность числа
  if(number % 2) cout << "\nНечетное";
  else cout << "\nЧетное";

  return 0;
}
```

Если число делится на 2 без остатка, это четное число, в этом случае `number % 2` возвращает значение 0, что воспринимается как `false`. Если число `number` делится на 2 с остатком, то этот остаток воспринимается оператором `if` как `true`.

II.1.8. Имя программы

Получить имя программы можно через параметры функции `main()`. Согласно стандарту C++ переносимыми являются лишь два определения функции `main()`, играющей в программе роль точки входа. Первое определение — это функция `main()` без параметров, которая использовалась нами до этого момента (листинг II.1.10).

Листинг II.1.10. Определение функции `main()` без параметров

```
int main()
{
  // ...
}
```

Второе стандартное определение функции `main()` позволяет передать функции два параметра: целочисленный *`argc`* — число аргументов программы и массив строк *`argv`*, через который передаются сами параметры (листинг II.1.11).

Замечание

Никакие другие формы функции `main()` не являются переносимыми.

Листинг II.1.11. Определение функции `main()` с параметрами

```
int main(int argc, char *argv[])
{
  // ...
}
```

При приеме параметров следует помнить, что их всегда на единицу больше, чем фактическое число параметров, т. к. в качестве первого аргумента передается имя программы. Принимая во внимание этот факт, программа, выводящая свое имя, может выглядеть так, как это представлено в листинге II.1.12.

Замечание

Это достаточно распространенная практика, когда поведение утилиты зависит от ее имени. Пользователь, для того чтобы не вводить каждый раз дополнительные параметры, может переименовать программу, которая в зависимости от того, как она названа, может менять режим своей работы.

Листинг II.1.12. Вывод имени программы

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

int main(int argc, char *argv[])
{
  cout << argv[0] << "\n";
}
```

II.1.9. Чем отличается *switch* от конструкции *if-else-if*?

Оператор `switch` — это своеобразный ограниченный оператор `goto`. Получив значение, оператор `switch` сравнивает его со значениями в конструкции `case`, и если совпадение найдено, то начинается выполнение операторов с текущего оператора `case`. Выполнение происходит до ближайшего оператора `break`. Если не указывать оператор `break`, то будут выполнены операторы из нижележащих конструкций `case` (листинг II.1.13).

Листинг II.1.13. Использование оператора `case`

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

int main()
{
  int number = 0;

  cout << "Input the nuber ";
  cin >> number;
```

```
  switch(number)
  {
    case 1:
      cout << "\nfirst\n";
    case 2:
      cout << "\nsecond\n";
    case 3:
      cout << "\nthird\n";
    case 4:
      cout << "\nfourth\n";
    case 5:
      cout << "\nfifth\n";
      break;
    default :
      cout << "\nInput a number between 1 and 5\n";
  }

  return 0;
}
```

При вводе 1 в ответ на запрос числа программа из листинга II.1.13 выведет последовательно все ответы:

```
first
second
third
fourth
fifth
```

Блоки в конструкции `if-else-if` не зависят друг от друга и выполняются только в том случае, если выполняется очередное условие.

С другой стороны, оператор `switch` позволяет осуществлять сравнение только по равенству, и в качестве одного из операндов всегда выступает только одна переменная, в то время как `if-else-if` позволяет осуществлять сравнение не только по равенству, но и по неравенству с применением операторов <, >, =<, >=, !=, а в качестве операндов сравнения в каждом из условий могут выступать каждый раз разные переменные и константы.

Другой отличительной чертой оператора `switch` является тот факт, что он не может иметь два одинаковых `case`-условия, в то время как конструкция `if-else-if` может иметь любое число повторяющихся условий.

Замечание

Оператор `switch` более эффективен по производительности, чем конструкция `if-else-if`.

Оператор switch оправдан, когда производится множество сравнений по равенству с одной-единственной переменной, как это происходит в условии задачи.

II.1.10. Вывод случайного числа символов

Для получения случайного числа удобно использовать библиотечную функцию rand(), которая становится доступной после включения заголовочного файла <cstdlib>. Функция rand() имеет следующий синтаксис:

```
int rand(void);
```

При каждом вызове функция возвращает случайное число от 0 до RAND_MAX (как правило, 32 767). Правда, каждый раз функция генерирует одну и ту же последовательность случайных чисел. Для того чтобы убедиться в этом, можно использовать программу из листинга II.1.14.

Листинг II.1.14. Вывод случайной последовательности

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

int main()
{
  for(int i = 0; i < 10; i++) cout << rand() << "\n";

  return 0;
}
```

Если при запуске программы из листинга II.1.14 результатом выполнения является последовательность:

```
130
10982
1090
11656
7117
17595
6415
22948
31126
9004
```

то при последующем запуске программы будет получена точно такая же последовательность случайных чисел. Для того чтобы последовательность каж-

дый раз была разная, используется дополнительная функция `srand()`, которая устанавливает новую стартовую точку и имеет следующий синтаксис:

```
void srand(unsigned int seed);
```

В качестве единственного параметра *seed* передается число инициализации, в качестве которого часто выступает временная метка, которую получают при помощи функции `time()` из библиотеки `<ctime>`. С учетом всего вышесказанного, программу из листинга II.1.14 можно переписать так, как это представлено в листинге II.1.15.

Замечание

Впрочем, если программу из листинга II.1.15 запускать чаще, чем системный таймер изменяет временную метку, одинаковые случайные последовательности все равно будут получаться.

Листинг II.1.15. Использование функции `srand()`

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

int main()
{
  srand(time(NULL));
  for(int i = 0; i < 10; i++) cout << rand() << "\n";

  return 0;
}
```

Теперь необходимо получить случайные числа не в диапазоне от 0 до `RAND_MAX`, а от 0 до 10. Для этого достаточно разделить результат на величину `RAND_MAX` и умножить на 10. Однако следует учитывать, что результатом деления двух целых чисел является целое число, а дробная часть отбрасывается. Поэтому для того чтобы каждый раз не получался 0, следует привести результат деления сначала к типу `float`, а затем после умножения на 10, чтобы получить целое число, следует привести результат к целому типу `int` (листинг II.1.16).

Листинг II.1.16. Получение случайных чисел от 0 до 10

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

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

int main()
{
  srand(time(NULL));
  for(int i = 0; i <= int((float)rand()/RAND_MAX*10); i++) cout << "*";
  cout << "\n";

  return 0;
}
```

II.1.11. Вывод четных чисел

Для реализации решения этой задачи удобно воспользоваться циклом `for` и оператором `continue`, который позволяет пропускать текущий цикл. Решение может выглядеть так, как это представлено в листинге II.1.17.

Листинг II.1.17. Вывод четных чисел до заданного

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

int main()
{
  int number = 0;

  cout << "Input the number ";
  cin >> number;

  for(int i = 0; i < number; i++)
  {
    if(i % 2) continue;
    cout << i << "\n";
  }

  return 0;
}
```

Как вариант, вместо оператора `continue` можно просто инвертировать условие блока `if` (листинг II.1.18).

Листинг II.1.18. Альтернативное решение

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

```
int main()
{
  int number = 0;

  cout << "Input the number ";
  cin >> number;

  for(int i = 0; i < number; i++)
  {
    if(!(i % 2)) cout << i << "\n";
  }

  return 0;
}
```

II.1.12. Вывод всех видимых ASCII-символов

Решение данной задачи может выглядеть так, как это представлено в листинге II.1.19.

Листинг II.1.19. Вывод всех видимых ASCII-символов

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

int main()
{
  unsigned char symbol = 32;

  while(symbol)
  {
    cout << symbol;
    symbol++;
  }

  return 0;
}
```

Данное решение основывается на том факте, что переменная типа `unsigned char` может принимать максимум 255 значений, именно столько входит в ASCII-кодировку. При достижении переменной `symbol` предела (255) ее значение сбрасывается в 0 и цикл `while` завершает свою работу. Видимые символы кодировки ASCII начинаются с пробела, который соответствует

32-му символу, именно поэтому переменная `symbol` инициируется этим значением.

II.1.13. Поиск простых чисел

Для решения данной задачи следует воспользоваться двойным циклом (листинг II.1.20). Управление выводом простых чисел осуществляется при помощи булевой переменной `flag`, равенство которой `true` означает, что найдено очередное простое число.

Замечание

Это не самое эффективное решение поиска простых чисел, здесь просто для каждого из чисел проверяется, делится ли оно на числа, которые ему предшествуют.

Листинг II.1.20. Поиск простых чисел

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

int main()
{
  int number = 0;
  bool flag = false;

  cout << "Введите число ";
  cin >> number;

  for(int i = 2; i < number; i++)
  {
    for(int j = 2; j < i; j++)
    {
      if(i % j) continue;
      else
      {
        flag = true;
        break;
      }
    }
    if(!flag) cout << i << " ";
    flag = false;
  }

  return 0;
}
```

Так, если в качестве верхней границы ввести число 100, то результат работы программы из листинга II.1.20 может выглядеть следующим образом:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

II.1.14. Упаковка цикла *for*

Решение задачи основано на том факте, что цикл `for` является чрезвычайно гибким и позволяет использовать несколько инициализирующих выражений и несколько выражений, которые выполняются на каждой итерации цикла. Если применяются несколько выражений, они должны быть отделены друг от друга запятой. Так, листинг 1.7 может быть переписан следующим образом (листинг II.1.21).

Замечание

В реальной практике не рекомендуется прибегать к подобному приему, т. к. ни к чему, кроме снижения читабельности кода и возникновению трудно улавливаемых ошибок, это не приведет. Существует годами устоявшаяся форма использования оператора `for`, и именно ее и следует придерживаться. Впрочем, о возможностях цикла `for` забывать не следует хотя бы из-за того, что кому-нибудь, чей код вам придется сопровождать, придет в голову воспользоваться этими возможностями.

Листинг II.1.21. Упаковка цикла `for`

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

int main()
{
  int number, j, i;
  for(number = 10, j = number, i = 0;
      i <= j;
      cout << i << " " << j << "\n", i++, j--);
  return 0;
}
```

II.1.15. Преобразование десятичного числа в двоичное

Остановимся подробнее на алгоритме преобразования. Обычно поступают следующим образом: десятичное число последовательно делят на 2, если число делится без остатка — записывают 0, если число не делится с остат-

ком — записывается 1. Деление производится до тех пор, пока не будет получен остаток, меньший двух.

```
10 / 2 = 5 (0)
5 / 2 = 2 (1)
2 / 2 = 1 (0)
```

После этого полученная последовательность записывается с дополнением в конец цифры 1:

```
010 => 0101
```

Полученная последовательность переворачивается

```
0101 => 1010
```

Для того чтобы лучше понять алгоритм, рассмотрим также преобразование числа 137 в двоичное представление.

```
137 / 2 = 68 (1)
68 / 2 = 34 (0)
34 / 2 = 17 (0)
17 / 2 = 8 (1)
8 / 2 = 4 (0)
4 / 2 = 2 (0)
2 / 2 = 1 (0)
1001000 => 10010001
10010001 = > 10001001
```

Реализуем данный алгоритм на C++. Для этого полученные в результате деления цифры будем складывать в строку `binary`. Строка занимает 80 позиций. Будем считать, что вводимое число в бинарном представлении должно вмещать не более 80 позиций. Вычисления удобно производить в цикле `do...while` (листинг II.1.22).

Листинг II.1.22. Преобразование десятичного числа в бинарное

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

int main()
{
  int number = 0;
  int counter = 0;
  cout << "Введите число ";
  cin >> number;
  char binary[80];
```

```
  do
  {
    if(number % 2) binary[counter++] = '1';
    else binary[counter++] = '0';
    number = number / 2;
  } while(number);

  // Завершаем строку
  binary[counter++] = '\0';
  // Выводим строку в обратном порядке
  for(int i = strlen(binary); i; i--) cout << binary[i - 1];
  cout <<  endl;

  return 0;
}
```

После того, как строка `binary` заполняется нулями и единицами в цикле `do...while()`, необходимо завершить ее нулевым символом `\0`, иначе длина строки будет вычисляться неправильно, т. к. длина строки определяется по первому нулевому символу. В завершение программы в цикле `for` перебираются символы, из которых состоит строка `binary`, с целью вывода их в обратном порядке.

Программа, представленная в листинге II.1.22, не является единственно возможной реализацией. Более того, это скорее реализация того, как бы действовал человек в ситуации, когда ему необходимо преобразовать десятичное число в двоичное. Компьютеры, для которых двоичное представление числа является родным, поступают немного по-другому: они не используют дорогие в плане затрат процессорного времени операции деления `/` и взятия остатка `%`, предпочитая им битовые операторы, которые реализованы на уровне процессора и выполняются очень быстро.

Среди битовых операторов выделяют `&` (И), `|` (ИЛИ), `~` (НЕ), `>>` (сдвиг вправо) и `<<` (сдвиг влево). Логика первых трех операторов представлена в табл. II.1.2.

Таблица II.1.2. *Логика битовых операторов*

x	y	x & y	x | y	~x
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

При использовании битовых операторов следует помнить, что все операции выполняются над числами в двоичном представлении.

Пусть имеются два числа: 6 и 10. Применение к ним оператора & дает 2 (листинг II.1.23).

Замечание

При выводе в стандартный поток результат побитовых вычислений следует брать в круглые скобки (листинг II.1.23), чтобы избежать конфликта с перегруженным оператором <<.

Листинг II.1.23. Использование оператора &

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

int main()
{
  int x = 6;
  int y = 10;

  cout << (x & y) << endl;

  return 0;
}
```

Этот, казалось бы, парадоксальный результат можно легко понять, если перевести числа 6 и 10 в двоичное представление, в котором они равны 0110 и 1010 соответственно. Сложение разрядов при использовании оператора & происходит согласно правилам, представленным в табл. II.1.2. Учитывая их, можно легко получить результат, который равен 0010 (в десятичной системе — 2).

```
0110 (6)
1010 (10)
0010 (2)
```

Рассмотрим более сложный пример — применение оператора & для чисел 113 и 45. Для наглядности представим складываемые числа в двоичном представлении

```
1110001 (113)
0101101 (45)
0100001 (33)
```

Для оператора | применение к числам 6 и 10 (`6 | 10`) может выглядеть следующим образом:

```
0110 (6)
1010 (10)
1110 (14)
```

Для второй пары чисел (`113 | 45`):

```
1110001 (113)
0101101 (45)
1111101 (125)
```

Оператор ~ является побитовой инверсией и заменяет все 0 на 1 и наоборот. В листинге II.1.24 демонстрируется использование данного оператора.

Листинг II.1.24. Использование оператора ~

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

int main()
{
  cout << ~45 << endl; // -46
  cout << ~92 << endl; // -93
  return 0;
}
```

Такие результаты являются следствием того, что все побитовые операции производятся над целыми числами — инверсия их приводит к тому, что число записывается де-факто в дополнительном коде (не хватает только единицы). По сути, смена знака числа на минус сводится к побитовой инверсии и прибавлению единицы.

В двоичных числах ведущие нули обычно не указывают, но т. к. операция ~ производит их инвертирование — они заменяются единицами, приводя к отрицательным значениям. Так операция `~45` в двоичном представлении выглядит следующим образом:

```
00000000000000000000000000101101 (45)
11111111111111111111111111010010 (-46)
```

Для `~92`:

```
00000000000000000000000001011100 (92)
11111111111111111111111110100011 (-92)
```

Оператор << сдвигает влево все биты первого операнда на число позиций, указанных во втором операнде (листинг II.1.25). Сдвиг на отрицательное значение приводит к нулевому значению.

Листинг II.1.25. Использование оператора <<

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

int main()
{

  cout << (6 << 1) << endl; // 12
  cout << (6 << 2) << endl; // 24
  cout << (6 << 10) << endl; // 6144
  cout << (6 << -1) << endl; // 0
  cout << (6 << 31) << endl; // 0
  cout << (6 << 30) << endl; // -2147483648

  return 0;
}
```

В двоичном представлении эти операции выглядят следующим образом:

- `6 << 1`:

  ```
  0110 (6)
  1100 (12)
  ```

- `6 << 2`:

  ```
  00110 (6)
  11000 (24)
  ```

- `6 << 10`:

  ```
  0000000000110 (6)
  1100000000000 (6144)
  ```

Как видно из листинга II.1.25, к нулевому результату приводит не только сдвиг на отрицательное число позиций, но и сдвиг на 31 позицию, т. к. это приводит к тому, что все 1 уходят за границу 32-битного числа и все разряды принимают значение 0.

```
00000000000000000000000000000000 (0)
```

Оператор >> сдвигает вправо все биты первого операнда на число позиций, указанных во втором операнде. Сдвиг на отрицательное значение приводит к нулевому значению (листинг II.1.26).

Листинг II.1.26. Использование оператора >>

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

int main()
{

  cout << (6144 >> 10) << endl; // 6
  cout << (45 >> 2) << endl;    // 11
  cout << (45 > 2) << endl;     // 1

  return 0;
}
```

В двоичном представлении эти операции выглядят следующим образом:

- `6144 >> 10`:

```
1100000000000 (6144)
0000000000110 (6)
```

- `45 >> 2`:

```
101101 (45)
001011 (11)
```

Как видно из листинга II.1.26, при использовании операторов сдвига очень легко ошибиться, указав только один знак >. В этом случае, компилятор C++ вернет либо 1 (истина), либо 0 (ложь). За подобными опечатками нужно следить очень внимательно, т. к. компилятор не предупреждает о такой опечатке, в то же время результат будет отличаться от ожидаемого.

Преобразование десятичного числа в двоичное в стиле "компьютерной логики" может выглядеть так, как это представлено в листинге II.1.27.

Замечание

Несмотря на то, что программа из листинга II.1.27 выполняется быстрее кода из листинга II.1.22, побитовые операторы следует использовать только как крайнее средство. Зачастую программа выполняется быстро и без лишней оптимизации, однако использование "хитрых" и неочевидных алгоритмов приводит к тому, что сопровождать ее становится очень сложно. В результате сэкономленные микросекунды, которые не отражаются на общем времени выполнения программы, оборачиваются часами работы сопровождающего программу программиста.

Листинг II.1.27. Альтернативное решение задачи

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

int main()
{
  int number = 0;
  int counter = 0;
  cout << "Введите число ";
  cin >> number;
  char binary[80];

  do
  {
    if(number & 1) binary[counter++] = '1';
    else binary[counter++] = '0';
    number = number >> 1;
  } while(number);

  // Завершаем строку
  binary[counter++] = '\0';
  // Выводим строку в обратном порядке
  for(int i = strlen(binary); i; i--) cout << binary[i - 1];
  cout <<  endl;

  return 0;
}
```

Эффект решения основан на том, что компьютер хранит десятичные числа в двоичном формате, нам следует только прочитать его. Мы применяем оператор `&`, который при вычислении пересечения введенного числа и 1 возвращает 1, если последний разряд введенного числа равен 1, и 0, если последний разряд равен 0. Затем мы сдвигаем вправо число `number` на один разряд. Так для числа 137 алгоритм может выглядеть следующим образом:

```
10001001 & 1 = 1
01000100 & 1 = 0
00100010 & 1 = 0
00010001 & 1 = 1
00001000 & 1 = 0
00000100 & 1 = 0
00000010 & 1 = 0
00000001 & 1 = 1
```

Последний сдвиг приводит к тому, что в переменной `number` остается лишь 0, и цикл `while()` прекращает работу по условию.

II.1.16. Преобразование двоичного числа в десятичное

Для преобразования числа из двоичной системы счисления в десятичную следует помнить, что позиции чисел пронумерованы от 0 до $n-1$, где n — количество цифр в числе. При этом цифры нумеруются справа налево, т. е. для числа 6583 цифра 3 занимает нулевую позицию, 8 — первую позицию, 5 — вторую позицию, а 6 — третью позицию. Для десятичного числа 6583 справедлива формула

$$6 \cdot 10^3 + 5 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0 = 6583.$$

Точно такая же формула применяется для преобразования двоичных чисел в десятичную систему счисления, только вместо основания 10 используется основание 2. То есть двоичное число 10100010 можно преобразовать в десятичное по формуле:

$$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 10^0 = 162.$$

Программа, реализующая данный алгоритм, может выглядеть так, как это представлено в листинге II.1.28.

Листинг II.1.28. Преобразование двоичного числа в десятичное

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

int main()
{
  int number = 0;
  char binary[80];
  int multiplier = 1;

  cout << "Введите число ";
  cin >> binary;

  for(int i = strlen(binary); i; i--, multiplier *= 2)
  {
    if(binary[i - 1] == '1') number += multiplier;
    else if(binary[i - 1] != '0')
    {
      cout << "Неверный формат бинарного числа";
```

```
      return 1;
    }
  }

  cout << number << endl;

  return 0;
}
```

Бинарная строка принимается от пользователя в переменную `binary`, после чего в цикле `for` перебираются символы, из которых состоит строка. Если текущий символ равен 1, то результирующей переменной `number` прибавляется значение множителя `multiplier`, который на каждой итерации цикла умножается на два и играет роль множителя 2^x. Если текущая позиция равна нулю, то никаких действий осуществлять не нужно, т. к. такая позиция не влияет на конечный результат. Если символ в строке `binary` не равен ни '0', ни '1' — это означает, что пользователь ошибся при вводе бинарного числа и следует прекратить вычисления (при этом пользователю выдается сообщение "Неверный формат бинарного числа").

Замечание

При создании промышленного кода удобнее воспользоваться библиотечной функцией `atoi()`, а не писать собственный код.

II.1.17. Преобразование десятичного числа в восьмеричное

Для того чтобы преобразовать десятичное число в восьмеричное, его обычно переводят в двоичную форму. Каждые три бинарные позиции — это восьмеричное число. Например, чтобы преобразовать число 137 в восьмеричное число, его можно представить в бинарной форме

```
137 => 10001001
10001001 => 10 001 001
010 001 001 => 211
```

Или, например, преобразование числа 423 423 можно представить следующим образом

```
423423 => 1100111010111111111
1100111010111111111 => 1 100 111 010 111 111 111
1 100 111 010 111 111 111 = > 1472777
```

Если остатку слева не хватает нулей, как в рассмотренном выше случае, то слева добавляются недостающие нули. Преобразование двоичных триад в восьмеричные числа происходит согласно табл. II.1.3.

Таблица II.1.3. *Соответствие двоичных чисел восьмеричным*

Восьмеричное число	Двоичное число
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Впрочем, получить восьмеричное число из двоичного *XYZ* достаточно просто по формуле

$$X \cdot 2^2 + Y \cdot 2^1 + Z \cdot 2^0.$$

Реализуем описанный алгоритм в виде программы (листинг II.1.29).

Листинг II.1.29. Преобразование десятичного числа в восьмеричное

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

int main()
{
  int number = 0;
  int counter = 0;
  int k, i, j;
  char binary[80], convert[80], triada[4], oct[80];

  cout << "Введите число ";
  cin >> number;

  // Преобразуем десятичное число в двоичное
  do
  {
    if(number % 2) binary[counter++] = '1';
```

```
    else binary[counter++] = '0';
    number = number / 2;
  } while(number);
  binary[counter++] = '\0';

  // Вычисляем длину полученной бинарной строки
  int lenght = strlen(binary);
  // Дополняем строку convert недостающими нулями,
  // чтобы ее можно было разбить на триады
  for(j = 0; j < 3 - lenght % 3; j++)
  {
    convert[j] = '0';
  }
  // Переворачиваем бинарную строку,
  // занося ее содержимое в переменную convert
  for(i = lenght; i > 0; i--)
  {
    convert[j++] = binary[i - 1];
  }
  convert[j++] = '\0';

  // Разбиваем бинарное число на триады,
  // преобразуя его в восьмеричное число
  for(i = 0, k = 0; i < strlen(convert); i += 3, k++)
  {
    // Собираем триаду посимвольно
    triada[0] = convert[i];
    triada[1] = convert[i + 1];
    triada[2] = convert[i + 2];
    triada[3] = '\0';
    // Сопоставляем полученную триаду и восьмеричную последовательность
    if(!strcmp(triada, "000")) oct[k] = '0';
    else if(!strcmp(triada, "001")) oct[k] = '1';
    else if(!strcmp(triada, "010")) oct[k] = '2';
    else if(!strcmp(triada, "011")) oct[k] = '3';
    else if(!strcmp(triada, "100")) oct[k] = '4';
    else if(!strcmp(triada, "101")) oct[k] = '5';
    else if(!strcmp(triada, "110")) oct[k] = '6';
    else oct[k] = '7';
  }
  oct[k++] = '\0';
  cout << oct <<  endl;

  return 0;
}
```

Как видно из листинга II.1.29, для преобразования программа запрашивает у пользователя десятичное число, которое помещается в переменную `number`, после чего оно преобразуется в двоичное число `binary` (вернее, в перевернутую последовательность двоичного числа). Перевернутая последовательность хранится в строке `convert`, которая выравнивается слева недостающими нулями прежде, чем в нее помещается содержимое из строки `binary`. После этого строка `convert` разбивается на триады, хранящиеся в переменной `triada`, которые сравниваются с табличными значениями бинарного представления восьмеричных чисел. По результату сравнения в массив `oct` записывается очередное восьмеричное число. Для того чтобы библиотечные функции считали массивы `binary`, `convert`, `triada` и `oct` строками, они завершаются нулевым признаком строки `\0`.

Массив `for` позволяет работать с переменными цикла, которые объявляются вне конструкции `for` (листинг II.1.30).

Листинг II.1.30. Объявление переменной цикла вне конструкции `for`

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

int main()
{
  int i = 0;

  for(i = 0; i < 10; i++);

  cout << i << endl;

  return 0;
}
```

При этом значение переменной, которое устанавливается циклом, доступно также вне цикла. В рассмотренном выше примере преобразования десятичного числа в двоичное мы пользовались именно этим приемом для того, чтобы завершить строку нулевым символом.

Помимо этого, допускается объявление переменной в конструкции `for`. Так, программа в листинге II.1.31 не будет откомпилирована, поскольку переменная `i` объявлена в цикле `for` и не может быть видна вне его.

Листинг II.1.31. Объявление переменной цикла в конструкции `for`

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

```
int main()
{
  for(int i = 0; i < 10; i++);

  // Ошибка, переменная i находится
  // за пределами зоны видимости цикла for
  cout << i << endl;

  return 0;
}
```

В реальной практике, для того чтобы получить десятичное число в восьмеричном представлении, специальных преобразований не осуществляют. Следует только "попросить" библиотеку ввода/вывода вывести число в бинарном формате. Для этого используется манипулятор `oct`. Поэтому листинг II.1.29 можно переписать так, как это представлено в листинге II.1.32. Можно также воспользоваться функцией `itoa()` из библиотеки `<stdlib>`.

Замечание

При создании промышленного кода лучше использовать средства стандартной библиотеки, а не изобретать велосипед. Дело в том, что в стандартной библиотеке используется эффективный и выверенный код, создавая аналоги стандартных функций, ничего, кроме появления лишних ошибок, не добиться. Современные условия таковы, что ценность приобретает компактный и понятный код, в котором содержится меньшее количество ошибок и который проще сопровождать, чем пусть более эффективный, но громоздкий и запутанный код. Время программиста, потраченное на разработку и сопровождение программы, стоит дороже, чем компьютерное время, затраченное на выполнение кода.

Листинг II.1.32. Использование флага форматирования `oct`

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

int main()
{
  int number = 0;

  cout << "Введите число ";
  cin >> number;

  cout << oct << number << endl;

  return 0;
}
```

II.1.18. Преобразование восьмеричного числа в десятичное

По аналогии с задачей из *разд. 1.16*, в которой рассматривалось преобразование двоичного числа в десятичное, для восьмеричного числа 1472777 справедлива следующая формула преобразования в десятичную систему счисления:

$$1 \cdot 8^6 + 4 \cdot 8^5 + 7 \cdot 8^4 + 2 \cdot 8^3 + 7 \cdot 8^2 + 7 \cdot 8^1 + 7 \cdot 8^0 = 423\,423.$$

Программа, реализующая данный алгоритм, может выглядеть так, как это представлено в листинге II.1.33.

Листинг II.1.33. Преобразование восьмеричного числа в десятичное

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

int main()
{
  int number = 0, mul = 0;
  char oct[80];
  int multiplier = 1;

  cout << "Введите число ";
  cin >> oct;

  for(int i = strlen(oct); i; i--, multiplier *= 8)
  {
    switch(oct[i - 1])
    {
      case '0':
        mul = 0;
        break;
      case '1':
        mul = 1;
        break;
      case '2':
        mul = 2;
        break;
      case '3':
        mul = 3;
        break;
      case '4':
        mul = 4;
        break;
```

```
      case '5':
        mul = 5;
        break;
      case '6':
        mul = 6;
        break;
      case '7':
        mul = 7;
        break;
      default :
        cout << "Неверный формат восьмеричного числа";
        return 1;
    }
    number += mul*multiplier;
  }

  cout << number << endl;

  return 0;
}
```

Как видно из листинга II.1.33, строковое представление чисел преобразуется в числовое, в противном случае умножение на символ `oct[i-1]` будет приводить не к умножению на текущее число, а на значение ASCII-кода числа. Для того чтобы этого избежать, используется оператор `switch`. В большинстве случаев это наиболее универсальное решение, однако оно выглядит несколько громоздким. Если известно, что работа программы будет осуществляться в ASCII-кодировке, можно сократить программу, воспользовавшись знанием того факта, что ASCII-код числа 0 равен 48 и далее цифры располагаются по возрастающей. Сокращенная версия листинга II.1.33 может выглядеть так, как это представлено в листинге II.1.34.

Замечание

Подход, демонстрирующийся в листинге II.1.34, часто характеризуется как "хакерский" или "грязный" — позволяя сократить код, он, тем не менее, приводит к появлению "магических констант", значение которых понять нелегко, и расшифровка кода требует длительного времени программиста. Такой подход не рекомендуется применять в промышленном программировании.

Листинг II.1.34. Учет ASCII-кодов

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

```
int main()
{
  int number = 0, mul = 0;
  char oct[80];
  int multiplier = 1;

  cout << "Введите число ";
  cin >> oct;

  for(int i = strlen(oct); i; i--, multiplier *= 8)
  {
    mul = oct[i - 1] - 48;
    if(mul < 0 || mul > 9)
    {
        cout << "Неверный формат восьмеричного числа";
        return 1;
    }
    number += mul*multiplier;
  }

  cout << number << endl;

  return 0;
}
```

Стандартная библиотека ввода/вывода позволяет не только преобразовать десятичное число в восьмеричное, но также решить и обратную задачу. Для этого, помимо ранее рассмотренного манипулятора `oct`, применяется флаг форматирования `dec`, сообщающий библиотеке ввода/вывода о необходимости выводить числа в десятичном формате (листинг II.1.35).

Листинг II.1.35. Использование флагов форматирования `oct` и `dec`

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

int main()
{
  int number = 0;

  cout << "Введите число ";
  cin >> oct >> number;
  cout << dec << number << endl;

  return 0;
}
```

II.1.19. Преобразование десятичного числа в шестнадцатеричное

Для того чтобы преобразовать десятичное число в шестнадцатеричное, его обычно переводят в двоичную форму и разбивают на блоки, содержащие 4 позиции бинарного числа, которые соответствуют одной позиции шестнадцатеричного числа.

Замечание

Так как вместо 10 цифр 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9 шестнадцатеричные числа используют 16 цифр, последовательность из 10 цифр дополняют 6-ю буквами A, B, C, D, E и F, которые соответствуют 10, 11, 12, 13, 14 и 15.

Например, чтобы перевести число 589, можно использовать следующие преобразования:

```
589 => 1001001101
1001001101 => 10 0100 1101
0010 0100 1101 => 24D
```

Если остатку слева не хватает нулей до четырех позиций, как в рассмотренном выше случае, то слева добавляются недостающие нули (`10 => 0010`). Преобразование двоичных последовательностей в шестнадцатеричные числа происходит согласно табл. II.1.4.

Таблица II.1.4. *Соответствие двоичных чисел шестнадцатеричным*

Шестнадцатеричное число	Двоичное число
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
F	1010
D	1011

Таблица II.1.4 *(окончание)*

Шестнадцатеричное число	Двоичное число
C	1100
D	1101
E	1110
F	1111

Соответствие между шестнадцатеричными и двоичными числами *XYZW* из табл. II.1.4 устанавливается при помощи формулы

$$X \cdot 10^3 + Y \cdot 10^2 + Z \cdot 10^1 + W \cdot 10^0.$$

Реализация рассмотренного алгоритма может выглядеть так, как это представлено в листинге II.1.36.

Листинг II.1.36. Преобразование десятичного числа в шестнадцатеричное

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

int main()
{
  int number = 0;
  int counter = 0;
  int k, i, j;
  char binary[80], convert[80], hex[80];

  cout << "Введите число ";
  cin >> number;

  // Преобразуем десятичное число в двоичное
  do
  {
    if(number % 2) binary[counter++] = '1';
    else binary[counter++] = '0';
    number = number / 2;
  } while(number);
  binary[counter++] = '\0';

  // Вычисляем длину полученной бинарной строки
  int lenght = strlen(binary);
```

```
// Дополняем строку convert недостающими нулями,
// чтобы ее можно было разбить на триады
for(j = 0; j < 4 - lenght % 4; j++)
{
  convert[j] = '0';
}
// Переворачиваем бинарную строку,
// занося ее содержимое в переменную convert
for(i = lenght; i > 0; i--)
{
  convert[j++] = binary[i - 1];
}
convert[j++] = '\0';

// Разбиваем бинарное число на четверки,
// преобразуя его в шестнадцатеричное число
for(i = 0, k = 0; i < strlen(convert); i += 4, k++)
{
  // Преобразуем последовательность из 4-х бинарных
  // символов в десятичное число
  number = 0;
  if(convert[i + 3] == '1') number += 1;
  if(convert[i + 2] == '1') number += 2;
  if(convert[i + 1] == '1') number += 2*2;
  if(convert[i]     == '1') number += 2*2*2;
  // Сопоставляем бинарное число шестнадцатеричному
  switch(number)
  {
    case 0:
      hex[k] = '0';
      break;
    case 1:
      hex[k] = '1';
      break;
    case 2:
      hex[k] = '2';
      break;
    case 3:
      hex[k] = '3';
      break;
    case 4:
      hex[k] = '4';
      break;
```

```
      case 5:
        hex[k] = '5';
        break;
      case 6:
        hex[k] = '6';
        break;
      case 7:
        hex[k] = '7';
        break;
      case 8:
        hex[k] = '8';
        break;
      case 9:
        hex[k] = '9';
        break;
      case 10:
        hex[k] = 'A';
        break;
      case 11:
        hex[k] = 'B';
        break;
      case 12:
        hex[k] = 'C';
        break;
      case 13:
        hex[k] = 'D';
        break;
      case 14:
        hex[k] = 'E';
        break;
      case 15:
        hex[k] = 'F';
        break;
    }
  }
  hex[k++] = '\0';
  cout << hex <<  endl;

  return 0;
}
```

Как видно из листинга II.1.36, бинарные последовательности из четырех символов преобразуются сначала в десятичное число, которое обрабатывается в

операторе `switch`. Для того чтобы сократить код программы, можно воспользоваться тем фактом, что ASCII-код символа '0' равен 48, после которого по возрастающей следуют все остальные цифры, а ASCII-код символа 'A' равен 65, после которого следуют символы 'B', 'C', 'D', 'E' и 'F'. Поэтому оператор `switch` в листинге II.1.36 можно заменить блоком `if` (листинг II.1.37).

Листинг II.1.37. Альтернативное сопоставление десятичных чисел шестнадцатеричным

```
...
if(number >= 0 && number <= 9) hex[k] = number + 48;
else if(number >= 10 && number <= 15) hex[k] = 65 + number - 10;
...
```

С помощью библиотеки ввода/вывода задачу преобразования десятичного числа в шестнадцатеричное можно решить посредством флага форматирования `hex` (листинг II.1.38).

Листинг II.1.38. Использование флага форматирования `hex`

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

int main()
{
  int number = 0;

  cout << "Введите число ";
  cin >> number;

  cout << hex << number << endl;

  return 0;
}
```

II.1.20. Преобразование шестнадцатеричного числа в десятичное

По аналогии с задачей из *разд. I.1.16*, в которой рассматривалось преобразование двоичного числа в десятичное, для шестнадцатеричного числа D65C справедлива следующая формула преобразования в десятичную систему счисления:

$$13 \cdot 16^3 + 6 \cdot 16^2 + 5 \cdot 16^1 + 12 \cdot 16^0 = 54\,876.$$

Для того чтобы сократить объем программы, в решении, представленном в листинге II.1.39, сразу используется прием, основанный на ASCII-кодах символов. Здесь, так же как и в предыдущих примерах, для хранения числа, введенного пользователем, применяется строковый массив `hexadecimal`.

Листинг II.1.39. Преобразование шестнадцатеричного числа в десятичное

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

int main()
{
  int number = 0, mul = 0;
  char hexadecimal[80];
  int multiplier = 1;

  cout << "Введите число ";
  cin >> hexadecimal;

  for(int i = strlen(hexadecimal); i; i--, multiplier *= 16)
  {
    mul = hexadecimal[i - 1] - 48;
    if(mul > 0 && mul < 10)
    {
      number += mul*multiplier;
    }
    else
    {
      // Преобразуем символ к верхнему регистру
      mul = toupper(hexadecimal[i - 1]) - 65;
      if(mul < 0 || mul > 6)
      {
        cout << "Неверный формат шестнадцатеричного числа";
        return 1;
      }
      number += (10 + mul)*multiplier;
    }
  }

  cout << number << endl;

  return 0;
}
```

Для того чтобы упростить вычисления и проверку, все шестнадцатеричные числа больше 10, такие как A, B, C, D, E, F, приводятся к верхнему регистру при помощи функции `toupper()` из библиотеки `ctype`.

II.1.21. Исключающее ИЛИ

Для реализации исключающего ИЛИ можно использовать формулу, представленную в листинге II.1.40.

Листинг II.1.40. Эмуляция исключающего ИЛИ

```
xor = (x || y) && !(x && y)
```

II.1.22. Возведение числа в степень

Самый простой способ возвести число в степень — это в цикле перемножить его само на себя столько раз, какова степень числа (листинг II.1.41).

Листинг II.1.41. Возведение числа в степень

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

int main()
{
  int number = 0;
  int power  = 0;
  int result = 1;

  cout << "Введите число ";
  cin >> number;
  cout << "Введите степень ";
  cin >> power;

  for(int i = 0; i < power; i++)
  {
    result *= number;
  }

  cout << result << endl;

  return 0;
}
```

Помимо этого, можно также воспользоваться битовыми операторами, которые подробно рассматриваются в *разд. II.1.15*. В этом случае цикл `for` следует заменить циклом `do` из листинга II.1.42. В нем разряды числа `power` постепенно сдвигаются вправо. Если текущий (самый правый разряд) равен 1 (это проверяется путем побитового И (`&`) с 1), то результат умножается на степень, соответствующую этому разряду.

Замечание

Представленное в листинге II.1.42 решение часто называют *быстрым возведением в степень*.

Листинг II.1.42. Альтернативный способ возведения в степень

```
...
do
{
  if(power & 1) result *= number;
  number *= number;
  power = power >> 1;
}
while(power > 0);
...
```

На первый взгляд, цикл в листинге II.1.42 выполняет гораздо больший объем работ, чем цикл в листинге II.1.41: помимо умножения выполняется оператор `if`, осуществляется сдвиг числа вправо. Однако цикл в листинге II.1.42 выполняется быстрее, т. к. для получения результата ему требуется меньшее число итераций. Допустим, необходимо вычислить значение 2^{30}. Просто перемножение числа 2 на себя потребует 30 итераций, в то время как решение со сдвигом числа 30 потребует столько итераций, сколько битов требуется для представления числа 30 (11110), т. е. 5 итераций, что в 6 раз меньше.

Замечание

В промышленном коде лучше воспользоваться функцией `pow()` из библиотеки `math`. Читающий ваш код разработчик не оценит даже самого красивого алгоритма, если для его реализации предусмотрена стандартная библиотечная функция.

II.1.23. Смена знака числа

Для того чтобы изменить знак вводимого числа, достаточно инвертировать бинарное представление числа при помощи побитового отрицания и прибавить единицу (листинг II.1.43).

Листинг II.1.43. Смена знака числа

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

int main()
{
  int number = 0;

  cout << "Введите число ";
  cin >> number;

  number = ~number + 1;

  cout << number << endl;

  return 0;
}
```

II.1.24. Изменение регистра символов строки

ASCII-коды английского алфавита выбраны так, что разница между кодами прописного и строчного символа составляет 32, а оно в свою очередь обозначается бинарным числом 0100000. Проанализировав бинарные коды любой строчной и прописной буквы, можно сделать вывод, что достаточно изменить шестую бинарную позицию на 0 или 1, и это приведет к изменению регистра.

```
A 1000001 (65)
a 1100001 (97)
```

Осуществить такое преобразование можно путем накладывания на бинарное представление символа битовой маски таким образом, чтобы изменению подвергался только нужный бит. Так, если к бинарному представлению символа 'a' прибавить (используя оператор `&`) число 95, получится символ 'A' (ASCII-код 65):

```
1100001 (97)
1011111 (95)
```

В листинге II.1.44 представлена программа, приводящая символы английского алфавита в строке к верхнему регистру.

Листинг II.1.44. Преобразование к верхнему регистру

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

int main()
{
  char input[80];
  char *ch;

  cout << "Введите строку ";
  cin >> input;

  for(ch = input; *ch; ch++) cout << char(*ch & 223);

  return 0;
}
```

Если потребуется наоборот привести каждый символ строки к нижнему регистру, то достаточно сложить (используя оператор |) ASCII-код символа с числом 32 (листинг II.1.45).

Листинг II.1.45. Преобразование к нижнему регистру

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

int main()
{
  char input[80];
  char *ch;

  cout << "Введите строку ";
  gets(input);

  for(ch = input; *ch; ch++) cout << char(*ch | 32);

  return 0;
}
```

II.1.25. Глобальные переменные

Глобальные переменные объявляются вне функции `main()` и вообще вне всяких функций. Они доступны в любой точке программы при условии, что объявлены раньше, чем используются. Это достаточно удобно, когда переменная должна быть видна из нескольких функций. Последнее не всегда возможно, например, в листинге 1.9 переменная `var` объявляется позже, чем используется (в функции `main()`). Ситуация усугубляется тем, что часто требуется доступ к глобальным переменным из нескольких файлов. Повторное объявление глобальных переменных не допустимо — в этом случае будут объявлены две разные переменные, значения которых не будут совпадать. Для того чтобы обойти это, используется ключевое слово `extern`, которое сообщает, что глобальная переменная уже объявлена в другом месте программы. Такие переменные называют *внешними*.

Ключевое слово `extern` имеет одну интересную особенность: если переменная инициализируется, то это становится определением, а не просто объявлением внешней глобальной переменной. Поэтому если в листинге 1.9 повторное определение переменной `var` после функции `main()` приводит к ошибке, следует либо убрать инициализацию переменной при объявлении ее с ключевым словом `extern` (листинг II.1.46), либо убрать определение переменной `var` после функции `main()` (листинг II.1.47).

Листинг II.1.46. Корректное объявление внешней глобальной переменной

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

extern int var;

int main()
{
  cout << var << endl;

  return 0;
}

int var = 10;
```

Листинг II.1.47. Альтернативное объявление внешней глобальной переменной

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

```
extern int var = 10;

int main()
{
  cout << var << endl;

  return 0;
}
```

II.1.26. Статическая глобальная переменная

Глобальная переменная доступна всем функциям программы без исключения, только если в функции не объявляется локальная переменная с тем же именем. Причем область видимости переменной распространяется не только на текущий, но и на другие файлы программы.

Замечание

Для того чтобы глобальная переменная стала доступна в других файлах, ее следует объявить в этих файлах с ключевым словом `extern`.

Объявление глобальной переменной с ключевым словом `static` означает, что глобальная переменная будет доступна только в текущем файле и не будет доступна в других файлах программы. Такое ограничение не является лишним, т. к. позволяет лимитировать область действия глобальной переменной.

Замечание

Обычно рекомендуется избегать использования глобальных переменных, т. к. их состояние сложно контролировать, а для того чтобы отладить программу с их участием, требуется держать в голове все блоки. Программу обычно разбивают на несколько блоков, каждый из которых отлаживают отдельно. Глобальные переменные приводят к тому, что блоки начинают влиять друг на друга и отлаживать их отдельно не получается. Использование статических глобальных переменных уменьшает их возможное разрушительное действие.

II.1.27. Оператор "запятая"

Оператор "запятая" используется в основном в цикле `for`, однако допускается применение данного оператора и вне оператора `for`. Все перечисленные через запятую операторы выполняются, вся конструкция возвращает значение последнего выражения. Поэтому в листинге I.1.11 переменная `var` принимает значение `var1 + 1`, т. е. `var = 11`. Переменные `var1` и `var2` принимают значения 10 и 11 соответственно. Следует помнить, что оператор "равно" имеет больший приоритет, чем оператор "запятая", поэтому в листинге I.1.12 переменная `var` принимает значение 10.

II.1.28. Использование структур и перечислений

Перечисление позволяет объявить последовательность констант, значения которых отличаются друг от друга. Структура `struct` позволяет создать новый тип данных, объединяющий несколько других существующих типов. Один из вариантов объявления структуры `forum_user` представлен в листинге II.1.48.

Листинг II.1.48. Использование перечисления

```
int main()
{
  enum status_enum {administrator, moderator, user};

  struct forum_user
  {
    char name[80];
    status_enum status;
  };
}
```

В листинг II.1.48 объявление перечисления `status_enum` предшествует объявлению структуры `forum_user`. Однако допускается объявление перечисления и непосредственно в структуре (листинг II.1.49).

Листинг II.1.49. Альтернативное объявление структуры `status_enum`

```
int main()
{
  struct forum_user
  {
    char name[80];
    enum status_enum {administrator, moderator, user} status;
  };
}
```

При объявлении перечисления непосредственно в теле структуры могут возникнуть трудности, если понадобится приведение типа к `status_enum`. Однако это требуется редко, т. к. получить строковое представление элементов перечисления можно только при помощи оператора `switch`, как это представлено в листинге II.1.50.

Замечание

Константы в перечислении `forum_user` нумеруются, начиная с 0, однако значения, которые могут принимать константы, допустимо изменять явно при помощи оператора "равно". Например, при объявлении `enum status_enum {administrator = 10, moderator, user}` константы `administrator`, `moderator`, `user` получат значения 10, 11 и 12, вместо 0, 1 и 2, когда корректировка значений не производится.

Листинг II.1.50. Заполнение структуры `forum_user` пользователем

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

int main()
{
  int number = 0;

  struct forum_user
  {
    char name[80];
    enum status_enum {administrator, moderator, user} status;
  } man;

  cout << "Введите имя ";
  cin >> man.name;
  cout << "Введите статус (0 - administrator, 1 - moderator, 2 - user) ";
  cin >> number;

  man.status = number;

  cout << "name  = " << man.name   << endl;

  switch(man.status)
  {
    case 0:
      cout << "status = administrator" << endl;
      break;
    case 1:
      cout << "status = moderator" << endl;
      break;
    case 2:
      cout << "status = user" << endl;
```

```
      break;
  }

  return 0;
}
```

В листинге II.1.50 объявляется структура `man`. Помимо объявления структуры непосредственно в объявлении допускается создание переменных типа `forum_user` позже при помощи объявления `forum_user man`. При обращении к элементам структуры после имени структуры `man` помещаются точка и имя поля, например, `man.name`.

II.1.29. Объединение и битовые поля

Битовое поле — это специальный вид поля в структуре, длиной 1 бит. Конечно, на самом деле память выделяется объемами минимум 8 битов (1 байт), и структура из 3 или 5 битов будет занимать 1 байт. Один байт (8 битов) может хранить 8 булевых состояний, т. е. 8 флагов, которые могут принимать либо значение истина (1), либо ложь (0).

Замечание

Объединения и структуры обычно имеют название. Однако допускается использование их без названия. В этом случае такие объединения и структуры называют *анонимными*.

В листинг II.1.51 используется анонимное объединение битового поля `byte` с переменной `var` типа `char`. Объединение означает, что переменная `var` и битовое поле `bit` занимают один и тот же объем памяти. Присвоив переменной `var` число 113, можно получить его бинарное представление. Для этого каждое битовое поле проверяется при помощи оператора `if`: если оно содержит 1 — выводим 1, если 0, то выводится 0.

Листинг II.1.51. Использование объединений и битовых полей

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

int main()
{
  // Битовое поле из 8 байтов
  struct byte {
    unsigned b1 : 1;
    unsigned b2 : 1;
```

```
    unsigned b3 : 1;
    unsigned b4 : 1;
    unsigned b5 : 1;
    unsigned b6 : 1;
    unsigned b7 : 1;
    unsigned b8 : 1;
  };
  // Объединение
  union {
    char var;
    struct byte bit;
  };
  var = 113;

  if(bit.b8) cout << "1";
  else cout << "0";

  if(bit.b7) cout << "1";
  else cout << "0";

  if(bit.b6) cout << "1";
  else cout << "0";

  if(bit.b5) cout << "1";
  else cout << "0";

  if(bit.b4) cout << "1";
  else cout << "0";

  if(bit.b3) cout << "1";
  else cout << "0";

  if(bit.b2) cout << "1";
  else cout << "0";

  if(bit.b1) cout << "1";
  else cout << "0";
}
```

Анонимное объединение не имеет имени и совмещает память независимых переменных. Последнее означает, что если в состав анонимного объединения входят переменные `var` и `bit`, то к ним можно обращаться как к обычным переменным так, как если бы они были объявлены без использования объеди-

нения. Именованное объявление позволяет создать новый тип, и к переменным, входящим в их состав, нельзя обращаться как к обычным переменным программ — допускается обращение только через имя экземпляра объединения (листинг II.1.52).

Замечание

Именованное объединение используется, когда необходима передача переменной данного типа внутрь функции, анонимное объединение создается, когда требуется совместить память двух или более переменных, а сам экземпляр объединения не требуется передавать в качестве параметра функции.

Листинг II.1.52. Использование именованного объединения

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

int main()
{
  // Битовое поле из 8 байтов
  struct byte {
    unsigned b1 : 1;
    unsigned b2 : 1;
    unsigned b3 : 1;
    unsigned b4 : 1;
    unsigned b5 : 1;
    unsigned b6 : 1;
    unsigned b7 : 1;
    unsigned b8 : 1;
  };
  // Объединение
  union char_union {
    char var;
    struct byte bit;
  } chr;
  chr.var = 113;

  if(chr.bit.b8) cout << "1";
  else cout << "0";

  if(chr.bit.b7) cout << "1";
  else cout << "0";

  if(chr.bit.b6) cout << "1";
  else cout << "0";
```

```
  if(chr.bit.b5) cout << "1";
  else cout << "0";

  if(chr.bit.b4) cout << "1";
  else cout << "0";

  if(chr.bit.b3) cout << "1";
  else cout << "0";

  if(chr.bit.b2) cout << "1";
  else cout << "0";

  if(chr.bit.b1) cout << "1";
  else cout << "0";
}
```

II.1.30. Преобразование арабского числа в римское

Преобразование арабского числа в римское имеет некоторые особенности, главная из которых заключается в том, что для представления римского числа используются цифры 1 и 5, причем для каждой десятичной позиции это обозначение свое. Это позволяет обойтись без цифры 0. В табл. II.1.5 представлено соответствие арабских чисел и римских.

Таблица II.1.5. *Соответствие арабских и римских чисел*

Арабское число	Римское число	Арабское число	Римское число	Арабское число	Римское число
1	I	10	X	100	C
2	II	20	XX	200	CC
3	III	30	XXX	300	CCC
4	IV	40	XL	400	CD
5	V	50	L	500	D
6	VI	60	LX	600	DC
7	VII	70	LXX	700	DCC
8	VIII	80	LXXX	800	DCCC
9	X	90	XC	900	M

Самым разумным является последовательный перебор позиций десятичного числа и обработка остатка от деления на 10 в конструкции `switch`. В конст-

рукции будет 9 позиций, позиция для числа 0 не нужна, т. к. в римских числах ее нет, и она никак не обозначается. Мы будем заполнять строку с римским числом `roman` справа налево. Для отслеживания текущей позиции будет использоваться переменная `current_index`. Так как на каждой десятичной позиции цифры 1 и 5 обозначаются своей собственной буквой, разумно хранить эти обозначения в специальных массивах `one[]` и `five[]`. В листинге II.1.53 представлено возможное решение задачи.

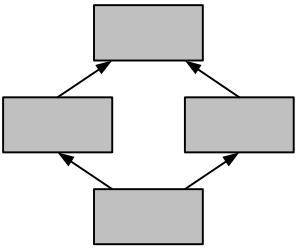
Листинг II.1.53. Преобразование арабского числа в римское

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

int main()
{
  int const COUNT = 11;

  // Арабское число
  int number = 0;
  // Строка для римского числа
  char roman[COUNT];
  // Текущая позиция в строке
  int current_index = COUNT - 1;

  // Заполняем строку roman пробелами
  for(int i = 0; i < COUNT; i++) roman[i] = ' ';
  // Завершаем строку roman нулевым символом
  roman[current_index--] = '\0';

  cout << "Введите арабское число (1 - 2000)";
  cin >> number;
  if(number > 2000 || number < 1)
  {
    cout << "Число должно принимать значение от 1 до 2000\n";
    return 1;
  }

  // Определяем обозначение 1, 10, 100 и 1000
  char one[4] = {'I','X','C','M'};
  // Определяем обозначение 5, 50 и 500
  char five[3] = {'V','L','D'};
  // Устанавливаем индекс для массивов one[] и five[]
  int index = 0;
```

```
do
{
  switch(number % 10)
  {
    case 1:
      roman[current_index--] = one[index];
      break;
    case 2:
      roman[current_index--] = one[index];
      roman[current_index--] = one[index];
      break;
    case 3:
      roman[current_index--] = one[index];
      roman[current_index--] = one[index];
      roman[current_index--] = one[index];
      break;
    case 4:
      roman[current_index--] = five[index];
      roman[current_index--] = one[index];
      break;
    case 5:
      roman[current_index--] = five[index];
      break;
    case 6:
      roman[current_index--] = one[index];
      roman[current_index--] = five[index];
      break;
    case 7:
      roman[current_index--] = one[index];
      roman[current_index--] = one[index];
      roman[current_index--] = five[index];
      break;
    case 8:
      roman[current_index--] = one[index];
      roman[current_index--] = one[index];
      roman[current_index--] = one[index];
      roman[current_index--] = five[index];
      break;
    case 9:
      roman[current_index--] = one[index + 1]; // Вместо I -> X
      roman[current_index--] = one[index];
      break;
  }
```

```
    number = number / 10;
    index++;
  }
  while(number);
  // Выводим результат
  cout << roman << "\n";

  return 0;
}
```

ГЛАВА II.2



Указатели, ссылки, массивы, строки

II.2.1. Укоротить строку

Для того чтобы укоротить строку, необходимо поставить символ `\0` — признак конца строки в шестой позиции, вместо запятой, разделяющей слова "Hello" и "world!" (листинг II.2.1).

