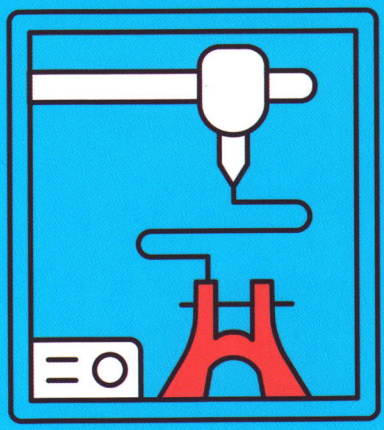


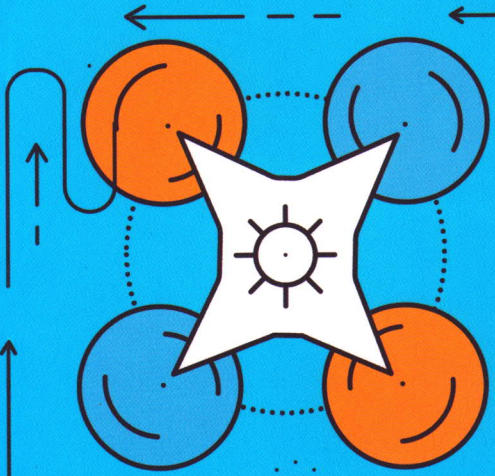
Основы языка C++ на практике

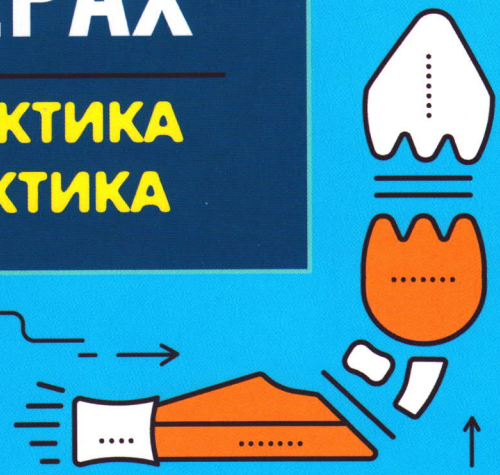
Показательные примеры (от простейшей программы до клиент-серверного приложения) с разбором кода

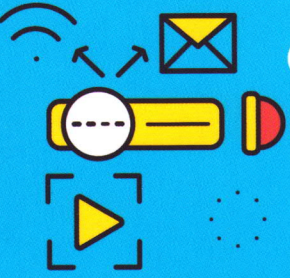
Соответствует всем последним стандартам (C++ 11, C++ 17)

C++ НА ПРИМЕРАХ

ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА







НиТ
издательство

"Наука и Техника"

Санкт-Петербург

Орленко П. А., Евдокимов П. В.

C++

на примерах

ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА

"Наука и Техника"

Санкт-Петербург

УДК 004.43 ; ББК 32.973

ISBN 978-5-94387-772-8

Орленко П. А., Евдокимов П. В.

C++ НА ПРИМЕРАХ. ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА —
СПб.: Наука и Техника, 2019. — 288 с., ил.

Серия "На примерах"

Эта книга является превосходным учебным пособием для изучения языка программирования C++ на примерах.

В книге рассмотрена базовая теоретическая часть языка C++, позволяющая ориентироваться в языке и создавать свои программы: типы, функции, операторы, логические конструкции, массивы, указатели, структуры, работа с файлами, объектно-ориентированное программирование. Отдельное внимание уделено программированию различных алгоритмов. В книге используется большое количество примеров с подробным анализом кода: от простых приложений для вывода текста на экран и проведения вычислений до клиент-серверного приложения.

Будет полезна как начинающим программистам, студентам, так и всем, кто хочет быстро начать программировать на C++.

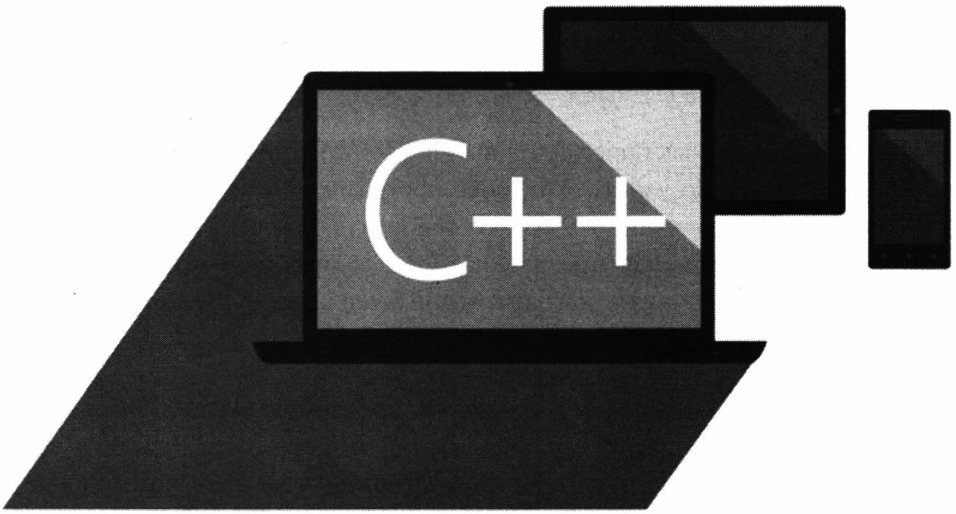
Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

Содержание

Глава 1.

Язык программирования C++



1.1. Что такое "Язык программирования C++"?

Прежде чем приступить к изучению языка программирования C++ очень желательно уже самого начала понимать, что это такое. Это было бы разумно. Согласитесь, что изначально иметь общее представление о том, что изучаешь, очень важно для эффективности самого изучения.

Итак, что же такое язык программирования C++? Чтобы ответить на данный вопрос, сначала надо узнать, что такое программирование. Тут совсем все просто: программирование – это деятельность по созданию компьютерных программ. Компьютерная программа – это записанный в понятном для компьютера виде алгоритм, который компьютер в состоянии исполнить.

Так вот язык, на котором записываются алгоритмы в понятном для компьютера виде, и называется языком программирования. Язык C++ является одним из них (C++ читается как "Си плюс плюс" или, что более правильно, как "Си плас плас"). К слову, алгоритм – это последовательность действий, которая приводит к определенному результату.

Проблема состоит в том, что компьютерные программы пишутся людьми, а исполняются компьютерами. Суть же проблемы в том, что человеку понятен язык, который хоть как-то похож на человеческий, в то время как компьютер по большому счету понимает только команды по работе с памятью: прочитать, записать, побитно преобразовать определенный участок памяти определенным образом. И все искусство создания языков

программирования в том и состоит, чтобы найти некоторое оптимальное решение, которое устраивало бы всех: и человека – для формулирования задачи, и компьютер – для понимания и выполнения задачи.

Полностью соединить "ужа с ежом" не представляется возможным и, что самое главное, нужным. То есть, чтобы компьютер понимал команды языка программирования, их необходимо "прошить" на аппаратном уровне в процессор – устройство в составе компьютера, которое осуществляет все операции данными в рамках исполнения алгоритма. Все возможные языки программирования в процессор не зашьешь, тем более что они постоянно дополняются: появляются как новые возможности в уже существующих языках программирования, так и создаются совершенно новые языки программирования. В связи с этим было принято следующее решение: разделить языки программирования на две группы: языки низкого уровня и языки высокого уровня. Первые понятны компьютеру и их ограниченное число (в рамках компьютера), вторые понятны человеку и их произвольное число (на одном компьютере вы можете программировать практически на любом языке программирования). И есть переводчики, которые переводят программу с языка высокого уровня на язык низкого уровня.

При этом возможны два варианта

- Первый вариант - это когда сначала осуществляется перевод программы с языка верхнего уровня на язык нижнего уровня, и только после этого программа может быть запущена и быть исполнена компьютером.

- Второй вариант – это когда программа сначала запускается, а перевод осуществляется в ходе ее исполнения.

В качестве аналогии данных двух вариантов можно привести пример с переводом с одного человеческого языка на другой (например, с английского на русский). При этом можно сначала отдать переводчику провести весь текст, и потом его прочитать. А можно сказать переводчику, чтобы он переводил "на лету" -- выполнить синхронный перевод.

В первом варианте, обычный переводчик называется, компилятором, а во втором варианте, синхронный переводчик -- интерпретатором. Все

языки программирования поделены на две группы: компилируемые – для которых доступен первый вариант, и интерпретируемые – для которых доступен второй вариант. При этом каждый язык программирования либо компилируемый, либо интерпретируемый.

И вот мы подошли к пониманию того, чем же является язык C++. Язык C++ -- это компилируемый язык программирования высокого уровня.

1.2. Немного истории, или откуда взялся язык программирования C++

Исторически язык C++ является эволюционным развитием языка программирования C (читается как Си). Эволюционный скачок был ознаменован добавлением к языку C новой парадигмы программирования: объектно-ориентированной. В то время как язык C подчинен только парадигме процедурного программирования. Примерно так и получился язык C++.

> **Примечание**. Что такое функциональное программирование и что такое объектно-ориентированное программирование мы свами обсудим чуть позже.

Язык же C в свою очередь появился в начале семидесятых годов прошлого века (1970-xx) в стенах компании Bell Laboratories при разработке операционной системы Unix. Автором языка C принято считать Денниса Ритчи (Dennis Ritchie), которому не хватало имевшихся на тот момент языковых средств для написания кода Unix, и он взял и придумал новый язык.

1.3. Общий порядок создания программы на C++

Общий порядок создания программы на C++, которую можно запускать прямо из операционной системы (например, двойным щелчком мыши по файлу программы), состоит и следующей последовательности шагов:

1. Создание исходного кода программы. Это непосредственное написание текста программы с помощью обычного или

специализированного текстового редактора. Текст программы можно набрать и в Блокноте, а при сохранении необходимо файлу присвоить расширение .cpp. Это стандартное и наиболее распространенное расширение файлов с исходным C++-кодом, но встречаются еще и другие. Главное, чтобы это расширение было понятно компилятору.

2. Создание объектного кода программы. Это, как мы уже писали выше, перевод программы с языка высокого уровня в бинарный вид, понятный компьютеру и выполняемый процессором. В рамках данной процедуры текстовый файл с исходным кодом передается компилятору для компилирования, который по итогам свой работы выдает объектный файл (с расширением o или obj).

3. Создание исполняемого кода программы. Компилятор в ходе своей работы за сеанс обрабатывает только один файл с исходным кодом только одной программы, игнорируя связи, которые могут быть в коде программы с кодом в других файлах. В то же время, без связанных элементов (а таковые, как правило, есть) скомпилированная программа работать не будет. Поэтому нужно к объектному коду программы добавить еще объектный код всех связанных элементов. Так вот процесс распознавания всех связей и объединения связанных объектных файлов в одно целое называется компоновкой. А служебная программа, которая это делает, – компоновщик. Результатом такого объединения является исполняемый код, который уже можно запускать на исполнение и распространять среди пользователей как законченный программный продукт.

Таким образом для создания программы на C++ вам понадобятся три служебных программных приложения: текстовый редактор для создания исходного кода, компилятор для создания объектного кода, компоновщик для создания исполняемого кода программы. Часто все

эти три составляющие образуют единый пул, объединенный под единым названием "интегрированная среда разработки" (IDE) и устанавливаются одновременно. Но, тем не менее, это все три разных приложения.

К слову сказать, Бьерн Страуструп при создании языка C++ первоначально не стал реализовывать прямой компилятор из языка C++ в объектный код, а написал компилятор-транслятор, который транслировал исходный код на C++ в исходный код на языке C, компилятор для которого уже существовал. Этот ход способствовал быстрому распространению языка C++.

1.4. Что нужно установить на компьютере, чтобы создавать программы на C++

Устанавливаем среду разработки

Для написания программ на C++ вам нужно установить интегрированную среду разработки, которая в себя бы сразу включала и редактор кода, и компилятор, и компоновщик.

Далее в книге используется бесплатная среда разработки Dev C++. Ее можно бесплатно скачать с сайтов:

- http://orwelldevcpp.blogspot.com/
- https://sourceforge.net/projects/orwelldevcpp/files/latest/download

Это очень хорошая среда разработки начального уровня. С ней вы без труда разберетесь. Для профессионального программирования можно порекомендовать среду разработки Microsoft Visual Studio.У нее тоже есть бесплатный вариант, но для начального уровня программирования на C++ она будет очень громоздкой.

Внимание! Имейте в виду, что названия файлов проектов на C++ нужно называть только с использованием латинских букв. Использование русских букв в имени файла проекта скорее всего приведет к тому, что проект не откопилируется.

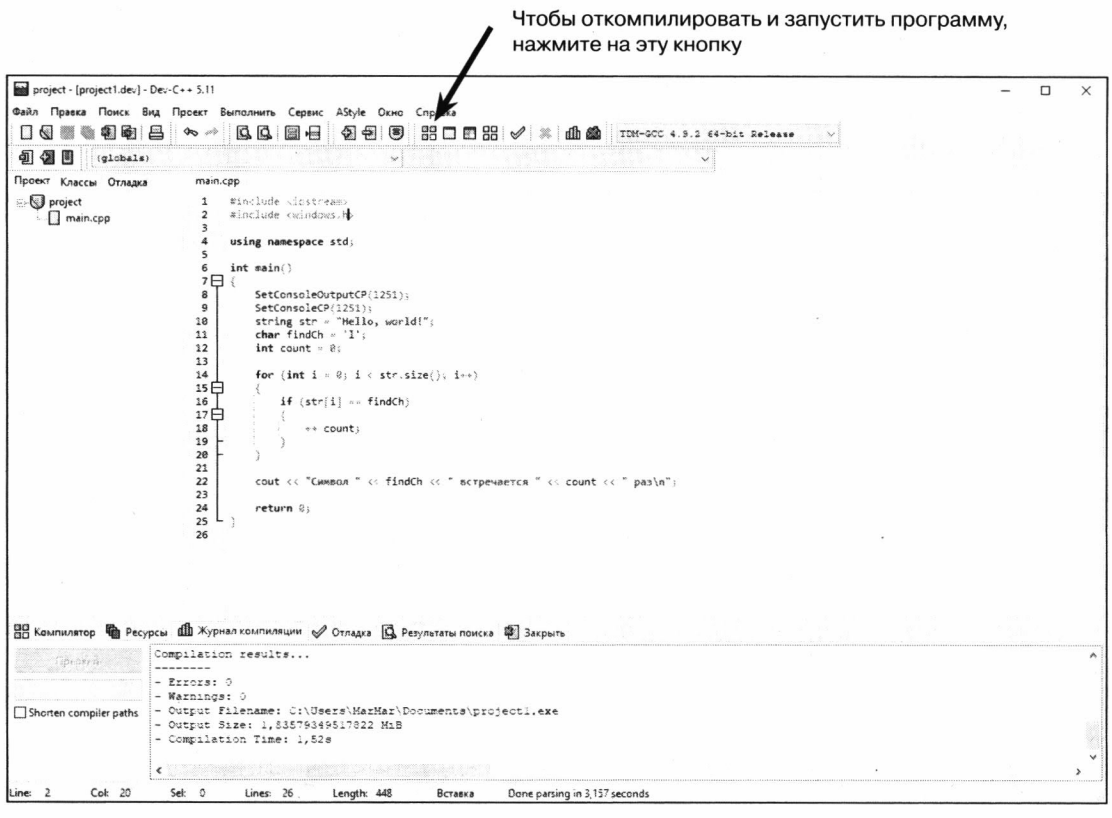


Рис. 1.1. Среда разработки Dev C++.

Как сделать так, чтобы текст при выполнении программ выводился на русском языке

Все программы, которые мы научимся с вами писать в рамках данной книги, запускаются и работают в терминальном режиме (то есть текстовом режиме). При этом могут быть сложности с отображение русских символов при работе программ.

Чтобы такого не было нужно:

- Либо настроить среду разработки. В Dev C++ для этого нужно в строке меню выбрать **Сервис> Параметры компилятора**, затем в поле "Добавить следующие команды" ввести: -fexec-charset=cp866 -finput-charset=cp1251

- Либо в начале программы , в начале основной функции main() добавить следующий оператор:

```
setlocale(LC_ALL, "Russian");
```

1.5. Какие программы правильные, а какие программы неправильные

Компьютерная программа, написанная на языке программирования, может быть правильной или неправильной. При этом правильной или неправильной она может быть как по смыслу, так и по содержанию.

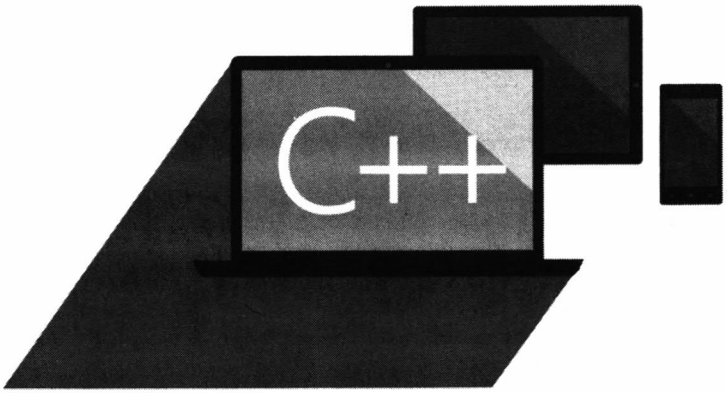
За правильность содержания следит компилятор. И если вы неправильно написали какое-то ключевое слово, не поставили точку с запятой после окончания оператора, забыли объявить переменную или сделали еще что-то в этом роде, то компилятор просто не откомпилирует вашу программу и выдаст ошибку. Так что первой проверкой правильности является то, компилируется ваша программа или нет. Если программа компилируется, то принято говорить, что она работает.

Смысловая правильность говорит о том, что ваша программа делает именно то, что она должна делать. Проконтролировать смысловую правильность

может только человек, поэтому в области информационных технологий существует такая профессия, как тестировщик. Когда программа не правильна по смыслу, то говорят, что она работает неправильно, глючит и т.п. То есть программа может откомпилироваться, она может запускаться, но делать она будет при этом совершенно не то, что нужно.

Глава 2.

Первая программа на языке C++



2.1. Из чего состоит программа на C++

Программа на языке C++ представляет собой набор функций, каждая из которых состоит из операторов. Операторы могут входить в состав выражения. Выражение – это последовательность операторов и операндов.

Оператор – это наименьшая автономная часть языка, которая выполняет какое-либо действие. Операнд в языке программирования – это аргумент оператора; данные, которые обрабатываются оператором при осуществлении его действия.

Примечание. Функции еще часто называют методами.

2.2. Самая короткая программа на C++

Самая короткая программа на языке C++ выглядит следующим образом:

```
int main()
{
}
```

Эта программа абсолютно правильна, она успешно откомпилируется, но при этом она ничего не делает. В этой программе определяется одна единственная функция под названием main(), которая не имеет параметров и ничего не делает. Параметры указываются в круглых скобках, а операторы, которые должны что-то делать в заданной функции, приводятся между фигурными скобками. И там, и там, как мы видим, ничего нет.

Однако данный пример позволяет нам продемонстрировать одну очень важную вещь: программа на C++ как минимум должна иметь функцию main(). Функция эта в программе может быть только одна. Работа программы начинается с выполнения этой функции и заканчивается выполнением

этой функции. Внутри функции main() могут использоваться операторы, которые определяют логику работы программы. В качестве операторов могут выступать вызовы других функций.

Помимо функции main() в программе могут быть и другие функции, но об этом мы поговорим отдельно.

2.3. Функция main()

Структура функции main() такова:

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

Структурно функция main() как и любая другая функция состоит из двух частей: заголовка функции и тела функции. Строка int main() называется заголовком функции. Все что находится между фигурными скобками называется телом функции. Вместе они образуют определение функции.

Давайте разберем в деталях определение функции main(). Это необходимо сделать, так как она является основополагающей. В C++ функция, которую вызывают из другой функции, возвращает результат своей работы в вызвавшую функцию, в место вызова. Так вот в заголовке перед именем функции указывается тип значения, возвращаемого функцией – возвращаемый тип функции. Перед именем функции main() стоит int , что указывает на целочисленный тип возвращаемого значения. О типах значений мы поговорим чуть позже. Сейчас же важно усвоить одно, что функция main() всегда возвращает целочисленное значение, и перед ее именем должен стоять тип int.

Закономерен вопрос: а кто это вызывает функцию main(), если эта функция является самой главной и все функции вызываются уже из нее? И кто будет использовать возвращаемое функцией main() значение? Ответ простой: операционная система. Именно она вызывает вашу программу, когда вы запускаете ее двойным щелчком мыши по имени файла. А по возвращаемому

целочисленному значению операционная система может судить, правильно ли отработала программа или с какими-то отклонениями и ошибками.

Для возврата целочисленного значения используется оператор return. Вообще оператор return используется не только в main(), но в во всех остальных функциях для возврата определенного значения в точку вызова функции. Понятно, что оператор return должен стоять в конце тела функции.

Выше можно видеть, что в конце тела функции main() стоит оператор return 0, который возвращает значение 0 в операционную систему. Значение 0 говорит системе о том, что программа была выполнена успешна и все хорошо.

Еще выше можно видеть, что в тексте нашей с вами короткой программы мы никакого оператора return не указывали. Дело в том, что стандарт C++ предписывает компилятору, если он дошел до конца функции main() и не встретил там return, автоматически подставлять return 0. Это называется неявным завершением, и он возможен только для функции main(), для любой другой функции неявное завершение недопустимо и компилятор просто выдаст ошибку.

2.4. Самая простая программа на C++

Выше мы привели самую короткую программу, но при этом и самую бестолковую. Зачем нужна программа, которая ничего не делает?

Давайте придадим смысл нашей программе и добавим в нее простейшее действие, которое даст видимый результат: пусть наша программа выведет какой-либо текст. Традиционно таким текстом является фраза "Hello, World!" (Привет, Мир!), которую по сложившейся традиции выводят в качестве первой фразы во всем мире, когда пишут свою первую программу на том или ином языке программирования. Мы тоже не будем нарушать данную традицию, код программы приведен в листинге 2.1.

Листинг 2.1. Код программы "Hello, World!"

```
// Программа, которая выводит строку текста
#include <iostream>
using namespace std;

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

После запуска данной программы на экране появится строчка:

```
Hello, World!
```

Давайте разберем, из чего состоит программа:

- Комментарии – первая строка, начинающаяся с // . Данная информация не обрабатывается компилятором. Вы можете в качестве комментариев добавлять любой текст. Обычно это делается для того, чтобы дать какие-либо пояснения по коду программы, а комментарии в самом начале программы обычно описывают, для чего программа предназначена. Вы можете добавлять неограниченное количество комментариев. При этом комментарием считается все, что расположено в строке справа от символов //. Если текст комментария у вас перешел на следующую строку, то необходимо перед ним также поставить // на следующей строке.

- Директива препроцессора #include – во второй строчке. С помощью этой директивы мы подключаем заголовочный файл iostream, позволяющий работать с потоками ввода/вывода (или, говоря по-простому, вводить данные с клавиатуры и выводить данные на экран). Язык C++ содержит в себе стандартную библиотеку классов функций, распределенных по заголовочным файлам. Чтобы воспользоваться какой-либо стандартной функцией или методом какого-либо класса, необходимо подключить заголовочный файл, в котором эти вещи определены. Класс, отвечающий за работу с потоком вывода (cout) определен в заголовочном файле iostream. Таким образом, чтобы вывести текст на экран нам нужен cout, а чтобы воспользоваться cout нам нужно подключить заголовочный файл iostream.

- Директива using namespace – в третьей строке. С помощью этой директивы мы устанавливаем стандартное пространство имен std. Дело в том, что каждая программа, программная библиотека, использует свое пространство имен. И когда мы подключаем с помощью директивы препроцессора #include один программный модуль к другому, то может возникнуть путаница с именами. Например, и в исходном файле и в подключенном файле могут быть определены две функции с одним и тем же именем kuponator(). Но тогда какую из них использовать? Указание пространства имен позволяет явно задавать, какую именно. Стандартная библиотека C++, в которую в том числе входит и заголовочный файл iostream имеет стандартное пространство имен std. Его то мы и подключаем.

- Выполнение программы на C начинается с функции main() - это точка входа в программу. В четвертой строке приведен заголовок функции main(), а далее идет тело функции main(), заключенное в фигурные скобки.

- В C++ нет функций вывода/ввода. Вместо них используются операторы вывода/ввода << и >>. Далее (в следующем примере) будет показано, как прочитать ввод с клавиатуры.

- Оператор << выводит на стандартный поток вывода (cout) строчку "Hello, world!"

- Оператор возврата return 0 завершает выполнение функции main() и программы в целом. Код возврата 0 обычно соответствует отсутствию ошибок. Любое другое значение означает код ошибки.

2.5. Использование переменных. Оператор объявления

В предыдущем разделе мы с вами написали программу, которая выводит строку "Hello, World!" на экран. Для этого мы использовали оператор <<, который помещал эту строку в поток вывода.

Допустим, теперь, что мы хотим, чтобы пользователь что-то ввел с клавиатуры, а мы в программе это как-то обработали. Например, пусть

пользователь введет какое-либо целое число, а мы его потом выведем на экран. И вот тут встает проблема: а куда нам положить введенное пользователем число. А нам это нужно сделать, чтобы впоследствии с этим числом работать (вывести на экран).

В компьютере все строго, и в компьютере все данные хранятся в памяти. Соответственно нам надо выделить память, в которую можно будет положить введенное пользователем число, а потом обратиться к этому фрагменту памяти, чтобы вывести его содержимое на экран. Таким образом нам требуется идентифицировать фрагмент памяти и его объем, достаточный для хранения числа. Вот мы и пришли к такому понятию, как переменная. **Переменная – это поименованная область памяти компьютера.** То есть это область памяти, имеющее имя. Понятное дело, что имя должно быть уникально, так как именно по имени переменной компьютер понимает к какой области памяти он должен обратиться. Значение, которое хранится в памяти, отведенной под переменную, называется **значением переменной**.

Объявление переменной в языке C++ осуществляется с помощью оператора объявления. Общий вид этого оператора таков:

Тип_переменной Имя_переменной;

Тип переменной определяет тип хранимой информации и характеризует объем памяти, который выделяется под переменную.

Например:

```
int polovnik;
```

Здесь мы объявили переменную polovnik целочисленного типа int. Размер типа int составляет 4 байта. Так что мы получили 4 байта с именем polovnik, в которые можно положить целое число от -2 147 483 648 до 2 147 483 647.

Объявлять переменную можно практически в любом месте программы, но перед использованием этой переменной. Также необходимо знать, что если переменная объявлена в определенном типе, то поменять ее тип в дальнейшем невозможно.

Давайте теперь напишем программу, которая, как мы и хотели, будет получать от пользователя какое-либо целое число, а потом выводить его на экран монитора. Текст программы приведен в листинге 2.2.

Листинг 2.2. Объявление переменной и ввод с клавиатуры

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

int main()
{
    int number;

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

    cout << "Вы ввели " << number << endl;
    return 0;
}
```

Рассмотрим отличия программы от листинга 2.1:

- Мы объявляем целочисленную (int) переменную с именем number.
- С помощью оператора вывода << мы выводим приглашение ввести число на стандартный поток вывода cout.

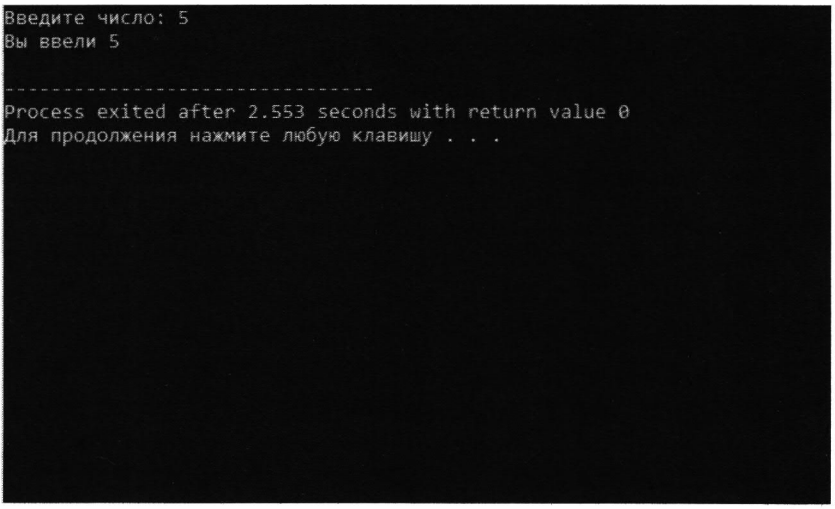


Рис. 2.1. Результат выполнения программы

- Используя оператор ввода >>, мы читаем введенное пользователем значение в переменную number.

- Затем мы выводим на стандартный вывод строку "Вы ввели ", введенное пользователем число и модификатор endl, который осуществляет переход на следующую строку.

2.6. Инициализация переменной. Оператор присваивания

В приведенном выше примере (листинг 2.2) мы объявили переменную и потом поместили в эту переменную значение, введенное пользователем. А как быть, если мы хотим поместить какое-либо значение в переменную по ходу выполнения программы, но при этом данное значение не вводится пользователем. Например, мы хотим поместить в переменную sum сумму двух чисел, введенных пользователем. В данном случае нам понядобится объвить три переменные: одну для первого числа, введенного пользователем, вторую – для второго числа, третью – для суммы двух чисел. И по-прежнему остается открытым вопрос, как поместить вычисленное значение суммы в третью переменную.

Для присваивания определенного значения переменной в языке C++ предназначен оператор присваивания.

Общая форма его такова:

Имя_переменной = Значение;

Присваивание какого-либо значения переменной возможно только после ее объявления, и присвоено может быть значение только того типа, который был задан переменной при ее объявлении. То есть нельзя присвоить символьное значение целочисленной переменной:

```
int a;
a = "б";  // данное присваивание недопустимо
```

Очень часто объявление и присваивание совмещаются. Это когда значение переменной присваивается при ее объявлении. В таком случае имеет место **инициализация переменной**. В общем виде инициализация выглядит следующим образом:

Тип_переменной Имя_переменной = Значение;

Если переменная была объявлена, но ей еще не было присовено никакое значение, то такая переменная называется **неинициализированной переменной.**

Обратите внимание, что объявлять и инициализаровать можно сразу по несколько переменных. Делается это через запятую. Например, в C++ вполне корректна следующая запись:

```
int a, b = 0, c;
```

Здесь мы объявили две переменные (a, c) и инициализировали одну переменную (b).

Теперь напишем код программы, которая просит пользователя ввести два целых числа, а затем вычисляет сумму этих чисел и сообщает пользователю результат. Листинг примера приведен ниже.

Листинг 2.3. Сумма двух чисел

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

int main()
{
    int first, second, sum;

    cout << "Введите два целых числа: ";
    cin >> first >> second;

    // Вычисляем сумму
    sum = first + second;

    // Выводим сумму
    cout << first << " + " <<  second << " = " << sum << endl;
```

```
    return 0;
}
```

Разбираем программу:

- На этот раз мы читаем не одно число, а два числа. Обратите внимание, как производится чтение двух чисел - просто в одной строчке мы указываем два оператора >>.
- Далее мы вычисляем сумму двух чисел с помощью оператора +.
- Выводим результат на экран в виде *Число + Число = Сумма.*
- Модификатор endl означает конец строки и переход на следующую строку. Вместо данного модификатора можно использовать сочетание "\n".

```
Введите два целых числа: 5
7
5 + 7 = 12

--------------------------------
Process exited after 5.269 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

Рис. 2.2. Сумма двух чисел

2.7. Базовые типы данных C++

2.7.1. Типы данных в C++

Тип данных определяет размер памяти, выделяемой под переменную данного типа при ее создании, а также позволяет судить о том, какого рода информация может содержится в той или иной переменной.

Все типы данных в языке C++ подразделяются на три категории:

- **Базовые типы** – это стандартные типы данных, предопределенные в самом языке. Их имена зафиксированы в самом стандарте и обозначаются ключевыми словами, которые никак кроме как для обозначения типа использовать нельзя. Базовы типы являются минимальными кирпичиками, на основе которых можно строить более сложные, пользовательские типы данных. В то же время базовые типы нельзя разложить на более простые.

- **Производные типы** – это типы данных, создаваемые разработчком при написании программы.

- **Типы класса** – в рамках объектно-ориентированного программирования каждый класс является типизирующим элементом. То есть каждый экземпляр класса (объект), используемый в программе, имеет тип по имени класса.

2.7.2. Базовые типы данных

Общее описание

В языке C++ определено 7 базовых типов данных:

- символьный;
- символьный двухбайтовый;
- логический;
- целочисленный;
- вещественный;

- вещественный двойной точности;
- пустой.

Соответственно в языке C++ предусмотрено семь базовых имен (идентификаторов) типа, приведенных в таблице 2.1.

Таблица 2.1. Имена базовых типов

Имя (идентификатор) типа	Тип данных
`bool`	Логический тип
`char`	Символьный тип
`wchar_t`	Символьный двухбайтовый тип
`double`	Вещественные числа двойной точности
`float`	Вещественные числа
`int`	Целые числа
`void`	Значение не возвращается (пустое значение)

Совместно с именами типов могут использоваться модификаторы типа. Модификаторы типа – это ключевые слова, которые могут указываться перед именем базового типа и модифицировать его. В языке C++ предусмотрено четыре модификатора типа:

- signed – значение со знаком;
- usigned – значение без знака;
- short – укороченный типа;
- long – расширенный тип.
- long long – целевой тип будет иметь размер не меньше 64 бит (нововведение стандарта C++11)

Таблица 2.2. Базовые типы C++ (с модификаторами) и их граничные значения

Обозначение		Название	Размер памяти, байт (бит)	Диапазон значений
Имя типа	**Другие имена**			
int	signed	целый	4 (32)	-2 147 483 648 до 2 147 483 647
	signed int			
unsigned int	unsigned	беззнаковый целый	4 (32)	0 до 4 294 967 295
short int	short	короткий целый	2 (16)	-32 768 до 32 767
	signed short int			
unsigned short int	unsigned short	беззнаковый короткий целый	2 (16)	0 до 65 535
long int	long	длинный целый	4 (32)	-2 147 483 648 до 2 147 483 647
	signed long int			
unsigned long int	unsigned long	беззнаковый длинный целый	4 (32)	0 до 4 294 967 295
long long int	long long	длинный-предлинный целый	8 (64)	-9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
	signed long long int			

unsigned long long int	unsigned long long	беззнаковый длинный-предлинный целый	8 (64)	0 до 18 446 744 073 709 551 615
char	signed char	байт (целый длиной не менее 8 бит)	1 (8)	-128 до 127
unsigned char	-	беззнаковый байт	1 (8)	0 до 255
wchar_t	-	расширенный символьный	2 (16)	0 до 65 535
float	-	вещественный одинарной точности	4 (32)	3.4E-38 до 3.4E+38 (7 значащих цифр)
double	-	вещественный двойной точности	8 (64)	1.7E-308 до 1.7E+308 (15 значащих цифр)
long double	-	вещественный максимальной точности	8 (64)	1.7E-308 до 1.7E+308 (15 значащих цифр)
bool	-	логический	1 (8)	true (1) или false (0)
void	-	пустой	-	-

Символьные типы

Символьный тип редназначен для объявления переменной в которой будет храниться какой- либо символ.

Например:

```
char parameter;
parameter = 'd';
```

В этом примере мы объявили переменную символьного типа, а потом присвоили ей символ d. Обратите внимание, что символы должны быть заключены в одинарные кавычки.

Здесь необходимо пару слов сказать о том, как хранятся символьные значения в памяти компьютера. Как вы знаете, память компьютера состоит из байт и бит, в каждый бит может быть записано 0 или 1. Соотвественно, получается, что таким образом моэно хранить только числа (преобразовава их в двуичную систему счисления). Как же тогда хранятся символьные данные? В памяти отведенной под симовльную переменную, также хранится число, но это число является кодом, которому соответсвует какой либо символ. Работая или присваивая символьные данные какой-либо переменной, компилятор преобразует символы в числовое представление, которое может быть помещено в память.

Числовое представление латинских символов A-Z, a-z, чисел 0-9, некоторых специальных клавиш (например, <DEL>) и специальных символов (таких, как "тильда", "слеш" и т. п.), было стандартизовано в таблице-кодировке ASCII, испльзуемой в C++. Так, в кодирове ASCII символу d соотвесвует значение 100, которое и будет храниться в памяти. Но по названию типа (char) мы и программа знаем, что в этой области памяти хранится не число, а символ. И если мы захотим вывести на экран значение символьной переменной, то вместо числа будет выведен соответсвующий ему символ по ASCII-кодировке.

Кстати, из-за того, что символьные переменные хранятся в памяти в числовом виде, с ними можно выполнять операции как с обычными числами: например, складывать. При этом сложатся числа, которые хранятся в памяти, и в результате получится число, по которому также можно отобразить соответствующую букву. Пример приведен в листине 2.4.