Листинг II.2.1. Превращение "Hello, world!" в "Hello"

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

int main()
{
  char str[] = "Hello, world!";
  str[6] = '\0';
  cout << str << endl;

  return 0;
}
```

Дело в том, что в C и C++ конец строки обозначается нулевым символом `\0`. То есть когда мы объявляем строку при помощи `char str[]`, инициируя массив `str[]` строкой "Hello, world!", массив `str[]` содержит 14 элементов (13 символов строки плюс завершающий символ `\0`). Поэтому стоит переместить символ `\0` из последней позиции в шестую, строка `str` будет содержать лишь слово "Hello". Все, что останется после 6-го символа, восприниматься не будет (рис. II.2.1).

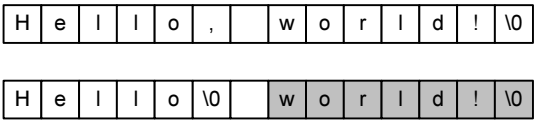


Рис. II.2.1. Структура строк "Hello, world!" и "Hello"

Важно отметить, что все, что находилось в строке "Hello, world!" после шестого символа, остается незатронутым, и эту память можно использовать в своих интересах, т. к. под массив `str[]` в программе выделяется 14 символов.

Увеличивать строку описанным выше способом нельзя, т. к. произойдет выход за границу массива: сразу за строкой "Hello, world!" может находиться все, что угодно — следующая переменная, служебные данные и т. п. Однако компилятор C++ позволит выполнить эту операцию, так программа, представленная в листинге II.2.2, в которой к строке `str` прибавляются дополнительные 6 символов справа. Программа даже работает, но эффект в разных системах будет разный. Нельзя даже гарантировать, что в одной и той же системе программа будет вести себя одинаково, т. к. происходит перезапись данных за границами массива. Например, у авторов эта программа вызвала бесконечный цикл, выводящий строку "Hello, world!", однако такие эффекты — редкость, чаще программа просто аварийно завершается.

Замечание

Тот факт, что программы на C/C++ не отслеживают выход за границы массива, часто используется злоумышленниками для взлома программного обеспечения. Если программа принимает строку данных от пользователя и хранит ее в короткой строке — специально подготовленная строка может переписать участки программы за пределами массива символов, при этом оставив программу работоспособной, но изменив ее логику. Такую атаку называют *атакой переполнения буфера*, а строки-программы, вызывающие изменение логики, — *эксплоитами*.

Листинг II.2.2. Ошибочное увеличение размера строки

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

int main()
{
  char str[] = "Hello, world!";
  str[14] = ' ';
  str[20] = '\0';
  cout << str << endl;

  return 0;
}
```

Таким образом, если необходимо использовать больший объем памяти — его нужно специально выделять, либо объявляя массив или строку достаточной длины, либо выделяя память динамически.

II.2.2. Объявление строки

Переменные, объявленные в листинге 2.2, содержат (или указывают) одно и то же значение "Hello". Однако природа переменных различна. Переменные `str1`, `str2`, `str3` и `str4` объявляются как указатели на строку "Hello" и являются полными аналогами:

```
char *str1   = "Hello";
char str2[]  = "Hello";
char str3[6] = "Hello";
char *str4;
str4 = "Hello";
```

Это означает, что можно поэлементно обращаться к строке `str1`, т. е. выражения `str[1]` и `str1[6]` вполне допустимы, а обращение к переменным `str2`, `str3` и `str4` без указания квадратных скобок вернет адрес переменной.

Замечание

Массивы введены для удобства записи, на самом деле любой массив представляет собой не что иное, как указатель.

Указатели `str1` и `str2` отличаются способом инициализации. В случае переменной `str1` происходит инициализация, в случае переменной `str4` — присвоение. Это не имеет значение для базовых типов, однако в случае пользовательских объектов разница будет более существенна — для инициализации потребуется реализовать копирующий конструктор, а для присвоения перегрузить оператор =.

Указатели `str5` и `str4` являются синонимами переменных `str4` и `str1` соответственно.

```
char *str5;
str5 = str4;
char *str6;
str6 = str1;
```

Это означает, что если содержимое переменных `str5` и `str6` поменяется, это автоматически приведет к изменению содержимого переменных `str4` и `str1`, т. к. эти пары имеют одинаковые адреса (листинг II.2.3).

Листинг II.2.3. Связанные друг с другом указатели

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

int main()
{
  char *str1   = "Hello";

  char *str6;
  str6 = str1;

  str1[0] = 'B';
  str1[1] = 'y';
  str1[2] = 'e';
  str1[3] = '\0';

  cout << str6 << endl; // "Bye"

  return 0;
}
```

Как видно из листинга II.2.3, если значение переменной `str1` меняется, то меняется и значение переменной `str6`, которая указывает на нее.

II.2.3. Размер строки

Задача сводится к тому, чтобы прочитать из потока ввода строку, которую введет пользователь, и подсчитать число символов, которые в нее входят. Для чтения строки удобно использовать функцию `gets()` из библиотеки `<stdio>`. Функция имеет следующий синтаксис:

```
char *gets(char *str)
```

В качестве аргумента функция принимает указатель на строку, куда будет помещен результат ввода пользователя, этот же указатель функция и возвращает. Все символы, которые пользователь введет до нажатия клавиши <Enter>, будут помещены в строку *str* и завершены нулевым символом `\0`.

Замечание

Применить стандартную библиотеку ввода/вывода для чтения переменной нельзя, т. к. объект `cin` прочитает строку только до первого пробельного символа и проигнорирует все остальное. То есть вместо строки "Hello, world!" в `input` будет помещена строка "Hello,".

С обобщением всего вышесказанного решение задачи определения размера строки может выглядеть так, как это представлено в листинге II.2.4.

Листинг II.2.4. Определение размера строки

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

int main()
{
  char input[80];
  int number = 0;

  cout << "Введите строку ";
  gets(input);

  while(input[number++]);

  cout << "Строка содержит " << number << " символов " << endl;

  return 0;
}
```

Для подсчета числа символов используется переменная `number`, которая увеличивает свое значение на единицу, тем самым пробегая от первого символа строки до последнего. Так как все символы строки, кроме признака конца строки `\0`, имеют ASCII-код, отличный от 0, работа цикла завершается только в том случае, если найден конец строки.

Программа в листинге II.2.4 потенциально опасна, т. к. пользователь может ввести строку, содержащую более 80 символов. Функция `gets()` не контролирует размер переменной `input` и размер вводимой строки: все, что вводит пользователь, будет записано по адресу переменной `input`. Это приведет к переполнению буфера и перезатиранию соседних с переменной `input` участков памяти. Для того чтобы предотвратить эту потенциальную опасность, вместо функции `gets()` применяется функция `fgets()`, которая предназначена для работы с файлами, но может быть использована для чтения с консоли, если вместо открытого файлового потока указать стандартный поток ввода `stdin`.

Замечание

В начале выполнения любой программы автоматически открываются три стандартных потока: `stdin` (стандартный поток ввода), `stdout` (стандартный поток

вывода) и stderr (стандартный поток ошибок). По умолчанию потоки связаны с консолью, но могут быть перенаправлены при помощи средств операционной системы и связаны, например, с файлом или вводом или выводом другой программы.

Функция fgets() из библиотеки <cstdio> читает в строку заданное количество символов из потока и имеет следующий синтаксис:

```
char *fgets(char *str, int number, FILE *stream)
```

Функция fgets() читает из потока *stream* только *number* — 1 символов и записывает их в строку *str*. Это позволяет избежать атаки переполнения буфера. Решение задачи, представленное в листинге II.2.4, можно переписать следующим образом (листинг II.2.5).

Листинг II.2.5. Безопасное решение

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

int main()
{
  char input[80];
  int number = 0;

  cout << "Введите строку " << endl;
  fgets(input, 80, stdin);

  while(input[number++]);

  cout << "Строка содержит " << number << " символов " << endl;

  return 0;
}
```

Замечание

В промышленном коде для определения размера строки используют стандартную функцию strlen(). Любой программист, читающий или сопровождающий код, без труда узнает ее и вспомнит ее предназначение. Самописный код потребует разбора в течение нескольких минут.

II.2.4. Количество элементов массива

Для того чтобы определить количество элементов массива, необходимо выяснить его размер при помощи конструкции sizeof(), а также размер отдель-

ного его элемента. Частное от этих величин будет равно количеству элементов массива (листинг II.2.6).

Листинг II.2.6. Определение количества элементов массива

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

int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  cout << sizeof(arr)/sizeof(arr[0]) << endl;

  return 0;
}
```

II.2.5. Увеличение размера строки

Так как размер строки, которую требуется создать, заранее не известен, необходимо прибегнуть к динамическому выделению памяти при помощи оператора `new` для указателя `result`, который будет играть роль новой расширенной строки. Решение с использованием этого оператора представлено в листинге II.2.7. Число, введенное пользователем, помещается в переменную `number`.

Замечание

Природа массивов такова, что элементы массива должны следовать один за другим, массив не может разрываться и состоять из нескольких частей. Поэтому когда требуется увеличить размер строки, приходится объявлять новую строку большего размера.

Далее необходимо заполнить новую переменную нулями в цикле `for`. Цикл `while` для этого не подходит, т. к. только что выделенная область памяти содержит мусор и может иметь нулевые символы, что приведет к тому, что строка заполнится нулями не до конца. Вторым циклом в начало строки `result` записывается содержимое строки `binary`.

Листинг II.2.7. Увеличение размера строки

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

int main()
```

```
{
  char *binary = "1111";
  char *result;
  int number = 0;

  cout << "Введите число ";
  cin >> number;

  if(number > 4 && number <= 81)
  {
    // Выделяем память под строку result
    result = new char[number];
    // Завершаем строку
    result[number - 1] = '\0';
    // Заполняем новую строку нулями
    for(int i = 0; i < number - 1; i++) result[i] = '0';
    // Копируем содержимое строки binary в начало строки result
    for(int i = 0; i < strlen(binary); i++) result[i] = binary[i];
    // Выводим строку result
    cout << result << endl;
    // Освобождаем память
    delete [] result;
  }

  return 0;
}
```

После того как работа с динамической переменной завершена, зарезервированную при помощи оператора `new` память следует вернуть системе. Если память выделялась под указатель, то используется оператор `delete`, если под массив — `delete []`.

II.2.6. Чередование символов строки и пробелов

Для того чтобы для введенной пользователем строки создать новую строку, в которой символы разделены пробелами, придется прибегнуть, как и в предыдущем примере, к динамическому выделению памяти. Это вызвано тем, что размер строки заранее не известен. После ввода пользователем строки `input` ее размер определяется при помощи стандартной функции `strlen()`. Размер результирующей строки `result` определяется как удвоенный размер строки `input` за вычетом единицы (число пробелов на единицу меньше, чем число символов). Именно этот размер передается оператору `new`, который вы-

деляет память под строку. Далее в цикле `for`, который пробегает по всем элементам строки `result`, четные символы заполняются символами строки `input`, а не четные — пробелами (листинг II.2.8).

Листинг II.2.8. Чередование строки пробелами

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

int main()
{
  char input[80];
  char *result;

  cout << "Введите строку ";
  fgets(input, 80, stdin);

  // Выделяем память под строку result
  result = new char[strlen(input)*2 - 1];
  // Завершаем строку
  result[strlen(input)*2 - 2] = '\0';

  for(int i = 0; i < strlen(input)*2 - 2; i++)
  {
    // Если строка четная - копируем символ
    // из строки input, иначе вставляем пробел
    if(i % 2) result[i] = ' ';
    else result[i] = input[i / 2];
  }

  // Выводим строку result
  cout << result << endl;
  // Освобождаем память
  delete [] result;

  return 0;
}
```

II.2.7. Сравнение строк

В листинге II.2.9 представлена программа, производящая сравнение строк, введенных пользователем. Первая строка помещается в переменную `in_fst`, а вторая в `in_snd`. Перед тем как сравнивать строки, необходимо определить их

длину, так чтобы производить сравнение по короткой строке. Длина наиболее короткой строки помещается в переменную `number`, после чего в цикле от 0 до `number` производится сравнение символов обеих строк. Если символы равны, текущая итерация прерывается при помощи оператора `continue`, и происходит переход к следующей итерации. Если встречается пара символов строк `in_fst` и `in_snd`, которые не равны между собой, то производится их сравнение. Если символ строки `in_fst` больше символа строки `in_snd`, то первая строка признается большей и в результирующую переменную `result` помещается 1, в противном случае больше вторая строка и в результирующую переменную помещается значение –1. Если строки `in_fst` и `in_snd` равны, то переменная `result` остается равной нулю.

Листинг II.2.9. Сравнение строк

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

int main()
{
  char in_fst[80];
  char in_snd[80];
  int number = 0;

  cout << "Введите первую строку ";
  fgets(in_fst, 80, stdin);
  cout << "Введите вторую строку ";
  fgets(in_snd, 80, stdin);

  // Определяем размер самой короткой строки
  int fst = strlen(in_fst);
  int snd = strlen(in_snd);
  if(fst <= snd) number = fst;
  else number = snd;

  int result = 0;
  for(int i = 0; i < number; i++)
  {
    if(in_fst[i] == in_snd[i]) continue;
    if(in_fst[i] > in_snd[i]) result = 1;
    else result = -1;
    break;
  }
```

```
  cout << result << endl;

  return 0;
}
```

II.2.8. Упаковка IP-адреса

Прежде всего, следует принять значения параметров. Для этого следует воспользоваться функцией `main()` с двумя параметрами так, как мы уже делали в решении задачи из *разд. I.1.8*.

```
int main(int argc, char *argv[])
{
  // ...
}
```

Если количество переданных функции аргументов *argc* меньше трех (в качестве первого аргумента выступает имя программы), следует прекратить работу, как впрочем, если второй элемент массива параметров *argv*`[]` не равен строке `-p` или `-u`. Сравнение удобно проводить при помощи функции `strncmp()` из библиотеки `<cstring>`, которая имеет следующий синтаксис:

```
int strncmp(const char *str1, const char str2, size_t count)
```

Функция производит лексикографическое сравнение первых *count* символов строк *str1* и *str2* и возвращает 0, если строки равны, значение меньше нуля, если *str1* меньше *str2*, и значение больше нуля — в противном случае.

IP-адрес *XXX.YYY.ZZZ.WWW* можно преобразовать в целое число по формуле:

$$XXX \cdot 256^3 + YYY \cdot 256^2 + ZZZ \cdot 256^1 + WWW \cdot 256^0.$$

Следует учитывать тот факт, что в обычную целую переменную типа `int` данное число не убирается, для хранения упакованного IP-адреса следует использовать 8-байтное число `long int`.

Обратное преобразование упакованного IP-адреса в строку вида *XXX.YYY.ZZZ.WWW* производится путем деления числа на 256. В листинге II.2.10 представлено одно из возможных решений задачи преобразования IP-адреса.

Листинг II.2.10. Упаковка и распаковка IP-адреса

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

int main(int argc, char *argv[])
```

```
{
  unsigned long int ipaddress = 0;

  // Проверка параметров
  if(argc < 3)
  {
    cout << "Too few parameters\n";
    return 1;
  }
  else if(strncmp(argv[1],"-p",2) && strncmp(argv[1],"-u",2))
  {
    cout << "Неверный формат параметров\n";
    return 1;
  }

  // Если передан IP-адрес — вычисляем число
  if(!strncmp(argv[1],"-p",2))
  {
    char ip[4][4];
    int i, j, count;
    // Помещаем отдельные фрагменты IP-адреса в массив ip
    // count — номер фрагмента IP-адреса
    // i — номер символа в параметре
    // j — номер символа во фрагменте
    for(i = 0, j = 0, count = 0; i < strlen(argv[2]); i++)
    {
      if(isdigit(argv[2][i]))
      {
        ip[count][j++] = argv[2][i];
      }
      else if(argv[2][i] == '.')
      {
        ip[count++][j++] = '\0';
        j = 0;
      }
    }
    if(count < 3)
    {
      cout << "Неверный формат IP-адреса\n";
      return 1;
    }
    ip[count++][j++] = '\0';

    // Переводим IP-адрес в бинарную форму
    for(int i = 3, mul = 1; i >= 0 ; i--, mul *= 256)
```

```
    {
      int fragment = 0;
      for(int k = strlen(ip[i]), mult = 1; k; k--, mult *= 10)
      {
        fragment += (ip[i][k — 1] — 48) * mult;
      }
      ipaddress += fragment * mul;
    }
    cout << ipaddress << "\n";
  }

  int ip_fragment[] = {0,0,0,0};
  // Если передано число — восстанавливаем IP-адрес
  // в формате XXX.YYY.ZZZ.WWW
  if(!strncmp(argv[1],"-u",2))
  {
    int number = 0;
    for(int k = strlen(argv[2]), mult = 1; k; k--, mult *= 10)
    {
      number = argv[2][k — 1] — 48;
      if(number < 0 || number > 9)
      {
        cout << "Неверный формат переменной\n";
        return 1;
      }
      ipaddress += number * mult;
    }
    number = 0;
    int j = 0;
    do
    {
      if(ipaddress % 256) ip_fragment[j++] = (ipaddress % 256);
      else ip_fragment[j++] = 0;
      ipaddress = ipaddress / 256;
    } while(ipaddress);
    // Выводим IP-адрес
    cout << ip_fragment[3] << '.'
         << ip_fragment[2] << '.'
         << ip_fragment[1] << '.'
         << ip_fragment[0] << endl;
  }

  return 0;
}
```

II.2.9. Адрес переменной

Для получения адреса переменной используется унарный оператор `&`. В листинге II.2.11 представлено возможное решение задачи.

Листинг II.2.11. Определение адреса переменной

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

int main()
{
  int var = 0;
  double arr[] = {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};

  cout << "var    = " << &var << endl;
  cout << "arr[0] = " << &arr << endl;
  cout << "arr[9] = " << &arr[9] << endl;
}
```

Результат работы программы может выглядеть следующим образом:

```
var    = 1245064
arr[0] = 1244984
arr[9] = 1245056
```

Следует отметить, что разница между адресами `arr[0]` и `arr[9]` составляет 72 байта, именно столько занимают 9 элементов массива по 8 байтов каждый. Интересно отметить, что переменная `var` оказалась сразу за последним элементом массива `arr[]`, хотя объявлена была первой. Однако строить на этом логику программы ни в коем случае нельзя, другой компилятор может разместить переменные в ином порядке или поместить между ними служебную информацию. Единственное, в чем можно быть твердо уверенным, — это то, что элементы массива `arr[]` будут располагаться друг за другом, причем в порядке увеличения индекса массива.

II.2.10. Обход массива при помощи указателей

Листинг 2.6 можно переписать без использования оператора `[]` так, как это представлено в листинге II.2.12.

Замечание

Для того чтобы получить значение, на которое указывает указатель, следует использовать оператор разыменования `*`, причем при использовании адресной

арифметики (сложение, вычитание указателей) выражение нужно заключать в круглые скобки, т. к. унарный оператор * имеет самый высокий приоритет.

Листинг II.2.12. Использование адресной арифметики

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

int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  for(int i = 0; i < 10; i++)
  {
    cout << *(arr + i) << endl;
  }

  return 0;
}
```

Решение основано на том факте, что имя массива является, по сути, указателем на начало массива, а сами элементы массива располагаются в памяти по порядку. Прибавление к указателю целого числа приводит к смещению адреса, на который указатель указывает, на число позиций слагаемого (рис. II.2.2).

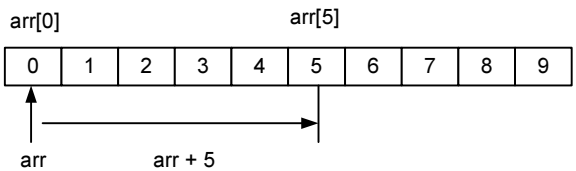


Рис. II.2.2. Целочисленные операции с указателями

Использование индексов массива зачастую более удобно и наглядно, однако применение указателей зачастую более эффективно, т. к. получаемый в результате бинарный код более компактен.

II.2.11. Получение старшего и младшего разрядов

Одно из решений может быть основано на применении указателей. Первые два байта числа типа `int` получаются путем присвоения его адреса указателю меньшей разрядности `short int`. Вторые два байта можно получить, присвоив

единицу первому указателю: единица в данном случае означает не число байтов, а количество переменных `short int`, т. е. смещение производится ровно на 2 байта. В листинге II.2.13 представлено возможное решение задачи.

Листинг II.2.13. Получение старшего и младшего разрядов

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

int main()
{
  int var = 2000000;
  short int *first = (short int *)&var;
  short int *second = first + 1;
  cout << "var    = " << var  << endl;
  cout << "first  = " << *first << endl;
  cout << "second = " << *second << endl;
}
```

Результат работы программы может выглядеть следующим образом:

```
var    = 2000000
first  = -31616
second = 30
```

Объяснить результат работы программы проще, если представить переменную `var` в двоичном исчислении.

```
00000000 00011110 10000100 10000000 (2000000)
00000000 00011110 (30)
10000100 10000000 (-31616)
```

Так как последовательность 1000010010000000 эквивалентна десятичному числу 33920, происходит выход за границу типа `short int`, который может принимать значения от –32 768 до 32 767, в результате выводится отрицательное значение –31 616. Однако можно добиться положительного числа. Для этого достаточно заменить тип `short int *` в листинге II.2.13 на беззнаковый тип `short unsigned int *`.

Замечание

Следует обратить внимание, что младший разряд `first` оказался справа, а старший `second` — слева. Это особенность 32-битной архитектуры Intel-компьютеров — указатель указывает на младший разряд.

Второе решение основано на применении *объединений* — структур, состоящих из нескольких переменных, которые разделяют одну и ту же область па-

мяти. Все переменные имеют один и тот же адрес, но различную длину. Поэтому можно поместить в объединение переменную типа `int` и массив из двух элементов `short int`, после этого можно будет записать в переменную типа `int` число, а старший и младший разряды прочитать из первого и второго элемента массива типа `short int` (рис. II.2.3).

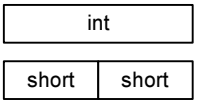


Рис. II.2.3. Отношение переменной `int` и массива из двух элементов `shor int` в объединении

В листинге II.2.14 представлено решение задачи с использованием объединения.

Листинг II.2.14. Использование анонимного объединения

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

int main()
{
  union {
    int var;
    unsigned short int arr[2];
  };
  var = 2000000;
  cout << "var    = " << var  << endl;
  cout << "arr[0] = " << arr[0] << endl;
  cout << "arr[1] = " << arr[1] << endl;
}
```

В листинге II.2.14 используется анонимное объединение, т. е. объединение не образующее новый тип данных, а помещающее независимые переменные в одну и ту же область памяти. Если применяется именованное объединение, то доступ к полям осуществляется через точку (листинг II.2.15).

Листинг II.2.15. Использование именованного объединения

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

int main()
```

```
{
  union {
    int var;
    unsigned short int arr[2];
  } addresses;
  addresses.var = 2000000;
  cout << "var    = " << addresses.var << endl;
  cout << "arr[0] = " << addresses.arr[0] << endl;
  cout << "arr[1] = " << addresses.arr[1] << endl;
}
```

II.2.12. Новый тип

Формат создания указателя на массив, которым является тип `char[6]`, отличается от формата создания указателя на обычный базовый тип. Поэтому создать программу по аналогии с листингом I.2.7 не получится. Решение может выглядеть так, как это представлено в листинге II.2.16.

Листинг II.2.16. Создание нового типа при помощи `typedef`

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

int main()
{
  typedef char (* char6)[6];

  char6 sixth;
  sixth = (char6)"Hello";
  cout << *sixth << endl;
}
```

II.2.13. Блочный вывод строки

Для решения задачи можно воспользоваться указателем типа `char *`, который будет пробегать значения от начала строки до последнего нулевого символа (листинг II.2.17). Это позволит избавиться от вызова библиотечной функции `strlen()`, определяющей длину строки.

Листинг II.2.17. Блочный вывод строки

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

```
int main()
{
  char input[80];
  char *current;
  int i = 1;

  cout << "Введите строку ";
  fgets(input, 80, stdin);

  for(current = input; *current; current++, i++)
  {
    cout << *current;
    if(!(i % 4)) cout << endl;
  }
}
```

Следует помнить, что получить значение, на которое ссылается указатель, можно только применив к нему оператор разыменования `*`. Если его не использовать, будет возвращаться адрес переменной.

II.2.14. Разбивка строки по пробелу

Для того чтобы разбить строку по пробельному символу, удобно воспользоваться вложенным циклом. Причем счетчик внешнего цикла изменять также во внутреннем цикле — это позволяет максимально быстро проверить строку на наличие пробелов (листинг II.2.18).

Листинг II.2.18. Разбивка строки по пробелу

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

int main()
{
  char input[80];
  char current[80];
  int i, j;

  cout << "Введите строку ";
  fgets(input, 80, stdin);

  for(i = 0; i < strlen(input); i++)
  {
```

```
    for(j = 0; input[i] != ' ' && input[i]; j++, i++)
    {
      current[j] = input[i];
    }
    current[j] = '\0';
    cout << current << endl;
  }
}
```

В качестве альтернативы решения, представленного в листинге II.2.18, можно использовать решение с применением указателей (листинг II.2.19).

Листинг II.2.19. Решение с применением указателей

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

int main()
{
  char input[80];
  char current[80];
  char *prt, *prt_cur;

  cout << "Введите строку ";
  fgets(input, 80, stdin);

  prt = input;

  while(*prt)
  {
    prt_cur = current;
    while(*prt != ' ' && *prt)
    {
      *prt_cur = *prt;
      prt_cur++;
      prt++;
    }
    // Пропускаем пробел
    if(*prt) prt++;
    *prt_cur = '\0';
    cout << current << endl;
  }
}
```

Решение в листинге II.2.19 использует вместо индексов цикла `for` непосредственное перемещение по массиву символов, образующих строку, через адресную арифметику указателей. Цикл `while` можно сократить до одной строки:

```
while(*prt != ' ' && *prt) *(prt_cur++) = *(prt++);
```

Однако таких сложных построений лучше избегать, т. к. их анализ требует значительного усилия.

II.2.15. Найдите ошибку

Ошибка заключается в том, что указатель не является инициализированным, он не указывает ни на одну переменную, под которую выделена статическая или динамическая память. Программа способна даже работать, но такого рода ошибки могут проявляться (в большом проекте это рано или поздно происходит) совершенно непредсказуемыми и необъяснимыми сбоями, которые могут обнаруживаться через некоторое время работы программы.

Для того чтобы программа работала корректно, необходимо выделить память под указатель, например, при помощи оператора `new` (листинг II.2.20).

Листинг II.2.20. Выделение динамической памяти

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

int main()
{
  int var, *prt;

  prt = new int;

  var = 10;
  *prt = var;
  cout << * prt << endl;

  delete prt;

  return 0;
}
```

II.2.16. Допустимо ли выражение *****k=56*?

Указатель на указатель является вполне допустимой конструкцией. В этом случае указатель содержит адрес не обычной переменной, а адрес указателя, который может указывать на следующий указатель (рис. II.2.4).

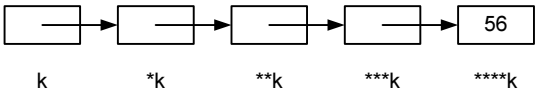


Рис. II.2.4. Цепочка указателей `****k`

В результате можно выстраивать длинные цепочки указателей. В примере `****k=56` цепочка состоит из четырех указателей. Последний из них указывает на ячейку памяти, которой присваивается значение 56 (листинг II.2.21).

Замечание

Таких длинных цепочек указателей следует избегать. Указатели являются чрезвычайно гибкими конструкциями, но они легко позволяют совершить ошибку, особенно при работе с динамической памятью. Если указатель ссылается на произвольную область памяти или при помощи него производится попытка перезаписать служебную информацию, программист может узнать об ошибке только по побочным проявлениям. Более того, он может вообще не столкнуться с деструктивными проявлениями, и они сработают спустя длительное время, после того, как программа будет сдана в эксплуатацию.

Листинг II.2.21. Создание цепочки указателей

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

int main()
{
  int * i;
  i = new int(10);
  int **j = &i;
  int ***n = &j;
  int ****k = &n;

  ****k = 56;

  cout << *i << endl;

  delete i;

  return 0;
}
```

Конструкция `****k=56` изменяет значение, выделенное оператором `new` и на которое в конечном итоге ссылаются все четыре указателя `i`, `j`, `n` и `k` из листинга II.2.21.

II.2.17. Массив строк

Так как максимальное число строк не может превышать 10, разумно объявить двумерный массив `char` объемом 10 на 80 символов. В листинге II.2.22 приводится возможное решение задачи.

Листинг II.2.22. Помещение введенного текста в массив строк

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

int main()
{
  char name[10][80];

  // Читаем строки
  for(int i = 0; i < 10; i++)
  {
    cout << "Введите строку ";
    fgets(name[i], 80, stdin);
    if(name[i][0] == '\0') break;
  }

  // Выводим строки
  for(int i = 0; i < 10; i++)
  {
    if(name[i][0] == '\0') break;
    cout << name[i] << "\n";
  }

  return 0;
}
```

II.2.18. Динамический массив

Для решения задачи потребуется функция `rand()` из библиотеки `<cstdlib>`. *(см. разд. II.1.10)*. Функция возвращает случайное число от 0 до `RAND_MAX` (как правило, 32 767).

Замечание

Для того чтобы функция `rand()` каждый раз генерировала действительно случайную последовательность, а не начинала генерацию случайных чисел в одном и том же порядке, для инициализации генератора случайных чисел используют дополнительную функцию `srand()`.

В листинге II.2.23 представлена программа, которая вычисляет случайное число `count`, принимающее случайное значение от 0 до 100, и заполняет его случайными дробными значениями от 0 до 1.

Замечание

Максимальное количество элементов в динамическом массиве определяет константа `NUMBER_RAND_MAX`, которая объявляется в начале функции `main()`.

Листинг II.2.23. Формирование массива со случайным количеством элементов

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

int main()
{
  // Максимальное количество случайных элементов массива
  const int NUMBER_RAND_MAX = 100;
  // Инициируем генератор случайных чисел
  srand(time(NULL));
  // Случайное количество элементов массива
  int count = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  // Выделяем память под динамический массив
  double *arr = new double[count];

  // Инициируем массив случайными числами
  for(int i = 0; i < count; i++)
  {
    arr[i] = (double)rand()/RAND_MAX;
    cout << arr[i] << endl;
  }

  // Освобождаем память, отведенную под динамический массив
  delete [] arr;

  return 0;
}
```

Теперь, когда массив со случайным количеством элементов, принимающих случайные значения, сформирован, можно приступать к поиску максимального и минимального элементов массива. Для этого обычно вводят перемен-

ную, которая будет содержать текущее максимальное (минимальное) значение, инициируя первым элементом массива. Затем, в цикле данная переменная сравнивается со всеми элементами массива. Если производится поиск максимального значения и переменная меньше, чем один из элементов массива, ей присваивается значение данного элемента. При поиске минимального значения переменной присваивается значение элемента массива, только если последний меньше значения переменной (листинг II.2.24).

Листинг II.2.24. Поиск максимального и минимального значений

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

int main()
{
  // Максимальное количество случайных элементов массива
  const int NUMBER_RAND_MAX = 100;
  // Инициируем генератор случайных чисел
  srand(time(NULL));
  // Случайное количество элементов массива
  int count = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  // Выделяем память под динамический массив
  double *arr = new double[count];

  // Инициируем массив случайными числами
  for(int i = 0; i < count; i++)
  {
    arr[i] = (double)rand()/RAND_MAX;
  }

  // Производим поиск минимального и максимального значений
  double min_value = arr[0];
  double max_value = arr[0];
  for(int i = 0; i < count; i++)
  {
    if(min_value > arr[i]) min_value = arr[i];
    if(max_value < arr[i]) max_value = arr[i];
  }

  cout << "Минимальное значение — " << min_value << endl;
  cout << "Максимальное значение — " << max_value << endl;
```

```
  // Освобождаем память, отведенную под динамический массив
  delete [] arr;

  return 0;
}
```

II.2.19. Динамический многомерный массив

Двумерный массив эквивалентен двойному указателю, т. е. указателю на массив указателей `double **`. Задача сводится к тому, чтобы выделить память под такой двойной указатель. Здесь существуют два подхода. Первый из них заключается в том, чтобы последовательно выделить память под массив указателей, а затем в цикле под каждый из указателей массива. Такое решение представлено в листинге II.2.25.

Замечание

Здесь при генерации случайного значения столбцов и рядов двумерного массива следует проверять, не равно ли случайное число 0, и если равно, то присваивать случайному числу единицу.

Листинг II.2.25. Выделение памяти под двумерный массив

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

int main()
{
  // Максимальное количество случайных элементов массива
  const int NUMBER_RAND_MAX = 10;
  // Инициируем генератор случайных чисел
  srand(time(NULL));
  // Случайное количество рядов
  int row = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  if(!row) row = 1;
  // Случайное количество столбцов
  int col = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  if(!col) col = 1;

  // Выделяем память под массив указателей
  double **arr = new double *[row];
```

```
  // Выделяем память под указатели массива
  for(int i = 0; i < row; i++)
  {
    arr[i] = new double[col];
  }

  // Освобождаем память, выделенную под указатели массива
  for(int i = 0; i < row; i++)
  {
    delete [] arr[i];
  }
  // Освобождаем память, выделенную под массив указателей
  delete [] arr;

  return 0;
}
```

Программа из листинга II.2.25 создает динамический двумерный массив с количеством столбцов `col` и количеством строк `row` (рис. II.2.5).

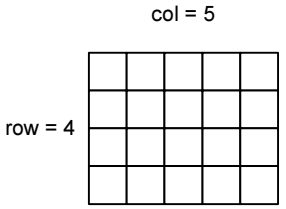


Рис. II.2.5. Прямоугольная матрица 4×5

Как видно из листинга II.2.25, под двойной указатель `arr` при помощи оператора `new` выделяется память под массив указателей. Следует обратить внимание, что вместо обычного типа `double` используется тип указателя `double *`. Затем в цикле память выделяется под каждый элемент массива указателей, здесь уже после оператора `new` используется обычный тип `double`. Для того чтобы освободить память, занимаемую двумерным динамическим массивом, следует пройти обратный путь: в цикле освободить память, выделенную под каждый из указателей массива, после чего освободить память, выделенную под массив указателей.

Представленный вариант является вполне работоспособным, однако он чрезвычайно громоздок (особенно, когда приходится реализовывать проверку корректности выделения памяти). Кроме того, выделение памяти при помощи оператора `new` является чрезвычайно затратной по ресурсам операцией —

число таких операций стараются свести к минимуму и избегать использования их в массивах. Опытные разработчики выделяют память один раз на все количество элементов будущего массива. То есть память выделяется под одномерный массив с таким количеством элементов, которое содержится в двумерном массиве `count = row * col` (рис. II.2.6).

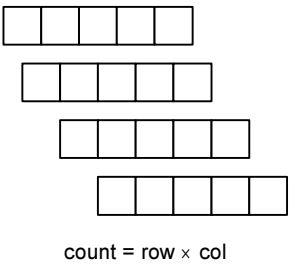


Рис. II.2.6. Одномерный массив с числом элементов, равным количеству элементов в двумерном массиве

При таком подходе выделение памяти, представленное в листинге II.2.25, можно переписать так, как это показано в листинге II.2.26.

Листинг II.2.26. Выделение памяти под одномерный массив

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

int main()
{
  // Максимальное количество случайных элементов массива
  const int NUMBER_RAND_MAX = 10;
  // Инициируем генератор случайных чисел
  srand(time(NULL));
  // Случайное количество рядов
  int row = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  if(!row) row = 1;
  // Случайное количество столбцов
  int col = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  if(!col) col = 1;

  // Выделяем память под массив указателей
  double *arr = new double [row*col];
```

```
  // Освобождаем память, выделенную под массив указателей
  delete [] arr;

  return 0;
}
```

Память в компьютере линейна, и любой двумерный массив располагается в развернутом состоянии: одна строка следует за другой. Точно так же можем поступить и мы: на основании координат (столбец, строка) вычислять линейную координату. Пусть индекс `i` пробегает значения от 0 до `row`, а индекс `j` — от 0 до `col`. Тогда линейный индекс `index` одномерного массива можно вычислить по формуле:

```
i * col + j
```

В листинге II.2.27 с использованием этой формулы случайными значениями заполняется двумерный массив, после чего его содержимое выводится в стандартный поток. В этом же цикле подсчитывается сумма всех элементов массива.

Листинг II.2.27. Сумма элементов матрицы

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

int main()
{
  // Максимальное количество случайных элементов массива
  const int NUMBER_RAND_MAX = 10;
  // Инициируем генератор случайных чисел
  srand(time(NULL));
  // Случайное количество рядов
  int row = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  if(!row) row = 1;
  // Случайное количество столбцов
  int col = int((double)rand()/RAND_MAX*NUMBER_RAND_MAX);
  if(!col) col = 1;

  // Выделяем память под массив указателей
  double *arr = new double [row*col];

  for(int i = 0; i < row; i++)
  {
```

```
    for(int j = 0; j < col; j++)
    {
      arr[i * col + j] = (double)rand()/RAND_MAX;
    }
  }

  // Выводим содержимое массива
  double summ = 0;
  for(int i = 0; i < row; i++)
  {
    for(int j = 0; j < col; j++)
    {
      cout << arr[i * col + j] << " ";
      summ += arr[i * col + j];
    }
    cout << "\n";
  }
  cout << "Сумма элементов матрицы " << row << " на " << col << " = "
       << summ << endl;

  // Освобождаем память, выделенную под массив указателей
  delete [] arr;

  return 0;
}
```

II.2.20. Заполнение элементов массива

Для того чтобы диагональный элемент матрицы был равным единице, а смежные элементы увеличивались на единицу, при удалении диагонали следует вычислять значение элемента массива по формуле:

```
abs(i − j) + 1
```

Здесь `i` пробегает все столбцы матрицы, а `j` — строки. Разницу значений `i` и `j` следует брать по модулю, т. е. без учета знака. Для операции взятия по модулю в стандартной библиотеке `<math>` предусмотрена функция `abs()` (листинг II.2.28).

Листинг II.2.28. Вывод матрицы

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

```
int main()
{
  int number = 0;

  cout << "Введите размер матрицы (1-10) ";
  cin >> number;

  if(number <=0 || number > 10)
  {
    cout << "Размер матрицы следует задать в диапазоне от 1 до 10";
    return 1;
  }

  // Выделяем память размером number * number
  int *arr = new int[number*number];

  // Формируем элементы массива
  for(int i = 0; i < number; i++)
  {
    for(int j = 0; j < number; j++)
    {
      arr[i * number + j] = abs(i - j) + 1;
    }
  }

  // Выводим массив в стандартный поток вывода
  for(int i = 0; i < number; i++)
  {
    for(int j = 0; j < number; j++)
    {
      cout << arr[i * number + j] << " ";
    }
    cout << "\n";
  }

  // Освобождаем ранее выделенную динамическую память
  delete [] arr;

  return 0;
}
```

Для того чтобы вывод матрицы был зеркальный, необходимо изменить порядок перебора индекса `j` во внутреннем цикле вывода массива в стандартный поток (листинг II.2.29).

Листинг II.2.29. Альтернативный вывод матрицы

```
...
  // Выводим массив в стандартный поток вывода
  for(int i = 0; i < number; i++)
  {
    for(int j = number — 1; j >= 0; j--)
    {
      cout << arr[i * number + j] << " ";
    }
    cout << "\n";
  }
...
```

Как видно из листинга II.2.29, индекс `j` вместо того, чтобы пробегать значения от 0 до `number` — 1 с последовательным увеличением индекса на каждой итерации цикла, уменьшается от `number` — 1 до 0.

II.2.21. Чем отличается *int * const* от *int const **?

Форма `int * const` означает константность указателя: указатель, объявленный с этим определением, не может менять адрес, на который он указывает (листинг II.2.30).

Листинг II.2.30. Использование `int * const`

```
int main()
{
  int number = 5;
  int var    = 10;
  int * const prt = &number;
  *prt = 100; // Допустимо
  prt = &var; // Ошибка

  return 0;
}
```

Форма `int const *` означает константность самого значения, на которое указывает указатель, т. е. адрес, на который указывает указатель можно менять, но изменить значение содержимого адреса при помощи указателя невозможно (листинг II.2.31).

Замечание

Модификатор `const` не позволяет модифицировать переменную только через указатель, саму переменную можно подвергать изменению.

Листинг II.2.31. Использование `int const *`

```
int main()
{
  int number = 5;
  int var    = 10;
  int const * prt = &number;
  prt = &var; // Допустимо
  var = 15;
  *prt = 100; // Ошибка

  return 0;
}
```

Если модификации не должен подвергаться ни сам указатель, ни значение, на которое он указывает, допускается использование типа `int const * const` (листинг II.2.32).

Листинг II.2.32. Использование `int const * const`

```
int main()
{
  int number = 5;
  int var    = 10;
  int const * const prt = &number;
  prt = &var; // Ошибка
  *prt = 100; // Ошибка

  return 0;
}
```

II.2.22. Отличие ссылки от указателя

Ссылка точно так же, как и указатель, содержит в качестве значения адрес другой переменной. В листинге II.2.33 объявляются указатель и ссылка на переменную `var`.

Листинг II.2.33. Сравнение ссылки и указателя

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

```
int main()
{
  int var    = 10;
  int *prt = &var;
  int &lnk = var;

  cout << *prt << "\n";
  cout << lnk << "\n";

  return 0;
}
```

В отличие от указателя при инициализации ссылки нет необходимости получать адрес переменной при помощи оператора `&`, а при дальнейших манипуляциях значением, на которую указывает ссылка, не нужно проводить операцию разыменования. Все это накладывает на ссылку ряд дополнительных ограничений по сравнению с указателями. Не допускается ссылаться на ссылку, организовывать двойные, тройные и т. д. ссылки. То есть тип `int &&` недопустим. По этой причине невозможно создать массив ссылок. Более того, адрес ссылки невозможно получить при помощи оператора `&`, поэтому указатель на ссылку также не возможен.

Следовательно, отвечая на вопрос задачи, можно утверждать, что ссылки и указатели отличаются и значительно.

II.2.23. Указатель и ссылка на структуру

Указатель на структуру объявляется точно так же, как и обычные переменные. Однако при обращении к элементам структуры посредством указателя необходимо использовать последовательность `->` вместо точки (листинг II.2.34).

Листинг II.2.34. Указатель на структуру

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

int main()
{
  struct person
  {
    char family[80];
    char patronymic[80];
```

```
    char name[80];
  } man;

  // Указатель на структуру
  person *per = &man;

  cout << "Введите фамилию ";
  fgets(per->family, 80, stdin);
  cout << "Введите имя ";
  fgets(per->name, 80, stdin);
  cout << "Введите отчество ";
  fgets(per->patronymic, 80, stdin);

  cout << per->family << per->name << per->patronymic << "\n";

  return 0;
}
```

При объявлении ссылки на структуру (листинг II.2.35) порядок работы с элементами структуры остается таким же, как с обычными структурами — обращение к элементам происходит через точку, а не через последовательность ->, как в случае указателей.

Листинг II.2.35. Ссылка на структуру

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

int main()
{
  struct person
  {
    char family[80];
    char patronymic[80];
    char name[80];
  } man;

  // Ссылка на структуру
  person &per = man;

  cout << "Введите фамилию ";
  fgets(per.family, 80, stdin);
  cout << "Введите имя ";
```

```
  fgets(per.name, 80, stdin);
  cout << "Введите отчество ";
  fgets(per.patronymic, 80, stdin);

  cout << per.family << per.name << per.patronymic << "\n";

  return 0;
}
```

II.2.24. Указатель на структуру

Для того чтобы объявить ссылку на указатель, удобно предварительно объявить при помощи конструкции `typedef` новый тип для `person *`. В листинге II.2.36 такой новый тип назван `prt_person`. После этого ссылку на указатель легко объявить при помощи конструкции `person &`. Так как ссылка ссылается на указатель, то для доступа к элементам структуры `person` следует использовать последовательность `->`, а не точку.

Листинг II.2.36. Указатель на структуру

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

struct person
{
  char family[80];
  char patronymic[80];
  char name[80];
};

typedef person * prt_person;

int main()
{
  person *prs = new person;

  prt_person &link_person = prs;

  cout << "Введите фамилию ";
  fgets(link_person->family, 80, stdin);
  cout << "Введите имя ";
  fgets(link_person->name, 80, stdin);
  cout << "Введите отчество ";
  fgets(link_person->patronymic, 80, stdin);
```

```
  cout << link_person->family << link_person->name
       << link_person->patronymic << "\n";

  delete prs;
}
```

II.2.25. Использование структур для хранения строк

Переменная `str` из листинга 2.12 ссылается на первый элемент структуры `word`, однако стандарт C или C++ вовсе не гарантирует, что элементы структур будут следовать один за другим и именно в том порядке, который определяется структурой. Более того, в большинстве случаев из-за выравнивания данных поля структуры не следуют одно за другим. Поэтому код из листинга I.2.12 в стандартный поток может выдать все, что угодно. Единственное, что можно утверждать, — первым символом будет выведен 'H', который хранится в элементе `wrd.ch1`.

II.2.26. Односвязный список

В листинге II.2.37 представлено возможное решение задачи по выводу элементов односвязного списка.

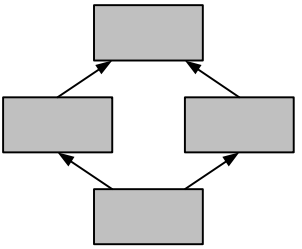
Листинг II.2.37. Вывод элементов односвязного списка

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

typedef struct point
{
  double x;
  double y;
  int next;
} point;

int main()
{
  point curve[] =
  {
    {1.0, 5.0, 2},
    {3.0, 6.0, 6},
    {2.0, 5.0, 1},
    {5.0, 6.0, 4},
```

```
    {6.0, 5.0, 5},
    {7.0, 6.0, -1},
    {4.0, 5.0, 3},
  };

  int current = 0;

  while(current != -1)
  {
    cout << "(x,y) = (" << curve[current].x << "," <<
         curve[current].y << ")\n";
    current = curve[current].next;
  }

  return 0;
}
```

Здесь удобнее воспользоваться циклом `while()`, который обходит массив по ссылкам односвязного списка.

ГЛАВА II.3



Функции

II.3.1. Подсчет числа вызовов функции

Переменные, объявленные в функциях, не сохраняют свое состояние от вызова к вызову. В этом легко убедиться, выполнив программу из листинга II.3.1.

Листинг II.3.1. Локальные переменные функции

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

void func()
{
  int var = 0;
  cout << var++ << endl;
}

int main()
{
  int var = 10;
  func();
  func();

  return 0;
}
```

Одним из решений проблемы является использование глобальной переменной, которая будет доступна во всех функциях программы (листинг II.3.2).

Листинг II.3.2. Использование глобальной переменной `var`

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

// Глобальная переменная
int var = 0;

void func()
{
  cout << var++ << endl;
}

int main()
{
  func();
  func();

  return 0;
}
```

Решение, представленное в листинге II.3.2, является работоспособным, но достаточно опасным, особенно в том случае, если проект разрастается. В объемном проекте становится все труднее и труднее контролировать состояние глобальной переменной. Так как ее может переписать любая функция в любое время. Поэтому когда требуется организовать счетчик вызовов функции или передавать состояние переменных от одного вызова к другому, прибегают к *статическим переменным*. Это переменные, которые объявляются с ключевым словом `static` и сохраняют свое значение между вызовами функций (листинг II.3.3). В отличие от глобальной переменной статическая переменная доступна только внутри функции и не может быть случайно изменена вне ее.

Листинг II.3.3. Использование статической переменной

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

void func()
{
  // Статическая переменная
  static int var = 0;
```

```
  cout << var++ << endl;
}

int main()
{
  func();
  func();

  return 0;
}
```

Следует обратить внимание, что функция `func()` объявляется до использования ее в теле программы. Если необходимо поместить реализацию функции после ее первого вызова, то следует в начале указать *прототип функции* — ее имя, список параметров и тип возвращаемого значения, которые заканчиваются точкой с запятой (листинг II.3.4). Прототип функции необходим компилятору, чтобы корректно обрабатывать вызовы функции до того, как он встретит ее реализацию.

Листинг II.3.4. Использование прототипа функции

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

// Прототип функции
void func();

int main()
{
  func();
  func();

  return 0;
}

void func()
{
  // Статическая переменная
  static int var = 0;
  cout << var++ << endl;
}
```

II.3.2. Подсчет среднего значения

В функции `arv_number()`, подсчитывающей среднее арифметическое, удобно ввести две статические переменные — `number` и `summ`, которые будут хранить количество и сумму введенных пользователем значений между вызовами функции. Функция, принимая новое значение через единственный параметр `var`, будет прибавлять его к значению переменной `summ`. При этом значение `number` будет увеличиваться на единицу при каждом вызове функции. В качестве результата будет возвращаться частное от `summ` и `number`.

В листинге II.3.5 представлено возможное решение задачи. Для опроса пользователя используется бесконечный цикл `while()`, выход из которого производится в том случае, если пользователь вводит число 0.

Листинг II.3.5. Подсчет среднего значения

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

double arv_number(double var);

int main()
{
  double var = 0;
  while(true)
  {
    cout << "Введите значение (0 для выхода) ";
    cin >> var;
    if(!var) break;
    cout << arv_number(var) << endl;
  }

  return 0;
}

double arv_number(double var)
{
  static double summ = 0;
  static int number = 0;
  summ += var;
  number++;
  return summ/number;
}
```

II.3.3. Обработка одномерного массива в функции

Функции могут принимать параметры по значению, указателю или ссылке. Если параметр передается по значению, на создание его копии расходуются значительные ресурсы, поэтому переменные, занимающие большие объемы памяти, стараются передать либо по значению, либо по ссылке. Массивы всегда передаются как указатели — это связано с их природой: они являются указателями, только имеющие отличную от классических указателей форму. Однако в функцию передается только указатель, определить размер массива по нему внутри функции уже не удастся. В этом можно легко убедиться, откомпилировав программу из листинга II.3.6.

Листинг II.3.6. Определение размера массива

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

void squared(int arr[]);

int main()
{
  int arr[] = {3, 4, 5, 6, 7, 8, 9, 10};

  cout << sizeof(arr)/sizeof(arr[0]) << endl;

  squared(arr);
  return 0;
}

void squared(int arr[])
{
  cout << sizeof(arr)/sizeof(arr[0]) << endl;
}
```

В функции `main()`, где объявлен массив, определить размер удается, а внутри функции `squared()` уже нет. Поэтому для определения массива потребуется либо вводить глобальную константу с числом элементов массива, либо добавить в функции, принимающие массив, дополнительный параметр с числом элементов в массиве. В листинге II.3.7 представлено возможное решение задачи.

Замечание

Вместо массива в качестве первого аргумента функций `squared()` и `print_arr()` может выступать указатель `int *`.

Листинг II.3.7. Возведение элементов массива в квадрат

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

void squared(int arr[], const int number);
void print_arr(int arr[], const int number);

int main()
{
  int arr[8] = {3, 4, 5, 6, 7, 8, 9, 10};

  squared(arr, sizeof(arr)/sizeof(arr[0]));
  print_arr(arr, sizeof(arr)/sizeof(arr[0]));
  return 0;
}

// Функция возведения элементов массива arr в квадрат
void squared(int arr[], const int number)
{
  for(int i = 0; i < number; i++)
  {
    arr[i] = arr[i]*arr[i];
  }
}

// Функция вывода элементов массива arr в окно браузера
void print_arr(int arr[], const int number)
{
  for(int i = 0; i < number; i++)
  {
    cout << arr[i] << "\n";
  }
}
```

II.3.4. Указатель на последний элемент массива

Функции могут возвращать значения не только через параметры, но и посредством оператора `return`, если функция имеет соответствующий прототип.

В листинге II.3.8 приводится пример функции `last_element()`, которая возвращает указатель на последний элемент массива `arr`.

Листинг II.3.8. Указатель на последний элемент массива

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

int *last_element(int arr[], const int number);

int main()
{
  int arr[8] = {3, 4, 5, 6, 7, 8, 9, 10};

  cout << *last_element(arr, sizeof(arr)/sizeof(arr[0]));
}

// Функция возведения элементов массива arr в квадрат
int *last_element(int arr[], const int number)
{
  return &arr[number - 1];
}
```

Полученный в результате вызова функции `last_element()` указатель не помещается в промежуточную переменную — непосредственно при выводе в стандартный поток производится его разыменование при помощи оператора `*`. Помимо указателя, в качестве возвращаемого значения может выступать и ссылка. В этом случае решение могло бы выглядеть так, как это представлено в листинге II.3.9.