Листинг 2.4. Складываем буквы

```
#include <iostream>
using namespace std;
int main()
{
    char simvol1 = 'A';
    char simvol2 = 'B';
    char simvol;

    simvol = simvol1 + simvol2;

    cout << "Получилось" << simvol << endl;
    return 0;
}
```

```
Получилось Г

--------------------------------
Process exited after 0.06843 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

Рис. 2.3. Результат сложения двух букв

Целочисленные типы

Целочисленные типы предназначены для объявления переменных, в которых будут храниться целые числа, размер которых ограничен ограничениями типа. При этом есть типы у которых целое число может испть

как положительно, так и отрицательное значение. А есть типы, переменные которых могут принимть только положждительные целочиселнные значения.

Целочисленное значение – э то значит целое число, без десятичного разделителя. Например: 34 – это целое число, а 36.6 – это не целое (вещественное) число.

Вещественные типы

Вещественные типы предназначены для хранения вещественнх чисел, то есть чисел, имеющих десятичный раздедитель, дробную часть. Например, вещественнеы переменные могут хранить такие числа, как: 0.99, 388.2, 10.0, -2334.0001

В качестве десятичного разделителя в C++ используется символ "." - точка.

Логический тип

Логический тип предназначен для определения логических переменных, то есть переменных, значением которых может быть true (истина) или false (ложь). Этот тип особенно полезен при хранении настроек, когда вам необходмо хранить информаци о том что-то вкл.чено (true) или выключено (false), что-то существует (true) или отсутствует (false).

Например:

```
bool flag;
flag = true;
```

Для логических переменных необходимо отметить, что true эквивалентно целому значению 1, а false – целому значению 0. При этом логической переменной можно присвоить целое значение 1 (и переменной будет присвоено значение true), а если попытаться вывести на экран значение переменной, имеющей значение true, то на экран будет выведено значение 1. Аналогично и со значением 0 и false.

Тип void

Тип void обычно используется для указания типа значения, возвращаемого функцией, когда эта функция ничего не возвращает. Согласно стандарта

C++ для кажой функции мы должны указывать тип возвращаемого значения. Но далеко не все функции в результате своей работы должны что-либо возвращать. Такие функции в языке Pascal называются процедурами.

Вот как будет выглядеть заголовок такой функции:

```
void function()
```

2.7.3. Модели памяти

Размер памяти, отводимый под переменную того или иного типа, как показано в таблице 2.2, согласно стандарта C++ имеет формулировку "не менее". То есть допускается и больший размер памяти, выделяемый, скажем, на переменную типа int. Меньше 4 байт нельзя, а вот больше – можно.

По факту размер памяти, выделяемой под переменные того или иного типа зависит от архитектуры компьютера и может быть разным (но не менее того, что указан в таблице). Совокупность значений размеров базовых типов для той или иной архитектуры назвается **моделью данных**. На данный момент наиболее распространены следующие 4 модели данных:

32-битные системы:

- **LP32** или **2/4/4** (int — 16 бит, long и указатель — 32 бита): в архитектуре Win16 API ;
- **ILP32** или **4/4/4** (int, long и указатель — 32 бита) в архитектурах: Win32 API, Unix и Unix-подобных системах (Linux, Mac OS X)

64-битные системы:

- **LLP64** или **4/4/8** (int и long — 32 бита, указатель — 64 бита): в архитектуре Win64 API
- **LP64** или **4/8/8** (int — 32 бита, long и указатель — 64 бита): в архитектурах Unix и Unix-подобных систем (Linux, Mac OS X)

Другие модели очень редки. Наглядно разница между моделями продемонстрирована в таблице 2.3, в которой приведены размеры целочисленных типов в разных моделях.

Таблица 2.3. Размеры целочесленных типов в разных моделях данных

Имя типа	Размер в битах согласно модели данных				
	Стандарт C++	LP32	ILP32	LLP64	LP64
short int	не меньше чем 16	16	16	16	16
unsigned short int					
int		16	32	32	32
unsigned int					
long int	не меньше чем 32	32	32	32	64
unsigned long int					
long long int	не меньше чем 64	64	64	64	64

2.7.4. Практический пример. Вычисляем размер типов int, float, double и char в вашей системе. Оператор sizeof

Ранее мы использовали только тип int, но кроме него существуют и другие типы данных. Наиболее часто используемыми являются (кроме самого int) float, double и char. Вот только в разных системах (в зависимости от архитектуры системы) размер одной и той же переменной каждого из этих типов может быть разным. Как мы знаем из предыдущего раздела, это ависит от модели данных.

Обычно переменная типа char занимает 1 байт в памяти, тип int - 4 байта, тип float - 4 байта, а double - 8 байтов. Проверим, сколько памяти занимают эти типы данных в вашей системе. Для вычисления размера, занимаемого типом в памяти, используется оператор sizeof.

Листинг 2.5. Вычисляем размер разных типов данных

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

```
int main()
{
    cout << "Размер char: " << sizeof(char) << " byte" << endl;
    cout << "Размер int: " << sizeof(int) << " bytes" << endl;
    cout << "Размер float: " << sizeof(float) << " bytes" << endl;
    cout << "Размер double: " << sizeof(double) << " bytes" <<
endl;

    return 0;
}
```

Результат работы программы показан на рис. 2.4. В зависимости от вашей системы и от вашего компилятора, у вас могут быть другие результаты.

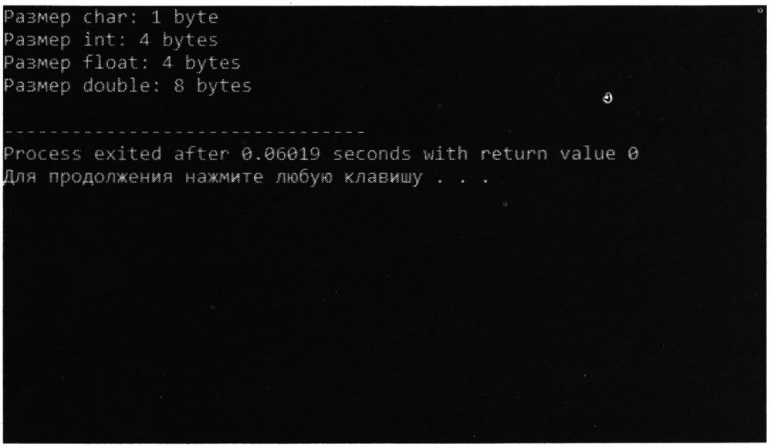


Рис. 2.4. Вычисление размера популярных типов данных

Зачем нужно знать размер типа данных? Представим, что вам нужно сформировать массив из 1 000 000 вещественных чисел. При использовании типа double такой массив займет 8 000 000 байтов или ~7.7 Мегабайта. При 10 миллионах значений это будет уже примерно 77 Мб. Для современных компьютеров 77 Мб - это немного, но и 10 миллионов значений при работе с теми же BigData - это тоже немного, да и не стоит забывать, что массивы вещественных чисел редко кто обрабатывает сами по себе, еще есть какая-то дополнительная информация, например, поясняющая, что означает тот или иной элемент массива - строка, а 10 миллионов строк в памяти могут занимать тоже существенный объем. Если одна строка будет состоять из 8 символов (8 символов типа char = одному значению double по потреблению

памяти), то это еще 77 Мб памяти. Хотя современные операционные системы очень эффективно управляют памятью, не стоит забывать об оптимизации. Например, если не нужна высокая точность, то можно использовать тип float, который занимает памяти в два раза меньше.

2.8. Константы и литералы

В языке программирования C++ под константами и литералами, как правило, подразумевают одно и то же: это такие значения, предназначенные для восприятия пользователем, которые не могут быть изменены в ходе выполнения программы. Приведенное определение относится большей частью к литералу. Под константами здесь и далее будем подразумевать именованные ячейки памяти, значения которых фиксируются на начальном этапе выполнения программы и затем в процессе выполнения программы не могут быть изменены. В этом смысле константы – это те же переменные, но только с фиксированным, определенным единожды, значением.

Для размещения переменной в постоянное запоминающее устройство, превращая их тем самым в константы, используют идентификатор `const`. Идентификатор указывается перед типом переменной и гарантирует неизменность значения переменной. Само значение переменной присваивается либо при объявлении, либо позже в программе, но только один раз и до первого использования переменной в программе.

Например, инструкцией `const int m=5` инициализируется целочисленная переменная `m` со значением `5`. После этого переменную `m` можно использовать в выражениях, однако изменить значение переменной не удастся.

Представление литералов в программном коде существенно зависит от типа, к которому относится литерал. Так, отдельные символы – литералы типа `char` указываются в одинарных кавычках, например `'a'` или `'d'`. Для литералов типа `wchar_t` (напомним, это расширенный 16-битовый символьный тип) перед непосредственно значением указывается префикс `L`. Примеры литералов типа `wchar_t`: `L'a'`, `L'A'`, `L'd'` и `L'D'`.

Несмотря на то, что в C++ нет отдельного типа для строчных (текстовых) переменных, в C++ существуют литералы типа текстовой строки. Соответствующие значения указываются в двойных кавычках. Примером текстового литерала, например, может быть фраза `"Hello, World!"`.

Числовые литералы, как и положено, задаются в стандартном виде с помощью арабских цифр. Однако здесь есть один важный момент. Связан он с тем, что в C++ есть несколько числовых типов, поэтому не всегда очевидно, к какому именно числовому типу относится литерал. Здесь действует общее правило: литерал интерпретируется в пределах минимально необходимого типа. Это правило, кстати, относится не только к числовым литералам. Исключение составляет тип `double`: литералы в формате числа с плавающей точкой интерпретируются как значения этого типа (а не типа `float`, как можно было бы ожидать). Тем не менее, в некоторых случаях необходимо в явном виде указать принадлежность литерала к определенному типу. Обычно это делают с помощью специальных суффиксов. Суффикс `F` после числа в формате с плавающей точкой означает принадлежность литерала к типу `float` (например, `5.3F`). Чтобы литерал относился к типу `long double`, используют суффикс `L` (например, `5.3L`). Этот же суффикс, но у целочисленного литерала означает принадлежность последнего к типу `long int` (например, `123L`). Суффикс `U` используется с целочисленными литералами для отнесения их к типу `unsigned int` (например, `123U`).

2.9. Приведение типов

В качестве примера рассмотрим программу, которая использует приведение типов для нахожения ASCII-кода символа, введенного пользователем. Что такое ASCII-таблица, было сказано в п 2.7.2. Код программы приведен в литинге .

Листинг 2.6. Определение ASCII-значения символа

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

int main()
{
 char c;
 cout << "Введите символ: ";
 cin >> c;
 cout << "ASCII-значение: "  << int(c) << endl;
 return 0;
}
```

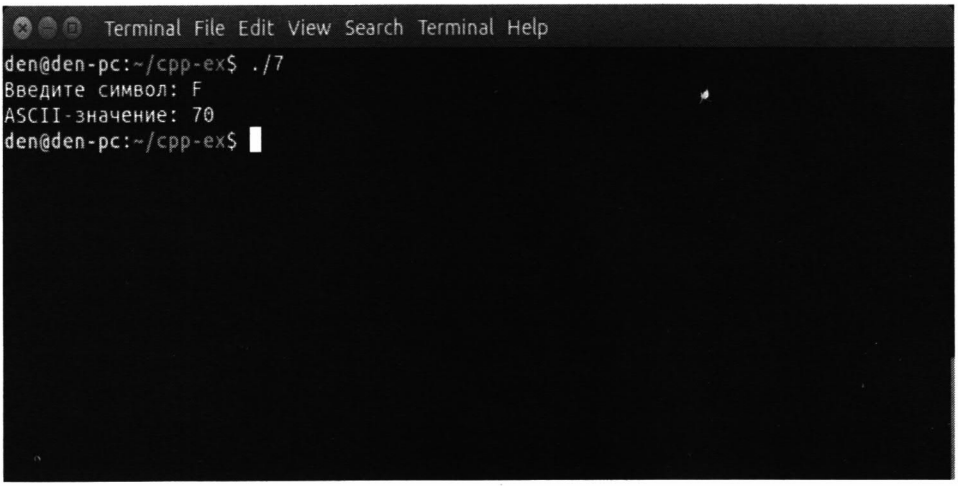
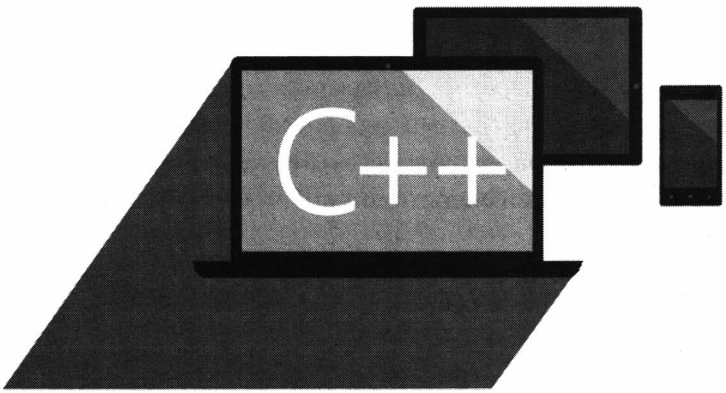


Рис. 2.5. Результат работы программы

Глава 3.

Операторы в языке C++



3.1. Что такое оператор и что такое операнд

Оператор – это наименьшая автономная часть языка, которая выполняет какое-либо действие. Пример оператора – это арифметический оператор сложения +. Еще пример оператора – это оператор присваивания.

То, с чем работает оператор, называется операндами. Например для оператора сложения операндами являются первое и второе слагаемые. **Операнд в языке программирования – это аргумент оператора; данные, которые обрабатываются оператором при осуществении его действия.**

Выражение – это последовательность операторов и операндов, задающее какое-то общее действие. Например, выражением может быть вычисление корня квадратного уровнения. В этом выражении будут операторы (сложения, деления и др.), и операнды – коэффициенты и свободный член квадратного уранения.

3.2. Арифметические операторы в C++

3.2.1. Общее описание

В языке C++ предусмотрены следующие арифметические операторы (см. таблицу 3.1). При этом все приведенные операторы можно разделить на унарные и бинарные. Бинарные операторы – это операторы, которые выполняют действия над двумя операндами (напримрем оператор сложения

выполняет действие над двумя операндами: первым слагаемым и вторым слагаемым). Унарные операторы – это операторы, которые выполняют действия над одним операндом (например, операторы инкремента и декремента, о них мы отдельно поговорим в 3.2.3).

Операция (выражение)		Оператор	Синтаксис выражения	Пример	
Сложение		+	a + b	3 + 6	9
Вычитание		-	a - b	25-5	20
Унарный плюс		+	+a	см. ниже	см. ниже
Унарный минус		-	-a	см. ниже	см. ниже
Умножение		*	a * b	5 * 3	15
Деление		/	a / b	24 / 6 9 / 4	4 2
Операция модуль (остаток от деления целых чисел)		%	a % b	24 / 6 9 / 4	0 1
Инкремент	префиксный	++	++a	см. п. 3.2.3	см. п. 3.2.3
	постфиксный	++	a++	см. п. 3.2.3	см. п. 3.2.3
Декремент	префиксный	--	--a	см. п. 3.2.3	см. п. 3.2.3
	постфиксный	--	a--	см. п. 3.2.3	см. п. 3.2.3

Обратите внимание, что при делении двух целых чисел, получается тоже целое число. То есть 9 / 4 будет не 2.25 , а просто 2. В данном случае, когда делению подлежат два числа типа int, то оператор деления / означает деление нацело. Но если хотя бы один из операндов деления (делимое или

делитель) будет вещественным числом, то и результат будет вещественным числом. То есть:

```
9.0 / 4 = 2.25
9.0 / 4.0 = 2.25
9 / 4.0 = 2.25
9 / 4 = 2
```

3.2.2. Вычисления с помощью программ на C++: практические примеры использования арифметических операторов

Рассмотрим практический пример, в котором мы будеи использовать арифметические операторы. Задача проста: пользователь должен ввести два числа, а наша программа - поменять числа местами. Существует несколько способов решения поставленной задачи. Первый из них заключается в использовании временной переменной такого же типа данных (она называется "пузырьком"). Мы будем использовать временную переменную temp (лист. 3.1).

Листинг 3.1. Первый вариант решения задачи

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

int main()
{
    int a = 5, b = 10, temp;

    cout << "До замены:" << endl;
    cout << "a = " << a << ", b = " << b << endl;

    temp = a;
    a = b;
    b = temp;

    cout << "\nПосле замены:" << endl;
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

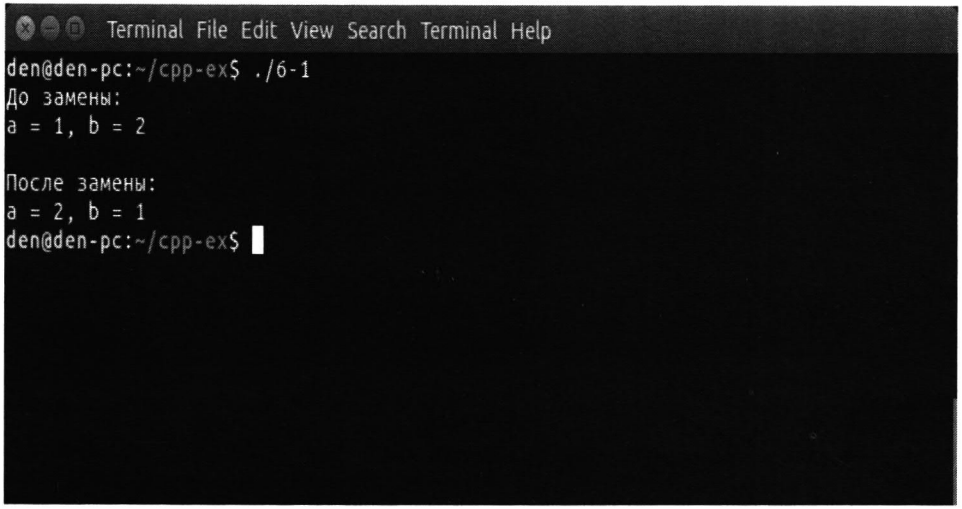


Рис. 3.1. Своп переменных

Думаю, программа в особых комментариях не нуждается. Значение первой переменной мы копируем во временную переменную temp. Далее значение второй переменной копируется в первую переменную, а значение временной переменной копируется обратно, но уже во вторую переменную. На этом все.

Теперь попробуем решить эту задачу математически - без использования еще одной переменной. Поскольку у нас числовые величины, то мы можем это сделать (листинг 3.2).

Листинг 3.2. Замена местами двух числовых переменных

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

int main()
{

    int a = 1, b = 2;

    cout << "До замены:" << endl;
    cout << "a = " << a << ", b = " << b << endl;

    a = a + b;     // a = 1 + 2 = 3
```

```
    b = a - b;      // b = 3 - 2 = 1
    a = a - b;      // a = 3 - 1 = 2

    cout << "\nПосле замены:" << endl;
    // a = 2, b = 1
    cout << "a = " << a << ", b = " << b << endl;

    return 0;
}
```

Здесь решение более хитрое. Сначала мы к первой переменной добавляем вторую, содержимое сохраняем в первой переменной. Затем из первой переменной (мы уже используем новое значение) отнимаем вторую переменную и сохраняем результат во вторую. Наконец, из первой переменной мы вычитаем вторую и сохраняем результат в первой переменной. Вывод у этой программы будет таким же, поэтому дополнительный скриншот не приводится. Напоминаю, что решение работает только с числовыми переменными.

Рассмотрим еще один пример - умножение двух вещественных чисел. Работать программа будет аналогично предыдущей - пользователь вводит два числа, программа вычисляет умножение и выводит результат.

Листинг 3.2. Умножение двух вещественных чисел (8.cpp)

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

int main()
{
    double first, second, product;
    cout << "Введите два числа: ";

    cin >> first >> second;

    product = first * second;

    cout << "Результат = " << product << endl;

    return 0;
}
```

Зачем был нужен этот пример, если был аналогичный, отличающийся только знаком операции. Ранее мы вычисляли сумму двух чисел, а теперь - произведение.

На самом деле разница есть. Во-первых, мы используем другой тип данных - double. Во-вторых, данный пример демонстрирует, как нужно вводить вещественные числа - в качестве разделителя дробной и целой части используется точка, а не запятая.

3.2.3. Операторы инкремента (++) и декремента (--)

Операторы инкремента и декремента являются своеобразоной визитной карточкой языка C++. Эти операторы являются унарными, то есть используются с одним операндом. Действие оператора инкремента ++ заключается в увеличении значения операнда на единицу. Например, результат выполнения оператора i++ есть увеличение значения переменной i на единицу. По сути оператор i++ является эквиалентом выражения i=i+1 с точки зрения влияния на значение переменной.

Действие оператора декремента -- , соответственно, состоит в том, чтобы уменьшить значение операнда на единицу. Результат выполнения оператора i-- есть уменьшение значения переменной i на единицу. По сути оператор i++ является эквиалентом выражения i=i-1 с точки зрения влияния на значение переменной.

Операторы декремента и инкремента имеют одну ключевую особенность: они оба могут использоваться в префиксной или в постфиксной форме. В префиксной форме оператор указывается перед операндом, а в постфиксной форме – после него. В приведенных выше примерах операторы инкремента и декремента использовались в постфиксной форме, то есть оператор ++ или -- стоит после имени переменной. В префиксной форме эти же операторы выглядели бы как ++i и --i.

Результат выполнения дектремента или инкремента, что в префиксной форме, что в постфиксной форме одинаков. Что i++ увеличивает значение переменной i на единицу, что ++i. Аналогично уменьшение значения переменной i на единицу мжно осуществить как оператором --i, так и оператором i--.

Разница между постфиксной и префиксной формами операторов инкремента и декремента проявляется в ситуации, когда эти операторы использованы в выражениях. Как вы знаете, в выражениях вычисления подчинены определенному порядку: например, умножение обладает большим приоритетом по сравнению со сложением и выполняется в первую очередь.

Использование же префиксной или посфиксной формы операторов инкремента и декремента определяет то, что сначала будет выполнено: вычисление выражения или изменение значения переменной, участвующей в выражении и к которой применен оператор инкремента или декремента.

Так вот, если вам надо чтобы сначала было изменено значение переменной, а потом прошло вычисление выражения, то используйте префиксную форму. Если же вам надо сначала вычислить выражение, использовать переменную в выражении, а только потом изменить ее значение, то в таком случае используйте постфиксную форму.

Рассмторим простой пример. Допустим у нас есть такой оператор присваивания:

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

В данном случае сначала значение x увеличивается на 1, а затем результирующее значение присваивается переменной y.

Если же оператор присваивания записать в следующем виде, используя посфиксную форму оператора инкремента:

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

то в этом случае сначала переменной y присваивается значение x, а потом значение x увеличивается на 1.

В листинге приведен расширенный пример вариантов использования инкремента и декрменета в различных формах.

Листинг 3.3. Использование операторов инкремента и декремента

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

```
{
   int a, b,
      i=4, j=4;

   cout<<"Значения переменных в самом начале:\n";
   cout<<"i = "<<i<<"\n";
   cout<<"j = "<<j<<"\n";

   cout<<"Значения переменных после выполнения n=i++ :\n";
   a=i++;
   cout<<"a = "<<a<<"\n";
   cout<<"i = "<<i<<"\n";

   cout<<"Значения переменных после выполнения m=++j :\n";
   b=++j;
   cout<<"b = "<<b<<"\n";
   cout<<"j = "<<j<<"\n";

   cout<<"Значения переменных после выполнения n=(--i)*(i--)
:\n";
   a=(--i)*(i--);
   cout<<"a = "<<a<<"\n";
   cout<<"i = "<<i<<"\n";

   cout<<"Значения переменных после выполнения m=(--j)*(--j)
:\n";
   b=(--j)*(--j);
   cout<<"b = "<<b<<"\n";
   cout<<"j = "<<j<<"\n";

   cout<<"Значения переменных после выполнения n=(--i)*(i++):\n";
   a=(--i)*(i++);
   cout<<"a = "<<a<<"\n";
   cout<<"i = "<<i<<"\n";

   cout<<"Значения переменных после выполнения m=(j--)*(++j)
:\n";
   b=(j--)*(++j);
   cout<<"b = "<<b<<"\n";
   cout<<"j = "<<j<<"\n";
```

```
   cout<<"Значения переменных после выполнения n=(--i)*(++i)
:\n";
   a=(--i)*(++i);
   cout<<"a = "<<a<<"\n";
   cout<<"i = "<<i<<"\n";

   return 0;
}
```

```
Значения переменных в самом начале:
i = 4
j = 4
Значения переменных после выполнения n=i++ :
a = 4
i = 5
Значения переменных после выполнения m=++j :
b = 5
j = 5
Значения переменных после выполнения n=(--i)*(i--) :
a = 12
i = 3
Значения переменных после выполнения m=(--j)*(--j) :
b = 9
j = 3
Значения переменных после выполнения n=(--i)*(i++) :
a = 6
i = 3
Значения переменных после выполнения m=(j--)*(++j) :
b = 9
j = 3
Значения переменных после выполнения n=(--i)*(++i) :
a = 9
i = 3

--------------------------------
Process exited after 0.09959 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

Рис. 3.2. Своп переменных

3.2.4. Операторы "унарный минус" и "унарный плюс"

Операции унарный минус и унарный плюс предназначены для добавления соответствующего знака к переменной, являющейся операндом. Влияние добавления знака на значение переменной осуществляется согласно законам математики: например, если к отрицательному значению добавить минус, то оно станет положительным, а если к отрицательному добавить плюс, то оно так и останется отрицательным.

3.3. Логические операторы

Логические операторы приведены в таблице.

Операция (выражение)	Оператор	Синтаксис выражения
Логическое отрицание, НЕ	!	!a
Логическое умножение, И	&&	a && b
Логическое сложение, ИЛИ	||	a || b

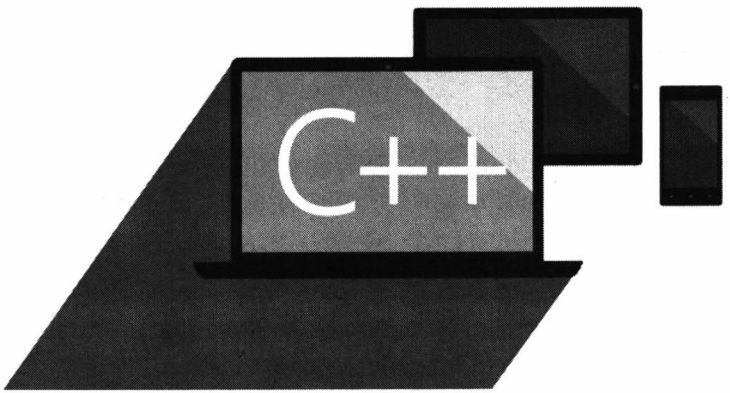
3.4. Операторы сравнения

Операторы сравнения приведены в таблице.

Операция (выражение)	Оператор	Синтаксис выражения
Равенство	==	a == b
Неравенство	!=	a != b
Больше	>	a > b
Меньше	<	a < b
Больше или равно	>=	a >= b
Меньше или равно	<=	a <= b

Глава 4.

Основные правила написания программ на C++



4.1. Алфавит языка C++

Программа на языке C++ может содержать следующие символы:

- прописные (маленькие), строчные (большие) латинские буквы: A, B, C, & , x, y, z и знак подчеркивания;
- арабские цифры от 0 до 9;
- специальные знаки: { } , |, [ ] ( ) + / % * . \ : ? < = > ! & # ~ ; ^
- символы пробела, табуляции и перехода на новую строку.

Из символов алфавита формируют ключевые слова и идентификаторы.

4.2. Правила именования переменных и пользовательских функций

Из символов алфавита формируют ключевые слова и идентификаторы. *Ключевые слова* это зарезервированные слова, которые имеют специальное значение для компилятора и используются только в том смысле, в котором они определены (операторы языка, типы данных и т.п.). *Идентификатор* - это имя программного объекта, представляющее собой совокупность букв, цифр и символа подчеркивания. Имя переменной – это идентификатор. Имя функции – это тоже индентификатор. Соответственно, все приведенные в данном разделе правла в равной степени относятся как именам переменных, так и к именам функций.

Первым символом идентификатора (имени переменной или функции) обязательно должна быть буква или знак подчеркивания, но ни в коем случае не цифра. Идентификатор не может содержать пробел.

Каждое имя(идентификатор) должно быть уникальным в пределах функции – это так называемое правло одного описания. Кроме того, ни одно имя (идентификатор) не должно совпадать с ключевыми словами.

В таблице 4.1. приведен список ключевых слов, зарезервированных в C++. То есть с такими названиями нельзя создавать переменные и новые функции.

Таблица 4.1. Ключевые слова C++

alignas (начиная с C++11)	enum	signed
alignof (начиная с C++11)	explicit	sizeof
and	export	static
and_eq	extern	static_assert (начиная с C++11)
asm	false	static_cast
auto	float	struct
bitand	for	switch
bitor	friend	template
bool	goto	this
break	if	thread_local (начиная с C++11)
case	inline	throw
catch	int	true
char	long	try
char16_t (начиная с C++11)	mutable	typedef
char32_t (начиная с C++11)	namespace	typeid
class	new	typename
compl	noexcept (начиная с C++11)	union
const	not	unsigned
constexpr (начиная с C++11)	not_eq	using
const_cast	nullptr (начиная с C++11)	virtual
continue	operator	void
decltype (начиная с C++11)	or	volatile
default	or_eq	wchar_t
delete	private	while
do	protected	xor
double	public	xor_eq
dynamic_cast	register	
else	reinterpret_cast	
	return	
	short	

4.3. Использование больших и маленьких букв

Язык программирования C++ является регистрозависимым. Это значит, что прописные (большие) и строчные (маленькие) буквы в именах различаются. Например, parameter, Parameter, PARAMETER, parameter – это четыре разных имени!

Соотвественно, если вы в коде своей программы напишете не main(), а Main(), то это будет ошибкой. Потому что в C++ определено, что такое есть функция main(), но ничего не сказано про функцию Main(), которую вы можете при желании создать сами.

4.4. Управляющие последовательности

Управляющие последовательности используются для описания определённых специальных символов внутри строковых литералов. В языке C++ могут быть использованы управляющие последовательности, приведенные в таблице 4.2.

Таблица 4.2. Управляющие последовательности C++

Управляющая последовательность	Описание
\'	одинарная кавычка
\"	двойная кавычка
\?	вопросительный знак
\\	обратный слеш
\0	нулевой символ
\a	звуковой сигнал
\b	забой
\f	перевод страницы - новая страница
\n	перевод строки - новая строка
\r	возврат каретки

\t	горизонтальная табуляция
\v	вертикальная табуляция
\nnn	произвольное восьмеричное значение
\xnn	произвольное шестнадцатеричное значение
\unnnn	произвольное Юникод-значение. Результатом могут быть несколько символов.
\Unnnnnnnn	произвольное Юникод-значение. Результатом могут быть несколько символов.

4.5. Указание точки с запятой (;) после операторов

Описание каждого оператора или блока операторов в C++ должно заканчиваться точкой с запятой.

4.6. Использование комментариев

В коде программ, написанных на языке програмирования C++ можно использовать комментарии. Информация, оформленная в коде как комментарии, не обрабатывается компилятором. Поэтому вы можете в качестве комментариев добавлять любой текст. Обычно это делается для того, чтобы дать какие-либо пояснения по коду программы, а комментарии в самом начале программы обычно описывают, для чего программа предназначена. Вы можете добавлять неограниченное количество комментариев.

В языке C++ комментарии можно задавать двумя способами:

- С помощью символа // . Его применение означает, что следующая после него информация в строке — комментарий. Этот способо удобен для однострочных комментариев. Если текст комментария у вас перешел на следующую строку, то необходимо перед ним также поставить // на следующей строке. Например:

```
// Это комментарий
// Это вторая строка комментария
```

```
int i;   / / Это комментарий, а слева - объявление переменной i
```

- С помощью конструкции /* и * /. Весь текст, содержащийся между символами /* и * / является комментарием, даже если он занимает несколько строк:

```
/* Это комментарий,
   занимающий две строки * /
```

При этом комментарием считается все, что расположено в строке справа от символов //. Точку с запятой ставить после комментариев не надо.

4.7. Строковые значения, использование двойных кавычек

Символьные значения должны заключаться в одинарные кавычки. Например:

```
char a;
a = 'B';
```

Строковые значения должны заключаться в двойные кавычки. Иначе говоря, все находится между двумя двойными кавычками воспринимается как строка. Строки мы, например, использовали для вывода сообщений на экран, когда посылали их в поток вывода cuot.

Например:

```
cout<<"Значения переменных после выполнения n=i++ :\n";
```

4.8. Составной оператор, использование фигурных скобок {}

Если вам необходимо какой-либо набор операторов в программе объединить в отдельный блок, то для этого используйте фигурные скобки { }.

Объединение операторов в блоки используется в тех случаях, когда вам нужно структурно выделить какой-либо набор операторов с целью единого управления этим набором. Например, когда нужно показать, что этот набор

выполняется при таком-то условии (об условиях в программе речь идет в следующей главе). В виде блока оформляется тело функции, в том числе и функции main(). И т.д.

4.9. Указание пространства имен. Или что означается std::cout

Ранее мы с вами узнали, что такое пространство имен и научились подключать пространства имен к программе с помощью директивы `using namespace`. Например:

```
using namespace std;
```

Так мы подключили все стандартное пространство имен std. Только после его подключения мы можем пользоваться объектами cout и cin для вывода и ввода данных. Но есть еще другой подход, когда пространсво имен не подключается, а индивидуально указывается для каждого имени. Делается это с помощью оператора :: (два двоеточия подряд).

В листинге 4.1 приведен код программы с поключением пространства имен с помощью директивы, а в листинге 2 – с индивидуальным указанием пространства для каждого имени, для которого это требуется

Листинг 4.1. Подключение пространства имен с помощью директивы using namespace

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

int main()
{
    double first, second, product;
    cout << "Введите два числа: ";

    cin >> first >> second;

    product = first * second;

    cout << "Результат = " << product << endl;
```

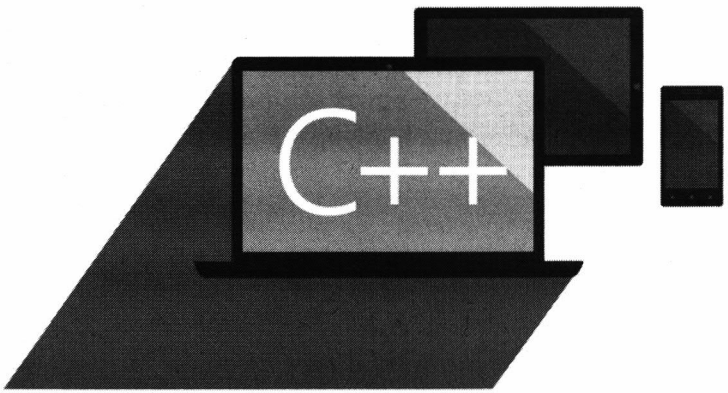
```
    return 0;
}
```

Листинг 4.2. Индивидуальное указание пространства имен

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

int main()
{
    double first, second, product;
    std::cout << "Введите два числа: ";

    std::cin >> first >> second;

    product = first * second;

    std::cout << "Результат = " << product << endl;

    return 0;
}
```

Глава 5.

Стандартные управляющие конструкции языка C++



Под управляющими конструкциями в программировании обычно понимют такие операторные конструкции, которые определяют или изменяют ход программы. Стандартный ход программы, в которой отсутствуют управляющие конструкции, идет последовательно, от одного оператора к другому, последовательно переходя от одного оператора к другому пока не будет выполнен последний оператор и достигнут конец программы.

В то же время могут возникнуть ситуации, когда надо будет выполнить только одно из нескольких действий, или возвращться и выполнять какой-либо набор действий пока не будет достигнут определенный результат. Вот для таких случаев и предназначены управляющие конструкции. Далее в этой главе мы и займемся рассмотрением таких конструкций, имеющихся в языке C++.

5.1. Условные операторы

Условные операторы – это операторы, которые в завивимости от определенного условия или условий направляют ход программы. В языке программирования C++ имеется два условных оператора: if и switch. Сначала рассмотрим оператор if, а потом оператор switch.

5.1.1. Условный оператор if

Логика работы оператора if

Оператор if – это оператор, который проверяет определенное условие и, в зависимости от того истинно оно или ложно, выполняет или не выполняет определенные действия.

Общий вид условного оператора таков:

```
if (условие) {блок операторов 1}
        else {блок операторов 2};
```

После имени оператора if в круглых скобках указывается условие, истинность которого проверяется. Для формирования условий предназначены логические операторы и операторы сравнения, рассмотренные ранее.

Логика выполнения условного оператора такова:

1. Проверяется условие
2. Если условие истинно, то выполняется блок операторов 1
3. Если условие ложно, то выполняется блок операторов 2 (то есть тот, что идет после ключевого слова else).

В языке C++ допускается использование сокращенного варианта условного оператора, у которого отсутвует альтернативная ветка else и блок 2. Такая упрощенная конструкция используется тогда, когда надо просто указать действия, которые выполняются при определенном условии.

Выглядит сокращенный оператор так:

```
if(условие) {блок операторов 1};
```

Логика выполения сокращенного условного ператора такова:

1. Проверяется условие
2. Если условие истинно, то выполняется блок операторов 1
3. Если условие ложно, то выполнение оператора if завершается и управление передается следующему за ним оператору.

Если какой-либо блок операторов состоит из одного оператора, то заключать его в фигурные скобки не обязательно. То есть возможна такая запись:

```
if(условие) оператор 1
        else оператор 2;
```

Рассмотрим несколько примеров.

Практический пример: проверка на четность

В качестве первого примера рассмотрим программу, которая проверяет на четность введенное пользователем число. Для проверки, является ли число четным, используем оператор %, позволяющий узнать остаток от деления. Как вы знаете, число является четным, если оно делится на 2 без остатка, например, 0, 8, 10, -50. А нечетное число не делится на 2 без остатка, например, 1, 3, 9, 11.

Код примера приведен в листинге 5.1. А пример выполнения программы показан на рис. 5.1.

Листинг 5.1. Является ли число четным

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

int main()
{
    int n;

    cout << "Введите число: ";
    cin >> n;

    if ( n % 2 == 0)
        cout << n << " - четное.";
    else
        cout << n << " - нечетное.";

    count << endl;
    return 0;
}
```

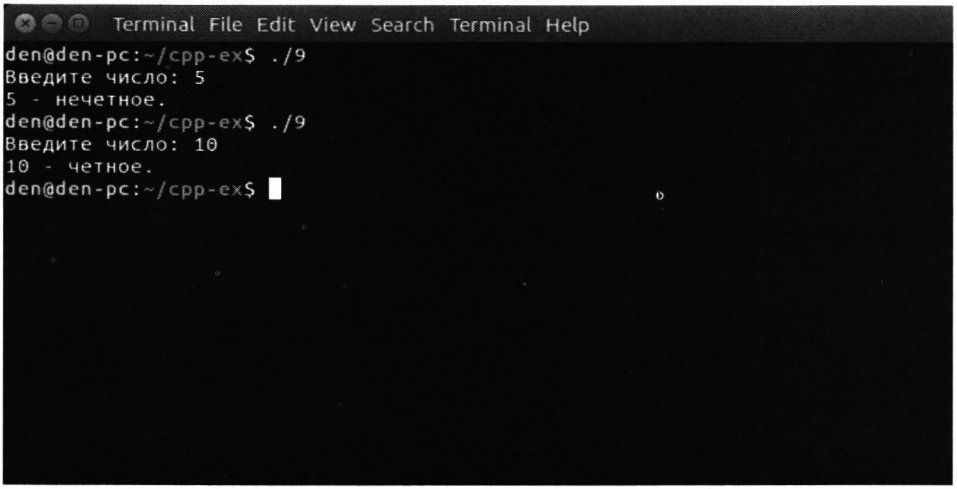


Рис. 5.1. Программа, определяющая, является ли число четным или нет

Практический пример: нахождение максимума

В следующем примере мы определим максимум среди трех чисел. При этом продемонстрируем использование логического оператора && (логическое И). Данный оператор возвращает true, если оба его операнда истинны. Например:

```
a && b
```

Вернет true, если a = true И b = true. В противном случае оператор возвращает false. Зная, как работает этот оператор, написать нашу программу труда не составит. Код программы приведен в листинге 5.2.

Листинг 5.2. Максимум среди трех чисел (10.cpp)

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

int main()
{
    int a, b, c, max;
    cout << "Введите три целых числа: ";
    cin >> a >> b >> c;

    if( a>=b && a>=c )
       max = a;
```

```
    if( b>=a && b>=c )
       max = b;

    if( c>=a && c>=b )
       max = c;

    cout << "Max: " << max;
    cout << "\n";

    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./10
Введите три целых числа: 100 234 22
Max: 234
den@den-pc:~/cpp-ex$ ./10
Введите три целых числа: 434 22 122
Max: 434
den@den-pc:~/cpp-ex$ ./10
Введите три целых числа: 99 1 1211
Max: 1211
den@den-pc:~/cpp-ex$
```

Рис. 5.2. Программа в действии (10.cpp)

Практический пример: вычисление корней квадратного уравнения

Следующий пример добавит немного математики. Стандартная форма квадратного уравнения выглядит так:

```
ax^2 + bx + c = 0,
где a, b, c - вещественные числа, и a - не равно 0.
```

Термин b^2 - 4ac также известен как детерминант квадратного уравнения. Детерминант позволяет определить природу корней:

- Если детерминант больше 0, корни являются вещественными и они разные. Корней два.

- Если детерминант равен 0, корни вещественные и одинаковые. Корень, по сути, один.

- Если детерминант меньше 0, корни являются комплексными и разными. Всего корней 2.

Если вы забыли курс математики, освежить знания можно на странички Википедии **https://ru.wikipedia.org/wiki/Квадратное_уравнение**. А мы же приступим к написанию кода программы (лист. 5.3).

Листинг 5.3. Вычисляем все корни квадратного уравнения

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

int main() {

    float a, b, c, x1, x2, discriminant, realPart, imaginaryPart;
    cout << "Введите коэффициенты a, b и c: ";
    cin >> a >> b >> c;
    discriminant = b*b - 4*a*c;

    if (discriminant > 0) {
        x1 = (-b + sqrt(discriminant)) / (2*a);
        x2 = (-b - sqrt(discriminant)) / (2*a);
        cout << "Корни являются вещественными и они разные" << endl;
        cout << "x1 = " << x1 << endl;
        cout << "x2 = " << x2 << endl;
    }

    else if (discriminant == 0) {
        cout << "Корни вещественные и одинаковые" << endl;
        x1 = (-b + sqrt(discriminant)) / (2*a);
        cout << "x1 = x2 =" << x1 << endl;
    }

    else {
        realPart = -b/(2*a);
        imaginaryPart =sqrt(-discriminant)/(2*a);
        cout << "Корни являются комплексными и разными"  << endl;
```

```
        cout << "x1 = " << realPart << "+" << imaginaryPart <<
"i" << endl;
        cout << "x2 = " << realPart << "-" << imaginaryPart <<
"i" << endl;
    }

    return 0;
}
```

Работает программа довольно просто: сначала мы вычисляем детерминант, а затем вычисляем корни по формуле. Корни вычисляются по-разному в зависимости от значения детерминанта.

Данная программа интересна не столько работой условного оператора, сколько математической библиотекой cmath. Для вычисления квадратного корня мы используем функцию sqrt(). Чтобы эта функция стала доступна, мы подключаем заголовочный файл cmath.

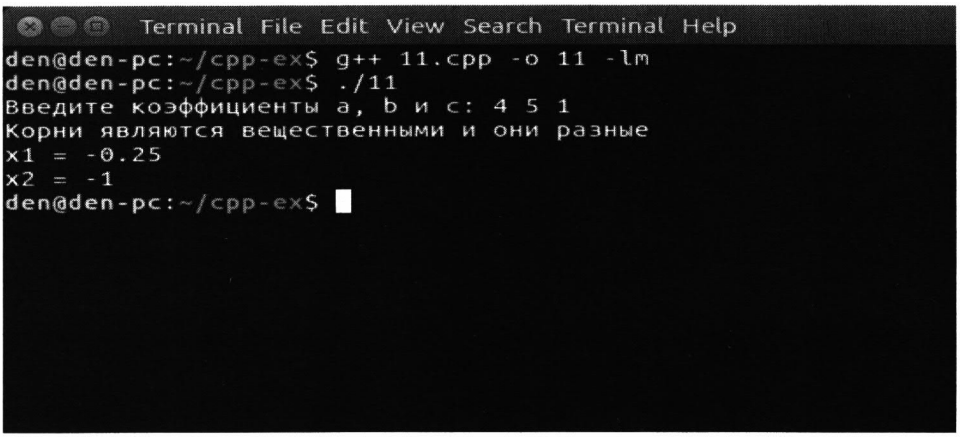


Рис. 5.3. Программа в действии

Вложенные условные операторы

В языке программирования C++ допускается использовать вложенные условные операторы. Вложенность означает, что внутри одного оператора if вы можете использовать другой оператор if в составе любого из блоков операторов.

Пример вложенных операторов:

```
if (условие)
      if (условие)
            оператор 1
  else оператор 2
else оператор 3;
```

Рассмотрим это дело на примере. В качестве напишем программу, которая определяет, является ли високосным год, введенный пользователем с клавиатуры.

Високосные года (в которых есть 29 февраля и соответственно - 366 дней) - это те, которые делятся на 4 без остатка: 2004, 2008, 2012, 2016, 2020, 2024.

Однако в григорианском католическом календаре, по которому мы ныне живем ("новый стиль") есть еще редкое и малоизвестное правило: те года, которые нацело делятся на 100 (т.е. оканчиваются на -00) и которые делятся нацело на 400 - високосные, а которые делятся с остатком - не високосные.

Поэтому 1700, 1800, 1900 года - были невисокосными (хотя и делятся нацело на 4), 2000 - был високосным как обычно (делится нацело на 400), 2100, 2200, 2300 - также будут невисокосными.

Как видите, держать всю эту информацию в уме, сложно, поэтому давайте напишем программу, проверяющую, является ли год високосным (лист. 5.4).

Листинг 5.4. Високосный год?

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

```
int main()
{
    int year;

    cout << "Введите год: ";
    cin >> year;

    if (year % 4 == 0)
    {
        if (year % 100 == 0)
        {
            if (year % 400 == 0)
                cout << year << " - високосный";
            else
                cout << year << " - невисокосный";
        }
        else
            cout << year << " - високосный";
    }
    else
        cout << year << " - невисокосный";

    cout << endl;
    return 0;
}
```

5.1.2. Оператор множественного выбора switch

Логика работы оператора switch

Оператор switch предназначен для сравнения результата определенного выражения с конечным множеством значений и выполнения какого-либо действия в зависимости от того, какому из сравниваемых значений оказался равен результат выражения. Поэтому оператор switch и называется оператором множественного выбора. По своей логике оператор switch является альтернативой нескольких вложенных операторов if.

Общий вид оператора множественного выбора таков:

```
switch (выражение) {
      case значение1:
      оператор;
      ….
            break;
      case значение2:
            оператор;
            ……
            break;
      ...
      default:
            операторы
      }
```

После ключевого слова switch в круглых скобках указывается выражение, результат которого будет сравниваться. При этом есть одно ограничение: в качестве результата выражения может быть только целое число или символ. Результат выражения последовательно сравнивается со значениями, указанными после ключевых слов case. Как только произошло совпадение, тут же начинается выполнение соответствующего набора операторов, после чего выполнение оператора switch завершается, проверка других совпадений не производится.

Если же никаких совпадений обранужено не было, то выполняются операторы, идущие после ключевого слова default и до фигурной скобки, закрывающей оператор switch. Наличие секции default является необязательным. Еслии секции default нет, то оператор просто завершает свое выполнение и управление переходит к следующему за switch оператору.

Пример использование оператора switch: пишем простой калькулятор на C++

В качестве примера напишем простой калькулятор, который может складывать, вычитать, умножать и делить два числа. Алгоритм программы будет такой. Сначала пользователь вводит оператор (например, *), затем два операнда - числа. После этого программа возвращает результат операции.

Оператор выбора switch .. case позволяет произвести действия (в нашем случае - математические) в зависимости от переданного ему значения. Оператор break внутри case нужен, чтобы прервать выполнение оператора switch - ведь нам нужно выполнить только одной из действий, а не все нижестоящие после того, как было встречено совпадение.