Листинг II.3.9. Ссылка на последний элемент массива

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

int &last_element(int arr[], const int number);

int main()
{
  int arr[8] = {3, 4, 5, 6, 7, 8, 9, 10};

  cout << last_element(arr, sizeof(arr)/sizeof(arr[0]));
}
```

```
// Функция возведения элементов массива arr в квадрат
int &last_element(int arr[], const int number)
{
  return arr[number - 1];
}
```

II.3.5. Функция обмена значений двух переменных

Одним из самых простых решений является передача значения первой переменной временной переменной `temp`, значения второй переменной — первой, а значение временной `temp` переменной передать второй переменной (листинг II.3.10).

Листинг II.3.10. Обмен значений при помощи указателей

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

void swap(int *first, int *second);

int main()
{
  int first = 10;
  int second = 20;

  swap(&first, &second);
  cout << "first  " << first  << "\n";
  cout << "second " << second << "\n";
}

// Функция обмена значений
void swap(int *first, int *second)
{
  int temp;

  temp = *first;
  *first = *second;
  *second = temp;

  return;
}
```

Функция `swap()` принимает в качестве параметров указатели, т. к. результат работы функции передается в функцию `main()` через них. Если использовать обычные переменные типа `int`, то после того, как они будут изменены в функции `swap()`, их новые значения не будут доступны в функции `main()`.

Второй способ реализации функции обмена значений двух переменных заключается в использовании ссылок (листинг II.3.11).

Листинг II.3.11. Обмен значений при помощи ссылок

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

void swap(int &first, int &second);

int main()
{
  int first = 10;
  int second = 20;

  swap(first, second);
  cout << "first  " << first  << "\n";
  cout << "second " << second << "\n";
}

// Функция обмена значений
void swap(int &first, int &second)
{
  int temp;

  temp = first;
  first = second;
  second = temp;

  return;
}
```

Важно отметить, что код, представленный в листинге II.3.12, работать будет неправильно. Это связано с тем, что параметры `first` и `second` передаются по значению, а не по ссылке, и, несмотря на то, что значения указателей меняются внутри функции `swap()`, в функции `main()` они будут иметь те же самые адреса.

Листинг II.3.12. Ошибочное решение

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

void swap(int *first, int *second);

int main()
{
  int first = 10;
  int second = 20;

  swap(&first, &second);
  cout << "first  " << first  << "\n";
  cout << "second " << second << "\n";
}

// Функция обмена значений
void swap(int *first, int *second)
{
  int *temp;

  temp = first;
  first = second;
  second = temp;

  return;
}
```

Из этого следует важный вывод: если вы собираетесь обменивать значения через указатели, следует либо прибегать к двойному указателю, либо передавать указатели по ссылке (листинг II.3.13).

Листинг II.3.13. Передача указателя по ссылке

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

typedef int * prt;

void swap(prt &first, prt &second);

int main()
```

```
{
  int first = 10;
  int second = 20;
  prt fst_prt = &first;
  prt snd_prt = &second;

  swap(fst_prt, snd_prt);
  cout << "first  " << *fst_prt << "\n";
  cout << "second " << *snd_prt << "\n";
}

// Функция обмена значений
void swap(prt &first, prt &second)
{
  prt temp;

  temp = first;
  first = second;
  second = temp;

  return;
}
```

Как видно из листинга II.3.13, для типа `int *` при помощи конструкции `typedef` вводится новый тип `prt`. Это позволяет получить ссылку на указатель.

Оба представленных варианта (см. листинги II.3.10 и II.3.11) используют для обмена данных промежуточные значения. Для того чтобы избежать этого, потребуется осуществить несколько арифметических преобразований (листинг II.3.14).

Листинг II.3.14. Обмен значений без использования промежуточных переменных

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

void swap(int &first, int &second);

int main()
{
  int first = 10;
  int second = 20;
```

```
  swap(first, second);
  cout << "first  " << first  << "\n";
  cout << "second " << second << "\n";
}

// Функция обмена значений
void swap(int &first, int &second)
{
  first  = first + second;
  second = first - second;
  first  = first - second;

  return;
}
```

Как видно из листинга II.3.14, для обмена значений двух переменных не используется промежуточная переменная. Однако такой подход следует применять лишь в условиях жесткой нехватки памяти (и то стоит подумать, много памяти все равно не выиграть), т. к. решение становится не очевидным, и, если комментарий отсутствует, для того чтобы понять назначение функции `swap()`, стороннему разработчику потребуется затратить дополнительное время.

II.3.6. Рекурсивный вызов

Рекурсивной называется функция, которая вызывает сама себя. Программа, которая запрашивает у пользователя положительное целое число `number` и затем выводит убывающую последовательность от `number` до 0, может выглядеть так, как это представлено в листинге II.3.15.

Листинг II.3.15. Рекурсивная функция

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

void print_number(int number);

int main()
{
  int number = 0;
  cout << "Input sum in rubles ";
  cin >> number;
  if(number > 0) print_number(number);
}
```

```
// Функция обмена значений
void print_number(int number)
{
  cout << number << "\n";
  if(number > 0) print_number(number - 1);

  return;
}
```

Функция `print_number()` из листинга II.3.15 вызывает сама себя до тех пор, пока значение аргумента `number` не достигнет нулевого значения. Обычно рекурсий стараются избегать, т. к. они являются сложными для восприятия и отладки. Ошибка в логике завершения рекурсивных функций может приводить к бесконечному вызову функций или зацикливанию — отлаживать такие ситуации чрезвычайно сложно. Обычно рекурсивные функции используют, когда природа самих данных рекурсивна, например, при обходе дерева (вложенных каталогов с файлами, иерархии и т. п.).

II.3.7. Переменная сумма

Организовать подсчет переменного числа сумм при помощи вложенных циклов крайне затруднительно, т. к. степень вложения постоянно изменяется в зависимости от того, какое число введет пользователь. Поэтому для решения задачи удобно прибегнуть к рекурсивному спуску. Сначала будет подсчитана крайняя правая сумма от 0 до `number`, которая будет использоваться, потом вторая, затем третья и т. д. В листинге II.3.16 представлена программа, реализующая такой подход.

Листинг II.3.16. Подсчет переменного числа сумм

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

int sum_number(int number, int factor);

int main()
{
  int number = 0;
  cout << "Input sum in rubles ";
  cin >> number;
  if(number > 0) cout << sum_number(number, 1) << "\n";
}
```

```
int sum_number(int number, int factor)
{
  int sum = 0;
  for(int i = 0; i < number; i++)
  {
    sum += factor;
  }

  if(number - 1 > 0) return sum_number(number - 1, sum);
  else return sum;
}
```

Решить представленную задачу также можно, не прибегая к рекурсии — итеративным способом (листинг II.3.17).

Листинг II.3.17. Итеративный вариант решения задачи

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

int sum_number(int number, int factor);

int main()
{
  int number = 0;
  cout << "Input sum in rubles ";
  cin >> number;
  if(number > 0) cout << sum_number(number, 1) << "\n";
}

int sum_number(int number, int factor)
{
  int sum = 0;
  for(int i = number; i > 0; i--)
  {
    sum = factor*i;
    factor = sum;
  }

  return sum;
}
```

Второй вариант, без использования рекурсии более предпочтителен, т. к. он потребляет меньше ресурсов.

II.3.8. Допустимо ли выражение *f() = 10.0* ?

Использование функций в левой части вполне допускается и даже имеет смысл, если функция возвращает ссылку или указатель на глобальную переменную (листинг II.3.18).

Листинг II.3.18. Использование функции в левой части оператора "равно"

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

double val = 0;
double &f();

int main()
{
  cout << val << "\n";
  f() = 10.0;
  cout << val << "\n";
}

double &f()
{
  return val;
}
```

Так как функция `f()` из листинга II.3.18 возвращает ссылку на глобальную переменную, то переменная принимает значение `10.0`, а результат выполнения программы выглядит следующим образом:

```
0.0
10.0
```

II.3.9. Предотвращение выхода за границы массива

Так как сам язык C++ не отслеживает ситуации выхода за границы массива, поэтому часто прибегают к созданию искусственного интерфейса к массивам, который позволяет создать такой контроль. Для этого часто прибегают к функциям, которые принимают в качестве аргумента элемент массива и контролируют, не принимает ли он значения меньше 0 или больше, чем количество элементов в массиве (листинг II.3.19).

Листинг II.3.19. Предотвращение выхода за границы массива

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

// Число элементов в массиве
const int COUNT = 100;
int arr[COUNT];

int &put(int);
int get(int);

int main()
{
  // В цикле заполняем массив arr[] квадратами чисел
  for(int i = 0; i < COUNT; i++) put(i) = i*i;
  // В цикле выводим содержимое массива arr[]
  for(int i = 0; i < COUNT; i++) cout << get(i) << " ";
  cout << "\n";
}

int &put(int index)
{
  if(index >= 0 && index < COUNT) return arr[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}
int get(int index)
{
  if(index >= 0 && index < COUNT) return arr[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
    return -1;
  }
}
```

II.3.10. Вывод строки в стандартный поток

Решить данную задачу можно двумя способами. Первый из них заключается в перегрузке функции. C++ допускает объявление нескольких функций с одним и тем же именем, но отличающихся количеством, типом параметров или

типом возвращаемого значения. В листинге II.3.20 приводится возможное решение задачи с использованием перегрузки функции `print_str()`.

Листинг II.3.20. Перегрузка функции `print_str()`

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

void print_str(char *str, int count);
void print_str(char * str);

int main()
{
  char *str = "Hello, world!";
  print_str(str, 4);
  print_str(str);
}

void print_str(char *str, int count)
{
  if(count < strlen(str))
  {
    for(int i = 0; i < count; i++) cout << str[i];
  }
  else
  {
    cout << str;
  }
  cout << "\n";
}

void print_str(char * str)
{
  const char MAX_CHAR = 30;
  if(MAX_CHAR < strlen(str))
  {
    for(int i = 0; i < MAX_CHAR; i++) cout << str[i];
  }
  else
  {
    cout << str;
  }
  cout << "\n";
}
```

Как видно из листинга II.3.20, в программе реализуются два варианта функции `print_str()`. Первый вариант, принимающий два параметра, используется, когда функции передаются два аргумента. Второй вариант с одним параметром применяется, когда функции передается лишь одна строка. Результат работы программы из листинга II.3.20 выглядит следующим образом:

```
Hell
Hello, world!
```

Второе решение поставленной задачи связано с необязательными параметрами, т. е. параметрами, которые принимают значение по умолчанию и которые можно не передавать при вызове функции (листинг II.3.21).

Замечание

Необязательные параметры всегда должны идти в конце, после обязательных параметров. Если необязательных параметров несколько и при вызове функции необходимо передать значение необязательному параметру, перед которым расположены другие необязательные параметры, этим параметрам также придется присвоить значения.

Листинг II.3.21. Использование необязательного параметра

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

void print_str(char *str, int count = 30);

int main()
{
  char *str = "Hello, world!";
  print_str(str, 4);
  print_str(str);
}

void print_str(char *str, int count)
{
  if(count < strlen(str))
  {
    for(int i = 0; i < count; i++) cout << str[i];
  }
  else
  {
    cout << str;
  }
  cout << "\n";
}
```

Следует отметить, что значение по умолчанию необязательному параметру нужно присваивать только один раз, при самом первом объявлении. Если прототип функции не используется, то значение по умолчанию необходимо назначать непосредственно в объявлении функции (листинг II.3.22).

Листинг II.3.22. Значение по умолчанию в объявлении функции

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

void print_str(char *str, int count = 30)
{
  if(count < strlen(str))
  {
    for(int i = 0; i < count; i++) cout << str[i];
  }
  else
  {
    cout << str;
  }
  cout << "\n";
}

int main()
{
  char *str = "Hello, world!";
  print_str(str, 4);
  print_str(str);
}
```

II.3.11. Функции *abs()*, *labs()* и *fabs()*

Дело в том, что функции `abs()`, `labs()` и `fabs()` перешли в язык C++ из стандартной библиотеки языка C, в котором операция перегрузки функций не предусмотрена. Их названия сохранены, чтобы обеспечить обратную совместимость с C-кодом. Однако ничто не мешает перегрузить собственную функцию `number_abs()` для того, чтобы она поддерживала остальные типы аргументов (листинг II.3.23).

Листинг II.3.23. Перегрузка функции `number_abs()`

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

```
int number_abs(int var);
long int number_abs(long int var);
double number_abs(double var);

int main()
{
  int var = -10;
  long int lvar = -10;
  double dvar = -10.0;
  cout << number_abs(var)  << "\n";
  cout << number_abs(lvar) << "\n";
  cout << number_abs(dvar) << "\n";
  return 0;
}

int number_abs(int var)
{
  if(var > 0) return var;
  else return -var;
}

long int number_abs(long int var)
{
  if(var > 0) return var;
  else return -var;
}

double number_abs(double var)
{
  if(var > 0) return var;
  else return -var;
}
```

II.3.12. Ошибка в перегрузке функции

При использовании перегрузки, представленной в листинге I.3.2, возникает так называемая *ошибка неоднозначности*, когда компилятор не может определить, какую из функций следует вызвать. Следует либо отказаться от такого способа перегрузки, либо вместо целого значения `20` передать дробное значение `20.0`, которое будет автоматически приведено к типу `double`, либо явно указать тип передаваемой переменной при помощи операции приведения типа (листинг II.3.24).

Листинг II.3.24. Явное приведение типа аргумента

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

float low(float var);
double low(double var);

int main()
{
  cout << low((float)20) << "\n";

  return 0;
}

float low(float var)
{
  return sin(var);
}

double low(double var)
{
  return cos(var);
}
```

II.3.13. Функция с переменным количеством параметров

Помимо перегрузки функции и необязательных параметров язык C/C++ допускает использование функций с переменным количеством параметров. Для объявления такой функции последним параметром следует задать многоточие:

тип_функции имя_функции(*явные_параметры*, ...)

Количество параметров в таких функциях определяется не в момент компиляции, а в момент выполнения. Основная сложность заключается в том, что встроенных механизмов для определения типа и количества неявных параметров не существует. Обычно такие механизмы контроля реализуют за счет явных параметров.

Для решения задачи в листинге II.3.25 объявляется функция `summa()`, первый параметр `count` которой передает количество последующих параметров.

Листинг II.3.25. Суммирование произвольного количества параметров

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

long int summa(int count, ...);

int main()
{
  cout << "summa = " << summa(6, 6, 2, 9, 12, 4, 2) << "\n";

  return 0;
}

long int summa(int count, ...)
{
  int *prt = &count;

  long total = 0;
  for(int i = 0; i < count; i++) total += *(++prt);

  return total;
}
```

Как видно из листинга II.3.25, для получения значений последующих параметров функции `summa()` объявляется указатель на целый тип `prt`. Это означает, что в качестве параметров функции мы ждем только целочисленные переменные. Далее в цикле `for` мы увеличиваем значение указателя `prt` на единицу, перебирая тем самым все неявные параметры. Функцию `summa()` из листинга II.3.25 можно переписать с использованием индексов, т. к. указатели и массивы взаимозаменяемы (листинг II.3.26).

Листинг II.3.26. Использование индексов, вместо адресной арифметики указателей

```
long int summa(int count, ...)
{
  int *prt = &count;

  long total = 0;
  for(int i = 0; i < count; i++) total += prt[i + 1];

  return total;
}
```

II.3.14. Указатель на функцию

Указатель на функцию позволяет ссылаться на функцию и вызывать ее при помощи указателя — это очень удобное средство, предоставляющее возможность динамически расширять системы. Функция, принимающая в качестве аргумента указатель на другую функцию, в дальнейшем может быть помещена в библиотеку. Если программист, пользующийся библиотекой, решит изменить внешнюю функцию или добавить новую, перекомпиляция библиотеки не потребуется, функция может быть добавлена динамически.

В общем виде указатель на функцию объявляется по схеме:

```
тип_функции (*имя_указателя) (параметры);
```

Вызов функции по указателю производится при помощи операции разыменования:

```
(*имя_указателя) (параметры);
```

В листинге II.3.27 объявляется указатель на функцию `prt`, которому последовательно присваиваются указатель на `f1()` и указатель на `f2()`.

Листинг II.3.27. Объявление указателя на функцию

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

void f1(void);
void f2(void);

int main()
{
  void (*prt) (void);
  prt = f2;
  (*prt)(); // Вызывается функция f2()
  prt = f1;
  (*prt)(); // Вызывается функция f1()

  return 0;
}

void f1(void)
{
  cout << "Выполняется функция f1()\n";
}
```

```
void f2(void)
{
  cout << "Выполняется функция f2()\n";
}
```

Объявление указателя на функцию достаточно громоздко, поэтому часто прибегают к объявлению нового типа при помощи конструкции `typedef` (листинг II.3.28).

Листинг II.3.28. Объявление типа `prt_type` — указателя на функцию

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

void f1(void);
void f2(void);

typedef void (*prt_type) (void);

int main()
{
  prt_type prt;
  prt = f2;
  (*prt)(); // Вызывается функция f2()
  prt = f1;
  (*prt)(); // Вызывается функция f1()

  return 0;
}
```

II.3.15. Обработка функцией элементов массива

В листинге II.3.29 приводится решение поставленной задачи. Функция `change()` принимает в качестве первого аргумента массив `double`, а в качестве второго — указатель на функцию.

Листинг II.3.29. Обработка каждого элемента массива функцией

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

```
// Количество элементов в массиве
const int COUNT = 10;
// Указатель на функцию
typedef double (*prt) (double);
// Прототип функции
void trig(double *arr, prt tr);

int main()
{
  double arr[COUNT];

  // Инициируем массив
  for(int i = 0; i < COUNT; i++) arr[i] = COUNT*i;

  trig(arr, sin);

  for(int i = 0; i < COUNT; i++) cout << arr[i] << "\n";

  return 0;
}

void trig(double *arr, prt tr)
{
  for(int i = 0; i < COUNT; i++)
  {
    arr[i] = (*tr)(arr[i]);
  }
  return;
}
```

Стандартные функции `sin()` и `cos()` имеют несколько перегруженных вариантов. Так как в листинге II.3.29 объявляется массив типа `double`, используются варианты функций, принимающих и возвращающих переменные типа `double`. Для удобства при помощи конструкции `typedef` объявляется новый тип `prt` — указатель на функцию, который в равной степени подходит и для `sin()`, и для `cos()`.

II.3.16. Односвязный список

Для того чтобы организовать односвязный список, следует создать структуру `element`, которая ссылалась бы на точно такую же структуру (листинг II.3.30). Структуры `element` будут ссылаться друг на друга через поле `next`, последняя структура в списке в данном поле будет иметь значение `NULL`.

Листинг II.3.30. Структура `element`

```
typedef struct element
{
  int number;
  struct element *next;
} element;
```

Для создания нового элемента структуры создадим функцию `add_element()`, которая будет принимать в качестве первого аргумента двойной указатель на структуру `element`, перед которой будет вставляться новый элемент. Второй параметр будет содержать значение `number`. Реализация функции `add_element()` представлена в листинге II.3.31.

Листинг II.3.31. Добавление элемента в односвязный список

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

typedef struct element
{
  int number;
  struct element *next;
} element;

// Добавление нового элемента
void add_element(element **begin, int number);

int main()
{
  // Указатель на начало списка
  element *list = NULL;
  add_element(&list,9);
  add_element(&list,5);
  add_element(&list,8);
  add_element(&list,1);
  add_element(&list,2);

  return 0;
}

// Добавление элемента
void add_element(element **begin, int number)
```

```
{
  element *new_element = new element;
  new_element->number = number;
  // Если список пуст
  if(*begin == NULL)
  {
    new_element->next = NULL;
    *begin = new_element;
  }
  // Вставка элемента после текущего
  else
  {
    new_element->next = (*begin)->next;
    (*begin)->next = new_element;
  }
}
```

Как видно из листинга II.3.31, в функции `main()` объявляется лишь указатель `list` на структуру `element`. Вначале указатель имеет значение `NULL`, что означает, что он пуст. Функция `add_element()` принимает адрес указателя `list` и оперирует двойным указателем. Мы будем вынуждены оперировать двойным указателем, т. к. функция `add_element()` должна возвращать в функцию `main()` новое положение вершины односвязного списка.

Для того чтобы обратиться к полю `next` структуры `begin`, необходимо сначала разыменовать структуру, а потом применить символ `->`. В отличие от указателя двойной символ `->` в C++ не предусмотрен. Так как приоритет оператора `*` выше оператора `->`, разыменование двойного указателя `begin` заключается в круглые скобки `(*begin)`.

Рассмотрим, как происходит добавление новых элементов в односвязный список (рис. II.3.1).

Как видно из рис. II.3.1, новый элемент добавляется всегда справа от элемента, указатель на который передается функции `add_element()`. Таким образом, двойной указатель `list` (см. листинг II.3.31) на протяжении работы всей программы указывает на первый элемент списка.

Чтобы проконтролировать содержимое односвязного списка, создадим функцию `print_lists()`, которая будет принимать указатель на элемент списка и распечатывать его содержимое, начиная с текущего элемента (листинг II.3.32).

Замечание

Функция `print_list()` может принимать обычный указатель на структуру `element`, т. к. не подвергает изменению адрес элементов.

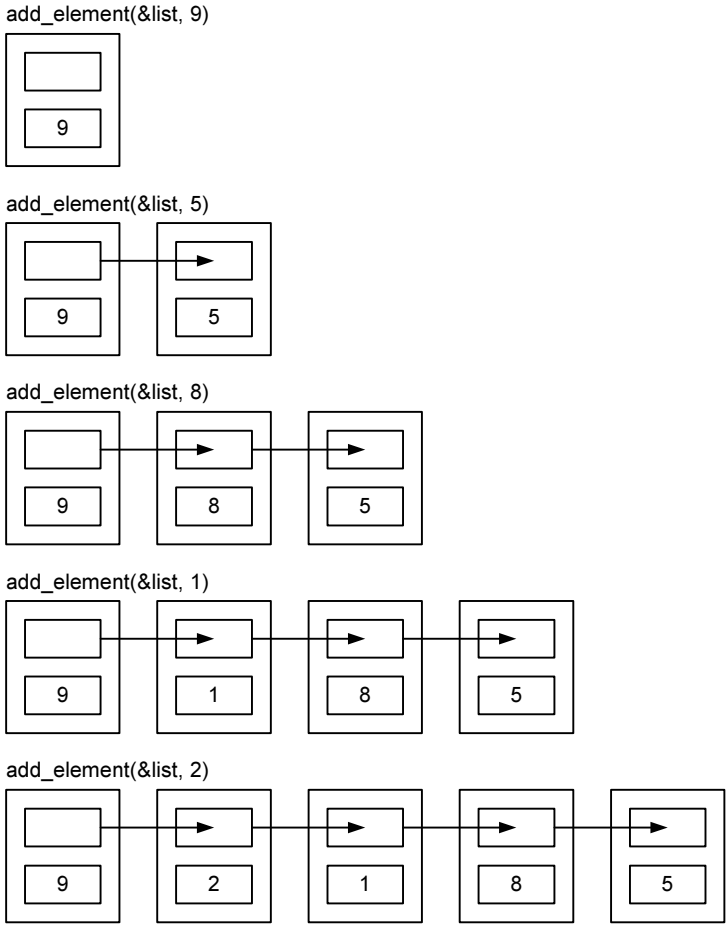


Рис. II.3.1. Добавление элементов в односвязный список

Листинг II.3.32. Вывод содержимого односвязного списка

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

typedef struct element
{
  int number;
  struct element *next;
} element;

// Добавление нового элемента
void add_element(element **begin, int number);
```

```
// Выводим содержимое списка
void print_lists(element *list);

int main()
{
  // Указатель на начало списка
  element *list = NULL;
  add_element(&list,9);
  add_element(&list,5);
  add_element(&list,8);
  add_element(&list,1);
  add_element(&list,2);

  print_lists(list);
  cout << "\n";

  return 0;
}

// Выводим содержимое списка
void print_lists(element *list)
{
  for(element *current = list; current != NULL; current = current->next)
  {
    // Выводим содержимое односвязного списка
    cout << current->number << "\n";
  }
}
```

Как видно из листинга II.3.32, содержимое списка перебирается в цикле `for` до тех пор, пока поле `next` очередной структуры не примет значение `NULL`. Скрипт из листинга II.3.32 выведет такую последовательность чисел:

```
9
2
1
8
5
```

При создании односвязного списка посредством функции `add_element()` динамически выделяется память с помощью оператора `new`. Если не предусмотреть способ возврата этой памяти, будет происходить утечка памяти.

Замечание

Утечка памяти не так страшна, если программа работает несколько секунд и потребляет несколько килобайт. Однако для серверов, которые работают дли-

тельное время, утечка памяти превращаются в настоящую проблему: серверы потребляют память и не возвращают ее в систему, администраторы вынуждены специально время от времени перезагружать серверы, чтобы освободить захваченную ими память.

В листинге II.3.33 представлена функция `delete_element()`, уничтожающая элемент, указатель на который передается ей в качестве аргумента, и возвращающая указатель на следующий элемент. Так как указатель должен вернуться в функцию `main()`, используется двойной указатель.

Замечание

В листинге II.3.33 не приводится реализация ранее рассмотренных функций `add_element()` и `print_lists()` — указываются лишь их прототипы.

Листинг II.3.33. Удаление элемента односвязного списка

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

typedef struct element
{
  int number;
  struct element *next;
} element;

// Добавление нового элемента
void add_element(element **begin, int number);
// Выводим содержимое списка
void print_lists(element *list);
// Удаление элемента списка
void delete_element(element **list);

int main()
{
  // Указатель на начало списка
  element *list = NULL;
  add_element(&list,9);
  add_element(&list,5);
  add_element(&list,8);
  add_element(&list,1);
  add_element(&list,2);

  delete_element(&list);
```

```
  print_lists(list);
  cout << "\n";

  return 0;
}

// Удаление элемента списка
void delete_element(element **list)
{
  element *next = NULL;

  if((*list)->next != NULL)
  {
    next = (*list)->next;
    delete *list;
    *list = next;
  }
}
```

В функции `delete_element()` промежуточному указателю `next` присваивается значение, на которое указывает переданный элемент списка `list`. После этого память, которая отводилась под текущий элемент, освобождается при помощи оператора `delete`, а указателю на начало списка присваивается значение `next`. Схематично работа функции `delete_element()` изображена на рис. II.3.2.

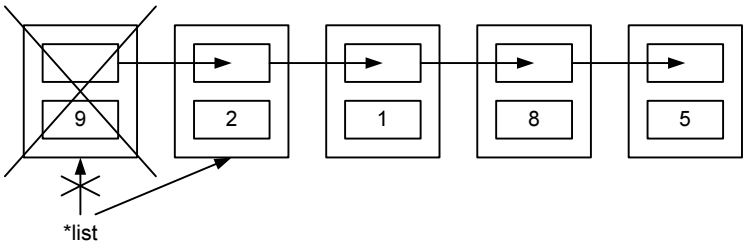


Рис. II.3.2. Удаление первого элемента односвязного списка

Результат работы программы из листинга II.3.33 выглядит следующим образом:

```
2
1
8
5
```

Для того чтобы удалить все элементы односвязного списка, необходимо прибегнуть к циклу, который последовательно будет уничтожать элементы от текущего до того, который ссылается на NULL. В листинге II.3.34 представлена функция delete_list(), которая уничтожает все элементы односвязного списка.

Замечание

В листинге II.3.34 не приводится реализация ранее рассмотренных функций add_element() и print_lists() — указываются лишь их прототипы.

Листинг II.3.34. Удаление всех элементов односвязного списка

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

typedef struct element
{
  int number;
  struct element *next;
} element;

// Добавление нового элемента
void add_element(element **begin, int number);
// Выводим содержимое списка
void print_lists(element *list);
// Удаление всех элементов
void delete_list(element **list);

int main()
{
  // Указатель на начало списка
  element *list = NULL;
  add_element(&list,9);
  add_element(&list,5);
  add_element(&list,8);
  add_element(&list,1);
  add_element(&list,2);

  delete_element(&list);
  delete_list(&list);

  print_lists(list);
```

```
  return 0;
}

// Удаление всех элементов
void delete_list(element **list)
{
  element *next = NULL;

  if((*list)->next != NULL)
  {
    do
    {
      next = (*list)->next;
      delete *list;
      *list = next;
    } while(next != NULL);
  }
}
```

II.3.17. Двухсвязный список

Двухсвязный список отличается от односвязного тем, что каждый его элемент ссылается не только на следующий элемент списка, но также и на предыдущий. Поэтому для реализации элемента двухсвязного списка подойдет структура `element`, представленная в листинге II.3.35.

Листинг II.3.35. Структура `element`

```
typedef struct element
{
  int number;
  // Ссылка на следующий элемент списка
  struct element *next;
  // Ссылка на предыдущий элемент списка
  struct element *prev;
} element;
```

Поле `number` структуры `element` содержит число, указатель `next` как ссылку на следующий элемент списка, а `prev` — на предыдущий элемент списка.

Для удобства построения новых элементов списка создадим функцию `create_element()`, которая будет выделять память под новый элемент, заполняя указатели `next` и `prev` значениями `NULL` (листинг II.3.36). Эта функция

удобна для выделения памяти под элементы, в том числе и первый элемент, который послужит началом списка.

Листинг II.3.36. Выделение памяти под элементы списка

```
// Выделение памяти под элемент списка
element *create_element(int number)
{
  element *item = new element;
  item->next = NULL;
  item->prev = NULL;
  item->number = number;
  return item;
}
```

Функция `create_element()` принимает целочисленное значение `number` для инициализации элемента и возвращает указатель `element *` на экземпляр структуры `create_element()`.

Теперь, когда можно создавать отдельные экземпляры списка, реализуем функции добавления нового элемента в список слева `add_prev()` и справа `add_next()` от текущего элемента. Реализация функции `add_next()` может выглядеть так, как это представлено в листинге II.3.37.

Листинг II.3.37. Добавление нового элемента справа от текущего элемента

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

typedef struct element
{
  int number;
  // Ссылка на следующий элемент списка
  struct element *next;
  // Ссылка на предыдущий элемент списка
  struct element *prev;
} element;

// Выделение памяти под элемент списка
element *create_element(int number);
// Добавление элемента справа от текущего
void add_next(element *current, element *new_element);
```

```
int main()
{
  // Указатель на начало списка
  element *list = create_element(9);
  add_next(list, create_element(2));
  add_next(list, create_element(1));
  add_next(list, create_element(8));
  add_next(list, create_element(5));

  return 0;
}

// Добавление элемента справа от текущего
void add_next(element *current, element *new_element)
{
  if(new_element != NULL && current != NULL)
  {
    new_element->next = current->next;
    new_element->prev = current;
    current->next = new_element;
    if(new_element->next != NULL) new_element->next->prev = new_element;
  }
}
```

Как видно из листинга II.3.37, функции `add_next()` передается сгенерированный при помощи функции `create_element()` элемент списка. Функция `add_next()` лишь исправляет ссылки в двухсвязном списке.

Для того чтобы добавить новый элемент справа от текущего, необходимо исправить четыре указателя. На рис. II.3.3 эти указатели изображены серым цветом. Элемент `current` — это элемент, после которого вставляется новая структура `new_element`, а `new_element->next` обозначает структуру, на которую ранее ссылался (при помощи указателя `next`) элемент `current`.

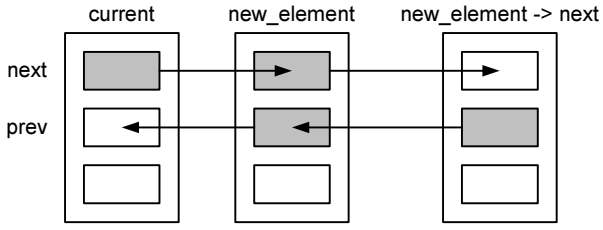


Рис. II.3.3. Вставка нового элемента `new_element` справа от текущего элемента `current`

Наличие в структуре `element` ссылки `prev` на предыдущий элемент позволяет получать доступ практически к любому элементу списка. В листинге II.3.37 указатель `list` ссылается на первый элемент очереди и позволяет получить содержимое его поля при помощи конструкции

```
list->number
```

Однако точно такой же результат можно получить при помощи конструкции

```
list->next->next->next->next->prev->prev->prev->prev->number
```

которая последовательно ссылается через указатели `next` на последний элемент списка, а затем возвращается через указатели `prev` к первому элементу списка.

По аналогии с функцией `add_next()`, вставляющей новый элемент списка справа от текущего, можно определить функцию `add_prev()`, которая будет вставлять новый элемент слева от текущего элемента списка (листинг II.3.38).

Листинг II.3.38. Добавление нового элемента слева от текущего элемента

```
// Добавление элемента слева от текущего
void add_prev(element *current, element *new_element)
{
  if(new_element != NULL && current != NULL)
  {
    new_element->next = current;
    new_element->prev = current->prev;
    current->prev = new_element;
    if(new_element->prev != NULL) new_element->prev->next = new_element;
  }
}
```

Функция `add_prev()` так же, как и `add_next()`, лишь исправляет четыре указателя трех элементов списка (рис. II.3.4).

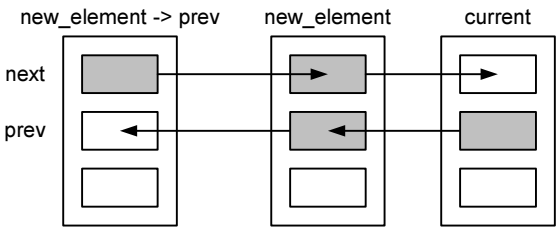


Рис. II.3.4. Вставка нового элемента `new_element` слева от текущего элемента `current`

Для удобства, реализуем функции, которые возвращают ссылку на первый (функция `get_first()`) и последний (функция `get_last()`) элементы списка (листинг II.3.39). Функции принимают в качестве единственного аргумента указатель на любой элемент списка.

Листинг II.3.39. Получение указателя на первый и последний элементы списка

```
// Получение первого элемента списка
element *get_first(element *list)
{
 if(list != NULL)
 {
   while(list->prev != NULL) list = list->prev;
 }
 return list;
}
// Получение последнего элемента списка
element *get_last(element *list)
{
 if(list != NULL)
 {
   while(list->next != NULL) list = list->next;
 }
 return list;
}
```

Для достижения конца или начала списка достаточно следовать ссылкам на следующий `next` или предыдущий `prev` элементы до тех пор, пока не встретится ссылка на `NULL`, что означает достижение конца ссылки. Теперь для того чтобы распечатать содержимое массива, можно перейти в его начало и выводить элементы в цикле до тех пор, пока ссылка на следующий элемент `next` не примет значение `NULL` (листинг II.3.40).

Листинг II.3.40. Вывод содержимого списка в стандартный поток

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

typedef struct element
{
  int number;
  // Ссылка на следующий элемент списка
  struct element *next;
```

```
  // Ссылка на предыдущий элемент списка
  struct element *prev;
} element;

// Выделение памяти под элемент списка
element *create_element(int number);
// Добавление элемента справа от текущего
void add_next(element *current, element *new_element);
// Добавление элемента слева от текущего
void add_prev(element *current, element *new_element);
// Получение первого элемента списка
element *get_first(element *list);
// Получение последнего элемента списка
element *get_last(element *list);
// Выводим содержимое списка
void print_list(element *list);

int main()
{
  // Указатель на начало списка
  element *list = create_element(9);
  add_next(list, create_element(2));
  add_next(list, create_element(1));
  add_next(list, create_element(8));
  add_next(list, create_element(5));

  print_list(list);

  return 0;
}

// Выводим содержимое списка
void print_list(element *list)
{
  if(list != NULL)
  {
    // Получаем первый элемент списка
    for(element *current = get_first(list);
        current != NULL;
        current = current->next)
    {
      // Выводим содержимое списка
      cout << current->number << "\n";
    }
  }
}
```

Результатом работы программы из листинга II.3.40 будет следующий столбец цифр:

```
9
5
8
1
2
```

Для подсчета количества элементов в списке (функция `get_count()`) можно реализовать отличный от функции `print_list()` механизм (листинг II.3.41).

Листинг II.3.41. Подсчет количества элементов в списке

```
// Подсчет числа элементов списка
int get_count(element *list)
{
  int count = 0;
  element *prev = list;
  element *next = list;
  if(list != NULL)
  {
    ++count;
    while(prev->prev != NULL)
    {
      prev = prev->prev;
      ++count;
    }
    while(next->next != NULL)
    {
      next = next->next;
      ++count;
    }
  }
  return count;
}
```

Как видно из листинга II.3.41, функция пробегает вправо и влево от текущего элемента по цепочке указателей `next` и `prev` до тех пор, пока не встретит значение `NULL`. Это позволяет избежать дополнительных издержек при поиске первого элемента списка.

Рассмотрим функцию `delete_element()`, удаляющую элемент списка (листинг II.3.42).

Листинг II.3.42. Удаление элемента списка

```
// Удаление элемента списка
void delete_element(element *element)
{
  if(element != NULL)
  {
    if(element->prev != NULL) element->prev->next = element->next;
    if(element->next != NULL) element->next->prev = element->prev;
    delete element;
  }
}
```

Прежде чем удалить элемент при помощи оператора `delete`, функция исправляет указатели `next` и `prev` соседних с удаляемым элементом структур. В листинге II.3.43 удаляется третий элемент списка.

Листинг II.3.43. Удаление третьего элемента списка

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

typedef struct element
{
  int number;
  // Ссылка на следующий элемент списка
  struct element *next;
  // Ссылка на предыдущий элемент списка
  struct element *prev;
} element;

// Выделение памяти под элемент списка
element *create_element(int number);
// Добавление элемента справа от текущего
void add_next(element *current, element *new_element);
// Получение первого элемента списка
element *get_first(element *list);
// Выводим содержимое списка
void print_list(element *list);
// Удаление элемента списка
void delete_element(element *element);
```

```
int main()
{
  // Указатель на начало списка
  element *list = create_element(9);
  add_next(list, create_element(2));
  add_next(list, create_element(1));
  add_next(list, create_element(8));
  add_next(list, create_element(5));

  print_list(list);
  cout << "\n";

  delete_element(get_first(list)->next->next);

  print_list(list);

  return 0;
}
```

В заключение рассмотрим функцию `delete_list()`, удаляющую весь список (листинг II.3.44). Это единственная функция из набора функций, обслуживающих двухсвязный список, которая принимает двойной указатель на элемент `element **`.

Листинг II.3.44. Удаление списка

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

typedef struct element
{
  int number;
  // Ссылка на следующий элемент списка
  struct element *next;
  // Ссылка на предыдущий элемент списка
  struct element *prev;
} element;

// Выделение памяти под элемент списка
element *create_element(int number);
// Добавление элемента справа от текущего
void add_next(element *current, element *new_element);
// Выводим содержимое списка
void print_list(element *list);
```

```
// Удаление всех элементов
void delete_list(element **list);

int main()
{
  // Указатель на начало списка
  element *list = create_element(9);
  add_next(list, create_element(2));
  add_next(list, create_element(1));
  add_next(list, create_element(8));
  add_next(list, create_element(5));

  delete_list(&list);

  print_list(list);

  return 0;
}

// Удаление всех элементов
void delete_list(element **list)
{
  if(*list != NULL)
  {
    // Получаем первый элемент списка
    for(element *current = get_first(*list)->next;
        current != NULL;
        current = current->next)
    {
      delete current->prev;
    }
    *list = NULL;
  }
}
```

Функция `delete_list()` устанавливает указатель на первый элемент списка и последовательно перебирает все элементы списка, удаляя предыдущий элемент.

II.3.18. Создание файла с уникальным именем

Язык C++, являясь системным языком программирования, предоставляет программистам множество способов работы с файлами, в первую очередь, позволяя выполнять системные вызовы UNIX или функции API Windows.

Несмотря на то, что использование средств операционной системы зачастую приводит к высокой производительности, более предпочтительными являются стандартные средства языка C++. Для работы с файлами можно использовать как функции стандартной библиотеки, перешедшей в C++ из C, так и объектно-ориентированные средства ввода/вывода, которые более подробно рассматриваются в ответах к задачам *главы 7*. Для решения данной задачи воспользуемся C-средствами языка программирования.

При работе с файлами используются функции из стандартной библиотеки `<stdio>`. Практически каждая из функций использует указатель на структуру `FILE`, определенную в заголовочном файле stdio.h. Непосредственно с полями этой структуры программист не работает, их заполняет и читает сама библиотека. Указатель на данную структуру нужен, чтобы различать, с каким файлом в данный момент работает программа (на тот случай, если работа ведется одновременно с несколькими файлами).

Замечание

Максимальное число одновременно открытых файлов определяется константой `FOPEN_MAX`.

Операции чтения и записи в файл осуществляются в несколько этапов:

1. Сначала выполняется открытие файла (получение указателя на структуру `FILE`) при помощи функции `fopen()`.
2. Далее производится операция чтения из файла при помощи одной из функций `getc()`, `fgetc()`, `fgets()`, `fread()`, `fprintf()` или запись при помощи одной из функций `putc()`, `fputc()`, `fputs()`, `fwrite()`, `fscanf()`.
3. Файл закрывается при помощи функции `fclose()`, тем самым библиотеке сообщается, что указатель на структуру `FILE` больше не нужен.

Кроме этого имеется еще несколько вспомогательных функций, которые будут рассмотрены далее.

Замечание

Файловые функции начинаются с символа `f`, для того чтобы подчеркнуть, что функции оперируют файлами, и из-за того, что имена вроде `open()`, `write()`, `read()` зарезервированы под системные вызовы.

Файл открывается при помощи функции `fopen()`, которая имеет следующий синтаксис:

```
FILE *fopen(const char *filename, const char *mode)
```

Функция принимает в качестве первого параметра имя файла *`filename`*, а в качестве второго параметра — режим открытия файла *`mode`*, который может принимать значения из табл. II.3.1.

Таблица II.3.1. *Режимы, в которых может быть открыт файл*

Режим	Значение
`r`	Открыть текстовый файл для чтения
`w`	Открыть текстовый файл для записи; если файл не существует, он будет создан
`a`	Открыть текстовый файл для записи в конец файла
`rb`	Открыть бинарный файл для чтения
`wb`	Открыть бинарный файл для записи; если файл не существует, он будет создан
`ab`	Открыть бинарный файл для записи в конец файла
`r+`	Открыть текстовый файл для чтения и записи
`w+`	Открыть текстовый файл для чтения и записи; если файл не существует, он будет создан
`a+`	Открыть текстовый файл для записи в конец файла или создать текстовый файл для чтения и записи
`r+b`	Открыть бинарный файл для чтения и записи
`w+b`	Открыть бинарный файл для чтения и записи; если файл не существует, он будет создан
`a+b`	Открыть бинарный файл для записи в конец файла или создать бинарный файл для чтения и записи

Структура `FILE` содержит поле, отмечающее текущий курсор, который в начале открытия файла в режимах `r` и `w` находится в начале файла. По мере чтения или записи информации он перемещается далее по файлу. В режиме `a` курсор устанавливается в конец файла, и информация может дописываться без затирания ранее записанных данных.

Для закрытия файла используется функция `fclose()`, которая имеет следующий синтаксис:

```
int fclose(FILE * fp)
```

Функция принимает указатель на структуру `FILE` открытого файла. Возвращает 0, если операция закрытия файла прошла успешно, или константу `EOF` в случае неудачи.

В листинге II.3.45 создается файл с именем text.txt. Так как используется режим `w`, то не существующий файл будет создан.

Листинг II.3.45. Создание файла text.txt

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

int main()
{
  FILE *fp;
  // Открываем файл
  fp = fopen("text.txt", "w")
  // Проверяем корректность открытия файла
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

В листинге II.3.45 указатель `fp` сравнивается со значением `NULL`, которое функция `fopen()` возвращает, если файл не может быть открыт (из-за переполнения диска или проблем с правами доступа). Код ошибки можно вывести при помощи функции `ferror()`, которая так же, как и функция `fclose()`, принимает указатель на структуру `FILE` открытого файла.

Функция `ferror()` имеет следующий синтаксис:

```
int ferror(FILE *fp)
```

Для того чтобы создать файл с уникальным именем, можно воспользоваться стандартной функцией `tmpnam()`, которая генерирует уникальное имя файла для текущего каталога и имеет следующий синтаксис:

```
char *tmpnam(char *name)
```

Функция принимает строку *name*, в которую помещается сгенерированное имя, причем длина строки не должна быть меньше константы `L_tmpnam`. В случае успеха функция возвращает указатель на строку *name*, в противном случае возвращается `NULL`.

Замечание

Помимо функции `tmpnam()` в библиотеке `<stdio>` имеется функция `tmpfile()`, которая непосредственно возвращает указатель `FILE` на открытый временный файл с уникальным именем в текущем каталоге. Однако для решения задачи функция не очень подходит, т. к. после закрытия файла при помощи функции `fclose()` файл автоматически уничтожается.

В листинге II.3.46 программа создает новый файл с уникальным именем.

Листинг II.3.46. Создание файла с уникальным именем

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

int main()
{
  FILE *fp;
  char tmpstr[L_tmpnam];
  // Открываем файл
  fp = fopen(tmpnam(tmpstr), "w");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

Теперь можно записывать ввод пользователя в файл, для чего можно использовать функцию `fwrite()`, которая имеет следующий синтаксис:

```
size_t fwrite(const void *buf, size_t size, size_t count, FILE *fp)
```

Функция принимает в качестве первого параметра *buf* указатель на область памяти, которую необходимо записать в открытый файл *fp*. Параметр `count` определяет, сколько элементов данных размером *size* должно быть записано в файл. Функция возвращает количество реально записанных элементов (если запись прошла успешно, то это количество равно значению параметра *count*). Следует обратить внимание на тот факт, что параметры и возвращаемое зна-

чение имеют тип `size_t`, который обычно определяется как целое без знака (`unsigned int`).

Для хранения текущей строки, в которую будет помещаться ввод пользователя перед тем, как сохранить его в файл, введем переменную `input`. Операцию чтения из стандартного потока и записи в файл можно осуществлять в цикле так, как это показано в листинге II.3.47.

Листинг II.3.47. Сохранение ввода пользователя в файл

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

int main()
{
  FILE *fp;
  char input[80] = "init";
  char tmpstr[L_tmpnam];
  // Открываем файл
  fp = fopen(tmpnam(tmpstr), "w");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }
  // Записываем в файл строки, которые вводит
  // пользователь, до тех пор, пока он не введет пустую строку
  while(strlen(input) > 1)
  {
    // Читаем строку из стандартного потока ввода
    cout << "Введите строку ";
    fgets(input, 80, stdin);
    // Записываем строку в файл
    fwrite(input, strlen(input), 1, fp);
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

Если во время работы цикла программы вместо того, чтобы ввести пустую строку, нажать комбинацию клавиш <Ctrl>+<C> или завершить работу про-

граммы любым другим нештатным способом, то можно обнаружить, что в файл либо не записана информация, либо записана только часть. Это связано с тем, что операция по записи информации на диск является достаточно ресурсоемкой и де-факто не производится после того, как отрабатывает функция `fwrite()`. Все поступающие на запись данные помещаются в буфер, и после того, как в буфере скопится большой объем информации, производится запись на жесткий диск. Это позволяет значительно ускорить работу программ, т. к. им не нужно дожидаться медленной операции записи. Вся информация, помещенная в буфер, принудительно записывается на жесткий диск во время закрытия файла при помощи функции `fclose()` или при штатном завершении программы. Такое поведение не всегда удобно, особенно если пользователь будет прибегать к закрытию программы при помощи комбинации клавиш <Ctrl>+<C>. Поэтому в стандартную библиотеку введена функция `fflush()`, которая сбрасывает содержимое буфера на диск. Функция `fflush()` имеет следующий синтаксис:

```
int fflush(FILE *fp)
```

Функция принимает в качестве единственного аргумента указатель на структуру `FILE` и возвращает 0 в случае успеха и значение константы `EOF` в случае неудачи.

В листинге II.3.48 представлен новый вариант программы записи текста в файл, который является устойчивым для нештатного завершения программы.

Листинг II.3.48. Использование функции `fflush()`

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

int main()
{
  FILE *fp;
  char input[80] = "init";
  char tmpstr[L_tmpnam];
  // Открываем файл
  fp = fopen(tmpnam(tmpstr), "w");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }
```

```
  // Записываем в файл строки, которые вводит
  // пользователь, до тех пор, пока он не введет пустую строку
  while(strlen(input) > 1)
  {
    cout << "Введите строку ";
    fgets(input, 80, stdin);
    // Записываем строку в файл
    fwrite(input, strlen(input), 1, fp);
    // Сбрасываем содержимое буфера на
    // жесткий диск
    fflush(fp);
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

II.3.19. Количество строк в файле

Для подсчета строк в файле необходимо открыть файл для чтения и в цикле читать строки из файла до тех пор, пока не будет достигнуто окончание файла. Для чтения строки из файла удобно воспользоваться функцией `fgets()`, которая имеет следующий синтаксис:

```
char *fgets(char *str, int length, FILE *fp)
```

Функция читает в строку *str* текущую строку из файла *fp*. Чтение производится до тех пор, пока не будет достигнут символ перевода строки или в строке не будет прочитано *length* символов. В случае успеха функция возвращает указатель на строку *str*, в противном случае функция возвращает `NULL`.

Замечание

Временная строка *str* имеет 10 000 символов, этого обычно более чем достаточно для чтения текстовых файлов, в которых строки гораздо короче.

Решение представлено в листинге II.3.49.

Листинг II.3.49. Подсчет количества строк в файле

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

```
int main()
{
  FILE *fp;
  // Открываем файл
  fp = fopen("filename.txt", "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  const int LENGTH = 10000;
  int count = 0;
  char str[LENGTH];
  // Подсчитываем количество строк в файле
  while(!feof(fp))
  {
    fgets(str, LENGTH, fp);
    count++;
  }
  cout << count << "\n";

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

Проверка, достигнут конец файла или нет, осуществляется при помощи функции `feof()`, которая имеет следующий синтаксис:

```
int feof(FILE *fp)
```

Функция возвращает `true`, если достигнут конец файла, и 0 — в противном случае.

II.3.20. Вывод случайной строки из файла

Для того чтобы выбрать случайную строку из файла, следует предварительно подсчитать количество строк в файле. Для удобства выделим решение из задачи II.3.19 в отдельную функцию (листинг II.3.50).

Листинг II.3.50. Подсчет количества строк в файле

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

```
// Подсчет количества строк в файле
int number_string(char *filename);

int main()
{
  // Имя файла
  char *filename = "text.txt";
  // Число строк в файле
  const int NUMBER_LINE = number_string(filename);

  cout << NUMBER_LINE << "\n";

  return 0;
}

// Подсчет количества строк в файле
int number_string(char *filename)
{
  FILE *fp;
  // Открываем файл
  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл " << filename << ": "
         << ferror(fp) <<  "\n";
    return -1;
  }

  const int LENGTH = 10000;
  int count = 0;
  char str[LENGTH];
  // Подсчитываем количество строк в файле
  while(!feof(fp))
  {
    fgets(str, LENGTH, fp);
    count++;
  }

  // Закрываем файл
  fclose(fp);

  return count;
}
```

Как видно из листинга II.3.50, константа NUMBER_LINE содержит количество строк в файле text.txt. Количество строк возвращает функция number_string(), которая принимает в качестве единственного аргумента имя файла.

После этого необходимо разбить содержимое файла на отдельные строки. Для удобства не будем создавать двумерный массив для строк. Вместо этого создадим одну строку str, в которой будем хранить все строки файла, логически разбив строку на блоки из расчета, что строка файла не занимает более MAX_NUMBER_CHAR символов. По умолчанию константа MAX_NUMBER_CHAR будет равна 80. В листинге II.3.51 представлена программа, которая разбивает содержимое файла на отдельные строки и помещает их в строку str, после чего строки выводятся в стандартный поток вывода.

Листинг II.3.51. Разбивка содержимого файла на строки

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

// Подсчет количества строк в файле
int number_string(char *filename);
// Функция, разбивающая содержимое файла на строки
// и помещающая каждую строку в качестве элемента массива arr
char *file_lines(char *filename,
                 char *arr,
                 const int COUNT,
                 const int MAX_NUMBER_CHAR = 80);

int main()
{
  // Имя файла
  char *filename = "text.txt";
  // Максимальное число символов в строке
  const int MAX_NUMBER_CHAR = 80;
  // Число строк в файле
  const int NUMBER_LINE = number_string(filename);

  // Выделяем память под массив строк
  char *str = new char[MAX_NUMBER_CHAR*NUMBER_LINE];

  // Получаем строки файла в массив str
  str = file_lines(filename, str, NUMBER_LINE, MAX_NUMBER_CHAR);
```

```
  // Выводим строки файла в стандартный поток
  for(int i = 0; i < NUMBER_LINE; i++)
  {
    cout << &str[i*MAX_NUMBER_CHAR];
  }

  // Освобождаем память
  delete [] str;

  return 0;
}

// Функция, разбивающая содержимое файла на строки
// и помещающая каждую строку в качестве элемента массива arr
char *file_lines(char *filename,
                 char *arr,
                 const int COUNT,
                 const int MAX_NUMBER_CHAR)
{
  if(arr == NULL) return NULL;

  FILE *fp;
  // Открываем файл
  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл " << filename << ": "
         << ferror(fp) <<  "\n";
    return NULL;
  }

  // Читаем строки файла и размещаем их в строке arr
  for(int i = 0; i < COUNT && !feof(fp); i++)
  {
    fgets(&arr[i*MAX_NUMBER_CHAR], MAX_NUMBER_CHAR, fp);
  }

  // Закрываем файл
  fclose(fp);

  return arr;
}
```

Программа из листинга II.3.51, получив количество строк в файле, объявляет строку `str` длинной, равной произведению количества строк в файле

NUMBER_LINE и максимального количества символов в строке MAX_NUMBER_CHAR. Данная строка передается функции file_lines(), которая в случае успеха возвращает указатель на эту строку. Строки файла в str начинаются с каждого 80-го символа (значение константы MAX_NUMBER_CHAR). Благодаря этому содержимое файла можно вывести при помощи конструкции

```
for(int i = 0; i < NUMBER_LINE; i++)
{
  cout << &str[i*MAX_NUMBER_CHAR];
}
```

Следует обратить внимание, что стандартному потоку передается адрес строки, который извлекается при помощи оператора &, без использования этого оператора будет выводиться лишь один первый символ строки.