Листинг 5.5. Простой калькулятор

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

int main()
{
    char op;
    float num1, num2;

    cout << "Введите оператор (+ - * /) ";
    cin >> op;

    cout << "Введите операнды: ";
    cin >> num1 >> num2;

    switch(op)
    {
        case '+':
            cout << num1+num2;
            break;

        case '-':
            cout << num1-num2;
            break;

        case '*':
            cout << num1*num2;
            break;

        case '/':
            if (num2 != 0)
               cout << num1/num2;
            else
               cout << "Divizion by zero!";
```

```
            break;

        default:
            // Если оператор отличается от + - * /
            cout << "Неправильный оператор!";
            break;
    }
    cout << endl;
    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) +
Введите операнды: 100 20
120
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) /
Введите операнды: 9 0
Divizion by zero!
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) *
Введите операнды: 5 20
100
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) /
Введите операнды: 100 4
25
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) -
Введите операнды: 2 2
0
den@den-pc:~/cpp-ex$
```

Рис. 5.4. Калькулятор в действии

На рис. 5.4 показано тестирование нашего калькулятора. Продемонстрировано, как программа реагирует на деление на 0, на ввод некорректного оператора.

5.2. Операторы цикла

Операторы цикла (или их еще иногда называют операторами зацикливания) позволяют многократно выполнять заданный блок операторов в зависимости от определенного условия.

В языке C++ предусмотрено три оператора цикла:

- Оператор for
- Оператор while
- Оператор do while

Далее мы их всех и рассмотрим в данном разделе.

5.2.1. Цикл for

Логика работы цикла for

Оператор цикла for еще называют циклом-счетчиком. Его удобно использовать, когда мы знаем, сколько итераций (повторений) должно быть. При этом в начале выполнения цикла счетчику присваивается определенное значение, по ходу выполнения цикла, с каждым повтором (каждой новой итерацией) значение счетчика изменяется (обычно увеличивается), а при достижении определенного значения счетчика цикл завершает свое выполнение.

Общий вид цикла for таков:

```
for (оператор_1; условие; оператор_2)
{
  Повторяющийся блок операторов (тело_цикла)
}
```

- `Оператор_1` – это оператор инициализации, в котором инициализируется переменная-счетчик цикла с указанием его начального значения. Данный оператор выполняется только однажды.

- `Условие` – это некоторое условие, в котором участвует переменная-счетчик. Если данное условие истинно, то операторы в теле цикла будут выполнены. Если же условие ложно, то цикл завершает свое выполнение.

- `Оператор_2` – это оператор, который изменяет значение переменной счетчика.

Логика выполнения оцикла for следующая:

1. Выполняется Оператор_1 – инициализируется переменная-счетчик.

2. Проверяется Условие.

3. Если Условие истинно, то выполняется Тело цикла (операторы в телецикла). Если условие ложно, то цикл завершается и управление переходит к следующему за циклом for оператору.

4. Выполняется Оператор_2 – изменяется значение переменной-счетчика.

5. Снова проверяется Условие

6. и т.д.

Рассмотрим цикл for на практическом примере. Давайте напишем программу, которая вычисляет сумму всех натуральных чисел 1 до n, а значение n пусть введет пользователь.

Вспомним, что натуральные числа - это целые положительные числа начиная с 1. Наша программа будет работать так: пользователь вводит число n, а программа вычисляет сумму чисел от 1 до n в цикле.

Код программы, использующий цикл for, приведен в листинге 5.6.

Листинг 5.6. Вычисляем сумму натуральных чисел с помощью цикла for

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

int main()
{
    int n, sum = 0;

    cout << "Введите положительное целое число: ";
    cin >> n;

    for (int i = 1; i <= n; ++i) {
        sum += i;
    }

    cout << "Сумма = " << sum << endl;
    return 0;
}
```

Первым делом цикл присваивает переменной i значение 1 - он инициализирует счетчик (нам нужно вычислить сумму от 1 до n). Далее цикл проверяет, истинно ли условие - если пользователь ввел 3 в качестве n, да, условие истинно, поскольку i (1) меньше 3. Если условие истинно, выполняется тело цикла, а именно к переменной sum добавляется значение переменной i. И так до тех пор, пока i не станет больше n (у нас условие выполнения цикла меньше или равно).

Вложенные циклы for

Циклы for могут быть вложенными: это когда в теле одного цикла for используется другой цикл for. Данное правило, кстати говоря, распространяется и на другие циклы, которые мы рассмотрим в следующих подразделах.

Рассмотрим пример.Давайте напишем программу, которая выводит пирамиду, состоящую из *. Для решения данной задачи мы будем использовать два вложенных цикла for. Первый будет "считать" строки, а второй - выводить звездочки в каждой строке.

Листинг 5.7. Создаем пирамиду из звездочек (*)

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

int main()
{
    int rows;

    cout << "Введите количество строк: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << "* ";
        }
        cout << "\n";
    }
    return 0;
}
```

Рис. 5.5. Пирамида из звездочек

5.2.2. Цикл while

Цикл while называют еще циклом ПОКА. Это из-за логики работы цикла, которая очень проста: заданный набор операторов (тело цикла) будет выполнться до тех пор, ПОКА истинно заданное условие. Как только условие станет ложным, цикл while завершит свое выполнение.

Общий вид цикла while таков:

```
while (условие)
{
  тело_цикла;
}
```

В качестве примера приведем решение уже знакомой задачи по суммированию натуральных числе из предыдущего раздела, но теперь с использование цикла while.

Листинг 5.8. Сумма натуральных чисел с помощью цикла while()

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

int main()
{
    int n, i, sum = 0;

    cout << "Введите целое число ";
    cin >> n;

    i = 1;
    while ( i <=n )
    {
        sum += i;
        ++i;
    }

    cout << "Сумма = " << sum << endl;

    return 0;
}
```

В отличие от цикла for, мы инициализируем счетчик i до цикла, затем увеличиваем его в теле цикла. При написании подобных циклов главное не забыть увеличить переменную-счетчик в теле цикла, иначе цикл будет выполняться вечно.

5.2.3. Цикл do while

Цикл do..while в отличие от цикла while сначала выполняет тело цикла, а затем уже проверяет условие. Поэтому цикл do..while еще называется циклом с постусловием. Его удобно использовать, когда нужно, чтобы тело цикла гарантированно выполнилось хотя бы один раз. Давайте доработаем программу подсчета натуральных чисел из предыдущих разделов так, чтобы она запрашивала у пользователя ввод числа, пока он не введет целое положительное число (так мы исключим неправильные ситуации, когда пользователь вводит отрицательное число). Далее вычислим сумму с помощью цикла for. Код программы в листинге 5.9.

Листинг 5.9. Демонстрация цикла do..while

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

int main()
{
    int n, i, sum = 0;

    do {
        cout << "Введите целое положительное число: ";
        cin >> n;
    }
    while (n <= 0);

    for(i=1; i <= n; ++i)
    {
        sum += i;    // sum = sum+i;
    }

    cout << "Сумма = " << sum << endl;
    return 0;
}
```

5.3. Сместное использование операторов цикла и условных операторов

Условные операторы и операторы цикла могут использоваться совместно друг с другом и быть вложенными друг в друга. Продемонстрируем это на примере: напишем программу, которая выводит заданное количество чисел Фибоначчи.

Числа Фибоначчи — элементы последовательности целых чисел, в которой каждое последующее число равно сумме двух предыдущих чисел. Начинается последовательность с 0 и 1. В этой программе мы используем две переменных t1 и t2 для хранения двух предыдущих чисел.

Листинг 5.10. Вычисление последовательности Фибоначчи

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

int main()
{
    int n, t1 = 0, t2 = 1, nextTerm = 0;

    cout << "Введите количество элементов последовательности: ";
    cin >> n;

    cout << "Последовательность Фибоначчи: ";

    for (int i = 1; i <= n; ++i)
    {
        // Выводим первый элемент
        if(i == 1)
        {
            cout << " " << t1;
            continue;
        }
        if(i == 2)
        {
            cout << t2 << " ";
            continue;
        }
```

```
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;

        cout << nextTerm << " ";
    }
    cout << endl;
    return 0;
}
```

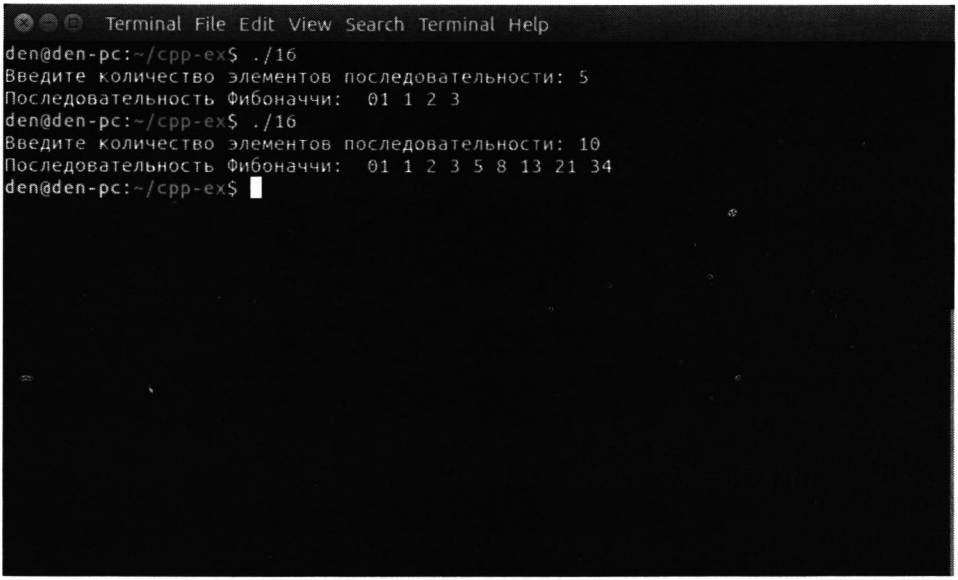


Рис. 5.6. Генерирование последовательности Фибоначчи

Задача вычисления последовательности Фибоначчи иногда ставится не так. В предыдущем примере мы генерировали последовательность, задавая количество ее членов. А иногда просят сгенерировать последовательность до определенного числа. В этом случае код будет таким:

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

int main()
{
    int t1 = 0, t2 = 1, nextTerm = 0, n;
```

```
    cout << "Введите положительное число: ";
    cin >> n;

    cout << "Последовательность Фибоначчи: " << t1 << ", " << t2 << ", ";

    nextTerm = t1 + t2;

    while(nextTerm <= n)
    {
        cout << nextTerm << " ";
        t1 = t2;
        t2 = nextTerm;
        nextTerm = t1 + t2;
    }
    cout << endl;
    return 0;
}
```

Вывод программы будет таким:

```
Введите целое положительное число: 60
Последовательность Фибоначчи: 0 1 1 2 3 5 8 13 21 34 55
```

5.4. Дополнительные практические примеры использования циклов и условных операторов

Пример: нахождение наибольшего общего делителя

Наибольший общий делитель (НОД) двух чисел - это максимальное число, на которое могут быть без остатка разделены оба числа. Пример: для чисел 70 и 105 наибольший общий делитель равен 35.

Существует множество способов определить НОД программно. Первый способ заключается в использовании цикла for. В нем мы перебираем делители в порядке возрастания: если будет найден такой, на который делятся без остатка оба числа, мы будем считать его общим делителем.

Поскольку делители перебираются в порядке возрастания, то последний общий делитель будет наибольшим.

Первый способ решения приведен в листинге 5.11.

Листинг 5.11. Определяем НОД с помощью цикла for

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

int main() {
    int n1, n2, hcf;
    cout << "Введите два числа: ";
    cin >> n1 >> n2;

    // Меняем местами переменные n1 и n2, если n2 > n1
    if ( n2 > n1) {
        int temp = n2;
        n2 = n1;
        n1 = temp;
    }

    for (int i = 1; i <=  n2; ++i) {
        if (n1 % i == 0 && n2 % i ==0) {
            hcf = i;
        }
    }

    cout << "НОД = " << hcf << endl;
    return 0;
}
```

Для определения НОД мы можем использовать и цикл while (лист. 5.12).

Листинг 5.12. Определение НОД с помощью цикла while

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

int main()
{
    int n1, n2;
```

```
    cout << "Введите два числа: ";
    cin >> n1 >> n2;

    while(n1 != n2)
    {
        if(n1 > n2)
            n1 -= n2;
        else
            n2 -= n1;
    }

    cout << "НОД = " << n1 << endl;
    return 0;
}
```

В прошлых примерах мы подразумевали, что введенные пользователем числа будут положительными. А что будет, если пользователь введет отрицательные числа? Тогда мы можем очень легко поменять знак на положительный и вычислить НОД:

```
// Если введены отрицательные числа, меняем знак на положительный
    n1 = ( n1 > 0) ? n1 : -n1;
    n2 = ( n2 > 0) ? n2 : -n2;
```

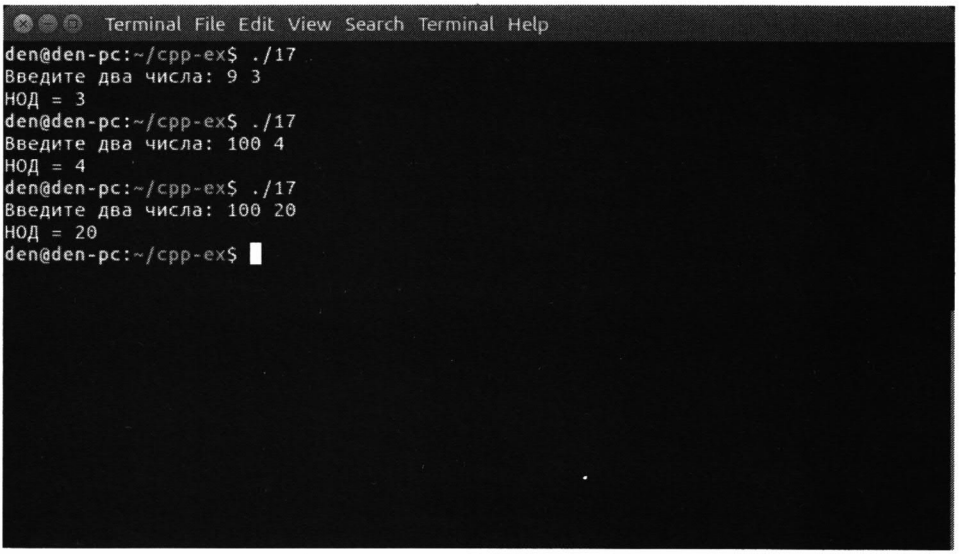


Рис. 5.7. Вычисление НОД

Пример: нахождение наименьшего общего кратного

Наименьшее общее кратное (НОК)[1] двух целых чисел m и n есть наименьшее натуральное число, которое делится на m и n без остатка. Как и в предыдущем примере, есть несколько способов вычислить НОК. Первый способ решения - с помощью деления. Этот способ приведен в листинге 5.13. Для разнообразия напишем этот пример на C++.

Листинг 5.13. Определение НОК с помощью бесконечного цикла

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

int main()
{
    int n1, n2, LCM;
    cout << "Введите два целых числа: ";
    cin >> n1 >> n2;

    // определяем максимум сред n1 и n2 и сохраняем в LCM
    LCM = (n1>n2) ? n1 : n2;

    // Бесконечный цикл
    while(1)
    {
        if( LCM%n1==0 && LCM%n2==0 )
        {
            cout << "НОК = " << LCM << "\n";
            break;
        }
        ++LCM;
    }
    cout << endl;
    return 0;
}
```

Особенность этого метода в том, что мы используем бесконечный цикл while. В этом и заключается опасность программы. Если вы не предусмотрите (или неправильно предусмотрите) условие выхода из цикла, то программа

1 Или least common multiple (LCM) - в анг. литературе - чтобы было понятно, откуда было взято название переменной LCM

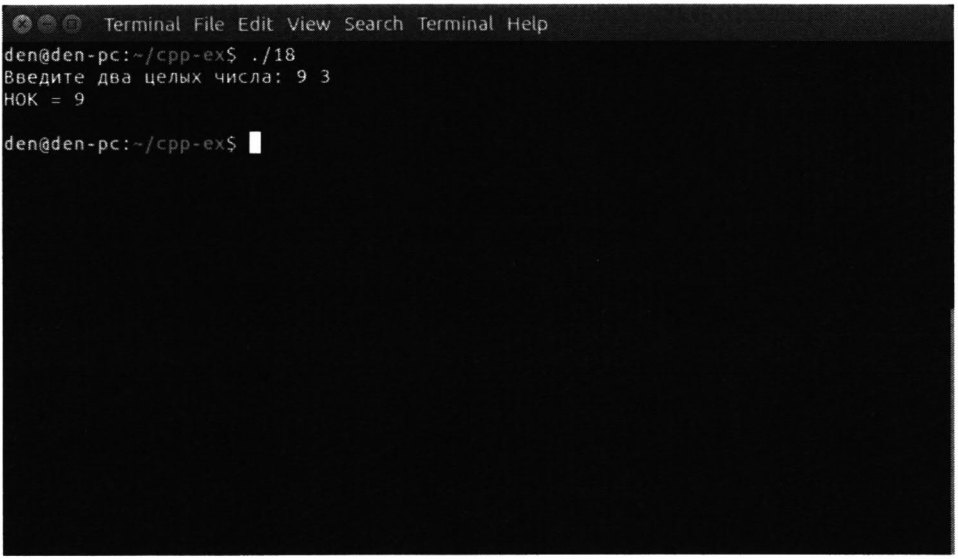


Рис. 5.8. Вычисление НОК двух чисел

"зациклится" - цикл будет выполняться бесконечно. Если такое произошло, нажмите Ctrl + C для аварийного завершения программы.

В нашем случае условием выхода из цикла является нахождение НОК, после чего выполняется оператор break, который и закрывает цикл while.

Новичкам я бы не рекомендовал в своих программах использовать бесконечные циклы, поэтому рассмотрим второй вариант. Если вы помните школьный курс математики, то вычислить НОК можно так:

```
НОК = (n1*n2)/НОД;
```

То есть сначала можно вычислить НОД, а затем разделить на него результат умножения двух чисел. Такой вариант гораздо безопаснее и не приведет к зацикливанию программы, поскольку используется цикл while. Второй вариант определения НОК приведен в листинге 5.14.

Листинг 5.14. Определение НОК через НОД

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

int main()
```

```
{
    int n1, n2, hcf, temp, lcm;

    cout << "Введите два числа: ";
    cin >> n1 >> n2;

    hcf = n1;
    temp = n2;

    while(hcf != temp)
    {
        if(hcf > temp)
            hcf -= temp;
        else
            temp -= hcf;
    }

    lcm = (n1 * n2) / hcf;

    cout << "НОК = " << lcm;
    return 0;
}
```

Пример: подстчет количества цифр целого числа

Попробуем вычислить количество цифр в целом числе. Например, если пользователь введет 1983, то программа выведет 4. Для этого мы будем последовательно делить число на 10 и подсчитывать, сколько раз мы этого сделали. Количество операций делений и будет равно количеству цифр.

Листинг 5.15. Определяем количество цифр в целом числе

```
#include <iostream>
using namespace std;
int main()
{
    long long n;
    int count = 0;

    cout << "Введите целое число: ";
```

```
    cin >> n;

    while(n != 0)
    {
        // n = n/10
        n /= 10;
        ++count;
    }

    cout << "Количество разрядов: " << count << endl;
}
```

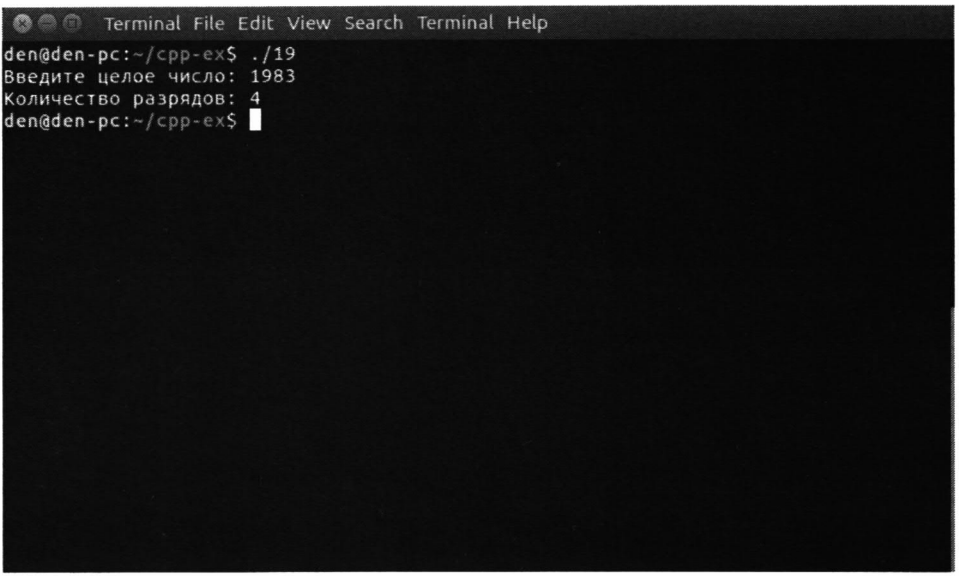


Рис. 5.9. Определяем количество цифр

Пример: вычисление обратного числа

Напишем программу, которая вычисляла бы обратное целое число, например, если пользователь вводит 1234, то программа возвращает 4321.

Алгоритм работы программы такой:

- В цикле мы вычисляем остаток от деления числа на 10
- Постепенно формируем обратное число - постепенно умножаем его на 10 и добавляем остаток от предыдущей операции деления

Листинг 5.16. Вычисляем обратное число

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

int main()
{
    int n, reversedNumber = 0, remainder;

    cout << "Введите целое число: ";
    cin >> n;

    while(n != 0)
    {
        remainder = n%10;
        reversedNumber = reversedNumber*10 + remainder;
        n /= 10;
    }

    cout << "Обратное число = " << reversedNumber << endl;

    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./20
Введите целое число: 7757
Обратное число = 7577
den@den-pc:~/cpp-ex$ ./20
Введите целое число: 1375
Обратное число = 5731
den@den-pc:~/cpp-ex$
```

Рис. 5.10. Вычисляем обратное число

Пример: палиндром

Палиндром - это число, которое одинаково выглядит в прямом и обратном направлении, например, 121, 53235 - это числа палиндромы. Напишем программу, которая будет определять, является ли введенное пользователем число палиндромом или нет.

Работать программа будет так: сначала она вычислит обратное число, а потом сравнит его с исходным числом. Если они одинаковые, то число является палиндромом.

Листинг 5.17. Проверяем, является ли число палиндромом

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

int main()
{
    int n, reversed = 0, remainder, original;

    cout << "Введите целое число: ";
    cin >> n;

    original = n;

    // вычисляем обратное число
    while( n!=0 )
    {
        remainder = n%10;
        reversed = reversed*10 + remainder;
        n /= 10;
    }

    // палиндром, если исходное число и обратное одинаковые
    if (original == reversed)
        cout << original << " - палиндром\n";
    else
        cout << original << " - не палиндром\n";

    return 0;
}
```

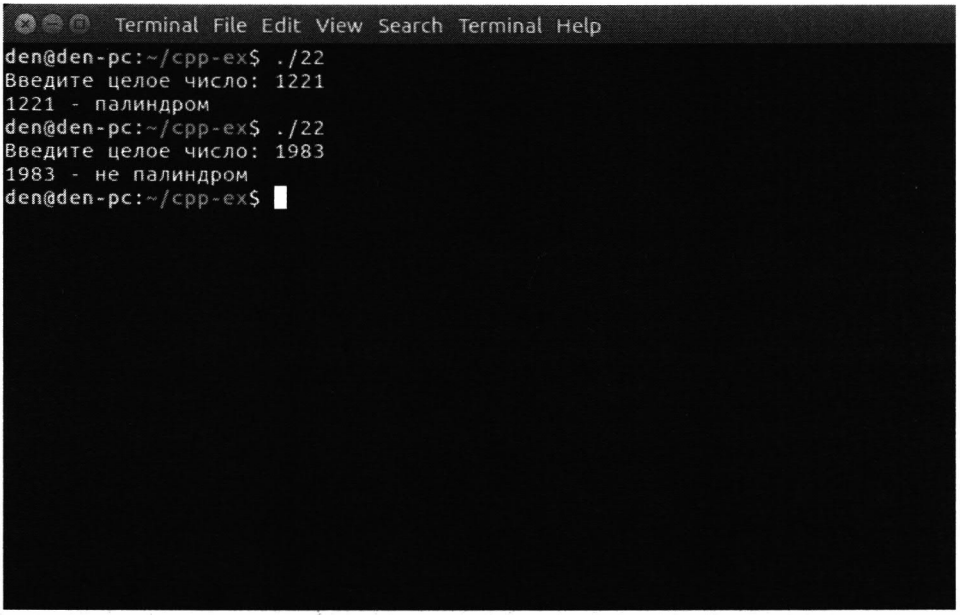


Рис. 5.11. Результат работы программы

Пример: простые числа

Простое число - это то, которое делится только на 1 и на себя. Например: 2, 3, 5, 7, 11, 13. Код программ, определяющей, является ли число простым, приведен в листинге 5.18.

Листинг 5.18. Является ли число простым

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

int main()
{
  int n, i;
  bool isPrime = true;

  cout << "Введите целое положительное число: ";
  cin >> n;

  for(i = 2; i <= n / 2; ++i)
  {
```

```
        if(n % i == 0)
        {
            isPrime = false;
            break;
        }
    }
    if (isPrime)
        cout << "Это простое число" << endl;
    else
        cout << "Это не простое число" << endl;

    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./23
Введите целое положительное число: 10
Это не простое число
den@den-pc:~/cpp-ex$ ./23
Введите целое положительное число: 9
Это не простое число
den@den-pc:~/cpp-ex$ ./23
Введите целое положительное число: 3
Это простое число
den@den-pc:~/cpp-ex$
```

Рис. 5.12. Определяем простые числа

Если цикл for заканчивается, когда тестовое выражение i<=n/2 = false, введенное число является простым. Значение переменной isPrime при этом будет равно 0.

Если цикл for заканчивается из-за оператора break внутри цикла, то число не является простым, при этом значение isPrime будет равно 1.

Усложним предыдущую задачу: представим, что нужно вывести простые числа, находящиеся в интервале между двумя введенными числами. Данная

задача решается с использованием вложенного цикла for и условного оператора if..else.

В этой программе (лист. 5.19) цикл while выполняет (high - low - 1) итераций. При каждой итерации проверяется, является ли число простым. Если да, оно выводится, если нет, просто происходит переход на следующую итерацию. Для определения, является ли число простым, мы используем уже знакомый по предыдущему примеру цикл for.

Листинг 5.19. Выводим простые числа между a и b (между low и high)

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

int main()
{
    int low, high, i, flag;

    cout << "Выводим простые числа между a и b, введите a и b: ";
    cin >> low >> high;

    while (low < high)
    {
        flag = 0;

        for(i = 2; i <= low/2; ++i)
        {
            if(low % i == 0)
            {
                flag = 1;
                break;
            }
        }

        if (flag == 0)
            cout << low << " ";

        ++low;
    }

    cout << endl;
```

```
    return 0;
}
```

Числа a и b в программе называются low и high, подразумевается, что пользователь введет сначала меньшее число, а потом большее. А что, если наоборот? Чтобы наша программа была более универсальной, мы должны предусмотреть этот вариант развития событий. Если low окажется больше, чем high, мы должны поменять их местами:

```
if (low > high) {
        temp = low;
        low = high;
        high = temp;
    }
```

Соответственно, в листинге 5.20 приведен полный исходный код программы, учитывая эту поправку.

Листинг 5.20. Выводим простые числа

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

int main()
{
    int low, high, i, flag, temp;

    cout << "Выводим простые числа между a и b, введите a и b: ";
    cin >> low >> high;

    if (low > high) {
        temp = low;
        low = high;
        high = temp;
    }


    while (low < high)
    {
```

```
        flag = 0;

        for(i = 2; i <= low/2; ++i)
        {
            if(low % i == 0)
            {
                flag = 1;
                break;
            }
        }

        if (flag == 0)
            cout << low << " ";

        ++low;
    }

    cout << endl;
    return 0;
}
```

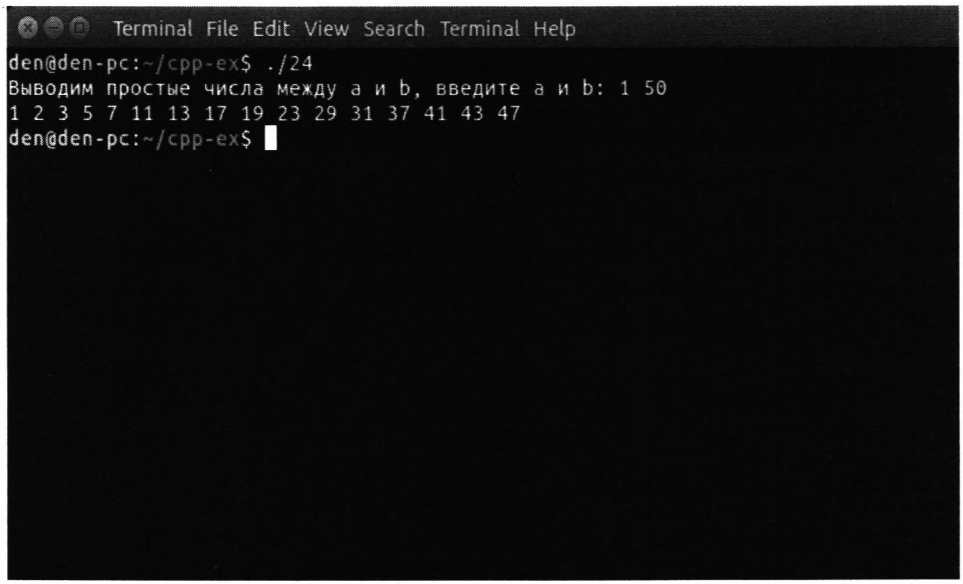
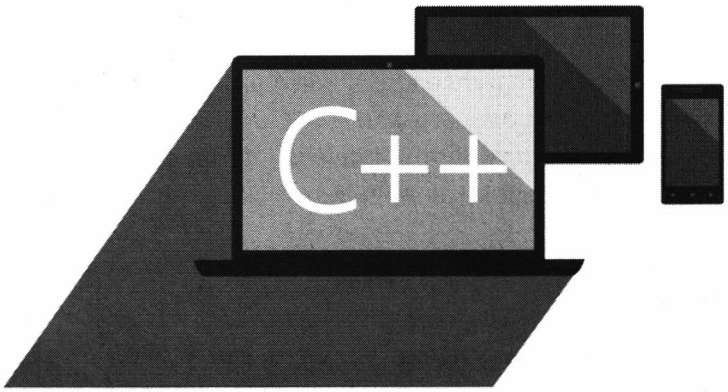


Рис. 5.13. Выводим простые числа между a и b

Глава 6.

Пользовательские функции в C++



6.1. Функция как программный модуль C++

Пользовательские функции - неотъемлемая часть программы. Никому не хочется писать один и тот же код несколько раз. Гораздо проще оформить его в виде функции или подпрограммы (так называются функции в других языках программирования). В этой главе мы поговорим о том, как создать свою функцию и рассмотрим много практических примеров.

Прежде, чем перейти к реальным программам, немного теории. Что такое функция?

Под функцией в программировании подразумевают именованный фрагмент программного кода, который может многократно вызываться в программе. По своей структуре функция – это блок операторов, заключенных в фигурные скобки и имеющих одно общее имя. Функции могут иметь аргументы и возвращать значения в качестве результата. Результатом функции может быть практически все, кроме массива.

Говоря простым яыком, функция - это подпрограмма, возвращающая некоторое значение. Как правило, функции передается какое-то значение (или несколько значений), она производит какие-то действия (например, определяет, является ли число простым) и возвращает определенный результат (например, 1, если число является простым и 0, если это не так).

Конечно, бывают и частные случаи, например, когда функции вообще не передается никаких значений и функции, которые ничего не возвращают. В других языках (например, в Pascal), такие подпрограммы называются процедурами, в C/C++ другого названия не предусмотрено.

Для чего нужны подпрограммы? Чтобы программисту не пришлось писать один и тот же код в разных частях программы. Во-первых, так неудобно. Во-вторых, если программист допустил ошибку в многократно используемом коде, то проще исправить ее один раз - в теле функции, чем искать, где он использовал ошибочный код по всей программе.

6.2. Создание и использование своих собственных функций в программе

Создание и испльзование пользовательских функций в C++ состоит из следующих шагов:

1. Сначала надо объявить функцию – предоставить ее прототип.

2. Определить функцию – описать, что именно должна делать функция.

3. Вызвать функцию

Рассмотрим напримере. Для этого используем уже знакомый нам пример: напишем программу, определяющую, является ли введенное пользователем число простым.

В нашей программе будет две функции: main() и checkPrimeNumber(). Первая - это основная функция программы, а вторая - наша пользовательская функция, возвращающая true, если число является простым. Тип функции checkPrimeNumber() можно использовать как int, поскольку true сводится к 1, а false - к 0, то есть к целым числам.

Чтобы функции могли быть вызваны в функции main(), нужно объявить их до функции main. Но это не всегда удобно, учитывая, что код некоторых функций может быть очень длинным. Поэтому мы можем использовать так называемое предварительное объявление функции - это когда вы описываете

прототип функции (тип возвращаемого значения, имя, параметры), но не пишете сам код функции. А после функции main() можно будет полностью описать наши функции. Предварительное объявление дает компилятору понять, что мы не забыли о функции и что она будет объявлена далее в коде программы.

Полный код программы приведен в листинге 6.1.

Листинг 6.1. Проверка на простое число

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

int checkPrimeNumber(int);    // это объявление функции

int main()
{
  int n;

  cout << "Введите положительное число: ";
  cin >> n;

  if(checkPrimeNumber(n) == true)  // здесь мы вызвали функцию
    cout << n << " - это простое число";
  else
    cout << n << " - не простое число";

  cout << endl;
  return 0;
}

int checkPrimeNumber(int n)  // определение функции
{
  bool flag = true;

  for(int i = 2; i <= n/2; ++i)
  {
      if(n%i == 0)
      {
          flag = false;
          break;
```

```
        }
    }
    return flag;
}
```

Как работает программа, показано на рис. 6.1.

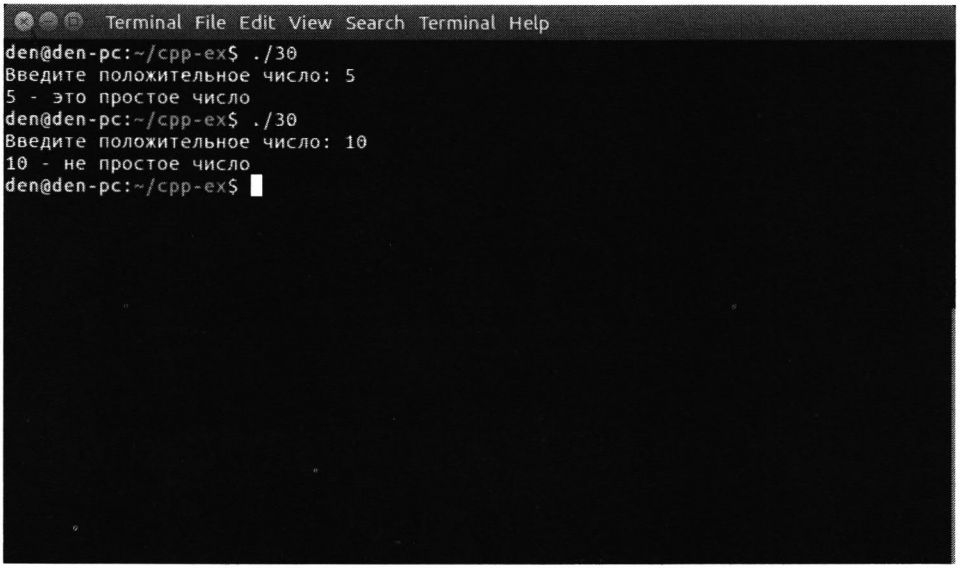


Рис. 6.1. Первая программа с использованием функции

Объявление функции

Объявление функции – это предоставление ее прототипа. При этом под прототипом функции понимается заголовок функции без тела функции. В прототипе указывается имя функции, аргументы (параметры) и их типы , возвращаемый тип данных.

Объявление функции является оператором, поэтому после прототипа функции нужно обязательно ставить точку с запятой.

Прототип используется компилятором, чтобы понимать, как работать с памятью и как контролировать этот процесс. Кстати говоря, для этого компилятору совершенно не важны имена аргументов, поэтому в прототипе функции имена аргументов (параметров) можно не указывать.

Например, можно предоставить такой прототип:

```
int checkPrimeNumber(int n)
```

А можно предоставить такой прототип:

```
int checkPrimeNumber(int)
```

Оба варианта правильны и для компилятора имеют одно и то же значение.

Определение функции и вызов функции

Определить функцию можно с помощью ключевого слова function:

```
<тип> function имя (параметры) {
   тело;
return значение; //значение приводится к Типу, возвращаемому функцией
}
```

Тип - это тип возвращаемого значения. В скобках после имени функции указываются параметры, для каждого параметра обязательно указывается его тип. В теле функции должен быть оператор return, возвращающий значение указанного при объявлении функции типа.

Если функция не возвращает значения, то ее тип void:

```
void function имя (параметры) {
   тело;
return; //указывать не обязательно
}
```

Вызов функции может быть осуществлен из любого места программы.

Рассмотрим пример. Ранее мы писали программу, выводящую простые числа в заданном диапазоне. Давайте модернизируем ее с использованием уже готовой функции checkPrimeNumber(). Код программы будет таким же, но вместо того, чтобы в цикле производить вычисления, мы просто будем вызывать нашу функцию.

Листинг 6.2. Выводим простые числа в заданном диапазоне

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

int checkPrimeNumber(int);

int main()
{
    int n1, n2;
    bool flag;

    cout << "Введите два положительных целых числа: ";
    cin >> n1 >> n2;

    cout << "Простые числа между " << n1 << " и " << n2 << ": ";

    for(int i = n1+1; i < n2; ++i)
    {
        // Если i - простое число, flag = 1
        flag = checkPrimeNumber(i);

        if(flag)
            cout << i << " ";
    }
    cout << endl;
    return 0;
}

// пользовательская функция для проверки на простое число
int checkPrimeNumber(int n)
{
    bool flag = true;

    for(int j = 2; j <= n/2; ++j)
    {
        if (n%j == 0)
        {
            flag = false;
            break;
        }
    }
    return flag;
}
```

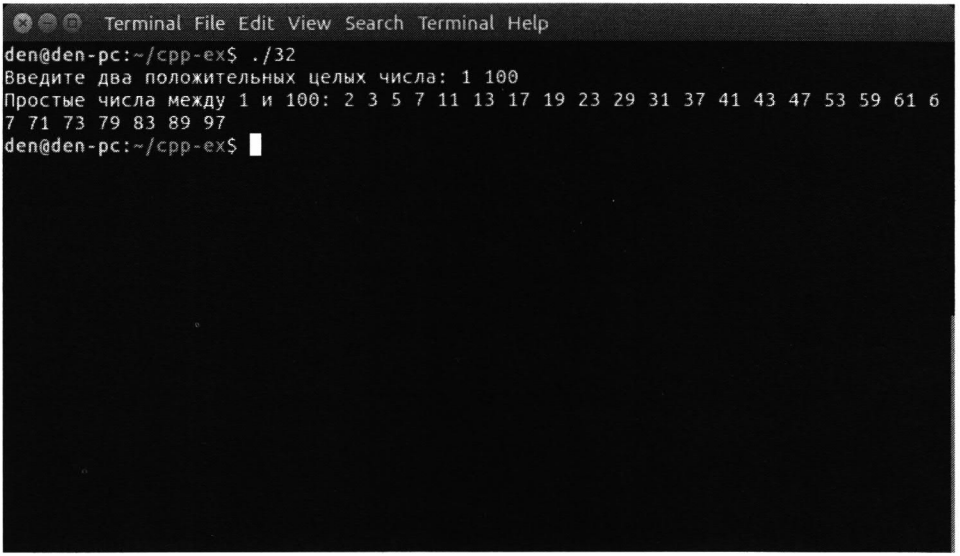


Рис. 6.2. Вывод простых чисел в заданном диапазоне

Продолжаем дальше наше путешествие по миру простых чисел. Давайте напишем программу, позволяющую проверить, может ли быть число выраженным в виде суммы двух простых чисел.

Наша программа в цикле for будет пытаться найти сумму двух простых чисел, равную заданному пользователем числу.

Листинг 6.3. Выражаем число как сумму двух простых чисел

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

bool checkPrime(int n);

int main()
{
    int n, i;
    bool flag = false;

    cout << "Введите положительное целое число: ";
    cin >> n;
```

```
    for(i = 2; i <= n/2; ++i)
    {
        if (checkPrime(i))
        {
            if (checkPrime(n - i))
            {
                cout << n << " = " << i << " + " << n-i << endl;
                flag = true;
            }
        }
    }

    if (!flag)
      cout << n << " не может быть выражено как сумма двух
простых чисел\n";

    return 0;
}

// проверка на простое число
bool checkPrime(int n)
{
    int i;
    bool isPrime = true;

    for(i = 2; i <= n/2; ++i)
    {
        if(n % i == 0)
        {
            isPrime = false;
            break;
        }
    }

    return isPrime;
}
```

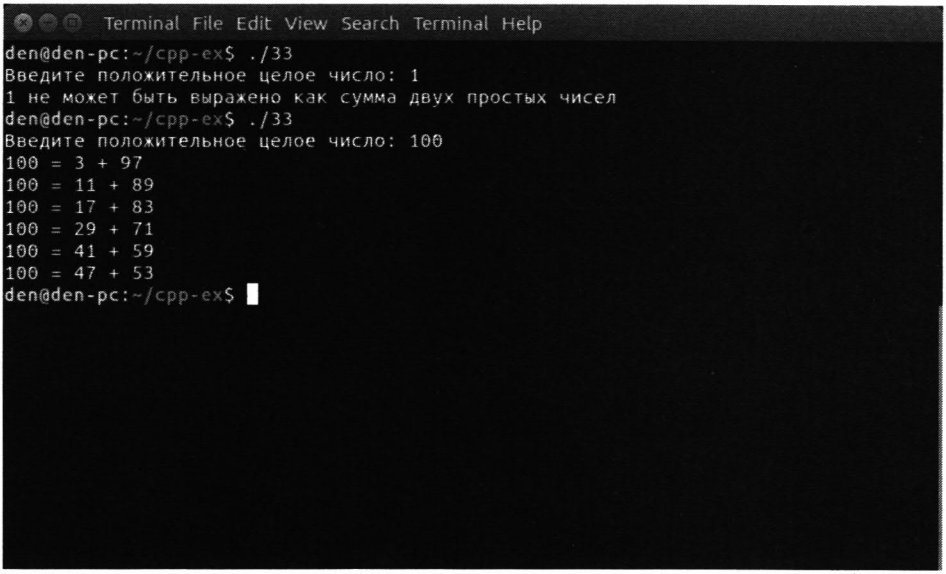


Рис. 6.3. Программа в действии

6.3. Рекурсия

Рекурсия - это явление, когда функция вызывает саму себя. Начинающим программистам не рекомендуется использовать рекурсию, поскольку она может привести к переполнению памяти, если не предусмотреть корректного условия выхода из рекурсии, то есть завершения функции.

Классический пример рекурсивной функции - это функция вычисления факториала. Именно на примере этой функции обучают рекурсии. Не будем отступать от традиции (лист. 6.4).

Листинг 6.4. Рекурсивное вычисление факториала

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

int factorial(int n);

int main()
{
    int n;
```

```
    cout << "Введите положительное число: ";
    cin >> n;

    cout << "Факториал " << n << " = " << factorial(n) << endl;

    return 0;
}

int factorial(int n)
{
    if(n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}
```

Разберемся, как работает функция. Сначала мы передаем этой функции значение, введенное пользователем. Пусть это будет 3. Если аргумент функции n больше 0, то она умножает n на значение, возвращенное самой собой, но при этом передавая ей другое значение аргумента - уменьшенное на 1. Следовательно, следующий вызов функции уже происходит для значения 2. Далее ситуация повторяется, пока n > 1. Когда n = 1, функция вернет 1 и не будет вызывать себя, то есть произойдет вызов из рекурсии:

```
return 3 * factorial(2);
  return 2 * factorial(1);
    return 1;
```

В итоге должно получиться 1 * 2 * 3, итого результат будет равен 6.