Теперь для вывода случайной строки остается лишь получить случайное число i, в диапазоне от 0 до NUMBER_LINE – 1, и вывести адрес строки str[i*MAX_NUMBER_CHAR] в стандартный поток (листинг II.3.52).

Листинг II.3.52. Вывод случайной строки из файла

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

// Подсчет количества строк в файле
int number_string(char *filename);
// Функция, разбивающая содержимое файла на строки
// и помещающая каждую строку в качестве элемента массива arr
char *file_lines(char *filename,
                 char *arr,
                 const int COUNT,
                 const int MAX_NUMBER_CHAR = 80);

int main()
{
  // Имя файла
  char *filename = "text.txt";
  // Максимальное число символов в строке
  const int MAX_NUMBER_CHAR = 80;
  // Число строк в файле
  const int NUMBER_LINE = number_string(filename);
```

```
  // Выделяем память под массив строк
  char *str = new char[MAX_NUMBER_CHAR*NUMBER_LINE];

  // Получение строк файла в массиве str
  str = file_lines(filename, str, NUMBER_LINE, MAX_NUMBER_CHAR);

  // Получение случайного номера строки файла
  srand(time(NULL));
  int i = int((float)rand()/RAND_MAX*10);
  // Вывод случайной строки файла
  cout << &str[i*MAX_NUMBER_CHAR];

  // Освобождаем память
  delete [] str;

  return 0;
}
```

II.3.21. Вывод трех случайных строк файла

Для решения данной задачи можно модифицировать решение задачи из *разд. II.3.20*. При выборе случайных элементов массива необходимо сравнивать значения новых индексов с ранее сгенерированными, и, если они совпадают, генерировать значение повторно (листинг II.3.53).

Листинг II.3.53. Вывод трех случайных строк файла

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

// Подсчет количества строк в файле
int number_string(char *filename);
// Функция, разбивающая содержимое файла на строки
// и помещающая каждую строку в качестве элемента массива arr
char *file_lines(char *filename,
                 char *arr,
                 const int COUNT,
                 const int MAX_NUMBER_CHAR = 80);
```

```
int main()
{
  // Имя файла
  char *filename = "TMP10.$$$";
  // Максимальное число символов в строке
  const int MAX_NUMBER_CHAR = 80;
  // Число строк в файле
  const int NUMBER_LINE = number_string(filename);

  // Выделяем память под массив строк
  char *str = new char[MAX_NUMBER_CHAR*NUMBER_LINE];

  // Получить строки файла в массиве str
  str = file_lines(filename, str, NUMBER_LINE, MAX_NUMBER_CHAR);

  srand(time(NULL));
  // Первая случайная строка
  int index1 = int((float)rand()/RAND_MAX*10);
  // Вторая случайная строка
  int index2 = index1;
  while(index1 == index2)
  {
    index2 = int((float)rand()/RAND_MAX*10);
  }
  // Третья случайная строка
  int index3 = index2;
  while(index3 == index2 || index3 == index1)
  {
    index3 = int((float)rand()/RAND_MAX*10);
  }
  // Вывод трех случайных строк файла
  cout << &str[index1*MAX_NUMBER_CHAR];
  cout << &str[index2*MAX_NUMBER_CHAR];
  cout << &str[index3*MAX_NUMBER_CHAR];

  // Освобождаем память
  delete [] str;

  return 0;
}
```

II.3.22. Последние три строки файла

Для решения данной задачи удобно воспользоваться функцией подсчета количества строк в файле `number_string()` из листинга II.3.50. После этого сле-

дует повторно пройтись циклом по файлу и, если номер текущей строки будет равен одной из трех последних строк, вывести строку в стандартный поток (листинг II.3.54).

Листинг II.3.54. Последние три строки файла

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

// Подсчет количества строк в файле
int number_string(char *filename);

int main()
{
  // Имя файла
  char *filename = "text.txt";
  // Число строк в файле
  const int NUMBER_LINE = number_string(filename);
  FILE *fp;

  // Открываем файл
  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл " << filename << ": "
         << ferror(fp) <<  "\n";
    return -1;
  }

  const int LENGTH = 10000;
  char str[LENGTH];

  // Читаем содержимое файла
  for(int i = 0; i < NUMBER_LINE; i++)
  {
    fgets(str, LENGTH, fp);
    if(i > NUMBER_LINE - 4) cout << str;
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

В листинге II.3.54 файл открывается два раза, однако часто возникает потребность перечитать файл, не закрывая его. Основная сложность в том, что после того, как файл прочитан, курсор указывает на конец файла, и для повторного чтения его необходимо установить в начало файла. Для этого применяется функция `rewind()`, которая имеет следующий синтаксис:

```
void rewind(FILE *fp)
```

В листинге II.3.55 представлен вариант решения задачи, при котором файл text.txt открывается один раз. После того как количество строк подсчитано, курсор устанавливается в начало файла, и начинает работу цикл, выводящий последние три строки файла.

Листинг II.3.55. Альтернативный вариант решения задачи

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

int main()
{
  // Имя файла
  char *filename = "TMP10.$$$";

  FILE *fp;

  // Открываем файл
  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл " << filename << ": "
         << ferror(fp) <<  "\n";
    return -1;
  }

  const int LENGTH = 10000;
  int count = 0;
  char str[LENGTH];
  // Подсчитываем количество строк в файле
  while(!feof(fp))
  {
    fgets(str, LENGTH, fp);
    count++;
  }
```

```
  // Устанавливаем курсор в начало файла
  rewind(fp);

  // Читаем содержимое файла
  for(int i = 0; i < count; i++)
  {
    fgets(str, LENGTH, fp);
    if(i > count - 4) cout << str;
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

II.3.23. Поиск строки в файле

Для поиска строки в другой строке удобно использовать функцию `strstr()` из библиотеки `<string>`, которая имеет следующий синтаксис:

```
char *strstr(const char *str, const char *search)
```

Функция ищет первое вхождение подстроки *search* в строку *str* и возвращает указатель на найденную строку в случае успеха или `NULL`, если подстрока *search* не найдена.

В листинге II.3.56 представлена программа, выводящая номера строк файла, в состав которых входит подстрока, введенная пользователем.

Листинг II.3.56. Поиск строки в файле

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

int main()
{
  const int LENGTH_STR = 80;
  char input[LENGTH_STR];
  // Читаем строку из стандартного потока ввода
  cout << "Введите строку ";
  fgets(input, LENGTH_STR, stdin);
  // Удаляем перевод строки
  input[strlen(input) - 1] = '\0';
```

```
  // Имя файла
  char *filename = "text.txt";
  FILE *fp;

  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл " << filename << ": "
         << ferror(fp) <<  "\n";
    return -1;
  }

  const int LENGTH = 10000;
  char str[LENGTH];
  int i = 1;
  // Выводим номера строк, в которых найдена подстрока input
  while(!feof(fp))
  {
    fgets(str, LENGTH, fp);
    if(strstr(str, input) != NULL) cout << i << "\n";
    i++;
  }

  fclose(fp);

  return 0;
}
```

После того как строка `input` получена от пользователя, необходимо удалить из нее перевод строки при помощи конструкции `input[strlen(input)-1]='\0'`, в противном случае будут найдены только те строки, которые заканчиваются на подстроку `input`.

II.3.24. Самая длинная и самая короткая строки в файле

Для того чтобы определить самое длинное и самое короткое слово в Linux-словаре linux.words, можно последовательно прочитать файл циклом. При этом использовать три строки:

- `current` — текущая строка файла;
- `min` — минимальная строка;
- `max` — максимальная строка.

Строки min и max инициализируются первой строкой словаря, если текущая строка current длиннее max, то строка max заменяется содержимым строки current, если строка current короче min, то строка min также заменяется значением current. К концу цикла строки max и min будут содержать самое длинное и самое короткое слово словаря. Для определения количества символов в строке удобно воспользоваться библиотечной функцией strlen(). Однако при выводе количества символов в словаре следует помнить, что подсчет их производится с учетом невидимого символа перевода строки \n, поэтому из результата следует вычесть единицу (листинг II.3.57).

Замечание

В различных операционных системах для перевода строки используются разные последовательности управляющих символов. В UNIX-подобных операционных системах это, как правило, одиночный символ перевода строки \n, в Windows — последовательность \r\n, в MacOS — \n\r. Файл linux.words создан в UNIX-подобной операционной системе, поэтому в нем для перевода строки используется символ \n.

Листинг II.3.57. Поиск самой длинной и самой короткой строк в файле

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

int main()
{
  FILE *fp;
  // Максимальное количество символов
  const int LENGTH = 80;
  // Текущая строка
  char current[LENGTH];
  // Минимальная строка
  char min[LENGTH];
  // Максимальная строка
  char max[LENGTH];

  fp = fopen("linux.words", "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }
  // Помещаем в строки min и max первую строку файла
  fgets(current, LENGTH, fp);
```

```
  strncpy(min, current, LENGTH);
  strncpy(max, current, LENGTH);

  while(!feof(fp))
  {
    fgets(current, LENGTH, fp);
    if(strlen(current) > strlen(max)) strncpy(max, current, LENGTH);
    if(strlen(current) < strlen(min)) strncpy(min, current, LENGTH);
  }
  fclose(fp);

  cout << "Самое короткое слово в словаре - " <<
          "(" << strlen(min) - 1 << " символов) " << min << "\n";
  cout << "Самое длинное слово в словаре  - " <<
          "(" << strlen(max) - 1 << " символов) " << max << "\n";

  return 0;
}
```

Для копирования содержимого строк в листинге II.3.57 используется библиотечная функция `strncpy()`, которая копирует в первый аргумент содержимое второго аргумента. При этом копируется не более определенного третьим аргументом количества символов.

II.3.25. Список слов заданной длины

Для решения данной задачи, так же как и в случае задачи из *разд. II.3.24*, можно просматривать файл словаря в цикле, последовательно сравнивая количество символов в текущем слове со значением, который ввел пользователь. Если библиотечная функция возвращает количество символов (за вычетом единицы, компенсирующей перевод строки), меньшее введенного пользователем числа `num`, то слово следует вывести в стандартный поток вывода, а значение счетчика слов `count_words` увеличить на единицу (листинг II.3.58).

Листинг II.3.58. Вывод списка слов заданной длины

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

int main()
{
  FILE *fp;
  // Максимальное количество символов
  const int LENGTH = 80;
```

```
  // Текущая строка
  char current[LENGTH];
  // Количество символов в строке
  int num;

  cout << "Введите количество символов (1-80) ";
  cin >> num;
  if(number > 80)
  {
    cout << "Недопустимое количество символов\n";
    return 1;
  }

  fp = fopen("linux.words", "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  // Количество слов
  int count_words = 0;
  while(!feof(fp))
  {
    fgets(current, LENGTH, fp);
    if(strlen(current) - 1 <= num)
    {
      cout << current;
      count_words++;
    }
  }
  fclose(fp);

  cout << "Количество найденных слов - " << count_words << "\n";

  return 0;
}
```

II.3.26. Поиск слов по первым символам

Для сравнения текущей строки файла со строкой, введенной пользователем, удобно воспользоваться функцией `strncmp()`, которая имеет следующий синтаксис:

```
int strncmp(const char *str1, const char *str2, size_t count);
```

Функция сравнивает *count* символов строк *str1* и *str2* и возвращает 0, если строки равны, число больше нуля, если строка *str1* больше *str2*, и число меньше нуля, если *str1* меньше *str2*. В простейшем случае решение могло бы выглядеть так, как показано в листинге II.3.59.

Листинг II.3.59. Простейшее решение задачи

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

int main()
{
  FILE *fp;
  // Максимальное количество символов
  const int LENGTH = 80;
  // Текущая строка
  char current[LENGTH];
  // Начальные символы строки
  char prefix[LENGTH];

  cout << "Введите строку ";
  fgets(prefix, LENGTH, stdin);

  fp = fopen("linux.words", "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  // Количество слов
  int count_words = 0;
  while(!feof(fp))
  {
    fgets(current, LENGTH, fp);
    if(!strncmp(current, prefix, strlen(prefix) - 1))
    {
      cout << current;
      count_words++;
    }
  }
  fclose(fp);
```

```
  cout << "Количество найденных слов - " << count_words << "\n";

  return 0;
}
```

Решение из листинга II.3.59 производит лексикографическое сравнение, поэтому сравнение зависит от регистра. Так, если пользователь вводит последовательность "abb", результат программы может выглядеть следующим образом:

```
abbe
abbey
abbeys
abbot
abbots
abbreviate
abbreviated
abbreviates
abbreviating
abbreviation
abbreviations
Количество найденных слов - 11
```

При этом в результирующий список не попадают слова "Abba" и "Abby", которые начинаются с заглавной буквы A. Исправить ситуацию можно, приводя к нижнему или верхнему регистру как слова из словаря, так и введенную пользователем последовательность символов. Для такого преобразования предназначены функции `tolower()` и `toupper()`, которые преобразуют символ в нижний и верхний регистры соответственно. Функции реализованы в стандартной библиотеке `<ctype>` и имеют следующий синтаксис:

```
int tolower(int ch);
int toupper(int ch);
```

Представленные функции работают только с одним символом, поэтому преобразование, которое бы обрабатывало всю строку, необходимо реализовать самостоятельно. В листинг II.3.60 приводится пример функции `strtolower()`, которая принимает указатель на строку и приводит все символы строки к нижнему регистру. Так как функции передается указатель на строку, нет необходимости возвращать результат оператором `return`. Изменения, которые проводятся над строкой в функции `strtolower()`, будут доступны и после того, как функция закончит работу.

Листинг II.3.60. Функция преобразования строки к нижнему регистру

```
void strtolower(char *str)
{
  for(char *ch = str; *ch; ch++)
  {
    *ch = tolower(*ch);
  }
}
```

В листинге II.3.61 показано альтернативное решение, которое не зависит от регистра вводимой последовательности символов и регистра слов в словаре linux.words.

Листинг II.3.61. Поиск слов по первым символам без учета регистра

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

void strtolower(char *str);

int main()
{
  FILE *fp;
  // Максимальное количество символов
  const int LENGTH = 80;
  // Текущая строка
  char current[LENGTH];
  // Начальные символы строки
  char prefix[LENGTH];

  cout << "Введите строку ";
  fgets(prefix, LENGTH, stdin);

  fp = fopen("linux.words", "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  // Количество слов
  int count_words = 0;
```

```
  strtolower(prefix);
  while(!feof(fp))
  {
    fgets(current, LENGTH, fp);
    strtolower(current);
    if(!strncmp(current, prefix, strlen(prefix) - 1))
    {
      cout << current;
      count_words++;
    }
  }
  fclose(fp);

  cout << "Количество найденных слов - " << count_words << "\n";

  return 0;
}
```

На запрос "abb", "Abb" или "ABB" программа из листинга II.3.61 вернет следующий список слов:

```
abba
abbe
abbey
abbeys
abbot
abbots
abbott
abbreviate
abbreviated
abbreviates
abbreviating
abbreviation
abbreviations
abby
Количество найденных слов - 14
```

II.3.27. Изменение порядка следования строк в файле

Для удобства решения задачи создадим функцию `max_line()`, которая принимает имя файла, а возвращает количество символов в самой длинной строке файла (листинг II.3.62).

Листинг II.3.62. Функция `max_line()`

```
// Количество символов в самой длинной строке файла filename
int max_line(char *filename)
{
  FILE *fp;
  // Максимальное количество символов
  const int LENGTH = 80;
  // Текущая строка
  char current[LENGTH];
  // Максимальное количество символов в строке файла
  int count_char;

  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return -1;
  }
  // Помещаем в строки min и max первую строку файла
  fgets(current, LENGTH, fp);
  count_char = strlen(current);

  while(!feof(fp))
  {
    fgets(current, LENGTH, fp);
    if(strlen(current) > count_char) count_char = strlen(current);
  }
  fclose(fp);

  return count_char;
}
```

Функция обрабатывает строки файла в предположении, что ни одна строка не превышает 80 символов.

С учетом функции из листинга II.3.62 программа, меняющая порядок следования строк словаря linux.words, может выглядеть так, как это представлено в листинге II.3.63.

Листинг II.3.63. Изменение порядка следования строк в файле

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

```
int max_line(char *filename);
int number_string(char *filename);
char *file_lines(char *filename,
                 char *arr,
                 const int COUNT,
                 const int MAX_NUMBER_CHAR);

int main()
{
  FILE *fp;

  // Имя файла
  char *filename = "linux.words";
  // Количество символов в самой длинной строке файла filename
  const int MAX_NUMBER_CHAR = max_line(filename);
  // Количество строк в файле
  const int NUMBER_LINE = number_string(filename);
  // Память под строки файла
  char *arr = new char[MAX_NUMBER_CHAR*NUMBER_LINE];

  // Получаем строки файла в массив arr
  arr = file_lines(filename, str, NUMBER_LINE, MAX_NUMBER_CHAR);

  // Открываем файл linux.words на запись
  fp = fopen(filename, "w");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    return 1;
  }

  // Записываем строки в файл в обратном порядке
  for(int i = NUMBER_LINE - 1; i >= 0; i--)
  {
    fputs(&arr[i*MAX_NUMBER_CHAR], fp);
  }

  fclose(fp);

  delete [] arr;

  return 0;
}
```

Функция опирается на ранее разработанные нами функции `number_string()`, которая подсчитывает количество строк в файле (см. листинг II.3.50) и функцию `file_lines()`, которая возвращает область памяти `arr`, разбитую на `MAX_NUMBER_CHAR` символов со строками файла `filename` (см. листинг II.3.51).

После этого лишь остается записать в файл linux.words слова из области `arr` в обратном порядке.

II.3.28. Разбить файл на части

Для формирования имени файла создадим вспомогательную функцию `get_filename()` (листинг II.3.64), принимающую в качестве первого параметра указатель на строку, в которую помещается имя файла, а в качестве второго параметра — номер файла. Функция возвращает имя файла в формате `part.`*xxx*, где *xxx* — число, передаваемое через параметр `number`.

Замечание

Функция `get_filename()` использует глобальную константу `LENGTH`, которая хранит количество символов во вспомогательных строках программы.

Листинг II.3.64. Вспомогательная функция для формирования имени файла

```
void get_filename(char *str, int number)
{
  char tmp[LENGTH];
  strcpy(str, "part.");
  itoa(number,tmp,10);
  strcat(str, tmp);
}
```

Для преобразования числа `number` в строку используется библиотечная функция `itoa()`, которая имеет следующий синтаксис:

```
char *itoa(int num, char *str, int baz)
```

В качестве первого параметра передается число *num*, в качестве второго — указатель на строку *str*, в которой сохраняется результат преобразования, в качестве третьего аргумента передается основание (от 2 до 36). В нашем случае, т. к. требуется десятичное число, в качестве второго параметра всегда передается значение 10.

В листинге II.3.65 приведена программа, разбивающая содержимое текстового файла linux.words на фрагменты, содержащие по 1000 строк.

Листинг II.3.65. Разбиение содержимого файла на фрагменты по 1000 строк

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

// Количество символов во вспомогательных строках
const int LENGTH = 80;

void get_filename(char *str, int number);

int main()
{
  FILE *fp, *pt;
  // Имя файла
  const char *filename = "linux.words";
  // Строк в одном файле
  const int NUMBER_LINES = 1000;
  // Количество строк в файле
  int count = 0;
  // Текущий номер файла
  int current_number = 0;

  char str[LENGTH];
  char name_old[LENGTH] = "";
  char name[LENGTH] = "";

  fp = fopen(filename, "r");

  // Формируем имя файла name вида "part.0"
  get_filename(name, 0);
  pt = fopen(name, "w");

  while(!feof(fp))
  {
    // Текущий номер файла
    current_number = count/NUMBER_LINES;
    // Формируем имя файла name вида "part.0"
    get_filename(name, current_number);
    if(strcmp(name_old,name))
    {
      fclose(pt);
      cout << name << "\n";
```

```
      pt = fopen(name, "w");
    }

    // Читаем строку из файла-источника
    fgets(str, LENGTH, fp);
    // Записываем строку в файл-приемник
    fputs(str, pt);

    strcpy(name_old, name);
    count++;
  }

  // Закрываем файл
  fclose(fp);

  return 0;
}
```

II.3.29. Шаблоны функций

Язык C++ является строго типизированным языком программирования, именно поэтому в нем вводится перегрузка функций и операторов, которые позволяют для каждого из типов данных создать собственную реализацию функции. С учетом того, что помимо базовых типов данных допускается создание собственных типов, перегрузка функций для каждого из используемых типов не всегда удобна. В том случае, когда функции различаются лишь типом принимаемых параметров, удобно использовать шаблоны функций. Шаблон функции объявляется при помощи ключевого слова `template`, за которым в угловых скобках следует объявление обобщенного типа. В листинге II.3.66 в качестве имени производного типа используется значение `X`.

Листинг II.3.66. Шаблон функции `swap()`

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

template <class X> void swap(X &fst, X &snd)
{
  X temp;
  temp = fst;
  fst = snd;
  snd = temp;
}
```

```
int main()
{
  int fst=10, snd=20;

  cout << "fst = " << fst << "\n";
  cout << "snd = " << snd << "\n";
  swap(fst, snd);
  cout << "fst = " << fst << "\n";
  cout << "snd = " << snd << "\n";

  double dfst=10, dsnd=20;

  cout << "dfst = " << dfst << "\n";
  cout << "dsnd = " << dsnd << "\n";
  swap(dfst, dsnd);
  cout << "dfst = " << dfst << "\n";
  cout << "dsnd = " << dsnd << "\n";

  return 0;
}
```

Результат выполнения программы из листинга II.3.66 может выглядеть следующим образом:

```
fst = 10
snd = 20
fst = 20
snd = 10
dfst = 10
dsnd = 20
dfst = 20
dsnd = 10
```

В функции вместо конкретного типа используется обобщенный тип `X`, вместо которого компилятор подставляет требуемый тип в зависимости от типа передаваемых параметров.

Функция `swap()` из листинга II.3.66 страдает одним недостатком: оба параметра должны иметь один и тот же тип. Если функции передать две переменных разного типа, это приведет к ошибке компиляции. Для обхода данного ограничения в шаблоне функции можно использовать несколько обобщенных типов данных. В листинге II.3.67 используются два обобщенных типа — `X` и `Y`.

Листинг II.3.67. Использование двух обобщенных типов

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

template <class X, class Y> void swap(X &fst, Y &snd)
{
  X temp;
  temp = fst;
  fst  = snd;
  snd  = temp;
}

int main()
{
  int fst = 10;
  double snd = 20.0;

  cout << "fst = " << fst << "\n";
  cout << "snd = " << snd << "\n";
  swap(fst, snd);
  cout << "fst = " << fst << "\n";
  cout << "snd = " << snd << "\n";

  return 0;
}
```

II.3.30. Перегрузка шаблона функций

При использовании шаблонов функций допускается их перегрузка. Компилятор сам выбирает, какая из функций требуется для того или иного вызова. В листинге II.3.68 представлено возможное решение задания.

Замечание

Для определения максимального и минимального значения среди двух значений функций используется тернарный оператор *условие* ? *выражение1* : *выражение2*, который возвращает *выражение1*, если *условие* принимает значение `true`, и *выражение2*, если *условие* принимает значение `false`.

Листинг II.3.68. Перегрузка шаблонов функций

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

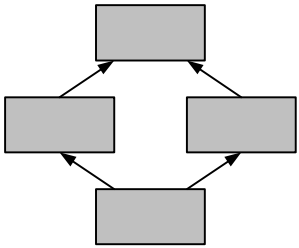
```
template <class X> X mmin(int count, X *arr)
{
  X val = arr[0];
  for(int i = 0; i < count; i++)
  {
    if(val > arr[i]) val = arr[i];
  }
  return val;
}

template <class X> X mmax(int count, X *arr)
{
  X val = arr[0];
  for(int i = 0; i < count; i++)
  {
    if(val < arr[i]) val = arr[i];
  }
  return val;
}

template <class X> X mmin(X fst, X snd)
{
  return fst < snd ? fst : snd;
}

template <class X> X mmax(X fst, X snd)
{
  return fst > snd ? fst : snd;
}

int main()
{
  int arr[] = {4,6,2,3,1,8,9,7};

  int count = sizeof(arr)/sizeof(arr[0]);
  cout << "mmin = " << mmin(count, arr) << "\n";
  cout << "mmax = " << mmax(count, arr) << "\n";
  cout << "(3,5) = " << mmin(3,5) << "\n";
  cout << "(1.5,5.4) = " << mmax(1.5,5.4) << "\n";

  return 0;
}
```

Как видно из листинга II.3.68, получаются два варианта каждой из функций, компилятор самостоятельно выбирает правильный вариант функции.

Замечание

Одним из ограничений шаблонов является тот факт, что их нельзя применять для функций с неопределенным количеством параметров.

ГЛАВА II.4



Объекты и классы

II.4.1. Чем отличается структура *struct* от класса *class*?

В языке C для создания структуры используется ключевое слово `struct`, в C++ дополнительно к ключевому слову `struct` добавляется ключевое слово `class`. Отличие заключается в том, что в `struct` члены по умолчанию являются открытыми (`public`), а в классе `class` закрытыми (`private`). Изменить атрибут членов можно при помощи ключевых слов `private` и `public`. В листинге II.4.1 объявляется структура `str` и класс `cls`, в которых член `prv` является закрытым (доступным только внутри класса), а `pub` — открытым (доступным вне класса).

Замечание

Атрибуты `private` и `public`, а также конструкция `class` (и вообще средства объектно-ориентированного программирования) доступны только в языке C++, в языке C они недоступны.

Листинг II.4.1. Объявление структуры `str` и класса `cls`

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

struct str
{
    int pub;
  private:
    int prv;
};
```

```
class cls
{
    int prv;
  public:
    int pub;
};

int main()
{
  str obj_str;
  cls obj_cls;

  obj_str.pub = 3;
  // obj_str.prv = 3; // Ошибка
  obj_cls.pub = 3;
  // obj_cls.prv = 3; Ошибка

  return 0;
}
```

Как видно из листинга II.4.1, после того как объекты `obj_str` и `obj_cls` объявлены, посредством точки можно обращаться лишь к открытым членам, обращение к закрытому члену вызовет ошибку компиляции. В структурах `struct` необходимо явно помечать закрытые члены, а в классах `class` явно помечать открытые члены. То же касается и методов: объявляя метод в классе или структуре, его можно помечать как открытый, так и закрытый (листинг II.4.2).

Замечание

Объявление и использование методов в структуре `struct` допустимо только в языке C++ и недоступно в языке C.

Листинг II.4.2. Закрытые и открытые методы

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

struct str
{
    int pub;
    int sum()
    {
      hello();
```

```
      return pub + prv;
    }
  private:
    int prv;
    int hello()
    {
      cout << "Hello, world\n";
    }
};

int main()
{
  str obj_str;

  obj_str.pub = 3;
  cout << obj_str.sum() << "\n";
  // obj_str.hello(); // Ошибка

  return 0;
}
```

Как видно из листинга II.4.2, в структуре `str` объявляются два метода: открытый метод `sum()` и закрытый метод `hello()`. Обращение к закрытым членам и методам доступно только из методов внутри класса и недоступно для использования в качестве членов и методов объекта.

Точно такой же механизм используется при наследовании от класса `class` и структуры `struct`. По умолчанию если не указывается спецификатор наследования (`public`, `private` или `protected`), для базового класса `class` выбирается спецификатор `private`, а для базовой структуры `struct` выбирается спецификатор `public` *(см. разд. II.4.22)*.

II.4.2. Чем отличается объединение *union* от класса *class*?

Объединение `union` *(см. разд. II.2.11)* позволяет выделить под одну переменную одну и ту же область памяти, в классе `class` под каждую переменную выделяются разные участки памяти. Точно так же, как и при сравнении `struct` и `class`, следует отметить, что члены в `union` по умолчанию являются открытыми, а в `class` — закрытыми. Кроме этого, от `union`, в отличие от `class`, нельзя наследовать другие классы. Во всем остальном `union` и `class` идентичны. В языке C++ конструкция `union` позволяет использовать методы точно так же, как и конструкция `class` (листинг II.4.3).

Листинг II.4.3. Использование методов в конструкции `union`

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

union addresses
{
  private:
    int var;
    unsigned short int arr[2];
  public:
    addresses(int value);
    unsigned short int first();
    unsigned short int second();
};

// Конструктор
addresses::addresses(int value)
{
  var = value;
}
// Получить младший адрес
unsigned short int addresses::first()
{
  return arr[0];
}
// Получить старший адрес
unsigned short int addresses::second()
{
  return arr[1];
}

int main()
{
  addresses adr(2000000);

  cout << "first  = " << adr.first() << "\n";
  cout << "second = " << adr.second() << "\n";

  return 0;
}
```

Методы класса необязательно объявлять в самом классе, их можно вынести за его пределы, указав, к какому классу принадлежит функция, при помощи

оператора разрешения области видимости ::. Как видно из листинга II.4.3, конструктор объединения addresses получает через аргумент целое значение типа int, которое делит область памяти с массивом arr, состоящим из двух элементов unsigned short int. Первый и второй элементы массива arr можно получить через методы first() и second() объединения. Следует отметить, что члены объединения являются закрытыми, поэтому доступно только преобразование целого числа int в две переменные unsigned short int, обратное преобразование при помощи объединения addresses уже невозможно.

II.4.3. Константы в классах

Трудность объявления константы заключается в том, что ее значение не может изменяться на протяжении срока жизни объекта. Инициировать константу непосредственно в классе нельзя, т. к. разные объекты должны иметь различные значения констант. Константы, как правило, применяются для определения количества элементов массивов, вычисления значений других членов объекта. Поэтому константа должна инициализироваться в момент создания объекта до выполнения конструктора. В связи с этим константы инициализируются в особой точке конструктора — списке инициализирующих значений (листинг II.4.4).

Замечание

Изначально список инициализирующих значений предназначался для инициализации конструкторов базового типа, однако в более поздних версиях C++ он был расширен для использования инициализации констант.

Листинг II.4.4. Использование констант в классе

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

class cls
{
  public:
    const int COUNT;
    // Конструктор
    cls::cls(int val) : COUNT(val){}
};

int main()
{
  cls fst(100), snd(200);
```

```
  cout << "fst.COUNT = " << fst.COUNT << "\n";
  cout << "snd.COUNT = " << snd.COUNT << "\n";

  return 0;
}
```

Как видно из листинга II.4.4, список инициализирующих значений объявляется после конструктора через двоеточие перед телом конструктора. Это позволяет инициализировать константы до того, как будет вызван конструктор, и при выполнении операторов конструктора константы класса являются гарантированно инициализированными. Результат выполнения программы из листинга II.4.4 выглядит следующим образом:

```
fst.COUNT = 100
snd.COUNT = 200
```

Если потребуется объявить две или более константы в классе, то их инициализирующие значения следует перечислить через запятую (листинг II.4.5).

Листинг II.4.5. Объявление двух констант в классе

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

class cls
{
  public:
    const int COUNT;
    const int TOTAL;
    // Конструктор
    cls::cls(int val, int tot) : COUNT(val), TOTAL(tot){}
};

int main()
{
  cls fst(100, 1000), snd(200, 1001);

  cout << "const = " << fst.COUNT << " " << fst.TOTAL << "\n";
  cout << "const = " << snd.COUNT << " " << snd.TOTAL << "\n";

  return 0;
}
```

Результат работы программы из листинга II.4.5 выглядит следующим образом:

```
const = 100 1000
const = 200 1001
```

Как и в случае обычных функций, в конструкторе допускается использование необязательных параметров. В листинге II.4.6 как раз приводится такой пример (в случае отсутствия параметров конструктора член `COUNT` принимает значение 100, а член `TOTAL` — значение 500).

Замечание

Перегрузка конструктора в данном случае не является выходом, т. к. инициализация членов `COUNT` и `TOTAL` производится с использованием инициализирующих значений.

Листинг II.4.6. Использование необязательных параметров конструктора

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

class cls
{
  public:
    const int COUNT;
    const int TOTAL;
    // Конструктор
    cls::cls(int val = 100, int tot = 500) : COUNT(val), TOTAL(tot){}
};

int main()
{
  cls fst(100, 1000), snd;

  cout << "const = " << fst.COUNT << " " << fst.TOTAL << "\n";
  cout << "const = " << snd.COUNT << " " << snd.TOTAL << "\n";

  return 0;
}
```

Результат работы программы из листинга II.4.6 выглядит следующим образом:

```
const = 100 1000
const = 100 500
```

Специальным случаем констант являются статические константы, объявленные при помощи ключевого слова `static`. Такие константы являются общими

для всех объектов класса, поэтому они не могут иметь различные значения для разных объектов, и порядок их инициализации несколько отличается от обычных констант. Во-первых, допускается инициализация статических констант непосредственно при объявлении в классе (это единственный случай, когда допускается инициализация члена непосредственно при объявлении). Во-вторых, статическую константу можно инициализировать, используя оператор разрешения области видимости :: в глобальной области видимости (листинг II.4.7).

Листинг II.4.7. Использование статических констант в качестве членов класса

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

class cls
{
  public:
    // Прямая инициализация
    static const int COUNT = 100;
    static const int TOTAL;
};
// Инициализация с использованием оператора
// разрешения области видимости
const int cls::TOTAL = 1000;

int main()
{
  cls fst, snd;

  cout << "const = " << fst.COUNT << " " << fst.TOTAL << "\n";
  cout << "const = " << snd.COUNT << " " << snd.TOTAL << "\n";

  return 0;
}
```

Как видно из листинга II.4.7, член `COUNT` инициализируется непосредственно в классе `cls`, в то время как член `TOTAL` инициализируется в глобальной области видимости. Результат работы программы из листинга II.4.7 выглядит следующим образом:

```
const = 100 1000
const = 100 1000
```

Так как константы являются статическими, то обращение к ним через оператор разрешения области видимости :: происходит без объявления объекта класса (листинг II.4.8).

Листинг II.4.8. Обращение к статическим членам, без объявления объекта класса

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

class cls
{
  public:
    // Прямая инициализация
    static const int COUNT = 100;
    static const int TOTAL;
};
// Инициализация с использованием оператора
// разрешения области видимости
const int cls::TOTAL = 1000;

int main()
{
  cout << "const = " << cls::COUNT << " " << cls::TOTAL << "\n";

  return 0;
}
```

II.4.4. Подсчет количества созданных объектов

Для реализации класса, подсчитывающего количество созданных объектов, удобно воспользоваться статическим членом класса, который объявляется при помощи ключевого слова `static`. Для статических членов всегда выделяется одна область памяти, независимо от того, сколько объектов класса будет создано в программе. Статический член точно так же, как и обычные члены класса, может быть объявлен открытым (`public`), закрытым (`private`) или защищенным (`protected`).

При объявлении членов класса их инициализация в конструкции `class` не допускается, именно для такой инициализации в C++ введен конструктор, который выполняется перед использованием всех остальных методов и осуществляет инициализацию объекта. Так как статическая переменная одна на все объекты класса, инициализация в конструкторе недопустима. Если статический член является константой, то его можно объявить непосредственно в

классе, однако для подсчета количества объектов необходима переменная. Инициализация статических переменных производится при помощи обращения к ним через оператор разрешения области видимости ::. В листинге II.4.9 представлен класс, реализующий подсчет количества созданных объектов класса `element`.

Листинг II.4.9. Количество созданных объектов

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

class element
{
  public:
    // Конструктор
    element()
    {
      count++;
      element::var = 100;
    }
    // Деструктор
    ~element()
    {
      count--;
    }
    // Интерфейсная функция к члену count
    int get_count()
    {
       return count;
    }
  private:
    static int count;
    const int var;
};

int element::count = 0;

int main()
{
  element fst, snd;

  cout << "count  = " << fst.get_count() << "\n";

  return 0;
}
```

Как видно из листинга II.4.9, для инициализации статического элемента `count` используется конструкция

```
int element::count = 0;
```

которая объявляется в глобальной области видимости. В конструкторе класса `element` значение переменной `count` увеличивается на единицу, а в деструкторе уменьшается на единицу. Поэтому в каждый момент времени, обратившись к методу `get_count()`, можно выяснить количество объектов `element` в программе.

В листинге II.4.10 демонстрируется создание объекта `in` внутри функции `funct()` и уничтожение его при выходе за ее приделы.

Замечание

Для уменьшения объема кода определение класса `element` в листингах II.4.10 и II.4.11 не приводится.

Листинг II.4.10. Создание объекта `element` внутри функции `funct()`

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

// Прототип функции
void funct(void);

int element::count = 0;

int main()
{
  element fst, snd;

  cout << "count = " << fst.get_count() << "\n";
  funct();
  cout << "count = " << fst.get_count() << "\n";

  return 0;
}

void funct(void)
{
  element in;
  cout << "count = " << in.get_count() << "\n";
}
```

Результат работы программы из листинга II.4.6 выглядит следующим образом

```
count = 2
count = 3
count = 2
```

Объект `in` существует только в области видимости функции `funct()`, поэтому, как только функция прекращает свою работу, он разрушается, вызывая деструктор и уменьшая значение статического члена `element::count` на единицу. Деструктор также срабатывает при вызове оператора `delete`, освобождающего динамически выделенную под объект класса `element` память (листинг II.4.11).

Листинг II.4.11. Выделение динамической памяти под объект

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

int element::count = 0;

int main()
{
  element fst, snd;

  cout << "count = " << fst.get_count() << "\n";
  element *prt = new element;
  cout << "count = "  << fst.get_count() << "\n";
  delete prt;
  cout << "count = "  << fst.get_count() << "\n";

  return 0;
}
```

Программа из листинга II.4.11 выведет в стандартный поток следующие строки:

```
count = 2
count = 3
count = 2
```

II.4.5. Найдите ошибку

Ошибка в листинге I.4.1 заключается в том, что конструктор имеет тип возвращаемого значения `void`. Это недопустимо, конструкторы не возвращают значения, и любое указание типа, даже `void`, является ошибкой.

II.4.6. Использование объекта в нескольких файлах

Для того чтобы один и тот же объект был доступен в разных файлах, его как обычную переменную объявляют с атрибутом `extern`. Объект можно объявлять несколько раз с атрибутом `extern`, но без данного атрибута объявление должно встречаться только один раз. Заголовочный файл myclass.h включается во все файлы, где используется объект класса `number`, поэтому в нем разумно объявить объект `num` с атрибутом `extern` (листинг II.4.12). Включая файл myclass.h в другие файлы, мы будем автоматически ссылаться на единственный объект `num`. При этом в файле myclass.cpp можно объявить объект `num` класса `number` без атрибута `extern` (листинг II.4.13). Файл myclass.cpp никуда не включается, и конфликта в программе не будет.

Листинг II.4.12. Файл myclass.h

```
#ifndef MyclassH
#define MyclassH
class number
{
  public:
    number(int value);
    int get_number();
  private:
    int var;
};
extern number num;
#endif
```

Листинг II.4.13. Файл myclass.cpp

```
#include "myclass.h"
number num(12);
number::number(int value)
{
  var = value;
}
int number::get_number()
{
  return var;
}
```

Как видно из листинга II.4.12, при объявлении объекта `num` в файле myclass.h его инициализация не требуется, в то время как при создании объекта в файле myclass.cpp потребуется его инициализация целым значением. Так как конструктора без параметров в классе `number` нет, объявить объект `num` без инициализации не получится.

Для того чтобы использовать объект `number` в любом другом файле, достаточно включить в него заголовочный файл myclass.h (листинг II.4.14).

Листинг II.4.14. Использование объекта `num` в стороннем файле

```
#include <iostream>
#include "myclass.h";
using namespace std;

int main()
{
  cout << num.get_number() << "\n";

  return 0;
}
```

II.4.7. Инициализация объекта при помощи =

Инициализацию при помощи оператора "равно" можно использовать в том случае, если конструктор принимает только один параметр. Поэтому объявление класса `cls` может выглядеть так, как это представлено в листинге II.4.15.

Замечание

Объявление класса традиционно помещают в файл с расширением h, например, cls.h, а реализацию его методов — в файл с расширением cpp, например, cls.cpp.

Листинг II.4.15. Объявление класса `cls`

```
class cls
{
  public:
    cls(int num);
    int number;
};
```

Так как класс `cls` объявляет член `number` открытым, реализация методов класса будет содержать лишь конструктор `cls()` (листинг II.4.16), поскольку не требуется никаких других методов для того, чтобы код из листинга I.4.4 заработал.

Листинг II.4.16. Реализация методов класса `cls`

```
cls::cls(int num)
{
  number = num;
}
```

Для того чтобы запретить инициализацию объекта при помощи оператора =, указывается ключевое слово `explicit`. В листинге II.4.17 демонстрируется пример использования данного ключевого слова.

Листинг II.4.17. Использование ключевого слова `explicit`

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

class cls
{
  public:
    explicit cls(int num);
    int number;
};

cls::cls(int num)
{
  number = num;
}

int main()
{
  // cls obj = 12; // Ошибка
  cls obj(12);
  cout << obj.number << "\n";

  return 0;
}
```

Как видно из листинга II.4.17, при использовании ключевого слова `explicit` допускается инициализация объекта лишь при помощи круглых скобок.

II.4.8. Класс с динамическим массивом

Так как количество элементов в массиве заранее не известно, в классе `myarr` лучше объявить закрытые указатель на целый тип `int *arr` и константу `COUNT_ELEMENT`, которая будет инициализироваться при вызове конструктора *(см. разд. II.4.3)*. Класс `myarr` будет содержать пять методов:

- `myarr()` — конструктор класса, выделяющий память под массив `arr[]` объемом `COUNT_ELEMENT` элементов;
- `~myarr()` — деструктор класса, освобождающий динамически выделенную память;
- `set()` — метод, устанавливающий значение для отдельного элемента массива `arr[]`;
- `get()` — метод, возвращающий значение элемента массива `arr[]`;
- `count()` — метод, возвращающий количество элементов в массиве `arr[]`.

В листинге II.4.18 представлено объявление класса `myarr` (которое можно поместить в заголовочный файл myarr.h).

Замечание

Метод `set()` возвращает ссылку на элемент массива, что позволяет использовать его в формате *имя_объекта*`.set(`*индекс_массива*`) = `*значение_элемента*.

Листинг II.4.18. Объявление класса `myarr`

```
class myarr
{
  public:
    myarr(int count = 100);
    ~myarr();
    int &set(int index);
    int get(int index);
    int count();
  private:
    const int COUNT_ELEMENT;
    int *arr;
};
```

Конструктор принимает необязательный параметр `count` — количество элементов. Если его значение не передается, по умолчанию параметр принимает значение 100. Методы `set()` и `get()` принимают в качестве аргумента `index`

индекс массива, допустимые значения которого лежат в интервале от 0 до `count` – 1.

В листинге II.4.19 приведена реализация класса `myarr` (данный код традиционно размещается в файле myarr.cpp).

Листинг II.4.19. Реализация класса `myarr`

```
// Конструктор
myarr::myarr(int count) : COUNT_ELEMENT(count)
{
  arr = new int[COUNT_ELEMENT];
}
// Деструктор
myarr::~myarr()
{
  delete [] arr;
}
// Установка элемента массива
int &myarr::set(int index)
{
  if(index >= 0 && index < COUNT_ELEMENT) return arr[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}
// Получение элемента массива
int myarr::get(int index)
{
  if(index >= 0 && index < COUNT_ELEMENT) return arr[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
    return -1;
  }
}
// Метод возвращает количество элементов в массиве
int myarr::count()
{
  return COUNT_ELEMENT;
}
```

В конструкторе `myarr()` константный член класса `COUNT_ELEMENT` инициируется параметром `count` конструктора. Кроме того, под указатель `arr` при по-

мощи оператора new выделяется память объемом COUNT_ELEMENT целочисленных значений. Данная память освобождается в деструкторе ~myarr(). Методы set() и get() проверяют, не выходит ли индекс массива, переданный им в качестве параметра, за границы интервала от 0 до COUNT_ELEMENT – 1, и возвращают ссылку и значение элемента соответственно.

В листинге II.4.20 приводится пример использования объекта obj класса myarr. Объект obj содержит 10 элементов, которые заполняются квадратами чисел от 0 до 9 и затем выводятся в стандартный поток.

Листинг II.4.20. Использование объекта obj класса myarr

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

int main()
{
  // Выделяем память под объект, содержащий
  // массив из 10 элементов
  myarr *obj = new myarr(10);
  // Инициируем элементы массива квадратом счетчика цикла
  for(int i = 0; i < obj->count(); i++)
  {
    obj->set(i) = i*i;
  }
  // Выводим элементы массива в стандартный поток
  for(int i = obj->count() — 1; i >= 0; i--)
  {
    cout << obj->get(i) << "\n";
  }
  delete obj;

  return 0;
}
```

Как видно из листинга II.4.20, память под объект obj выделяется динамически, а сама программа оперирует указателем, поэтому для доступа к методам класса вместо точки используется последовательность ->.

II.4.9. Класс-интерфейс к файлу

В листинге II.4.21 представлен класс fileclass, реализующий интерфейс к содержимому файла.

Листинг II.4.21. Класс-интерфейс к файлу

```
class fileclass
{
  public:
    fileclass(char *filename = "linux.words");
    ~fileclass();
    void print();
    void print(char *prefix);
    void print(int min, int max);
    void print(int count);

  private:
    // Файловый дескриптор
    FILE *fp;
    // Содержимое файла
    char *content;
    // Количество строк в файле
    int max_lines;
    // Максимальное количество символов строки
    int max_chars;
    // Количество символов во вспомогательных строках
    static const int LENGTH = 80;
    // Получение количества строк в файле
    int count_lines();
    // Получение максимальной длины строк файла
    int count_chars();
    // Чтение содержимого файла во внутренний массив content
    void read_lines();
    // Вспомогательная функция, приводящая строку к
    // нижнему регистру
    void strtolower(char *str);
};
```

Как видно из листинга II.4.21, в открытой части класса находятся только конструктор, деструктор и перегруженный метод `print()`, подробное описание которого приводится в табл. I.4.1.

В закрытой части класса содержатся файловый дескриптор `fp`, указатель `content` для хранения строк файла, переменная `max_lines`, содержащая количество строк в файле, переменная `max_chars`, содержащая максимальное число символов в строке, которые могут встречаться в файле, и статическая константа `LENGTH`, определяющая размер вспомогательных строк.

Закрытые методы предназначены главным образом для инициализации закрытых членов:

- `count_lines()` — получение количества строк в файле;
- `count_chars()` — получение максимальной длины строк файла;
- `pread_lines()` — заполнение строки `content` строками файла;
- `strtolower()` — вспомогательный метод для преобразования строки к нижнему регистру.

Зачастую возникает соблазн поместить в открытой части и закрытые методы, однако его следует подавлять. Особенно в случае, если методы зависят от порядка вызова друг друга. Чем меньше методов содержит интерфейс класса, тем проще с ним работать и тем меньше вероятность совершить ошибку. Если класс будет содержать дублирующие друг друга или просто лишние методы, у программистов, использующих объекты данного класса, будет постоянный соблазн вызывать их, в результате чего будут появляться несколько альтернативных решений одной и той же задачи. Опыт показывает, что рано или поздно альтернативные методы начинают давать разные результаты, а программы с такой реализацией демонстрируют самые замысловатые ошибки.

Замечание

При разработке класса можно перечислить прототипы методов класса без их реализации — такой класс компилируется без сбоев, а затем постепенно добавлять реализацию методов.

Реализацию класса в данном случае следует начать с конструктора и деструктора, которые должны содержать код, открывающий и закрывающий файл соответственно (листинг II.4.22).

Листинг II.4.22. Реализация конструктора и деструктора

```
// Конструктор
fileclass::fileclass(char *filename)
{
  // Открываем файл
  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    exit(1);
  }
}
```

```
// Деструктор
fileclass::~fileclass()
{
  fclose(fp);
}
```

Как видно из листинга II.4.21, конструктор класса `fileclass` имеет необязательный параметр `filename`. Если он не указывается, параметр принимает значение `linux.words`. Указывать значение по умолчанию следует лишь один раз: либо в прототипе конструктора в конструкции `class`, либо в реализации конструктора `fileclass::fileclass`. Поэтому в листинге II.4.22 значение по умолчанию уже не приводится.

Для удобства и увеличения скорости работы можно прочитать содержимое файла в переменную и работать с ней. Для этого нам необходимо определить количество строк в файле `max_lines`, максимальное количество символов в одной строке `max_chars` и выделить память для указателя `content` объемом `max_lines*max_chars`, после чего разместить в этой области памяти содержимое файла словаря. Для этого потребуется реализация закрытых методов класса (листинг II.4.23).

Листинг II.4.23. Реализация методов `count_lines()`, `count_chars()` и `read_lines()`

```
// Подсчет количества строк в файле
int fileclass::count_lines()
{
  int count = 0;
  char str[LENGTH];
  // Подсчитываем количество строк в файле
  while(!feof(fp))
  {
    fgets(str, LENGTH, fp);
    count++;
  }
  // Устанавливаем файловый курсор на начало файла
  rewind(fp);

  return count;
}
// Количество символов в самой длинной строке файла
int fileclass::count_chars()
{
  // Текущая строка
  char current[LENGTH];
```

```
  // Максимальное количество символов в строке файла
  int count_char;

  // Помещаем в строки min и max первую строку файла
  fgets(current, LENGTH, fp);
  count_char = strlen(current);

  while(!feof(fp))
  {
    fgets(current, LENGTH, fp);
    if(strlen(current) > count_char) count_char = strlen(current);
  }
  // Устанавливаем файловый курсор на начало файла
  rewind(fp);

  return count_char;
}
// Чтение содержимого файла во внутренний массив content
void fileclass::read_lines()
{
  // Читаем строки файла и размещаем их в строке contents
  for(int i = 0; i < max_lines && !feof(fp); i++)
  {
    fgets(&content[i*max_chars], max_chars, fp);
  }
  // Устанавливаем файловый курсор на начало файла
  rewind(fp);
}
```

Так как файл постоянно открыт, то открывать и закрывать файл в каждом из методов нет надобности, требуется только устанавливать файловый курсор в начало файла, чтобы следующий метод мог начать работу с начала файла. Теперь в конструкторе можно выделить память под указатель `content` и разместить в этом объеме памяти строки файла, в деструкторе следует позаботиться об освобождении ранее выделенной памяти (листинг II.4.24).

Листинг II.4.24. Размещение содержимого файла в переменной `content`

```
// Конструктор
fileclass::fileclass(char *filename)
{
  // Открываем файл
  fp = fopen(filename, "r");
```

```
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    exit(1);
  }
  // Устанавливаем количество строк в файле
  max_lines = count_lines();
  // Устанавливаем максимальное количество
  // символов строки
  max_chars = count_chars();
  // Выделяем память под содержимое файла
  content = new char[max_lines*max_chars];
  // Заполняем выделенную память содержимым файла
  read_lines();
}
// Деструктор
fileclass::~fileclass()
{
  delete [] content;
  fclose(fp);
}
```

Как видно из листинга II.4.24, при такой реализации конструктора и деструктора файл linux.words остается открытым на протяжении всего времени существования файла. Если открытый файл linux.words не требуется другим методам класса, то функцию `fclose()`, закрывающую файл, можно переместить из деструктора в конец конструктора. В этом случае файл будет освобождаться моментально, как только содержимое файла будет прочитано в оперативную память.

Теперь, когда содержимое файла размещается в области памяти, адресуемой указателем `content`, можно приступить к реализации методов вывода содержимого файла в стандартный поток (листинг II.4.25).

Листинг II.4.25. Реализация метода `print()`

```
// Вывод содержимого файла
void fileclass::print()
{
  // Вывод всего содержимого файла
  for(int i = 0; i < max_lines; i++)
  {
    cout << &content[i*max_chars];
  }
}
```

```
// Вспомогательная функция, приводящая строку к
// нижнему регистру
void fileclass::strtolower(char *str)
{
  for(char *ch = str; *ch; ch++)
  {
    *ch = tolower(*ch);
  }
}
// Вывод слов, начинающихся с префикса prefix
void fileclass::print(char *prefix)
{
  // Количество слов
  strtolower(prefix);
  // Вывод всего содержимого файла
  for(int i = 0; i < max_lines; i++)
  {
    strtolower(&content[i*max_chars]);
    if(!strncmp(&content[i*max_chars], prefix, strlen(prefix)))
    {
      cout << &content[i*max_chars];
    }
  }
}
// Вывод слов, число символов в которых лежит в
// диапазоне (min, max)
void fileclass::print(int min, int max)
{
  // Проверяем параметры
  if(min < 0 || max < 0 || min > max) return;
  // Вывод всего содержимого файла
  int num = 0;
  for(int i = 0; i < max_lines; i++)
  {
    num = strlen(&content[i*max_chars]) - 1;
    if(num >= min && num <= max)
    {
      cout << &content[i*max_chars];
    }
  }
}
// Выводим слова, содержащие не больше count символов
void fileclass::print(int count)
```

```
{
  // Проверяем параметр
  if(count < 0) return;
  // Вывод всего содержимого файла
  int num = 0;
  for(int i = 0; i < max_lines; i++)
  {
    num = strlen(&content[i*max_chars]) - 1;
    if(num <= max)
    {
      cout << &content[i*max_chars];
    }
  }
}
```

Перегруженный метод `print()` обращается к области памяти `content` и выводит результат в зависимости от условий, которые задаются в том или ином перегруженном варианте метода.

В листинге II.4.26 приводится пример использования объекта `dict` класса `fileclass`, выводится список всех слов, начинающихся на префикс "Abb", а также слов, состоящих из трех символов.

Листинг II.4.26. Использование класса `fileclass`

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

int main()
{
  fileclass dic;
  dic.print("Abb");
  dic.print(3);

  return 0;
}
```

II.4.10. Постраничная навигация

Объекты, как и переменные базовых типов, могут быть объединены в массивы. Объект `dict`, моделирующий разбитый на страницы словарь, будет содержать массив объектов `part`, каждый из которых в свою очередь будет содержать по 10 слов из словаря.

В листинге II.4.27 представлено определение класса `dict`. Этот класс помимо конструктора и деструктора имеет метод `get_count()`, позволяющий определить количество объектов `part`, которое содержит объект `dict`, а также метод `print()`, позволяющий вывести по индексу десяток слов.

Листинг II.4.27. Определение класса `dict`

```
class dict
{
  public:
    dict(char *filename = "linux.words");
    ~dict();
    void print(int index);
    int get_count();
  private:
    // Массив объектов part
    part *dic_array;
    // Количество элементов в массиве dic_array
    int count_obj;
};
```

В конструкторе класса `dict` необходимо обеспечить подсчет количества требуемых объектов `part` и выделить под них память, деструктор должен освобождать выделенную под массив объектов память (листинг II.4.28).

Листинг II.4.28. Конструктор и деструктор класса `dict`

```
dict::dict(char *filename)
{
  FILE *fp;

  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    exit(1);
  }

  char str[part::NUMBER_CHARS];
  // Подсчитываем количество строк в файле
  int count = 0;
  while(!feof(fp))
```

```
  {
    fgets(str, part::NUMBER_CHARS, fp);
    count++;
  }

  rewind(fp);

  // Подсчитываем количество объектов part, которые
  // необходимо зарезервировать под файл
  count_obj = count/part::NUMBER_STRINGS + 1;
  if(!(count % part::NUMBER_STRINGS)) count_obj--;

  // Выделяем память под массив объектов dic
  dic_array = new part[count_obj];

  // Заполняем объекты массива
  count = 0;
  while(!feof(fp))
  {
    fgets(dic_array[count/part::NUMBER_STRINGS].
            str[count%part::NUMBER_STRINGS],
          part::NUMBER_CHARS,
          fp);
    count++;
  }

  fclose(fp);
}
dict::~dict()
{
  delete [] dic_array;
}
```

В отличие от задачи II.4.9 файл словаря не остается открытым на протяжении всего времени существования объекта, а закрывается в конструкторе. При заполнении элементов массива `dic_array[]` используются статические константы `part::NUMBER_STRINGS` и `part::NUMBER_CHARS` класса `part`. Так как элементы класса `part` объявлены с атрибутом `public`, для доступа к массиву `str` не требуются интерфейсные методы, и его можно заполнять непосредственно.

Для получения количества элементов в массиве `dic_array[]` определим метод `get_count()`, а для вывода содержимого отдельного элемента массива — метод `print()` (листинг II.4.29).

Листинг II.4.29. Реализация методов `print()` и `get_count()`

```
void dict::print(int index)
{
  if(index >= 0 && index < count_obj)
  {
    dic_array[index].print();
  }
}
int dict::get_count()
{
  return count_obj;
}
```

Для вывода содержимого каждой "страницы" из 10 слов словаря используется метод `print()` объектов `part`, образующих массив `dic_array`.

Теперь, когда класс `dict` построен, достаточно объявить объект, чтобы реализовать программу, выводящую 10 слов словаря по индексу, который вводит пользователь (листинг II.4.30).

Листинг II.4.30. Использование класса `dict`

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

int main()
{
  dict dic("linux.words");

  int number;
  cout << "Введите номер 10-ки (0-" << (dic.get_count() — 1) << ") ";
  cin >> number;

  dic.print(number);

  return 0;
}
```

II.4.11. Алфавитная навигация

В листинге II.4.31 представлено определение класса `symbol`, конструктор которого читает файл словаря и размещает в оперативной памяти слова, начинающиеся с определенного символа `symb`, передаваемого в качестве первого параметра конструктора.

Листинг II.4.31. Определение класса `symbol`

```
class symbol
{
  public:
    symbol(char symb, char *filename = "linux.words");
    ~symbol();
    void print();
    char get_symbol();
  private:
    static const int NUMBER_CHARS = 80;
    int count;
    char symbol_char;
    char *content;
};
```

Помимо конструктора и деструктора, открытая часть класса `symbol` содержит метод `print()` для вывода содержимого объекта и метод `get_symbol()`, возвращающий символ, с которого начинаются слова, хранящиеся в объекте.

В листинге II.4.32 представлены реализации конструктора и деструктора класса `symbol`. В отличие от предыдущих задач, ведется подсчет не всех строк файла, а только тех, которые начинаются с символа `symb`. Именно этот символ и задается первым параметром конструктора. После этого при помощи оператора `new` под строки выделяется память, и файл словаря перечитывается повторно для заполнения выделенного объема памяти строками.

Листинг II.4.32. Реализация конструктора и деструктора класса `symbol`

```
// Конструктор
symbol::symbol(char symb, char *filename)
{
  FILE *fp;

  // Инициируем закрытый член symbol_char
  symbol_char = symb;

  fp = fopen(filename, "r");
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    exit(1);
  }
```

```
  char str[NUMBER_CHARS];
  // Подсчитываем количество строк в файле,
  // которые отводятся под слова, начинающиеся с
  // символа symb
  count = 0;
  while(!feof(fp))
  {
    fgets(str, NUMBER_CHARS, fp);
    if(tolower(str[0]) == tolower(symb)) count++;
  }

  rewind(fp);

  // Выделяем память под содержимое файла
  content = new char[count*NUMBER_CHARS];

  // Заполняем объекты массива
  int index = 0;
  while(!feof(fp))
  {
    fgets(str, NUMBER_CHARS, fp);
    if(tolower(str[0]) == tolower(symb))
    {
      strncpy(&content[index*NUMBER_CHARS], str, NUMBER_CHARS);
      index++;
    }
  }

  fclose(fp);
}
// Деструктор
symbol::~symbol()
{
  delete [] content;
}
```

В листинге II.4.32 при сравнении первого символа строки `str[0]` и переданного через параметр конструктора символа `symb` используется библиотечная функция `tolower()`, которая приводит переданный ей в качестве параметра символ к нижнему регистру.

В листинге II.4.33 представлена реализация метода `print()`, который выводит строки, содержащиеся в объекте класса `symbol`, а также реализация метода

get_symbol(), возвращающего символ, с которого начинается каждое слово, хранящееся в объекте.

Листинг II.4.33. Реализация методов print() и get_symbol()

```
// Возвращаем символ, с которого начинаются
// слова, хранящиеся в объекте
char symbol::get_symbol()
{
  return symbol_char;
}
// Вывод содержимого объекта
void symbol::print()
{
  for(int i = 0; i < count; i++)
  {
    cout << &content[i*NUMBER_CHARS];
  }
}
```

Программа в листинге II.4.34 выведет символ 'h', а также все слова словаря linux.words, которые начинаются с символа 'h'.

Замечание

Здесь и далее в листингах, демонстрирующих работу класса, не приводится полная реализация класса, которая предшествует функции main(). Перед функцией main() следует либо поместить всю предыдущую реализацию класса, рассмотренную в листингах II.4.31—II.4.33, либо выделить класс в отдельные файл и включить его при помощи директивы препроцессора #include.

Листинг II.4.34. Пример использования класса symbol

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

int main()
{
  symbol char_a('h');
  cout << char_a.get_symbol() << "\n";
  char_a.print();

  return 0;
}
```

Для того чтобы объявить массив объектов `symbol` по количеству символов в английском алфавите, достаточно воспользоваться конструкцией, представленной в листинге II.4.35.

Листинг II.4.35. Массив объектов `symbol`

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

int main()
{
  symbol dic[26] =
  {'a','b','c','d','e','f','g','h','i',
   'j','k','l','m','n','o','p','q','r',
   's','t','u','v','w','x','y','z'};

  return 0;
}
```

Инициализация массива объектов `symbol` как массива, состоящего из элементов базового типа, возможна благодаря тому, что конструктор позволяет осуществлять передачу одного параметра. На самом деле инициализация в листинге II.4.35 является сокращенной записью программы из листинга II.4.36.

Замечание

Для сокращения объема листинга II.4.36 приводится лишь часть инициализирующих конструкторов.

Листинг II.4.36. Полная инициализация массива объектов `symbol`

```
int main()
{
  symbol dic[26] =
  {
    symbol('a'),
    symbol('b'),
    ...
    symbol('z')
  }

  return 0;
}
```

Полная форма применяется в том случае, когда инициализация объекта требует передачи более одного параметра. Например, в листинге II.4.37 при инициализации элементов массива `dic` посредством второго необязательного параметра конструктора изменяется имя файла словаря, используемого для инициализации объектов класса `symbol`.

Листинг II.4.37. Использование более одного параметра в конструкторе

```
int main()
{
  symbol dic[26] =
  {
    symbol('a', "text.txt"),
    symbol('b', "text.txt"),
    ...
    symbol('z', "text.txt")
  }

  return 0;
}
```

Однако воспользоваться инициализацией, представленной в листингах II.4.35 и II.4.36, не удастся, т. к. инициализация в самом классе не допускается, а при динамическом выделении памяти под массив объектов `symbol` будет невозможна инициализация каждого из объектов. Поэтому для реализации класса `dict` удобно воспользоваться массивом указателей на объекты `symbol`, под которые можно выделить динамически память в цикле. Определение класса `dict` представлено в листинге II.4.38.

Листинг II.4.38. Определение класса `dict`

```
class dict
{
  public:
    // Количество символов в английском алфавите
    static const int NUMBER_SYMBOLS_EN = 26;
    dict();
    ~dict();
    void print(char symb);
  private:
    // Массив указателей объектов symbol
    symbol *dic_array[NUMBER_SYMBOLS_EN];
};
```

Как видно из листинга II.4.38, в классе объявляется статическая константа NUMBER_SYMBOLS_EN, принимающая значение 26, по числу символов в английском алфавите, а также массив указателей dic_array, под каждый из которых в конструкторе выделяется, а в деструкторе освобождается память (листинг II.4.39).

Листинг II.4.39. Реализация конструктора и деструктора класса dict

```
dict::dict()
{
  for(char ch = 'a', i = 0; ch <= 'z'; ch++, i++)
  {
    dic_array[i] = new symbol(ch, "linux");
  }
}
dict::~dict()
{
  for(int i = 0; i < NUMBER_SYMBOLS_EN; i++)
  {
    delete dic_array[i];
  }
}
```

В листинге II.4.40 демонстрируется возможная реализация метода print(), в котором производится поиск объекта, содержащего слова, начинающиеся с символа symb и вызывающего метод print() объекта symbol.

Листинг II.4.40. Реализация метода print() класса dict

```
void dict::print(char symb)
{
  for(char ch = 'a', i = 0; ch <= 'z'; ch++, i++)
  {
    if(tolower(symb) == ch) dic_array[i]->print();
  }
}
```

Следует обратить внимание на то, что метод print() элемента массива указателей dic_array[] вызывается при помощи последовательности ->, а не точки, т. к. обращение происходит посредством указателя, а не обычного объекта.

В листинг II.4.41 демонстрируется пример использования объекта класса dict, в частности выводятся все слова из словаря, начинающиеся на символ 'h'.

Листинг II.4.41. Использование объекта класса `dict`

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

int main()
{
  dict dic;
  dic.print('h');

  return 0;
}
```

II.4.12. Дружественная функция

Дружественные функции объявляются в `public`-разделе класса при помощи ключевого слова `friend` (листинг II.4.42). Важно отметить, что сама функция при этом не является методом класса, она просто получает возможность обращаться к закрытым членам и методам класса.

Листинг II.4.42. Объявление дружественной функции `print()` в классе `arr`

```
class arr
{
  public:
    arr(int num)
    {
      count = num;
      array = new int[count];
      for(int i = 0; i < count; i++) array[i] = i*i;
    }
    ~arr()
    {
      delete [] array;
    }
    // Дружественная функция
    friend void print(arr obj);
  private:
    int count;
    int *array;
};
```

Дружественные функции принимают в качестве аргумента объект класса, по отношению к которому они являются дружественными. Однако внутри дру-

жественных функций можно обращаться не только к открытым, но также и закрытым членам и методам класса. Реализация функции `print()` может выглядеть так, как это представлено в листинге II.4.43.

Листинг II.4.43. Реализация функции `print()`

```
void print(arr obj)
{
  for(int i = 0; i < obj.count; i++) cout << obj.array[i] << "\n";
}
```

Следует обратить внимание, что дружественные функции не являются методами класса, поэтому их не следует предварять областью видимости `arr::`.

В листинге II.4.44 приведен пример использования дружественной функции.

Листинг II.4.44. Использование дружественной функции `pirnt()`

```
int main()
{
  arr object(10);
  print(object);

  return 0;
}
```

Результат работы программы из листинга II.4.44 может выглядеть следующим образом:

```
0
1
4
9
16
25
36
49
64
81
```

II.4.13. Блокировка файла по статическому члену класса

В листинге II.4.45 показана реализация класса `put_file`. В методе `add()` перед открытием файла при помощи цикла `while` проверяется состояние статиче-

ской переменной open_file: если она равна true (файл в данный момент открыт), цикл продолжает работу. Как только переменная open_file принимает значение false, цикл прекращает работу, а метод add() тут же присваивает переменной open_file значение true, чтобы другие объекты ожидали, когда файл text.txt освободится для записи. После того, как метод add() записывает строку в конец файла text.txt, он присваивает переменной open_file значение false, чтобы другие объекты могли продолжить работу с файлом.

Замечание

В реальных программах, где важно обеспечить блокировку файла, лучше использовать системные вызовы, которые не допускают сбоев и более надежны. Если в библиотеке компилятора доступны функции sleep() и delay(), обеспечивающие задержку выполнения программы, следует их применить в цикле while(), чтобы уменьшить вероятность того, что два экземпляра класса put_file одновременно изменят значение переменной open_file.

Листинг II.4.45. Блокировка файла по статическому члену класса

```
class put_file
{
  public:
    void add(char *str)
    {
      FILE *fp;

      while(open_file) {}
      open_file = true;

      fp = fopen("text.txt", "a");
      if(fp == NULL)
      {
        open_file = false;
        cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
        return;
      }

      fwrite(str, strlen(str), 1, fp);

      fclose(fp);
      open_file = false;
    }
    void print()
    {
      FILE *fp;
```

```
      fp = fopen("text.txt", "r");

      char line[80];
      while(!feof(fp))
      {
        fgets(line, 80, fp);
        cout << line;
      }

      fclose(fp);
    }
  private:
    static bool open_file;
};

bool put_file::open_file = false;
```

Так как переменная `open_file` является статичной, ее инициализация производится за пределами класса. В листинге II.4.46 приводится пример использования объектов класса `put_file`.

Листинг II.4.46. Использование объектов класса `put_file`

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

int main()
{
  put_file fst, snd;
  fst.add("Hello, world!\n");
  snd.add("second line\n");
  snd.print();

  return 0;
}
```

II.4.14. Блокировка файла двумя классами

Дружественная функция может быть дружественной сразу для нескольких классов, в этом случае функции следует передать объекты (или указатели) на объекты обоих классов. В листинге II.4.47 представлена реализация решения с использованием дружественной функции.

Замечание

Для того чтобы можно было указывать в классе `fst` объявления с применением еще нереализованного класса `snd`, обычно используется опережающее объявление для класса `snd`: `class snd;`.

Листинг II.4.47. Использование дружественной функции для блокировки файла

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

// Опережающее объявление класса snd
class snd;
class fst
{
  public:
    void add(char *str, snd *second)
    {
      FILE *fp;

      while(!is_free_file(this, second)) {}
      open_file = true;

      fp = fopen("text.txt", "a");
      if(fp == NULL)
      {
        open_file = false;
        cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
        return;
      }

      fwrite(str, strlen(str), 1, fp);

      fclose(fp);
      open_file = false;
    }
    friend bool is_free_file(fst *first, snd *second);
  private:
    static bool open_file;
};
class snd
{
  public:
    void add(char *str, fst *first)
```

```
    {
      FILE *fp;

      while(!is_free_file(first, this)) {}
      open_file = true;

      fp = fopen("text.txt", "a");
      if(fp == NULL)
      {
        open_file = false;
        cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
        return;
      }

      fwrite(str, strlen(str), 1, fp);

      fclose(fp);
      open_file = false;
    }
    friend bool is_free_file(fst *first, snd *second);
  private:
    static bool open_file;
};

bool snd::open_file = false;
bool fst::open_file = false;

bool is_free_file(fst *first, snd *second)
{
  return !(first->open_file || second->open_file);
}

int main()
{
  fst first;
  snd second;
  first.add("Hello, world!\n", &second);
  second.add("second line\n", &first);

  return 0;
}
```

Как видно из листинга II.4.47, дружественной функции `is_free_file()` передаются два указателя: на класс `fst` и класс `snd` соответственно. Поэтому ме-

тод `add()` каждого из классов принимает указатель на объект другого класса. При использовании дружественной функции внутри класса получить ссылку на текущий объект можно посредством специального ключевого слова `this`.

Решение, представленное в листинге II.4.47, подходит лишь в том случае, если программа использует только по одному объекту каждого из классов `fst` и `snd`. Если объектов много, решение не годится. В последнем случае в качестве дружественных функций используются методы другого класса. В листинге II.4.48 представлено решение задачи, в котором в качестве дружественной функции выступает метод `is_free_file()` класса `fst`.

Листинг II.4.48. Использование метода одного класса в качестве дружественной функции другого класса

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

class snd;
class fst
{
  public:
    void add(char *str, snd *second)
    {
      FILE *fp;

      while(!is_free_file(second)) {}
      open_file = true;

      fp = fopen("text.txt", "a");
      if(fp == NULL)
      {
        open_file = false;
        cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
        return;
      }

      fwrite(str, strlen(str), 1, fp);

      fclose(fp);
      open_file = false;
    }
    bool is_free_file(snd *second);
  private:
    static bool open_file;
};
```

```
class snd
{
  public:
    void add(char *str, fst *first)
    {
      FILE *fp;

      while(!first->is_free_file(this)) {}
      open_file = true;

      fp = fopen("text.txt", "a");
      if(fp == NULL)
      {
        open_file = false;
        cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
        return;
      }

      fwrite(str, strlen(str), 1, fp);

      fclose(fp);
      open_file = false;
    }
    // Дружественный метод класса fst
    friend bool fst::is_free_file(snd *second);
  private:
    static bool open_file;
};

bool fst::is_free_file(snd *second)
{
  return !(this->open_file || second->open_file);
}

bool snd::open_file = false;
bool fst::open_file = false;

int main()
{
  fst first;
  snd second;
  first.add("Hello, world!\n", &second);
  second.add("second line\n", &first);

  return 0;
}
```

Нельзя взаимообъявить методы двух классов дружественными, т. к. объявление одного класса всегда будет предварять объявление второго класса. В связи с чем, для первого класса будут недоступны методы второго класса. Поэтому в листинге II.4.48 в обоих классах для проверки блокировки файла text.txt используется лишь одна функция `is_free_file()` класса `fst`. При использовании ее в классе `snd` в качестве параметра следует передавать указатель на текущий объект `this`. Так как переменные `open_file` в обоих класса объявлены статическими, их всего две в программе — по одной на объекты класса `fst` и объекты класса `snd`. Благодаря чему при вызове метода `add()` можно передавать ссылку на любой объект противоположенного класса.

II.4.15. Копирующий конструктор

Если в реализации класса не указывается конструктор, компилятор генерирует его самостоятельно. Можно явно определить конструктор, но ничего страшного не произойдет, если в классе конструктор будет отсутствовать. Существует еще одна ситуация, когда компилятор может генерировать конструктор самостоятельно, — это инициализация одного объекта другим. В этом случае сгенерированный компилятором конструктор создает побитовое копирование объекта. Такое поведение является неприемлемым, если конструктор выделяет динамическую память, например, как в случае класса `arr` (см. листинг I.4.7). При побитовом копировании указатель `array` обоих объектов будет указывать на одну и ту же область памяти, а деструктор два раза удалит одну и ту же динамическую память.

Замечание

Удаление ранее удаленной памяти может привести к непредсказуемым последствиям, компилятор не отслеживает таких ситуаций.

Для решения подобных проблем в язык C++ введен *копирующий конструктор*, который определяется как конструктор, принимающий в качестве единственного параметра константную ссылку на объект текущего класса:

имя_класса(const *имя_класса* &obj)

В листинге II.4.49 демонстрируется реализация копирующего конструктора для класса `arr`.

Листинг II.4.49. Копирующий конструктор для класса `arr`

```
class arr
{
  public:
    arr(int number);
```

```
    // Копирующий конструктор
    arr(const arr &obj);
    ~arr();
    int get_count();
    int get_arr(int index);
  private:
    int count;
    int *array;
};
// Копирующий конструктор
arr::arr(const arr &obj)
{
  cout << "Вызов копирующего конструктора" << "\n";
  count = obj.get_count();
  array = new int[obj.get_count()];
  for(int i = 0; i < obj.get_count(); i++)
  {
    array[i] = obj.get_arr(i);
  }
}
```

Как видно из листинга II.4.49, вместо побитовой копии в копирующем конструкторе выделяется динамическая память под указатель `array` и происходит инициализация членов текущего объекта значениями из объекта `obj`, который передается копирующему конструктору в качестве параметра. При реализации собственной версии копирующего конструктора важно не забыть ни один член класса, иначе он останется неинициализированным. То есть если полю `count` не присвоить значение, оно получит случайное значение: это может быть и 0, и 4 463 216, и любое другое число.

В реализации копирующего конструктора в листинге II.4.49 добавлен код, выводящий при вызове копирующего конструктора в стандартный поток фразу "Вызов копирующего конструктора", которая позволит лучше понять, когда конструктор вызывается, а когда нет. В листинге II.4.50 приводится пример типичного использования копирующего конструктора — один объект класса `arr` инициализируется другим объектом.

Листинг II.4.50. Использование копирующего конструктора

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

int main()
{
```

```
  arr obj(10);
  arr fst = obj;

  return 0;
}
```

В листинге II.4.51 копирующий конструктор уже не используется, здесь происходит не *инициализация* одного объекта другим, а *присваивание*, а для управления этой операцией необходимо перегружать оператор =.

Кроме этого, копирующий конструктор незаменим, когда объект возвращается из функции. Дело в том, что любой объект, который не предается по указателю или ссылке, а объявляется в функции, при выходе за область видимости функции уничтожается. Именно поэтому объекты не рекомендуется возвращать из функции. Однако обойти это ограничение можно при помощи копирующего конструктора, который создаст копию объекта при возврате его из функции оператором `return` (листинг II.4.51).

Листинг II.4.51. Использование копирующего конструктора при возврате из функции

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

arr get_object(int number)
{
  arr obj(number);
  return obj;
}

int main()
{
  arr fst = get_object(15);

  return 0;
}
```

Однако программа из листинга II.4.52 помимо копирующего конструктора уже потребует перегрузки оператора =.

Листинг II.4.52. Некорректное использование копирующего конструктора

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

```
int main()
{
  arr fst(10)
  fst = get_object(15);

  return 0;
}
```

II.4.16. Перегрузка оператора =

Классы представляют собой новые типы данных, которые по умолчанию не поддерживают операторы базовых типов (=, +, -, *, /, ++, --, >, <, >=, <=, |, &, <<, >>). Однако все операторы можно перегрузить для классов, благодаря чему они более органично вписываются в программную среду. Для перегрузки оператора используется специальный метод с именем `operator`, после которого следует символ перегружаемого оператора, например, `operator=()`. В листинге II.4.53 демонстрируется определение класса `arr`, в котором реализуется копирующий конструктор и перегружается оператор =.

Листинг II.4.53. Перегрузка оператора =

```
class arr
{
  public:
    arr(int number);
    // Копирующий конструктор
    arr(const arr &obj);
    // Перегрузка оператора =
    arr operator=(arr obj);
    ~arr();
    int get_count();
    int get_arr(int index);
  private:
    int count;
    int *array;
};
// Перегрузка оператора =
arr arr::operator=(arr obj)
{
  cout << "Вызов оператора =\n";
  // Если размеры массивов не равны, уничтожаем старый
  // массив и выделяем память под новый
  if(count != obj.get_count())
```

```
  {
    delete [] array;
    array = new int[obj.get_count()];
  }
  count = obj.get_count();
  for(int i = 0; i < obj.get_count(); i++)
  {
    array[i] = obj.get_arr(i);
  }
  return *this;
}
// Копирующий конструктор
arr::arr(const arr &obj)
{
  cout << "Вызов копирующего конструктора\n";
  count = obj.get_count();
  array = new int[obj.get_count()];
  for(int i = 0; i < obj.get_count(); i++)
  {
    array[i] = obj.get_arr(i);
  }
}
```

Как видно из листинга II.4.53, методу `operator=()` передается правый аргумент (`obj`) оператора =, если количество элементов в массивах текущего объекта (левого аргумента оператора =) и объекта `obj` не совпадают, то для текущего объекта выделяется новый участок памяти.

В листинге II.4.54 приводится пример использования оператора = и копирующего конструктора.

Листинг II.4.54. Использование оператора = и копирующего конструктора

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

int main()
{
  arr fst(15);
  arr snd(8);

  snd = fst;

  return 0;
}
```

Программа из листинга II.4.54 выводит в стандартный поток следующие строки:

```
Вызов копирующего конструктора
Вызов оператора =
Вызов копирующего конструктора
```

Первый вызов копирующего конструктора осуществляется при копировании объекта `fst` в объект `obj` для передачи его методу `operator=()`. Далее происходит вызов метода `operator=()` для непосредственного осуществления операции копирования. Последний вызов копирующего конструктора связан с тем, что оператор = всегда возвращает результат присвоения, поэтому становится возможным код, приведенный в листинге II.4.55.

Листинг II.4.55. Множественное присвоение

```
int main()
{
  arr fst(15);
  arr snd(8);
  arr thd(11);

  thd = snd = fst;

  return 0;
}
```

Программа из листинга II.4.55 вернет следующие строки:

```
Вызов копирующего конструктора
Вызов оператора =
Вызов копирующего конструктора
Вызов оператора =
Вызов копирующего конструктора
```

Теперь, когда реализованы копирующий конструктор и оператор =, можно корректно выполнить программу из листинга II.4.53, которая вернет следующие строки:

```
Вызов копирующего конструктора
Вызов оператора =
Вызов копирующего конструктора
```

II.4.17. Перегрузка логических операторов

Для перегрузки логических операторов необходимо добавить в класс `arr` прототипы соответствующих функций (листинг II.4.56).

Листинг II.4.56. Перегрузка логических операторов

```
class arr
{
  public:
    arr(int number);
    // Копирующий конструктор
    arr(const arr &obj);
    arr operator=(arr obj);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);

    bool operator==(arr obj);
    bool operator!=(arr obj);
    bool operator>(arr obj);
    bool operator>=(arr obj);
    bool operator<(arr obj);
    bool operator<=(arr obj);
  private:
    int count;
    int *array;
};
```

Каждый из логических операторов принимает в качестве аргумента объект класса `arr` (правый аргумент) и возвращает булево значение. В листинге II.4.57 приводится реализация методов, которые вызываются при обращении к логическим операторам.

Листинг II.4.57. Реализация логических операторов

```
bool arr::operator==(arr obj)
{
 int min_count = count;
 if(min_count > obj.get_count()) min_count = obj.get_count();
 for(int i = 0; i < min_count; i++)
```

```
 {
   if(array[i] != obj.get_arr(i)) return false;
 }

 return true;
}
bool arr::operator!=(arr obj)
{
 int min_count = count;
 if(min_count > obj.get_count()) min_count = obj.get_count();
 for(int i = 0; i < min_count; i++)
 {
   if(array[i] == obj.get_arr(i)) return false;
 }

 return true;
}
bool arr::operator>(arr obj)
{
 int min_count = count;
 if(min_count > obj.get_count()) min_count = obj.get_count();
 for(int i = 0; i < min_count; i++)
 {
   if(array[i] <= obj.get_arr(i)) return false;
 }

 return true;
}
bool arr::operator>=(arr obj)
{
 int min_count = count;
 if(min_count > obj.get_count()) min_count = obj.get_count();
 for(int i = 0; i < min_count; i++)
 {
   if(array[i] < obj.get_arr(i)) return false;
 }

 return true;
}
bool arr::operator<(arr obj)
{
 int min_count = count;
 if(min_count > obj.get_count()) min_count = obj.get_count();
```

```
 for(int i = 0; i < min_count; i++)
 {
   if(array[i] >= obj.get_arr(i)) return false;
 }

 return true;
}
bool arr::operator<=(arr obj)
{
 int min_count = count;
 if(min_count > obj.get_count()) min_count = obj.get_count();
 for(int i = 0; i < min_count; i++)
 {
   if(array[i] > obj.get_arr(i)) return false;
 }

 return true;
}
```

В листинге II.4.58 приводится пример использования оператора ==.

Листинг II.4.58. Использование оператора ==

```
int main()
{
  arr fst(10), snd(12);

  if(fst == snd) cout << "Объекты равны\n";
  return 0;
}
```

II.4.18. Перегрузка операторов +, –, / и *

Существуют две методики перегрузки бинарных операторов. В первом случае операторы реализуются как методы класса, во втором случае как дружественные функции. Для реализации первого метода модифицируем определение класса `arr` путем ввода дополнительных методов для операторов `+`, `-`, `/` и `*` (листинг II.4.59).

Листинг II.4.59. Перегрузка бинарных операторов +, -, / и *

```
class arr
{
  public:
    arr(int number);
```

```
    // Копирующий конструктор
    arr(const arr &obj);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);

    arr operator+(arr obj);
    arr operator-(arr obj);
    arr operator/(arr obj);
    arr operator*(arr obj);
  private:
    int count;
    int *array;
};
```

В листинге II.4.60 приводится реализация методов, обеспечивающих выполнение операторов +, -, / и *.

Листинг II.4.60. Реализация методов для +, -, / и *

```
arr arr::operator+(arr obj)
{
  int min_count = count;
  if(min_count > obj.get_count()) min_count = obj.get_count();
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    tmp.set_arr(i) = array[i] + obj.get_arr(i);
  }
  return tmp;
}
arr arr::operator-(arr obj)
{
  int min_count = count;
  if(min_count > obj.get_count()) min_count = obj.get_count();
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    tmp.set_arr(i) = array[i] - obj.get_arr(i);
  }
  return tmp;
}
```

```
arr arr::operator*(arr obj)
{
  int min_count = count;
  if(min_count > obj.get_count()) min_count = obj.get_count();
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    tmp.set_arr(i) = array[i] * obj.get_arr(i);
  }
  return tmp;
}
arr arr::operator/(arr obj)
{
  int min_count = count;
  if(min_count > obj.get_count()) min_count = obj.get_count();
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    if(obj.get_arr(i)) tmp.set_arr(i) = array[i] / obj.get_arr(i);
  }
  return tmp;
}
```

Следует отметить, что методы, реализующие операторы, возвращают объект `arr`, объявленный внутри функции (который уничтожается при завершении работы метода), поэтому для корректной работы класса в нем необходим копирующий конструктор.

В листинге II.4.61 демонстрируется пример использования перегруженных операторов класса `arr`.

Листинг II.4.61. Демонстрация использования перегруженных операторов

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

int main()
{
  arr fst(10), snd(12);

  arr thd = fst + snd;
  thd = fst - thd;
```

```
  for(int i = 0; i < thd.get_count(); i++)
  {
    cout << thd.get_arr(i) << "\n";
  }

  thd = fst * snd;
  thd = thd / snd;

  for(int i = 0; i < thd.get_count(); i++)
  {
    cout << thd.get_arr(i) << "\n";
  }

  return 0;
}
```

Альтернативным способом реализации бинарных операторов +, –, / и * является использование дружественных функций (листинг II.4.62).

Листинг II.4.62. Использование дружественных функций для перегрузки операторов

```
class arr
{
  public:
    arr(int number);
    arr(const arr &obj);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);

    friend arr operator+(arr left, arr rigth);
    friend arr operator-(arr left, arr rigth);
    friend arr operator/(arr left, arr rigth);
    friend arr operator*(arr left, arr rigth);
  private:
    int count;
    int *array;
};
```

Дружественные функции принимают два параметра, т. к. не являются членами класса. Однако в силу того, что функции имеют статус дружественных, они могут обращаться к закрытым членам класса напрямую (листинг II.4.63).

Замечание

Операторы можно перегружать не только для двух объектов `arr`, но также для ситуаций, когда объект `arr` складывается с целым числом или любым другим базовым или производным типом. Для реализации таких операторов удобнее воспользоваться шаблонами.

Листинг II.4.63. Реализация дружественных функций

```
arr operator+(arr left, arr rigth)
{
  int min_count = left.count;
  if(min_count > rigth.count) min_count = rigth.count;
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    tmp.set_arr(i) = left.array[i] + rigth.array[i];
  }
  return tmp;
}
arr operator-(arr left, arr rigth)
{
  int min_count = left.count;
  if(min_count > rigth.count) min_count = rigth.count;
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    tmp.set_arr(i) = left.array[i] - rigth.array[i];
  }
  return tmp;
}
arr operator*(arr left, arr rigth)
{
  int min_count = left.count;
  if(min_count > rigth.count) min_count = rigth.count;
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    tmp.set_arr(i) = left.array[i] * rigth.array[i];
  }
  return tmp;
}
```

```
arr operator/(arr left, arr rigth)
{
  int min_count = left.count;
  if(min_count > rigth.count) min_count = rigth.count;
  arr tmp(min_count);
  for(int i = 0; i < min_count; i++)
  {
    if(rigth.get_arr(i)) tmp.set_arr(i) = left.array[i] / rigth.array[i];
  }
  return tmp;
}
```

Для реализации унарных операторов можно также применять как методы класса, так и дружественные функции, правда, в отличие от бинарных операторов метод класса вообще не принимает параметров (т. к. нет дополнительного операнда), а дружественная функция принимает лишь один параметр. Реализуем унарные операторы — и + как члены класса `arr` (листинг II.4.64).

Листинг II.4.64. Объявление унарных операторов – и + в классе `arr`

```
class arr
{
  public:
    arr(int number);
    arr(const arr &obj);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);
    arr operator+();
    arr operator-();
  private:
    int count;
    int *array;
};
```

Реализация унарных операторов + и – применительно к классу `arr` представлена в листинге II.4.65.

Замечание

Для применения библиотечной функции `abs()` в программу следует включить заголовочный файл библиотеки `<math>`.

Листинг II.4.65. Реализация унарных операторов + и –

```
arr arr::operator+()
{
  for(int i = 0; i < count; i++) array[i] = abs(array[i]);
  return *this;
}
arr arr::operator-()
{
  for(int i = 0; i < count; i++) array[i] = -array[i];
  return *this;
}
```

Использование операторов + и – демонстрируется в листинге II.4.66.

Листинг II.4.66. Использование унарных операторов + и –

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

int main()
{
  arr fst(5), snd(6);

  arr thd = -fst;

  for(int i = 0; i < thd.get_count(); i++)
  {
    cout << thd.get_arr(i) << "\n";
  }
  cout << "\n";

  thd = +fst;

  for(int i = 0; i < thd.get_count(); i++)
  {
    cout << thd.get_arr(i) << "\n";
  }

  return 0;
}
```

II.4.19. Перегрузка операторов ++ и --

Особенностью операторов ++ и -- является тот факт, что они имеют постфиксную и префиксную формы. Префиксная форма оператора не принимает никаких аргументов, в то время как постфиксная принимает в качестве аргумента целочисленное значение (листинг II.4.67). Целочисленное значение никак не используется внутри метода и служит только для того, чтобы прототип постфиксного оператора отличался от прототипа метода префиксного оператора.

Листинг II.4.67. Перегрузка операторов ++ и --

```
class arr
{
  public:
    arr(int number);
    arr(const arr &obj);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);

    // Префиксная форма операторов
    arr operator++();
    arr operator--();
    // Постфиксная форма операторов
    arr operator++(int);
    arr operator--(int);
  private:
    int count;
    int *array;
};
```

При реализации методов, реализующих операторы инкремента (++) и декремента (--) для класса `arr` следует помнить, что префиксная форма возвращает объект, в котором преобразования совершены. Постфиксная форма возвращает исходный объект, не затронутый оператором (листинг II.4.68).

Замечание

Ничего не мешает изменить логику поведения операторов, добившись, чтобы префиксная форма оператора вела себя как постфиксная и наоборот. Однако от этого стоит воздержаться, т. к. это не внесет в интерфейс класса ничего, кроме путаницы, что в свою очередь повлечет ошибки.

Листинг II.4.68. Реализация постфиксной и префиксной форм операторов ++ и --

```
// Префиксная форма оператора ++
arr arr::operator++()
{
  for(int i = 0; i < count; i++) array[i]++;
  return *this;
}
// Префиксная форма оператора --
arr arr::operator--()
{
  for(int i = 0; i < count; i++) array[i]--;
  return *this;
}
// Постфиксная форма оператора ++
arr arr::operator++(int notused)
{
  arr tmp = *this;
  for(int i = 0; i < count; i++) array[i]++;
  return tmp;
}
// Постфиксная форма оператора --
arr arr::operator--(int notused)
{
  arr tmp = *this;
  for(int i = 0; i < count; i++) array[i]--;
  return tmp;
}
```

Как видно из листинга II.4.68, для того чтобы в постфиксной форме вернуть неизмененный объект, необходимо сохранять его во временном объекте `tmp`, который затем и возвращается методом. Для того чтобы инициализация проходила корректно, в классе должен быть реализован копирующий конструктор.

II.4.20. Перегрузка оператора *[]*

Для того чтобы элемент массива, который возвращает оператор `[]`, можно было использовать в правой части выражения (осуществлять присваивание), необходимо возвращать элемент массива по ссылке (листинг II.4.69).

Листинг II.4.69. Объявление и реализация оператора `[]` в классе `arr`

```
class arr
{
  public:
    arr(int number);
    arr(const arr &obj);
    arr operator=(arr obj);
    ~arr();
    int get_count();
    int get_arr(int index);
    int &set_arr(int index);
    int &operator[](int index);
  private:
    int count;
    int *array;
};
int &arr::operator[](int index)
{
  if(index >= 0 && index < count) return array[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}
```

II.4.21. Перегрузка оператора *()*

Оператор `()` является очень удобным оператором, т. к. в отличие от предыдущих операторов может принимать произвольное количество параметров. В нашем случае параметр будет только один — значение для нового элемента массива `array` (листинг II.4.70).

Листинг II.4.70. Перегрузка оператора `()`

```
class arr
{
  public:
    arr(int number);
    arr(const arr &obj);
    ~arr();
    int get_count();
    int get_arr(int index);
```

```
    int &set_arr(int index);
    void operator()(int value);
  private:
    int count;
    int *array;
};
void arr::operator()(int value)
{
  int *tmp = new int[count + 1];
  for(int i = 0; i < count; i++) tmp[i] = array[i];
  tmp[count] = value;
  count++;
  delete [] array;
  array = tmp;
}
```

Как видно из листинга II.4.70, в методе, реализующем оператор `()`, выделяется новый объем динамической памяти `tmp`, больший текущего массива `array` на единицу. Новый массив `tmp` инициируется значениями из массива `array`. При этом последний элемент массива, который отсутствовал в `array`, получает значение параметра `value`. После этого текущее количество элементов в объекте увеличивается на единицу (`count++`), память, выделенная под массив `array`, уничтожается, а самому указателю `array` присваивается значение `tmp`, в результате чего объект получает новый массив с дополнительным членом.

II.4.22. Наследование одного класса другим

При наследовании одного класса другим можно указать спецификатор `public`, `private` или `protected` (листинг II.4.71—II.4.73).

Замечание

Здесь и далее в листингах не приводится класс `base` из листинга I.4.9.

Листинг II.4.71. Наследование класса со спецификатором `public`

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

class child : public base
{
  public:
    void print()
```

```
    {
      cout << first << "\n";
      // cout << second << "\n"; // Ошибка
      cout << third << "\n";
    }
};

int main()
{
  child obj;
  obj.print();

  cout << obj.first << "\n";
  // cout << obj.second << "\n"; // Ошибка
  // cout << obj.third << "\n";  // Ошибка

  return 0;
}
```

При наследовании нового класса со спецификатором `public` методы и члены базового класса, объявленные со спецификатором `public`, в производном классе также доступны со спецификатором `public`. Закрытые методы базового класса не доступны в том числе и для класса-наследника. Методы и члены, объявленные со спецификатором `protected`, доступны внутри базового и производного классов, но остаются закрытыми для внешнего использования. В листинге II.4.71 недопустимые обращения к закрытым переменным помечены комментарием "Ошибка".

При использовании спецификатора `private`, при наследовании класса `public`-члены и методы базового класса становятся закрытыми методами производного класса, `private`- и `protected`-члены базового класса недоступны в производном классе (листинг II.4.72).

Листинг II.4.72. Наследование класса со спецификатором `private`

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

class child : private base
{
  public:
    void print()
    {
      cout << first << "\n";
```

```
      // cout << second << "\n"; // Ошибка
      cout << third << "\n";
    }
};

int main()
{
  child obj;
  obj.print();

  // cout << obj.first << "\n";  // Ошибка
  // cout << obj.second << "\n"; // Ошибка
  // cout << obj.third << "\n";  // Ошибка

  return 0;
}
```

При наследовании класса со спецификатором `protected` открытые члены и методы базового класса в производном классе становятся защищенными (листинг II.4.73).

Листинг II.4.73. Наследование класса со спецификатором `protected`

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

class child : protected base
{
  public:
    void print()
    {
      cout << first << "\n";
      // cout << second << "\n"; // Ошибка
      cout << third << "\n";
    }
};

int main()
{
  child obj;
  obj.print();
```

```
  // cout << obj.first << "\n";  // Ошибка
  // cout << obj.second << "\n"; // Ошибка
  // cout << obj.third << "\n";  // Ошибка

  return 0;
}
```

Помимо спецификаторов `public`, `private` и `protected` разработчик класса может управлять статусом члена при помощи *объявлений доступа*. При объявлении метода или члена базового класса с префиксом *имя_базового_класса*:: для этого метода или члена в производном классе восстанавливается статус доступа. В листинге II.4.74 при наследовании класса `child`, несмотря на то, что член `first` должен получить статус закрытого члена `private`, он остается открытой (`public`) в производном классе.

Замечание

При использовании объявления доступа тип члена указывать нет необходимости. Методы также объявляются без возвращаемых значений и параметров — только имя метода.

Листинг II.4.74. Использование объявления доступа

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

class child : private base
{
  public:
    // Восстанавливаем статус члена first до public
    base::first;
    void print()
    {
      cout << first << "\n";
      // cout << second << "\n"; // Ошибка
      cout << third << "\n";
    }
};

int main()
{
  child obj;
  obj.print();
```

```
  cout << obj.first << "\n";
  // cout << obj.second << "\n"; // Ошибка
  // cout << obj.third << "\n";  // Ошибка

  return 0;
}
```

Благодаря объявлению `base::first`, член `first` в производном классе `child` получает статус `public`.

II.4.23. Расширение функциональности класса

Основная сложность, которая возникает при наследовании класса `file_arr` от `arr`, заключается в том, что конструктор базового класса `arr` требует передачи ему параметра (количество элементов в массиве). В этом случае используется расширенная форма объявления конструктора, когда имя конструктора базового класса помещается за конструктором производного класса через двоеточие (листинг II.4.75).

Замечание

Если класс наследуется от нескольких классов, конструкторы которых требуют параметров, они перечисляются через запятую. Конструкторы базовых классов всегда вызываются перед конструкторами производных классов. Деструкторы вызываются в обратном порядке — сначала деструкторы производных классов, затем деструкторы базовых классов.

Листинг II.4.75. Расширение функциональности класса `arr`

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

class file_arr : public arr
{
  public:
    file_arr(int number, char *filename);
    ~file_arr();
  private:
    FILE *fp;
};

file_arr::file_arr(int number, char *filename) : arr(number)
{
  fp = fopen(filename, "w");
```

```
  if(fp == NULL)
  {
    cout << "Невозможно открыть файл: " << ferror(fp) <<  "\n";
    exit(1);
  }
  char str[80];
  for(int i = 0; i < get_count(); i++)
  {
    // Преобразуем элемент массива
    // с индексом i в строку str
    itoa(get_arr(i),str,10);
    fputs(str, fp);
    fputs("\n", fp);
  }
}
file_arr::~file_arr()
{
  fclose(fp);
}

int main()
{
  file_arr obj(10, "text.txt");
  return 0;
}
```

Конструктор `file_arr()`, объявленный с двумя параметрами, использует только один параметр `filename`, другой параметр `number` передается конструктору базового класса `arr`. Параметры для базового класса всегда передаются через параметры производного класса. Поэтому конструкторы всех наследников класса `arr` обязательно будут иметь хотя бы один параметр.

II.4.24. Перегрузка метода базового класса

При объявлении в производном классе метода с таким же именем, что и в базовом классе, новый метод замещает собой метод базового класса (листинг II.4.76).

Листинг II.4.76. Перегрузка метода базового класса

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

```
class base
{
  public:
    void print()
    {
      cout << "Вызов метода print() базового класса base\n";
    }
};

class child : base
{
  public:
   void print()
   {
      cout << "Вызов метода print() производного класса child\n";
   }
};

int main()
{
  child obj;
  obj.print();

  return 0;
}
```

Программа из листинга II.4.76 выведет только одну строку:

```
Вызов метода print() производного класса child
```

Для того чтобы вызвать метод базового класса, необходимо прибегнуть к объявлению доступа (листинг II.4.77).

Листинг II.4.77. Вызов метода `print()` базового класса `base`

```
class child : base
{
  public:
   void print()
   {
      // Вызов метода print() базового класса base
      base::print();
      cout << "Вызов метода print() производного класса child\n";
   }
};
```

Для вызова метода `print()` базового класса достаточно указать имя класса `base` через оператор разрешения области видимости `::`. При использовании класса `child` из листинга II.4.77 программа выведет две строки:

```
Вызов метода print() базового класса base
Вызов метода print() производного класса child
```

II.4.25. Виртуальный класс

Виртуальный класс предназначен для предотвращения ситуации, когда при множественном наследовании класс содержит две или более копий базового класса. Рассмотрим программу в листинге II.4.78.

Листинг II.4.78. Множественное наследование

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

class base
{
  public:
    int number;
};

class first : public base {};
class second : public base {};

class last : public first, second {};

int main()
{
  last obj;
  // obj.number = 10; // Ошибка

  return 0;
}
```

Классы `first` и `second` являются наследниками класса `base`. В свою очередь класс `last` наследует от обоих классов `first` и `second`. Однако обращение члену `number` базового класса `base` вызывает неоднозначность. Дело в том, что каждый из промежуточных классов `first` и `second` имеет свою собственную копию базового класса `base` (рис. II.4.1).

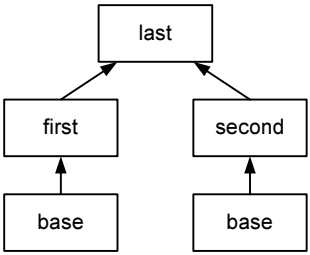


Рис. II.4.1. Каждый из классов `first` и `second` имеет собственную копию базового класса `base`

Поэтому, для того чтобы получить доступ к члену `number`, следует при помощи оператора разрешения области видимости явно указать, какая из копий имеется в виду (листинг II.4.79).

Листинг II.4.79. Правильное обращение к членам `number`

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

int main()
{
  last obj;
  obj.first::number = 10;
  obj.second::number = 10;

  return 0;
}
```

Однако часто требуется лишь одна копия в иерархии наследуемых классов (рис. II.4.2). Достигается это с помощью виртуальных классов.

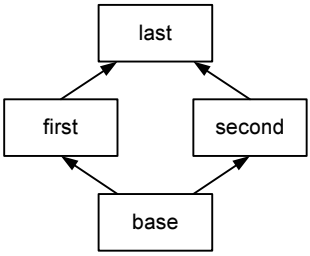


Рис. II.4.2. Единственная копия класса `base` в иерархии наследуемых классов

При наследовании базового класса, как виртуального, его имя предваряется ключевым словом `virtual` (листинг II.4.80).

Листинг II.4.80. Использование виртуальных классов

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

class base
{
  public:
    int number;
};

class first : virtual public base {};
class second : virtual public base {};

class last : public first, second {};

int main()
{
  last obj;
  // Теперь член number — единственный
  obj.number = 10;

  return 0;
}
```

Теперь член `number` является единственным, т. к. метод `base` включается лишь один раз.

II.4.26. Указатель на объект базового типа

Указатель одного типа не может указывать на объект другого типа, однако в случае базового указателя делается исключение. Указатель (или ссылка) на объект базового класса может также указывать на объект производного класса. В листинге II.4.81 объявляется указатель `p` на объект класса `base`.

Листинг II.4.81. Указатель на объект базового класса `base`

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

```
class base
{
  public:
    void print()
    {
      cout << "Вызов метода базового класса base\n";
    }
};

class child : public base
{
  public:
    void print()
    {
      cout << "Вызов метода производного класса child\n";
    }
};

int main()
{
  child obj;
  base *p = &obj;
  // Вызываем метод базового класса
  p->print();
  // Вызываем метод производного класса
  ((child *)p)->print();

  return 0;
}
```

При вызове методов и объектов при помощи указателя `p` доступны лишь методы и объекты базового типа. Если при помощи указателя `p` требуется обратиться к методу производного класса `child`, требуется приведение типа. В листинге II.4.81 выражение приведения типа `(child *)p` заключается в кавычки, т. к. оператор `->` имеет больший приоритет.

Замечание

Механизм виртуальных функций позволяет вызывать методы производных классов без приведения типа.

II.4.27. Чем отличается виртуальная функция от чисто виртуальной функции?

В листинге II.4.81, для того чтобы при помощи указателя на базовый тип вызвать метод производного класса, мы прибегали к приведению типа. Вирту-

альные функции позволяют обойтись без этой потенциально опасной операции. Для методов, которые объявляются в базовом классе при помощи ключевого слова `virtual`, программа сама определяет, какой метод следует вызывать: если в производном классе метод не реализован, вызывается метод базового класса, если реализация имеется, то вызывается метод производного класса (листинг II.4.82).

Замечание

Метод, объявленный с ключевым словом `virtual`, называется *виртуальным методом*, а класс, содержащий виртуальную функцию, называется *полиморфным классом*. Деструктор может быть виртуальным, конструктор класса виртуальным быть не может. Атрибут `virtual` передается по наследству, т. е., какой бы сложной ни была иерархия классов, функция, объявленная виртуальной, остается виртуальной для всех производных классов.

Листинг II.4.82. Объявление виртуальной функции `print()`

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

class base
{
  public:
    virtual print()
    {
      cout << "Вызов метода базового класса base\n";
    }
};
class first : public base
{
  public:
    print()
    {
      cout << "Вызов метода производного класса first\n";
    }
};
class second : public base
{
};

int main()
{
  first fst;
```

```
  base *p = &fst;
  p->print(); // Метод производного класса first

  second snd;
  p = &snd;
  p->print(); // Метод базового класса base

  return 0;
}
```

В листинге II.4.82 первый вызов метода `p->print()` приведет к вызову метода `print()` базового класса `first`, т. к. указатель `p` ссылается на объект класса `first`, реализующего виртуальный метод `print()`. Второй вызов `p->print()` приведет к вызову метода `print()` базового класса. Последнее связано с тем, что указатель `p` ссылается на объект класса `second`, который не перегружает метод `print()`.

Замечание

Виртуальный метод можно вызывать не только через указатель с использованием оператора `->`, но и как метод объекта с помощью оператора "точка".

Виртуальные методы позволяют реализовывать полиморфизм, т. е. ситуацию, когда все производные классы имеют в своем составе заданный метод. Виртуальные методы необходимы для того, чтобы все производные классы имели одинаковый интерфейс. Это позволяет обрабатывать схожим образом объекты разных классов. Каждый класс, унаследованный от `base`, будет иметь в своем составе метод `print()`, но каждый из классов может реализовать его по-своему. Иногда невозможно корректно реализовать виртуальный метод в базовом классе, т. к. недостаточно данных о внутреннем устройстве будущих классов. В этом случае прибегают к чисто виртуальным функциям. *Чисто виртуальной функцией* называют виртуальный метод, который не имеет реализации. Для того чтобы отличить такой метод от прототипа обычной виртуальной функции, в конце функции добавляется последовательность `= 0` (листинг II.4.83). В этом случае компилятор требует, чтобы все производные классы реализовывали такой виртуальный метод, в противном случае программа не компилируется.

Замечание

Класс, который содержит хотя бы одну чисто виртуальную функцию, называется *абстрактным*. У такого класса не может быть объектов, он используется только в качестве базового класса при наследовании.

Листинг II.4.83. Использование чисто виртуальной функции

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

class base
{
  public:
    virtual void print() = 0;
};
class first : public base
{
  public:
    void print()
    {
      cout << "Вызов метода производного класса first\n";
    }
};
class second : public base
{
  public:
    void print()
    {
      cout << "Вызов метода производного класса second\n";
    }
};

int main()
{
  first fst;
  base *p = &fst;
  p->print(); // Вызов метода производного класса first

  second snd;
  p = &snd;
  p->print(); // Вызов метода производного класса second

  return 0;
}
```

Использование чисто виртуальных функций позволяет не только организовать полиморфизм (единый интерфейс для всей иерархии классов), но и потребовать перегрузки метода во всех производных классах. Это гарантирует,

что разработчики не забудут реализовать чисто виртуальные методы в классах-наследниках.

II.4.28. Динамическая идентификация типов

Динамическая идентификация типов (RTTI) позволяет определить тип объекта во время выполнения программы. Как было показано в *разд. II.4.27*, указатель на базовый класс может ссылаться на объекты производного класса. Поэтому при использовании полиморфных классов (классов, содержащих виртуальные функции) часто требуется определить, на объект какого класса ссылается указатель.