Пройдемся еще раз по вызовам функции. Пользователь передает число 5:

```
return 5 * fact(4);
  return 4 * fact(3);
    return 3 * fact(2);
      return 2 * fact(1);
        return 1;
```

Результат 1 * 2 * 3 * 4 * 5 = 120

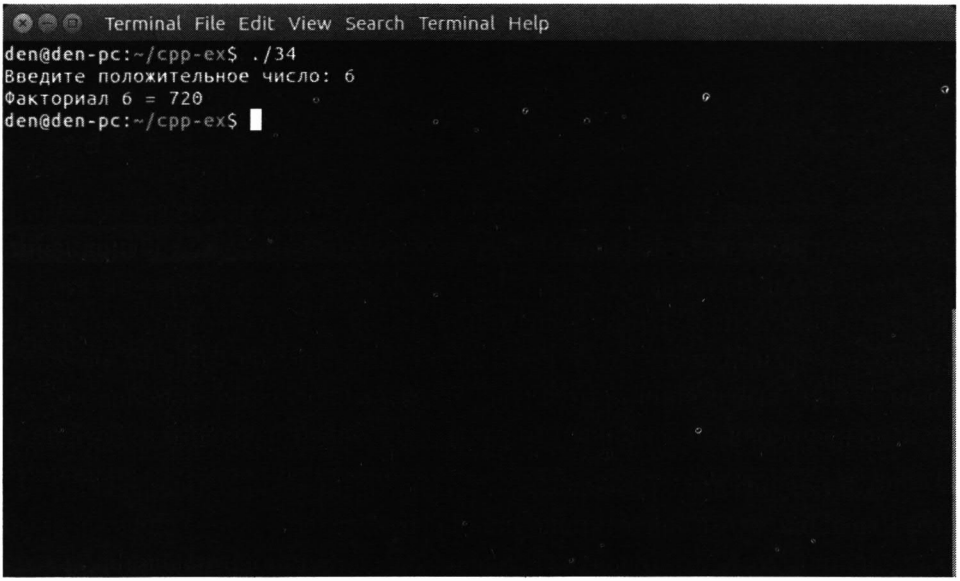


Рис. 6.4. Вычисление факториала

По возможности старайтесь не использовать рекурсивные алгоритмы. Помните, что любой рекурсивный алгоритм можно превратить в нерекурсивный. Опасность рекурсии в том, что если забыть предусмотреть условие выхода из рекурсии либо же задать неправильное условие, то это приведет к зацикливанию программы. Но сколько бы ни не советовали программистам отказаться от рекурсии, ее все равно используют. Почему? Да потому, что это удобно. Именно поэтому далее последует ряд примеров с использованием рекурсии, чтобы вы научились писать рекурсивные программы правильно.

Рассмотрим еще один рекурсивный алгоритм. На этот раз мы вычислим сумму n натуральных чисел с использованием рекурсии (лист. 6.5).

Листинг 6.5. Вычисление суммы натуральных чисел с использованием рекурсии

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

int add(int n);

int main()
{
```

```
    int n;

    cout << "Введите положительное целое число: ";
    cin >> n;

    cout << "Сумма =  " << add(n) << endl;

    return 0;
}

int add(int n)
{
    if(n != 0)
        return n + add(n - 1);
    return 0;
}
```

Рекурсия делает код компактнее и в этом ее прелесть. Посмотрите, насколько компактной получилась наша программа. Давайте разберемся, что происходит. Допустим, пользователь ввел цифру 2. Функция addNumbers() вызывается с параметром 2. Функция проверяет, что переданное ей значение не равно 0 и поэтому возвращает значение:

```
2 + addNumbers(1);
```

Снова вызывается функция, но уже с параметром 1. Так как 1 - не равно 0, то функция возвращает значение:

```
1 + addNumbers(0);
```

Функция проверяет, что переданное ей значение равно 0 и возвращает его. Теперь что у нас есть:

- 0
- 1
- 2

Очень важно предусмотреть условие выхода из рекурсии. В нашем случае условием выхода является n = 0: функция просто возвращает 0 и не вызывает снова саму себя.

Результат работы программы приведен на рис. 6.5.

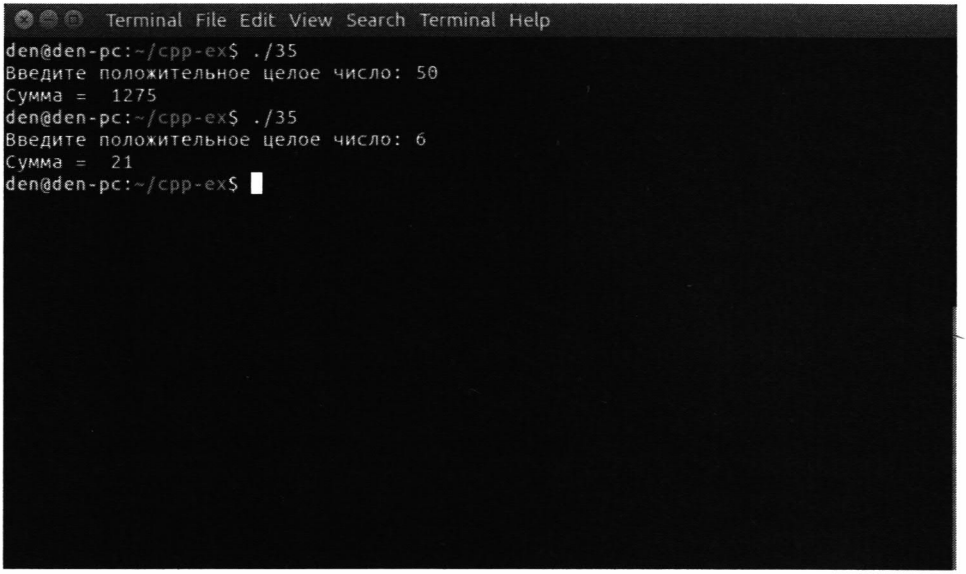


Рис. 6.5. Результат работы программы

До этого мы работали только с числами. На этот раз мы напишем рекурсивную функцию reverseSentence(), которая будет выводить введенное пользователем предложение в обратном порядке. Здесь мы забегаем немного вперед, так как строковые данные мы рассмотрим в одной из следующих глав.

Листинг 6.6. Вывод предложения в обратном порядке

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

void reverse(const string& a);

int main()
{
    string str;
    cout << "Введите строку: " << endl;
```

```
    getline(cin, str);
    reverse(str);
    return 0;
}

void reverse(const string& str)
{
    size_t numOfChars = str.size();

    if(numOfChars == 1)
       cout << str << endl;
    else
    {
       cout << str[numOfChars - 1];
       reverse(str.substr(0, numOfChars - 1));
    }
}
```

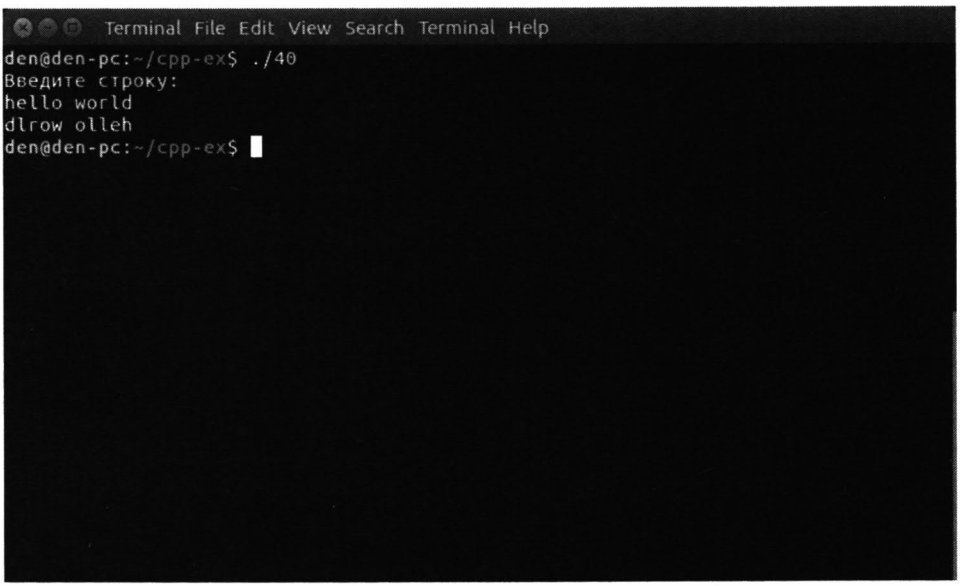


Рис. 6.6. Программа в действии

Сначала программа выводит приглашение. Затем вызывается наша рекурсивная функция. Она сохраняет первый введенный пользователем символ в переменной c. Если это не \n, то функция вызывается снова.

Когда функция вызывается во второй раз, второй символ сохраняется в c. Но эта переменная c - не та, которая была в первый раз, поскольку функция была вызвана снова. Обе эти переменные занимают разное место в памяти.

Этот процесс продолжается, пока пользователь не нажмет Enter (символ '\n'); Когда пользователь нажал Enter, последняя функция reverseSentence() возвращает последний символ - print("%c", c) и возвращается ко второй с конца функции. Та функция выводит, соответственно, второй последний символ и так продолжается до тех пор, пока не будет выведено все предложение в обратном порядке. Данный пример демонстрирует всю силу и гибкость рекурсии.

6.4. Передача параметров по ссылке и по значению

Когда мы вызываем функцию, то мы должны передать ей параметры(аргументы), которая она будет использовать в своей работе. В языке программирования C++ продусмотрены два механизма передачи аргументов функциям: по значению и по ссылке.

При передаче аргумента функции по значению при вызове функции для переменных, которые указаны ее аргументами, создаются копии, которые, фактически, и передаются функции. После завершения выполнения кода функции эти копии уничтожаются (выгружаются из памяти). При передаче аргументов функции по ссылке функция получает непосредственный доступ (через ссылку) к переменным, указанным аргументами функции. С практической точки зрения разница между этими механизмами заключается в том, что при передаче аргументов по значению изменить передаваемые функции аргументы в теле самой функции нельзя, а при передаче аргументов по ссылке – можно. По умолчанию используется механизм передачи аргументов функции по значению.

Проиллюстрируем сказанное на конкретном примере. Рассмотрим программный код, приведенный в листинге 6.7.

Листинг 6.7. Передача аргумента по значению

```
#include <iostream>
using namespace std;
//Аргумент передается по значению:
int incr(int m){
m=m+1;
return m;
}
int main(){
int n=5;
cout<<"n ="<<incr(n)<<endl;
cout<<"n ="<<n<<endl;
return 0;
}
```

Программный код функции `incr()` предельно прост: функция в качестве значения возвращает целочисленную величину, на единицу превышающую аргумент функции. Особенность программного кода состоит в том, что в теле функции выполняется команда `m=m+1`, которой с формальной точки зрения аргумент функции `m` увеличивается на единицу, и именно это значение возвращается в качестве результата. Поэтому с точки зрения здравой логики после вызова функции переменная, переданная ей в качестве аргумента должна увеличиться на единицу. Однако это не так. Вывод результатов выполнения программы на экран будет выглядеть так:

```
n =6
n =5
```

Если с первой строкой проблем не возникает, то вторая выглядит несколько неожиданно. Хотя это только на первый взгляд. Остановимся подробнее на том, что происходит при выполнении программы.

В начале главного метода `main()` инициализируется со значением `5` целочисленная переменная `n`. Далее на экран выводится результат вычисления выражения `incr(n)` и затем значение переменной `n`. Поскольку функция `incr()` возвращает значение, на единицу большее аргумента, результат этого выражения равен `6`. Но почему же тогда не меняется переменная `n`? Все дело в механизме передачи аргумента функции. Поскольку аргумент передается по значению, при выполнении инструкции `incr(n)` для переменной `n` автоматически создается копия,

которая и передается аргументом функции `incr()`. В соответствии с кодом функции значение переменной-копии увеличивается на единицу и полученное значение возвращается в качестве результата функции. Как только результат функцией возвращен, переменная-копия прекращает свое существование. Поэтому функция вычисляется корректно, а переменная-аргумент не меняется!

Для того чтобы аргумент передавался не по значению, а по ссылке, перед именем соответствующего аргумента необходимо указать оператор `&`. В листинге 6.8 приведен практически тот же программный код, что и в листинге 6.7, однако аргумент функции `incr()` в этом случае передается по ссылке.

Листинг 6.8. Передача аргумента по ссылке

```
#include <iostream>
using namespace std;
//Аргумент передается по ссылке:
int incr(int &m){
m=m+1;
return m;
}
int main(){
int n=5;
cout<<"n ="<<incr(n)<<endl;
cout<<"n ="<<n<<endl;
return 0;
}
```

Результат выполнения программы выглядит так:

```
n =6
n =6
```

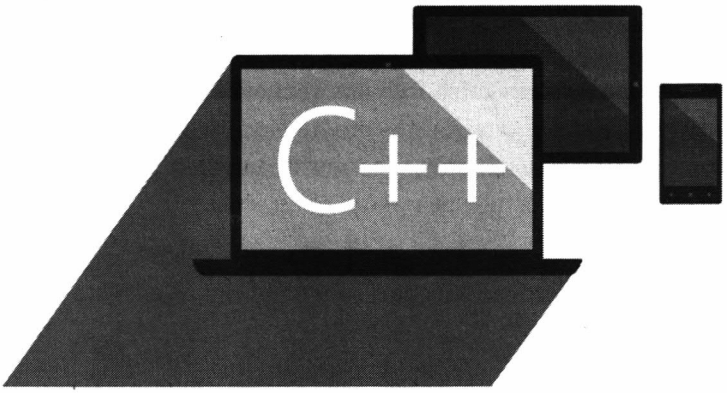
Поскольку аргумент функции передается по ссылке, то все манипуляции в теле функции выполняются не с копией аргумента, а непосредственно с аргументом. Таким образом, вызов функции `incr(n)` не только возвращает в качестве результата увеличенное на единицу значение аргумента, но

приводит к тому, что этот аргумент действительно увеличивается на единицу.

Если у функции несколько аргументов, часть из них (или все) могут передаваться по ссылке, а часть – по значению.

Глава 7.

Массивы в C++



7.1. Что такое массив

В программировании довольно часто приходится иметь дело с наборами данных одного типа. Обычно такие данные в программе реализуют в виде массива. **Массив – это множество значений одного типа, объединенных общим именем.**

Переменные, входящие в состав массива, называются **элементами массива**. Доступ к элементам массива осуществляется путем индексирования. Размерность массива определяется количеством индексов, необходимых для однозначного определения элемента массива. Остановимся более детально на том, как массивы реализуются в C++.

Для создания массива используется оператор объявления. При этом объявление массива должно задавать три аспекта:

- Имя массива
- Количество элементов массива
- Тип для всех элементов массива

7.2. Одномерные массивы

Одномерный массив – это массив, для индексации элементов которого используют один индекс. Как и в случае с обычной переменной, перед использованием массива его следует объявить.

В общем выиде объявление одномерного массива выглядит так:

Тип_элементов_массива имя_Массива [размер_массива]

Размер массива указывается в квадратных скобках сразу после имени массива. Например, командой `int m[10]` объявляется целочисленный массив с именем `m`, который состоит из `10` элементов. Размер массива задается числовым литералом или числовой константой. Размер массива должен быть известен на момент компиляции программы и не изменяется в процессе ее выполнения.

Обращение к элементу массива выполняется через имя массива с индексом элемента в квадратных скобках. При этом следует помнить, что индексация элементов в C++ начинается с нуля. Таким образом, первым элементом означенного выше массива является `m[0]`, а последним, десятым — элемент `m[9]`. Эта особенность языка C++ становится особенно важной с учетом того, что при компиляции и выполнении программ проверка на предмет выхода за пределы массива не выполняется.

Давайте рассмотрим пример. Представим, что есть некий массив типа float, введенный пользователем и нам нужно вычислить среднее арифметическое его элементов.

Наша программа будет демонстрировать:

1. Заполнение массива. В цикле for мы читаем каждый элемент массива.

2. Параллельный подсчет среднего арифметического. Обратите внимание, мы вычисляем среднее в том же цикле, что и ввод данных - это пример оптимизации. Мы отказались от лишнего прохода по элементам массива.

Листинг 7.1. Вычисление среднего арифметического

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

int main()
{
```

```
    int n, i;
    float num[100], sum=0.0, average;

    cout << "Количество элементов массива: ";
    cin >> n;

    while (n > 100 || n <= 0)
    {
        cout << "Количество может быть от 1 до 100" << endl;
        cout << "Введите количество снова ";
        cin >> n;
    }

    for(i = 0; i < n; ++i)
    {
        cout << i + 1 << ". Введите элемент массива: ";
        cin >> num[i];
        sum += num[i];
    }

    average = sum / n;
    cout << "Среднее = " << average << endl;

    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./42
Количество элементов массива: 5
1. Введите элемент массива: 5
2. Введите элемент массива: 2
3. Введите элемент массива: 7
4. Введите элемент массива: 2
5. Введите элемент массива: 1
Среднее = 3.4
den@den-pc:~/cpp-ex$
```

Рис. 7.1. Результат вычисления среднего арифметического

В предыдущем примере мы вычисляли среднее прямо в том же массиве, в котором организовали ввод пользователя. Аналогично, можно было бы вычислить и максимум массива. Но в этом примере мы покажем другой трюк. Пусть у нас есть массив элементов arr[100]. Мы пройдемся по элементам массива от 1 до 100, точнее до n (n вводит пользователь, а максимум n = 100) и попробуем найти максимум в массиве. При этом максимум мы будем хранить не в отдельной переменной, а в самом массиве - в элементе arr[0].

Листинг 7.2. Вычисляем максимум в массиве

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

int main()
{
    int i, n;
    float arr[100];

    cout << "Введите количество элементов (1-100): ";
    cin >> n;
    cout << endl;

    // Сохраняем элементы массива
    for(i = 0; i < n; ++i)
    {
       cout << "Введите число " << i + 1 << " : ";
       cin >> arr[i];
    }

    // Ищем макс. элемент и сохраняем его в arr[0]
    for(i = 1;i < n; ++i)
    {
       // Замените < на > если вам нужно найти минимальный элемент
       if(arr[0] < arr[i])
           arr[0] = arr[i];
    }
    cout << "Максимум = " << arr[0] << endl;

    return 0;
}
```

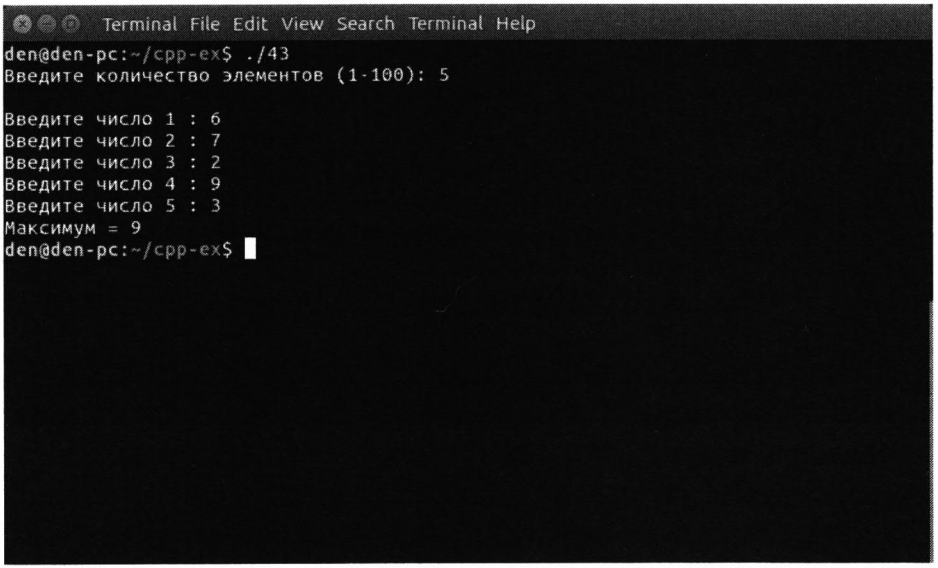


Рис. 7.2. Поиск максимума в массиве

7.3. Многомерные массивы

Размерность массива может быть больше единицы (напомним, что размерность массива определяется количеством индексов, с помощью которых реализуется доступ к элементу массива). В таком случае говорят о многомерных массивах. Объявление многомерного массива выполняется так же просто, как и объявление одномерного массива, с той лишь разницей, что теперь для массива указывается размер по каждому из индексов. Для каждого индекса используется собственная пара квадратных скобок. При объявлении массива размер массива по соответствующему индексу также указывается в отдельных квадратных скобках.

Среди многомерных массивов самым простым является двумерный массив. В известном смысле двумерный массив – это массив из одномерных массивов. Например, инструкцией `double n[4][5]` объявляется двумерный массив действительных чисел двойной точности размером 4x5. Как и ранее, чтобы обратиться к отдельному элементу массива, необходимо после имени массива указать индексы этого элемента (каждый индекс в отдельных квадратных скобках). Индексация по каждому индексу начинается с нуля.

Рассмотрим использование двухмерных массивов на примере сложения двух матриц. У нас будет три матрицы: a[100][100], b[100][100] и sum[100][100]. Последняя, как вы уже догадались, будет содержать сумму матриц a и b.

Пользователь вводит количество строк r и количество колонок c. Значения r и c в этой программе должны быть меньше 100.

После ввода r и c, пользователь должен будет ввести элементы обеих матриц. Далее программа выполнит сложение матриц и отобразит результат.

Листинг 7.3. Сложение двух матриц с использованием многомерных массивов

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

int main()
{
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;

    cout << "Введите количество строк (1-100): ";
    cin >> r;

    cout << "Введите количество колонок (1-100): ";
    cin >> c;

    cout << endl << "Введите элементы первой матрицы: " << endl;

    // Вводим элементы первой матрицы
    for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << "Введите элемент a" << i + 1 << j + 1 << " : ";
           cin >> a[i][j];
       }

    // Вводим элементы второй матрицы
    cout << endl << "Вводим элементы второй матрицы: " << endl;
    for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
```

```
            cout << "Введите элемент b" << i + 1 << j + 1 << " : ";
            cin >> b[i][j];
        }

    // Выполняем суммирование двух матриц
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
            sum[i][j] = a[i][j] + b[i][j];

    // Отображаем результат
    cout << endl << "Сумма двух матриц: " << endl;
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
        {
            cout << sum[i][j] << "  ";
            if(j == c - 1)
                cout << endl;
        }
   cout << endl;
    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./45
Введите количество строк (1-100): 2
Введите количество колонок (1-100): 2

Введите элементы первой матрицы:
Введите элемент a11 : 1
Введите элемент a12 : 1
Введите элемент a21 : 1
Введите элемент a22 : 1

Вводим элементы второй матрицы:
Введите элемент b11 : 1
Введите элемент b12 : 1
Введите элемент b21 : 1
Введите элемент b22 : 1

Сумма двух матриц:
2  2
2  2

den@den-pc:~/cpp-ex$
```

Рис. 7.3. Сложение двух матриц

Рассмотрим еще один пример работы с многомерными массивами - умножение двух матриц. Чтобы умножить две матрицы, число колонок первой матрицы должно быть равно количество строк второй матрицы. Программа отобразит ошибку, если это не так.

Произведение матриц AB состоит из всех возможных комбинаций скалярных произведений вектор-строк матрицы A и вектор-столбцов матрицы B. Элемент матрицы AB с индексами i, j есть скалярное произведение i-ой вектор-строки матрицы A и j-го вектор-столбца матрицы B. Общая формула выглядит так:

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = 1, 2, \ldots l;\ j = 1, 2, \ldots n).$$

Как видите, это не просто умножить элемент a[i,j] на элемент b[i,j]. Подробнее можно прочитать в Википедии: https://ru.wikipedia.org/wiki/Умножение_матриц.

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./46
К-во строк и колонок первой матрицы: 2 2
К-во строк и колонок второй матрицы: 2 2

Введите элементы матрицы 1:
Вводим элемент a11 : 2
Вводим элемент a12 : 2
Вводим элемент a21 : 2
Вводим элемент a22 : 2

Вводим элементы матрицы 2:
Введите элмент b11 : 3
Введите элмент b12 : 3
Введите элмент b21 : 3
Введите элмент b22 : 3

Результат умножения матрицы:
 12 12
 12 12

den@den-pc:~/cpp-ex$
```

Рис. 7.4. Результат умножения двух матриц

На рис. 7.4 видно, что были созданы две матрицы размером 2x2. Все элементы первой матрицы равны 2, второй - 3. В результате мы получим также матрицу размером 2x2, все элементы которой равны 12. Компоненты матрицы C (результирующая матрица) вычисляются следующим образом:

```
c11 = a11 · b11 + a12 · b21 = 2 · 3 + 2 · 3 = 6 + 6 = 12
c12 = a11 · b12 + a12 · b22 = 2 · 3 + 2 · 3 = 6 + 6 = 12
c21 = a21 · b11 + a22 · b21 = 2 · 3 + 2 · 3 = 6 + 6 = 12
c22 = a21 · b12 + a22 · b22 = 2 · 3 + 2 · 3 = 6 + 6 = 12
```

Листинг 7.4. Умножение матриц

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

int main()
{
  int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;

    cout << "К-во строк и колонок первой матрицы: ";
    cin >> r1 >> c1;
    cout << "К-во строк и колонок второй матрицы: ";
    cin >> r2 >> c2;

    // Проверяем, можем ли мы умножить две матрицы
    while (c1!=r2)
    {
        cout << "Ошибка! К-во колонок первой матрицы не равно
количеству строк второй.";

        cout << "К-во строк и колонок первой матрицы: ";
        cin >> r1 >> c1;

        cout << "К-во строк и колонок второй матрицы: ";
        cin >> r2 >> c2;
    }
    // Вводим элементы первой матрицы
    cout << endl << "Введите элементы матрицы 1:" << endl;
    for(i = 0; i < r1; ++i)
        for(j = 0; j < c1; ++j)
        {
```

```
            cout << "Вводим элемент a" << i + 1 << j + 1 << " : ";
            cin >> a[i][j];
        }

    // Для второй матрицы
    cout << endl << "Вводим элементы матрицы 2:" << endl;
    for(i = 0; i < r2; ++i)
        for(j = 0; j < c2; ++j)
        {
            cout << "Введите элмент b" << i + 1 << j + 1 << " : ";
            cin >> b[i][j];
        }
    // Инициализируем элементы результирующей матрицы mult
    // путем их установки в 0.
    for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
        {
            mult[i][j]=0;
        }

    // Умножаем матрицы a и b
    // Результат сохраняем в матрице result
    for(i = 0; i < r1; ++i)
        for(j = 0; j < c2; ++j)
            for(k = 0; k < c1; ++k)
            {
                mult[i][j] += a[i][k] * b[k][j];
            }

    // Отображаем результат
    cout << endl << "Результат умножения матрицы: " << endl;
    for(i = 0; i < r1; ++i)
    for(j = 0; j < c2; ++j)
    {
        cout << " " << mult[i][j];
        if(j == c2-1)
            cout << endl;
    }
    cout << endl;
    return 0;
}
```

В следующем примере пользователь вводит количество строк и колонок (r и c соответственно). Значения обоих переменных в этой программе должно быть меньше 10. Затем программа вычисляет транспонированную матрицу и выводит результат на экран (лист. 7.5). Напомню, что транспонированная матрица — матрица, полученная из исходной матрицы заменой строк на столбцы. В листинге 7.5 приводится код программы, вычисляющей транспонированную матрицу. Результат выполнения программы изображен на рис. 7.5: обратите внимание на исходную матрицу и результат.

```
            cout << " " << mult[i][j];
            if(j == c2-1)
                cout << endl;
        }
        cout << endl;
        return 0;
}
```

Листинг 7.5. Транспонированная матрица

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

int main()
{
    int a[10][10], transpose[10][10], r, c, i, j;
    cout << "Введите количество строк и колонок: ";
    cin >> r >> c;

    // Сохраняем элементы
    cout << "\nВведите элементы матрицы:\n";
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
```

```
        {
            cout << "a" << i+1 << j+1 << ": ";
            cin >> a[i][j];
        }

    // Показываем a[][] */
    cout << "\nИсходная матрица: \n";
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            cout << a[i][j] << " ";
            if (j == c-1)
                cout << "\n\n";
        }

    // Вычисляем транспонированную матрицу
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            transpose[j][i] = a[i][j];
        }

    // Результат
    cout << "\nТранспонированная матрица:\n";
    for(i=0; i<c; ++i)
        for(j=0; j<r; ++j)
        {
            cout << transpose[i][j] << " ";
            if(j==r-1)
                cout << "\n\n";
        }
    cout << endl;
    return 0;
}
```

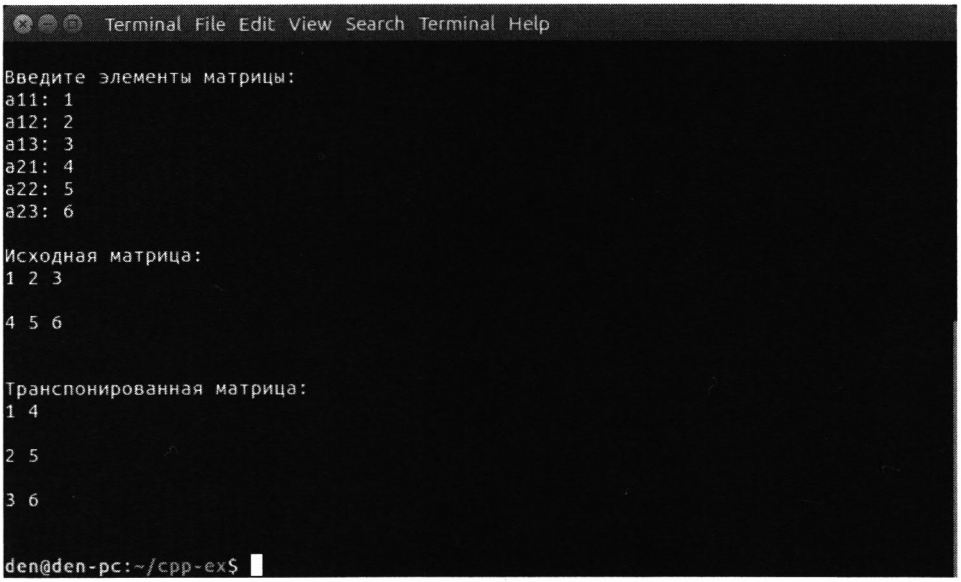


Рис. 7.5. Вычисление транспонированной матрицы

7.4. Передача массивов в функцию в качестве аргумента

Иногда требуется массив передать какой-либо функции в качестве аргумента. В языке C++ это можно сделать.

При передаче массива в качестве аргумента функции необходио указать его имя и размерность, например:

```
float calculateSD(float data[10]); // передача одномерного массива
float calculateHD(float data[10][25]); // передача двумерного
                                       // массива
```

При этом в C++ у массивов есть такая особенность: имя массива является ссылкой на первый его элемент. Благодаря этому, при передаче массива можно не указывать первую размерность, то есть вместо приведенного выше примера можно записать:

```
float calculateSD(float data[]); // передача одномерного массива
float calculateHD(float data[][25]); // передача двумерного массива
```

В качестве практического примера рассмотрим программу, которая вычисляет среднеквадратическое отклонение (СО). Для его вычисления мы создали функцию calculateSD(), поэтому данный пример будет демонстрировать передачу массива данных функции и работу с ним в функции (лист. 7.6). Результат работы программы, а также команда ее компиляции станут понятны после создания листинга.

Листинг 7.6. Передача массива функции

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

float calculateSD(float data[]);

int main()
{
    int i;
    float data[10];

    cout << "Введите десять элементов: ";
    for(i = 0; i < 10; ++i)
        cin >> data[i];

    cout << endl << "CO = " << calculateSD(data) << endl;

    return 0;
}

float calculateSD(float data[])
{
    float sum = 0.0, mean, standardDeviation = 0.0;

    int i;

    for(i = 0; i < 10; ++i)
    {
        sum += data[i];
    }

    mean = sum/10;
```

```
    for(i = 0; i < 10; ++i)
        standardDeviation += pow(data[i] - mean, 2);

    return sqrt(standardDeviation / 10);
}
```

Ранее мы рассматривали умножение матриц. Перепишем этот пример с использованием функций. Как обычно, программа просит пользователя ввести размер матрицы (количество строк и колонок). Затем, она просит пользователя ввести элементы двух матрицы, выполняет умножение и отображает результат.

Для осуществления всего вышеизложенного программа использует следующие функции:

- enterData() - ввод данных от пользователя.
- multiplyMatrices() - умножение двух матриц.
- display() - отображение результата матрицы после умножения.

Листинг 7.7. Умножение двух матриц с использованием функций

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

void enterData(int firstMatrix[][10], int secondMatrix[][10],
int rowFirst, int columnFirst, int rowSecond, int columnSecond);
void multiplyMatrices(int firstMatrix[][10],
int secondMatrix[][10], int multResult[][10], int rowFirst,
int columnFirst, int rowSecond, int columnSecond);
void display(int mult[][10], int rowFirst, int columnSecond);

int main()
{
   int firstMatrix[10][10], secondMatrix[10][10], mult[10][10],
rowFirst, columnFirst, rowSecond, columnSecond, i, j, k;

   cout << "Введите строки и колонки первой матрицы: ";
   cin >> rowFirst >> columnFirst;
```

```
   cout << "Введите строки и колонки второй матрицы: ";
   cin >> rowSecond >> columnSecond;

   // Проверяем, можем ли мы умножить матрицы
   while (columnFirst != rowSecond)
   {
      cout << "Ошибка! К-во колонок первой матрицы не равно
количеству строк второй." << endl;
      cout << "К-во строк и колонок первой матрицы: ";
      cin >> rowFirst >> columnFirst;
      cout << "К-во строк и колонок второй матрицы: ";
      cin >> rowSecond >> columnSecond;
   }

   // Вводим матрицы
         enterData(firstMatrix, secondMatrix, rowFirst,
columnFirst, rowSecond, columnSecond);

         // Умножаем матрицы
         multiplyMatrices(firstMatrix, secondMatrix, mult,
rowFirst, columnFirst, rowSecond, columnSecond);

         // Отображаем
         display(mult, rowFirst, columnSecond);

   return 0;
}

void enterData(int firstMatrix[][10], int secondMatrix[][10], int
rowFirst, int columnFirst, int rowSecond, int columnSecond)
{
   int i, j;
   cout << endl << "Введите элементы матрицы 1:" << endl;
   for(i = 0; i < rowFirst; ++i)
   {
      for(j = 0; j < columnFirst; ++j)
      {
         cout << "Введите элемент a"<< i + 1 << j + 1 << ": ";
         cin >> firstMatrix[i][j];
      }
```

```
    }

    cout << endl << "Введите элементы матрицы 2:" << endl;
    for(i = 0; i < rowSecond; ++i)
    {
        for(j = 0; j < columnSecond; ++j)
        {
            cout << "Введите элемент b" << i + 1 << j + 1 << ": ";
            cin >> secondMatrix[i][j];
        }
    }
}

void multiplyMatrices(int firstMatrix[][10], int secondMatrix[]
[10], int mult[][10], int rowFirst, int columnFirst, int
rowSecond, int columnSecond)
{
    int i, j, k;

    // Инициализация элементов результирующей матрицы
    for(i = 0; i < rowFirst; ++i)
    {
        for(j = 0; j < columnSecond; ++j)
        {
            mult[i][j] = 0;
        }
    }

    // Умножаем матрицы, результат в mult
    for(i = 0; i < rowFirst; ++i)
    {
        for(j = 0; j < columnSecond; ++j)
        {
            for(k=0; k<columnFirst; ++k)
            {
                mult[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}
```

```
void display(int mult[][10], int rowFirst, int columnSecond)
{
   int i, j;

   cout << "Результат умножения матриц:" << endl;
   for(i = 0; i < rowFirst; ++i)
   {
      for(j = 0; j < columnSecond; ++j)
      {
         cout << mult[i][j] << " ";
         if(j == columnSecond - 1)
            cout << endl << endl;
      }
   }
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./48
Введите строки и колонки первой матрицы: 2 2
Введите строки и колонки второй матрицы: 2 2

Введите элементы матрицы 1:
Введите элемент a11: 2
Введите элемент a12: 2
Введите элемент a21: 2
Введите элемент a22: 2

Введите элементы матрицы 2:
Введите элемент b11: 3
Введите элемент b12: 3
Введите элемент b21: 3
Введите элемент b22: 3
Результат умножения матриц:
12 12

12 12

den@den-pc:~/cpp-ex$
```

Рис. 7.6. Умножение 2 матриц с использованием функций

7.5. Векторы. Класс vector

Класс vector является альтернативой обычным массивам. Подробнее о том, что такое класс говорится в след. главах. Но нас сейчас это не интересует. Сейчас нам важно только то, что массив можно объявить, как объект класса vector. И вот после этого применять к нему методы (функции), предусмотренные в классе vector. И именно поэтому может быть полезно объявлять массивы как вектора: набор функций класса vector очень разнообразен.

Шаблон vector расположен в заголовочном файле <vector>, который расположен в пространстве имен std.

Общий вид объявления массива как объекта класса vector выглядит следующим образом:

```
vector <Тип> имя_массива (количество элементов)
```

Функции класса vector приведены в таблице 7.1.

Таблица 7.1. Методы (функции) класса vector

	Метод	Описание
Конструкторы	vector::vector	Конструктор по умолчанию. Не принимает аргументов, создает новый экземпляр вектора
	vector::vector(const vector& c)	Конструктор копии по умолчанию. Создает копию вектора c
	vector::vector(size_type n, const T& val = T())	Создает вектор с n объектами. Если val объявлена, то каждый из этих объектов будет инициализирован её значением; в противном случае объекты получат значение конструктора по умолчанию типа T.
	vector::vector(input_iterator start, input_iterator end)	Создает вектор из элементов, лежащих между start и end
Деструктор	vector::~vector	Уничтожает вектор и его элементы

Операторы	vector::operator=	Копирует значение одного вектора в другой.
	vector::operator==	Сравнение двух векторов
Доступ к элементам	vector::at	Доступ к элементу с проверкой выхода за границу
	vector::operator[]	Доступ к определенному элементу
	vector::front	Доступ к первому элементу
	vector::back	Доступ к последнему элементу
Итераторы	vector::begin	Возвращает итератор на первый элемент вектора
	vector::end	Возвращает итератор на место после последнего элемента вектора
	vector::rbegin	Возвращает reverse_iterator на конец текущего вектора.
	vector::rend	Возвращает reverse_iterator на начало вектора.
Работа с размером вектора	vector::empty	Возвращает true, если вектор пуст
	vector::size	Возвращает количество элементов в векторе
	vector::max_size	Возвращает максимально возможное количество элементов в векторе
	vector::reserve	Устанавливает минимально возможное количество элементов в векторе
	vector::capacity	Возвращает количество элементов, которое может содержать вектор до того, как ему потребуется выделить больше места.
	vector::shrink_to_fit	Уменьшает количество используемой памяти за счет освобождения неиспользованной (C++11)

Модификаторы	vector::clear	Удаляет все элементы вектора
	vector::insert	Вставка элементов в вектор
	vector::erase	Удаляет указанные элементы вектора (один или несколько)
	vector::push_back	Вставка элемента в конец вектора
	vector::pop_back	Удалить последний элемент вектора
	vector::resize	Изменяет размер вектора на заданную величину
	vector::swap	Обменять содержимое двух векторов
Другие методы	vector::assign	Ассоциирует с вектором поданные значения
	vector::get_allocator	Возвращает аллокатор, используемый для выделения памяти

Пример использование приведен ниже.

Листинг 7.8. Использование векторов

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

int main()
{
  // Инициализируем массив с использованием vector ( требует C++11 )
   std::vector<std::string> names = {"john", "bobby", "dear",
"test1", "catherine", "nomi", "shinta", "martin", "abe", "may",
"zeno", "zack", "angeal", "gabby"};

    // Сортируем имена с помощью std::sort
    std::sort(names.begin(), names.end() );

    // Выводим имена (требует C++11)
```

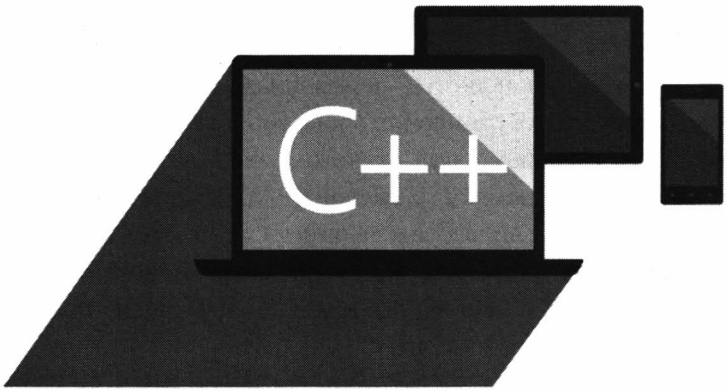
```
    for(const auto& currentName : names)
    {
        std::cout << currentName << std::endl;
    }

    for(int y = 0; y < names.size(); y++)
    {
       std:: cout << names[y] << std::endl;
    }
    return 0;

}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./89
abe
angeal
bobby
catherine
dear
gabby
john
martin
may
nomi
shinta
test1
zack
zeno
abe
angeal
bobby
catherine
dear
gabby
john
martin
may
```

Рис. 7.7. Использование векторов

Глава 8.

Указатели в C++



8.1. Понятие указателя

Указатели и стратегия их использования являются очень важной составляющей программирования на языке C++. До сих пор управления данными мы с вами осуществлли с помощью переменных. При этом переменную мы определяли как именованную область памяти. В эту память сохраняются данные, и обращаются к ним по имени переменнной. Когда в программе объявляется переменная, для этой переменной в памяти выделяется место, размер которого соответсвует типу переменной. При присваивании переменной какого-либо значения это значение записывается в область памяти, выделенную для переменной. Технически при работе с переменной пользователь ничего не знает (да это и не нужно!) о физическом адресе той области, куда записывается значение переменной. Другими словами, вся "кухня" по размещению значения переменной в памяти скрыта от пользователя. С одной стороны, это удобно. Но на практике нередко встречаются ситуации, когда необходимо иметь прямой доступ к областям памяти, содержащим значения переменных. Делается это с помощью указателей.

Указатель – это переменная, значением которой является адрес памяти.

Таким образом, если значением обычной переменной является то значение, что записано в определенной области памяти, то значением переменной-указателя является адрес области памяти (в котором может что-то храниться).

Чтобы лучше понять разницу между обычной переменной и переменной-указателем давайте рассмотрим пример-аналогию из обычной жизни.

Допустим, есть менеджер по продажам, который занимается поиском покупателей. Когда он находит нового покупателя, он записывает его имя и номер телефона. В аналогии с пемеменными и указателями, имя покупателя – это имя переменной, а его номер телефона – указатель. В чем разница между переменной (имя покупателя) и указателем (номер телефона)? Ведь по известному номеру телефона мы в принципе можем определить имя абонента. Зная имя, если постараться, можно определить его номер телефона.

Если абонент не меняет совего номера телефона, то разница между указателем и переменной состоит только в способе использования переменной и указателя. Переменная – когда нужно обратиться к конкретному человеку, указатель – когда нужно позвонить по телефону, и не важно какой конкретно там человек поднимет трубку, если вы значете, что это телефон покупателя.

Оперируя в программе с переменными, мы, фактически, даем инструкции на предмет того, что делать со значениями этих переменных. Указатели позволяют выполнять действия со значениями, записанными по определенному адресу. Если указатель содержит в качестве значения адрес памяти с данными, то говорят, что указатель ссылается на эти данные.

8.2. Объявление указателей

Поскольку указатель – это переменная, его необходимо объявлять. Как и для обычных переменных, для указателя важен тип данных, к которому он относится. Дело в том, что объем памяти, отводимый под переменную, зависит от ее типа. Хотя формально указатель в качестве значения может иметь любой адрес памяти, от типа хранящихся там данных зависит результат арифметических операций с адресами. Поэтому при объявлении указателей необходимо указывать тип данных, на которые эти указатели могут ссылаться. При объявлении указателя используется оператор "звездочка" *. Оператор указывается перед именем указателя. Во всем остальном способ объявления указателя мало отличается от объявления обычной переменной.

Общий вид объявления указателя таков:

тип_данных *имя_указателя;

Например, командой `int *p` объявлен указатель `p` на значение целочисленного типа. Причем одновременно можно объявлять как обычные переменные, так и указатели. Командой `int *q,n,*p` объявляется целочисленная переменная `n` и два указателя `q` и `p` на значения целого типа.

8.3. Операции * и & по работе с указателями

Существует две операции, которые необходимо часто выполнят при работе с указателями. Во-первых, это определение адреса ячейки по ее имени и, во-вторых, определение значения, записанного по указанному адресу. Для этих целей используются операторы `&` и `*` соответственно. В частности, для того, чтобы определить адрес, по которому записана переменная, необходимо перед ее именем указать оператор `&`. Чтобы по адресу (указателю) определить значение, перед соответствующим указателем ставим оператор `*`. Рассмотрим приведен пример использования указателя на значение целого типа.

Листинг 8.1. Использование указателя

```
#include <iostream>
using namespace std;
int main(){
int *q,n,*p;
n=100;
p=&n;
q=p;
(*p)++;
cout<<*q<<"\n";
cout<<n<<"\n";
cout<<p<<"\n";
return 0;
}
```

Сначала объявляется целочисленная переменная n и да указателя q и p. Командой n=100 переменной n присваивается значение 100. Далее с помощью команды p=&n в переменную-указатель p записывается адрес области памяти, в которой хранится значение переменной n. Обращаем внимание читателя на то, что значением переменной p является именно адрес! Командой q=p переменной q в качестве значения присвоен адрес, являющийся значением переменной p. В данном случае использовано то свойство, что при совпадении типов один указатель можно присвоить в качестве значения другому указателю. В результате указатель p и указатель q ссылаются на одну и ту же область памяти.

Командой (*p)++ значение, записанное по адресу p, увеличивается на единицу. Напомним, что инструкция *p означает ссылку на то значение, что записано по адресу p. Команду (*p)++ следует понимать так: взять значение, записанное по адресу p и увеличить его на единицу (поскольку оператор ++ имеет более высокий приоритет, чем оператор *, используем скобки). У этой команды более серьезные последствия, чем может показаться на первый взгляд. Прежде всего, в результате на единицу увеличивается значение переменной n.

Действительно, поскольку адрес p определялся командой p=&n, то, фактически, по определению, это адрес области памяти, где хранится значение переменной n. Поскольку соответствующее значение увеличено на единицу, это означает, что на единицу увеличилось значение переменной n. Далее, указатель q ссылается на эту же область памяти. Поэтому результатом команды *q (значение, записанное по адресу q) будет увеличенное на единицу значение переменной n, т.е. число 101. В конце выполнения программы на экран выводятся значения *q, n и p. Первые два значения, в силу указанных выше причин, равны 101. В качестве значения p выводится адрес области памяти – это число в шестнадцатеричном представлении. Результат может быть, например, таким:

```
101
101
0012FF78
```

В данном случае измениться может только третья строка вывода результатов программы, первые две будут неизменными

8.4. Практический примеры использования указателей

Доступ к элементам массива с использованием указателей

В данном примере мы сохраняем элементы целочисленного массива в переменную data. Далее мы просто выводим элементы массива с использованием метода указателей, без использования квадратных скобок.

Листинг 8.2. Доступ к элементам массива с использованием указателей

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

int main()
{
   int data[5];
   cout << "Введите элементы: ";

   for(int i = 0; i < 5; ++i)
      cin >> data[i];

   cout << "Вы ввели: ";
   for(int i = 0; i < 5; ++i)
      cout << endl << *(data + i);
   cout << endl;
   return 0;
}
```

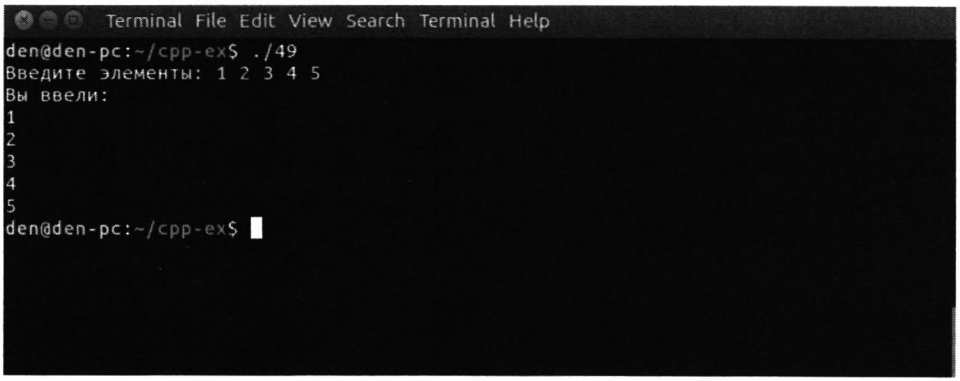


Рис. 8.1. Результат работы программы

Замена местами чисел в массиве с помощью указателей

Рассмотрим еще один пример. Пользователь вводит три числа, которые мы сохраняем в переменные a, b, c соответственно. Затем, эти переменные передаются функции cyclicSwap(). Вместо передачи значений переменных мы передаем в функцию адреса переменных - посмотрите, что возле имени переменной есть *, а функцию мы передаем переменные с помощью &.

После того как cyclicSwap() закончит работу, переменные будут поменяны местами и в основной программе.

Листинг 8.3. Своп чисел с помощью вызова по ссылке

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

void cyclicSwap(int *a, int *b, int *c);

int main()
{
    int a, b, c;

    cout << "Введите значения a, b, c: ";
    cin >> a >> b >> c;

    cout << "Значения перед заменой: " << endl;
    cout << "a, b, c соответственно: " << a << ", " << b << ", "
<< c << endl;

    cyclicSwap(&a, &b, &c);

    cout << "После замены: " << endl;
    cout << " a, b, c соответственно: " << a << ", " << b << ", "
<< c << endl;

    return 0;
}

void cyclicSwap(int *a, int *b, int *c)
{
    int temp;
```

```
    temp = *b;
    *b = *a;
    *a = *c;
    *c = temp;
}
```

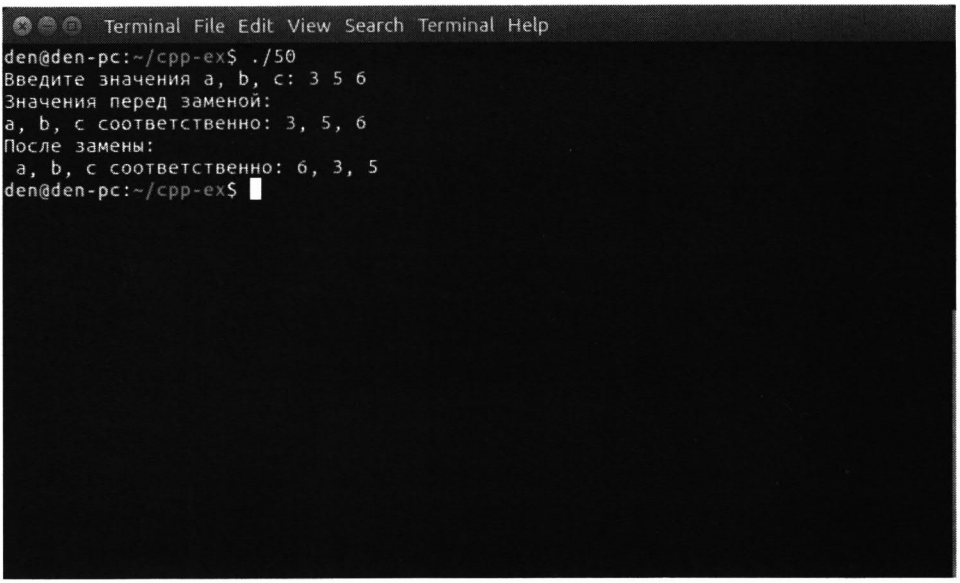
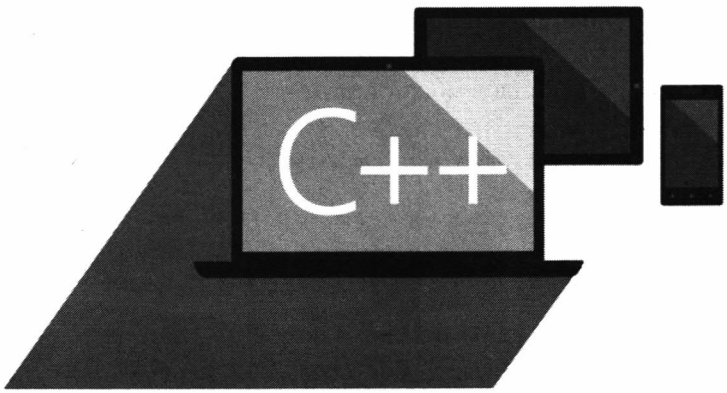


Рис. 8.2. Результат замены местами чисел с помощью указателей

Глава 9.

Работа со строками в C++



9.1. Строки в C++

Текстовые строки в C++ могут быть реализованы в одном из двух видов:

- либо в виде массива символов (так называемые C-строки);
- либо в виде объектов класса `string` (так называемые C++-строки);.

9.2. Строка как массив символов

Объявление строки как массива символов

Строка как массив символов может быть определена следующи образом:

```
char имя_переменной [размер_строки];
```

Например:

```
char str[20];
```

При объявлении можно сразу инициализировать строку-массив:

```
char string[10] = "abcdefghf"
```

Функции для работы со строками-массивами символов

Для работы с текстовыми строками, реализованными в виде символьных массивов, есть целый ряд встроенных функций. Одна из них уже упоминалась: это функция `strlen()` для определения длины строки, записанной в массив. В таблице 9.1 перечислены другие полезные в этом отношении функции.

Таблица 9.1. Функции для работы с текстовыми строками

Функция	Заголовок	Описание
strcpy(s1,s2)	<cstring>	Копирование строки s2 в строку s1
strcat(s1,s2)	<cstring>	Строка s2 присоединяется к строке s1
strcmp(s1,s2)	<cstring>	Сравнение строк s1 и s2: если строки равны, возвращается значение 0
strchr(s,ch)	<cstring>	Указатель на первую позицию символа ch в строке s
strstr(s1,s2)	<cstring>	Указатель на первую позицию подстроки s2 в строке s1
atoi(s)	<cstdlib>	Преобразование состоящей из цифр строки s в целое число типа int
atol(s)	<cstdlib>	Преобразование состоящей из цифр строки s в целое число типа long
atof(s)	<cstdlib>	Преобразование состоящей из цифр строки s в действительное число типа double
tolower(ch)	<cctype>	Преобразование буквенного символа ch к строчному формату
toupper(ch)	<cctype>	Преобразование буквенного символа ch к прописному формату

Понятно, что встроенных функций C++ для работы с текстом и символами намного больше, но здесь приведены те, что будут использоваться в книге. Для вызова функций необходимо подключить заголовок `<cstring>`, `<cctype>` или `<cstdlib>`, как это указано в таблице.

9.3. Строка как объект класса string

Строка как объякт класса string может быть определена гораздо проще:

```
string имя_переменной;
```

Например:

```
string str;
```

Только при это необъодимо подключить заголовочный файл <string>.

При задании строки-объекта ее можно сразу инициализировать, то есть присвоить значение по умолчанию. Например:

```
string str = "Hello, world!";
```

9.4. Практические примеры

Разница между различным представлением строк в C++

Строки - неотъемлемая часть любой программы. В этой части мы рассмотрим ряд примеров, демонстрирующих обработку строк в программах.

Чтобы получить размер строки (количества символов), хранимой строковым объектом (класса string) используйте функцию size():

Напишем программу, подсчитывающую сколько раз искомый символ встречается в заранее определенной строке. Программа в цикле "проходится" по строке и подчитывает частоту символа, то есть считает, сколько раз в строке встречается искомый символ. Код программы приведен в листинге 9.1.

Листинг 9.1. Подсчет частоты знаков строке

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

int main()
{
```

```
    string str = "Hello, world!";
    char findCh = 'l';
    int count = 0;

    for (int i = 0; i < str.size(); i++)
    {
        if (str[i] == findCh)
        {
            ++ count;
        }
    }

    cout << "Символ " << findCh << " встречается " << count <<
" раз\n";

    return 0;
}
```

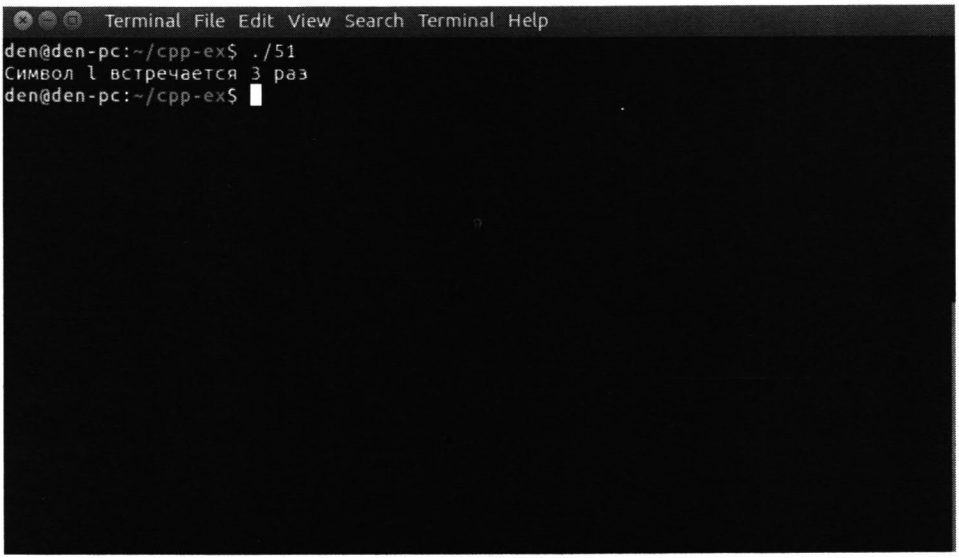


Рис. 9.1. Сколько раз встречается символ

В языке C используется несколько иной стиль строки. Типа string нет и строка представляет собой массив символов, то есть элементов типа char. Все строки в C заканчиваются знаком \0, означающим конец строки. Пример работы с такой строкой приведен в листинге 9.2.

Листинг 9.2. Посимвольный проход по строке в C-стиле

```
#include <iostream>

using namespace std;
int main()
{
   char c[] = "Hello, world!", findC = 'o';
   int count = 0;

   for(int i = 0; c[i] != '\0'; ++i)
   {
       if(findC == c[i])
           ++count;
   }
   cout << "Частота " << findC <<  " = " << count << endl;
   return 0;
}
```

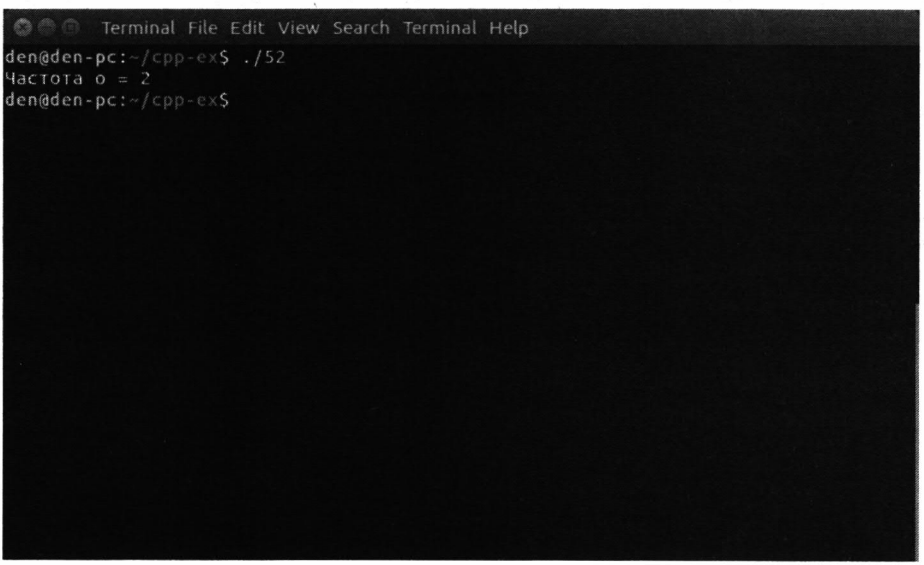


Рис. 9.2. Посимвольный проход по строке в C-стиле

Подсчет количества цифр и пробелов

Задача проста: дан текст и нужно вычислить:

- Общее количество символов
- Количество пробелов в тексте
- Количество цифр в тексте

То есть нужно написать довольно простую статистическую программу. Далее в этой книге мы напишем аналог приложения wc в Linux - полноценную программу для подсчета слов. А пока небольшая разминка.

Исходный код нашей программы приведен в листинге 9.3.

Листинг 9.3. Статистика о строке

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

int main()
{
    char line[150];
    int total, digits, spaces;

    total = digits = spaces = 0;

    cout << "Введите строку: ";
    cin.getline(line, 150);
    for(int i = 0; line[i]!='\0'; ++i)
    {
      ++total;

      if(line[i]>='0' && line[i]<='9')
        {
            ++digits;
        }
        else if (line[i]==' ')
        {
            ++spaces;
```

```
        }
    }

    cout << "Цифры: " << digits << endl;
    cout << "Пробелы: " << spaces << endl;
    cout << "Всего: " << total << endl;

    return 0;
}
```

Программа работает так: в цикле она перебирает все символы строки. Если будет найдена цифра, то увеличивается значение переменной digits, если найден пробел, то увеличивается переменная spaces, переменная total увеличивается при каждой итерации цикла. Количество символов можно было бы узнать с помощью функции strlen(), но раз у нас все равно есть цикл, то мы подсчитывали количество символов в нем.

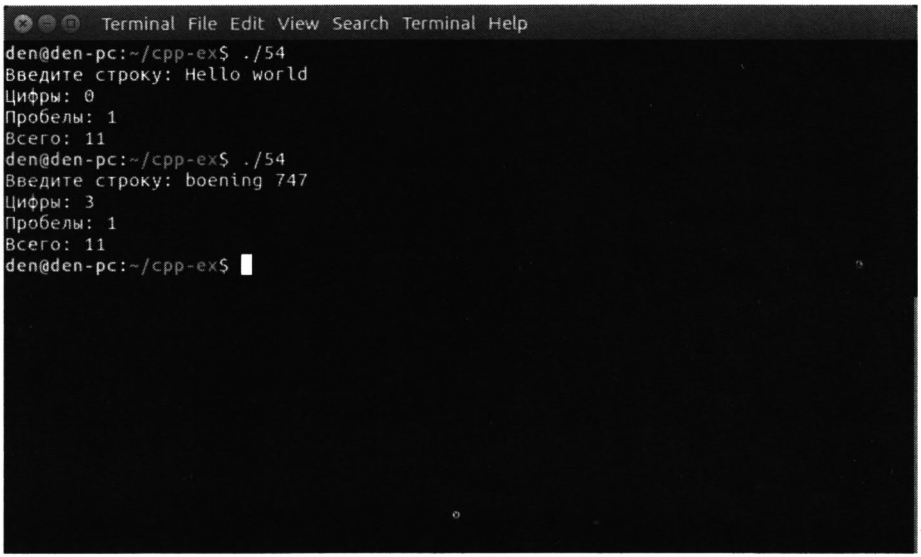


Рис. 9.3. Статистика о строке

Удаляем все символы в строке, кроме цифровых

В данном примере мы напишем программу, удаляющую из строки все символы, кроме цифровых. Такая задача часто встречается при очистке строк, содержащих номера телефонов, номера банковских карт и др.

Работает программа так:

- Пользователь вводит строку, которую мы записываем в переменную line
- В цикле мы проверяем, является ли символ цифровым.
- Если нет, то все символы после него, включая нулевой символ, смещаются на 1 позицию влево.

Код программы приводится в листинге 9.4.

Листинг 9.4. Удаляем все символы в строке, кроме цифровых

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

int main() {
    string line;
    cout << "Введите строку: ";
    getline(cin, line);

    for(int i = 0; i < line.size(); ++i)
    {
        if (!((line[i] >= 'a' && line[i]<='z') || (line[i] >= 'A'
&& line[i]<='Z')))
        {
            line[i] = '\0';
        }
    }
    cout << "Результат: " << line << endl;
    return 0;
}
```

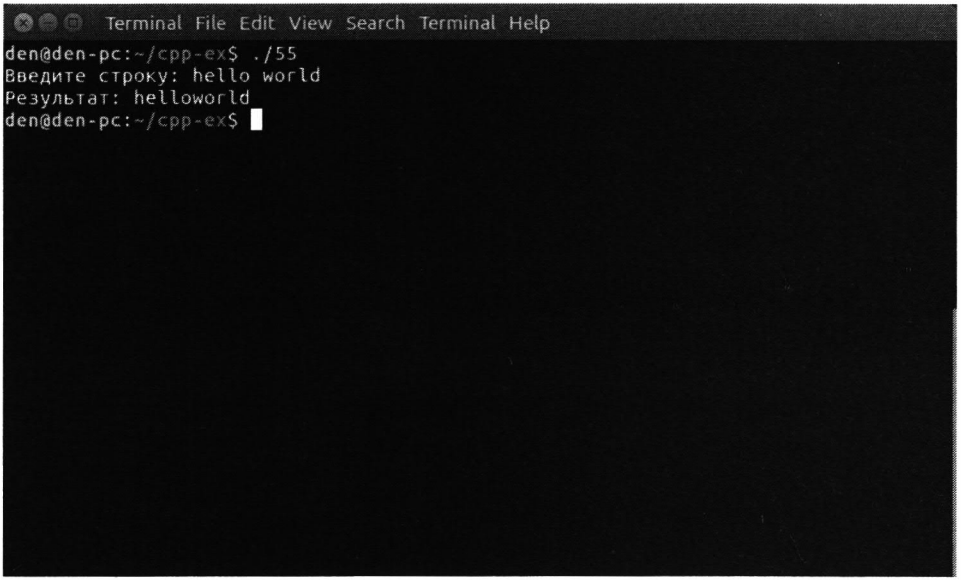


Рис. 9.4. Удаляем все символы, кроме цифровых

Определение длины строки

Для определения длины строки используется функция strlen() либо метод size():

```
string str = "Hello, world!";
// также можно использовать метод str.length()
cout << "String Length = " << str.size();

char str[] = "Hello, world!";
cout << "String Length = " << strlen(str);
```

Давайте напишем программу, определяющую длину строки, без использования готовых функций (лист. 9.5).

Вместо массива символов нам нужно использовать тип string, чтобы была возможность использовать функцию getline(). Функция getline() принимает два параметра - поток ввода (cin - стандартный ввод) и название переменной, в которую будет записана строка. Если мы будем использовать оператор >>, то строка будет введена до первого пробельного символа. Функция getline() читает строку со всеми пробельными символами. Далее

алгоритм программы тот же - мы "проходимся" по всем символам строки (пока не будет встречен конец строки) и увеличиваем счетчик i.

Листинг 9.5. Подсчитываем количество символов в строке

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

int main()
{
    int i;
    string s;

    cout << "Введите строку: ";
    getline(cin, s);

    for(i = 0; s[i] != '\0'; ++i);

    cout << "Длина: " << i << "\n";
    return 0;
}
```

Объединение нескольких строк в одну

Для конкатенации (объединения) двух строк обычно используется функция strcat(). Но мы напишем программу, показывающую, как устроена эта функция - чтобы вы знали, как можно объединить две строки без ее использования.

Листинг 9.6. Конкатенация двух строк без функции strcat()

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

int main()
{
    string s1, s2, result;

    cout << "Введите строку s1: ";
    getline (cin, s1);                    // читаем строку 1
```

```
    cout << "Введите строку s2: ";
    getline (cin, s2);                    // читаем строку 2

    result = s1 + s2;

    cout << "Результат = "<< result << endl;

    return 0;
}
```

Если мы используем строки в стиле C (то есть массивы символов), то код будет несколько иным:

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

int main()
{
    char s1[50], s2[50], result[100];

    cout << "Введите строку s1: ";
    cin.getline(s1, 50);

    cout << "Введите строку s2: ";
    cin.getline(s2, 50);

    strcat(s1, s2);

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2;

    return 0;
}
```

На этот раз мы использовали функцию strcat(), чтобы продемонстрировать ее работу.

Копирование двух строк

При использовании объекта типа string для копирования строк вы можете использовать просто оператор присваивания =. При использовании строк в стиле C лучшим решением является использование функции strcpy. Рассмотрим оба варианта (лист. 9.7 и лист. 9.8).

Листинг 9.7. Копирование объектов типа string

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

int main()
{
    string s1, s2;

    cout << "Введите строку s1: ";
    getline (cin, s1);

    s2 = s1;

    cout << "s1 = "<< s1 << endl;
    cout << "s2 = "<< s2 << endl;

    return 0;
}
```

Листинг 9.8. Копирование строк в стиле C

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char s1[100], s2[100];

    cout << "Введите строку s1: ";
    cin.getline(s1, 100);
```

```
    strcpy(s2, s1);

    cout << "s1 = "<< s1 << endl;
    cout << "s2 = "<< s2 << endl;

    return 0;
}
```

Если нужно обойтись без функции strcpy(), то код будет примерно таким:

```
    for(i = 0; s1[i] != '\0'; ++i)
    {
        s2[i] = s1[i];
    }

    s2[i] = '\0';
```

Операторы сравнения строк

Напишем программу, которая отсортирует в лексикографическом порядке массив строк. Для сравнения строк мы используем функцию strcmp(), а функция strcpy() используется для копирования строки в переменную temp в процессе сортировки. Будем работать со строками, как массивами символов.

Алгоритм прост: во время перебора двухмерного массива строк str мы сравниваем две строки. Если первая строка больше (лексикографически), чем вторая, то функция strcmp() возвращает 0. При этом мы меняем местами эти две строки и так до тех пор, пока строки не будут расположены в лексикографическом порядке.

Примечание. Поскольку это демонстрационная программа, то количество элементов ограничено. При желании вы можете увеличить это значение

Листинг 9.9. Сортировка в лексикографическом порядке

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

```
int main()
{
    string str[10], temp;

    cout << "Введите 10 слов: " << endl;
    for(int i = 0; i < 10; ++i)
    {
      getline(cin, str[i]);
    }

    for(int i = 0; i < 9; ++i)
       for( int j = i+1; j < 10; ++j)
       {
          if(str[i] > str[j])
          {
            temp = str[i];
            str[i] = str[j];
            str[j] = temp;
          }
    }
```

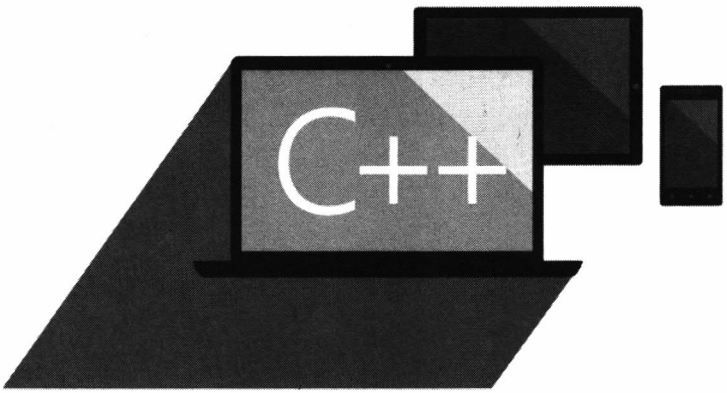
```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./59
Введите 10 слов:
banana
apple
car
bus
train
house
home
freedoom
wulf
zero
Сортируем:
apple
banana
bus
car
freedoom
home
house
train
wulf
zero
den@den-pc:~/cpp-ex$
```

Рис. 9.5. Сортировка в лексикографическом порядке

```
    cout << "Сортируем: " << endl;

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

Глава 10.

Структуры и объединения в C++



10.1. Структуры

Под структурами подразумевают группу переменных, объединенных общим именем. Удобство структуры состоит в первую очередь в том, что она позволяет группировать разнородные данные, что бывает весьма полезно при работе со всевозможными базами данных и записями.

Объявление структуры начинается с ключевого слова `struct`, после которого следует имя структуры и, в фигурных скобках, перечисляются поля структуры (типы и имена переменных, входящих в структуру).

Общий вид объявления структуры следующий:

```
struct имя{
   тип1 поле1;
   тип2 поле2;
   ...
   типN полеN;

   }список_переменных;
```

После фигурных скобок, заканчивающих непосредственно описание структуры, можно указать (а можно и не указывать) список переменных структуры. Дело в том, что само по себе описание структуры эту структуру не создает. Описание структуры – это всего лишь некий шаблон, по которому впоследствии создаются переменные. Процесс создания переменных структуры можно не откладывать в долгий ящик, а создать их сразу, указав список с названиями после описания структуры.

С точки зрения использования в программе имеет смысл говорить лишь о переменных структуры. Фактически, переменная структуры – это объект, который имеет имя (имя переменной структуры) и поля, тип и названия которых определяются описанием структуры. Чтобы в программе создать переменную структуры, необходимо, указать имя структуры, в соответствии с описанием которой создается переменная, и имя этой переменной. Другими словами, переменная структуры в программе создается точно так же, как и переменная любого базового типа, только вместо названия типа переменной указывается название структуры.

Сама по себе переменная структуры интерес представляет больше спортивный, чем практический. Все данные, которые могут понадобиться в процессе выполнения программы, записаны в поля переменной. Обращение к полю переменной структуры осуществляется через так называемый точечный синтаксис (стандартный синтаксис для объектно-ориентированного программирования) – указывается имя переменной структуры, и, через точку, имя поля, к которому выполняется обращение, т.е. в формате `структура.поле`.

Рассмотрим пример. Цель этого примера - продемонстрировать, как можно хранить информацию о студенте (имя, номер студенческого билета, номер группы) в структуре. В этой программе мы создадим структуру student, у нее будет три члена - name (string), roll (integer), group (integer). Для хранения информации мы будем использовать переменную s. Также будет показано, как вывести информацию из структуры на экран. Код программы находится в листинге 10.1.

Листинг 10.1. Храним информацию о студенте в структуре

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

struct student
{
    char name[50];
    int roll;
    float mark;
};

int main()
```

```
{
    student s;
    cout << "Инфо о студенте" << endl;
    cout << "Имя: ";
    cin >> s.name;
    cout << "Курс: ";
    cin >> s.roll;
    cout << "Оценка: ";
    cin >> s.mark;

    cout << "\nОтображаем инфо," << endl;
    cout << "Имя: " << s.name << endl;
    cout << "Курс: " << s.roll << endl;
    cout << "Оценка: " << s.mark << endl;
    return 0;
}
```

Обратите внимание на использование символа & при формировании (заполнении) структуры и при ее отображении.

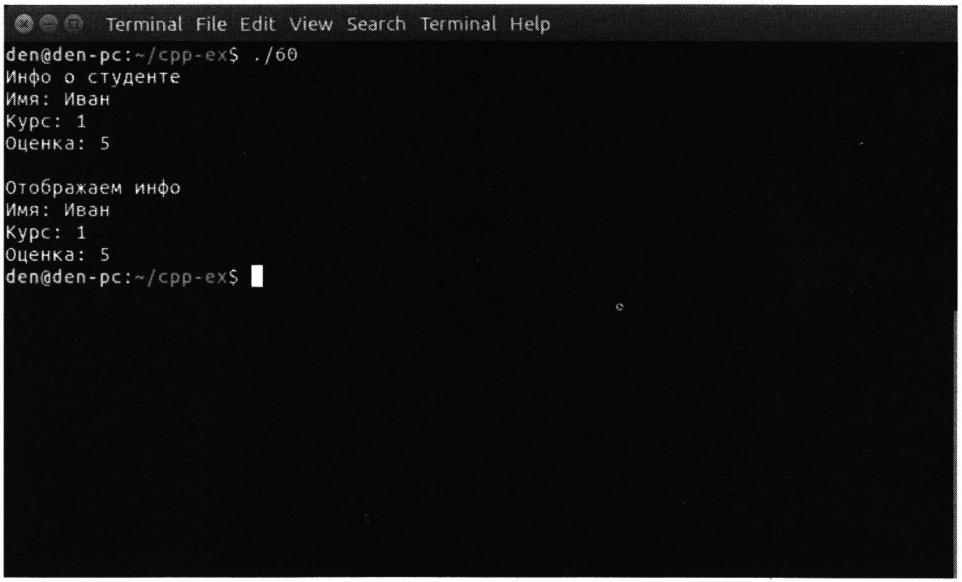


Рис. 10.1. Результат работы программы

10.2. Объединения

Под объединениями подразумевают область памяти, в которой свои значения хранит сразу несколько различных переменных, но единовременно только одна. Общий синтаксис объявления объединения подразумевает использование ключевого слова `union` и имеет следующий вид:

```
union имя_объединения{
   тип_переменной1 имя_переменной1;
   тип_переменной2 имя_переменной2;
   ...
   тип_переменнойN имя_переменнойN;
   } список экземпляров;
```

Как и в случае структуры, само по себе объявление объединения не приводит к созданию новых переменных. Поэтому необходимо создать экземпляр объединения. При объявлении экземпляра объединения в качестве типа переменной указывают имя объединения.

Ссылка на члены экземпляров объединения выполняется с помощью того же синтаксиса, что и ссылка на поля структуры: член объединения указывается через точку после имени экземпляра объединения, или через оператор `->` после указателя на экземпляр объединения.

10.3. Операции над структурами. Сложение двух структур

Представим, что у нас есть две структуры, содержащие информацию о расстоянии - в шагах и в метрах. Программа из этого примера выполнит сложение двух структур и отобразит результат на экране.

Листинг 10.2. Сложение двух структур

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

struct Distance{
    int f;
    float meters;
}d1 , d2, sum;
```

```
int main()
{
    cout << "Первая структура" << endl;
    cout << "Количество шагов: ";
    cin >> d1.f;
    cout << "Количество метров: ";
    cin >> d1.meters;

    cout << "\nВторая структура" << endl;
    cout << "Количество шагов: ";
    cin >> d2.f;
    cout << "Количество метров: ";
    cin >> d2.meters;

    sum.f = d1.f + d2.f;
    sum.meters = d1.meters + d2.meters;

    cout << endl << "Сумма = " << sum.f << " шагов  ";
    cout << sum.meters << " метров" << endl;
    return 0;
}
```

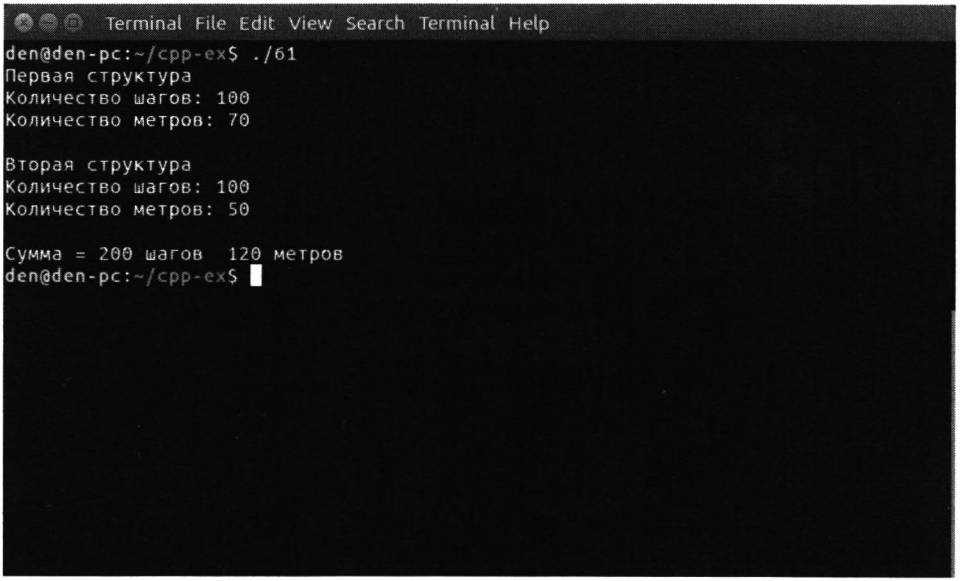


Рис. 10.2. Результат работы программы

В этой программе мы определили структуру Distance, состоящую из двух членов - feet (int) и m (float). Первый член - это количество шагов, второй - количество метров. Далее мы создаем три переменных типа Distance, выполняем сложение структур поэлементно и выводим результат.

Рассмотрим еще одни пример: сложение двух комплексных чисел с использованием структуры и передачей структуры функции.

Данная программа похожа на предыдущую, но в ней сложение будет выполнять функция add(). В нее мы будем передавать две структуры и она же будет вычислять результат. Код программы приведен в листинге 10.3.

Листинг 10.3. Сложение двух структур с использованием функции

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

typedef struct complex
{
    float real;
    float imag;
} complexNumber;

complexNumber add(complex, complex);

int main()
{
    complexNumber n1, n2, temporaryNumber;
    char signOfImag;

    cout << " Первое комплексное число" << endl;
    cout << " Введите действительную и мнимую часть
соответственно:" << endl;
    cin >> n1.real >> n1.imag;

    cout << endl << " Второе комплексное число" << endl;
    cout << " Введите действительную и мнимую часть
соответственно:" << endl;
    cin >> n2.real >> n2.imag;

    signOfImag = (temporaryNumber.imag > 0) ? '+' : '-';
```

```
    temporaryNumber.imag = (temporaryNumber.imag > 0) ?
temporaryNumber.imag : -temporaryNumber.imag;

    temporaryNumber = add (n1, n2);
    cout << "Сумма = "  << temporaryNumber.real <<
temporaryNumber.imag << "i";
    cout << endl;
    return 0;
}

// Функция выполняет сложение комплексных чисел
complexNumber add(complex n1,complex n2)
{
      complex temp;
      temp.real = n1.real+n2.real;
      temp.imag = n1.imag+n2.imag;
      return(temp);
}
```

Функция add() вычисляет сумму и возвращает переменную temp функции main(). Результат работы программы показан на рис. 10.3.

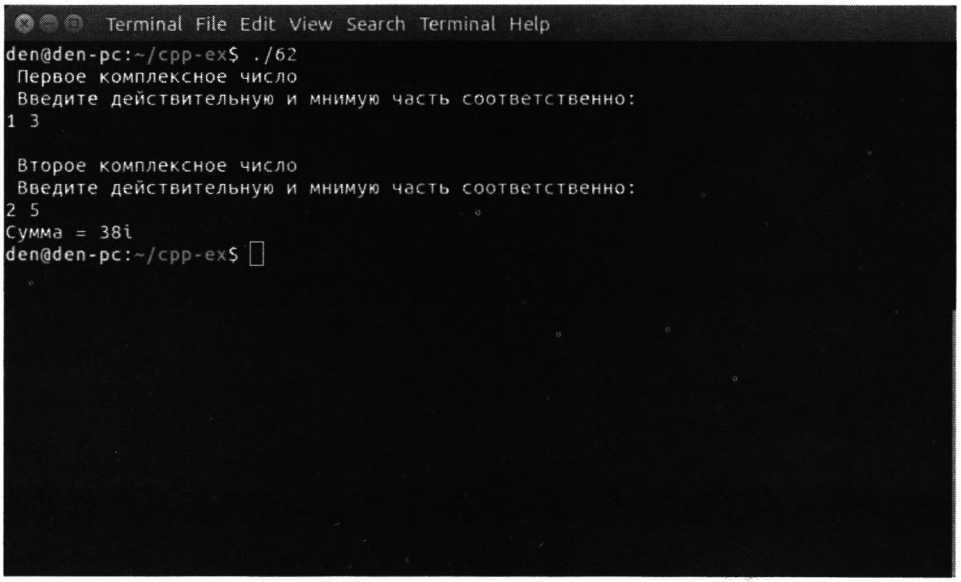


Рис. 10.3. Результат работы программы: сложение комплексных чисел

Усложним нашу задачу. Напишем программу, вычисляющую разницу между двумя периодами времени. Для этого мы определим структуру TIME (содержащую информацию о часах, минутах, секундах) и пользовательскую функцию differenceBetweenTimePeriod(), которой мы передадим три структуры. Первые две содержат начальное и конечное время соответственно, третья - результат, то есть разницу между этими двумя структурами.

В этой программе мы просим пользователя вести два периода времени, мы сохраняем их в переменных типа TIME - это наша структура, содержащая информацию о времени. Далее, функция differenceBetweenTimePeriod() вычисляет разницу. Обратите внимание, поскольку мы используем технику вызова по ссылке, то данная функция ничего не возвращает в функцию main() - результат сразу записывается в переменную diff.

Листинг 10.4. Вычисление разницы между двумя периодами времени

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

struct TIME
{
  int seconds;
  int minutes;
  int hours;
};

void computeTimeDifference(struct TIME, struct TIME, struct TIME *);

int main()
{
    struct TIME t1, t2, difference;

    cout << "Начальное время." << endl;
    cout << "Введите часы, минуты, секунды: ";
    cin >> t1.hours >> t1.minutes >> t1.seconds;

    cout << "Конечное время." << endl;
    cout << "Введите часы, минуты, секунды: ";
    cin >> t2.hours >> t2.minutes >> t2.seconds;
```

```
    computeTimeDifference(t2, t1, &difference);

    cout << endl << "Разница: " << t1.hours << ":" << t1.minutes
<< ":" << t1.seconds;
    cout << " -> " << t2.hours << ":" << t2.minutes << ":" <<
t2.seconds;
    cout << " = " << difference.hours << ":" << difference.minutes
<< ":" << difference.seconds;
    cout << endl;
    return 0;
}
void computeTimeDifference(struct TIME t1, struct TIME t2, struct
TIME *difference){

    if(t2.seconds > t1.seconds)
    {
        --t1.minutes;
        t1.seconds += 60;
```

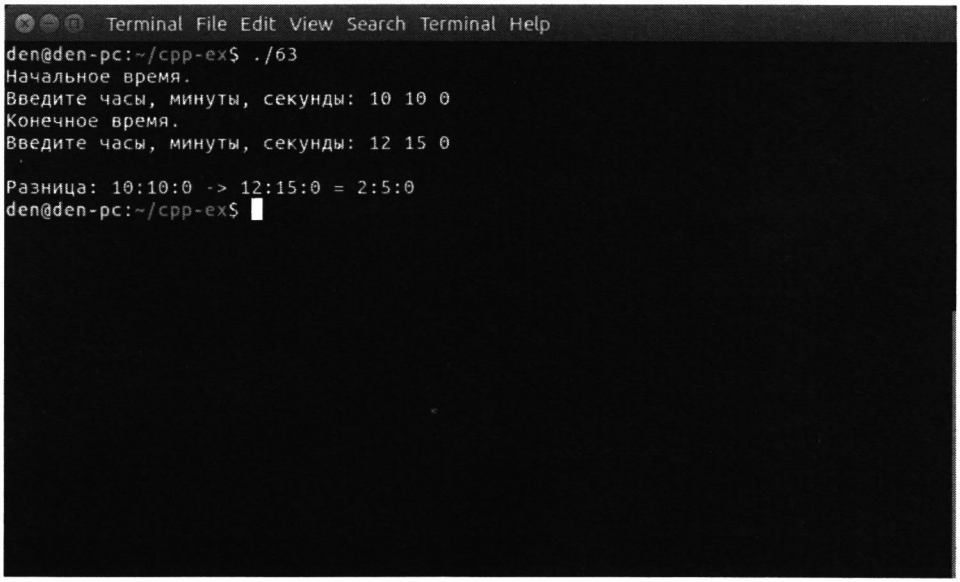


Рис. 10.4. Разница между двумя периодами времени

```
    }

    difference->seconds = t1.seconds - t2.seconds;
    if(t2.minutes > t1.minutes)
    {
        --t1.hours;
        t1.minutes += 60;
    }
    difference->minutes = t1.minutes-t2.minutes;
    difference->hours = t1.hours-t2.hours;
}
```

10.4. Массивы структур

В примере, рассмотренном в начале главы, было показано, как создать одну структуру, хранящую информацию о студенте. Представим, что нам нужно хранить несколько структур в памяти - для нескольких объектов. Когда мы знаем количество объектов, мы можем объявить массив структур:

```
struct student
{
    char name[50];
    int roll;
    float mark;
} s[10];
```

Пример из листинга 10.5 демонстрирует, как можно работать с массивом структур на C++. Сначала мы в цикле for заполняем информацию о студентах, а затем с помощью этого же списка - выводим ее.