Для определения типа объекта применяется оператор `typeid()`, который допускается использовать после включения в программу заголовочного файла `<typeinfo>`. Оператор `typeid()` принимает объект *`object`*, тип которого следует получить, и имеет следующий синтаксис:

```
typeid(object)
```

Оператор возвращает ссылку на объект класса `type_info`, который описывает тип объекта *`object`*. Среди открытых методов класса `type_info` имеется оператор сравнения `==`, оператор неравенства `!=` и метод `name()`, который возвращает указатель на имя класса объекта *`object`*. В листинге II.4.84 приводится пример использования оператора `typeid()`.

Листинг II.4.84. Оператор `typeid()`

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

class base
{
  public:
    void print();
};

int main()
{
  int i;
  double var;
  base obj;
  cout << "i = " << typeid(i).name() << "\n";
```

```
  cout << "var = " << typeid(var).name() << "\n";
  cout << "obj = " << typeid(obj).name() << "\n";

  return 0;
}
```

Как видно из листинга II.4.84, метод `typeid()` может работать как с базовыми типами, так и с объектами классов, созданных программистом. Если оператору `typeid()` передается указатель на полиморфный класс (т. е. класс, который содержит хотя бы одну виртуальную функцию), то он возвращает класс реального объекта, на который указывает указатель (листинг II.4.85).

Замечание

Следует обратить внимание, что оператор `typeid()` принимает объект, поэтому указатель при передаче этому оператору следует разыменовывать при помощи оператора `*`.

Листинг II.4.85. Использование оператора `typeid()` с полиморфными классами

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

class base
{
  public:
    virtual void print()
    {
      cout << "Вызов метода print() базового класса base";
    }
};

class child : public base
{
  public:
    void print()
    {
      cout << "Вызов метода print() производного класса child";
    }
};
```

```
int main()
{
  child chd;
  base *fst = &chd;
  cout << "fst = " << typeid(*fst).name() << "\n";
  base obj;
  base *snd = &obj;
  cout << "snd = " << typeid(*snd).name() << "\n";

  return 0;
}
```

Результат выполнения программы из листинга II.4.85 выглядит следующим образом:

```
fst = child
snd = base
```

Если базовый класс `base` не является полиморфным (т. е. не содержит виртуальных функций), то даже в том случае, когда указатель базового типа ссылается на объект производного типа, оператор `typeid()` возвратит тип базового типа. Например, убрав ключевое слово `virtual` из определения метода `print()` базового класса `base`, можно получить следующий результат выполнения программы:

```
fst = base
snd = base
```

II.4.29. Приведение типов

Приведение типа при помощи круглых скобок (листинг II.4.86) является традиционным способом приведения типа, перешедшим в C++ из языка C.

Листинг II.4.86. Приведение типа при помощи круглых скобок

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

int main()
{
  float var = 4.5;

  cout << (int)var << "\n";

  return 0;
}
```

Такой традиционный способ приведения типов проектировался без учета объектно-ориентированного программирования. Поэтому в язык C++ помимо оператора круглых скобок было введено еще четыре оператора приведения типа: `dynamic_cast`, `const_cast`, `reinterpret_cast`, `static_cast`, которые являются частными случаями оператора круглых скобок.

Оператор `dynamic_cast` предназначен для полиморфного приведения типа, т. е. для приведения указателя или ссылки на базовый класс к указателю или ссылке на производный класс, если такое преобразование допустимо. Если преобразование не допустимо, то оператор возвратит пустой указатель (для ссылок, которые не могут быть нулевыми, генерируется исключение `bad_cast`). Следует отметить, что при использовании круглых скобок преобразование осуществляется в любом случае, приводит ли это к выходу за границы объекта или нет.

В листинге II.4.87 приводится пример преобразования указателя на базовый класс `base` к указателю на производный класс `child` с использованием оператора `dynamic_cast`.

Листинг II.4.87. Использование оператора `dynamic_cast`

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

class base
{
  public:
    virtual void print()
    {
      cout << "Вызов метода print() базового класса base";
    }
};

class child : public base
{
  public:
    void print()
    {
      cout << "Вызов метода print() производного класса child";
    }
};

int main()
{
  child chd;
```

```
  base *obj = &chd;
  child *fst = NULL;

  fst = dynamic_cast<child *>(obj);
  // Если преобразования выполнены успешно,
  // продолжаем работу с указателем fst
  if(fst != NULL)
  {
    fst->print();
  }

  return 0;
}
```

Следует отметить, что проверку на допустимость приведения типа можно выполнить при помощи оператора динамической идентификации типа `typeid()` (листинг II.4.88).

Листинг II.4.88. Альтернативный способ преобразования указателя на базовый класс к указателю на производный класс

```
#include <iostream>
#include <typeinfo>
using namespace std;
 int main()
{
  child chd;
  base *obj = &chd;
  child *fst = NULL;

  // Приведение базового типа к производному
  // без использования оператора dynamic_cast
  if(typeid(*obj) == typeid(child))
  {
    fst = (child *)obj;
    fst->print();
  }

  return 0;
}
```

Оператор `dynamic_cast` является более краткой формой записи конструкции безопасного приведения типа с предварительной проверкой при помощи оператора `typeid()`.

Оператор static_cast выполняет неполиморфное приведение типа. По сути, этот оператор является заменителем традиционных круглых скобок. В листинге II.4.89 оба преобразования типов (из float в int), как с использованием оператора static_cast, так и без него, являются эквивалентными.

Листинг II.4.89. Использование оператора static_cast

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

int main()
{
  float var = 4.5;

  cout << (int)var << "\n";
  cout << static_cast<int>(var) << "\n";

  return 0;
}
```

Оператор const_cast позволяет изменять константный тип данных на неконстантный и наоборот (листинг II.4.90).

Листинг II.4.90. Использование оператора const_cast

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

int main()
{
  const float const_var = 4.5;
  float var = const_cast<float>(const_var);
  cout << var + 1 << "\n";

  return 0;
}
```

Оператор reinterpret_cast позволяет выполнять фундаментальное изменение типа. В листинге II.4.91 указатель str типа char * преобразуется к типу int.

Листинг II.4.91. Использование оператора reinterpret_cast

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

```
int main()
{
  int i;
  char *str = "Hello world";
  i = reinterpret_cast<int>(str);
  // i = static_cast<int>(str); // Ошибка
  cout << i << "\n";

  return 0;
}
```

Если для выполнения такого преобразования будет использован любой другой оператор — `const_cast`, `static_cast` или `dynamic_cast`, — будет возвращена ошибка при компиляции программы.

Традиционный оператор приведения типа — круглые скобки — может применяться вместо любого из перечисленных операторов, код с их использованием более читабелен, а разработка удобна. Тем не менее использование новых операторов приведения типов позволяет избежать целого класса ошибок, связанных с неправильным приведением типов.

II.4.30. Обобщенный класс безопасного массива

Помимо шаблонов функций, допускается создание *шаблонов классов*. Это особенно удобно, когда необходимо реализовывать классы, чьи объекты обрабатывают массивы разных типов. Как и в случае обобщенных функций, определение класса предваряется ключевым словом `template`, в угловых скобках после которого следует объявление обобщенного типа данных, используемого в классе. Если метод реализуется за пределами класса, то его также должно предварять ключевое слово `template`, а угловые скобки с обобщенным типом должны дублироваться после имени метода (листинг II.4.92).

Листинг II.4.92. Шаблон класса безопасного массива

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

template <class X>
class arr
{
  public:
    arr(int number);
    ~arr();
```

```
    int get_count();
    X get_arr(int index);
    X &set_arr(int index);
  private:
    int count;
    X *array;
};
template <class X> arr<X>::arr(int number)
{
  count = number;
  array = new X[number];
  for(int i = 0; i < number; i++) array[i] = i*i;
}
template <class X> arr<X>::~arr()
{
  delete [] array;
}
template <class X> int arr<X>::get_count()
{
  return count;
}
template <class X> X arr<X>::get_arr(int index)
{
  if(index >= 0 && index < count) return array[index];
  else return -1;
}
template <class X> X &arr<X>::set_arr(int index)
{
  if(index >= 0 && index < count) return array[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}

int main()
{
  arr <char> obj(10), obj(100);
  for(int i = 0; i < 10; i++)
  {
    cout << obj.get_arr(i) << "\n";
  }
  return 0;
}
```

При объявлении объекта класса шаблона используемый в объекте тип объявляется в угловых скобках сразу после имени класса. В программе, представленной в листинге II.4.92, в качестве такого типа выступает `char`.

Замечание

Допускается использование и более одного типа данных в шаблонах. В этом случае их перечисляют через запятую в угловых скобках.

II.4.31. Использование параметров в шаблонах классов

В шаблонах можно указывать аргументы, которые не являются типами, т. е. допускается использование обычных параметров. В листинге II.4.94 в качестве второго параметра шаблона передается количество элементов в массиве, который представляет класс `arr`.

Листинг II.4.93. Использование параметров в шаблонах классов

```
template <class X, int size>
class arr
{
  public:
    arr();
    ~arr();
    int get_count();
    X get_arr(int index);
    X &set_arr(int index);
  private:
    X *array;
};
template <class X, int size> arr<X, size>::arr()
{
  array = new X[size];
  for(int i = 0; i < size; i++) array[i] = i*i;
}
template <class X, int size> arr<X, size>::~arr()
{
  delete [] array;
}
template <class X, int size> int arr<X, size>::get_count()
{
  return size;
}
```

```
template <class X, int size> X arr<X, size>::get_arr(int index)
{
  if(index >= 0 && index < size) return array[index];
  else return -1;
}
template <class X, int size> X &arr<X, size>::set_arr(int index)
{
  if(index >= 0 && index < size) return array[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}
```

Более того, параметрам шаблонов можно присваивать значения по умолчанию. В листинге II.4.94 представлена реализация безопасного массива, в котором по умолчанию используется 10 элементов, а в качестве типа массива выступает `int`.

Листинг II.4.94. Использование значений по умолчанию в шаблонах

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

template <class X=int, int size=10>
class arr
{
  public:
    arr();
    ~arr();
    int get_count();
    X get_arr(int index);
    X &set_arr(int index);
  private:
    X *array;
};
template <class X, int size> arr<X, size>::arr()
{
  array = new X[size];
  for(int i = 0; i < size; i++) array[i] = i*i;
}
template <class X, int size> arr<X, size>::~arr()
{
  delete [] array;
}
```

```
template <class X, int size> int arr<X, size>::get_count()
{
  return size;
}
template <class X, int size> X arr<X, size>::get_arr(int index)
{
  if(index >= 0 && index < size) return array[index];
  else return -1;
}
template <class X, int size> X &arr<X, size>::set_arr(int index)
{
  if(index >= 0 && index < size) return array[index];
  else
  {
    cout << "Ошибка: выход за границы массива\n";
  }
}

int main()
{
  arr <char, 15> obj;
  arr <double> obj1;
  arr <> obj2;

  return 0;
}
```

Как видно из листинга II.4.94, даже при использовании значений по умолчанию употребление угловых скобок все равно остается обязательным.

II.4.32. Перегрузка шаблонов

Для того чтобы перегрузить шаблон, для какого-то отдельного типа предназначена конструкция `template <>`. В угловых скобках после имени класса уточняется тип, для которого перегружается реализация. В листинге II.4.95 демонстрируется перегрузка шаблона для типа `int`. Если будет объявлен объект `arr` с параметром `int`, вместо значения закрытого члена `var`, метод `get()` будет возвращать строку "Передано целочисленное значение".

Листинг II.4.95. Перегрузка шаблона

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

```
template <class X>
class arr
{
  public:
    arr(X par);
    X get();
  private:
    X var;
};
template <class X> arr<X>::arr(X par)
{
  var = par;
}
template <class X> X arr<X>::get()
{
  return var;
}

template <>
class arr<int>
{
  public:
    arr(int par);
    char *get();
  private:
    int var;
};
arr<int>::arr(int par)
{
  var = par;
}
char *arr<int>::get()
{
  return "Передано целочисленное значение";
}

int main()
{
  arr <double> fst(10.2);
  cout << fst.get() << "\n"; // 10.2
  arr <int> snd(5);
  cout << snd.get() << "\n"; // Передано целочисленное значение

  return 0;
}
```

II.4.33. Обобщенный двухсвязный список

Данная задача перекликается с задачей II.3.17, только при решении этой задачи удобнее воспользоваться не структурой и глобальным указателем, а двумя классами. Первый класс `element` представляет обобщенный элемент списка, его определение представлено в листинге II.4.96.

Замечание

При определении шаблона вместо конструкции `class` используется ключевое слово `typename`, которое было введено в новом стандарте языка, чтобы не путать ключевые слова `class`, предназначенные для создания шаблонов и классов.

Листинг II.4.96. Класс `element`

```
template <typename X>
class element
{
  public:
    X number;
    // Ссылка на следующий элемент списка
    element *next;
    // Ссылка на предыдущий элемент списка
    element *prev;
    // Конструктор
    element(X &num);
};
// Конструктор элемента списка
template <typename X> element<X>::element(X &num)
{
  next = NULL;
  prev = NULL;
  number = num;
}
```

Объекты класса `element` не предназначены для непосредственного использования, данными объектами будет оперировать класс списка `list` (листинг II.4.97).

Листинг II.4.97. Класс `list`

```
template <typename X>
class list
```

```
{
  public:
    // Конструктор
    list();
    // Деструктор
    ~list();
    // Добавить элемент справа
    void add_next(X number);
    // Добавить элемент слева
    void add_prev(X number);
    // Вывести содержимое списка
    void print_list();
    // Удалить последний элемент списка
    void delete_last_element();
    // Получить количество элементов в списке
    int get_count();
  private:
    element <X> *lst;
    element <X> *get_first();
    element <X> *get_last();
};
// Добавление элемента справа от текущего
template <typename X> void list<X>::add_next(X number)
{
  element<X> *item = new element <X> (number);
  if(item != NULL && lst != NULL)
  {
    item->next = lst->next;
    item->prev = lst;
    lst->next = item;
    if(item->next != NULL) item->next->prev = item;
  }
  if(lst == NULL) lst = item;
}
// Добавление элемента слева от текущего
template <typename X> void list<X>::add_prev(X number)
{
  element<X> *item = new element<X> (number);
  if(item != NULL && lst != NULL)
  {
    item->next = lst;
    item->prev = lst->prev;
    lst->prev = item;
```

```
    if(item->prev != NULL) item->prev->next = item;
  }
  if(lst == NULL)
  {
    lst = item;
    lst->next = NULL;
    lst->prev = NULL;
  }
}
// Получение первого элемента списка
template <typename X> element<X> *list<X>::get_first()
{
 if(lst != NULL)
 {
   while(lst->prev != NULL) lst = lst->prev;
 }
 return lst;
}
// Получение последнего элемента списка
template <typename X> element<X> *list<X>::get_last()
{
 if(lst != NULL)
 {
   while(lst->next != NULL) lst = lst->next;
 }
 return lst;
}
// Выводим содержимое списка
template <typename X> void list<X>::print_list()
{
  if(lst != NULL)
  {
    // Получаем первый элемент списка
    for(element<X> *current = get_first();
        current != NULL; current = current->next)
    {
      // Выводим содержимое списка
      cout << current->number << "\n";
    }
  }
}
// Удаление последнего элемента списка
template <typename X> void list<X>::delete_last_element()
```

```
{
  lst = get_last();
  if(lst != NULL)
  {
    if(lst->prev != NULL)
    {
      lst->prev->next = lst->next;
      lst = lst->prev;
      delete lst->next;
    }
    else
    {
      delete lst;
      lst = NULL;
    }
  }
  return;
}
// Деструктор — удаляем элементы
template <typename X> list<X>::~list()
{
  // Получаем первый элемент списка
  for(lst = get_first(); lst != NULL; lst = lst->next)
  {
    delete lst->prev;
  }
  lst = NULL;
}
// Подсчет числа элементов списка
template <typename X> int list<X>::get_count()
{
  int count = 0;
  lst = get_first();
  if(lst != NULL)
  {
    ++count;
    while(lst->next != NULL)
    {
      lst = lst->next;
      ++count;
    }
  }
  return count;
}
```

В листинге II.4.98 демонстрируется пример, создающий двухсвязный список, состоящий из element<double> значений.

Листинг II.4.98. Использование двухсвязного списка

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

int main()
{
  list <double> obj;
  // Указатель на начало списка
  obj.add_next(9.3);
  obj.add_next(2.2);
  obj.add_next(1.1);
  obj.add_next(8.2);
  obj.add_next(5.3);

  obj.print_list();

  return 0;
}
```

В качестве добавляемых элементов могут выступать даже другие двухсвязные списки (листинг II.4.99). Однако для полноценной работы с такими конструкциями понадобится реализовать копирующий конструктор и перегрузить оператор = для классов element и list. Кроме того, потребуется перегрузить оператор вывода в стандартный поток <<.

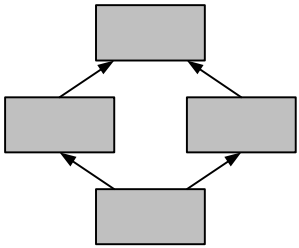
Замечание

Обратите внимание на пробел после конструкции <double>: он требуется для того, чтобы не образовывался оператор >>, который имеет больший приоритет по сравнению с угловыми скобками.

Листинг II.4.99. Объявление списка списков

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

int main()
{
  list <list<double> > lst;

  return 0;
}
```

ГЛАВА II.5



Исключения

II.5.1. Генерация исключений

В качестве объекта исключения может выступать любой базовый или производный объект. В листинге II.5.1 приводится пример обработки целочисленного исключения.

Листинг II.5.1. Обработка целочисленного исключения

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

void funct(int number);

int main()
{
  try
  {
    funct(-1);
  }
  catch(int arg)
  {
    cout << "Ошибка N " << arg << "\n";
  }

  return 0;
}

void funct(int number)
{
  if(number < 0 || number > 100) throw 0;
}
```

Если функция `funct()` генерирует при помощи оператора `throw` исключение целочисленного типа, его можно обработать при помощи обработчика `catch`, принимающего целый аргумент. Результатом работы программы из листинга II.5.1 будет вывод строки

```
Ошибка N 0
```

Объект исключения не обязательно может быть целочисленным, он может принимать, например, тип `float` (листинг II.5.2).

Листинг II.5.2. Обработка исключения типа `float`

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

void funct(int number);

int main()
{
  try
  {
    funct(-1);
  }
  catch(int arg)
  {
    cout << "Ошибка N (int) " << arg << "\n";
  }
  catch(float arg)
  {
    cout << "Ошибка N (float) " << arg << "\n";
  }

  return 0;
}

void funct(int number)
{
  float var = 3.2;
  if(number < 0 || number > 100) throw var;
}
```

При выполнении программы из листинга II.5.2 первый обработчик `catch()`, принимающий целочисленный аргумент, будет проигнорирован, т. к. тип генерируемого исключения не совпадает с типом аргумента обработчика. Ис-

ключение будет "поймано" вторым обработчиком `catch()`, а результатом программы будет строка:

```
Ошибка N (float) 3.2
```

Однако в реальных программах исключения, как правило, являются объектами пользовательских классов, т. к. они более подходят для описания ошибок (листинг II.5.3).

Листинг II.5.3. Использование объекта класса в качестве исключения

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

class myExp
{
  private:
    char var[80];
  public:
    myExp(char *str)
    {
      strcpy(var, str);
    }
    char *get_error()
    {
      return var;
    }
};

int main()
{
  int var = 0;
  try
  {
    cout << "Введите число ";
    cin >> var;
    if(var < 0 || var > 100)
    {
      throw myExp("Выход за границы массива\n");
    }
  }
  catch(myExp e)
  {
    cout << e.get_error() << "\n";
  }

  return 0;
}
```

В листинге II.5.3 в качестве исключения генерируется объект класса myExp, это позволяет при генерации сообщения передать строку с текстовым сообщением об ошибке, а при необходимости передать и другую информацию, нужную для корректной обработки исключительной ситуации.

II.5.2. Перехват исключений в иерархии классов

Если имеет место иерархия исключений, т. е. от базового класса исключения наследуются производные классы исключений, то обработчик базового класса может перехватывать исключение производного типа. Однако обработчик производного типа не может перехватывать исключение базового класса.

Пусть производный класс child_exception наследуется от базового класса base_exception (листинг II.5.4).

Листинг II.5.4. Иерархия классов исключений

```
class base_exception
{
  private:
    char var[80];
  public:
    base_exception(char *str)
    {
      strcpy(var, str);
    }
    char *get_error()
    {
      return var;
    }
};
class child_exception : public base_exception
{
  public:
    child_exception(char *str) : base_exception(str) {};
};
```

Так как базовый класс base_exception имеет единственный конструктор, принимающий параметр типа char *, производный класс child_exception должен осуществлять инициализацию базового класса путем передачи параметра str конструктору base_exception().

```
child_exception(char *str) : base_exception(str) {};
```

За основу демонстрационной программы возьмем листинг II.5.3, в случае возникновения исключительной ситуации (выхода введенного пользователем числа из интервала 0—100) будет генерироваться исключение базового типа `child_exception` (листинг II.5.5).

Листинг II.5.5. Перехват исключения производного класса обработчиком базового класса

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

int main()
{
  int var = 0;
  try
  {
    cout << "Введите число ";
    cin >> var;
    if(var < 0 || var > 100)
    {
      throw child_exception("Выход за границы массива\n");
    }
  }
  catch(base_exception e)
  {
    cout << "base_exception " << e.get_error();
  }
  catch(child_exception e)
  {
    // Этот обработчик никогда не сработает, т. к. предыдущий
    // обработчик будет перехватывать все исключения base_exception
    // и child_exception
    cout << "child_exception " << e.get_error();
  }

  return 0;
}
```

Программа из листинга II.5.5 в случае возникновения исключительной ситуации возвратит строку

```
base_exception Выход за границы массива
```

Обработчик для исключений класса `child_exception` никогда не сработает. Поэтому, если необходимо, чтобы обработчик `child_exception` перехватывал исключения производного класса `child_exception`, а обработчик базового

класса обрабатывал исключения base_exception и других производных классов, следует размещать обработчик base_exception самым последним (листинг II.5.6).

Листинг II.5.6. Обработчик base_exception лучше размещать последним

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

int main()
{
  int var = 0;
  try
  {
    cout << "Введите число ";
    cin >> var;
    if(var < 0 || var > 100)
    {
      throw child_exception("Выход за границы массива\n");
    }
  }
  catch(child_exception e)
  {
    cout << "child_exception " << e.get_error();
  }
  catch(base_exception e)
  {
    cout << "base_exception " << e.get_error();
  }

  return 0;
}
```

II.5.3. Перехват всех исключений

Когда в системе существует большое количество типов исключений, удобно иметь обработчик, который бы перехватывал все необработанные исключения. Для его создания необходимо конструкции catch передать вместо параметра многоточие, как функции с неопределенным количеством параметров (листинг II.5.7).

Листинг II.5.7. Перехват всех исключений

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

```
int main()
{
  int var = 0;
  try
  {
    cout << "Введите число ";
    cin >> var;
    if(var < 0 || var > 100)
    {
      throw 99;
    }
    else
    {
      throw "Исключение";
    }
  }
  catch(...)
  {
    cout << "Исключительная ситуация";
  }

  return 0;
}
```

Конструкция `catch(...)` перехватывает любые исключения, которые не обрабатываются предыдущими обработчиками (если они имеются). Обработчик, перехватывающий все исключения, следует располагать последним.

II.5.4. Функция, генерирующая исключение

Функция `print_array()`, которая содержит в своем составе контролирующий блок `try...catch`, должна передавать исключение за свои пределы. Для этого обычно применяется повторное генерирование исключения, для чего в обработчике `catch()` добавляется вызов `throw` без параметров (листинг II.5.8).

Листинг II.5.8. Повторный вызов исключений

```
void print_array(const int *arr, int count)
{
  try
  {
    if(count <= 10) throw -1;
    if(count > 100) throw "too_long";
```

```
    for(int i = 0; i < count; i++)
    {
      if(arr[i] < 0) throw 1.0;
      cout << arr[i] << "\n";
    }
  }
  catch(int e)
  {
    cout << "Количество элементов в массиве меньше 10\n";
    // Повторное генерирование исключения
    throw;
  }
  catch(char *e)
  {
    cout << "Количество элементов в массиве больше 100\n";
    // Повторное генерирование исключения
    throw;
  }
  catch(float e)
  {
    cout << "Один из элементов массива меньше нуля\n";
    // Повторное генерирование исключения
    throw;
  }
}
```

После того, как исключительная ситуация будет обработана, исключение будет передано за пределы функции, и его сможет обработать внешний обработчик (листинг II.5.9).

Листинг II.5.9. Использование функции `print_array()`

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

void print_array(const int *arr, int count);

int main()
{
  int number = 0;
  int *arr;
  // Запрашиваем у пользователя количество элементов в массиве
  cout << "Введите количество элементов в массиве ";
```

```
  cin >> number;
  try
  {
    arr = new int[number];
    for(int i = 0; i < number; i++) arr[i] = i;
    print_array(arr, number);
    delete [] arr;
  }
  catch(...)
  {
    // Какое бы исключение не произошло, освобождаем память
    cout << "Удаляем память, выделенную под массив arr\n";
    delete [] arr;
  }

  return 0;
}
```

Как видно из листинга II.5.9, при генерации любого из исключений внутри функции `print_array()` исключение будет перехвачено внешним обработчиком, в котором ранее выделенная динамическая память возвращается системе.

Иногда требуется регламентировать явно, какие исключения функция может генерировать. Для этого используется ключевое слово `throw`, размещаемое сразу после имени функции (листинг II.5.10).

Листинг II.5.10. Ограничение типа исключений, генерируемых функцией `print_array()`

```
void print_array(const int *arr, int count) throw (int, char *, float)
{
  try
  {
    if(count <= 10) throw -1;
    if(count > 100) throw "too_long";
    float var = 1.0;
    for(int i = 0; i < count; i++)
    {
      if(arr[i] < 0) throw var;
      cout << arr[i] << "\n";
    }
  }
```

```
  catch(int e)
  {
    cout << "Количество элементов в массиве меньше 10\n";
    // Повторное генерирование исключения
    throw;
  }
  catch(char *e)
  {
    cout << "Количество элементов в массиве больше 100\n";
    // Повторное генерирование исключения
    throw;
  }
  catch(float e)
  {
    cout << "Один из элементов массива меньше нуля\n";
    // Повторное генерирование исключения
    throw;
  }
}
```

Если необходимо запретить функции `print_array()` генерировать какие бы то ни было исключения, достаточно передать пустой список после ключевого слова `throw()` (листинг II.5.11).

Листинг II.5.11. Запрет генерации функцией `print_array()` генерации исключений

```
void print_array(const int *arr, int count) throw ()
{
  ...
}
```

Функция `print_array()` из листинга II.5.11 не будет выпускать за свои пределы ни одного из исключений. При попытке генерации исключения, которое не поддерживается функцией, будет вызываться библиотечная функция `unexpected()`, по умолчанию вызывающая функцию `abort()`. Функция `abort()` в свою очередь аварийно завершает программу.

II.5.5. Выделение динамической памяти

Если оператор `new` по той или иной причине не может выделить динамическую память, он генерирует исключение `bad_alloc`, получить доступ к которому можно, включив в программу библиотеку `<new>`. Если исключение не

обрабатывается, программа завершается аварийно. Такое поведение приемлемо для коротких программ, но совершенно недопустимо в больших приложениях, работающих длительное время. В листинге II.5.12 приводится пример обработки исключительной ситуации `bad_alloc` при выделении памяти под динамический массив.

Листинг II.5.12. Обработка исключения `bad_alloc`

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

int main()
{
  int number = 0;
  int *arr;
  // Запрашиваем у пользователя количество элементов в массиве
  cout << "Введите количество элементов в массиве ";
  cin >> number;

  try
  {
    arr = new int[number];

    delete [] arr;
  }
  catch(bad_alloc expt)
  {
    cout << "Не удалось выделить память под объект arr\n";
  }

  return 0;
}
```

Теперь, если пользователь введет отрицательное число или объем, превышающий объем оперативной памяти, программа выведет сообщение:

```
Не удалось выделить память под объект arr
```

Ранее в случае неудачи вместо генерации исключения оператор `new` возвращал значение `NULL`, по аналогии с оператором `malloc` в языке C. Такое поведение допускается и для оператора `new`, если используется ключевое слово `nothrow` (листинг II.5.13).

Листинг II.5.13. Использование ключевого слова nothrow

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

int main()
{
  int number = 0;
  int *arr;
  // Запрашиваем у пользователя количество элементов в массиве
  cout << "Введите количество элементов в массиве ";
  cin >> number;

  arr = new(nothrow) int[number];
  if(!arr)
  {
    cout << "Не удалось выделить память под объект arr\n";
    return 1;
  }

  delete [] arr;

  return 0;
}
```

Если при выделении динамической памяти при помощи оператора `new` указывается ключевое слово `nothrow`, при неудаче исключение `bad_alloc` не генерируется, а вместо этого оператор `new` возвращает значение `NULL`.

II.5.6. Перегрузка операторов *new* и *delete*

При перегрузке операторов `new` и `delete` следует помнить, что используются два варианта этих операторов: для работы с одним элементом (`new` и `delete`) и массивом (`new []` и `delete []`), поэтому в классе `var_class` необходимо реализовать по два варианта этих операторов (листинг II.5.14).

Замечание

Для перегрузки `nothrow`-версий операторов `new` и `delete` *(см. разд. II.5.5)* функциям, перегружающим операторы, необходимо передать второй параметр типа `const nothrow_t`. Данный параметр не используется и служит лишь маркером для того, чтобы компилятор отличал операторы, генерирующие исключение `bad_alloc`, от операторов, не генерирующих это исключение.

Листинг II.5.14. Перегрузка операторов new и delete

```
class var_class
{
  public:
    int var;
    void *operator new(size_t size);
    void operator delete(void *prt);
    void *operator new[](size_t size);
    void operator delete[](void *prt);
};
```

Операторы new принимают объем памяти в байтах, который необходимо выделить, а операторы delete принимают указатели типа void * на участки памяти, которые следует освободить. В листинге II.5.15 приводится возможная реализация операторов.

Листинг II.5.15. Реализация операторов new и delete

```
void *var_class::operator new(size_t size)
{
  void *prt;
  prt = malloc(size);
  // Если память не выделена - генерируем исключение bad_alloc
  if(!prt)
  {
    bad_alloc expt;
    throw expt;
  }
  return prt;
}

void var_class::operator delete(void *prt)
{
  free(prt);
}

void *var_class::operator new[](size_t size)
{
  void *prt;
  prt = malloc(size);
  // Если память не выделена - генерируем исключение bad_alloc
  if(!prt)
```

```
  {
    bad_alloc expt;
    throw expt;
  }
  return prt;
}

void var_class::operator delete[](void *prt)
{
  free(prt);
}
```

Пользоваться операторами `new` и `delete` можно как стандартными операторами (листинг II.5.16).

Замечание

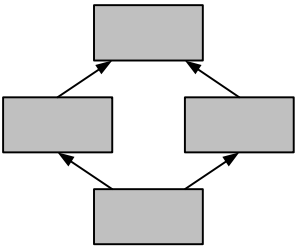
В данной задаче поведение операторов `new` и `delete` мало, чем отличается от стандартных операторов. Обычно эти операторы перегружают, если требуется изменить поведение операторов, например, вместо размещения в оперативной памяти размещать полученную информацию на жестком диске.

Листинг II.5.16. Использование перегруженных операторов `new` и `delete`

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

int main()
{
  int number = 0;
  var_class *arr;
  // Запрашиваем у пользователя количество элементов в массиве
  cout << "Введите количество элементов в массиве ";
  cin >> number;

  try
  {
    arr = new var_class[number];

    delete [] arr;
  }
```

```
  catch(bad_alloc expt)
  {
    cout << "Не удалось выделить память под объект arr\n";
  }

  return 0;
}
```

ГЛАВА II.6



Стандартная библиотека

II.6.1. Стандартное пространство имен

Любая программа на C++ подключает большое количество библиотечных файлов. Имена функций или идентификаторов в разных библиотеках могут совпадать, приводя к конфликту. Для решения этой проблемы в C++ используется механизм пространства имен, который позволяет объединить идентификаторы в именованной области видимости. Для объявления пространства имен используется ключевое слово `namespace` (листинг II.6.1).

Замечание

Несмотря на то, что механизм пространства имен позволяет избежать конфликта идентификаторов из разных библиотек, имена пространств имен могут конфликтовать с другими именами пространства имен.

Листинг II.6.1. Объявление пространства имен

```
namespace mySpace
{
  class myClass;
  void myGlobalFunct();
  ...
}
```

Теперь для того чтобы объявить объект класса `myClass` из пространства имен или вызвать функцию `myClobalFunct`, можно предварить их имена префиксом `mySpace::` (листинг II.6.2).

Замечание

Существуют и неименованные пространства имен, которые позволяют ограничить видимость идентификаторов текущим файлом. Имена из неименованного

пространства имен используются без префикса и не видны в других файлах, даже если текущий файл включается при помощи директивы `#include`.

Листинг II.6.2. Использование идентификаторов из пространства имен

```
mySpace::myClass obj;
...
mySpace::myGlobalFunct();
```

В отличие от классов пространства имен открыты для включения в них новых идентификаторов. Поэтому пространства имен могут использоваться для защиты от конфликтов уже разработанных библиотек. Более того, пространство имен может быть разделено на части, а разные его фрагменты могут находиться в разных файлах (листинг II.6.3).

Листинг II.6.3. Пространство имен может быть разбито на части

```
namespace mySpace
{
  class myClass;
  ...
}
...
namespace mySpace
{
  void myGlobalFunct();
  ...
}
```

Раньше идентификаторы стандартной библиотеки располагались в глобальной области видимости. С введением механизма пространства имен идентификаторы стандартной библиотеки располагаются в пространстве имен `std` (листинг II.6.4).

Листинг II.6.4. Использование префикса `std::`

```
#include <iostream>

int main()
{
  std::cout << "Hello, world!\n";

  return 0;
}
```

Для того чтобы не писать каждый раз префикс пространства имен, можно открыть прямой доступ ко всем идентификаторам пространства имен. Для этого используется директива `using namespace` (листинг II.6.5).

Замечание

Так как в данной книге работа ведется только со стандартной библиотекой, на протяжении всей книги в примерах приведена директива `using namespace`. Однако в реальных программах, использующих несколько библиотек, это не всегда целесообразно. Обычно директиву `using namespace` используют по отношению к пространству имен, чьи идентификаторы и функции применяются наиболее часто.

Листинг II.6.5. Использование директивы `using namespace`

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

int main()
{
  cout << "Hello, world!\n";

  return 0;
}
```

Можно открыть прямой доступ не ко всему пространству имен, а лишь к отдельным его элементам. В листинге II.6.6 доступ открывается лишь к объекту-потоку вывода `cout`, при этом обращение к объекту-потоку ввода потребует использования префикса `std::cin` (листинг II.6.6).

Листинг II.6.6. Использование директивы `using`

```
#include <iostream>
using std::cout;

int main()
{
  cout << "Hello, world!\n";

  return 0;
}
```

II.6.2. Класс *auto_ptr*

"Умными" или "интеллектуальными" указателями называют указатели, которые позволяют предотвратить утечку ресурсов, повторное удаление уже уда-

ленного объема памяти, предоставляют счетчик ссылки и т. п. Как правило, такие указатели реализуются в виде класса-обертки к обычному указателю. Существует большое количество реализаций умных указателей, стандартная библиотека предоставляет одну из реализаций `auto_ptr`, которая предназначена для предотвращения утечки ресурсов при работе с исключениями.

Замечание

Помимо класса `auto_ptr` стандартная библиотека снабжена еще одним видом умных указателей — итераторов, которые позволяют обходить коллекции.

Указатель `auto_ptr` является владельцем объекта, на который он ссылается, устроен он так, что при уничтожении указателя автоматически уничтожается и объект. Поэтому при использовании указателя `auto_ptr` можно не беспокоиться о явном освобождении ресурсов в случае возникновения исключения (листинг II.6.7).

Замечание

Указатель `auto_ptr` не может работать с массивами, однако массив может входить в состав объекта. При разработке стандартной библиотеки предполагалось, что вместо массивов будут использоваться стандартные контейнерные классы.

Листинг II.6.7. Функция `print_obj` с использованием класса `auto_ptr`

```
void print_obj(int number)
{
  try
  {
    auto_ptr<positive> obj(new positive(number));
    cout << obj->get_variable();
  }
  catch(bad_alloc expt)
  {
    cout << "Не удалось выделить память под объект arr\n";
  }
}
```

При сравнении листингов I.6.2 и II.6.7 можно заметить, что блок, обрабатывающий исключение, которое генерирует метод `get_variable()`, исчез, т. к. теперь нет необходимости освобождать динамическую память, выделенную под объект — память будет освобождена автоматически при уничтожении объекта `obj`.

II.6.3. Присваивание и класс *auto_ptr*

Класс `auto_ptr` не является полноценной заменой копирующему конструктору и оператору присваивания. Дело в том, что при выполнении копирования указателя `auto_ptr` или инициализации им другого указателя происходит передача прав владения. В листинге II.6.8 объявляются два указателя `auto_ptr` на объект `position` *(см. разд. II.6.2)*.

Замечание

Оператор = перегружен таким образом, что он модифицирует правый операнд, лишая его прав владения объектом.

Листинг II.6.8. Передача объекта от одного указателя другому

```
auto_ptr<positive> fst(new positive(number));
auto_ptr<positive> snd;
snd = fst;
```

При присваивании указателя `fst` указателю `snd` последний получает в свое личное пользование объект `positive`, при этом объект `fst` указывает на `NULL`. Это позволяет обойти проблему, когда выделенная внутри объекта память удаляется дважды или два указателя `auto_ptr` ссылаются на один и тот же участок памяти. Поэтому оператор "равно" = и копирующий конструктор объекта при использовании указателя `auto_ptr` просто не используются, однако и копию объекта получить невозможно. Поэтому инициализация объекта, на который ссылается указатель `auto_ptr`, осуществляется при помощи круглых скобок, а не оператора = (листинг II.6.9).

Листинг II.6.9. Инициализация объекта при помощи оператора = не допускается

```
auto_ptr<positive> obj(new positive(number));  // Правильно
auto_ptr<positive> obj = new positive(number); // Ошибка
```

Из этого следует важный вывод: если указатель `auto_ptr` передается функции, функция должна вернуть его при помощи оператора `return` назад, иначе объект будет уничтожен при завершении функции, а параметр, переданный функции, будет ссылаться на `NULL` (листинг II.6.10).

Листинг II.6.10. Передача `auto_ptr` внутрь функции

```
// obj становится владельцем объекта класса positive
auto_ptr<positive> funct(auto_ptr<positive> obj)
```

```
{
  ...
  // Работа с obj
  ...
  // Функция передает права владения за пределы, если права не передать
  // объект positive будет уничтожен
  return obj;
}
```

II.6.4. Какие типы контейнеров поддерживаются в STL?

Контейнерные классы библиотеки STL (которые коротко называют контейнерами) управляют коллекциями элементов. Различают последовательные и ассоциативные контейнеры. В последовательных контейнерах позиция элемента зависит от места вставки, т. е. элементы следуют в том порядке, в котором они были помещены в контейнер. В ассоциативных контейнерах элементы хранятся в отсортированном виде, положение элемента в коллекции определяется не позицией вставки, а критерием сортировки (который можно назначать).

К последовательным контейнерам относят:

- вектор (`vector`) — тип контейнера очень похожий на динамический массив, позволяет выполнять обращение к произвольному элементу по его индексу (каждый элемент коллекции имеет свою позицию). Быстро выполняются операции вставки и удаления элементов из конца коллекции. На рис. I.6.1 этот тип контейнера обозначен буквой В;
- дека (`deque`) — двухсторонняя очередь (double-ended queue), так же представляет собой динамический массив с возможностью обращения к произвольному элементу коллекции по его индексу. На рис. I.6.1 этот тип контейнера обозначен буквой Г;
- список (`list`) — двухсвязный список, в котором элементы не имеют четкой позиции, а имеют в своем составе указатели на предыдущий и следующий элементы, образуя цепочку, в любую часть которой очень просто вставлять новый элемент (в отличие от вектора и деки, в которых такие операции очень дороги, т. к. связаны с удалением старой коллекции и созданием новой). При вставке нового значения необходимо лишь исправить указатели соседних элементов. Однако в отличие от векторов и дек, прямого обращения к элементам по индексу коллекция не предусматривает, для обращения к десятому элементу придется осуществить десять переходов по цепочке ссылок от первого элемента. На рис. I.6.1 этот тип контейнера обозначен буквой Б.

Замечание

Создание двухсвязного списка рассматривается в *разд. II.3.17* и *II.4.33*.

К ассоциативным контейнерам относят:

- множество (`set`) — коллекция, которая хранит уникальные (неповторяющиеся) элементы в отсортированном порядке (критерий сортировки можно изменять). Помимо множества библиотека STL предоставляет контейнер "мультимножество" (`multiset`), в котором можно хранить, в том числе, и повторяющиеся элементы. Данный тип контейнера, как и список, не допускает прямого обращения по индексу (оно не имеет значения, т. к. положение элемента определяется критерием сортировки). Но данный тип контейнера удобен для поиска, который в отличие от предыдущих типов контейнеров осуществляется быстрее за счет того, что такой контейнер реализуется в виде бинарного дерева. На рис. I.6.1 этот тип контейнера обозначен буквой А;
- отображение (`map`) — коллекция для хранения уникальных пар соответствий "ключ-значение", так же как и в контейнере "множество", элементы поддерживаются в отсортированном виде. Тут также имеется модификация мультиотображения (`multimap`) для хранения коллекций с повторяющимися элементами. На рис. I.6.1 этот тип контейнера обозначен буквой Д.

II.6.5. Работа с вектором

Вектор (`vector`) управляет элементами, которые хранятся в динамическом массиве. Характерной особенностью данной коллекции является способность обращаться к элементам коллекции напрямую по индексу. Кроме того, операции добавления и удаления из конца коллекции происходят очень быстро (другие операции требуют выделять новый участок памяти под коллекцию, копировать элементы из старого участка с последующим его удалением). Для работы с коллекцией "вектор" необходимо подключить библиотеку `<vector>` (листинг II.6.11).

Листинг II.6.11. Заполнение коллекции "вектор"

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

int main()
{
  // Вектор с целочисленными элементами
  vector<int> coll;
```

```
  int count = 0;
  cout << "Введите число ";
  cin >> count;

  try
  {
    // Заполняем коллекцию
    for(int i = 1; i <= count; i++)
    {
      coll.push_back(i);
    }

    // Выводим содержимое коллекции
    for(int i = 0; i < coll.size(); i++)
    {
      cout << coll[i] << ' ';
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Как видно из листинга II.6.11, объявление векторной коллекции `coll` осуществляется при помощи шаблонного класса `vector`:

```
vector<int> coll;
```

Весь дальнейший код, который осуществляет работу с коллекцией, заключен в блок `try...catch`, перехватывающий исключение динамического выделения памяти `bad_alloc`. Это необходимо, если пользователь введет слишком большое число и в системе не окажется нужного объема оперативной памяти или введенное число выйдет за границы числа. Впрочем, последнюю ситуацию можно исправить, если тип `int` исправить на `unsigned int` *(см. разд. II.1.4 и II.1.5)*.

Метод `push_back()` коллекции "вектор" позволяет добавить в коллекцию новый элемент:

```
coll.push_back(i);
```

Метод `size()` позволяет получить количество элементов в коллекции. Как видно из листинга II.6.11, обращение к элементам производится при помощи квадратных скобок

```
cout << coll[i] << ' ';
```

Представленный в листинге II.6.11 способ перебора коллекции не является типичным. Обычно для этого используются итераторы, которые позволяют создать единый код для перебора коллекции любого типа. Например, обращение по индексу с помощью квадратных скобок не подойдет для списков, множеств и отображений, вместо этого используется итератор (листинг II.6.12).

Замечание

Как видно из листинга II.6.12, применительно к итератору используется префиксная форма инкремента вместо постфиксной, которая возвращает значение и поэтому более затратная по скорости и ресурсам. Если используется базовый тип (например, `int`), компилятор легко распознает такую ситуацию и сам заменяет постфиксную форму на префиксную. В случае класса, компилятор не может делать каких-либо предположений относительно реализации оператора ++ (оператор может быть перегружен совершенно неожиданным образом и осуществлять отличную от инкремента операцию), поэтому автоматическая оптимизация не производится. В подобных случаях лучше использовать префиксную форму.

Листинг II.6.12. Использование константного итератора

```
...
// Выводим содержимое коллекции
vector<int>::const_iterator pos;
for(pos = coll.begin(); pos != coll.end(); ++pos)
{
  cout << *pos << ' ';
}
cout << "\n";
...
```

Итератор — это "умный" (или "интеллектуальный") указатель, предназначенный для перебора элементов коллекции. Итератор указывает на один из элементов коллекции и представляет собой класс, который поддерживает операции разыменования `*`, инкремента `++` и декремента `--` для перемещения по коллекции, операторы сравнения `==`, `!=` (для итераторов вектора и деки допускаются операторы сравнения `>`, `>=`, `<` и `<=`). Кроме этого, поддерживается оператор присваивания `=`, позволяющий назначить итератору позицию в коллекции.

Как видно, интерфейс класса реализован таким образом, чтобы соответствовать обычному указателю. Внутреннее поведение итератора зависит от типа коллекции — каждая коллекция обладает своим собственным итератором.

Каждый из контейнеров поддерживает методы `begin()` и `end()`, которые возвращают итератор, установленный, соответственно, на начало коллекции и позицию за последним элементом (рис. II.6.1).

Замечание

Следует обратить внимание, что позиция, на которую указывает `end()`, не существует, и проводить операции с ней нельзя — это позиция, куда будет вставлен новый элемент коллекции.

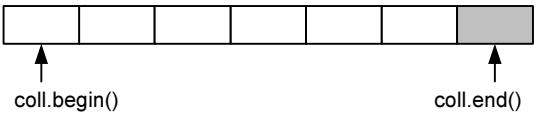


Рис. II.6.1. Методы `begin()` и `end()`

Как видно из листинга II.6.12, цикл выполняется до тех пор, пока итератор `pos` не будет указывать на позицию, следующую за последним элементом коллекции. Для векторов и дек, итераторы которых поддерживают операторы сравнения >, >=, < и <=, допускается форма перебора коллекции, представленная в листинге II.6.13.

Несмотря на то, что сравнение с последним элементом коллекции надежнее и не допускает ситуации, когда итератор случайно выходит за границы коллекции, в качестве критерия выхода из цикла чаще используют неравенство итератора и позиции, возвращаемой методом `end()`, т. к. такой подход может быть применен ко всем коллекциям библиотеки STL.

В каждом контейнере определены два типа итераторов:

- `контейнер::iterator` — итератор используется для перебора элементов в режиме чтения/записи;
- `контейнер::const_interator` — итератор используется для перебора элементов в режиме чтения.

В листингах II.6.11 и II.6.12 использовался константный итератор `const_iterator`, т. к. программе требовалось лишь вывести элементы при помощи итератора, и необходимости редактировать элементы не возникало. В листинге II.6.13 перед выводом элемента коллекции в стандартный поток его значение возводится в квадрат, поэтому используется неконстантный итератор `iterator`, позволяющий изменять значения элементов.

Листинг II.6.13. Использование неконстантного итератора

```
...
// Выводим содержимое коллекции
vector<int>:: iterator pos;
for(pos = coll.begin(); pos != coll.end(); ++pos)
{
  // Возводим в квадрат текущий элемент коллекции
  *pos = *pos * *pos;
  // Выводим элемент коллекции
  cout << *pos << ' ';
}
cout << "\n";
...
```

Итератор обозначает позицию в коллекции и часто используется в методах, поддерживаемых коллекциями. Например, можно осуществить преобразование или копирование не всей коллекции, а ее части, начало и конец которой указываются двумя итераторами. В табл. II.6.1 представлены методы вставки и удаления элементов, которые поддерживает контейнер "вектор".

Таблица II.6.1. *Операции вставки и удаления, поддерживаемые контейнером "вектор"*

Операция	Описание
insert(*pos*, *element*)	Метод вставляет в позицию, на которую указывает итератор *pos*, копию элемента *element*, и возвращает итератор, указывающий на позицию нового элемента
insert(*pos*, *n*, *element*)	Метод вставляет в позицию, на которую указывает итератор *pos*, *n* копий элемента *element* (при этом ничего не возвращается)
insert(*pos*, *begin*, *end*)	Метод вставляет копии элементов коллекции из интервала, определяемыми итераторами *begin* и *end*, в позицию, которую указывает итератор *pos*
push_back(*element*)	Метод вставляет копию элемента *element* в конец коллекции
pop_back()	Метод удаляет последний элемент коллекции
erase(*pos*)	Метод удаляет элемент, на который указывает итератор *pos*, и возвращает позицию следующего элемента
erase(*begin*, *end*)	Метод удаляет все элементы из интервала, определяемого итераторами *begin* и *end*, и возвращает позицию следующего элемента

Таблица II.6.1 *(окончание)*

Операция	Описание
`resize(num)`	Изменяет размер коллекции до значения *num* (для вектора возможно только увеличение коллекции, уменьшение игнорируется), новые элементы вставляются в конец коллекции
`resize(num, element)`	Изменяет размер коллекции до значения *num* (для вектора возможно только увеличение коллекции, уменьшение игнорируется), новые элементы заполняются значением *element*
`clear()`	Удаляет все элементы коллекции

II.6.6. Работа с деком

Дек (`deque`) управляет элементами, которые хранятся в динамическом массиве. Характерной особенностью данной коллекции является способность обращаться к элементам коллекции напрямую по индексу. Кроме того, дек позволяет легко наращивать коллекцию как с начала, так и с конца, для этого предназначаются методы `push_front()` и `push_back()`, соответственно. Для работы с коллекцией "дек" необходимо подключить библиотеку `<deque>` (листинг II.6.14).

Листинг II.6.14. Заполнение коллекции "дек"

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

int main()
{
  // Дек с целочисленными элементами
  deque<int> coll;

  int count = 0;
  cout << "Введите число ";
  cin >> count;

  try
  {
    // Заполняем коллекцию
    for(int i = -count; i <= count; ++i)
```

```
    {
      coll.push_back(i);
    }

    // Выводим содержимое коллекции
    deque<int>::iterator pos;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        cout << *pos << ' ';
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

В листинге II.6.14 для вставки нового элемента использовался метод `push_back()`, добавляющий новый элемент коллекции в конец коллекции, однако с таким же успехом можно вставлять элементы в начало коллекции (листинг II.6.15).

Листинг II.6.15. Использование метода `push_back()`

```
...
for(int i = count; i >= -count; --i)
{
  coll.push_front(i);
}
...
```

Кроме того, можно использовать цикл от 0 до `count` для заполнения коллекции, используя оба метода (и `push_back()`, и `push_front()`) для вставки новых элементов (листинг II.6.16).

Листинг II.6.16. Использование обоих методов `push_back()` и `push_front()`

```
...
coll.push_front(0);
for(int i = 1; i < count; ++i)
```

```
{
  coll.push_back(i);
  coll.push_front(-i);
}
...
```

Как видно из листинга II.6.14, для вывода элементов коллекции `deque` применяется стандартный цикл вывода с помощью итератора. Он аналогичен коду из листинга II.6.13, только вместо итератора `vector` используется итератор `deque`. Это означает, что такой код можно поместить в полиморфную функцию базового класса, и он может быть использован для работы с любым типом контейнера STL.

Для дек характерны такие же операции вставки и удаления элементов, как и для векторов (см. табл. II.6.1), однако, в силу того, что этот тип коллекции допускает работу с началом, появляются дополнительные методы (табл. II.6.2).

Таблица II.6.2. *Операции вставки и удаления, поддерживаемые контейнером "дек"*

Операция	Описание
`push_front(`*`element`*`)`	Метод вставляет копию элемента *`element`* в начало коллекции
`pop_front()`	Метод удаляет первый элемент коллекции

II.6.7. Работа со списком

Список (`list`) организован в виде двухсвязного списка, т. е. в отличие от вектора (`vector`) и дека (`deque`) в его основе лежит не динамический массив. Каждый элемент списка занимает отдельный блок памяти и содержит ссылки на предыдущий и последующий элементы. Список не поддерживает прямое обращение к элементам коллекции при помощи их индекса. Например, чтобы обратиться к пятому элементу, необходимо перебрать первые пять элементов по цепочке ссылок.

Основным достоинством списка является высокая скорость операции вставки и удаления элемента (по сравнению с векторами и деками), независимо от того, осуществляется вставка в начало, конец или середину коллекции. Для работы с коллекцией "список" необходимо подключить библиотеку `<list>` (листинг II.6.17).