Листинг 10.5. Работа с массивом структур

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

struct student
{
    char name[50];
    int roll;
```

```
    float mark;
} s[5];

int main()
{
    cout << "Вводим информацию о студентах: " << endl;

    // storing information
    for(int i = 0; i < 5; ++i)
    {
        s[i].roll = i+1;
        cout << "Курс: ";
        cin >> s[i].roll;

        cout << "Имя: ";
        cin >> s[i].name;

        cout << "Оценка: ";
        cin >> s[i].mark;

        cout << endl;
    }
```

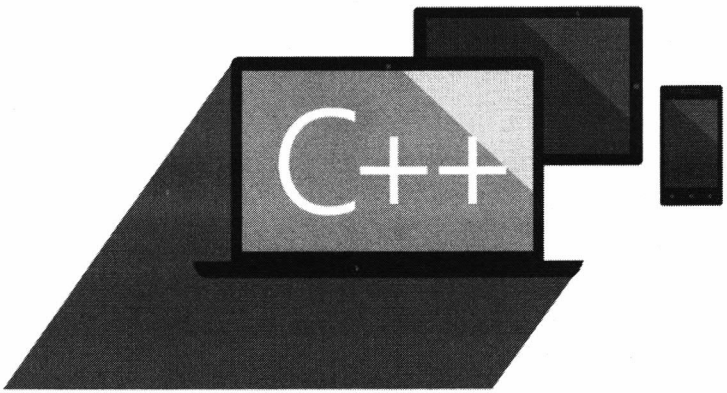
```
Terminal File Edit View Search Terminal Help
Оценка: 3

Отображаем информацию:

Курс: 1
Имя: Иван
Оценка: 5

Курс: 2
Имя: Николай
Оценка: 4

Курс: 3
Имя: Анна
Оценка: 5

Курс: 4
Имя: Василий
Оценка: 2

Курс: 5
Имя: Юрий
Оценка: 3
den@den-pc:~/cpp-ex$
```

Рис. 10.5. Работа с массивом структур

```
    cout << "Отображаем информацию: " << endl;

    for(int i = 0; i < 5; ++i)
    {
        cout << "\nКурс: " << i+1 << endl;
        cout << "Имя: " << s[i].name << endl;
        cout << "Оценка: " << s[i].mark << endl;
    }

    return 0;
}
```

Глава 11.

Программирование работы с файлами на C++



Ввод/вывод в C++ реализован посредством потоков и очень похож на консольный ввод/вывод, который также реализован с помощью потоков. Поэтому многое, что вам известно о консольном вводе/выводе, применимо и к файлам.

11.1. Возможности C++ для программирования работы с файлами

Для реализации файлового ввода/вывода нужно подключить заголовок <fstream>. В нем определено несколько классов:

- ifstream
- ofstream
- fstream

Первый используется для ввода, второй - для вывода, третий - для ввода и вывода. Прежде, чем открыть файл, нужно создать поток. Далее создаются потоки ввода, вывода и ввода/вывода:

```
ifstream in;
ofstream out;
fstream io;
```

После этого нужно использовать функцию open() непосредственно для открытия файла. Этой функции нужно передать имя файла и режим открытия:

```
void ifstream::open(const char *имя_файла, openmode режим);
void ofstream::open(const char *имя_файла, openmode режим);
void fstream::open(const char *имя_файла, openmode режим);
```

Значение режим типа openmode может принимать следующие значения:

- ios::app - используется для открытия файла с целью добавления информации в его конец. Применяется только к файлам, открытым для вывода.

- ios::ate - вызывает поиск конца файла, но операции ввода/вывода могут быть выполнены в любой части файла.

- ios::binary - двоичный режим. По умолчанию все файлы открываются в текстовом режиме, в котором имеет место преобразование некоторых символов, например, последовательность символов \r\n превращается в \n. Если файл открывается в двоичном режиме, этого преобразования нет. Любой файл (независимо от того, какие данные он содержит) может быть открыт, как в текстовом, так и в двоичном режиме. Разница только в наличии или отсутствии упомянутого преобразования.

- ios::in, ios::out - соответственно, задает режим для ввода и режим для вывода.

- ios::trunk - приводит к удалению содержимого файла при его открытию и усечению его до нулевой длины.

Пример открытия файла:

```
ofstream file;          // создаем поток
   file.open("test");        // попытка открыть файл
   // Проверяем, получилось ли открыть файл
if (!file) {
     cout << "Error!";
}
```

Для закрытия файла используется функция-член close:

```
   file.close();
```

Еще одна полезная функция - eof(), позволяющая проверить, достигнут ли конец файла, ее полезно использовать при чтении до конца файла.

Ввод/вывод реализован с помощью операторов >>/<< - как и в случае консоли. Только вместо потоков cout и cin нужно использовать поток вашего файла.

Давайте напишем программу, которая создает файл, выводит в него строку, закрывает файл (лист. 11.1).

Листинг 11.1. Запись в файл

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

int main()
{
  // создаем поток и сразу открываем файл
  ofstream fout("test.txt");

  if (!fout) {
    cout << "Не могу открыть файл для записи!";
    return 1;
  }
  // Записываем в поток две строки
  fout << "Hello, world!\n";
  fout << "Yet another line\n";

  fout.close();

  cout << "Готово!" << endl;

  return 0;
}
```

Итак, у нас есть файл test.txt, в который мы только что записали две строчки. В следующих двух примерах будет показано, как прочитать информацию из этого файла.

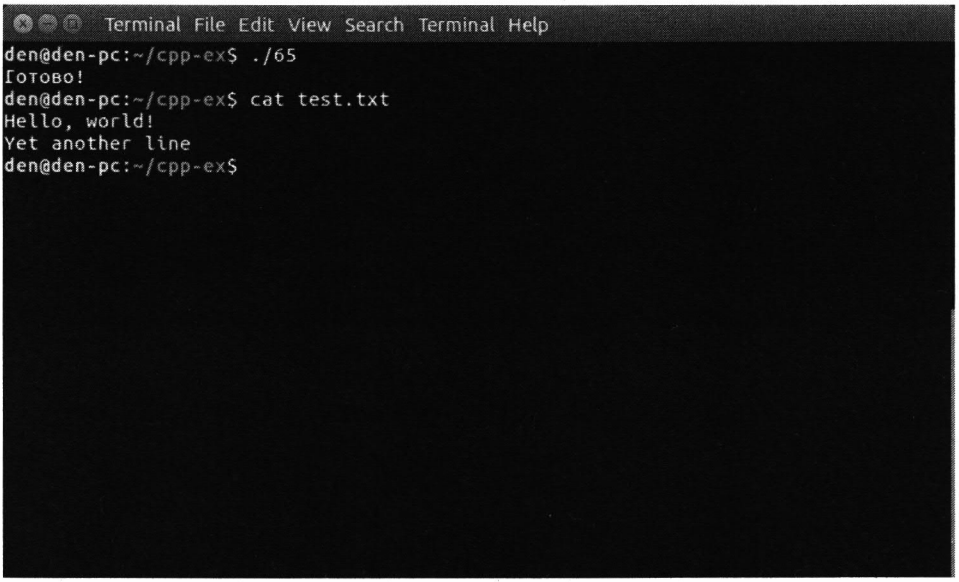


Рис. 11.1. Результат работы программы

11.2. Чтение из файла

11.2.1. Посимвольное чтение из файла

Следующий пример показывает, как осуществляется посимвольное чтение информации из файла. Мы прочитаем каждый символ файла и выведем его на экран.

Листинг 11.2. Чтение информации построчно

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

int main()
{
  // открываем поток для ввода
  ifstream fin("test.txt");
  // не пропускать пробелы
  fin.unsetf(ios::skipws);
  if (!fin) {
```

```
    cout << "Не могу открыть файл для чтения!";
    return 1;
  }

  char ch;
  // читаем файл посимвольно
  while (!fin.eof()) {
    fin >> ch;
    cout << ch;
  }

  cout << endl;
  fin.close();

  return 0;
}
```

Посмотрите на рис. 11.2. На нем продемонстрирована работа следующего оператора:

```
fin.unsetf(ios::skipws);
```

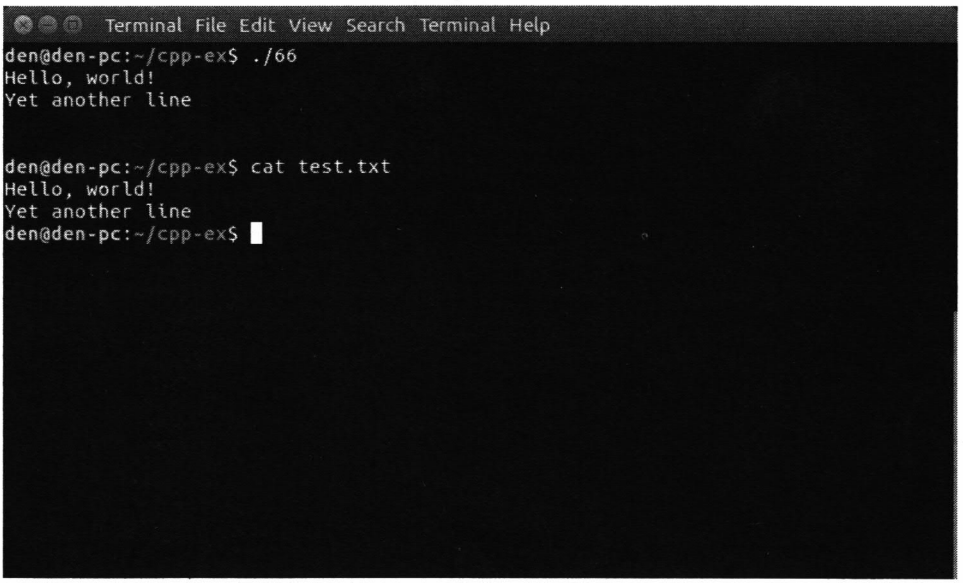


Рис. 11.2. Результат работы программы из листинга 66

Сначала мы запускаем программу, которая скомпилирована без него. Программа пропускает пробелы, хотя они были записаны в файл, что подтверждает команда cat, выводящая содержимое этого файла. Затем я перекомпилировал программу, запретив ей пропуск пробелов - теперь они также выведены на консоль.

11.2.2. Построчное чтение из файла

Наша программа читает файл посимвольно. Это не всегда удобно. На практике часто нужно читать файл построчно. Для этого нужно использовать функцию getline(). Пример построчного чтения файла приведен в листинге 11.3. В цикле while производится построчное чтение - из файла читается строка и присваивается переменной s, содержимое которой выводится на экран. Далее к строке добавляется + - как пример обработки строки - и модифицированная строка также выводится на экран (консоль).

Листинг 11.3. Построчное чтение файла на C++

```
#include <iostream>        // консольный ввод/вывод
#include <string>    // операции со строками
#include <fstream>         // файловый ввод/вывод

using namespace std;

int main(){
    string s;      // сюда будем читать строки из файла
    ifstream file("data.dat");

    // пока не достигнут конец файла читать строки из потока
    // file в строковую переменную s
    while(getline(file, s)){
        cout << s << endl;              // выводим s на экран
        s += "+";          // что-нибудь делаем со строкой
        cout << s << endl;              // снова выводим
    }

    file.close();          // закрываем поток

    return 0;
}
```

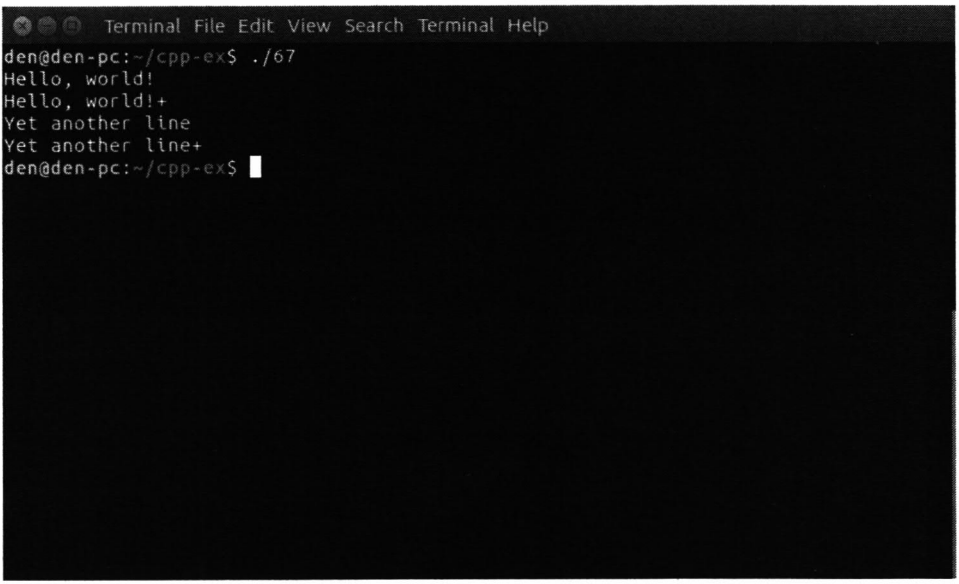
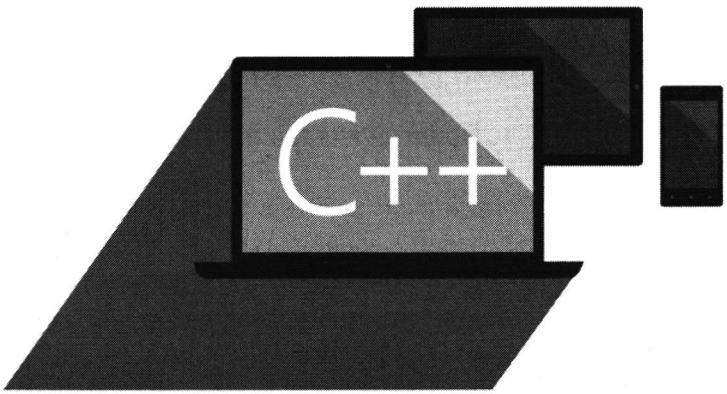


Рис. 11.3. Результат работы программ

Глава 12.

Объектно-ориентированное программирование на C++



Объектно-ориентированное программирование является стандартом в технологии современного программирования и в подавляющее большинство новейших средств разработки программных продуктов заложен именно объектный подход. C++ обладает всеми необходимыми возможностями современного объектно-ориентированного языка программирования.

12.1. Классы и объекты. Инкапсуляция

12.1.1. Понятие класса и объекта

Классом называется описание некоторой структуры программы, обладающей набором внутренних переменных — свойств, и функций (процедур), имеющих доступ к свойствам — методов. Процесс объединения переменных и методов, в результате которого и получается класс, называется инкапсуляцией.

Итак, класс — это всего лишь описание, аналогичное описанию типа данных, и недоступное для прямого использования в программе. Для получения доступа к свойствам и методам класса (за исключением методов класса, описанных ниже) необходимо создать экземпляр класса, называемый также объектом. В этом есть некоторое сходство между классами и структурами. Однако классы являются гораздо более мощным инструментом программирования в силу присущих им, в отличие от записей, механизмов наследования и свойств полиморфизма, описанных ниже.

12.1.2. Структура класса

Описание класса

Для объявления класса используется ключевое слово class:

```
class имя_класса {
   закрытые методы (функции) и переменные класса
public:
   открытые методы и переменные класса
} [список объектов класса];
```

Примечание. Обратите внимание, что ; после объявления класса - обязательна!

Обратите внимание, что список объектов - необязательная часть. Вы можете объявить список объектов позже в программе, когда вам это будет нужно.

Функции (методы) и переменные, объявленные внутри объявления класса, называют членами (members) класса. Чтобы не возникало путаницы, функции класса часто называют методами класса.

Все члены класса, объявленные после служебного слова public, являются публичными (общедоступными) - их можно использовать, как и другим членам класса, так и в любой другой части программы, в которой находится этот класс.

Практический пример создания класса на C++

Сейчас мы разработаем собственный класс. Чтобы пример имел практическую ценность, мы разработаем класс стека, который можно использовать для хранения символов.

В листинге 12.1 приводится код программы, имитирующей функционал стека (структуры типа LIFO - Last In, First Out).

Листинг 69. Стек на C++

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

```
const int SIZE = 26;

class stack {
  char stck[SIZE];
  int tos;
public:
  void init();
  void push(char ch);
  char pop();
};

void stack::init()
{
  tos = 0;
}

void stack::push(char ch)
{
  if (tos==SIZE) {
     cout << "Стек полон!" << endl;
     return;
  }
  stck[tos] = ch;
  tos++;
}

char stack::pop() {
  if (tos==0) {
    cout << "Стек пуст!" << endl;
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack s1, s2;
  int i;

  s1.init();
```

```
  s2.init();

  s1.push('a');
  s2.push('b');
  s1.push('c');
  s2.push('d');
  s1.push('e');
  s2.push('f');

  for (i=0; i<3; i++) cout << s1.pop() << " ";
  cout << endl;
  for (i=0; i<3; i++) cout << s2.pop() << " ";
  cout << endl;
}
```

Для компиляции программы с помощью g++ используйте опцию -pedantic, иначе получите сообщение об ошибке, связанное с использованием константы SIZE (другие компиляторы, возможно, не будут "ругаться"):

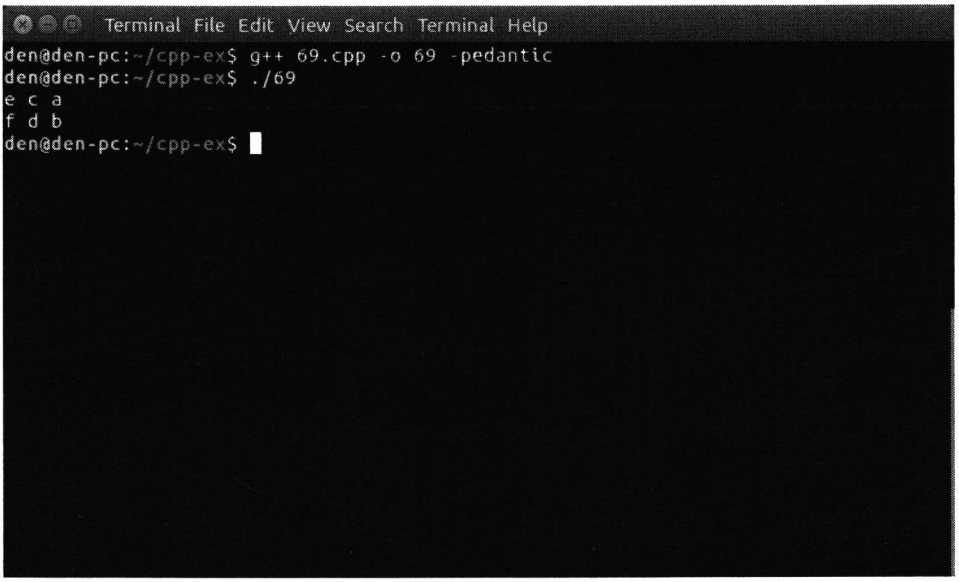


Рис. 12.1. Результат работы программы из листинга 69

Теперь проанализируем программу. Класс stack содержит две приватных (закрытых) переменных stck и tos. Массив stck содержит символы, добавленные в стек, а tos содержит индекс вершины стека. Функция pop() выталкивает символ со стека, функция push() добавляет символ в стек, а функция init() инициализирует стек.

Внутри функции main() мы создаем два стека - s1 и s2, в который добавляем символы. Оба объекта стека абсолютно независимы друг от друга и нет никакого способа заставить стек s1 повлиять на s2 и наоборот. У каждого объекта собственная копия членов stck и tos. Вы должны понимать, что хотя все объекты класса имеют общие функции-члены, каждый объект работает со своими собственными данными.

12.2. Конструкторы и деструкторы

Обратите внимание на программу из листинга 12.2. После создания объектов типа stack, мы вызываем функцию init() для каждого объекта, которая выполняет инициализацию стека, а именно устанавливает tos в 0. Если этого не сделать, значение tos не будет определено и дальше все зависит от компилятора. Некоторые могут инициализировать переменную, установив ее в 0, некоторые же ничего не будут делать, тогда ошибка времени выполнения гарантирована.

Было бы хорошо, чтобы функция инициализации вызывалась автоматически. Ведь вы можете легко забыть ее вызвать или вызвать, но не для всех объектов - для s1, например, вызовите, а для s2 - забудете. Да и вообще это неудобно - вызывать функцию инициализации вручную.

Разработчики языка C++ также так думаю, поэтому они разработали конструкторы и деструкторы. Конструктор класса вызывается всякий раз при создании объекта этого класса. Код нашей функции init идеально было бы поместить в конструктор класса - тогда вы никогда не забудете инициализировать объект.

Функция-деструктор вызывается при удалении объекта. Код этой функции обычно содержит освобождение выделенной памяти, закрытие соединения с базой данных или Интернет-сервером, закрытие файла и т.д.

Наша программа ничего такого не делает, поэтому в деструкторе прямой необходимости нет.

Функция-конструктор называется так же, как и класс. Объявляется он так:

```
stack::stack()
{
}
```

Имя деструктора предваряется тильдой ~:

```
stack::~stack()
{
}
```

Код программы, использующей конструкторы и деструкторы, приведен в листинге 12.2. Обратите внимание: теперь init() можно не вызывать, но саму функцию init() я оставил в классе - вдруг понадобится в процессе работы со стеком выполнить заново инициализацию. Инициализация стека осуществляется автоматически с помощью конструктора. Деструктор просто выводит сообщение о том, что он работает - для демонстрации его возможностей (в нашей простой программе в нем нет необходимости).

Листинг 12.2. Стек с конструктором и деструктором

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

const int SIZE = 26;

class stack {
  char stck[SIZE];
  int tos;
public:
  stack();
  ~stack();
  void init();
  void push(char ch);
  char pop();
};

stack::stack()
{
```

```
  cout << "Инициализируем стек\n";
  tos = 0;
}

stack::~stack()
{
  cout << "Работает деструктор...\n";
}

void stack::init()
{
  tos = 0;
}

void stack::push(char ch)
{
  if (tos==SIZE) {
     cout << "Стек полон!" << endl;
     return;
  }
  stck[tos] = ch;
  tos++;
}

char stack::pop() {
  if (tos==0) {
    cout << "Стек пуст" << endl;
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack s1, s2;
  int i;

  s1.push('a');
  s2.push('b');
  s1.push('c');
```

```
  s2.push('d');
  s1.push('e');
  s2.push('f');

  for (i=0; i<3; i++) cout << s1.pop() << " ";
  cout << endl;
  for (i=0; i<3; i++) cout << s2.pop() << " ";
  cout << endl;
}
```

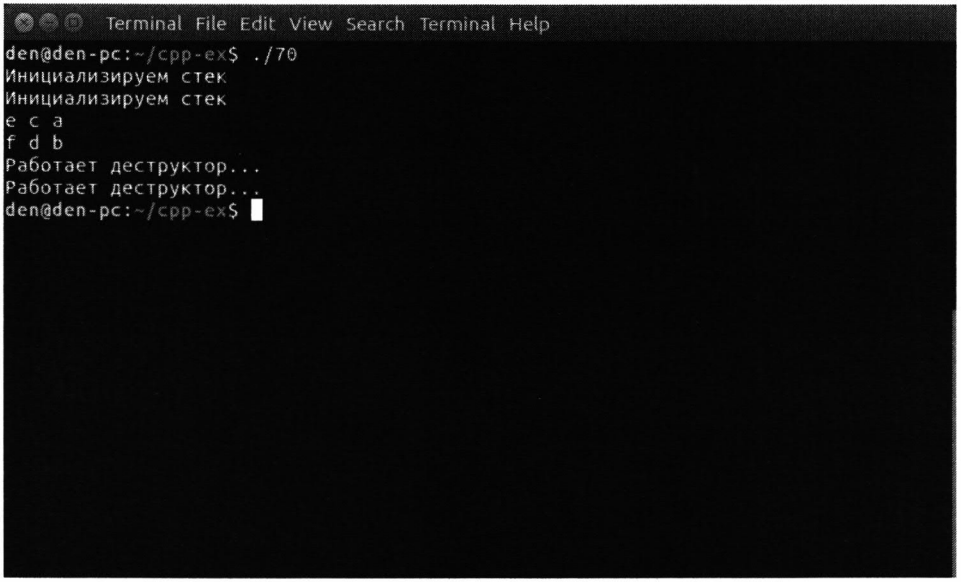


Рис. 12.2. Конструктор и деструктор

У нашей программы есть один недостаток. Она выводит строки Initializing stack и Destructor is working, но при этом непонятно, какой стек инициализируется и какой разрушается.

Конструкторы могут принимать параметры. Это свойство конструкторов мы будем использовать для идентификации стеков. Мы добавим еще один член - stackID типа int, затем добавим параметр id к нашему конструктору:

```
stack::stack(int id)
{
  stackID = id;
```

```
  cout << "Initializing stack " << stackID << endl;
  tos = 0;
}
```

Конструктор устанавливает ID стека и выводит информацию об этом. Аналогично, деструктор будет выглядеть так:

```
stack::~stack()
{
  cout << "Деструктор стека #" << stackID << " выполняется...\n";
}
```

Инициализация объектов типа stack будет выглядеть так:

```
stack s1(1), s2(2);
```

Полный код программы приведен в листинге 12.3. Результат выполнения изображен на рис. 12.3.

Листинг 12.3. Идентификация стеков

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

const int SIZE = 26;

class stack {
  char stck[SIZE];   // Элементы стека
  int tos;         // Вершина стека
  int stackID;       // ID стека
public:
  stack(int id);           // Конструктор класса
  ~stack();          // Деструктор
  void init();       // Инициализация стека
  void push(char ch);            // Добавить символ в стек
  char pop();        // Вытолкнуть символ из стека
};

stack::stack(int id)
{
  stackID = id;    // Устанавливаем ID стека
  cout << "Инициализация стека " << stackID << endl;
  tos = 0;
}
```

```
stack::~stack()
{
  cout << "Деструктор стека #" << stackID << " выполняется...\n";
}

void stack::init()
{
  tos = 0;
}

void stack::push(char ch)
{
  if (tos==SIZE) {
     cout << "Стек полон!" << endl;
     return;
  }
  stck[tos] = ch;
  tos++;
}

char stack::pop() {
  if (tos==0) {
    cout << "Стек пуст" << endl;
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Создаем объекты и устанавливаем их ID
  stack s1(1), s2(2);
  int i;

  // Добавляем элементы в стеки
  s1.push('a');
  s2.push('b');
  s1.push('c');
  s2.push('d');
```

```
  s1.push('e');
  s2.push('f');

  // Выводим содержимое стеков
  for (i=0; i<3; i++) cout << s1.pop() << " ";
  cout << endl;
  for (i=0; i<3; i++) cout << s2.pop() << " ";
  cout << endl;
}
```

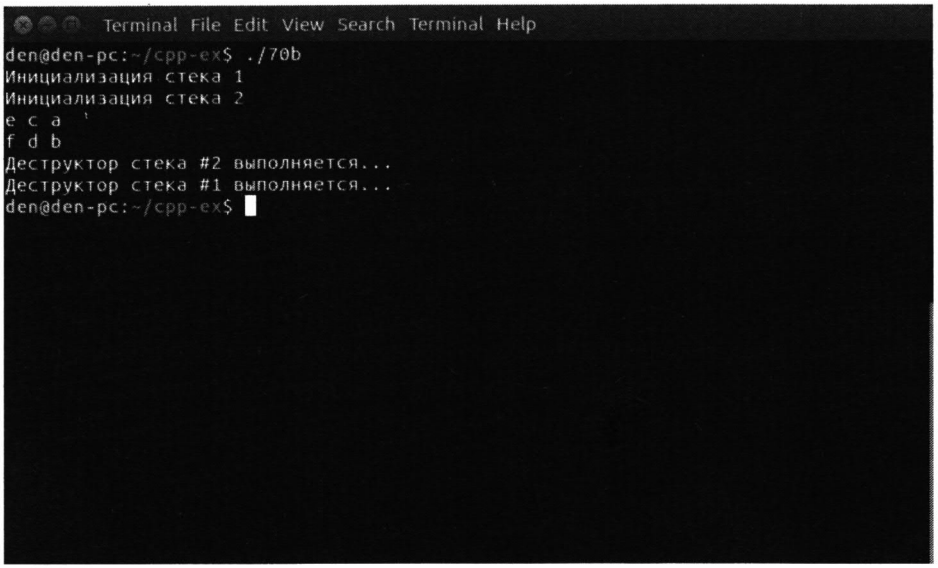


Рис. 12.3. Результат идентификации стеков

12.3. Массивы объектов

Объекты - это обычные переменные. Следовательно, мы можем создать массив таких переменных, то есть массив объектов. В листинге 12.4 создан демо-класс и массив объектов этого класса. Заодно этот пример показывает, как описываются простые функции класса. Также обратите внимание на то, как вызываются функции-члены класса для каждого элемента массива: имя массива индексируется, затем к члену применяется оператор доступа, за которым следует имя вызываемой функции-члена класса.

Листинг 12.4. Массив объектов

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

class demo {
  int a;
public:
  void set_a(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  demo ar[3];
  int i;

  for (i=0; i<3; i++) ar[i].set_a(i);
  for (i=0; i<3; i++) cout << ar[i].get_a() << " ";
  cout << endl;

  return 0;
}
```

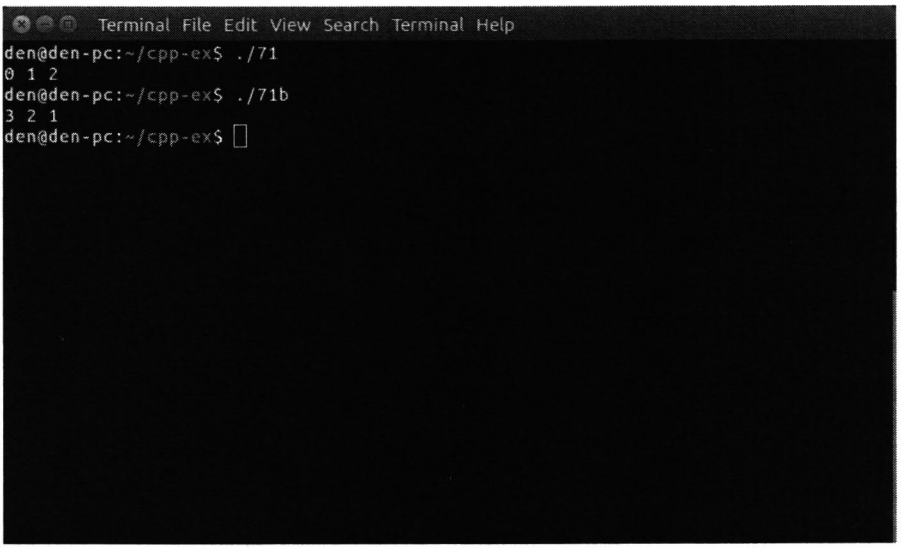


Рис. 12.4. Результат работы программ из лист. 12.3 и 12.4

Теперь усложним задачу. Представим, что наш демо класс использует конструктор с параметром. Как тогда инициализировать массив? В этом случае значения, которые будут переданы конструкторам, указываются в фигурных скобках (см. лист. 12.5).

Листинг 12.5. Массив объектов, конструктор которых принимает аргумент

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

class demo {
  int a;
public:
  demo(int n) { a = n; }
  int get_a() { return a; }
};

int main()
{
  demo ar[3] = {3, 2, 1};
  int i;

  for (i=0; i<3; i++) cout << ar[i].get_a() << " ";
  cout << endl;

  return 0;
}
```

Данная программа выведет 3 2 1 - именно эти числа мы передавали при объявлении массива.

12.4. Наследование

Наследование - это один из краеугольных принципов объектно-ориентированного программирования. Допустим, есть какой-то базовый класс, функциональность которого вам нужно расширить. Вы создаете производный класс на его базе и вам не нужно повторять в нем функционал базового класса - он будет унаследован.

Наследование описывается так:

```
class имя_производного_класса: доступ имя_базового_класса {
//
};
```

Здесь доступ - это модификатор доступа, который может быть public, private или protected. Чаще всего используются public или private. В первом случае все открытые члены базового класса останутся открытыми и в производном. Во втором (private) все открытые члены базового класса в производном станут закрытыми.

Пример наследования приведен в листинге 12.6.

Листинг 12.6. Пример наследования класса

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

class parent {
  int x;
public:
  void setx(int n) { x = n; }
  void showx() { cout << x << endl; }
};

class child: public parent {
  int y;
public:
  void sety(int n) { y = n; }
  void showy() { cout << y << endl; }
};

int main() {
  child ob;

  ob.setx(100); // получаем доступ к члену базового класса
  ob.sety(200); // получаем доступ к члену производного класса

  ob.showx(); // доступ к члену базового класса
  ob.showy(); // доступ к члену производного класса
```

```
  return 0;
}
```

Если мы укажем модификатор доступа private, то получим ошибку при обращении к членам базового класса, а именно:

```
ob.setx(100); // получаем доступ к члену базового класса
ob.showx(); // доступ к члену базового класса
```

12.5. Перегрузка операторов

В прошлой главе мы создавали программу, которая выполняла сложение пользовательской структуры данных. В C++ данная задача решается с помощью классов и механизма перезагрузки операторов. Представим, что у нас есть какой-то класс coord:

```
coord a, b, c;
```

Что произойдет, когда мы попытаемся сложить два объекта этого класса:

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

Механизм перезагрузки операторов как раз и позволяет определить, что же произойдет. В листинге 12.7 мы перегрузили операторы + и - для класса coord, содержащего координаты точки в двухмерном пространстве (x и y).

Листинг 12.7. Пример перезагрузки операторов + и -

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

class coord {
  int x, y;
public:
  coord() { x = 0; y = 0;}
  coord(int i, int j) { x = i; y = j; }
  void get_xy() { cout << "X: " << x << " Y: " << y << endl; }
  coord operator+(coord ob2);
  coord operator-(coord ob2);
};
```

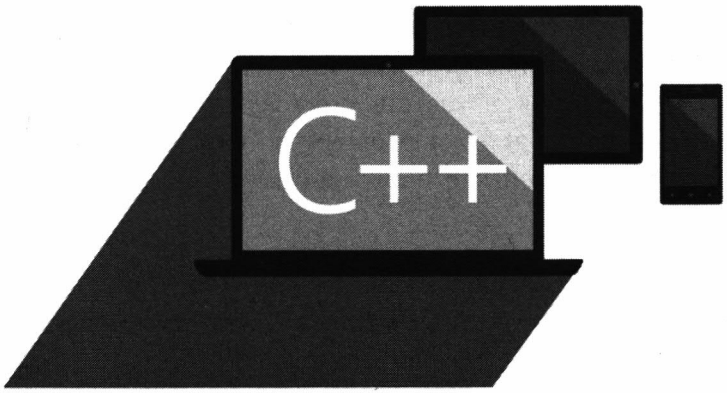
```
coord coord::operator+(coord ob2) {
  coord temp;
  temp.x = x + ob2.x;
  temp.y = y + ob2.y;
  return temp;
}

coord coord::operator-(coord ob2) {
  coord temp;
  temp.x = x - ob2.x;
  temp.y = y - ob2.y;
  return temp;
}

int main()
{
  coord a(100, 200), b(50, 70), c, d;

  c = a + b;
  d = a - b;
  c.get_xy();
  d.get_xy();

  return 0;
}
```

Наши функции operator+ и operator- возвращают объект типа coord. Внутри каждый из операторов производит вычисление координат: к текущим координатам x и y добавляются координаты x и y второго объекта (ob2) и все это помещается в объект temp, который и возвращается оператором.

Затем возвращенные объекты присваиваются соответствующим переменным типа coord, в нашем случае это переменные c и d.

Глава 13.

Сетевое программирование на C++



13.1. Клиент-серверная архитектура

В этой главе мы разработаем простое приложение клиент/сервер. Сервер будет хранить и обрабатывать данные, а клиент получать и отображать данные. Такое приложение поможет вам в трудоустройстве, поскольку очень часто встречается в качестве тестового задания. Тем не менее, наш клиент-сервер хоть и будет простым, но если сравнивать его с ранее разрабатываемыми в этой книге приложениями, то простым его назвать сложно.

Во-первых, наше приложение будет состоять из нескольких файлов и вы получите опыт разработки сложных приложений - ведь сложные приложения редко состоят из одного файла.

Во-вторых, наше приложение будет объектно-ориентированным - именно поэтому в прошлой главе мы повторили навыки ООП-разработки.

В-третьих, вы получите навыки создания Makefile - файла сборки. Такие файлы создаются для облегчения компиляции сложных приложений. В Makefile указывается все, что нужно для сборки приложения, а именно инструкции компилятора: пути для include-файлов, опции компилятора и т.д. Впоследствии для компиляции программы (например, при внесении в нее изменений) вам всего лишь придется ввести команду make для ее сборки, а не запускать g++ непосредственно, указывая с десяток опций.

В-четвертых, наш сервер будет многопокотовым, что позволит к нему подключаться сразу нескольким клиентам. А это очень важно, поскольку все реальные серверы являются многопокотовыми.

13.2. Разработка на C++ клиентской части сетевого приложения

Начнем мы с приложения-клиента. Оно будет отправлять серверу случайное число в цикле - при каждой итерации будет отправляться новое число. После отправки этого случайного числа приложение будет читать ответ сервера, выводить его на экран и засыпать на одну секунду.

Работать приложение будет так. У нас будет класс TCPClient, объект которого мы создадим в программе. Для подключения к серверу будет использоваться метод setup(), которому нужно будет передать два параметра - IP-адрес сервера и нужный порт (сервер должен прослушивать этот порт, поэтому если вы его измените на клиенте, нужно будет изменить и на сервере):

```
tcp.setup("127.0.0.1",11999);
```

Метод send() используется для отправки строки на сервер. Метод можно вызывать только после установки соединения. В случае успешной установки соединения метод setup() возвращает true. Мы не производим проверку на установку соединения для упрощения кода примера, но вы можете такую реализовать. Это несложно.

Получить ответ от сервера можно методом receive(). Если у сервера есть ответ, то возвращается непустая строка, которую мы просто выводим на экран с помощью оператора <<.

Наш класс TCPClient будет описан в заголовочном файле TCPClient.h, который мы подключаем инструкцией:

```
#include "TCPClient.h"
```

Собственно, когда мы знаем, что к чему, мы готовы рассмотреть первый листинг из этого примера - файла client.cpp.

Листинг 13.1. Файл client.cpp. Приложение-клиент

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

TCPClient tcp;     // наш основной класс

// обработчик выхода из программы
void sig_exit(int s)
{
   tcp.exit();     // вызов метода exit()
   exit(0);
}

int main(int argc, char *argv[])
{
      // Установка обработчика выхода из программы
   signal(SIGINT, sig_exit);

   tcp.setup("127.0.0.1",11999);
   while(1)
   {
            // Инициализация генератора случайных чисел
      srand(time(NULL));
            // Отправляем строку на сервер
      tcp.Send(to_string(rand()%25000));
            // Получаем ответ сервера
      string rec = tcp.receive();
      if( rec != "" )
      {
               // Выводим ответ сервера
         cout << "Server Response:" << rec << endl;
      }
      sleep(1);     // Засыпаем на 1 секунду
   }
   return 0;
}
```

В листинге 13.2 приведен заголовочный файл TCPClient.h. В нем мы подключаем другие необходимые заголовочные файлы, а также объявляем сам класс и его методы. Реальный код будет в третьем файле - TCPClient.cpp.

Листинг 13.2. Заголовочный файл TCPClient.h

```
#ifndef TCP_CLIENT_H
#define TCP_CLIENT_H

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <netdb.h>
#include <vector>

using namespace std;

class TCPClient
{
  private:
    int sock;
    std::string address;
    int port;
    struct sockaddr_in server;

  public:
    TCPClient();
    bool setup(string address, int port);
    bool Send(string data);
    string receive(int size = 4096);
    string read();
    void exit();
};
```

```
#endif
```

Файл TCPClient.cpp мы добавим к нашему проекту уже при компиляции программы - мы укажем его название в опциях компилятора. Файл TCPClient.cpp содержит реальный код, поэтому основное внимание нужно уделить именно ему. Начнем с метода setup().

Первым делом нам нужно открыть сокет. Это мы делаем так:

```
if(sock == -1)
{
   sock = socket(AF_INET , SOCK_STREAM , 0);
   if (sock == -1)
   {
           cout << "Could not create socket" << endl;
      }
      }
```

Член класса sock содержит открытый сокет. Если сокет не открыт, то его значение будет равно -1. Это и есть значение по умолчанию, заданное в конструкторе класса:

```
TCPClient::TCPClient()
{
   sock = -1;
   port = 0;
   address = "";
}
```

Для подключения к серверу сначала нужно заполнить структуру server:

```
struct sockaddr_in server
```

Мы должны указать адрес сервера, протокол и порт сервера соответственно:

```
server.sin_addr.s_addr = inet_addr( address.c_str() );
server.sin_family = AF_INET;
server.sin_port = htons( port );
```

После того, как структура server заполнена мы можем использовать функцию connect() для подключения к серверу. Этой функции нужно передать наш сокет, структуру server и размер этой структуры:

```
   if (connect(sock , (struct sockaddr *)&server ,
sizeof(server)) < 0)
   {
        perror("connect failed. Error");
        return false;
   }
   return true;
```

Если функция connect() вернула значение меньше 0, то подключиться к серверу не получилось.