Листинг II.6.17. Использование списка (`list`)

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

int main()
{
  // Список с целочисленными элементами
  list<int> coll;

  try
  {
    // Заполняем коллекцию
    for(char ch = 'a'; ch <= 'z'; ++ch)
    {
      coll.push_back(ch);
    }

    // Выводим содержимое коллекции
    list<int>::iterator pos;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        cout << (char)*pos << ' ';
    }
    cout << "\n";

    // Изменяем регистр символов в коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        *pos = toupper(*pos);
    }

    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        cout << (char)*pos << ' ';
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }
```

```
  return 0;
}
```

Для перебора коллекции используется итератор. Следует обратить внимание, что в случае итератора списка (`list`) уже не допускается использование операторов неравенства (>, >=, <, <=).

Список поддерживает все методы вставки и удаления элементов, которые характерны для векторов (см. табл. II.6.1) и дек (см. табл. II.6.2), плюс два дополнительных метода (табл. II.6.3).

Таблица II.6.3. *Операции вставки и удаления, поддерживаемые контейнером "список"*

Операция	Описание
`remove(`*`value`*`)`	Метод удаляет все элементы со значением `value`
`remove_if(`*`op`*`)`	Метод удаляет все значения, для которых предикат *`op`*`()` возвращает значение `true`. Предикаты более подробно рассматриваются в *разд. II.6.20*

Одним из достоинств связных списков является быстрое удаление и вставка элементов в произвольную позицию. Поэтому контейнер `list` снабжен рядом специальных методов, позволяющих перемещать элементы как внутри одного списка, так и между разными списками (табл. II.6.4).

Таблица II.6.4. *Специальные методы модификации списков*

Операция	Описание
`unique()`	Метод удаляет дубликаты из текущей коллекции, в результате чего коллекция содержит только уникальные значения
`unique(`*`op`*`)`	Метод удаляет дубликаты, для которых предикат *`op`*`()` возвращает значение `true`. Предикаты более подробно рассматриваются в *разд. II.6.20*
`splice(`*`pos, coll`*`)`	Метод перемещает все элементы списка *`coll`* в текущую коллекцию перед позицией, определяемой итератором *`pos`*
`splice(`*`pos, coll, pos_coll`*`)`	Метод перемещает все элементы списка *`coll`*, начиная с позиции, определяемой итератором *`pos_coll`*, в текущую коллекцию перед позицией, определяемой итератором *`pos`*

Таблица II.6.4 *(окончание)*

Операция	Описание
`splice(pos, coll, coll_begin, coll_end)`	Метод перемещает все элементы из интервала, определяемого итераторами *coll_begin* и *coll_end* списка *coll*, в текущую коллекцию перед позицией, определяемой итератором *pos*
`sort()`	Метод сортирует все элементы оператором <
`sort(op)`	Метод сортирует все элементы при помощи предиката *op*(). Предикаты более подробно рассматриваются в *разд. II.6.20*
`merge(coll)`	Метод перемещает все элементы списка *coll* в текущую коллекцию с сохранением сортировки (предполагается, что и текущая коллекция, и список *coll* содержат отсортированные элементы)
`merge(coll, op)`	Метод перемещает все элементы списка *coll* в текущую коллекцию с сохранением сортировки при помощи предиката *op*() (предполагается, что оба контейнера содержат элементы, отсортированные по критерию *op*()). Предикаты более подробно рассматриваются в *разд. II.6.20*
`reverse()`	Метод переставляет все элементы в обратном порядке

II.6.8. Работа с множеством

Множество `set` хранит элементы в отсортированном порядке. Для работы с данным типом коллекции необходимо в программу включить заголовочный файл `<set>` (листинг II.6.18).

Листинг II.6.18. Использование множества (`set`)

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

int main()
{
  // Множество
  set<int> coll;
  int input;
```

```
  try
  {
    // Читаем числа из стандартного потока
    while(true)
    {
      cout << "Введите число ";
      cin >> input;
      if(input < 0) break;
      coll.insert(input);
    }

    set<int>::const_iterator pos;
    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        cout << *pos << " ";
    }
    count << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Программа из листинга II.6.18 при помощи бесконечного цикла `while()` читает числа из стандартного потока ввода до тех пор, пока пользователь не введет отрицательное число. При этом каждое число помещается в коллекцию `coll` при помощи метода `insert()`. Ассоциативные контейнеры (множества, мультимножества, отображения и мультиотображения) не поддерживают методы `push_back()` и `push_front()`, которые характерны для последовательных контейнеров. Это связано с тем, что данные коллекции не поддерживают индексацию элементов, позиция каждого из элементов определяется автоматически в соответствии с критерием сортировки.

Пусть пользователь вводит 14 чисел в такой последовательности:

```
3, 14, 1, 10, 2, 7, 9, 13, 4, 6, 8, 11, 5, 12
```

При вставке их в ассоциативный контейнер элементы объединяются в древовидную структуру таким образом, что "левый" потомок любого элемента всегда меньше текущего элемента, а "правый" всегда больше (рис. II.6.2).

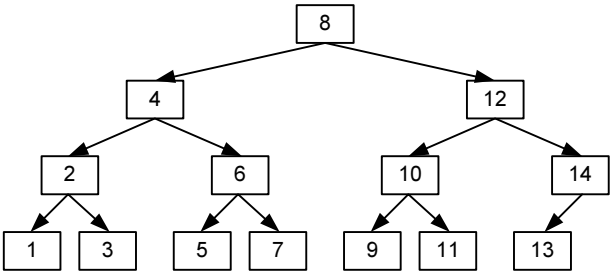


Рис. II.6.2. Расположение элементов в коллекции "множество"

Итератор множества устроен таким образом, что он автоматически выводит элементы коллекции в отсортированном состоянии. Программа из листинга II.6.18 при вводе последовательности 14 чисел в случайном порядке выведет их в порядке возрастания:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Следует отметить, что присутствие дубликатов в контейнере "множество" не допускается. Если пользователь введет последовательность

```
5 5 5 3 3
```

то коллекция будет содержать только два элемента 3 и 5. Для того чтобы в коллекции сохранялись и дублирующие значения, необходимо использовать мультимножество (`multiset`) (листинг II.6.19).

Листинг II.6.19. Использование мультимножества (`multiset`)

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

int main()
{
  // Мультимножество
  multiset<int> coll;
  int input;

  try
  {
    // Читаем строки из стандартного потока
    while(true)
    {
      cout << "Введите число ";
```

```
      cin >> input;
      if(input < 0) break;
      coll.insert(input);
    }

    multiset<int>::const_iterator pos;
    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        cout << *pos << " ";
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

При вводе последовательности

```
5 5 5 3 3
```

программа из листинга II.6.19 выведет строку:

```
3 3 5 5 5
```

Для того чтобы задать другой критерий сортировки, например, чтобы элементы коллекции сортировались в убывающем порядке, необходимо передать такой критерий в качестве второго параметра шаблона. Например, в листинге II.6.20 в качестве критерия сортировки используется стандартный объект-функция `greater<>`. Для удобства в программе определяется новый тип `IntMultiSet`, который используется в программе (это удобно, если в программе определяется несколько коллекций и итераторов).

Замечание

Следует обратить внимание на пробел перед завершающей угловой скобкой >, его отсутствие вызывает ошибку, т. к. компилятор считает последовательность >> оператором сдвига.

Листинг II.6.20. Сортировка элементов коллекции в обратном порядке

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

```
int main()
{
  // Множество
  typedef multiset<int, greater<int> > IntMultiSet;
  IntMultiSet coll;
  int input;

  try
  {
    // Читаем строки из стандартного потока
    while(true)
    {
      cout << "Введите число ";
      cin >> input;
      if(input < 0) break;
      coll.insert(input);
    }

    IntMultiSet::const_iterator pos;
    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
        cout << *pos << " ";
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

В листинге II.6.20 обратная сортировка элементов коллекции достигается изменением критерия сортировки, однако существует еще один способ вывода элементов коллекции в обратном порядке — при помощи адаптера обратного итератора.

Контейнеры "множество" (`set`) и "мультимножество" (`multiset`) поддерживают методы вставки и удаления элементов, которые представлены в табл. II.6.5.

Множества и мультимножества оптимизированы для поиска элементов, поэтому контейнеры данного типа поддерживают специальные функции поиска (табл. II.6.6).

Таблица II.6.5. *Операции вставки и удаления, поддерживаемые контейнерами "множество" и "мультимножество"*

Операция	Описание
`insert(`*`element`*`)`	Метод вставляет новый элемент *element* и возвращает его позицию. Для множества метод возвращает `true` в случае успешной вставки элемента и `false`, если такой элемент уже присутствует в коллекции
`insert(`*`pos`*`, `*`element`*`)`	Метод вставляет новый элемент *element* в позицию, определяемую итератором *pos*. Так как позиция в множестве определяется критерием сортировки, то итератор *pos* задает рекомендуемую позицию, с которой следует начинать поиск конечной позиции для вставляемого элемента
`insert(`*`begin`*`, `*`end`*`)`	Метод вставляет значения из интервала, определяемого итераторами *begin* и *end*
`erase(`*`element`*`)`	Метод удаляет все элементы со значением *element* и возвращает количество удаленных элементов
`erase(`*`pos`*`)`	Метод удаляет элемент, на который указывает итератор *pos*
`erase(`*`begin`*`, `*`end`*`)`	Метод удаляет элементы из интервала, определяемого итераторами *begin* и *end*
`clear()`	Метод удаляет все элементы из коллекции

Таблица II.6.6. *Специальные методы поиска в множествах и мультимножествах*

Операция	Описание
`count(`*`element`*`)`	Метод возвращает количество элементов со значением *element*
`find(`*`element`*`)`	Метод возвращает позицию первого элемента со значением *element*, если такой элемент не найден, возвращается итератор `end()`
`lower_bound(`*`element`*`)`	Метод возвращает первую позицию, в которую может быть вставлен элемент *element*
`upper_bound(`*`element`*`)`	Метод возвращает последнюю позицию, в которую может быть вставлен элемент *element*
`equal_range(`*`element`*`)`	Метод возвращает первую и последнюю позиции, в которых может быть вставлен элемент *element* (интервал, в котором все элементы эквивалентны *element*)

II.6.9. Работа с отображением

Коллекция "отображение" (map) управляет элементами, которые представляют собой пары "ключ/значение". Для облегчения работы с такими элементами STL предоставляет вспомогательный шаблонный класс pair, который содержит два открытых члена — first и second — для ключа и значения соответственно. Для объектов класса pair перегружены операторы сравнения (==, !=, <, >, <=, >=), а также реализована функция make_pair(), позволяющая создавать объекты pair. Например, строки в листинге II.6.21 эквиваленты.

Листинг II.6.21. Использование функции make_pair()

```
make_pair(1, "first");
pair<int, char *>(1, "first");
```

То есть функция make_pair() позволяет заменить конструктор класса pair.

Благодаря тому, что коллекция "отображение" оперирует парами значений, отображение можно использовать в качестве ассоциативного массива. В листинге II.6.22 представлено решение поставленной задачи. Пользователь вводит число прописью, а программа выводит соответствующую ему цифру.

Замечание

Для работы с отображением программа должна подключать заголовочный файл <map>.

Листинг II.6.22. Использование отображения

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

int main()
{
  // Отображение
  typedef map<string, int> StringIntMap;
  StringIntMap coll;

  try
  {
    coll.insert(make_pair("one", 1));
    coll.insert(make_pair("two", 2));
```

```
    coll.insert(make_pair("three", 3));
    coll.insert(make_pair("four", 4));
    coll.insert(make_pair("five", 5));
    coll.insert(make_pair("six", 6));
    coll.insert(make_pair("seven", 7));
    coll.insert(make_pair("eight", 8));
    coll.insert(make_pair("nine", 9));
    coll.insert(make_pair("ten", 9));

    cout << "Введите название числа ";
    char input[80];
    fgets(input, 80, stdin);
    // Удаляем завершающий перевод строки
    input[strlen(input) - 1] = '\0';
    // Создаем строку типа string
    string str(input);

    cout << coll[input] << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Для удобства работы в программе из листинга II.6.22 используется еще один вспомогательный класс `string`, который является улучшенным классом строк, допускающим сравнение строк при помощи логических операторов, а также предоставляющим ряд полезных методов.

Как видно из листинга II.6.22, добавление нового элемента в коллекцию производится при помощи метода `insert()`, который принимает объект класса `pair()`, создаваемый посредством функции `make_pair()`.

Пользователь вводит название цифры, которое помещается в строку `input` типа `char *`, а та, в свою очередь, инициирует строку `str` типа `string`. Предварительно из строки `input` удаляется перевод строки `\n`, который оставляет в ней функция `fgets()`. Характерной особенностью коллекции отображения является то, что к элементам коллекции можно обращаться через ассоциативный ключ при помощи квадратных скобок. Именно этот прием используется в приведенной выше программе.

Замечание

Следует подчеркнуть, что отсутствие индекса не вызывает ошибку, но приводит к вставке нового элемента, что не всегда является желательным эффектом.

Оператор индексирования может использоваться не только для получения значений, но также и для заполнения коллекции. В листинге II.6.23 приводится программа, заполняющая коллекцию `map` и выводящая результаты в стандартный поток.

Замечание

Оператор индексирования `[]` не может использоваться применительно к мультиотображению (`multimap`), т. к. этот тип коллекции допускает несколько элементов с одинаковыми ключами.

Листинг II.6.23. Вывод содержимого коллекции `map`

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

int main()
{
  // Отображение
  typedef map<string, int> StringIntMap;
  StringIntMap coll;

  try
  {
    coll["one"] = 1;
    coll["two"] = 2;
    coll["three"] = 3;
    coll["four"] = 4;
    coll["five"] = 5;
    coll["six"] = 6;
    coll["seven"] = 7;
    coll["eight"] = 8;
    coll["nine"] = 9;
    coll["ten"] = 10;

    StringIntMap::iterator pos;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
```

```
    {
      cout << pos->first << " => " << pos->second << "\n";
    }
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Результат выполнения программы из листинга II.6.23 выглядит следующим образом:

```
eight => 8
five => 5
four => 4
nine => 9
one => 1
seven => 7
six => 6
ten => 10
three => 3
two => 2
```

Следует обратить внимание, что доступ к ключу осуществляется за счет обращения к открытому элементу `first`, а доступ к значению — к открытому элементу `second` объекта класса `pair`. Класс `pair` устроен так, что при сравнении первый компонент `first` (т. е. ключ) обладает более высоким приоритетом по сравнению со вторым компонентом `second` (т. е. значением). Поэтому сортировка происходит по ключу.

Коллекции "отображение" (`map`) и "мультиотображение" (`multimap`) поддерживают те же методы вставки, удаления (см. табл. II.6.5) и поиска (см. табл. II.6.6), что и множества (`set`) и мультимножества (`multiset`).

II.6.10. Преобразование одной коллекции в другую

Да, коллекции допускают преобразование одной коллекции в другую, например, на этапе инициализации. Помимо объявления коллекции и заполнения ее методами `push_back()`, `push_front()` и `insert()` коллекция может быть инициализирована другой коллекцией (листинг II.6.24).

Листинг II.6.24. Инициализация коллекции элементами другой коллекции

```
#include <iostream>
#include <vector>
...
vector<int> coll;
for(int i = 0; i < 10; i++) coll.push_back(i);
vector<int> coll_vector(coll);
...
```

Однако такой способ инициализации подходит только в том случае, если коллекции являются однотипными и содержат одинаковые элементы. В любом другом случае инициализировать коллекцию можно при помощи интервала. В листинге II.6.25 коллекция типа `vector`, содержащая целочисленные значения (`int`), используется для инициирования коллекции типа `list`, которая, в свою очередь, содержит элементы типа `float`.

Замечание

Допускается инициировать коллекцию лишь частью другой коллекции.

Листинг II.6.25. Инициирование коллекции интервалом

```
#include <iostream>
#include <vector>
#include <list>
...
vector<int> coll;
for(int i = 0; i < 10; i++) coll.push_back(i);
list<float> coll_list(coll.begin(), coll.end());
...
```

В листинге II.6.25 происходит не только инициализация коллекции одного типа элементами коллекции другого типа, осуществляется также автоматическое приведение типа `int` к типу `float`.

II.6.11. Допускается ли сравнение коллекций друг с другом?

Коллекции, так же как и элементы, можно сравнивать друг с другом при помощи операторов ==, !=, >, >=, <, <=. Оба контейнера считаются равными друг другу, если их элементы равны и следуют в одинаковом порядке. Отношение "больше" и "меньше" между контейнерами определяется лексикографиче-

ским критерием: элементы контейнеров сравниваются по парам, пока не будет выполнено одно из условий:

- если два элемента не равны, то результат их сравнения определяет результат всего сравнения коллекций;
- коллекция, содержащая меньшее количество элементов, считается меньше другой.

Замечание

Оба контейнера, которые подвергаются сравнению, должны относиться к одному и тому же типу.

Помимо операторов сравнения, каждая из коллекций содержит функции `size()` и `empty()`, первая из которых возвращает количество элементов в коллекции, а вторая возвращает `true`, если коллекция пуста, и `false`, если коллекция содержит хотя бы один элемент.

II.6.12. Сортировка строк

Для решения данной задачи так же удобно воспользоваться вспомогательным классом `string`, который представляет собой более удобный вариант (по сравнению с `char *`) строк. Возможно два варианта решения задачи:

- использование ассоциативного контейнера `multiset`, поддерживающего элементы коллекции в отсортированном состоянии;
- использование алгоритма сортировки `sort()`.

Рассмотрим оба варианта, начав с первого. Для того чтобы строки автоматически сортировались в коллекции, следует объявить мультимножество `multiset<string>` (листинг II.6.26).

Листинг II.6.26. Использование мультимножества для сортировки строк

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

int main()
{
  // Отображение
  typedef multiset<string> StringMultiset;
  StringMultiset coll;
```

```
  try
  {
    char input[80];
    string str;
    while(true)
    {
      cout << "Введите строку ";
      fgets(input, 80, stdin);
      // Если введена пустая строка - выходим из цикла
      if(strlen(input) <= 1) break;
      // Удаляем завершающий перевод строки
      input[strlen(input) - 1] = '\0';
      // Присваиваем строке str значение input
      str = input;
      coll.insert(str);
    }

    StringMultiset::const_iterator pos;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << "\n";
    }
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Как видно из листинга II.6.27, никаких усилий по сортировке прилагать не нужно, программа просто заполняет коллекцию, которая в свою очередь сама заботится о сортировке элементов. Однако не всегда получается использовать ассоциативный контейнер, по соображениям производительности обработки строки может понадобиться хранить в контейнере вектор (`vector`). В этом случае на помощь приходят алгоритмы библиотеки STL.

Замечание

Алгоритмы — это глобальные функции библиотеки STL, которые работают с итераторами. Использование итераторов позволяет создавать алгоритм сразу для всех типов контейнеров и элементов контейнеров.

Сортировка контейнера осуществляется при помощи алгоритма `sort()`, который принимает в качестве первого параметра итератор на начало, а в качестве второго параметра — итератор на конец коллекции. Для использования алгоритмов в код программы необходимо включить заголовочный файл `<algorithm>` (листинг II.6.27).

Замечание

Списки (`list`) поддерживают помимо глобальной функции `sort()` метод коллекции `sort()`.

Листинг II.6.27. Использование алгоритма `sort()` для сортировки коллекции

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  typedef vector<string> StringVector;
  StringVector coll;

  try
  {
    char input[80];
    string str;
    while(true)
    {
      cout << "Введите строку ";
      fgets(input, 80, stdin);
      // Если введена пустая строка - выходим из цикла
      if(strlen(input) <= 1) break;
      // Удаляем завершающий перевод строки
      input[strlen(input) - 1] = '\0';
      // Присваиваем строке str значение input
      str = input;
      coll.push_back(str);
    }

    // Сортируем коллекцию
    sort(coll.begin(), coll.end());
```

```
    StringVector::const_iterator pos;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << "\n";
    }
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Любой алгоритм, в том числе и `sort()`, работает с одним или несколькими интервалами. Например, в листинге II.6.28 в качестве такого интервала выступает вся коллекция (хотя это не обязательно). Следует отметить, что за тем, является ли переданный интервал корректным (итераторы должны быть действительными, а начальный итератор обязан быть больше конечного), должен следить сам программист — библиотека STL не осуществляет такого контроля.

Функция `sort()` может принимать в качестве третьего параметра функцию-критерий сортировки. В листинге II.6.28 приводится модифицированный вариант программы из листинга II.6.27, в котором в качестве критерия сортировки используется функция `less_str()`, позволяющая отсортировать коллекцию в обратном порядке.

Листинг II.6.28. Использование функции-критерия для сортировки коллекции

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

bool less_str(const string &fst, const string &snd)
{
  return fst > snd;
}
```

```
int main()
{
  // Отображение
  typedef vector<string> StringVector;
  StringVector coll;

  try
  {
    char input[80];
    string str;
    while(true)
    {
      cout << "Введите строку ";
      fgets(input, 80, stdin);
      // Если введена пустая строка - выходим из цикла
      if(strlen(input) <= 1) break;
      // Удаляем завершающий перевод строки
      input[strlen(input) - 1] = '\0';
      // Присваиваем строке str значение input
      str = input;
      coll.push_back(str);
    }

    // Сортируем коллекцию
    sort(coll.begin(), coll.end(), less_str);

    StringVector::const_iterator pos;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << "\n";
    }
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Функция `less_str()` (листинг II.6.28) может вводить совершенно другие критерии сортировки, например, выбирая в качестве сортировки количество элементов в строке (листинг II.6.29).

Листинг II.6.29. Сортировка по количеству элементов в строке

```
...
bool less_str(const string &fst, const string &snd)
{
  return fst > snd;
}
...
```

II.6.13. Поиск максимального и минимального значений коллекции

Здесь так же, как и в предыдущем примере, можно либо помещать целочисленные значения, вводимые пользователем в множество, которое будет их автоматически сортировать, либо прибегнуть к алгоритмам библиотеки STL. Первый вариант (размещение вводимых чисел в коллекции "множество") представлен в листинге II.6.30. После того как коллекция заполнена, остается только получить первый и последний ее элементы.

Замечание

Необходимо учитывать, что метод `end()` возвращает итератор, указывающий не на последний элемент коллекции, а на элемент *за* последним элементом коллекции, поэтому следует получать предыдущий элемент при помощи операции декремента.

Листинг II.6.30. Поиск минимального и максимального элементов коллекции

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

int main()
{
  // Множество
  multiset<int> coll;
  int input;

  try
  {
    // Читаем строки из стандартного потока
    while(true)
    {
      cout << "Введите число ";
```

```
      cin >> input;
      if(input < 0) break;
      coll.insert(input);
    }

    // Выводим минимальный элемент коллекции
    cout << "Min = " << *coll.begin() << "\n";
    // Выводим максимальный элемент коллекции
    cout << "Max = " << *(--coll.end()) << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Как видно из листинга II.6.30, вывод результатов производится даже без использования промежуточного итератора. Для вывода минимального значения разыменовывается итератор, возвращаемый методом `begin()`. Метод `end()` указывает на позицию за последним элементом коллекции, поэтому перед разыменовкой итератора производится операция декремента, которая переводит указатель на последний элемент коллекции.

Для поиска минимального и максимального элементов коллекции применяются алгоритмы `min_element()` и `max_element()` соответственно. Точно так же, как и алгоритм `sort()` *(см. разд. II.6.12)*, данные алгоритмы принимают итератор, указывающий на начало интервала, и итератор, указывающий на конец интервала. При необходимости в качестве третьего аргумента может быть передана функция-критерий сортировки. В листинге II.6.31 представлена программа, которая находит максимальное и минимальное значения из последовательности, введенной пользователем.

Листинг II.6.31. Использование алгоритмов `min_element()` и `max_element()`

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

int main()
{
  vector<int> coll;
```

```
  vector<int>::iterator pos;
  int input;

  try
  {
    // Читаем строки из стандартного потока
    while(true)
    {
      cout << "Введите число ";
      cin >> input;
      if(input < 0) break;
      coll.push_back(input);
    }

    // Выводим минимальный элемент коллекции
    pos = min_element(coll.begin(), coll.end());
    cout << "Min = " << *pos << "\n";
    // Выводим максимальный элемент коллекции
    pos = max_element(coll.begin(), coll.end());
    cout << "Max = " << *pos << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

II.6.14. Обращение порядка следования элементов

Для поиска значения в заданном интервале используется алгоритм `find()`, который принимает итератор, указывающий на начало интервала, и итератор, указывающий на конец интервала. При необходимости в качестве третьего параметра может быть передана функция, определяющая критерий сортировки. В случае успеха алгоритм `find()` возвращает итератор, указывающий на найденный элемент, в случае неудачи возвращается указатель на `coll.end()`, т. е. на позицию, которая следует за последним элементом коллекции.

Для того чтобы переставить элементы в заданном интервале, обычно используется алгоритм `reverse()`. В листинге II.6.32 представлена программа, которая обращает порядок следования элементов во второй половине коллекции вектора из листинга I.6.3.

Листинг II.6.32. Обращение части элемента коллекции

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

int main()
{
  vector<int> coll;
  vector<int>::iterator pos;

  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);

    // Получаем итератор, указывающий на элемент со значением 4
    pos = find(coll.begin(), coll.end(), 4);
    // Переставляем элементы коллекции, начиная с текущего
    // и до последнего
    reverse(pos, coll.end());

    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << " ";
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Следует отметить, что в листинге II.6.32 демонстрируется случай, когда алгоритм применяется не ко всей коллекции в целом, а к отдельной ее части. От-

дельно следует рассмотреть случай, когда алгоритм `find()` не находит заданный элемент. В этом случае возвращается значение `coll.end()`, т. е. алгоритму `reverse()` передается пустой интервал:

```
reverse(coll.end(), coll.end())
```

Это вполне допустимо, однако нужно помнить, что элемент `end()` возвращает итератор, который указывает на позицию, следующую за последним элементом коллекции, которая не является действительным элементом. Операции с таким итератором, например, его разыменование, приводят к ошибке.

Помимо обычных итераторов библиотека STL предоставляет итераторные адаптеры, которые кроме обычных свойств итераторов обладают рядом новых возможностей. К таким итераторным адаптерам относят обратные итераторы. Эти итераторы работают в обратном направлении: вызов оператора `++` преобразуется на внутреннем уровне в вызов оператора `--`, и наоборот. Объявить обратный итератор можно при помощи класса `reverse_iterator`, каждый контейнер обладает методами `rbegin()` и `rend()`, также возвращающими обратные итераторы. В листинге II.6.33 демонстрируется программа, использующая обратные итераторы.

Листинг II.6.33. Использование обратных итераторов

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

int main()
{
  vector<int> coll;

  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);

    vector<int>::reverse_iterator rpos;
    for(rpos = coll.rbegin(); rpos != coll.rend(); ++rpos)
```

```
    {
      cout << *rpos << " ";
    }
    cout << "\n";

  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Результатом выполнения программы из листинга II.6.33 будет строка:

```
6 5 4 3 2 1
```

Следует отметить, что значение `*coll.rend()` не определено и указывает на позицию, предшествующую начальному элементу коллекции.

II.6.15. Сортировка содержимого файла

Решение задачи разбивается на три этапа. Во-первых, выполняется чтение файла словаря и занесение отдельных слов в качестве элемента коллекции (пусть это будет вектор). Во-вторых, далее при помощи алгоритма `find()` необходимо найти позиции слов "deep" и "submit" в полученной коллекции, произвести сортировку в интервале от "deep" до "submit". Наконец, в-третьих, записать полученную коллекцию в исходный файл. Поставленные задачи решает программа из листинга II.6.34.

Листинг II.6.34. Сортировка содержимого файла

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

bool less_str(const string &fst, const string &snd)
{
  return fst.length() < snd.length();
}
```

```
int main()
{
  const int LENGTH = 80;
  vector<string> coll;
  vector<string>::iterator it_deep, it_submit, pos;
  const char *filename = "linux.words";
  char input[LENGTH];
  string str;
  FILE *fp;

  try
  {
    // Читаем содержимое файла в коллекцию
    fp = fopen(filename, "r");
    while(!feof(fp))
    {
      // Читаем строку из файла-источника
      fgets(input, LENGTH, fp);
      str = input;
      coll.push_back(str);
    }
    fclose(fp);

    // Получаем итератор, указывающий на элемент со значением deep
    it_deep = find(coll.begin(), coll.end(), "deep\n");
    // Получаем итератор, указывающий на элемент со значением submit
    it_submit = find(coll.begin(), coll.end(), "submit\n");
    // Переставляем элементы коллекции, начиная с текущего
    // и до последнего
    sort(it_deep, it_submit, less_str);

    // Перезаписываем файл новыми значениями
    fp = fopen(filename, "w");
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      fputs((*pos).c_str(), fp);
    }
    fclose(fp);
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Как видно из листинга II.6.34, строкам "deep" и "submit" в конец добавляется символ перевода строки `\n`, т. к. именно в таком виде функция `fgets()` читает строки из файла. Сортировка фрагмента коллекции от слова "deep" до "submit" производится при помощи алгоритма `sort()`, в качестве критерия в котором выступает функция `less_str()`. Функция `less_str()` сравнивает количество символов в сопоставляемых строках при помощи метода `length()` и возвращает результат сравнение в виде булевой величины.

В рассматриваемом случае заранее известно, что позиция слова "deep" предшествует позиции слова "submit", т. к. файл отсортирован по алфавиту. Однако в реальных файлах слова могут быть не отсортированы, и при осуществлении преобразований над частью коллекции важно выяснить, какой из итераторов является началом интервала, а какой его концом. Если функции, реализующей алгоритм, передать не корректный интервал, это может привести к непредсказуемым последствиям. Одно из простейших решений этой проблемы представлено в листинге II.6.35.

Листинг II.6.35. Определение позиций итераторов относительно друг друга

```
...
it_deep = find(coll.begin(), coll.end(), "deep\n");
it_submit = find(coll.begin(), it_deep, "submit\n");
if(it_deep != it_submit)
{
  // it_submit предшествует it_deep, следовательно, корректным будет
  // интервал it_submit - it_deep
}
else
{
  it_submit = find(it_deep, coll.end(),"submit\n");
  if(it_deep != it_submit)
  {
    // it_deep предшествует it_submit, следовательно, корректным будет
    // интервал it_deep - it_submit
  }
  else
  {
     // Оба итератора равны и находятся в позиции coll.end()
  }
}
...
```

II.6.16. Создание копии коллекции

Копирования коллекции производится при глобальной функции `copy()`, которая копирует элементы одного интервала в элементы другого. Функция принимает лишь три параметра: первые два задают начало и конец первого интервала, третий параметр задает начало второго интервала.

Замечание

Алгоритмы, использующие несколько интервалов, ориентируются на количество элементов, которые задает первый интервал. Об том, чтобы остальные интервалы имели достаточное количество элементов, должен позаботиться программист.

Программа в листинге II.6.36 ошибочна, т. к. коллекция `copy_coll` не содержит достаточное количество элементов. Для того чтобы копирование прошло успешно, коллекция `copy_coll` должна иметь не меньше элементов, чем `coll`.

Листинг II.6.36. Ошибочное использование функции `copy()`

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

int main()
{
  vector<int> coll;
  vector<int> copy_coll;

  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);

    copy(coll.begin(), coll.end(), // Источник
         copy_coll.begin());       // Приемник
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }
```

```
  return 0;
}
```

Решение данной проблемы состоит в том, чтобы зарезервировать в коллекции-приемнике `copy_coll` достаточное количество элементов или использовать итератор вставки.

Чтобы коллекция-приемник обладала достаточным количеством элементов, можно во время создания указать их количество в параметре конструктора (листинг II.6.37).

Замечание

Данный способ резервирования элементов подходит только для последовательных контейнеров (векторов, дек и списков).

Листинг II.6.37. Резервирования элементов в коллекции

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

int main()
{
  vector<int> coll;

  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);

    // Задаем количество элементов в
    // конструкторе контейнера
    vector<int> copy_coll(coll.size());

    copy(coll.begin(), coll.end(), // Источник
         copy_coll.begin());       // Приемник
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }
```

```
  return 0;
}
```

Как видно из листинга II.6.37, после того как коллекция `coll` заполнена элементами, определить их количество можно при помощи метода `size()`. Конструктору коллекции-приемника `copy_coll` передается количество элементов, которым должен обладать вектор сразу после создания.

Подход, рассмотренный в листинге II.6.37, не всегда приемлем, коллекция может быть частью класса, и задать количество элементов на этапе объявления не всегда возможно. В этом случае прибегают к динамическому изменению размера коллекции при помощи метода `resize()` (листинг II.6.38).

Замечание

Использование метода `resize()`, как и резервирование элементов в конструкторе, допустимо только для последовательных контейнеров (векторов, дек и списков). Метод `resize()` может и уменьшать размер коллекции, однако уменьшение игнорируется для контейнера "вектор".

Листинг II.6.38. Динамическое изменение размера коллекции при помощи `resize()`

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

int main()
{
  vector<int> coll;
  vector<int> copy_coll;

  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);

    // Динамически изменяем количество
    // элементов в коллекции
    copy_coll.resize(coll.size());
```

```
    copy(coll.begin(), coll.end(), // Источник
         copy_coll.begin());       // Приемник
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Конструктор последовательных коллекций и метода `resize()` позволяют задать второй параметр, который инициализирует элементы коллекции. В листинге II.6.39 создается вектор целочисленных значений из 7 элементов, каждый из которых инициализирован значением –1, после чего при помощи метода `resize()` размер коллекции увеличивается до 10 элементов, при этом новые элементы инициализируются значением 1.

Листинг II.6.39. Инициализация элементов коллекции при расширении

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

int main()
{

  try
  {
    // Вектор из семи элементов, инициализированных
    // значением -1
    vector<int> coll (7, -1);
    vector<int>::iterator pos;

    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << " ";
    }
    cout << "\n";

    // Увеличиваем размер коллекции до 10 элементов,
    // инициализируя новые элементы значением 1
    coll.resize(10, 1);
```

```
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << " ";
    }
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Программа из листинга II.6.39 при выполнении выводит следующий результат:

```
-1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 1 1 1
```

Как уже упоминалось ранее, помимо обычных итераторов библиотека STL предоставляет итераторные адаптеры, которые кроме обычных свойств итераторов обладают рядом новых возможностей. Одним из таких итераторных адаптеров является итератор вставки, который позволяет использовать алгоритмы в режиме вставки (вместо режима перезаписи).

Различают три вида итераторов вставки:

- `back_inserter` — новые элементы присоединяются в конец коллекции при помощи метода `push_back()`. Поддерживается векторами, деками и списками;
- `front_inserter` — новые элементы присоединяются в начало коллекции при помощи метода `push_front()`. Поддерживается деками и списками;
- `intserter` — новые элементы вставляются в заданную позицию коллекции при помощи метода `insert()`. Поддерживается всеми типами коллекций.

В листинге II.6.40 демонстрируется использование всех трех видов итераторов для копирования элементов коллекции `coll` в коллекции вектора, дека и списка.

Листинг II.6.40. Использование итераторов вставки

```
#include <iostream>
#include <vector>
#include <deque>
```

```
#include <list>
#include <algorithm>
using namespace std;

int main()
{

  try
  {
    // Вектор из семи элементов, инициализированных
    // значением -1
    vector<int> coll (7, -1);
    vector<int>::iterator pos;

    vector<int> vector_coll;
    copy(coll.begin(), coll.end(),    // Источник
         back_inserter(vector_coll)); // Приемник

    deque<int> deque_coll;
    copy(coll.begin(), coll.end(),    // Источник
         front_inserter(deque_coll)); // Приемник

    list<int> list_coll;
    copy(coll.begin(), coll.end(),                // Источник
         inserter(list_coll, list_coll.begin())); // Приемник
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Следует отметить, что для использования того или иного итератора вставки необходимо, чтобы метод поддерживал соответствующий метод вставки. Например, совместно с коллекцией типа "вектор" не получится использовать итератор `front_inserter`, т. к. коллекция не поддерживает метод `push_front()`.

II.6.17. Удаление элементов коллекции

Проблема программы из листинга I.6.4 заключается в том, что метод `remove()` не изменяет количество элементов в коллекции. Он удаляет элементы со значением 3, сдвигая все последующие элементы на освободившиеся места.

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

В результате последние два элемента, не перезаписанные алгоритмом `remove()`, остались без изменения. На логическом уровне эти элементы уже не принадлежат коллекции. Механизм перераспределения элементов коллекции демонстрируется на рис. II.6.3.

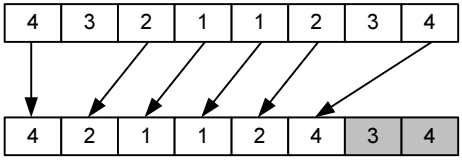


Рис. II.6.3. Перераспределение элементов коллекции при использовании алгоритма `remove()`

Определить новое окончание коллекции можно при помощи итератора, который возвращает метод `remove()`. Это позволяет получить актуальный интервал (листинг II.6.41).

Листинг II.6.41. Получение актуального интервала коллекции

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

int main()
{
  deque<int> coll;
  deque<int>::const_iterator pos;

  try
  {
    // Заполняем коллекцию
    for(int i = 1; i < 5; i++)
    {
      coll.push_front(i);
      coll.push_back(i);
    }

    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != coll.end(); ++pos)
```

```
    cout << *pos << ' ';
    cout << "\n";

    // Удаляем все элементы со значением 3
    deque<int>::iterator end = remove(coll.begin(), coll.end(), 3);

    // Выводим содержимое коллекции
    for(pos = coll.begin(); pos != end; ++pos) cout << *pos << ' ';
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Программа из листинга II.41 выведет следующую последовательность строк:

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

Для того чтобы узнать количество удаленных элементов, можно воспользоваться алгоритмом `distance()`, который позволяет определить количество элементов между двумя итераторами, в нашем случае между логическим и фактическим концом коллекции (листинг II.6.42).

Листинг II.6.42. Вычисление количества удаленных элементов

```
...
// Вычисление количества удаленных элементов
cout << distance(end, coll.end()) << "\n";
...
```

Для того чтобы элементы были фактически удалены из контейнера, необходимо воспользоваться методом `erase()` (листинг II.6.43).

Листинг II.6.43. Удаление элементов коллекции методом `erase()`

```
...
// Удаление "удаленных" элементов коллекции
coll.erase(end, coll.end());
...
```

Реализация метода `remove()` таким образом, что он фактически не изменяет размер коллекции, связана с тем, что итератор не может вызывать методы своего контейнера. Подобная архитектура позволяет работать с абстрактными интервалами, а не со всей коллекцией в целом. Операция фактического удаления элементов, особенно для векторов и дек, достаточно дорогостоящая и зачастую выгоднее не удалять элементы, а использовать новый логический конец коллекции.

II.6.18. Вывод содержимого произвольной коллекции

Для создания своей собственной функции, которая бы принимала ссылку на произвольную коллекцию, следует воспользоваться шаблонами. Первая функция `print_collection()` может выглядеть так, как это представлено в листинге II.6.44.

Листинг II.6.44. Реализация функции `print_collection()`

```
template <class T>
void print_collection(const T& coll)
{
  typename T::const_iterator pos;
  for(pos = coll.begin(); pos != coll.end(); ++pos)
  {
    cout << *pos << ' ';
  }
  cout << "\n";
}
```

Шаблон `T` позволяет передать функции `print_collection()` коллекцию любого типа (вектор, дек, список, множество, мультимножество, отображение или мультиотображение). Для вывода коллекции будет создан итератор `pos` соответствующего типа. Благодаря тому, что порядок работы с итератором любой коллекции один и тот же, код функции одинаково подходит для коллекций всех типов.

В листинге II.6.45 демонстрируется возможная реализация функции `print_back_collection()`, в которой для вывода содержимого коллекции используется цикл от конечного до начального элемента.

Листинг II.6.45. Реализация функции `print_back_collection()`

```
template <class T>
void print_back_collection(const T& coll)
```

```
{
  typename T::const_iterator pos, temp;
  temp = --coll.begin();
  for(pos = --coll.end(); pos != temp; --pos)
  {
    cout << *pos << ' ';
  }
  cout << "\n";
}
```

Для реализации функции `print_back_collection()` можно воспользоваться и обратным итератором. Здесь удобнее создать вспомогательную функцию `print()`, которая будет выводить элементы коллекции (листинг II.6.46).

Замечание

К сожалению, функцию `print()` не удастся реализовать в виде шаблона, и она будет жестко привязываться к типу элемента контейнера.

Листинг II.6.46. Альтернативный вариант реализации `print_back_collection()`

```
void print(int element)
{
  cout << element << ' ';
}

template <class T>
void print_back_collection(const T& coll)
{
  for_each(coll.rbegin(), coll.rend(), print);
  cout << "\n";
}
```

В листинге II.6.46 используется алгоритм `for_each()`, который позволяет применить функцию, определяемую третьим параметром, к интервалу, определяемому первыми двумя параметрами.

Последняя функция `print_interval()` может быть реализована одним из способов, представленных в листингах II.6.47 и II.6.48.

Листинг II.6.47. Реализация функции `print_interval()`

```
template <class T>
void print_interval(T begin, T end)
```

```
{
  T pos;
  for(pos = begin; pos != end; ++pos)
  {
    cout << *pos << ' ';
  }
}
```

Листинг II.6.48. Альтернативная реализация функции `print_interval()`

```
template <class T>
void print_interval(T begin, T end)
{
  for_each(begin, end, print);
  cout << "\n";
}
```

То есть, как и в случае функции `print_back_collection()`, можно осуществлять обход интервала в цикле или применять по отношению к каждому элементу коллекции функцию `print()` при помощи алгоритма `for_each()`.

В листинге II.6.49 представлена программа, которая демонстрирует использование функций `print_collection()`, `print_back_collection()` и `print_interval()`.

Листинг II.6.49. Использование функций вывода коллекций

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

int main()
{
  list<int> coll;

  try
  {
    // Заполняем коллекцию
    for(int i = 0; i < 10; i++)
    {
      coll.push_back(i);
    }
```

```
    print_collection(coll);
    print_back_collection(coll);
    print_interval(++coll.begin(), --coll.end());
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Результатом выполнения программы из листинга II.6.49 будут следующие строки:

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

II.6.19. Преобразование коллекции при копировании

Преобразование коллекции `coll` в коллекцию `coll_list` удобно осуществить при помощи алгоритма `transform()`, который так же, как и функция `for_each()`, принимает в качестве последнего параметра функцию преобразования элемента коллекции (листинг II.6.50).

Листинг II.6.50. Преобразование коллекции элементов в коллекцию квадратов элементов

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

int square(int value)
{
  return value*value;
}

void print(int element)
{
  cout << element << ' ';
}
```

```
int main()
{
  vector<int> coll;
  list<int> coll_list;

  try
  {
    coll.push_back(1);
    coll.push_back(2);
    coll.push_back(3);
    coll.push_back(4);
    coll.push_back(5);
    coll.push_back(6);

    // Выводим элементы коллекции
    for_each(coll.begin(), coll.end(), print);
    cout << "\n";

    // Преобразуем коллекцию coll в коллекцию
    // coll_list, возводя в квадрат элементы коллекции
    transform(coll.begin(), coll.end(), // Источник
              back_inserter(coll_list), // Приемник
              square);                  // Операция

    // Выводим элементы коллекции
    for_each(coll_list.begin(), coll_list.end(), print);
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Программа из листинга II.6.50 преобразует коллекцию `coll` в коллекцию `coll_list` при помощи алгоритма `transform()`, который использует функцию обратного вызова `square()`, подвергающую элементы коллекции возведению в квадрат.

Следует отметить, что для вставки элементов в новую коллекцию используется итератор вставки `back_inserter`, в противном случае копирование в пустую коллекцию привело бы к ошибке.

II.6.20. Что такое предикат?

Предикатом называется функция, которая возвращает логическое значение. С помощью предикатов часто определяют критерий сортировки и поиска. Различают унарные предикаты и бинарные.

Классическим примером унарного предиката является функция `isPrimeNumber()`, которая проверяет, является ли переданный ей параметр простым числом (листинг II.6.51).

Листинг II.6.51. Функция, проверяющая, является ли число простым

```
bool isPrimeNumber(int number)
{
  // Получаем абсолютное значение
  number = abs(number);

  // 0 и 1 не относятся к простым числам
  if(number == 0 || number == 1) return false;

  // Ищем множитель, на который число делится
  // без остатка
  int multiplier = 1;
  for(multiplier = number/2; number % multiplier != 0; --multiplier);

  // Если единственным множителем является 1,
  // то число простое
  return multiplier == 1;
}
```

Функция `isPrimeNumber()` принимает в качестве единственного параметра целое число и проверяет, делится ли оно на какой-либо множитель, отличный от 1. Если нет, то число является простым.

В листинге II.6.52 для поиска первого простого числа в интервале от 75 до 100 используется алгоритм `find_if()`, который принимает в качестве первых двух параметров начало и конец интервала, а в качестве последнего параметра — имя предиката.

Листинг II.6.52. Поиск первого простого числа в интервале от 75 до 100

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

```
int main()
{
  list<int> coll;

  try
  {
    // Ищем первое просто число в интервале
    // от 75 до 100
    for(int i = 75; i < 100; i++)
      coll.push_back(i);

    list<int>::iterator pos;
    pos = find_if(coll.begin(), coll.end(), isPrimeNumber);

    if(pos == coll.end()) cout << "Простое число не найдено\n";
    else cout << "Простое число " << *pos << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Результатом выполнения программы является строка:

```
Простое число 79
```

Бинарные предикаты сравнивают свойство двух аргументов и, как правило, используются для сортировки элементов коллекции по нестандартному критерию. Пусть имеется структура `person`, состоящая из двух элементов — `firstname` и `secondname`, которые определяют имя и фамилию человека соответственно (листинг II.6.53).

Листинг II.6.53. Структура `person`

```
struct person
{
  person(string first, string last)
  {
    firstname = first;
    lastname = last;
  }
```

```
  string firstname;
  string lastname;
};
```

В качестве бинарного предиката может выступать функция `person_sort()`, реализация которой представлена в листинге II.6.54.

Листинг II.6.54. Бинарный предикат `person_sort()`

```
bool person_sort(const person &fst, const person &snd)
{
  if(fst.lastname < snd.lastname) return true;
  else if(fst.lastname > snd.lastname) return false;
  else if(fst.firstname < snd.firstname) return true;
  else return false;
}
```

Функция `person_sort()` сравнивает два объекта типа `person`. Первый объект `fst` считается меньше второго объекта `snd`, если строка с фамилией `lastname` первого объекта меньше строки с фамилией второго объекта (по лексикографическому признаку). Если фамилии оказываются равными, то сравнению подвергаются имена `firstname`.

В листинге II.6.55 приводится пример программы, в которой объявляется вектор, состоящий из четырех элементов `person`. Эти элементы сортируются по критерию `persont_sort()` и выводятся в стандартный поток при помощи функции `print()`.

Листинг II.6.55. Использование предиката `person_sort()`

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

void print(person &psn)
{
  cout << psn.firstname << " " << psn.lastname << "\n";
}

int main()
{
  vector<person> coll;
```

```
  try
  {
    coll.push_back(person("Игорь", "Симдянов"));
    coll.push_back(person("Максим", "Кузнецов"));
    coll.push_back(person("Сергей", "Гольшев"));
    coll.push_back(person("Сергей", "Осипов"));

    // Сортируем коллекцию
    sort(coll.begin(), coll.end(), person_sort);

    // Выводим элементы коллекции
    for_each(coll.begin(), coll.end(), print);
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

II.6.21. Что такое объект-функция?

Ряд алгоритмов, таких как `for_each()` или `sort()`, допускают передачу в качестве одного из параметров функцию, которая задает ту или иную операцию. Например, в листинге II.6.55 в качестве такой функции выступает `print()`, передача алгоритму `for_each()` которой позволяет вывести в стандартный поток все элементы коллекции. Вместо функции можно использовать объект класса, который ведет себя как функция, т. е. перегружает круглые скобки. Такие объекты называют *объектами-функциями* или *функторами*.

Создадим объект-функцию `PrintElement` для вывода элемента в стандартный поток (листинг II.6.56). Класс `PrintElement` может заменить собой функцию `print()` из листинга II.6.55.

Листинг II.6.56. Объект-функция `PrintElement`

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

class PrintElement
```

```
{
  public:
    void operator() (int elem) const
    {
      cout << elem << ' ';
    }
};

int main()
{
  vector<int> coll;

  try
  {
    for(int i = 0; i < 10; i++)
    {
      coll.push_back(i);
    }

    // Выводим элементы коллекции
    for_each(coll.begin(), coll.end(), // Интервал
             PrintElement());          // Операция
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Класс `PrintElement` определяет объект, для которого оператор "круглые скобки" `()` перегружен таким образом, чтобы выводить целочисленный аргумент. Класс `PrintElement` явно зависит от типа элемента коллекции. Несмотря на то, что можно применить шаблоны, при вызове алгоритма потребуется явное определение типа элемента коллекции (листинг II.6.57).

Листинг II.6.57. Использование шаблонного класса в качестве объекта-функции

```
...
template <class X>
class PrintElement
```

```
{
  public:
    void operator() (X elem) const
    {
      cout << elem << ' ';
    }
};
...
// Выводим элементы коллекции
for_each(coll.begin(), coll.end(), // Интервал
           PrintElement<int>());   // Операция
...
```

Объект-функция часто более удобен, чем просто функция, т. к. обладает состоянием и позволяет передавать аргументы. Пусть стоит задача прибавить каждому элементу коллекции `coll` из листинга II.6.56 единицу. Можно решить задачу при помощи обычной функции (листинг II.6.58).

Листинг II.6.58. Увеличение каждого элемента коллекции на единицу

```
...
void add(int &element)
{
  element += 1;
}
...
// Выводим элементы коллекции
for_each(coll.begin(), coll.end(), // Интервал
          add);                    // Операция
...
```

Однако если потребуется изменить значение 1 на 5, то потребуется создать новую функцию, т. к. передать дополнительный параметр не удастся. Ситуацию можно решить, используя шаблонную функцию (листинг II.6.59).

Листинг II.6.59. Использование шаблонной функции

```
...
template <int value>
void add(int &element)
{
  element +=value;
}
```

```
...
// Выводим элементы коллекции
for_each(coll.begin(), coll.end(), // Интервал
         add<10>);                 // Операция
...
```

Теперь можно использовать функцию `add()` для прибавления элементам коллекции произвольного числа. Однако число жестко прописывается в коде, изменять его динамически, например, получая его извне во время выполнения программы, нельзя. Решением служит создание объекта функции, который сможет принять переменную через параметр конструктора (листинг II.6.60).

Листинг II.6.60. Использование объекта-функции

```
class add
{
  private:
    int value;
  public:
    // Конструктор
    add(int val)
    {
      value = val;
    }
    // Оператор ()
    void operator() (int &element) const
    {
      element += value;
    }
};
...
int val = 10;
// Выводим элементы коллекции
for_each(coll.begin(), coll.end(), // Интервал
         add(val));                // Операция
...
```

При использовании объекта-функции `add()` появляется возможность динамически изменять значение приращения.

Помимо возможности создавать собственные объекты-функции, библиотека STL предоставляет стандартные объекты-функции для наиболее типичных случаев. К таким стандартным объектам-функциям относятся критерии сор-

тировки. По умолчанию элементы ассоциативных контейнеров сортируются с критерием сортировки `less<>` (меньше), поэтому объявления, представленные в листинге II.6.61, эквивалентны.

Листинг II.6.61. Использование критерия сортировки `less<>`

```
...
set<int> coll;
set<int, less<int> > coll;
...
```

Для того чтобы элементы в коллекции `coll` сортировались в обратном порядке, достаточно применить критерий сортировки `greater<>` (листинг II.6.62).

Листинг II.6.62. Использование критерия сортировки `greater<>`

```
...
set<int, greater<int> > coll;
...
```

Стандартный объект-функция `negate<>` позволяет изменить знак элемента на противоположный. Этот объект-функцию часто применяют совместно с алгоритмом `transform()`, который преобразует одну коллекцию в другую, обрабатывая каждый элемент заданной функцией. В листинге II.6.63 содержимое коллекции `coll` копируется в коллекцию `coll_copy` с изменением знака каждого из элементов.

Листинг II.6.63. Совместное использование объекта функции `negate<>` и алгоритма `transform()`

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

template <class X>
class PrintElement
{
  public:
    void operator() (X elem) const
    {
      cout << elem << ' ';
    }
};
```

```
int main()
{
  list<int> coll, coll_transform;

  try
  {
    // Заполняем элементы коллекции
    for(int i = 1; i < 5; i++)
    {
      coll.push_back(i);
      coll.push_front(-i);
    }

    // Выводим элементы коллекции
    for_each(coll.begin(), coll.end(),
             PrintElement<int>());
    cout << "\n";

    // Изменяем знак элементов коллекции
    transform(coll.begin(), coll.end(),
              back_inserter(coll_transform),
              negate<int>());

    // Выводим элементы коллекции
    for_each(coll_transform.begin(), coll_transform.end(),
             PrintElement<int>());
    cout << "\n";
  }
  catch(bad_alloc)
  {
    cout << "Не удалось выделить память под коллекцию\n";
  }

  return 0;
}
```

Результат выполнения программы из листинга II.6.63 выглядит следующим образом:

```
-4 -3 -2 -1 1 2 3 4
4 3 2 1 -1 -2 -3 -4
```

При использовании алгоритма `transform()` коллекция-источник и коллекция-приемник могут совпадать, в этом случае преобразование будет выполнено

над элементами коллекции. Использовать итератор вставки в этом случае не нужно (листинг II.6.64).

Листинг II.6.64. Преобразование элементов коллекции `coll`

```
...
// Изменяем знак элементов коллекции
transform(coll.begin(), coll.end(),
          coll.begin(),
          negate<int>());
...
```

Функция `transform()` может работать и с тремя коллекциями, объединяя первые две коллекции в третью при помощи стандартных или пользовательских объектов-функций. В этом случае объект-функция должна принимать два параметра.

В табл. II.6.7 приводится список стандартных объектов-функций, доступных в библиотеке STL. В описании схематично объясняется операция, выполняемая объектом-функцией. Ряд функций, таких как `negate<>`, `logical_not<>`, поддерживают только один параметр, остальные — два и применяются либо для объединения двух коллекций в одну, либо с функциональными адаптерами.

Таблица II.6.7. *Стандартные объекты функции*

Объект-функция	Пример	Описание
`negate<>`	`-prm`	Меняет знак на противоположный
`plus<>`	`prm1 + prm2`	Складывает значения
`minus<>`	`prm1 - prm2`	Вычитает значения
`multiplies<>`	`prm1 * prm2`	Умножает значения
`divides<>`	`prm1 / prm2`	Делит значения
`modulus<>`	`prm1 % prm2`	Извлекает остаток от деления
`equal_to<>`	`prm1 == prm2`	Проверяет элементы на равенство
`not_equal_to<>`	`prm1 != prm2`	Проверяет элементы на неравенство
`less<>`	`prm1 < prm2`	Критерий сортировки "меньше"
`greater<>`	`prm1 > prm2`	Критерий сортировки "больше"
`less_equal<>`	`prm1 <= prm2`	Критерий сортировки "меньше либо равно"

Таблица II.6.7 *(окончание)*

Объект-функция	Пример	Описание
greater_equal<>	prm1 >= prm2	Критерий сортировки "больше либо равно"
logical_not<>	!prm	Возвращает противоположенное логическое значение
logical_and<>	prm1 && prm2	Возвращает логическое И
logical_or<>	prm1 || prm2	Возвращает логическое ИЛИ

В листинге II.6.65 приводится пример, в котором элементы коллекции `coll` заполняются целочисленными элементами от 0 до 9, после чего при помощи алгоритма `transform()` и объекта-функции `multiplies<>` производится возведение элементов коллекции в квадрат.

Листинг II.6.65. Возведение элементов коллекции в квадрат

```
...
 vector<int> coll;

 // Заполняем элементы коллекции
 for(int i = 0; i < 10; i++)
 {
   coll.push_back(i);
 }

 // Выводим элементы коллекции
 for_each(coll.begin(), coll.end(),
          PrintElement<int>());
 cout << "\n";

 // Возводим элементы коллекции в квадрат
 transform(coll.begin(), coll.end(), // Первый источник
           coll.begin(),             // Второй источник
           coll.begin(),             // Коллекция-приемник
           multiplies<int>());       // Операция

 // Выводим элементы коллекции
 for_each(coll.begin(), coll.end(),
          PrintElement<int>());
 cout << "\n";
...
```

Программа из листинга II.6.65 возвращает следующий результат:

```
0 1 2 3 4 5 6 7 8 9
0 1 4 9 16 15 36 49 64 81
```

Однако не всегда удобно оперировать двумя коллекциями, сохраняя результат в третью. Глупо для того, чтобы прибавить к 100 элементам коллекции значение 5, создавать еще одну коллекцию из 100 элементов со значением 5. Здесь на помощь приходят функциональные адаптеры. Функциональные адаптеры позволяют комбинировать объекты-функции друг с другом, с переменными и специальными функциями. В листинге II.6.66 функциональный адаптер `bind2nd()` для того, чтобы прибавить к каждому элементу коллекции значение 5.

Листинг II.6.66. Прибавление значения 5 к каждому элементу коллекции

```
...
// Заполняем элементы коллекции
for(int i = 0; i < 10; i++)
{
  coll.push_back(i);
}

// Выводим элементы коллекции
for_each(coll.begin(), coll.end(),
         PrintElement<int>());
cout << "\n";

// Возводим элементы коллекции в квадрат
transform(coll.begin(), coll.end(), // Первый источник
          coll.begin(),             // Коллекция-приемник
          bind2nd(plus<int>(),5));  // Операция

// Выводим элементы коллекции
for_each(coll.begin(), coll.end(),
         PrintElement<int>());
cout << "\n";
...
```

Программа из листинга II.6.66 возвращает в качестве результата следующие строки:

```
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9 10 11 12 13 14
```

II.6.22. В чем особенность контейнера *vector<bool>*?

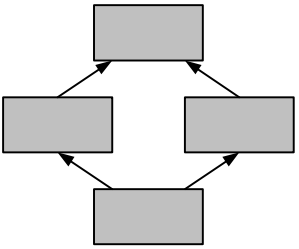
Контейнер `vector<bool>` реализуется отдельно от других типов векторов. Это осуществлено с целью повышения эффективности данного контейнера. Так как каждое значение может принимать только два значения: `true` и `false`, под каждый элемент отводится только один бит. Это позволяет в восемь раз сократить занимаемый контейнером объем, однако приводит к некоторым ограничениям. В частности, итератор `vector<bool>::iterator` не является итератором произвольного доступа, кроме того, скорость работы контейнера может быть существенно ниже по сравнению с обычным контейнером, например, `vector<int>`, т. к. приходится преобразовывать обычные операции в операции с битами.

Контейнер `vector<bool>` имеет ряд специальных методов по сравнению с контейнером "вектор" для других типов. Дополнительные методы перечислены в табл. II.6.8, в которой `coll` обозначает коллекцию `vector<bool>`.

Таблица II.6.8. *Специальные методы контейнера* `vector<bool>`

Операция	Описание
`coll.flip()`	Метод `flip()` инвертирует все элементы коллекции (0 заменяется на 1, и наоборот)
`coll[i].flip()`	Если указывается индекс элемента `i`, конвертации подвергается только этот элемент
`coll[i] = value`	Присвоение логического значения `value` элементу с индексом `i`
`coll[i] = coll[j]`	Присвоение значения элемента с индексом `j` элементу с индексом `i`

ГЛАВА II.7



Ввод/вывод

II.7.1. Что такое поток?

Каждая компьютерная система обладает собственными устройствами ввода/вывода. Это может быть принтер или телетайп, монитор компьютера или карманного устройства, файл на диске или другом устройстве долговременного хранения информации. Синтаксис языка не должен зависеть от конкретной реализации той или иной компьютерной системы. Поэтому программа на языке C/C++ работает со специальным интерфейсом, который называется *потоком*. Любое конечное устройство, будь то принтер, монитор или файл на жестком диске, называют *файлом*. Такая абстракция позволяет использовать одни и те же функции для работы со всем многообразием устройств.

Открыть поток в рамках языка C можно при помощи функции `fopen()`, которая принимает в качестве первого аргумента имя файла, а в качестве второго — режим его открытия. В листинге II.7.1 открывается файл text.txt, в который записывается фраза "Hello, world".

Листинг II.7.1. Запись в файл

```
#include <stdio>
#include <string>
using namespace std;

int main()
{
  FILE *fp;

  // Открываем файл
  fp = fopen("text.txt", "w");
```

```
  // Записываем в него информацию
  char *text = "Hello, world";
  fwrite(text, strlen(text), 1, fp);
  // Закрываем файл
  fclose(fp);

  return 0;
}
```

Помимо пользовательских потоков, каждая программа в начале своей работы открывает стандартные потоки:

- `stdin` — стандартный поток ввода (по умолчанию связан с клавиатурой);
- `stdout` — стандартный поток вывода (по умолчанию связан с консолью);
- `stderr` — стандартный поток ошибок (по умолчанию связан с консолью).

Стандартные потоки открываются и закрываются автоматически, поэтому их не нужно открывать или закрывать. В листинге II.7.2 приводится пример вывода строки "Hello, world" в стандартный поток `stdout` (по умолчанию данный поток связан с консолью).

Замечание

Большинство операционных сред позволяют перенаправлять стандартные потоки в файл (для `stdout` и `stderr`) или из файла (для `stdin`), для этого используются команды `>` и `<` (для перезаписи файла), а также `>>` и `<<` для дозаписи информации в конец файла. Например, если имя исполняемого файла из листинга II.7.2 является test, то переправить поток `stdout` в файл text.txt можно при помощи команды `test > text.txt`.

Листинг II.7.2. Вывод в стандартный поток `stdout`

```
#include <stdio>
#include <string>
using namespace std;

int main()
{
  FILE *fp;

  char *text = "Hello, world";
  fwrite(text, strlen(text), 1, stdout);

  return 0;
}
```

По умолчанию стандартные потоки связаны с монитором, однако при помощи функции `freopen()` можно перенаправить стандартные (или пользовательские) потоки в файл. В листинге II.7.3 стандартный поток `stdout` перенаправляется в файл text.txt.

Листинг II.7.3. Перенаправление стандартного потока в файл text.txt

```
#include <stdio>
#include <string>
using namespace std;

int main()
{
  FILE *fp;

  freopen("text.txt", "w", stdout);
  char *text = "Hello, world";
  fwrite(text, strlen(text), 1, stdout);

  return 0;
}
```

Язык C++ предоставляет альтернативную языку C систему ввода/вывода, которая основывается на объектно-ориентированном интерфейсе.

Замечание

Язык C++ допускает использование системы ввода/вывода в стиле C, однако в отличие от C, средства ввода/вывода не являются глобальными, а расположены в пространстве имен `std`.

В отличие от C, язык C++ автоматически открывает четыре стандартных потока, три из которых соответствуют потокам C, а четвертый является разновидностью стандартного потока ошибок:

- `cin` — стандартный поток ввода (по умолчанию связан с клавиатурой);
- `cout` — стандартный поток вывода (по умолчанию связан с консолью);
- `cerr` — стандартный поток вывода ошибок (по умолчанию связан с консолью);
- `clog` — буферизованный вариант потока `cerr`.

В листинге II.7.4 приводится пример вывода строки "Hello, world" в стандартный поток `cout`.

Замечание

Для того чтобы воспользоваться средствами ввода/вывода языка C++ в программу, необходимо включить заголовочный файл <iostream>.

Листинг II.7.4. Вывод в стандартный поток в стиле C++

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

int main()
{
  char *text = "Hello, world\n";
  cout << text;

  return 0;
}
```

Вывод информации осуществляется при помощи перегруженного оператора <<, ввод — при помощи перегруженного оператора >>.

II.7.2. Выравнивание строк по правому краю

Решение задачи можно разбить на несколько этапов: получение строк от пользователя и их сортировка (тут удобно воспользоваться средствами библиотеки STL), определение количества символов в самом длинном слове и вывод всех строк в цикле при помощи форматирующей функции printf().

Прием строк от пользователей с их последующей сортировкой рассматривался в *разд. II.6.12* (см. листинг II.6.27).

В листинге II.7.5 приводится пример форматного вывода содержимого вектора строк при помощи функции printf().

Замечание

Для использования функции printf() в теле программы необходимо подключить библиотеку <stdio>.

Листинг II.7.5. Форматный вывод строк при помощи функции printf()

```
#include <vector>
#include <string>
#include <stdio>
using namespace std;
```

```
int main()
{
  vector<string> coll;
  vector<string>::const_iterator pos;

  try
  {
    coll.push_back("1");
    coll.push_back("22");
    coll.push_back("333");
    coll.push_back("4444");
    coll.push_back("55555");

    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      // Вместо метода size() можно использовать length()
      printf("str = %20s\n", *pos);
    }
  }
  catch(bad_alloc)
  {
    printf("Не удалось выделить память под коллекцию\n");
  }

  return 0;
}
```

Результатом выполнения программы из листинга II.7.5 являются следующие строки:

```
str =                    1
str =                   22
str =                  333
str =                 4444
str =                55555
```

Вывести содержимое контейнера `vector` в стандартный поток позволяет функция `printf()`, которая имеет следующий синтаксис:

```
int printf(const char format, ...)
```

Функция содержит один-единственный обязательный параметр *`format`* и произвольное количество необязательных параметров. Параметр *`format`* может состоять из обычных символов и спецификаторов, которые начинаются с символа `%`.

Обычные символы выводятся в стандартный поток без изменений, именно так выводится строка "Не удалось выделить память под коллекцию\n" в листинге II.7.5.

Спецификаторы позволяют подставить в выводимую строку необязательные аргументы функции printf(), осуществляя их форматирование. Количество спецификаторов и необязательных параметров, следующих за параметром format, должно совпадать. Строка "str = %20s\n" позволяет вывести содержимое строковой коллекции с выравниванием элементов по правому краю. Здесь фрагменты "str = " и "\n" выводятся в стандартный поток как есть, а спецификатор %20s заменяется строкой *pos.

Замечание

В C++ различают два вида строк: строки в стиле C — char * и строки стандартной библиотеки C++ — string. На протяжении почти всей книги использовались C-строки, в листинге II.7.5 используются string-строки. string-строки являются объектом и имеют множество удобных методов. Для того чтобы получить C-строку, следует обратиться к методу c_str(), поэтому строка вывода в листинге II.7.5 может выглядеть и следующим образом: printf("str = %20s\n", pos->c_str()).

В табл. II.7.1 представлены спецификаторы формата функции printf().

Таблица II.7.1. *Спецификаторы формата функции* printf()

Спецификатор	Описание
%c	Спецификатор символа, используется для подстановки в строку формата символов char, например, a, w, 0, \0
%d	Спецификатор десятичного целого числа со знаком, используется для подстановки целых чисел, например, 0, 100, –45
%i	Спецификатор десятичного целого числа со знаком
%e	Спецификатор числа в научной нотации, например, число 1200, в данной нотации записывается как 1.2e+03, а 0.01, как 1e-02
%E	Спецификатор числа в научной нотации, например, число 1200, в данной нотации записывается как 1.2E+03, а 0.01 — как 1E-02. От %e данный спецификатор отличается только регистром символа E
%f	Спецификатор десятичного числа с плавающей точкой, например, 156.001
%g	При использовании данного спецификатора десятичное число записывается либо в формате %e, либо в формате %f, в зависимости от того, в каком случае результат получится короче

Таблица II.7.1 *(окончание)*

Спецификатор	Описание
%G	При использовании данного спецификатора десятичное число записывается либо в формате %E, либо в формате %f, в зависимости от того, в каком случае результат получится короче
%o	Спецификатор используется для подстановки в строку формата восьмеричного числа без знака
%s	Спецификатор используется для подстановки в строку формата строки
%u	Спецификатор десятичного целого числа со знаком, используется для подстановки целых чисел без знака
%x	Спецификатор используется для подстановки в строку формата шестнадцатеричного числа без знака (строчные буквы для a, b, c, d, e, f)
%X	Спецификатор используется для подстановки в строку формата шестнадцатеричного числа без знака (прописные буквы для A, D, C, D, E, F)
%p	Спецификатор используется для подстановки в строку формата указателя
%n	Указатель на целочисленную переменную. Спецификатор вызывает присвоение этой целочисленной переменной количества символов, выведенных перед ним
%%	Двойная последовательность %% используется для обозначения одиночного символа % в строке вывода

В листинге II.7.6 приводится пример программы, демонстрирующей некоторые из спецификаторов типа.

Листинг II.7.6. Использование спецификаторов типа

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

int main()
{
  int number = 29;
  int *prt = &number;

  printf("Десятичное число: %d\n", number);
  printf("Восьмеричное число: %o\n", number);
```

```
  printf("Шестнадцатеричное число (нижний регистр): %x\n", number);
  printf("Шестнадцатеричное число (верхний регистр): %X\n", number);
  printf("Указатель: %p\n", prt);

  return 0;
}
```

Результатом работы программы из листинге II.7.6 являются следующие строки:

```
Десятичное число: 29
Восьмеричное число: 35
Шестнадцатеричное число (нижний регистр): 1d
Шестнадцатеричное число (верхний регистр): 1D
Указатель: 0012FF88
```

Спецификаторы из табл. II.7.1 часто называют *определителями типа*. Они сообщают функции `printf()`, как следует интерпретировать передаваемый аргумент. Между символом `%` и спецификатором типа может быть расположено число, которое называют *спецификатором минимальной ширины поля*. В листинге II.7.5 в строке формата `"%20s"` таким спецификатором служит число 20. Данный тип спецификатора определяет, сколько позиций будет отводиться под вывод числа или строки (если потребуется больше позиций, то подстрока автоматически расширяется, однако параметр при выводе не может занимать меньше позиций, чем указано в спецификаторе минимальной ширины поля). Перед спецификатором минимальной ширины поля можно указать символ заполнения ведущих позиций: либо 0, либо пробел (листинг II.7.7).

Листинг II.7.7. Использование спецификатора минимальной ширины поля

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

int main()
{
  printf("%4d\n",45);  // "   45"
  printf("% 4d\n",45); // "   45"
  printf("%04d\n",45); // "00045"

  return 0;
}
```

Через точку можно использовать спецификатор точности, его значение зависит от типа модифицируемого формата. Для строк спецификатор точности

задает максимальное количество выводимых символов, например, формат "%4.6s" означает, что будет выведено не менее 4 и не более 6 символов (лишние символы отбрасываются). В листинге II.7.8 приводится пример использования спецификатора точности.

Листинг II.7.8. Использование спецификатора точности со спецификатором %s

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

int main()
{
  printf("%4.6s\n","the");        // " the"
  printf("%4.6s\n","component"); // "compon"

  return 0;
}
```

Для форматов %e, %E и %f модификатор точности задает количество цифр после запятой (листинг II.7.9).

Листинг II.7.9. Использование спецификатора точности со спецификаторами %e, %E и %f

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

int main()
{
  double number = 123.456789;
  printf("%4.2f\n", number); // 123.46
  printf("%4.2e\n", number); // 1.23e+02

  return 0;
}
```

При использовании спецификатора точности совместно с форматами %g и %G можно задать количество значащих цифр.

По умолчанию все выводимые при помощи функции printf() данные выравниваются по правому краю. Именно на этом основывается эффект выравнивания по правому краю в листинге II.7.5. Однако можно выровнять данные по левому краю, для этого необходимо сразу после символа % поставить знак "минус", например, "%-20s".

Теперь можно приступить к формированию окончательного решения, однако в отличие от формата "%20s", используемого в листинге II.7.5, необходимо динамически изменять спецификатор минимальной ширины поля, т. к. вводимая строка может быть как больше, так и меньше 20 символов. Для решения этой задачи удобно воспользоваться функцией sprintf(), которая позволяет поместить форматируемые данные не в выходной поток, а в строку. Функция имеет следующий синтаксис:

```
int sprintf(char *str, const char *format, ...)
```

Функция выводит необязательные параметры, форматируя их согласно строке *format*, при этом результат помещается в строку *str*. В листинге II.7.10 приводится конечное решение задачи.

Листинг II.7.10. Конечное решение задачи выравнивая строк по правому краю

```
#include <set>
#include <string>
#include <stdio>
using namespace std;

int main()
{
  // Отображение
  multiset<string> coll;
  multiset<string>::const_iterator pos;

  try
  {
    char input[80];
    string str;
    while(true)
    {
      printf("Введите строку ");
      fgets(input, 80, stdin);
      // Если введена пустая строка — выходим из цикла
      if(strlen(input) <= 1) break;
      // Удаляем завершающий перевод строки
      input[strlen(input) — 1] = '\0';
      // Присваиваем строке str значение input
      str = input;
      coll.insert(str);
    }
```

```
    // Определяем длину максимальной строки
    int max_lenght = 0;
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      // Вместо метода size() можно использовать length()
      if(pos->size() > max_lenght) max_lenght = pos->size();
    }

    // Строка форматирования
    char format[80];
    sprintf(format, "%%%ds\n", max_lenght);
    // Выводим строки, выровненные по правому краю
    for(pos = coll.begin(); pos != coll.end(); ++pos)
    {
      printf(format, *pos);
    }
  }
  catch(bad_alloc)
  {
    printf("Не удалось выделить память под коллекцию\n");
  }

  return 0;
}
```

Как видно из листинга II.7.10, конечный формат формируется динамически в строке `format`, которая затем используется в функции `printf()` для вывода пользовательских строк.

II.7.3. Выравнивание строк по правому и левому краям

Для вывода строк в два столбца необходимо разбить коллекцию на две части для первого и второго столбца. В двух частях коллекции следует найти строки с максимальным количеством символов и подготовить две форматирующие строки: с выравниванием по левому краю для первого столбца и выравниванием по правому краю для правого столбца. В листинге II.7.11 представлена одна из возможных реализаций решения задачи.

Листинг II.7.11. Вывод коллекции строк в два столбца

```
#include <vector>
#include <string>
```

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

int main()
{
  vector<string> coll;
  vector<string>::const_iterator pos;

  try
  {
    char input[80];
    string str;
    while(true)
    {
      printf("Введите строку ");
      fgets(input, 80, stdin);
      // Если введена пустая строка — выходим из цикла
      if(strlen(input) <= 1) break;
      // Удаляем завершающий перевод строки
      input[strlen(input) — 1] = '\0';
      // Присваиваем строке str значение input
      str = input;
      coll.push_back(str);
    }
    // Если количество элементов не четное,
    // добавляем дополнительный пустой элемент
    if(coll.size() % 2) coll.push_back("");

    // Определяем количество элементов в каждом из столбцов
    int half_count = coll.size() / 2;

    // Определяем длину максимальной строки
    // в первом столбце
    int max_lenght_first = 0;
    for(int i = 0; i < half_count; i++)
    {
      if(coll[i].size() > max_lenght_first)
        max_lenght_first = coll[i].size();
    }

    // Определяем длину максимальной строки
    // во втором столбце
    int max_lenght_second = 0;
```

```
    for(int i = half_count; i < coll.size(); i++)
    {
      if(coll[i].size() > max_lenght_second)
        max_lenght_second = coll[i].size();
    }

    // Строки форматирования для первого и второго столбцов
    char format_first[80], format_second[80];
    sprintf(format_first, "%%-%ds", max_lenght_first + 2);
    sprintf(format_second, "%%%ds\n", max_lenght_second);

    for(int i = 0; i < coll.size(); i++)
    {
      if(i % 2) printf(format_second, coll[i]);
      else printf(format_first, coll[i]);
    }
  }
  catch(bad_alloc)
  {
    printf("Не удалось выделить память под коллекцию\n");
  }

  return 0;
}
```

Представленное в листинге II.7.11 решение выводит элементы коллекции `coll` по схеме, изображенной на рис. II.7.1.

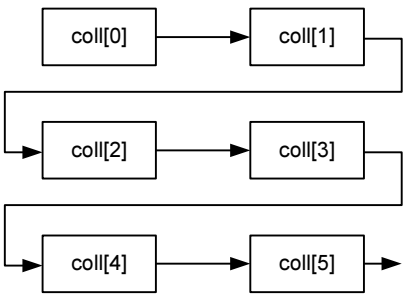


Рис. II.7.1. Вывод элементов построчно

Однако схема, представленная на рис. II.7.1, не является единственно возможной, вместо построчного вывода можно выводить элементы коллекции в столбцах (рис. II.7.2). Такой вариант более трудоемок, но зачастую более нагляден.

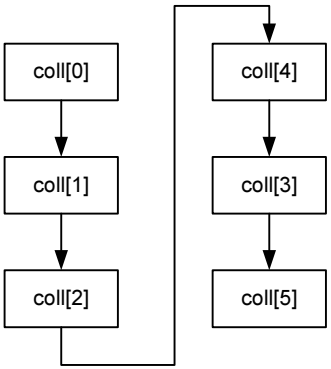


Рис. II.7.2. Вывод элементов в столбцах

В листинге II.7.12 представлена программа, которая выводит элементы коллекции именно в таком формате.

Листинг II.7.12. Альтернативный вывод коллекции в два столбца

```
#include <vector>
#include <string>
#include <stdio>
using namespace std;

int main()
{
  vector<string> coll;
  vector<string>::const_iterator pos;

  try
  {
    char input[80];
    string str;
    while(true)
    {
      printf("Введите строку ");
      fgets(input, 80, stdin);
      // Если введена пустая строка — выходим из цикла
      if(strlen(input) <= 1) break;
      // Удаляем завершающий перевод строки
      input[strlen(input) — 1] = '\0';
      // Присваиваем строке str значение input
      str = input;
```

```
      coll.push_back(str);
    }
    // Если количество элементов не четное,
    // добавляем дополнительный пустой элемент
    if(coll.size() % 2) coll.push_back("");

    // Определяем количество элементов в каждом из столбцов
    int half_count = coll.size() / 2;

    // Определяем длину максимальной строки
    // в первом и втором столбцах
    int max_lenght_first = 0;
    int max_lenght_second = 0;
    for(int i = 0; i < half_count; i++)
    {
      if(coll[i].size() > max_lenght_first)
        max_lenght_first = coll[i].size();
      if(coll[i + half_count].size() > max_lenght_second)
        max_lenght_second = coll[i + half_count].size();
    }

    // Строки форматирования для первого и второго столбцов
    char format_first[80], format_second[80];
    sprintf(format_first, "%%-%ds", max_lenght_first + 2);
    sprintf(format_second, "%%%ds\n", max_lenght_second);

    for(int i = 0; i < half_count; i++)
    {
      printf(format_first, coll[i]);
      printf(format_second, coll[i + half_count]);
    }
  }
  catch(bad_alloc)
  {
    printf("Не удалось выделить память под коллекцию\n");
  }

  return 0;
}
```

II.7.4. Ввод строк пользователем

Для С-строк в языке C++ не отслеживается выход за границу строки, т. е. выделив память под 80 символов, можно перезаписать память, следующую непосредственно за границей строки. На этом основана атака переполнения

буфера, когда в соседние области памяти записываются управляющие команды, изменяющие логику работы приложения. В языке C для предотвращения выхода за границу строки используется функция `fgets()`, позволяющая задать количество символов, которые следует прочитать из стандартного потока ввода. Функция `fgets()` имеет следующий синтаксис:

```
char *fgets(char *str, int number, FILE *stream)
```

Функция читает из потока *stream* только *number* – 1 символов и записывает их в строку *str*. Это позволяет избежать атаки переполнения буфера (листинг II.7.13).

Замечание

При использовании функции `fgets()` в программу следует включать заголовочный файл `<stdio>`.

Листинг II.7.13. Использование функции `fgets()`

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

int main()
{
  char input[80];

  printf("Введите строку ");
  fgets(input, 80, stdin);

  printf("%s", input);

  return 0;
}
```

Управление потоками ввода/вывода осуществляется при помощи манипуляторов. *Манипуляторы* — это объекты, передача которых в потоки приводит к изменениям в интерпретации данных. Программист может использовать собственные манипуляторы, но чаще прибегает к стандартным манипуляторам. В табл. II.7.2 приводится список наиболее популярных стандартных манипуляторов.

Как видно из табл. II.7.2, для того чтобы ограничить пользовательский ввод 80 символами в библиотеке `<iostream>` языка C++, удобно воспользоваться манипулятором `setw()`, передав ему в качестве параметра количество символов, которые должен прочитать поток (листинг II.7.14).

Таблица II.7.2. *Стандартные манипуляторы*

Манипулятор	Поток	Описание
`flush`	`stdout`	Принудительный сброс буфера выходного потока
`endl`	`stdout`	Запись символа новой строки в буфер и сброс буфера выходного потока
`ends`	`stdout`	Запись символа завершения строки в буфер
`ws`	`stdin`	Чтение информации из выходного потока с игнорированием пропусков
`setw(`*`val`*`)`	`stdin`	Устанавливает ширину поля при вводе, равную значению параметра `val`
`setfill(`*`char`*`)`	`stdout`	Назначает символ-заполнитель для форматируемой строки (по умолчанию пробел)
`left`	`stdout`	Устанавливает выравнивание по левому краю
`right`	`stdout`	Устанавливает выравнивание по правому краю
`internal`	`stdout`	Устанавливает выравнивание знака числа по левому краю, а самого значения по правому краю
`showpos`	`stdout`	Вывод знака для положительного числа
`noshowpos`	`stdout`	Вывод положительного числа без знака
`uppercase`	`stdout`	Вывод символов в числах (`1.23E+02`) в верхнем регистре
`nouppercase`	`stdout`	Вывод символов в числах (`1.23e+02`) в нижнем регистре
`oct`	`stdin`, `stdout`	Запись и чтение в восьмеричной системе счисления
`dec`	`stdin`, `stdout`	Запись и чтение в десятичной системе счисления
`hex`	`stdin`, `stdout`	Запись и чтение в шестнадцатеричной системе счисления
`showbase`	`stdout`	Вывод идентификатора системы счисления
`noshowbase`	`stdout`	Запрет на вывод идентификатора системы счисления
`showpoint`	`stdout`	Десятичная точка всегда присутствует при выводе
`noshowpoint`	`stdout`	Десятичная точка не обязательна при выводе
`setprecision(`*`val`*`)`	`stdout`	Манипулятор устанавливает точность *`val`* вывода чисел
`fixed`	`stdin`, `stdout`	Запись числа в десятичном формате (`1200`), а не в научном (`1.2e+03`)

Таблица II.7.2 *(окончание)*

Манипулятор	Поток	Описание
scientific	stdin, stdout	Запись числа в научном формате (1.2e+03) вместо десятичного (1200)
skipws	stdin	Автоматическое игнорирование начальных пробелов при чтении данных оператором >>
noskipws	stdin	Начальные пробелы не игнорируются при чтении данных оператором >>
unitbuf	stdout	Принудительный вывод содержимого буфера после каждой операции записи
nounitbuf	stdout	Отмена принудительного вывода содержимого буфера после каждой операции записи

Замечание

При использовании манипуляторов ввода/вывода в программу следует включить заголовочный файл <iomanip>.

Листинг II.7.14. Ограничение вводимых символов при помощи манипулятора `setw()`

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

int main()
{
  char input[80];

  cout << "Введите строку ";
  cin >> setw(80) >> input;
  cout << input << endl;

  return 0;
}
```

II.7.5. Перегрузка операторов >> и <<

Выведем объект класса `cls` (см. листинг II.7.1) в формате `key: value`. Для этого перегрузим оператор <<, который принимает в качестве первого аргумента поток, а в качестве второго — ссылку на объект класса `cls` (листинг II.7.15).

Листинг II.7.15. Перегрузка оператора <<

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

ostream& operator << (ostream & strm, const cls &obj)
{
  strm << obj.key << ": " << obj.value;
  return strm;
}

int main()
{
  cls obj("Значение", 56);
  cout << obj << endl;

  return 0;
}
```

Следует отметить, что функция, перегружающая операторы << и >>, должна возвращать объект потока, чтобы имелась возможность выстраивать каскад из нескольких операторов << или >>.

Для того чтобы иметь возможность вводить объект класса `cls` из стандартного потока ввода, необходимо перегрузить оператор >> (листинг II.7.16).

Листинг II.7.16. Перегрузка оператора >>

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

ostream& operator << (ostream & strm, const cls &obj)
{
  strm << obj.key << ": " << obj.value;
  return strm;
}

istream& operator >> (istream & strm, cls &obj)
{
  string str;
```

```
  int val;
  cin >> str;
  cin >> val;
  obj = cls(str, val);

  return strm;
}

int main()
{
  cls obj;
  cout << "Введите строку и число ";
  cin >> obj;
  cout << obj << endl;

  return 0;
}
```

Если объект класса `cls` использует динамически выделенную память, то для успешной перегрузки оператора `>>` необходимо реализовать копирующий конструктор и перегрузить оператор `=`.

II.7.6. Собственный манипулятор

В *разд. II.7.4* рассматриваются стандартные манипуляторы, однако ничто не мешает разработчику создавать собственные манипуляторы. Манипуляторы представляют собой обычные функции, передаваемые операторам ввода/вывода в аргументах, которые, в свою очередь, осуществляют выполнение такой функции-аргумента. В листинге II.7.17 приводится пример перегрузки оператора вывода потока `ostream` для работы с манипуляторами `manip`.

Листинг II.7.17. Перегрузка оператора << для манипуляторов `manip`

```
ostream &ostream::operator << (ostream &(*manip)(ostream &))
{
  return (*manip)(*this);
}
```

Аргумент `manip` представляет собой указатель на функцию, которая должна получать и возвращать ссылку на поток `ostream`. При реализации манипуляторов обычно используют функции потоков. В листинге II.7.18 представлен манипулятор `endll`, который выводит в стандартный поток `stdout` два перевода строки и сбрасывает буфер потока.

Замечание

Допускается не только создание собственных манипуляторов, но также и перегрузка стандартных.

Листинг II.7.18. Реализация манипулятора `endll`

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

ostream& endll (ostream & strm)
{
  // Выводим два перевода строк
  strm.put('\n');
  strm.put('\n');
  // Сбрасываем буфер вывода
  strm.flush();

  return strm;
}

int main()
{
  cout << "Первая строка" << endll << "Вторая строка" << endll;

  return 0;
}
```

Как видно из листинга II.7.18, для реализации манипулятора использовались методы класса `ostream`. В табл. II.7.3 приводится список наиболее популярных методов класса `ostream`.

Таблица II.7.3. *Методы класса* `ostream`

Метод	Описание
`put(char ch)`	Метод записывает в поток вывода символ *ch* и возвращает объект класса `ostream`, по состоянию которого можно проверить, успешно ли выполнена запись
`write(const char *str, streamsize count)`	Метод записывает *count* символов строки *str* в поток `ostream` и возвращает объект класса `ostream`, по состоянию которого можно проверить, успешно ли выполнена запись. Перед вызовом необходимо убедиться в том, что строка *str* содержит не менее *count* символов, иначе поведение метода не предсказуемо

Таблица II.7.3 *(окончание)*

Метод	Описание
`flush()`	Метод сбрасывает все данные, которые накопились в буфере вывода в устройство записи, очищая буфер. Обычно информация сбрасывается из буфера при достижении определенного объема, однако это может быть неудобно при выполнении длительных вычислений, когда часть информации сбрасывается на устройство записи, а часть остается в буфере
`tellp()`	Метод возвращает текущую позицию записи. Применяется в основном при работе с файлами. Метод не применим при работе со стандартными потоками `cin`, `cout` и `cerr`
`seekp(pos_type` *`pos`*`)`	Устанавливает абсолютную позицию записи *`pos`*. Применяется в основном при работе с файлами. Метод не применим при работе со стандартными потоками `cin`, `cout` и `cerr`
`seek(int` *`offset`*`, pos_type` *`pos`*`)`	Устанавливает относительную позицию, смещая ее относительно позиции *`pos`* на *`offset`* позиций. Метод не применим при работе со стандартными потоками `cin`, `cout` и `cerr`

Класс `istream`, предназначенный для создания потоков ввода, также снабжен методами, которые представлены в табл. II.7.4.

Таблица II.7.4. *Методы класса* `istream`

Метод	Описание
`get()`	Метод возвращает следующий в потоке символ или константу `EOF`, если достигнут конец передаваемых данных
`get(char &`*`ch`*`)`	Метод помещает следующий в потоке символ в аргумент *`ch`*. Метод возвращает объект класса `istream`, по состоянию которого можно проверить, успешно ли выполнено чтение
`get(char *`*`ch`*`, streamsize` *`count`*`)`	Метод читает *`count`* символов из потока `istream` и помещает их в строку *`char`*. Чтение символов завершается досрочно, если читаемый символ является символом новой строки (символ новой строки не попадает в *`ch`*). Метод возвращает объект класса `istream`, по состоянию которого можно проверить, успешно ли выполнено чтение. Перед вызовом необходимо убедиться в том, что размер строки *`str`* достаточен для хранения *`count`* символов

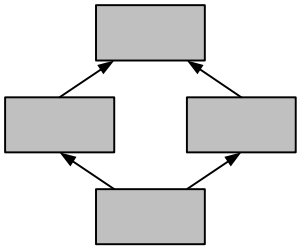
Таблица II.7.4 *(продолжение)*

Метод	Описание
`get(char *ch, streamsize count, char delim)`	Метод читает *count* символов из потока `istream` и помещает их в строку *char*. Чтение символов завершается досрочно, если читаемый символ совпадает с символом *delim* (символ *delim* не попадает в *ch*). Метод возвращает объект класса `istream`, по состоянию которого можно проверить, успешно ли выполнено чтение. Перед вызовом необходимо убедиться в том, что размер строки *str* достаточен для хранения *count* символов
`getline(char *str, streamsize coutn)`	Метод читает *count* символов из потока `istream` и помещает их в строку *str*. Чтение символов завершается досрочно, если читаемый символ является символом новой строки (символ новой строки попадает в *str*). Метод возвращает объект класса `istream`, по состоянию которого можно проверить, успешно ли выполнено чтение. Перед вызовом необходимо убедиться в том, что размер строки *str* достаточен для хранения *count* символов
`getline(char *str, streamsize count, char delim)`	Метод читает *count* символов из потока `istream` и помещает их в строку *str*. Чтение символов завершается досрочно, если читаемый символ совпадает с символом *delim* (символ *delim* попадает в *str*). Метод возвращает объект класса `istream`, по состоянию которого можно проверить, успешно ли выполнено чтение. Перед вызовом необходимо убедиться в том, что размер строки *str* достаточен для хранения `count` символов
`read(char *str, streamsize count)`	Метод читает `count` символов в строку *str*, в отличие от предыдущих методов, строка не завершается автоматически символом завершения строки. Метод возвращает объект класса `istream`, по состоянию которого можно проверить, успешно ли выполнено чтение. Перед вызовом необходимо убедиться в том, что размер строки *str* достаточен для хранения *count* символов. Обнаружение признака конца файла в процессе чтения считается ошибкой
`readsome(char *str, streamsize count)`	Метод читает *count* символов в строку *str*, в отличие от предыдущих методов, строка автоматически не оканчивается символом завершения строки. Функция возвращает количество прочитанных символов. Перед вызовом необходимо убедиться в том, что размер строки *str* достаточен для хранения *count* символов. Обнаружение признака конца файла в процессе чтения не считается ошибкой

Таблица II.7.4 *(окончание)*

Метод	Описание
`gcount()`	Метод возвращает количество символов, прочитанных последней операцией неформатированного ввода
`ignore()`	Метод извлекает один символ из потока `istream` и игнорирует его
`ignore(streamsize` *`count`*`)`	Метод извлекает *`count`* символов из потока `istream` и игнорирует их
`ignore(streamsize` *`count`*`, int` *`delim`*`)`	Метод извлекает *`count`* символов из потока `istream` и игнорирует их. Метод прекращает работу, если в потоке встречается символ *`delim`*
`peek()`	Метод возвращает следующий символ потока без реального чтения его из потока `istream`
`unget()`	Метод помещает в поток `istream` последний считанный символ таким образом, что его можно прочитать следующей операцией чтения
`putback(char` *`ch`*`)`	Метод помещает в поток `istream` последний считанный символ (если он совпадает с параметром *`ch`*) таким образом, что его можно прочитать следующей операцией чтения
`tellg()`	Метод возвращает текущую позицию чтения
`seekg(pos_type` *`pos`*`)`	Устанавливает абсолютную позицию записи *`pos`*. Применяется в основном при работе с файлами. Метод не применим при работе со стандартными потоками `cin`, `cout` и `cerr`
`seek(int` *`offset`*`,` `pos_type` *`pos`*`)`	Устанавливает относительную позицию, смещая ее относительно позиции *`pos`* на *`offset`* позиций. Метод не применим при работе со стандартными потоками `cin`, `cout` и `cerr`

ГЛАВА II.8



Разное

II.8.1. Кривая Безье

Для того чтобы построить кривую Безье, необходимо разбить отрезок от 0 до 1 на 1000 шагов и подставить полученные значения *t* в формулы из *разд. I.8.1*. В результате будет получено 1000 точек. Решение может выглядеть так, как это представлено в листинге II.8.1.

Листинг II.8.1. Сглаживание кривой

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

int main()
{
  float x0 = 0, x1 = 0, x2 = 0, x3 = 0, x = 0;
  float y0 = 0, y1 = 0, y2 = 0, y3 = 0, y = 0;

  cout << "Введите первую точку X Y ";
  cin >> x0 >> y0;
  cout << "Введите вторую точку X Y ";
  cin >> x1 >> y1;
  cout << "Введите третью точку X Y ";
  cin >> x2 >> y2;
  cout << "Введите четвертую точку X Y ";
  cin >> x3 >> y3;

  ofstream file("text.txt");
  for(float t = 0.0; t <= 1.0; t += 0.001)
```

```
  {
    x = (1 — t) * (1 — t) * (1 — t) * x0 +
        3 * t * (1 — t) * (1 — t) * x1 +
        3 * t * t * (1 — t) * x2 +
        t * t * t * x3;
    y = (1 — t) * (1 — t) * (1 — t) * y0 +
        3 * t * (1 — t) * (1 — t) * y1 +
        3 * t * t * (1 — t) * y2 +
        t * t * t * y3;
    file << x << "\t" << y << "\n";
  }
  file.close();

  return 0;
}
```

В листинге II.8.1 для вывода в файл используется объект класса `ofstream` из библиотеки `<fstream>`. Цикл `for` пробегает отрезок от 0 до 1 с интервалом 0.001, создавая 1000 точек сглаженной кривой.

Замечание

В реальной практике часто требуется строить сглаженные кривые по большему количеству точек. Кроме того, кривая Безье обладает тем недостатком, что кривая проходит только через две крайние точки и может не проходить через две средние точки. В общем случае при интерполяции прибегают к сплайнам, реализацию которых можно найти в соответствующих математических библиотеках.

II.8.2. Преобразование строк в массив

Для того чтобы прочитать содержимое файла, воспользуемся библиотекой ввода/вывода C++. Для этого необходимо объявить объект класса `ifstream` из библиотеки `<fstream>`, который будет связан с файлом. Для чтения содержимого файла можно воспользоваться методом `read()`, первый аргумент которого принимает строку, в которую помещается информация, а второй — количество байтов, которые необходимо прочитать из потока `ifstream`. Для того чтобы узнать количество байтов в файле, можно посимвольно читать файл до тех пор, пока не будет достигнут его конец. После чего можно повторно открыть поток для чтения содержимого файла (листинг II.8.2).

Листинг II.8.2. Чтение содержимого файла в строку `str`

```
#include <fstream>
#include <iostream>
```

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

int main()
{
  // Подсчитываем количество символов в файле
  ifstream file("linux.words");
  char ch;
  int length_file = 0;
  while(file.get(ch)) ++length_file;
  file.close();

  // Читаем length_file символов в строку str
  ifstream out("linux.words");
  char *str = new char[length_file];
  out.read(str, length_file);
  out.close();
  delete [] str;

  return 0;
}
```

Как видно из листинга II.8.2, первый блок программы подсчитывает количество символов в файле linux.words и помещает его в переменную `length_file`. Второй блок динамически выделяет `length_file` байтов под строку `str` и читает содержимое файла в эту строку. Два раза файл открывается из-за того, что после перебора содержимого первым блоком файловый курсор устанавливается в конец файла, повторное открытие файла позволяет переместить его в начало. Однако переместить курсор можно также и при помощи специального метода `seekg()` (листинг II.8.3).

Листинг II.8.3. Использование функции `seekg()`

```
#include <fstream>
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{

  ifstream file("linux.words");
  // Подсчитываем количество символов в файле
  char ch;
```

```
  int length_file = 0;
  while(file.get(ch)) ++length_file;

  // Устанавливаем файловый курсор на начало
  file.seekg(0, ios::beg);

  // Читаем length_file символов в строку str
  char *str = new char[length_file];
  file.read(str, length_file);
  file.close();
  delete [] str;

  return 0;
}
```

Метод `seekg()` используется применительно к объектам `istream`, для объектов класса `ostream` используется аналогичный метод `seekp()`. Функции имеют следующий синтаксис:

```
istream &seekg(off_type offset, seekdir origin)
ostream &seekp(off_type offset, seekdir origin)
```

Параметр *`offset`* является производным целого типа и определяет смещение относительно второго параметра *`origin`*, который может принимать три значения:

- `ios::beg` — начало файла;
- `ios::cur` — текущее положение;
- `ios::end` — конец файла.

Теперь, когда содержимое файла прочитано, можно приступить к созданию функций `explode()` и `implode()`. Для удобства функции будут работать не с C-строками, а `string`-строками библиотеки STL. В листинге II.8.4 представлена реализация функции `explode()`, которая разбивает содержимое строки `str` по символу `separator`, помещая фрагменты в коллекцию `vec`.

Листинг II.8.4. Функция `explode()`

```
void explode(char separator, string &str, vector<string> &vec)
{
  int begin = -1;
  int end = 0;
  for(int i = 0; i < str.length(); i++)
  {
    if(str[i] == separator)
```

```
    {
      begin = end;
      end = i;
      vec.push_back(str.substr(begin + 1, end − begin − 1));
    }
  }
}
```

В листинге II.8.5 представлена реализация функции `implode()`, которая объединяет подстроки из коллекции `vec` в единую строку `str`. Отдельные фрагменты разделяются между собой символом `separator`.

Листинг II.8.5. Функция `implode()`

```
void implode(char separator, vector<string> &vec, string &str)
{
  vector<string>::iterator pos;
  str = '\0';
  for(pos = vec.begin(); pos != vec.end(); ++pos)
  {
    str += *pos + separator;
  }
  str = str.substr(0,str.length() − 1);
}
```

Как видно из листинга II.8.5, коллекция `vec` перебирается при помощи итератора. Так как в конце цикла в строке `str` остается лишний разделяющий символ `separator`, он удаляется при помощи выделения из строки `str` подстроки (содержащей на один символ меньше, чем строка `str`) при помощи метода `substr()`. Метод `substr()` принимает в качестве первого параметра позицию, с которой начинается подстрока, а в качестве второго параметра — количество символов, которые следует извлечь. Если второй параметр не передается, то извлекается подстрока до конца файла.

В условиях задачи необходимо преобразовать файл linux.words, заменив в нем все переводы строк `\n` на пробелы. С использованием функций `explode()` и `implode()` решение задач может выглядеть так, как это представлено в листинге II.8.6.

Листинг II.8.6. Замена символов перевода строк `\n` на пробелы

```
#include <fstream>
#include <iostream>
```

```
#include <vector>
#include <string>
using namespace std;

void explode(char separator, string &str, vector<string> &vec);
void implode(char separator, vector<string> &vec, string &str);

int main()
{

  // Подсчитываем количество символов в файле
  ifstream file("linux.words");
  char ch;
  int length_file = 0;
  while(file.get(ch)) ++length_file;
  file.close();

  // Читаем содержимое файла в строку strng
  ifstream out("linux.words");
  char *str = new char[length_file];
  out.read(str, length_file);
  out.close();
  string strng = str;
  delete [] str;

  // Заменяем символы перевода строк на пробелы
  vector<string> vec;
  explode('\n', strng, vec);
  implode(' ', vec, strng);
  ofstream rewrite("linux.words");
  rewrite.write(strng.c_str(), strng.length());
  rewrite.close();

  return 0;
}
```

Метод `write()` потока `ofstream` принимает в качестве первого параметра С-строку. Поэтому для того чтобы записать `string`-строку, необходимо воспользоваться методом `c_str()`, который возвращает строку типа `char *`.

II.8.3. Разгрузка баржи

Создадим функцию `unloading()`, которая будет принимать объем баржи `volume`, объем грейфера `grab`, угол поворота `turn` и время оборота вокруг 360°

в минутах `minute`, а возвращать время в минутах, необходимое для разгрузки баржи (листинг II.8.7).

Листинг II.8.7. Функция `unloading()`

```
int unloading(float volume, float grab, int turn, int minute)
{
  return volume/grab*turn/360*2*minute;
}
```

В листинге II.8.8 представлена результирующая программа, которая подсчитывает количество минут, затрачиваемых на разгрузку баржи одним плавкраном каждого вида и двумя плавучими кранами в различных комбинациях.

Листинг II.8.8. Подсчет времени работы плавучих кранов

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

int unloading(float volume, float grab, int turn, int minute);

int main()
{
  int minI = unloading(2000, 1.5, 120, 1);
  int minII = unloading(2000, 3.5, 120, 2);
  int minIII = unloading(2000, 8.5, 120, 5);

  cout << "I       : " << minI << "\n";
  cout << "II      : " << minII << "\n";
  cout << "III     : " << minIII << "\n";

  minI = unloading(2000, 1.5, 240, 1);
  minII = unloading(2000, 3.5, 240, 2);
  minIII = unloading(2000, 8.5, 240, 5);

  cout << "I-I     : " << minI/2 << "\n";
  cout << "I-II    : " << (minI + minII)/2 << "\n";
  cout << "I-III   : " << (minI + minIII)/2 << "\n";
  cout << "II-II   : " << minII/2 << "\n";
  cout << "II-III  : " << (minII + minIII)/2 << "\n";
  cout << "III-III : " << minIII/2 << "\n";

  return 0;
}
```

Результатом работы программы из листинга II.8.8 являются следующие строки:

```
I       : 888
II      : 761
III     : 784
I-I     : 888
I-II    : 1650
I-III   : 1672
II-II   : 761
II-III  : 1545
III-III : 784
```

В результате получается, что наиболее быстрым решением является выбор второго типа плавкрана. Причем не имеет значения, работает один плавучий кран или два (впрочем, это можно было сказать заранее, т. к. 120 : 240 = 1 : 2). На разгрузку баржи один плавкран с грейфером объемом 3,5 тонны затратит 784 минуты, или 13 часов.

II.8.4. Длительность жизни ученого

В году 365 дней, однако раз в четыре года выпадает високосный год, в котором 366 дней. Поэтому при вычислениях на длительные сроки часто используют цифру 365.25, которая позволяет скомпенсировать високосные года. В программе количество дней в году помещается в переменную `year`. Уменьшение количества дней в году в результате негативного воздействия ученого совета помещается в переменную `academic_council`, а увеличение срока жизни из-за рыбалки — в переменную `fishing`. Общий эффект за год и от ученых советов, и от рыбалки хранится в переменной `delta`. При помощи программы, представленной в листинге II.8.9, можно выяснить, что каждый год уменьшает жизнь ученого на 108 дней.

Листинг II.8.9. Убыль срока жизни ученого в год

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

int main()
{
  // Дней в году
  float year = 365.25;
  // Убыль дней из-за ученого совета
  int academic_council = -14*12;
  // Увеличение срока жизни из-за рыбалки
  int fishing = 10*6;
```

```
  // Убыль или увеличение срока жизни в год
  int delta = academic_council + fishing;

  cout << delta << "\n";

  return 0;
}
```

Просто умножить величину `delta` на 40 лет (70 — 30) нельзя, иначе получится, что после смерти ученого ученые советы и рыбалка продолжают свое тлетворное и благотворное влияние. В результате срок жизни окажется искусственно заниженным. Необходимо использовать цикл от 30 лет до 70, сравнивая текущее значение срока жизни со счетчиком цикла. Как только счетчик превысит текущий срок жизни, необходимо прекращать работу цикла (листинг II.8.10).

Листинг II.8.10. Подсчет срока жизни ученого

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

int main()
{
  // Дней в году
  float year = 365.25;
  // Убыль дней из-за ученого совета
  int academic_council = -14*12;
  // Увеличение срока жизни из-за рыбалки
  int fishing = 10*6;
  // Убыль или увеличение срока жизни в год
  int delta = academic_council + fishing;

  float total = 70*365.25;
  for(float days = 30*365.25; days <= 70*365.25; days += year)
  {
    total += delta;
    if(days > total) break;
  }
  cout << total/year << "\n";

  return 0;
}
```

В результате работы программы можно узнать, что срок жизни ученого составит 60 с половиной лет.

II.8.5. Выгода предпринимателя

Простейшее решение, основанное на переборе всех номеров от 000000 до 999999, представлено в листинге II.8.11.

Листинг II.8.11. Вычисление выгоды предпринимателя

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

bool in_array(int value, int fst, int snd, int thd)
{
  return value == fst || value == snd || value == thd;
}

int main()
{
  // Количество счастливых билетов
  int cnt = 0;
  // Количество симметричных билетов
  int sym = 0;

  char number[6];
  for (int i=0; i<1000000; i++)
  {
    sprintf(number, "%06d", i);
    if(number[0] + number[1] + number[2] ==
       number[3] + number[4] + number[5])
    {
      cnt++;
      if(in_array(number[0], number[3], number[4], number[5]) &&
         in_array(number[1], number[3], number[4], number[5]) &&
         in_array(number[2], number[3], number[4], number[5])) sym++;
    }
  }
  // Выручка с рулона за счастливые несимметричные билеты
  int summ = (cnt - sym)*3;
  // Выручка с рулона за счастливые симметричные билеты
  summ += sym*10;
  // Выручка в месяц
  summ /= 10;
  // Прибыль в месяц
  summ -= 20000;
```

```
  cout << "Прибыль в месяц: " << summ << endl;

  return 0;
}
```

В результате выполнения программы из листинга II.8.11 можно выяснить, что прибыль предпринимателя составит 257 рублей в месяц, т. е. накладные расходы будут "съедать" весь заработок.

Подсчет количества симметричных билетов основывается на помещении номера билета в строку `number`, состоящую из 6 символов. Далее работа идет с ASCII-значениями цифр в строке, а не с самими значениями цифр — т. к. везде в дальнейшем сравниваются ASCII-значения, это не приводит к неправильному результату.

Для того чтобы отнести билет к счастливому, достаточно убедиться, что сумма первых трех цифр совпадает с суммой последних трех цифр. Если это условие выполняется, то счетчик счастливых билетов `cnt` увеличивается на единицу.

Для того чтобы убедиться, что билет является симметричным, используется функция `in_array()`, которая проверяет, совпадает ли первый аргумент функции с одним из следующих аргументов. Если имеет место совпадение, функция возвращает `true`, в противном случае возвращается `false`. Если каждая из трех первых цифр совпадает с одной из последних трех цифр (при условии того, что билет счастливый), то билет является симметричным.

На последнем этапе нужно вычесть из общего количества счастливых билетов симметричные билеты (по 10 рублей), чтобы получить количество счастливых несимметричных билетов (по 3 рубля). Так как в месяц продается лишь одна десятая от общего количества билетов (от 000000 до 999999), то для того чтобы получить ежемесячную прибыль, общую сумму необходимо поделить на 10 и вычесть из нее ежемесячные издержки в сумме 20 000 рублей.

Заключение

Чтобы не говорилось, чтение книги — это всегда в том или ином виде слушание авторского монолога. Что, в общем-то, не очень хорошо, поскольку быстрая обратная связь в равной мере необходима как читателям, так и авторам. С этой целью, чтобы авторский монолог превратить в полноценный диалог с читателем, специально для этой книги мы создали на своем сайте форум **http://www.softtime.ru/cpp/**, на котором можно задать свои вопросы. Поэтому авторы искренне надеются, что после того как вы перелистнете эту страницу, мы с вами не расстанемся, а встретимся на нашем форуме.

Успехов вам и всего доброго!

ПРИЛОЖЕНИЕ

Описание компакт-диска

На компакт-диске, поставляемом вместе с книгой, представлены решения задач, которые рассмотрены в разных главах. Поскольку в книге существует однозначное соответствие глав *части I* главам *части II*, то на компакт-диске приводятся решения задач из *части II*, постановка которых была сформулирована в *части I*. Коды программ сгруппированы в папках, перечисленных в табл. П1.

Таблица П1. *Содержимое компакт-диска*

Папка	Описание
code\1	Ответы к *главе 1*
code\2	Ответы к *главе 2*
code\3	Ответы к *главе 3*
code\4	Ответы к *главе 4*
code\5	Ответы к *главе 5*
code\6	Ответы к *главе 6*
code\7	Ответы к *главе 7*
code\8	Ответы к *главе 8*

Предметный указатель

П

С

У

Ф

Ш

Э