В принципе все понятно. Полный код метода setup() будет приведен в листинге 74в. Далее переходим к методу Send(). Нам нужно использовать одноименную функцию send(), указав сокет, передаваемые данные и длину этих данных:

```
if( send(sock , data.c_str() , strlen( data.c_str() ) , 0) < 0)
     {
        cout << "Send failed : " << data << endl;
        return false;
     }
```

Метод read() позволяет получить ответ от сервера. Для чтения данных мы будем использовать метод recv. Читать данные будем в массив buffer. Чтение будет происходить посимвольно, а как прочитаем последний байт (когда встретим символ '\n'), мы вернем полученную строку reply:

```
   char buffer[1] = {};          // буфер
   string reply;          // результат
   while (buffer[0] != '\n') {
        if( recv(sock , buffer , sizeof(buffer) , 0) < 0)
        {
             cout << "receive failed!" << endl;
        return nullptr;
        }
     // добавляем каждый прочитанный символ к reply
     reply += buffer[0];
```

```
    }
    return reply;     // возвращаем результат
```

Кроме метода read() у нас есть еще метод receive(), который делает все то же самое, но немного иначе. Здесь у нас будет не посимвольное чтение, а чтение строки определенного размера size:

```
string TCPClient::receive(int size)
{
   char buffer[size];
   memset(&buffer[0], 0, sizeof(buffer));

   string reply;
   if( recv(sock , buffer , size, 0) < 0)
   {
        cout << "receive failed!" << endl;
      return nullptr;
   }
   buffer[size-1]='\0';
   reply = buffer;
   return reply;
}
```

Какой метод использовать, решайте сами. Для примера проще использовать метод receive(), в реальной жизни, где ответ сервера не имеет фиксированного размера - метод read().

Метод exit() закрывает сокет:

```
    close( sock );
```

Полный код TCPClient.cpp приведен в листинге 13.3.

Листинг 13.3. Файл TCPClient.cpp

```
#include "TCPClient.h"

TCPClient::TCPClient()
{
   sock = -1;
   port = 0;
```

```
    address = "";
}

bool TCPClient::setup(string address , int port)
{
    if(sock == -1)
    {
        sock = socket(AF_INET , SOCK_STREAM , 0);
        if (sock == -1)
        {
                cout << "Could not create socket" << endl;
            }
            }
    if(inet_addr(address.c_str()) == -1)
    {
          struct hostent *he;
          struct in_addr **addr_list;
          if ( (he = gethostbyname( address.c_str() ) ) == NULL)
          {
              herror("gethostbyname");
                   cout<<"Failed to resolve hostname\n";
              return false;
          }
          addr_list = (struct in_addr **) he->h_addr_list;
          for(int i = 0; addr_list[i] != NULL; i++)
          {
                   server.sin_addr = *addr_list[i];
              break;
          }
    }
    else
    {
          server.sin_addr.s_addr = inet_addr( address.c_str() );
    }
    server.sin_family = AF_INET;
    server.sin_port = htons( port );
    if (connect(sock , (struct sockaddr *)&server ,
sizeof(server)) < 0)
    {
          perror("connect failed. Error");
          return false;
```

```
    }
    return true;
}

bool TCPClient::Send(string data)
{
    if(sock != -1)
    {
        if( send(sock , data.c_str() , strlen( data.c_str() ) , 0) <
0)
        {
            cout << "Send failed : " << data << endl;
            return false;
        }
    }
    else
        return false;
    return true;
}

string TCPClient::receive(int size)
{
    char buffer[size];
    memset(&buffer[0], 0, sizeof(buffer));

    string reply;
    if( recv(sock , buffer , size, 0) < 0)
    {
            cout << "receive failed!" << endl;
        return nullptr;
    }
    buffer[size-1]='\0';
    reply = buffer;
    return reply;
}

string TCPClient::read()
{
    char buffer[1] = {};
    string reply;
    while (buffer[0] != '\n') {
```

```
        if( recv(sock , buffer , sizeof(buffer) , 0) < 0)
        {
             cout << "receive failed!" << endl;
        return nullptr;
        }
      reply += buffer[0];
   }
   return reply;
}

void TCPClient::exit()
{
    close( sock );
}
```

Теперь посмотрим на рис. 13.1. На нем изображен клиент, получающий ответ от сервера. Окно терминала слева - это вывод сервера, а окно терминала справа (на переднем плане) - это окно нашего клиента. Он выводит полученные от сервера сообщения.

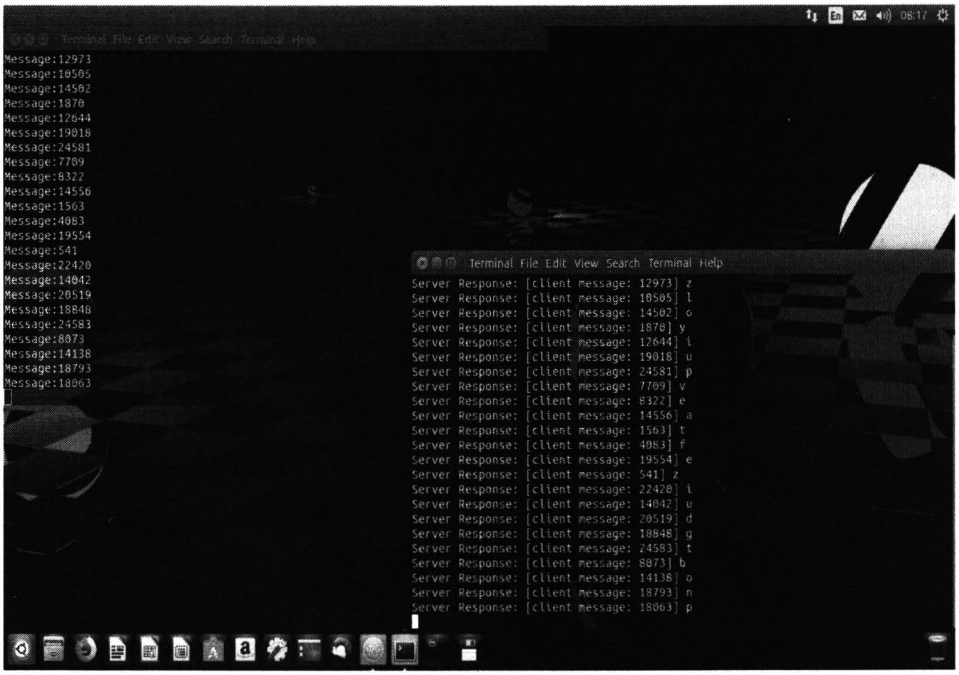


Рис. 13.1. Клиент в действии

Небольшой итог. Мы только что разработали приложение-клиент, состоящее из трех файлов: client.cpp, TCPClient.cpp, TCPClient.h.

13.3. Разработка на C++ серверной части сетевого приложения

Приложение-сервер гораздо сложнее. Главным образом из-за того, что мы используем многопотоковую обработку. Многопоточность - это тема для отдельной книги, но попробуем разобраться, что и к чему.

Функция pthread_create() создает новый поток. Ей нужно передать четыре параметра:

1. Переменную потока типа pthread_t
2. Аргументы потока, у нас будет NULL, то есть передавать аргументы мы не будем.
3. Функцию, которая будет выполняться в потоке
4. Аргументы для этой функции.

Прототип pthread_create() выглядит так:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

Подробное описание этой функции приведено по адресу http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_create.html. А мы вернемся к нашему коду:

```
pthread_t msg;    // поток
tcp.setup(11999);// настройка TCP, указываем порт сервера
// Создаем поток
if( pthread_create(&msg, NULL, loop, (void *)0) == 0)
{
   tcp.receive();
}
```

Если поток создан успешно, мы производим чтение методом receive(). Это серверный метод и он отличается от одноименного клиентского метода. Его мы рассмотрим позже. Функция loop() выглядит так:

```
void * loop(void * m)
{
        pthread_detach(pthread_self());
   while(1)
   {
      srand(time(NULL));
      char ch = 'a' + rand() % 26;
      string s(1,ch);
      string str = tcp.getMessage();
      if( str != "" )
      {
         cout << "Message:" << str << endl;
         tcp.Send(" [client message: "+str+"] "+s);
         tcp.clean();
      }
      usleep(1000);
   }
   tcp.detach();
}
```

Что мы здесь делаем? Во-первых, получаем сообщение клиента методом getMessage(). Результат записываем в переменную str. Во-вторых, выводим сообщение клиента (напомню, это случайное число) на экран. В-третьих, отправляем клиенту свое сообщение методом Send(). Метод clean() используется просто для очистки члена Message.

Полный код server.cpp приведен в листинге 13.4.

Листинг 13.4. Приложение-сервер

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

TCPServer tcp;

void * loop(void * m)
```

```
{
        pthread_detach(pthread_self());
   while(1)
   {
      srand(time(NULL));
      char ch = 'a' + rand() % 26;
      string s(1,ch);
      string str = tcp.getMessage();
      if( str != "" )
      {
         cout << "Message:" << str << endl;
         tcp.Send(" [client message: "+str+"] "+s);
         tcp.clean();
      }
      usleep(1000);
   }
   tcp.detach();
}

int main()
{
   pthread_t msg;
   tcp.setup(11999);
   if( pthread_create(&msg, NULL, loop, (void *)0) == 0)
   {
      tcp.receive();
   }
   return 0;
}
```

Данный файл использует заголовочный файл TCPServer.h. Его код приведен в листинге 13.5.

Листинг 13.5. Заголовочный файл TCPServer.h

```
#ifndef TCP_SERVER_H
#define TCP_SERVER_H

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

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#include <pthread.h>

using namespace std;

#define MAXPACKETSIZE 4096

class TCPServer
{
   public:
   int sockfd, newsockfd, n, pid;
   struct sockaddr_in serverAddress;
   struct sockaddr_in clientAddress;
   pthread_t serverThread;
   char msg[ MAXPACKETSIZE ];
   static string Message;

   void setup(int port);
   string receive();
   string getMessage();
   void Send(string msg);
   void detach();
   void clean();

   private:
   static void * Task(void * argv);
};

#endif
```

Как и в случае с клиентом, мы подключаем заголовочные файлы и объявляем класс TCPServer в этом файле.

А теперь реальный код. Метод setup(), выполняющий установку сервера выглядит так:

```
   sockfd=socket(AF_INET,SOCK_STREAM,0);
   memset(&serverAddress,0,sizeof(serverAddress));
   serverAddress.sin_family=AF_INET;
   serverAddress.sin_addr.s_addr=htonl(INADDR_ANY);
   serverAddress.sin_port=htons(port);
   bind(sockfd,(struct sockaddr *)&serverAddress,
sizeof(serverAddress));
   listen(sockfd,5);
```

Ему нужно передать только один параметр - port. Здесь мы также заполняем структуру

```
   struct sockaddr_in serverAddress;
```

А затем вызываем функции bind() и listen(). Первая связывает наш сокет со структурой serverAddress. Мы передаем ей три параметра - сокет, структуру serverAddress и размер этой структуры. Функция listen() запускает прослушивание сокета. Первый параметр - наш сокет, второй параметр - максимальная длина очереди. В данном случае 5 будет вполне достаточно, но вы можете увеличить это значение при необходимости.

Метод receive(), получающий информацию от клиента, выглядит так:

```
   string str;
   while(1)
   {
      socklen_t sosize  = sizeof(clientAddress);
      newsockfd = accept(sockfd,(struct
sockaddr*)&clientAddress, &sosize);
      str = inet_ntoa(clientAddress.sin_addr);
      pthread_create(&serverThread,NULL,&Task,(void *)
newsockfd);
   }
   return str;
```

Здесь появляется новая структура, содержащая адрес клиента:

```
struct sockaddr_in clientAddress;
```

Мы создаем новый сокет - под конкретного клиента и его дескриптор помещаем в переменную newsockfd. Функция accept() принимает соединение от клиента. Мы указываем сокет сервера (sockfd), структуру clientAddress и ее размер. Затем мы создаем отдельный поток, обрабатывающий конкретного клиента - для этого вызываем функцию pthread_create, передав ей обработчик Task и в качестве параметра этого обработчика - сокет клиента (newsockfd).

Метод Task заполняет свойство Message, которое потом возвращается методом getMessage():

```
    char msg[MAXPACKETSIZE];
pthread_detach(pthread_self());
while(1)
{
   n=recv(newsockfd,msg,MAXPACKETSIZE,0);
   if(n==0)
   {
      close(newsockfd);
      break;
   }
   msg[n]=0;
   Message = string(msg);
```

А вот простой метод getMessage():

```
string TCPServer::getMessage()
{
   return Message;
}
```

Можно было бы обойтись и без него, но с ним код красивее. Полный код нашего TCPServer.cpp приведен в листинге 13.6.

Листинг 13.6. Файл TCPServer.cpp

```
#include "TCPServer.h"

string TCPServer::Message;
```

```
// Основной метод сервера, обработка клиента
void* TCPServer::Task(void *arg)
{
   int n;
   int newsockfd = (long)arg;
   char msg[MAXPACKETSIZE];
   pthread_detach(pthread_self());
   while(1)
   {
      n=recv(newsockfd,msg,MAXPACKETSIZE,0);
      if(n==0)
      {
         close(newsockfd);
         break;
      }
      msg[n]=0;
      //send(newsockfd,msg,n,0);
      Message = string(msg);
         }
   return 0;
}

// Установка сервера
void TCPServer::setup(int port)
{
   sockfd=socket(AF_INET,SOCK_STREAM,0);
   memset(&serverAddress,0,sizeof(serverAddress));
   serverAddress.sin_family=AF_INET;
   serverAddress.sin_addr.s_addr=htonl(INADDR_ANY);
   serverAddress.sin_port=htons(port);
   bind(sockfd,(struct sockaddr *)&serverAddress,
sizeof(serverAddress));
   listen(sockfd,5);
}

// Получение информации от клиента
string TCPServer::receive()
{
   string str;
   while(1)
   {
```

```
      socklen_t sosize  = sizeof(clientAddress);
      newsockfd = accept(sockfd,(struct sockaddr*)&clientAddress,&
sosize);
      str = inet_ntoa(clientAddress.sin_addr);
      pthread_create(&serverThread,NULL,&Task,(void *)newsockfd);
   }
   return str;
}

// Возвращаем сообщение киента
string TCPServer::getMessage()
{
   return Message;
}

// Отправка сообщения клиенту
void TCPServer::Send(string msg)
{
   send(newsockfd,msg.c_str(),msg.length(),0);
}

// Очистка сообщения
void TCPServer::clean()
{
   Message = "";
   memset(msg, 0, MAXPACKETSIZE);
}

// Закрываем сокеты клиента и сервера
void TCPServer::detach()
{
   close(sockfd);
   close(newsockfd);
}
```

На рис. 13.2 показан вывод сервера и двух подключенных клиентов. Наш сервер хоть и простой, но многопотоковый.

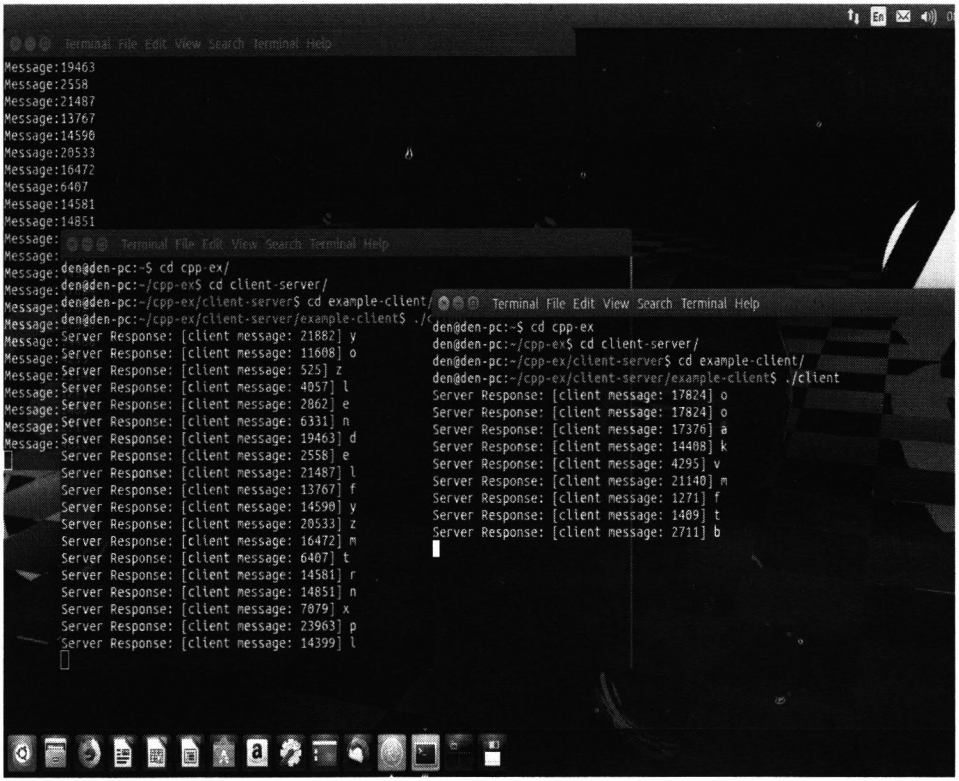


Рис. 13.2. Сервер и два клиента

13.4. Сборка клиент-серверного приложения

Используем команду make для сборки сложного проекта. Собираем все воедино. Для сборки нашего клиента нужно ввести команду (используем компилятор g++):

```
g++ -Wall -o client client.cpp -I../src/ ../src/TCPServer.cpp ../
src/TCPClient.cpp -std=c++11 -lpthread
```

Для сборки сервера используется несколько иная команда

```
g++ -Wall -o server server.cpp -I../src/ ../src/TCPServer.cpp ../
src/TCPClient.cpp -std=c++11 -lpthread
```

Здесь мы указываем имена выходных файлов, имена основных файлов, дополнительные файлы, необходимые для компиляции (-I), задаем стандарт кода (c++11), подключаем многопотоковую библиотеку.

Согласитесь, сложно запомнить все эти параметры, еще сложнее вводить их при каждой сборке программы, например, когда вы хотите что-то усовершенствовать. Можно, конечно, было создать сценарий командной оболочки, но программисты так не поступают. Они создают Makefile.

Сначала определимся со структурой проекта:

```
client-server
   client
   server
   src
```

Все файлы проекта находятся в папке client-server. Файлы client.cpp и server.cpp будут находиться в папках client и server соответственно. Все вспомогательные файлы TCP* будут находиться в папке src.

В каждую из папок client и server нужно добавить по файлу с именем Makefile (именно в таком регистре). Содержимое будет следующим. Для файла clint/Makefile:

```
all:
   g++ -Wall -o client client.cpp -I../src/ ../src/TCPServer.cpp
../src/TCPClient.cpp -std=c++11 -lpthread
```

Файл server/Makefile:

```
all:
   g++ -Wall -o server server.cpp -I../src/ ../src/TCPServer.cpp
../src/TCPClient.cpp -std=c++11 -lpthread
```

В каждом файле есть всего одна цель - all, для выполнения которой нужно выполнить соответствующую команду g++.

Теперь как использовать make. Перейдите в папку client и введите make. Затем проделайте то же самое с сервером:

```
cd client-server
```

```
cd client
make
cd ..
cd server
make
```

Запустить сервер можно так:

```
./server
```

Запустить клиент можно так:

```
cd ../client
./client
```

Сначала нужно запускать сервер, а потом уже клиент (иначе клиент не сможет подключиться к серверу).

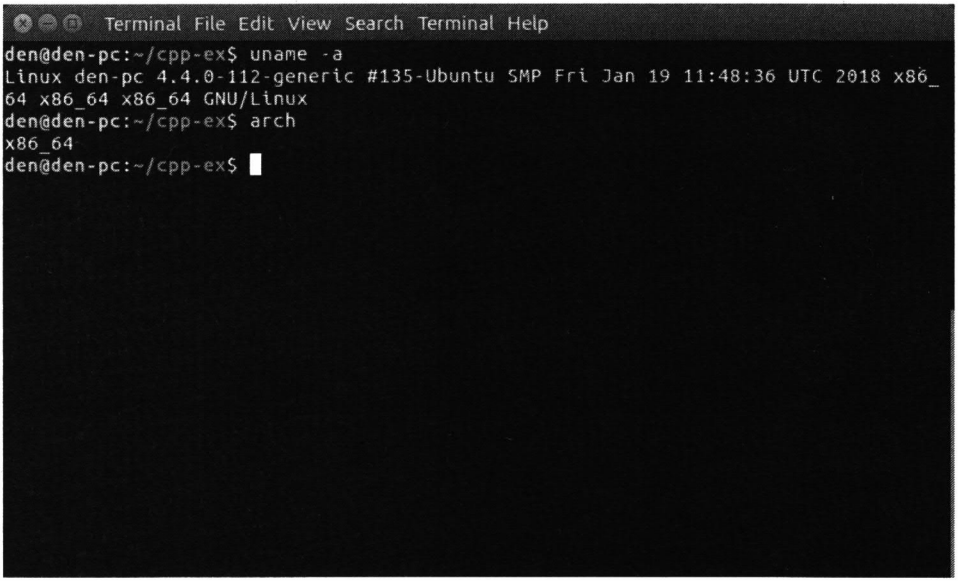
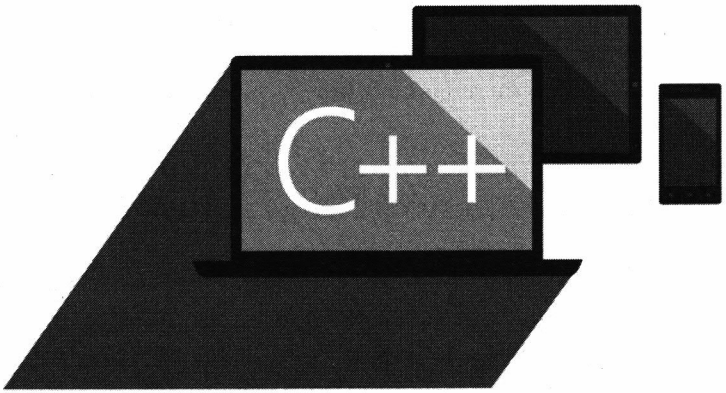


Рис. 13.3. Параметры системы, на которой осуществлялась сборка проекта

Глава 14.

Программирование алгоритмов на C++



Какая же серьезная программа на C++ обходится без поиска и сортировки данных? Огромное внимание алгоритмам поиска и сортировки уделяется в настоящем шедевре - книге Дональда Кнута "Искусство программирования". Вот только "Искусство программирования" - отличный выбор, когда есть время на его изучение. А вот когда времени нет, а программа нужна прямо здесь и сейчас, то приходится обращаться к другим источникам, например, к этой книге. В этой главе мы рассмотрим множество примеров, которые, помогут вам при написании лабораторной, курсовой работы и собственного реального приложения.

14.1. Алгоритмы поиска. Бинарный поиск

Бинарный (он же двоичный) поиск — классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Данный метод также известен как метод деления пополам.

Если у нас есть массив, содержащий упорядоченную последовательность данных, то очень эффективен двоичный поиск. Да, вы все правильно поняли, бинарный поиск работает только на уже отсортированных массивах, поэтому перед применением бинарного поиска к произвольному массиву (прочитанному из файла или введенному пользователем), его нужно отсортировать.

Переменные left и right содержат, соответственно, левую и правую границы отрезка массива, где находится нужный нам элемент. Мы начинаем всегда

с исследования среднего элемента отрезка (middle). Если искомое значение меньше среднего элемента, мы переходим к поиску в верхней половине отрезка, где все элементы меньше только что проверенного. Другими словами, значением right становится (middle – 1) и на следующей итерации мы работаем с половиной массива. Таким образом, в результате каждой проверки мы вдвое сужаем область поиска. Так, в нашем примере, после первой итерации область поиска – всего лишь 5 элементов.

Двоичный поиск - очень мощный и эффективный метод. Если представить, что длина массива равна 1023, после первого сравнения область сужается до 511 элементов, а после второй - до 255. Легко посчитать, что для поиска в массиве из 1023 элементов достаточно 10 сравнений.

Листинг 14.1. Двоичный поиск в целом (int) массиве

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

#define TRUE  0
#define FALSE 1

int main(void) {
 int array[10] = {0, 1, 2, 3, 4, 6, 7, 8, 9, 10};
 int left = 0;
 int right = 10;
 int middle = 0;
 int number = 0;
 int bsearch = FALSE;
 int i = 0;

 cout << "Массив: ";
 for(i = 0; i < 10; i++)
  cout << array[i] << " ";

 cout << "\nИщем число: ";
 cin >> number;

 while(bsearch == FALSE && left <= right) {
  middle = (left + right) / 2;
```

```
  if(number == array[middle]) {
   bsearch = TRUE;
   cout << "** Найдено! **\n";
  } else {
   if(number < array[middle]) right = middle - 1;
   if(number > array[middle]) left = middle + 1;
  }
 }

 if(bsearch == FALSE)
  cout << "-- Не найдено --\n";

 return 0;
}
```

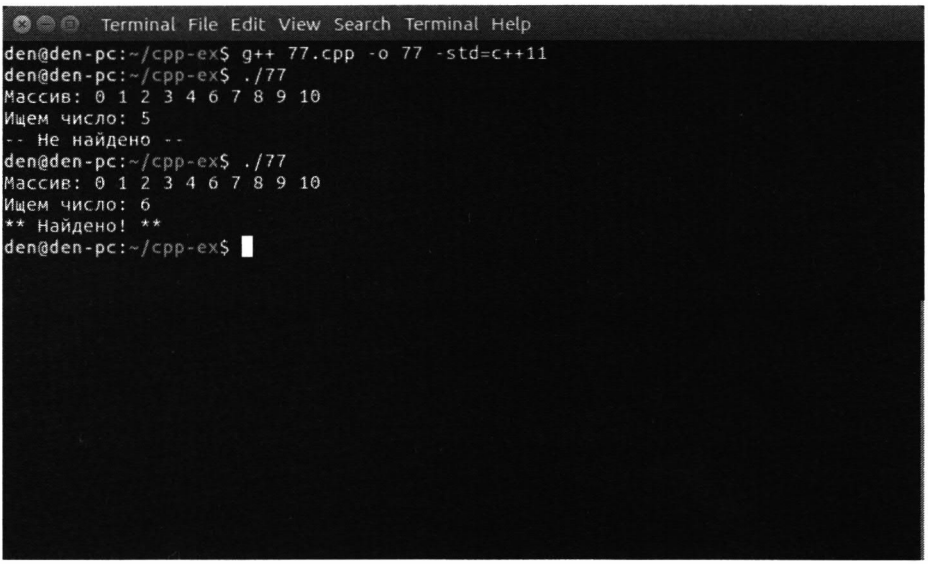


Рис. 14.1. Результат двоичного поиска

Посмотрите на рис. 14.1 и код программы. В нашем массиве элемента 5 нет. Мы вводим сначала 5, а потом элемент, который есть в массиве, чтобы проверить правильность работы программы.

Дополнительную информацию по этому методу поиска и дополнительный пример кода вы можете получить в Википедии: **https://goo.gl/SKVJYx**

Прошлый пример показывал, как выполнить поиск по упорядоченному массиву целых чисел. Но на практике чаще возникают задачи поиска определенной строки, нежели определенного числа. Именно поэтому сейчас будет рассмотрен пример двоичного поиска по массиву указателей строк.

Принцип тот же. Исходный массив должен быть отсортирован. В функцию binsearch передается массив строк, размер массива и искомое значение. Функция возвращает 0, если значение не найдено или же позицию найденного значения. Учитывая, что массив отсортирован, средняя позиция определяется как сумма начальной и последней (begin + end), разделенная на 2. Далее нужно сравнить функцией strcmp() искомое слово со словом в получившейся позиции. Функция strcmp() возвращает значение

- < 0, если первый ее аргумент лексикографически меньше, чем второй;
- > 0, если первый аргумент лексикографически больше, чем второй
- 0, если аргументы равны.

Так вот, функция strcmp() не только сравнивает строки, но и еще и подсказывает нам в каком направлении двигаться - в соответствии с этим мы или увеличиваем позицию или уменьшаем ее. Если функция вернула 0, то мы можем вернуть позицию (переменная position), в которой это произошло.

Прототип функции strcmp() выглядит так:

```
int strcmp(const char *str1, const char *str2)
```

Код примера, реализующего бинарный поиск по массиву строк, приведен в листинге 14.2, а результат его работы, как обычно, показан на рис. 14.2.

Листинг 14.2. Бинарный поиск по массиву строк

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

```
static int binsearch(char *str[], int max, char *value);

int main(void) {
 /* этот массив будем сортировать... */
 char *strings[] = { "audi", "bentley", "bmw", "cadillac", "ford"
};
 int i, asize, result;

 i = asize = result = 0;

 asize = sizeof(strings) / sizeof(strings[0]);

 for(i = 0; i < asize; i++)
  //printf("%d: %s\n", i, strings[i]);
  cout << i << " " << strings[i] << endl;

 cout << endl;

 if((result = binsearch(strings, asize, "bmw")) != 0)
  cout << "`bmw' найдено на позиции: " << result << endl;
 else
  cout << "`bmw' не найдено..\n";

 if((result = binsearch(strings, asize, "mercedes")) != 0)
  cout << "`mercedes' найдено на позиции: " << result << endl;
 else
  cout << "`mercedes' не найдено..\n";

 return 0;
}

static int binsearch(char *str[], int max, char *value) {
 int position;
 int begin = 0;
 int end = max - 1;
 int cond = 0;

 while(begin <= end) {
  position = (begin + end) / 2;
  if((cond = strcmp(str[position], value)) == 0)
   return position;
```

```
  else if(cond < 0)
   begin = position + 1;
  else
   end = position - 1;
 }

 return 0;
}
```

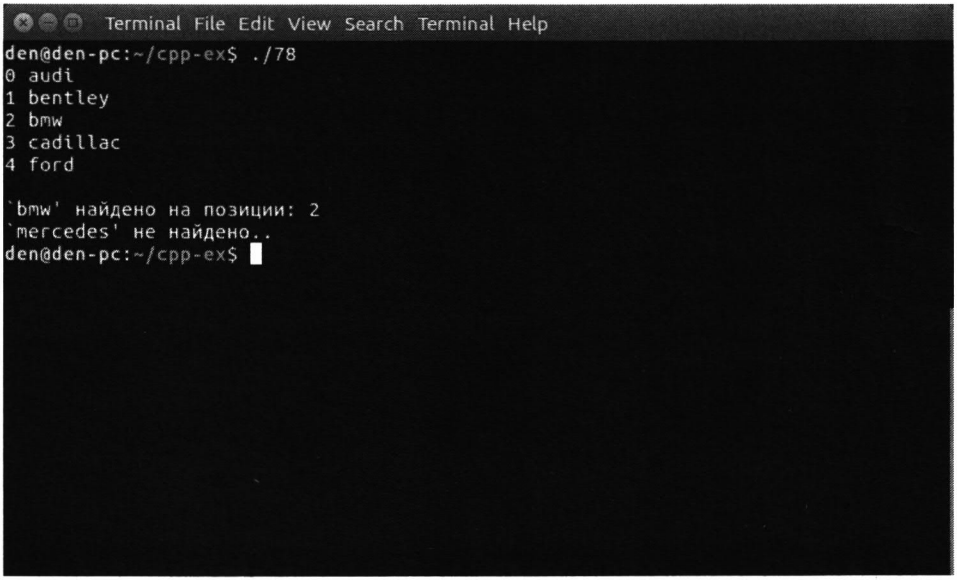


Рис. 14.2. Результат двоичного поиска строки

14.2. Алгоритмы сортировки

14.2.1. Сортировка методом пузырька

Еще один популярный в программировании метод сортировки - это сортировка пузырьком (buble sort в англ. литературе).

Алгоритм пузырьковой сортировки считается самым простым, но довольно неэффективным. Его можно использовать разве что для сортировки небольших массивов. Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяются

более эффективные алгоритмы сортировки. Но поскольку мы как раз учимся программировать, данный алгоритм - настоящая находка.

Суть алгоритма заключается в следующем. Программа несколько раз проходится по сортируемому массиву. При каждой итерации элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Получается, что элементы как бы выталкиваются вверх, как пузырьки в воде, отсюда и название алгоритма.

Проходы (итерации) по массиву повторяются N − 1 {\displaystyle N-1} N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован.

При каждой итерации очередной наибольший элемент массива ставится на свое место в конце массива - рядом с предыдущим наибольшим элементом, а наименьший элемент перемещается на одну позицию к началу массива - "всплывает".

Думаю, принцип понятен. Осталось все это закодировать. В нашей программе мы создадим функцию bubble_sort(), которой нужно передать массив элементов и его размер. Функция использует два цикла for - внутренний и внешний. Внешний проходится от 0 до size, а переменная size содержит количество элементов в массиве. Во внутреннем цикле функция проходится от 0 до size - i. Если a[j] > a[j+1], то элементы a[j] и a[j+1] меняются местами. Переменная hold используется для хранения временного значения при свопе элементов.

Листинг 14.3. Пузырьковая сортировка

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

void bubble_sort(int a[], int size);

int main(void) {
 int arr[10] = {10, 2, 4, 1, 6, 5, 8, 7, 3, 9};
 int i = 0;

 cout << "До сортировки:\n";
 for(i = 0; i < 10; i++) cout << arr[i] << " ";
```

```
 cout << endl;

 bubble_sort(arr, 10);

 cout << "После:\n";
 for(i = 0; i < 10; i++) cout << arr[i] << " ";
 cout << endl;

 return 0;
}

void bubble_sort(int a[], int size) {
 int switched = 1;
 int hold = 0;
 int i = 0;
 int j = 0;

 size -= 1;

 for(i = 0; i < size && switched; i++) {
  switched = 0;
```

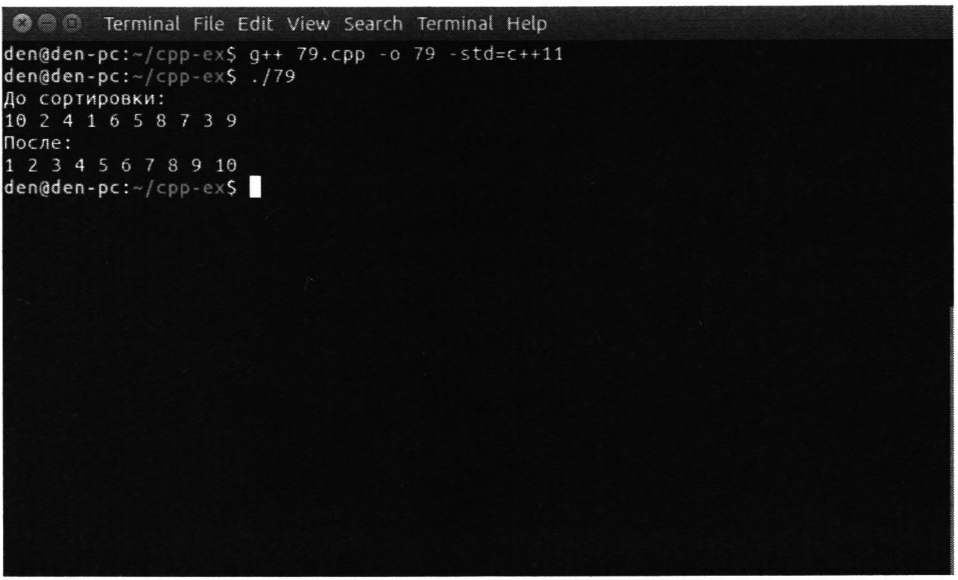


Рис. 14.3. Пузырьковая сортировка массива

```
  for(j = 0; j < size - i; j++)
   if(a[j] > a[j+1]) {
    switched = 1;
    hold = a[j];
    a[j] = a[j + 1];
    a[j + 1] = hold;
   }
 }
}
```

Еще раз отмечу, что данный алгоритм очень неэффективный: общее число сравнений равно (N-1)N, то есть если массив состоит из 10 элементов, как у нас, то программа выполнила 90 сравнений, чтобы отсортировать массив. Это настоящее расточительство ресурсов: представьте, что будет, если элементов будет не 10, а один миллион?! Тем не менее, этот алгоритм часто используется при обучении программированию. Если вы так и не разобрались, как он работает, на страничке в Википедии можно увидеть анимацию, демонстрирующую алгоритм в динамике: **https://goo.gl/KGE6yn**

14.2.2. Быстрая сортировка или сортировка Хоара

Быстрая сортировка или сортировка Хоара (по имени разработчика алгоритма) - широко известный алгоритм сортировки, разработанный английским программистом Чарльзом Хоаром в 1960 году. Не удивляйтесь - большинство алгоритмов сортировки были разработаны очень давно, примерно в 60-ых годах 20-го век, но они не потеряли свою актуальность до сих пор - пока никто ничего лучше не придумал.

Часто быструю сортировку называют qsort - по имени в стандартной библиотеке языка Си. Да, есть функция qsort(), можно использовать ее, но нам это не интересно. Гораздо интереснее написать собственную реализацию.

Быстрая сортировка - это улучшенный вариант пузырьковой сортировки, но эффективность этого алгоритма значительно выше. Принципиальное отличие заключается в том, что первым делом производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы

делятся на две независимые группы. Интересно, что незначительное улучшение самого неэффективного алгоритма породило один из самых эффективных алгоритмов сортировки. Он эффективен до такой степени, что его включили в стандартную библиотеку функций С.

Алгоритм заключается в следующем. Мы выбираем некоторый элемент - опорный элемент. Обычно это медиана - то есть элемент в середине массива.

Далее выполняется операция разделения: реорганизуем массив таким образом, чтобы все элементы со значением меньшим или равным опорному элементу, оказались слева от него, а все элементы, превышающие по значению опорный — справа от него.

Рекурсивно нужно упорядочить подмассивы, лежащие слева и справа от опорного элементы. Условие выхода из рекурсии - массив, состоящий из одного элемента (или пустой массив). Учитывая, что при каждой итерации длина обрабатываемого отрезка массива уменьшается как минимум на единицу, условие выхода из рекурсии обязательно будет достигнуто и обработка массива гарантированно будет прекращена.

Программная реализация приведена в листинге 14.4.

Листинг 14.4. Быстрая сортировка массива

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

#define MAXARRAY 10

void quicksort(int arr[], int low, int high);

int main(void) {
 int array[MAXARRAY] = {0};
 int i = 0;

 /* загружаем в массив случайные числа */
 for(i = 0; i < MAXARRAY; i++)
  array[i] = rand() % 100;
```

```
 /* выводим массив */
 cout << "До сортировки: ";
 for(i = 0; i < MAXARRAY; i++) {
  cout << array[i] << " ";
 }
 cout << endl;

 quicksort(array, 0, (MAXARRAY - 1));

 /* выводим результат */
 cout << "После: ";
 for(i = 0; i < MAXARRAY; i++) {
  cout << array[i] << " ";
 }
 cout << endl;

 return 0;
}

/* сортируем все между `low' <-> `high' */
void quicksort(int arr[], int low, int high) {
 int i = low;
 int j = high;
 int y = 0;
 /* опорный элемент */
 int z = arr[(low + high) / 2];

 /* разделение */
 do {
  /* находим элемент левее */
  while(arr[i] < z) i++;

  /* находим элемент правее */
  while(arr[j] > z) j--;

  if(i <= j) {
   /* меняем местами 2 элемета */
   y = arr[i];
   arr[i] = arr[j];
   arr[j] = y;
   i++;
```

```
    j--;
   }
  } while(i <= j);

  /* рекурсия */
  if(low < j)
   quicksort(arr, low, j);

  if(i < high)
   quicksort(arr, i, high);
}
```

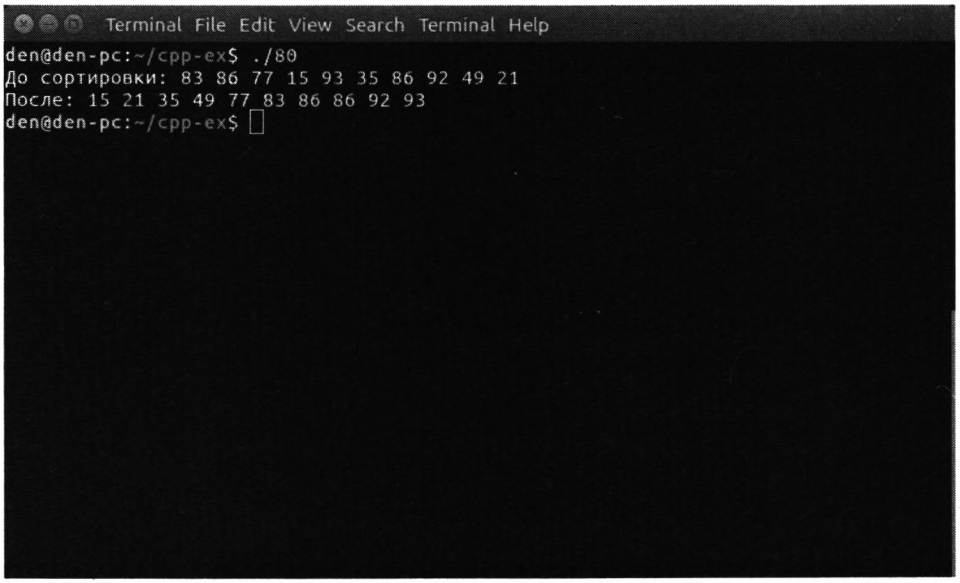


Рис. 14.4. Сортировка массива методом быстрой сортировки

14.2.3. Сортировка выбором

Сортировка выбором (англ. selection sort) - еще один алгоритм сортировки. Алгоритм сам по себе довольно простой:

1. Находим номер минимального значения в текущем списке.

2. Производим обмен найденного значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции).

3. Сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

Не смотря на простоту описания алгоритма, сама программа не очень простая и занимает целых 145 (!) строк, см. лист. 14.5. Как обычно, результат выполнения программы приводится на рис. 14.5.

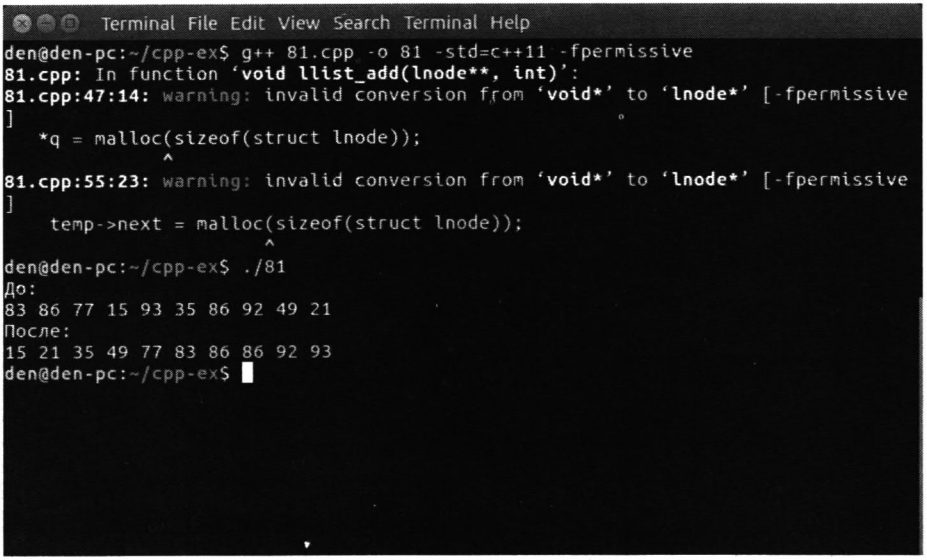


Рис. 14.5. Сортировка выбором

Листинг 14.5. Сортировка выбором

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

#define MAX 10

struct lnode {
 int data;
 struct lnode *next;
} *head, *visit;

/* добавляем новый узел в связный список */
void llist_add(struct lnode **q, int num);
/* выборочная сортировка списка */
```

```
void llist_selection_sort(void);
/* выводим связный список */
void llist_print(void);

int main(void) {
 /* связный список */
 struct lnode *newnode = NULL;
 int i = 0; /* общий счетчик */

 /* добавляем в список случайные данные */
 for(i = 0; i < MAX; i++) {
  llist_add(&newnode, (rand() % 100));
 }

 head = newnode;
 cout << "До сортировки:\n";
 llist_print();
 cout << "После:\n";
 llist_selection_sort();
 llist_print();

 return 0;
}

/* добавляем узел в список связного списка */
void llist_add(struct lnode **q, int num) {
 struct lnode *temp;

 temp = *q;

 /* если список пуст, создаем первый элемент */
 if(*q == NULL) {
  *q = malloc(sizeof(struct lnode));
   temp = *q;
 } else {
  /* переходим к последнему узлу */
  while(temp->next != NULL)
   temp = temp->next;

   /* добавляем узел в конец списка */
   temp->next = malloc(sizeof(struct lnode));
```

```
    temp = temp->next;
  }

  /* назначаем данные последнему узлу */
  temp->data = num;
  temp->next = NULL;
}

/* выводим связный список */
void llist_print(void) {
  visit = head;

  /* проходимся по списку и выводим его */
  while(visit != NULL) {
   cout << visit->data << " ";
   visit = visit->next;
  }
  printf("\n");
}

/* функция сортировки выбором */
void llist_selection_sort(void) {
  struct lnode *a = NULL;
  struct lnode *b = NULL;
  struct lnode *c = NULL;
  struct lnode *d = NULL;
  struct lnode *tmp = NULL;

  a = c = head;
  while(a->next != NULL) {
   d = b = a->next;
   while(b != NULL) {
    if(a->data > b->data) {
     /* соседний связанный узел списка */
     if(a->next == b) {
     /* если a = голова */
      if(a == head) {
       a->next = b->next;
       b->next = a;
       tmp = a;
       a = b;
```

```
      b = tmp;
      head = a;
      c = a;
      d = b;
      b = b->next;
     } else {
      a->next = b->next;
      b->next = a;
      c->next = b;
      tmp = a;
      a = b;
      b = tmp;
      d = b;
      b = b->next;
     }
    } else {
     if(a == head) {
      tmp = b->next;
      b->next = a->next;
      a->next = tmp;
      d->next = a;
      tmp = a;
      a = b;
      b = tmp;
      d = b;
      b = b->next;
      head = a;
    } else {
     tmp = b->next;
     b->next = a->next;
     a->next = tmp;
     c->next = b;
     d->next = a;
     tmp = a;
     a = b;
     b = tmp;
     d = b;
     b = b->next;
    }
   }
  } else {
```

```
    d = b;
    b = b->next;
   }
  }
   c = a;
   a = a->next;
  }
}
```

14.2.4. Сортировка вставками

Сортировка вставками (Insertion Sort) — это простой алгоритм сортировки. Суть его заключается в том что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Нужно отметить, что массив из 1-го элемента считается отсортированным.

Данный пример демонстрирует не только алгоритм сортировки вставками, но и работу со связным списком. Связный список — это базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки на следующий и/или предыдущий узел списка. Понятно, что первый узел списка содержит ссылку только на следующий элемент, а последний - только на предыдущий.

Для реализации связного списка мы используем структуру node, состоящую из двух членов: number - это число, которое несет в себе узел списка, и node - указатель на следующий узел. В нашем случае можно обойтись без указателя на предыдущий узел - для алгоритма сортировки вставками он не нужен.

Первый узел списка называется head (голова списка). У последнего узла списка член node равен NULL. Сортировка вставками осуществляется функцией insert_node(), которая вставляет новый элемент в нужное место списка. Элементы берутся из массива test. Затем программа выводит массив тест и получившийся список, который уже является отсортированным.

Листинг 14.6. Сортировка вставками

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

```
using namespace std;

struct node {
 int number;
 struct node *next;
};

struct node *head = NULL;

/* функция вставляет узел в правильное место связного списка */
void insert_node(int value);

int main(void) {
 struct node *current = NULL;
 struct node *next = NULL;
 int test[] = {8, 3, 2, 6, 1, 5, 4, 7, 9, 0};
 int i = 0;

 /* вставляем некоторые элементы в связный список */
 for(i = 0; i < 10; i++)
  insert_node(test[i]);

 /* выводим список */
 cout << "До  После\n";
 i = 0;
 while(head->next != NULL) {
  cout << test[i++] << "\t" << head->number << endl;
  head = head->next;
 }

 /* очищаем список */
 for(current = head; current != NULL; current = next)
  next = current->next, free(current);

 return 0;
}

void insert_node(int value) {
 struct node *temp = NULL;
 struct node *one = NULL;
 struct node *two = NULL;
```

```
 // если список пуст, нужно выделить память под голову списка
 if(head == NULL) {
  head = (struct node *)malloc(sizeof(struct node *));
  head->next = NULL;
 }

 // первый элемент - голова, второй - следующий элемент
 one = head;
 two = head->next;

 // временный узел
 temp = (struct node *)malloc(sizeof(struct node *));
 temp->number = value;

 // меняем one и two местами в случае необходимости
 while(two != NULL && temp->number < two->number) {
  one = one->next;
  two = two->next;
 }

 one->next = temp;
 temp->next = two;
}
```

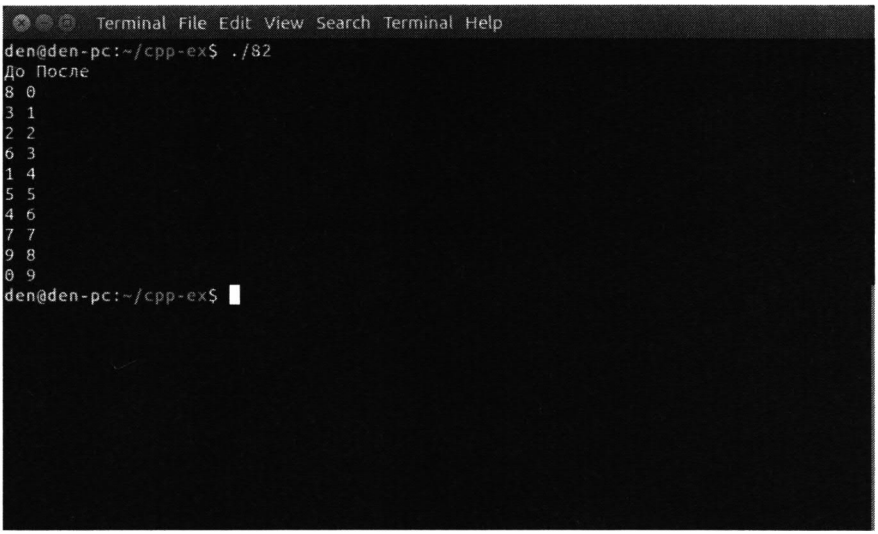


Рис. 14.6. Вывод программы

Давайте усложним нашу предыдущую задачу и выполним пузырьковую сортировку связного списка. Алгоритм будет таким же, но работать мы будем не с массивом, а со связным списком. Подобная задача - хорошая практика по работе с указателями, а они играют в C++ очень важную роль - ни одна серьезная программа на этом языке программирования не обходится без указателей. В то же время большинство ошибок, допускаемых начинающими программистами, связаны как раз с работой с указателями, поэтому чем больше практики по работе с указателями у вас будет, тем лучше.

Как уже было отмечено, сам алгоритм сортировки останется тем же (только мы его слегка модифицируем). Но кроме него нам нужно реализовать еще две вспомогательных функции:

```
/* добавляет новый узел в связный список */
void llist_add(struct lnode **q, int num);
/* выводит результат */
void llist_print(void);
```

Рассмотрим сначала функцию llist_add(). Ей передаются два параметра - указатель на список и число, которое нужно добавить в список. Если список пуст, то она создает первый узел - выделяет память с помощью malloc():

```
if(*q == NULL) {
  *q = malloc(sizeof(struct lnode));
```

Функция "перематывает" список, чтобы добраться к последнему узлу:

```
  /* переходим к последнему узлу */
  while(tmp->next != NULL)
   tmp = tmp->next;

   /* добавляем узел в конец списка */
   tmp->next = malloc(sizeof(struct lnode));
   tmp = tmp->next;
 }
```

Напомню, последним считается узел, у которого указатель на следующий узел (next) равен NULL. Поэтому в самой "перемотке" нет ничего сложного - нужно двигаться, пока next не будет равен NULL.

Как только мы "перемотали" список и добрались до последнего элемента, нужно присвоить ему данные:

```
tmp->data = num;
tmp->next = NULL;
```

Функция вывода связного списка очень проста. Она похожа на перемотку списка, только при этой самой перемотке мы выводим значение текущего элемента списка:

```
void llist_print(void) {
 visit = head;

 while(visit != NULL) {
  cout << visit->data << " ";
  visit = visit->next;
 }
 cout << endll
}
```

При программировании связных списков очень важно не "потерять голову". Следите за указателем head - одно "неправильное движение" и вы можете потерять весь список. Именно поэтому везде нужно работать с указателем visit (можете назвать его temp - это уже как вам захочется). А указатель head должен оставаться неизменным.

Сортировка связного списка осуществляется функцией llist_bubble_sort(). В ней, как и в предыдущем случае, есть два цикла - внешний и внутренний, только для большего удобства циклы заменены на while():

```
while(e != head->next) {
 c = a = head;
 b = a->next;
  while(a != e) {
```

Полный код программы приведен в листинге 14.7, а результат ее выполнения - на рис. 14.7. В программе мы будем генерировать случайные числа, и ними же будем заполнять наш список - чтобы избавить вас от ввода чисел вручную.

Листинг 14.7. Пузырьковая сортировка связного списка

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

#define MAX 10

using namespace std;

struct lnode {
 int data;
 struct lnode *next;
} *head, *visit;

/* добавляем новый узел в связный список */
void llist_add(struct lnode **q, int num);
/* осуществляем сортировку связного списка */
void llist_bubble_sort(void);
/* выводим результат */
void llist_print(void);

int main(void) {
 /* связный список */
 struct lnode *newnode = NULL;
 int i = 0;      /* общий счетчик */

 /* загружаем случайные числа в связный список */
 for(i = 0; i < MAX; i++) {
  llist_add(&newnode, (rand() % 100));
 }

 head = newnode;
 cout << "До сортировки:\n";
 llist_print();
 cout << "После:\n";
 llist_bubble_sort();
```

```
 llist_print();

 return 0;
}

/* добавляем узел в конец связного списка */
void llist_add(struct lnode **q, int num) {
 struct lnode *tmp;

 tmp = *q;

 /* если список пуст, создаем первый узел */
 if(*q == NULL) {
  *q = malloc(sizeof(struct lnode));
   tmp = *q;
 } else {
  /* переходим к последнему узлу */
  while(tmp->next != NULL)
   tmp = tmp->next;

   /* добавляем узел в конец списка */
   tmp->next = malloc(sizeof(struct lnode));
   tmp = tmp->next;
 }

 /* присваиваем данные последнему узлу */
 tmp->data = num;
 tmp->next = NULL;
}

/* выводим связный список */
void llist_print(void) {
 visit = head;

 while(visit != NULL) {
  cout << visit->data << " ";
  visit = visit->next;
 }
 cout << endl;
}
```

```
/* пузырьковая сортировка связного списка */
void llist_bubble_sort(void) {
 struct lnode *a = NULL;
 struct lnode *b = NULL;
 struct lnode *c = NULL;
 struct lnode *e = NULL;
 struct lnode *tmp = NULL;

// Алгоритм пузырьковой сортировки, адаптированный
// под связный список
while(e != head->next) {
 c = a = head;
 b = a->next;
  while(a != e) {
   if(a->data > b->data) {
    if(a == head) {
     tmp = b -> next;
     b->next = a;
     a->next = tmp;
     head = b;
     c = b;
    } else {
     tmp = b->next;
     b->next = a;
     a->next = tmp;
     c->next = b;
     c = b;
    }
   } else {
    c = a;
    a = a->next;
   }
   b = a->next;
   if(b == e)
    e = a;
  }
 }
}
```

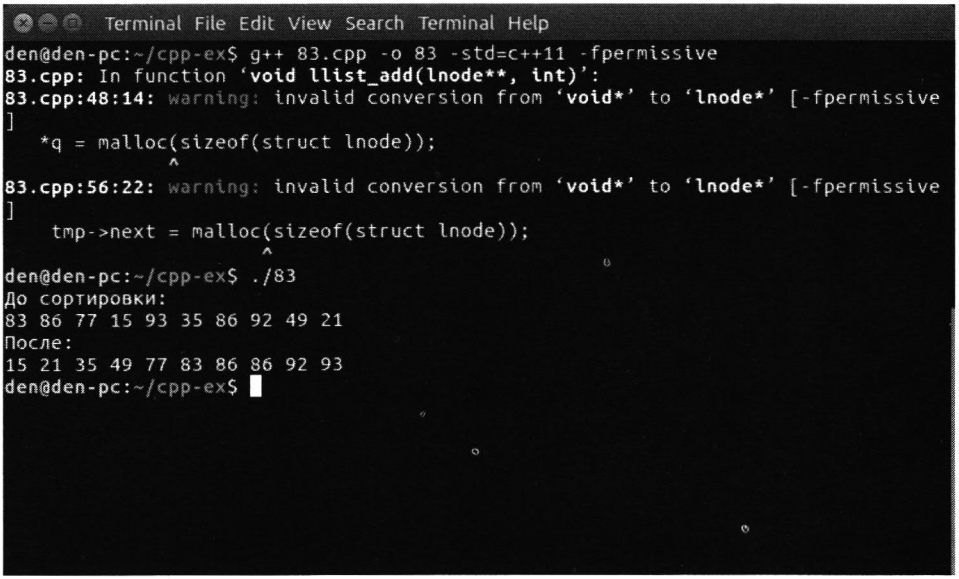


Рис. 14.7. Программа в действии. Пузырьковая сортировка связного списка

14.2.5. Пирамидальная сортировка

Наш следующий пример - пирамидальная сортировка, она же сортировка кучей (heap sort). Данный алгоритм является модификацией пузырьковой сортировки и представляет собой что-то среднее между сортировкой выбором и пузырьковой сортировкой.

Идея алгоритма заключается в следующем: ищем максимальный элемент в неотсортированной части массива и ставим его в конец этого подмассива. В поисках максимума подмассив перестраивается в так называемое сортирующее дерево (она же двоичная куча, она же пирамида), в результате чего максимум сам "всплывает" в начало массива.

После этого над оставшейся частью массива снова осуществляется процедура перестройки в сортирующее дерево с последующим перемещением максимума в конец подмассива.

Что такое сортирующее дерево? Это такое дерево, у которого любой родитель не меньше, чем каждый из его потомков - так называемое неубывающее дерево. Есть и невозрастающее дерево - это когда любой родитель не больше, чем каждый из его потомков.

Листинг 14.8. Пирамидальная сортировка

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

/* максимальная длина массива ... */
#define MAXARRAY 5

using namespace std;

/* осуществляет пирамидальную сортировку */
void heapsort(int ar[], int len);
/* помогает heapsort() "выталкивать" элементы, начиная с позиции
pos */
void heapbubble(int pos, int ar[], int len);

int main(void) {
 int array[MAXARRAY];
 int i = 0;

 /* загружаем случайные элементы в массив */
 for(i = 0; i < MAXARRAY; i++)
  array[i] = rand() % 100;

 /* выводим исходный массив */
 cout << "До: ";
 for(i = 0; i < MAXARRAY; i++) {
  cout << array[i] << " ";
 }
 cout << endl;

 /* Сортировка */
 heapsort(array, MAXARRAY);

 /* результат */
 cout << "После: ";
```

```
 for(i = 0; i < MAXARRAY; i++) {
  cout << array[i] << " ";
 }
 cout << endl;

 return 0;
}

void heapbubble(int pos, int array[], int len) {
 int z = 0;
 int max = 0;
 int tmp = 0;
 int left = 0;
 int right = 0;

 z = pos;
 for(;;) {
  left = 2 * z + 1;
  right = left + 1;

  if(left >= len)
   return;
  else if(right >= len)
   max = left;
  else if(array[left] > array[right])
   max = left;
  else
   max = right;

  if(array[z] > array[max])
   return;

  tmp = array[z];
  array[z] = array[max];
  array[max] = tmp;
  z = max;
 }
}

void heapsort(int array[], int len) {
 int i = 0;
```

```
 int tmp = 0;

 for(i = len / 2; i >= 0; --i)
  heapbubble(i, array, len);

 for(i = len - 1; i > 0; i--) {
  tmp = array[0];
  array[0] = array[i];
  array[i] = tmp;
  heapbubble(0, array, i);
 }
}
```

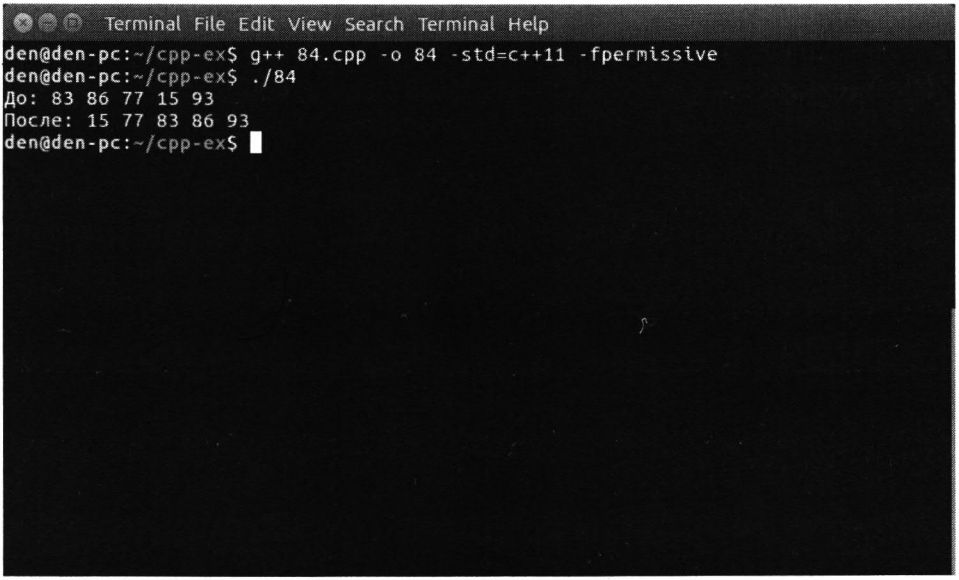


Рис. 14.8. Результат работы программы

Как видно на рис. 14.8, программа сгенерировала случайные значения, поместила их в массив и выполнила сортировку этого массива.

14.2.6. Сортировка вставкой массива по убыванию и по возрастанию

Ранее мы рассмотрели сортировку вставкой связного списка. Но сортировать вставкой можно не только связные списки, хотя, нужно признаться, что

делать это в случае со связным списком - одно удовольствие, учитывая наличие указателей. В этом примере будет показана сортировка вставкой массива float-чисел.

У нас будут два массива. Первый мы оставим в качестве исходного, чтобы его можно было вывести для сравнения, а второй отсортируем вставкой. Для сортировки мы будем использовать написанную нами же функцию isort(), которой нужно передать массив элементов и его размер - количество элементов в массиве. Функция fm() используется для поиска минимума в массиве, точнее в его промежутке, который задается параметрами b и n. Функция ищет минимум и возвращает его позицию.

Листинг 14.9. Сортировка вставкой массива float-чисел

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

using namespace std;

void isort(float arr[], int n);
int fm(float arr[], int b, int n);

int main(void) {
 float arr1[5] = {4.3, 6.7, 2.8, 8.9, 1.0};
 float arr2[5] = {4.3, 6.7, 2.8, 8.9, 1.0};
 int i = 0;

 isort(arr2, 5);

 cout << "\nДо\tПосле\n--------------\n";

 for(i = 0; i < 5; i++)
  cout << arr1[i] << "\t" << arr2[i] << endl;

 return 0;
}

int fm(float arr[], int b, int n) {
 int f = b;
 int c;
```

```
 for(c = b + 1; c < n; c++)
  if(arr[c] < arr[f])
   f = c;

 return f;
}

void isort(float arr[], int n) {
 int s, w;
 float sm;

 for(s = 0; s < n - 1; s++) {
  w = fm(arr, s, n);
  sm = arr[w];
```

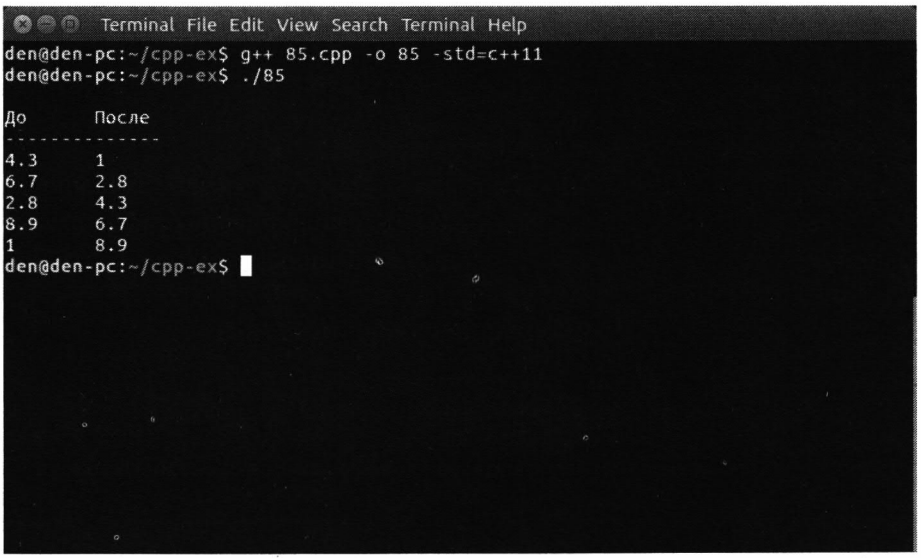


Рис. 14.9. Сортировка массива чисел (по убыванию и возрастанию)

```
  arr[w] = arr[s];
  arr[s] = sm;
 }
}
```

Примечательно, что если изменить функцию fm() так, чтобы она возвращала не меньший, а больший элемент, то есть искала максимум, а не минимум, то сортировка будет не по возрастанию, а по убыванию. Код функции fm() для сортировки по убыванию следующий:

```
int fm(float arr[], int b, int n) {
 int f = b;
 int c;

 for(c = b + 1; c < n; c++)
  if(arr[c] > arr[f])
   f = c;

 return f;
}
```

14.2.7. Сортировка слиянием

Связный список

Рассмотрим еще один алгоритм сортировки - сортировка слиянием (merge sort в англ. литературе). Это, нужно отметить, довольно эффективный алгоритм.

Сортировка слиянием — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

Слияние означает объединение двух (или более) последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент.

Алгоритм довольно непростой. Попробую объяснить все по-простому. У нас есть два списка (или массива - не важно). Мы будем брать поочередно по одному элементу из каждого массива, сравнивать их и "сливать" в один массив. Меньший элемент будем ставить первым, больший – вторым.

А что делать, если у нас есть только один список (массив)? Тогда его нужно разбирать на две части примерно одинакового размера. Далее каждая из получившихся частей сортируется отдельно, после чего два упорядоченных массива соединяются в один. Это и есть сортировка слиянием.

В процессе сортировки мы рекурсивно вызываем функцию сортировки, пока размер массива не достигнет единицы. Любой массив (список), состоящий из одного элемента, можно считать упорядоченным. За сортировку слиянием отвечает функция mergesort(), которая была реализована специально для этого примера:

```
struct node *mergesort(struct node *head) {
 struct node *head_one;
 struct node *head_two;

 if((head == NULL) || (head->next == NULL))
  return head;

 head_one = head;
 head_two = head->next;
 while((head_two != NULL) && (head_two->next != NULL)) {
  head = head->next;
  head_two = head->next->next;
 }
 head_two = head->next;
 head->next = NULL;

 return merge(mergesort(head_one), mergesort(head_two));
}
```

Поскольку мы используем рекурсию, то мы должны предусмотреть условие выхода из рекурсии. В нашем случае условие выхода будет таким:

```
if((head == NULL) || (head->next == NULL))
```

```
  return head;
```

То есть или список пуст или список состоит из одного элемента (нет следующего, поэтому next указывает на NULL). В этом случае мы возвращаем head, во всех остальных мы возвращаем merge(mergesort(head_one), mergesort(head_two));

Функция merge() выполняет непосредственно слияние списков. Мы передаем ей две головы двух списков, она выполняет слияние и возвращает его результат.

Дополнительную информацию об этом алгоритме вы можете получить на страничке Википедии: **https://goo.gl/natPWf**. На ней также вы найдете реализацию алгоритма на разных языках программирования - C, C++. Не будет лишнем и просмотреть визуализацию алгоритма - как он работает. А я привожу собственную реализацию - см. листинг 14.10.

Листинг 14.10. Сортировка связного списка слиянием

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

using namespace std;

struct node {
 int number;
 struct node *next;
};

/* добавляем узел в связный список  */
struct node *addnode(int number, struct node *next);
/* сортировка слиянием */
struct node *mergesort(struct node *head);
/* слияние списков */
struct node *merge(struct node *head_one, struct node *head_two);

int main(void) {
 struct node *head;
 struct node *current;
 struct node *next;
```

```
 int test[] = {8, 3, 2, 6, 1, 5, 4, 7, 9, 0};
 int i;

 head = NULL;
 /* вставляем числа в связный список */
 for(i = 0; i < 10; i++)
  head = addnode(test[i], head);

 /* сортируем список */
 head = mergesort(head);

 /* выводим результат */
 cout << "До После\n";
 i = 0;
 for(current = head; current != NULL; current = current->next)
  cout << test[i++] << " " << current->number << endl;

 /* освобождаем память */
 for(current = head; current != NULL; current = next)
  next = current->next, free(current);

 /* все */
 return 0;
}

/* добавляем узел в связный список */
struct node *addnode(int number, struct node *next) {
 struct node *tnode;

 tnode = (struct node*)malloc(sizeof(*tnode));

 if(tnode != NULL) {
  tnode->number = number;
  tnode->next = next;
 }

 return tnode;
}

/* сортировка слиянием связного списка */
struct node *mergesort(struct node *head) {
```

```
 struct node *head_one;
 struct node *head_two;

 if((head == NULL) || (head->next == NULL))
  return head;

 head_one = head;
 head_two = head->next;
 while((head_two != NULL) && (head_two->next != NULL)) {
  head = head->next;
  head_two = head->next->next;
 }
 head_two = head->next;
 head->next = NULL;

 return merge(mergesort(head_one), mergesort(head_two));
}

/* слияние списков */
struct node *merge(struct node *head_one, struct node *head_two)
{
 struct node *head_three;

 if(head_one == NULL)
  return head_two;

 if(head_two == NULL)
  return head_one;

 if(head_one->number < head_two->number) {
  head_three = head_one;
  head_three->next = merge(head_one->next, head_two);
 } else {
  head_three = head_two;
  head_three->next = merge(head_one, head_two->next);
```

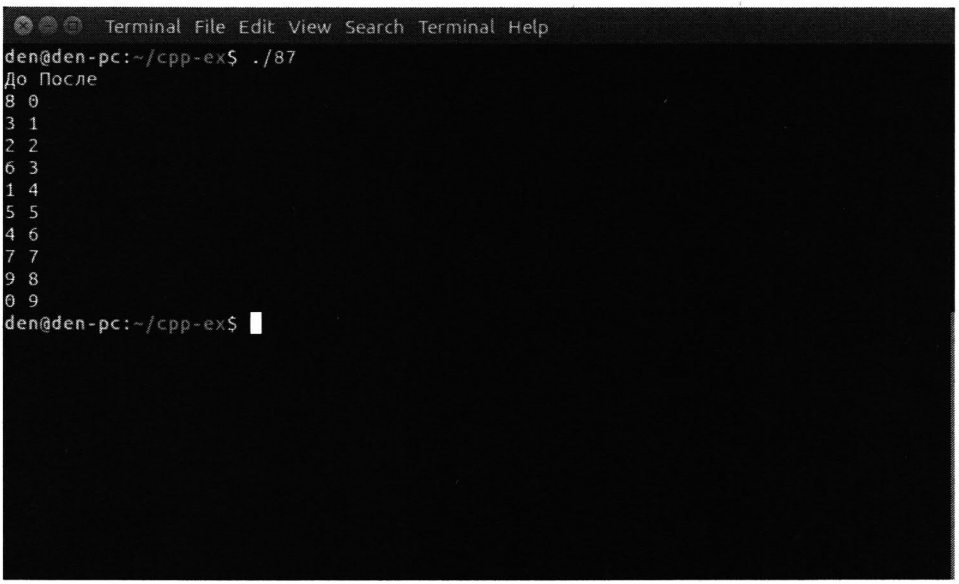


Рис. 14.10. Результат сортировки слиянием

```
 }

 return head_three;
}
```

Сортировка массива

Рассмотрим еще один пример сортировки слиянием, на этот раз сортировать будем не связный список, а массив целых чисел. Сам алгоритм остается тем же, но функция mergesort() будет адаптирована под работу с массивом. Также не будет функции merge(), а слиянием будем производить сразу в функции mergesort().

Переменная pivot - это центр массива. Мы разбиваем массив на две части (условно) и для каждой запускаем процесс сортировки:

```
mergesort(a, low, pivot);
mergesort(a, pivot + 1, high);
```

Условие выхода из рекурсии - когда low = high, то есть массив у нас состоит из одного элемента:

```
if(low == high)
  return;
```

Полный код сортировки слиянием массива приведен в листинге 14.11. Результат сортировки массива показан на рис. 14.11.

Листинг 14.11. Сортировка слиянием массива

```
#include <iostream>
#include <cstdlib>   // для функции rand()
using namespace std;

#define MAXARRAY 10

void mergesort(int a[], int low, int high);

int main(void) {
 int array[MAXARRAY];
 int i = 0;

 /* загружаем в массив случайные данные */
 for(i = 0; i < MAXARRAY; i++)
  array[i] = rand() % 100;

 /* До сортировки */
 cout << "До сортировки:";
 for(i = 0; i < MAXARRAY; i++)
  cout << array[i] << " ";

cout << endl;

 /* Сортировка */
 mergesort(array, 0, MAXARRAY - 1);

 /* после */
 cout << "После:";
 for(i = 0; i < MAXARRAY; i++)
  cout << array[i] << " ";
```

```
 cout << endl;
 return 0;
}

void mergesort(int a[], int low, int high) {
 int i = 0;
 int length = high - low + 1;
 int pivot  = 0;
 int merge1 = 0;
 int merge2 = 0;
 int working[length];

 if(low == high)
  return;

 pivot  = (low + high) / 2;

 mergesort(a, low, pivot);
 mergesort(a, pivot + 1, high);

 for(i = 0; i < length; i++)
  working[i] = a[low + i];

 merge1 = 0;
 merge2 = pivot - low + 1;

 for(i = 0; i < length; i++) {
  if(merge2 <= high - low)
   if(merge1 <= pivot - low)
    if(working[merge1] > working[merge2])
     a[i + low] = working[merge2++];
    else
     a[i + low] = working[merge1++];
   else
    a[i + low] = working[merge2++];
  else
   a[i + low] = working[merge1++];
 }
}
```

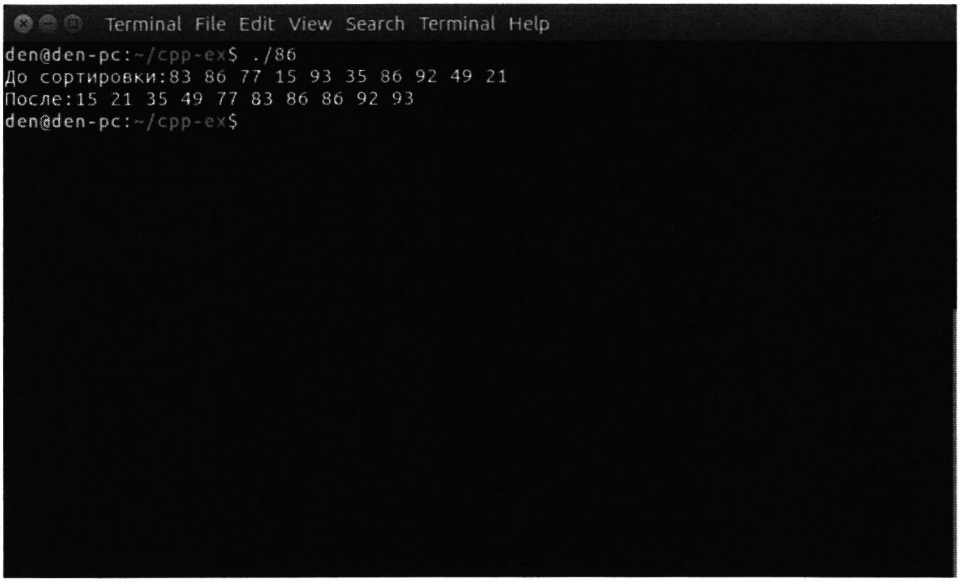


Рис. 14.11. Сортировка массива слиянием

Сортировка массива строк стандартными средствами

В этой главе были рассмотрены различные алгоритмы сортировки. В первую очередь - для развития ваших навыков программирования, чтобы продемонстрировать, как можно практически работать с массивами и списками в C++. Конечно же, есть и стандартные, уже готовые средства сортировки и вам не придется изобретать колесо.

Функция sort() может использоваться для сортировки массива строк. Ей нужно передать первый и последний элементы массива. С первым элементом все ясно - можно передать просто сам массив. А вот, чтобы вычислить последний элемент массива, нужно знать размер самого массива и размер одного элемента. Размер массива и одного элемента можно узнать

функций sizeof(). Затем к нашему массиву нужно добавить полученное число - размер массива плюс размер одного элемента. Готовый пример кода приведен в листинге 14.12.

Листинг 14.12. Сортировка с использованием функции sort()

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main() {
    string name[] = {"john", "bobby", "dear", "test1",
"catherine", "nomi", "shinta", "martin", "abe", "may", "zeno",
"zack", "angeal", "gabby"};

    int sname = sizeof(name)/sizeof(name[0]);

    sort(name, name + sname);

    for(int i = 0; i < sname; ++i)
        cout << name[i] << endl;
```

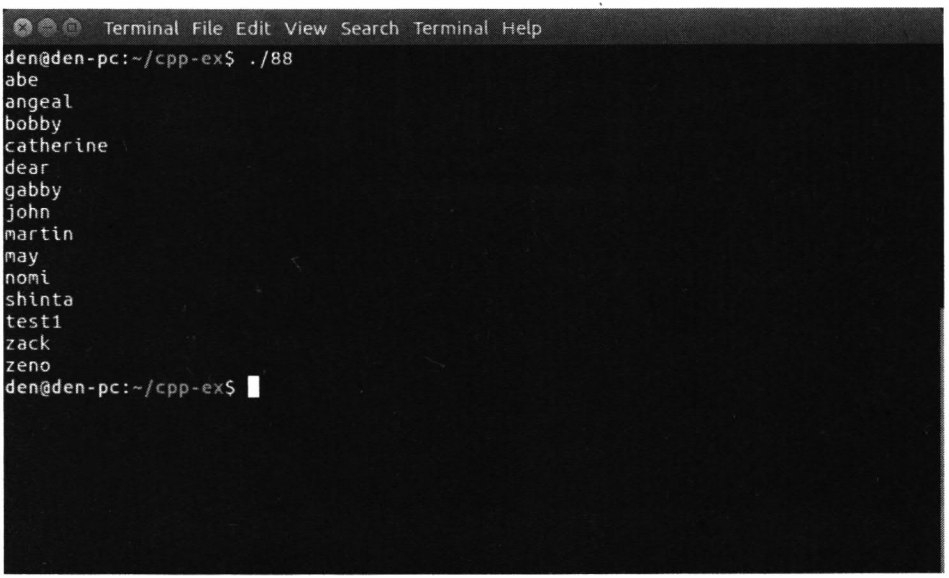
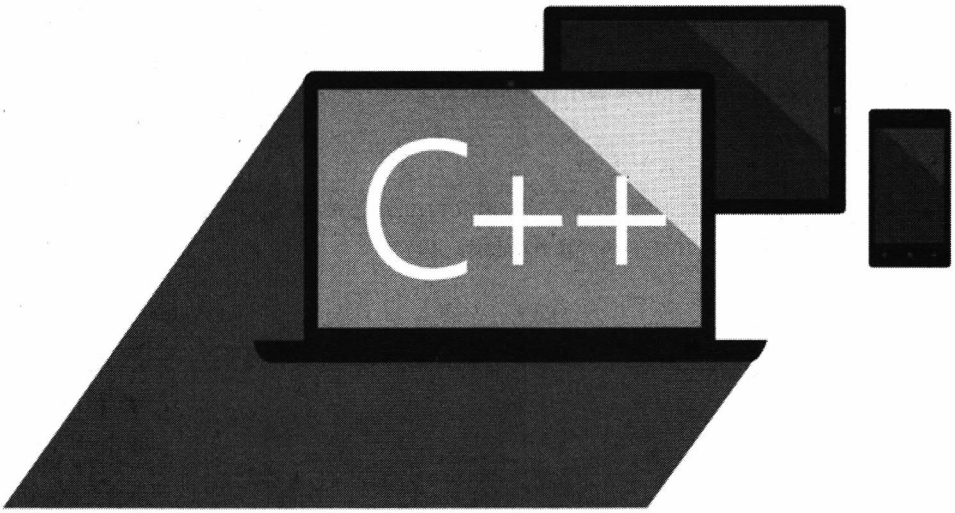


Рис. 14.12. Сортировка с использованием функции sort()

Приложения



Стандартные заголовочные файлы

Нижеперечисленные файлы содержат объявления Стандартной Библиотеки. Стандартная библиотека - это библиотека, которая изначально входит в состав языка.

Контейнеры

<bitset>

Реализует специализированный класс контейнеров std::bitset — битовый массив.

<deque>

Реализует шаблон класса контейнера std::deque — двусвязная очередь.

<list>

Реализует шаблон класса контейнера std::list — двусвязный список.

<map>

Реализует шаблоны классов контейнеров std::map и std::multimap — Ассоциативный массив и мультиотображение.

<queue>

Реализует класс адаптер-контейнера std::queue — односторонняя очередь.

<set>

Реализует шаблоны классов контейнеров std::set и std::multiset — сортированные ассоциативные контейнеры или множества.

<stack>

Реализует класс адаптер-контейнера std::stack — стек.

<vector>

Реализует шаблон класса контейнеров std::vector — динамический массив.

Общие

<algorithm>

Реализует определения многих алгоритмов для работы с контейнерами.

<functional>

Реализует несколько объект-функций, разработанных для работы со стандартными алгоритмами.

<iterator>

Реализует классы и шаблоны для работы с итераторами.

<locale>

Реализует классы и шаблоны для работы с локалями.

<memory>

Реализует инструменты управления памятью в C++, включая шаблон класса std::auto_ptr.

<stdexcept>

Содержит стандартную обработку ошибок классов, например, std::logic_error и std::runtime_error, причем оба происходят из std::exception.

<utility>

реализует шаблон класса std::pair для работы с парами (двучленными кортежами) объектов.

Строковые

<string>

Реализует стандартные строковые классы и шаблоны.

Поточные и ввода-вывода

<fstream>

Реализует инструменты для файлового ввода и вывода. Смотри fstream.

<ios>

Реализует несколько типов и функций, составляющих основу операций с iostreams.

<iostream>

Реализует основы ввода и вывода языка C++. Смотрите iostream.

<iosfwd>

Реализует предварительные объявления нескольких шаблонов классов, связанных с вводом-выводом.

<iomanip>

Реализует инструменты для работы с форматированием вывода, например базу, используемую при форматировании целых и точных значений чисел с плавающей запятой.

<istream>

Реализует шаблон класса std::istream и других необходимых классов для ввода.

<ostream>

Реализует шаблон класса std::ostream и других необходимых классов для вывода.

<sstream>

Реализует шаблон класса std::sstream и других необходимых классов для работы со строками.

<streambuf>

Числовые

<complex>

Реализует шаблон класса std::complex и связанные функции для работы с комплексными числами.

<numeric>

Реализует алгоритмы для числовой обработки

<valarray>

Реализует шаблон класса std::valarray — класс массивов, оптимизированный для числовой обработки.

Языковая поддержка

<exception>

Реализует несколько типов и функций, связанных с обработкой исключений, включая std::exception — базовый класс всех перехватов исключений в Стандартной Библиотеке.

<limits>

реализует шаблон класса std::numeric_limits, используемый для описания свойств базовых числовых типов.

<new>

Реализует операторы new и delete, а также другие функции и типы, составляющие основу управления памятью в C++.

<typeinfo>

Реализует инструменты для работы с динамической идентификацией типа данных в C++.

Для заметок

Издательство "Наука и Техника" рекомендует:

JAVASCRIPT НА ПРИМЕРАХ. ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА — СПб.: Наука и Техника. — 272 с., ил.

Серия «На примерах»

Эта книга является превосходным учебным пособием для изучения языка программирования JavaScript на примерах. Изложение ведется последовательно: от написания первой программы, до создания полноценных проектов: интерактивных элементов (типа слайдера, диалоговых окон) интернет-магазина, лендинговой страницы и проч. По ходу даются все необходимые пояснения и комментарии.

Книга написана простым и доступным языком. Лучший выбор для результативного изучения JavaScript!

Издательство "Наука и Техника" рекомендует:

JAVA НА ПРИМЕРАХ. ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА — СПб.: Наука и Техника. — 256 с., ил.

Серия «На примерах»

Эта книга является превосходным базовым учебным пособием для изучения языка программирования Java с нуля. В книге содержатся рецепты и практические указания по решению задач, часто встречающихся при программировании на языке Java. Многие авторы книг в своих трудах рассматривают теоретические основы языка и уделяют основное внимание базовому синтаксису языка, не рассматривая при этом практическую сторону его применения. Эта же книга старается восполнить недостаток практического материала, содержит множество примеров с комментариями, которые вы сможете использовать в качестве основы своих программных решений при изучении Java.

Орленко Павел Алексеевич,
Евдокимов Петр Валентинович

C++ на примерах

ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

ООО "Наука и Техника"

Лицензия №000350 от 23 декабря 1999 года.

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107.

Подписано в печать 09.11.2018. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 18 п. л.

Тираж 1500. Заказ 11090.

Отпечатано с готовых файлов заказчика
в АО "Первая Образцовая типография"
филиал "УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ"
432980, г. Ульяновск, ул. Гончарова, 14.

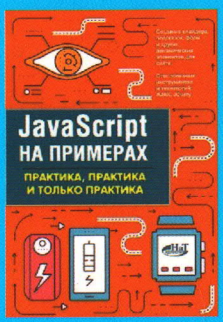
Орленко П. А., Евдокимов П. В.

C++ НА ПРИМЕРАХ

ПРАКТИКА, ПРАКТИКА И ТОЛЬКО ПРАКТИКА

Эта книга является превосходным учебным пособием для изучения языка программирования C++ на примерах.
В книге рассмотрена базовая теоретическая часть языка C++, позволяющая ориентироваться в языке и создавать свои программы: типы, функции, операторы, логические конструкции, массивы, указатели, структуры, работа с файлами, объектно-ориентированное программирование. Отдельное внимание уделено программированию различных алгоритмов. В книге используется большое количество примеров с подробным анализом кода: от простых приложений для вывода текста на экран и проведения вычислений до клиент-серверного приложения.
Будет полезна как начинающим программистам, студентам, так и всем, кто хочет быстро начать программировать на C++.

Издательство "Наука и Техника" рекомендует:



ISBN 978-5-94387-772-8
978-5-94387-772-8

Издательство "Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru