Акимова А.В., Кольцов Д.М.

100 примеров на C++

НиТ издательство

"Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973.2
ISBN 978-5-94387-756-8

Акимова А.В., Кольцов Д.М.

100 ПРИМЕРОВ НА С++ — СПб.: "Наука и Техника", 2018. — 256 с., ил.

Серия "Просто о сложном"

Эта книга является превосходным учебным пособием для изучения языка программирования С++ на примерах. Изложение ведется последовательно: от написания первой программы, до многопоточного программирования, создания клиент-серверных приложений, объектно-ориентированного программирования и программирования игр. По ходу изложения даются все необходимые пояснения и комментарии.

Книга написана простым и доступным языком. Начните сразу писать программы на С++. Лучший выбор для результативного изучения С++

ISBN 978-5-94387-756-8

9 78- 5- 94387 - 756- 8

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

Содержание

Вместо введения

Данная книга - это не очередной самоучитель по C++. Таких самоучителей достаточно много и все они скучные, поскольку в большинстве случаев они повторяют официальную документацию. Возьмите два-три самоучителя или справочника по C++ - они будут все как один - похожи друг на друга.

Данная книга - совсем другое. Скорее, это сборник реальных задач, решенных с помощью языка программирования C++. Книга будет полезна студентам, изучающих программирование на C++ - они найдут в ней большинство задач, которым им придется решать в процессе обучению программированию на C++.

Все примеры тщательно протестированы. Для их компиляции вы можете использовать любой компилятор. Для каждого примера приводится его исходный код и скриншот, подтверждающий работоспособность программы. Для компиляции программ мы использовали компилятор **g++**, но вы можете использовать любой из них.

В большинстве случае откомпилировать программу можно командой **g++** <имя файла исходного кода> -o <имя_исполняемого_файла>. Если компиляция программы потребует дополнительных опций, мы об этом сообщим.

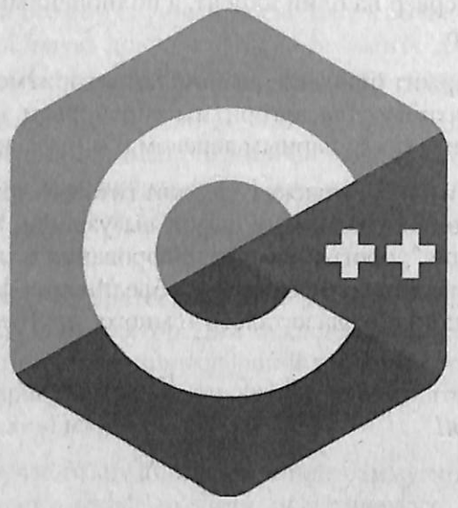
Книга разделена на несколько частей:

- **Часть 1** знакомит читателя с самыми простыми примерами. Подобные программы может написать любой, кто только-только познакомился с синтаксисом C++.
- **Часть 2** описывает операторы принятия решений и циклы. Здесь программы уже значительно сложнее.
- **Часть 3** демонстрирует, как создавать и использовать собственные функции.
- **Часть 4** одна из самых сложных, поскольку затрагивает самую сложную для новичков тему - указатели. Очень много ошибок в программах начинающих программистов связано с указателями.
- **Часть 5** - показывает, как работать со строками.
- **Часть 6** демонстрирует несколько примеров, показывающих как работать с различными структурами данных.
- **Часть 7** содержит несколько примеров по работе с файлами.

- **Глава 8** содержит примеры, относящиеся к объектно-ориентированному программированию.
- **Часть 9** описывает готовое приложение клиент-сервер, которое может стать даже предметом курсовой работы. В этой главе будет описан не простой эхо-сервер на один клиент, а полноценный многопотоковый сетевой сервер.
- **Часть 10** содержит описание различных алгоритмов поиска и сортировки. Рассматриваются алгоритмы сортировки массивов, связных списков, сортировка бинарным деревом и многое другое.
- **Часть 11**, как и часть 9, также содержит готовые приложения. Это сугубо практическая глава, из которой вы узнаете, как создать мини-игру "Эволюция", программу для шифрования и расшифровки файлов, программы для копирования и объединения файлов, программу для рекурсивного обхода каталога и многое другое.

Приятного чтения!

Часть 1. Простые примеры



В этой части будут рассмотрены следующие примеры:

Первая часть содержит самые простые примеры и знакомит читателя с синтаксисом C++ на практике. Поскольку - это не учебник по C++, скучного объяснения синтаксиса вы здесь не найдете - смотрите, учитесь, повторяйте, как в книге. Если вы допустите ошибку, то компилятор подскажет, что именно вы сделали не так.

Пример 1. Программа "Hello, world!". Компиляция и запуск программы

Традиционно, многие книги, посвященные программированию, начинаются с такой программы. Суть ее предельно проста: программа выводит строку "Hello, world!". Тем не менее, начинающий программист видит структуру программы, а также знакомиться с одним из основных операторов - оператором вывода.

Листинг 1. Код программы

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

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

Не нужно быть семи пядей во лбу, чтобы догадаться, что выведет программа. На экране будет строчка:

```
Hello, world!
```

Посмотрим, как работает программа:

- В первой строчке мы подключаем заголовочный файл **iostream**, позволяющий работать с потоками ввода/вывода. Если вы ранее программировали на C, то заметили, что название заголовочного файла отличается от привычного вам **stdio.h** и расширение файла не указывается. Хотя можно встретить ранние реализации C++, где еще указывалось расширение файла #include <iostream.h>.
- Вторая строчка использует стандартное пространство имен. В ранних реализациях C++ пространства имен не использовались, но мы ведь пишем современный код (хоть и простой), не так ли?
- Выполнение программы на C начинается с функции **main()** - это точка входа в программу.

- В C++ нет функций ввода/вывода. Вместо них используются операторы ввода/вывода << и >>. Далее (в следующем примере) будет показано, как прочитать ввод с клавиатуры.
- Оператор << выводит на стандартный поток вывода (cout) строчку "Hello, world!"
- Выражение **return 0**; означает код возврата (exit code) программы. С помощью кода возврата вы можете дать другим программа, запустившим вашу программу, как прошло ее выполнение. Код возврата **0** обычно соответствует отсутствию ошибок. Любое другое значение означает код ошибки. Вряд ли вы будете использовать нашу простую программу в сценариях и проверять код ее возврата, поэтому без этой инструкции можно было бы и обойтись, но правила хорошего тона обязывают нас ее использовать.

После того, как программа написана, ее нужно откомпилировать. Вы можете использовать любой компилятор, поддерживающий C++. Далее приведена строка компиляции этой программы с помощью компилятора **g++** (GNU C++ compiler):

```
g++ 1.cpp -o 1
g++ 1.cpp -o 1.exe              (Windows)
```

Примечание. Компилятор **g++** по умолчанию установлен в Linux. О том, как правильно установить его в Windows, можно прочитать по этому адресу http://www1.cmc.edu/pages/faculty/alee/g++/g++.html

Мы задаем имя файла исходного кода (1.cpp) и имя выходного файла (1). Далее запустить программу можно, указав в качестве команды имя выходного файла:

```
./1
```

В Windows это будет 1.exe, поэтому запускать его нужно, указывая расширение и не указывая ./ (текущий каталог):

```
1.exe
```


Посмотрите на рисунок 1. На нем изображен процесс компиляции и запуска программы. Наша программа откомпилировалась, запустилась, вывела строчку и завершила работу. На фоне изображен текстовый редактор Atom (существует, как для Windows, так и для Linux). Именно этот редактор я рекомендую использовать для написания ваших программ. Он поддерживает подсветку синтаксиса, автодополнение кода и многие другие штуки, облегчающие программисту жизнь.

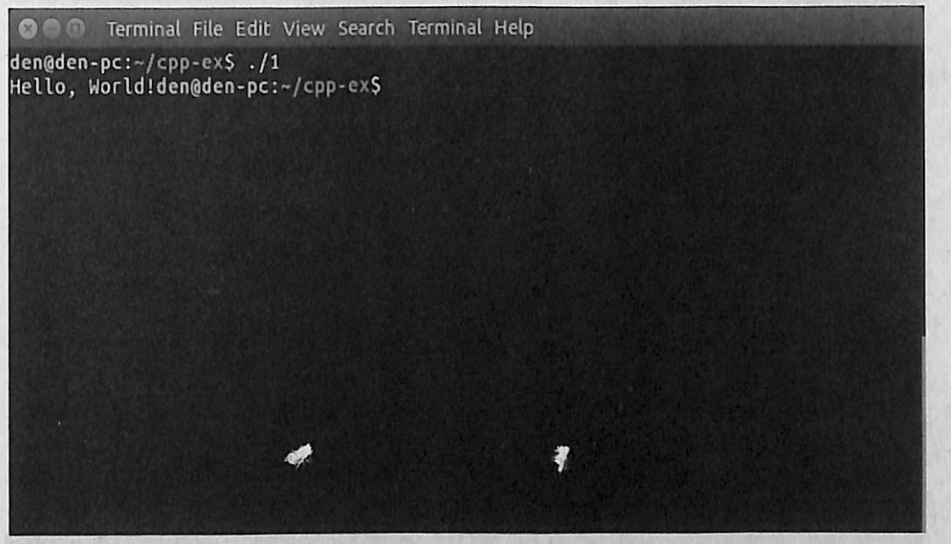


Рис. 1. Компиляция и запуск примера 1

Пример 2. Выводим число, введенное пользователем

Этот пример демонстрирует чтение ввода с клавиатуры, а также устраняет небольшой недостаток программы из примера 1.

Листинг 2. Демонстрация ввода с клавиатуры (2.cpp)

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

```
int main()
{
    int number;

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

    cout << "Вы ввели " << number << "\n";
    return 0;
}
```

Рассмотрим отличия программы от примера 1:

- Мы объявляем целочисленную (**int**) переменную с именем **number**.
- С помощью оператора вывода << мы выводим приглашение ввести число на стандартный поток вывода **cout**.
- Используя оператор ввода >>, мы читаем введенное пользователем значение в переменную **number**.
- Затем мы выводим строку "Вы ввели ", введенное пользователем число и символ новой строки "\n" на стандартный вывод. Посмотрите на рисунок 1. Выводится наша заветная строчка "Hello, world!", затем строка НЕ переносится и сразу же выводится приглашение командной оболочки. С технической точки зрения в нашей программе нет ошибки, но эстетически такой вывод выглядит некрасиво. Поэтому мы добавили символ новой строки к нашему выводу для обеспечения перевода строки.

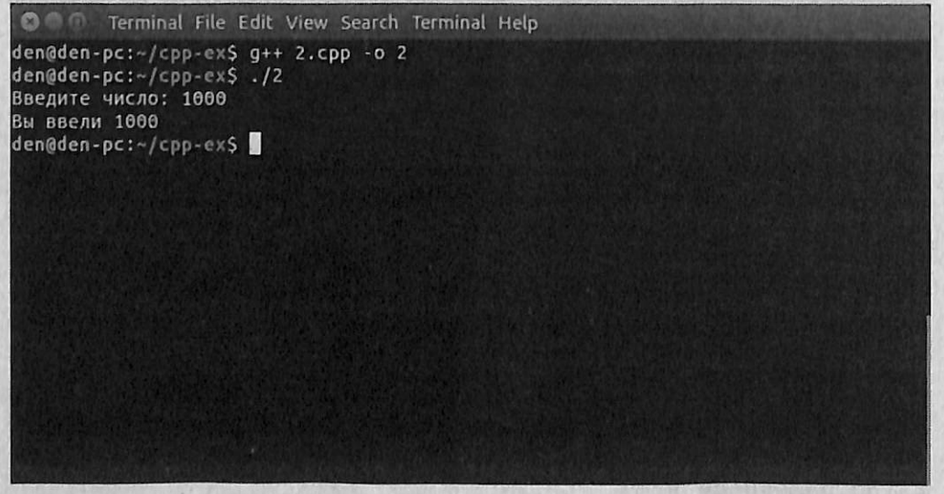


Рис. 2. Пример 2: ввод числа

Пример 3. Сложение двух чисел

Усложним наш пример 2. На этот раз попросим пользователя ввести два целых числа, а затем вычислим сумму этих чисел и сообщим пользователю результат. Листинг примера приведен ниже.

Листинг 3. Сумма двух чисел (3.cpp)

```
#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.
- Константа **endl** означает конец строки, по сути, является тем самым символом "**\n**". Она особо не нужна, но ее использование повышает читабельность результата.

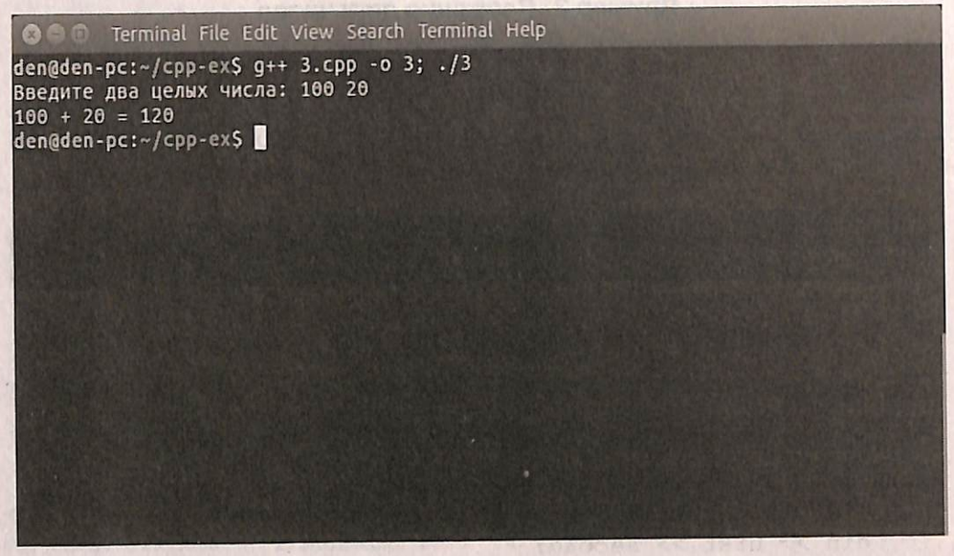


Рис. 3. Сумма двух чисел

Пример 4. Вычисляем частное и остаток

Теперь немного математики: попробуем вычислить частное и остаток при делении двух чисел. Для тех, кто в школе пропустил урок математики, давайте разберемся, что есть что:

- Делимое - это число, стоящее слева от знака деления.
- Делитель - это число, стоящее справа от знака деления, по сути, это число, на которое делим.
- Частное - это число, стоящее после знака равно, результат деления.
- Остаток - это число, оставшееся не делимым, которое меньше делителя.

Теперь несколько примеров:

```
10 / 5 = 2
```

Здесь 10 - делимое, 5 - делитель, 2 - частное.

Теперь другой пример:

```
13 / 5 = 2 (3)
```

Здесь 13 - делимое, 5 - делитель, 2 - неполное частное (так как есть остаток) и 3 - остаток от деления. Осталось только все это запрограммировать, см. листинг 4.

Листинг 4. Вычисляем частное и остаток от деления (4.cpp)

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

int main()
{
    int divisor, dividend, quotient, remainder;

    cout << "Введите делимое: ";
    cin >> dividend;

    cout << "Введите делитель: ";
    cin >> divisor;

    quotient = dividend / divisor;
    remainder = dividend % divisor;

    cout << "Частное = " << quotient << endl;
    cout << "Остаток = " << remainder << endl;

    return 0;
}
```

Результат работы программы приведен на рис. 4.

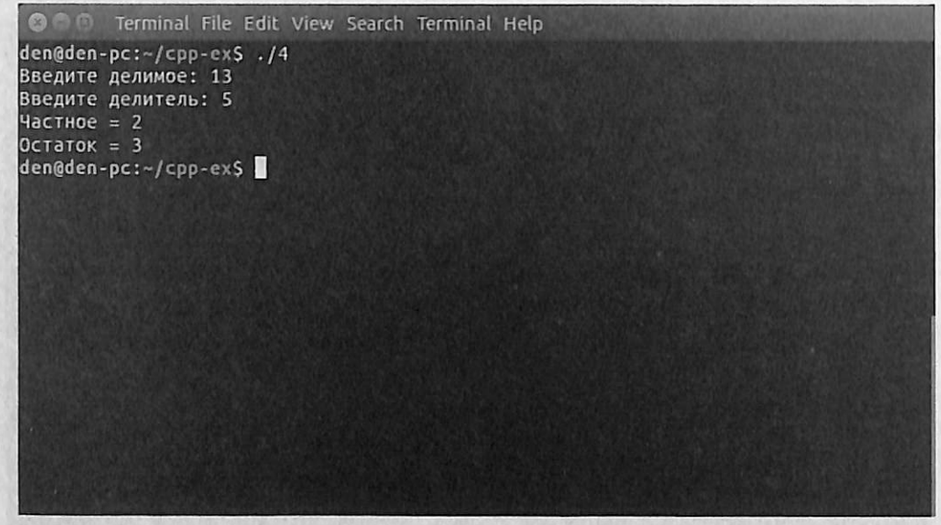


Рис. 4. Вычисляем частное и остаток

Работает программа так:

- Мы вводим делимое и делитель - **dividend** и **divisor** соответственно.
- Используя оператор /, мы вычисляем частное (выражаясь математическим языком - неполное частное во многих случаях)
- Используя оператор **%**, мы вычисляем остаток от деления.
- Наконец, вы выводим частное и остаток с помощью оператора вывода <<.

Пример 5. Вычисляем размер типов int, float, double и char в вашей системе

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 переменной каждого из этих типов может быть разным. 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 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. 5. В зависимости от вашей системы и от вашего компилятора, у вас могут быть другие результаты.

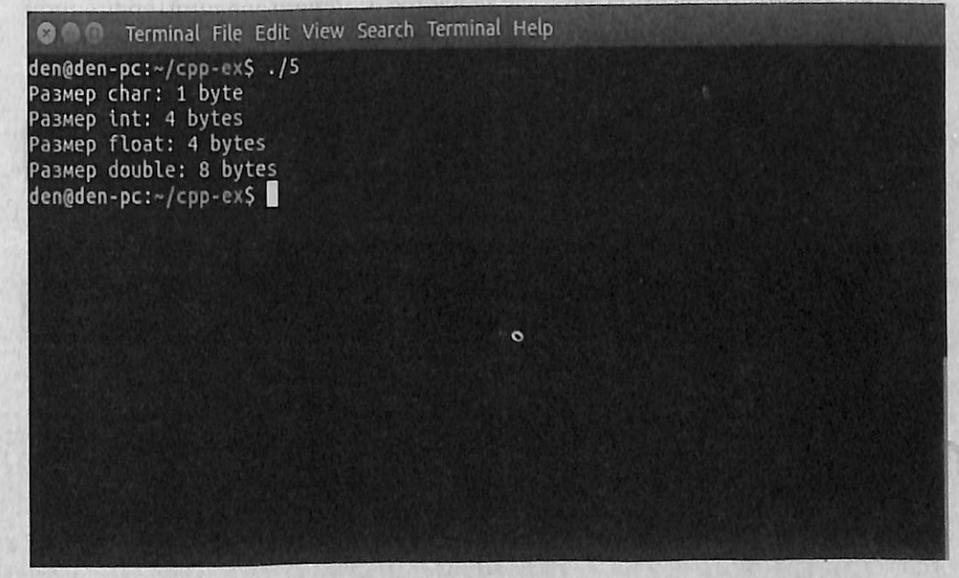


Рис. 5. Вычисление размера популярных типов данных

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.

Пример 6. Меняем местами два числа

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. Мы будем использовать временную переменную **temp** (лист. 6.1).

Листинг 6.1. Первый вариант решения задачи (6-1.cpp)

```
#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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./6-1
До замены:
a = 1, b = 2

После замены:
a = 2, b = 1
den@den-pc:~/cpp-ex$
```

Рис. 6. 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 (лист. 6-2).

Листинг 6.2. 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.

Пример 7. Находим ASCII-значение символа

Этот пример демонстрирует как найти ASCII-значение символа. Символьная переменная (тип **char**) хранит в себе не сам символ, а его ASCII-код. В языке C для вывода ASCII-значения использовалась функция **printf()** с определенным модификатором вывода. В C++ нужно использовать функцию **int()**, см. лист. 7.

Листинг 7. Определение ASCII-значения символа (7.cpp)

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

int main()
{
 char c;
 cout << "Введите символ: ";
 cin >> c;
```



```
 cout << "ASCII-значение: "  << int(c) << endl;
 return 0;
}
```

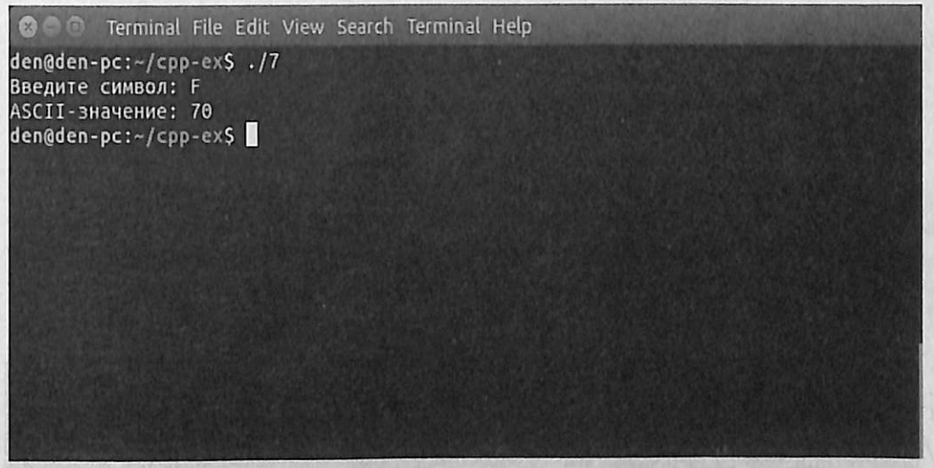


Рис. 7. Результат работы программы

Пример 8. Умножение двух вещественных чисел

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

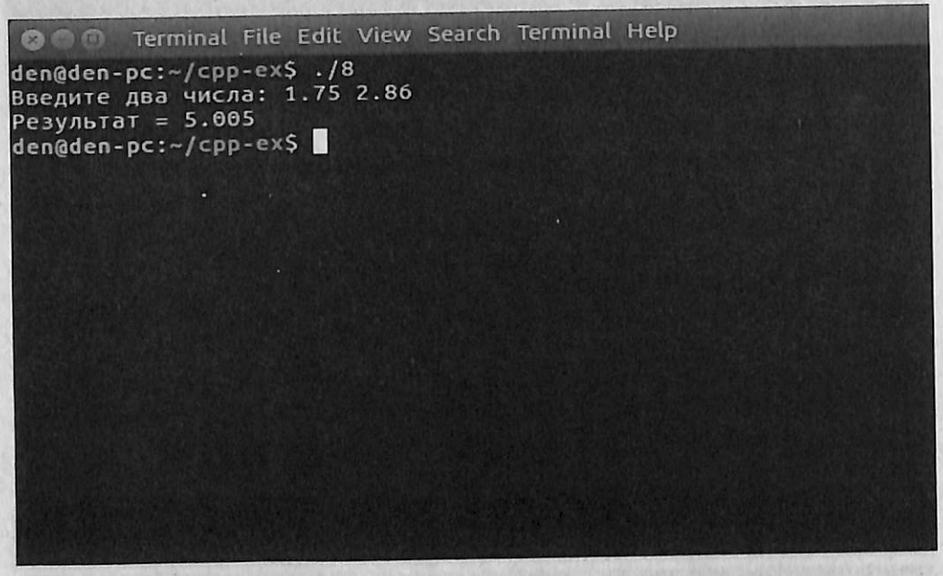
Листинг 8. Умножение двух вещественных чисел (8.cpp)

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

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

    cin >> first >> second;

    product = first * second;
```

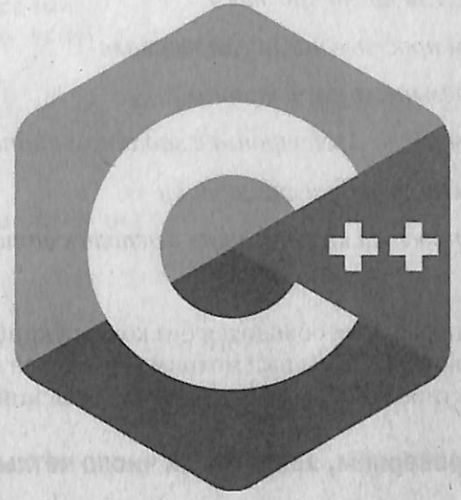
```
    cout << "Результат = " << product << endl;

    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.



Часть 2. Циклы и конструкции принятия решений



В этой части будут рассмотрены следующие примеры:

Какая же серьезная программа обходится без конструкций принятия решений и циклов. В данной части мы рассмотрим множество примеров, демонстрирующих на практике конструкции принятия решений и циклов.

Пример 9. Проверяем, является ли число четным или нет

Для проверки, является ли число четным, используется оператор **%**, позволяющий узнать остаток от деления. Напомню, что 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 10. Мы используем оператор **if**, который имеет следующий вид:

```
if (логическое выражение)
  оператор_1;
else
  оператор_2;
```

Если логическое выражение в скобках окажется истинным, то будет выполнен оператор_1 (или группа операторов, заключенных в фигурные скобки), в противном случае будет выполнен оператор_2.

Условный оператор **if** можно использовать без **else**, например:

```
if (логическое выражение)
  оператор_1;
```


В этом случае, если логическое выражение окажется истинным, будет выполнен оператор_1, а если нет - программа ничего делать не будет, и перейдет к выполнению следующего за **if** оператора. Пример работающей программы приведен на рис. 9.

Листинг 9. Является ли число четным (9.cpp)

```
#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;
}
```

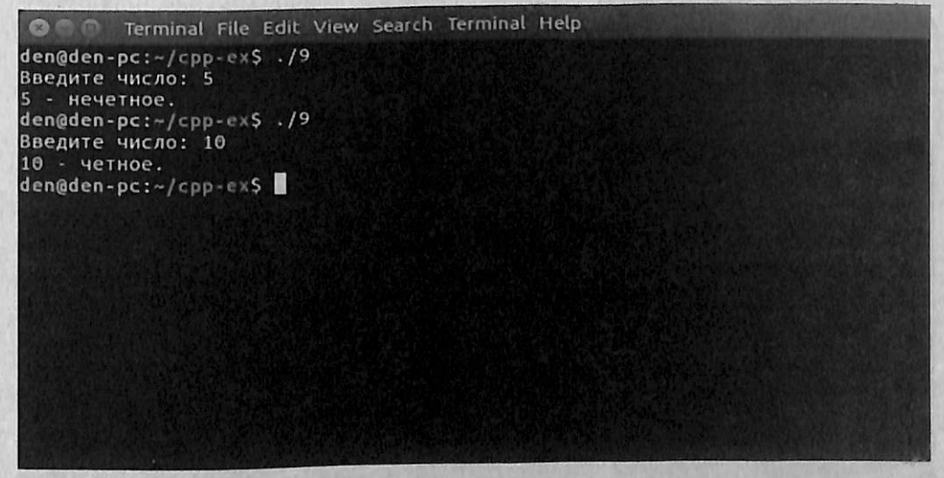


Рис. 9. Программа, определяющая, является ли число четным или нет

Пример 10. Определяем максимум среди трех чисел

В этом примере мы определим максимум среди трех чисел. При всей своей простоте программа демонстрирует, как использовать логический оператор **&&** (логическое И). Данный оператор возвращает **true**, если оба его операнда истинны.

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 10.

Листинг 10. 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. 10. Программа в действии (10.cpp)

Конечно, это не единственный способ решения задачи. Можно определить максимум немного иначе, например, так:

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

int main()
{
    float n1, n2, n3;

    cout << "Введите три целых числа: ";
    cin >> n1 >> n2 >> n3;

    if((n1 >= n2) && (n1 >= n3))
        cout << "Максимум: " << n1;
    else if ((n2 >= n1) && (n2 >= n3))
        cout << "Максимум: " << n2;
    else
        cout << "Максимум: " << n3;
    cout << endl;
    return 0;
}
```

Думаю, программа настолько проста, что не нуждается в комментариях. Ее отличие от первого варианта в том, что она использует оператор **if..else**, а в первом случае мы использовали просто оператор **if** (сокращенную форму условного оператора).

Пример 11. Вычисляем все корни квадратного уравнения. Подключение библиотеки math

Теперь немного математики. Стандартная форма квадратного уравнения выглядит так:

```
ax^2 + bx + c = 0,
где a, b, c - вещественные числа, а 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. 11).

Листинг 11. 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, полученной на страничке Википедии. Корни вычисляются по-разному в зависимости от значения детерминанта.

Данная программа интересна не столько работой условного оператора, сколько математической библиотекой **math.h** и ее правильным подключением. Для вычисления квадратного корня мы используем функцию **sqrt()**. Чтобы эта функция стала доступна, мы подключаем заголовочный файл **math.h**, однако при компиляции программы мы получаем сообщение о том, что ссылка на **sqrt** не определена. Оказывается, математическую библиотеку нужно подлинковать, для этого необходима опция **-lm** компилятора:

```
g++ 11.cpp -o 11 -lm
```

Здесь 11.cpp - имя файла исходного кода, опция **-o** 11 задает имя выходного (исполнимого) файла, а **-lm** подключает библиотеку **math**. Процесс компиляции программы и ее работа изображены на рис. 11.

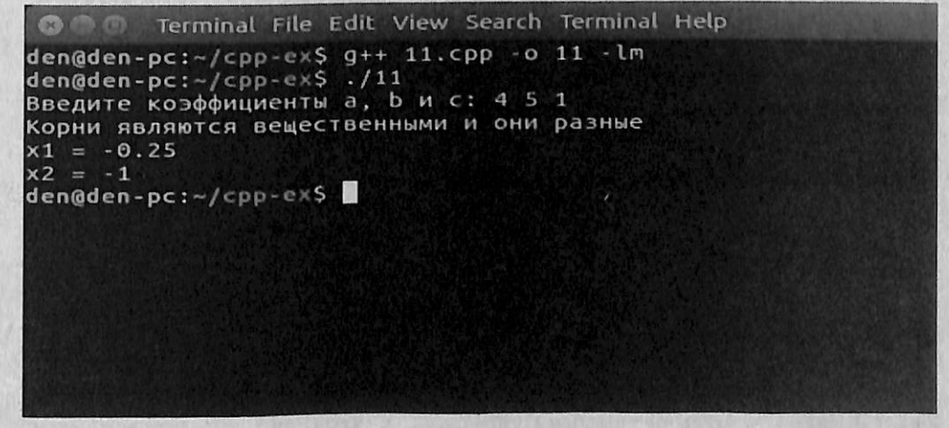


Рис. 11. Программа в действии

Пример 12. Является ли год високосным

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. 12).

Листинг 12. Високосный год? (12.cpp)

```
#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;
}
```

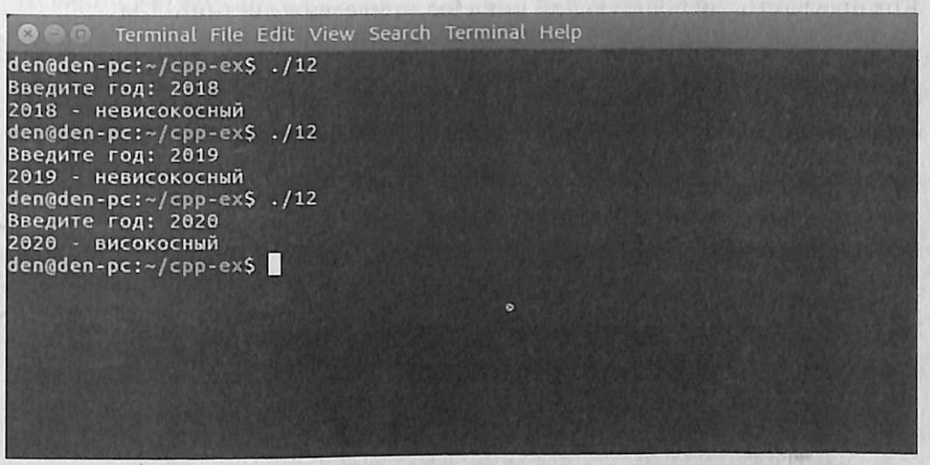


Рис. 12. Високосный год или нет

Пример 13. Вычисляем сумму натуральных чисел

Натуральное число - это целое положительное число. Наша программа будет работать так: пользователь вводит число n, а программа вычисляет сумму чисел от **1** до **n** в цикле.

Вот только какой цикл использовать? Мы знаем, что в C++ существует три цикла - **for**, **while** и **do..while**.

Первый цикл называют циклом-счетчиком. Его удобно использовать, когда мы знаем, сколько итераций (повторений) должно быть. В нашем случае удобно использовать именно его. Общий синтаксис выглядит так:

```
for (оператор_1; условие; оператор_2)
{
  тело_цикла
}
```

Здесь оператор_1 выполняется в самом начале и один раз, перед первым выполнения тела цикла. Его удобно использовать для инициализации переменной-счетчика. Оператор 2 выполняется после каждой итерации, в нем

удобно увеличивать счетчик. Наш цикл будет выполнять операции в теле цикла до тех пор, пока истинно условие, заданное при объявлении цикла.

Код программы, использующий цикл **for**, приведен в листинге 13а.

Листинг 13а. 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).

Как уже было отмечено, поскольку мы знаем количество итераций (**n**), нашу задачу удобнее всего решить именно с циклом **for**. Однако ее можно решить и с помощью других циклов. Цикл **while()** выполняется, пока истинно его условие:

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


Такой цикл удобно использовать, когда мы не знаем заранее количество итераций, например:

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

int main()
{
  int k = 0, i = 0;

  while (k < 7) {
    k = 1 + rand() % 10;
    cout << k << " ";
    i++;
  }

  cout << "Итераций " << i << endl;

  return 0;
}
```

Программа вызывает функцию **rand()**, которая возвращает случайное число до 10. В цикле **while** у нас условие: работать, пока k < 7, если полученное значение будет больше 7, например 8, то цикл прекратит свою работу. Программа выведет счетчик итераций - сугубо для информации и все сгенерированные случайным образом числа. Например:

```
4 7 Итераций 2
```

Что же касается нашего примера, то решение задачи с помощью цикла **while** приведено в листинге 13б.

Листинг 13б. 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.

Цикл **do..while** в отличие от цикла **while** сначала выполняет тело цикла, а затем уже проверяет условие. Его удобно использовать, когда нужно, чтобы тело цикла выполнилось хотя бы один раз. Давайте переделаем программу из листинга 13a так, чтобы она запрашивала у пользователя ввод, пока он не введет целое положительное число. Далее вычислим сумму с помощью цикла **for**. Код программы в листинге 13c.

Листинг 13c. 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;
}
```

Пример 14. Вычисление факториала

Факториал положительного целого числа n - это результат умножения чисел от **1** до **n**:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Обычно на примере вычисления факториала демонстрируют рекурсию - событие, когда функция вызывает саму себя. Листинг 14 содержит так называемое нерекурсивное решение, когда факториал вычисляется с помощью цикла **for**, а не рекурсии.

Листинг 14. Вычисление факториала (14.cpp)

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

int main()
{
    unsigned int n;
    unsigned long long factorial = 1;

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

```
    cin >> n;

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

    cout << "Факториал " << n << " = " << factorial << endl;
    return 0;
}
```

Результат выполнения программы приведен на рис. 14. Поскольку факториал - это таки операция умножения, очень легко выйти за пределы диапазона. Для переменной **factorial** мы использовали тип **unsigned long long**, чтобы сделать максимальное значение максимально большим. Но и это не позволяет вычислить факториал даже 66. Максимальное значение, которое помещается в **unsigned long long** - это 65! В случае переполнения вы получите результат **0**.

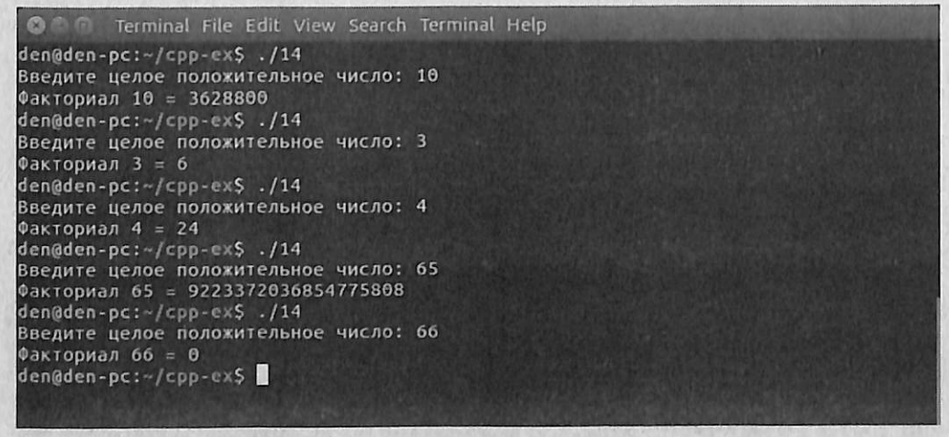


Рис. 14. Результат выполнения программы

Пример 15. Выводим таблицу умножения

Напишем простенькую программку, выводящую таблицу умножения заданного пользователем числа. Логика программы проста: пользователь вводит число n и мы его последовательно (в цикле **for**) умножаем на числа от 1 до 10 и выводим результат.

Код может быть примерно таким:


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

    for(i=1; i<=10; ++i)
    {
        cout << n << " * " << i << " = " << n * I;
    }
```

Однако, если честно, такая табличка смотрится уж очень просто для книги примеров. Давайте выведем полноценную таблицу умножения для чисел от 1 до 9. Для ее построения мы будем использовать два цикла: первый будет внешний и будет считать строки - от 1 до 9. Счетчиком этого цикла будет переменная **i**. Второй будет внутренним - он будет выполняться внутри первого цикла **for**. Его счетчик - это переменная **j**. Этот цикл будет умножать **i** * **j** и выводить результат. На C код может выглядеть так:

```
for (i = 1; i <= 9; i++) {
   for (j = 1; j <= 9; j++)
    printf("%2d  ", i*j);
   printf("\n");
}
```

В обычном языке C для форматирования вывода использовалась функция **printf()**. Теперь ее нет. Вместо нее нужно использовать оператор <<, но вот проблема: наша табличка умножения выглядит явно непрезентабельно. Чтобы наша таблица умножения выглядела прилично, нам нужно установить ширину стандартного вывода с помощью **cout.width()**. Код программы на C++ приведен в листинге 15.

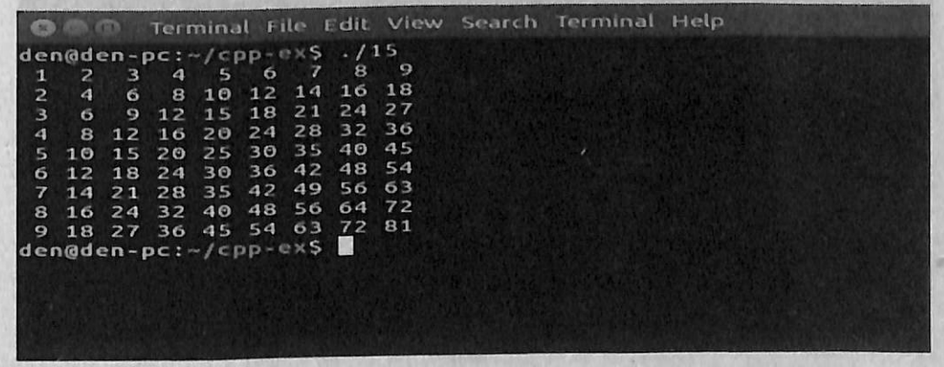


Рис. 15. Табличка умножения создана

Листинг 15. Полноценная таблица умножения

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

int main()
{
  int i, j;


  for (i = 1; i <= 9; i++) {
     for (j = 1; j <= 9; j++) {
      cout.width(2);          // ширина 2 символа
      cout << i*j << " ";
    }
     cout << "\n";
  }

  return 0;
}
```

Пример 16. Выводим последовательность Фибоначчи

Числа Фибоначчи — элементы последовательности в которой каждое последующее число равно сумме двух предыдущих чисел. Напишем программу, выводящую последовательность Фибоначчи. В этой программе мы используем две переменных **t1** и **t2** для хранения двух предыдущих чисел.

Листинг 16. Вычисление последовательности Фибоначчи (16.cpp)

```
#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;
}
```

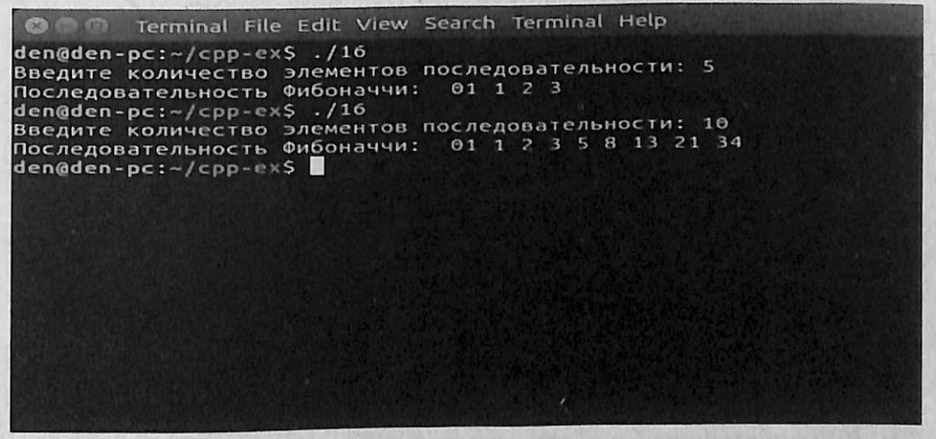


Рис. 16. Генерирование последовательности Фибоначчи

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
```

Пример 17. Вычисления НОД двух чисел

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 17а.

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./17
Введите два числа: 9 3
НОД = 3
den@den-pc:~/cpp-ex$ ./17
Введите два числа: 100 4
НОД = 4
den@den-pc:~/cpp-ex$ ./17
Введите два числа: 100 20
НОД = 20
den@den-pc:~/cpp-ex$
```

Рис. 17. Вычисление НОД

Листинг 17а. 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. 17б).

Листинг 17б. 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;
```

Пример 18. Наименьшее общее кратное

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

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

в предыдущем примере, есть несколько способов вычислить НОК. Первый способ решения - с помощью деления. Этот способ приведен в листинге 18а. Для разнообразия напишем этот пример на C++.

Листинг 18а. 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 "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**.

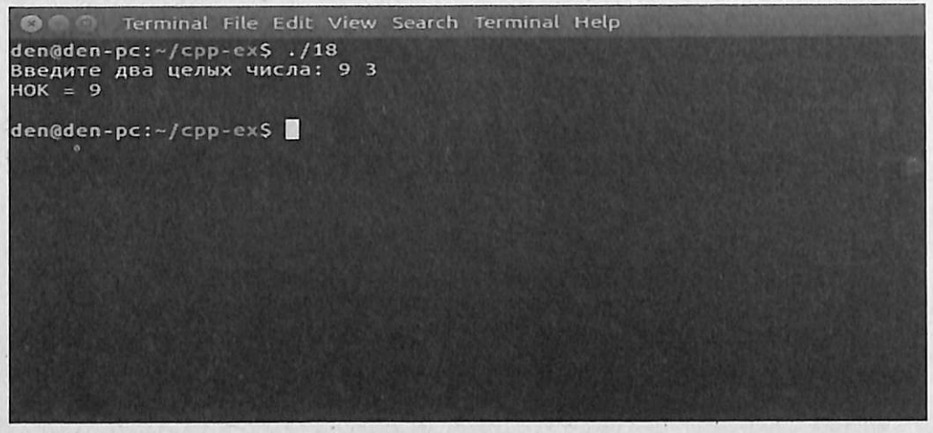


Рис. 18. Вычисление НОК двух чисел

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 18б.

Листинг 18б. 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 19. Подсчитываем количество 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 19. Определяем количество цифр в целом числе (19.cpp)

```
#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. 19. Определяем количество цифр

Пример 20. Вычисляем обратное число

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 20. 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;
}
```

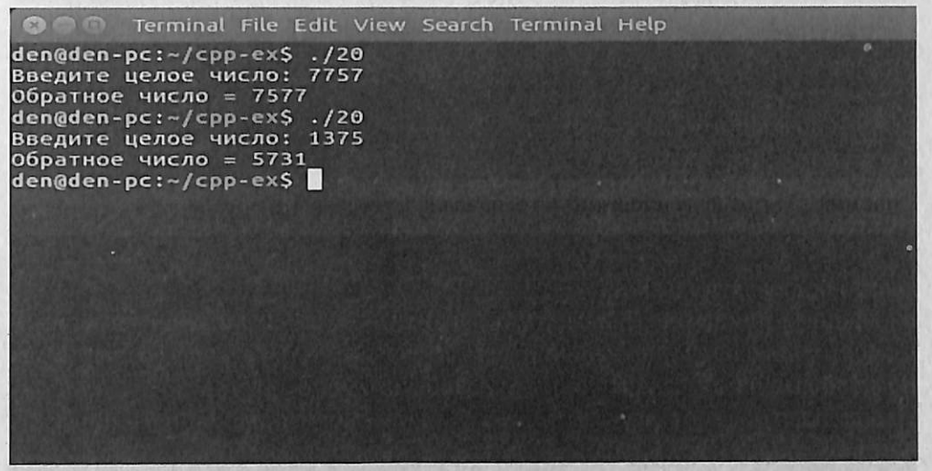


Рис. 20. Вычисляем обратное число

Пример 21. Вычисляем степень числа

Напишем программу, возводящую число в заданную степень. Для вычисления степени принято использовать функцию **pow()**, находящуюся в **math.h**. Однако если вам не хочется подключать всю математическую библиотеку из-за одной функции, то вам пригодится этот пример.

Листинг 21. Вычисления степени числа без функции pow()

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

int main()
{
    int exponent;
```

```
    float base, result = 1;

    cout << "Введите число и степень соответственно:  ";
    cin >> base >> exponent;

    cout << base << "^" << exponent << " = ";

    while (exponent != 0) {
        result *= base;
        --exponent;
    }

    cout << result << endl;

    return 0;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./21
Введите число и степень соответственно:  2 8
2^8 = 256
den@den-pc:~/cpp-ex$ ./21
Введите число и степень соответственно:  2 16
2^16 = 65536
den@den-pc:~/cpp-ex$
```

Рис. 21. Вычисляем степень числа

Если бы мы использовали функцию **pow()**, то все вычисление реализовалось бы одной строчкой:

```
result = pow(base, exponent);
```

Только не забудьте подключить **math.h** и указать опцию **-lm** при компиляции программы.

Пример 22. Проверяем, является ли число палиндромом или нет

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 22. 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;
}
```

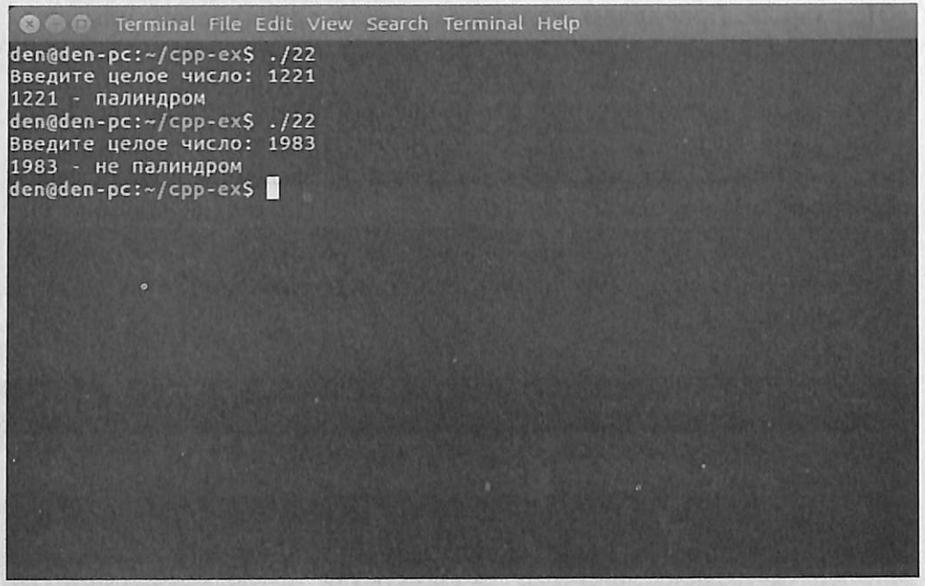


Рис. 22. Результат работы программы

Пример 23. Является ли число простым

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 23.

Листинг 23. Является ли число простым (23.cpp)

```
#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;
}
```

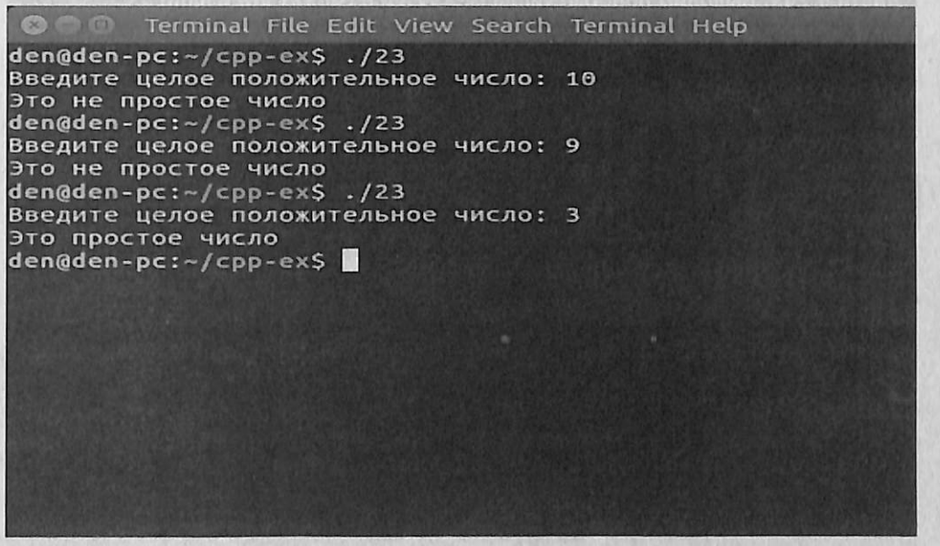


Рис. 23. Определяем простые числа

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**.

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

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

        ++low;
    }

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

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./24
Выводим простые числа между a и b, введите a и b: 1 50
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
den@den-pc:~/cpp-ex$
```

Рис. 24. Выводим простые числа между a и b

Пример 25. Проверяем число Армстронга

Самовлюблённое число, или совершенный цифровой инвариант (англ. pluperfect digital invariant, PPDI), или число Армстронга — натуральное

число, которое в данной системе счисления равно сумме своих цифр, возведённых в степень, равную количеству его цифр.

Пример числа Армстронга:

```
1634 = 1^4 + 6^4 + 3^4 + 4^4
153 = 1^3 + 5^3 + 3^3
```

Для решения этой задачи нам нужно определить количество разрядов в числе. Именно для этого мы и писали программу в одном из предыдущих примеров. Ведь нам нужно знать, в какую степень возводить составляющие числа.

Код программы приведен в листинге 25.

Листинг 25. Является ли число числом Армстронга (25.cpp)

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

int main()
{
  int origNum, num, rem, digit, sum = 0;
  cout << "Введите целое положительное число: ";
  cin >> origNum;

  num = origNum;

  while(num != 0)
  {
      digit = num % 10;
      sum += digit * digit * digit;
      num /= 10;
  }

  if(sum == origNum)
    cout << origNum << " - число Армстронга." << endl;
  else
    cout << origNum << " - не является числом Армстронга." <<
endl;

  return 0;
}
```

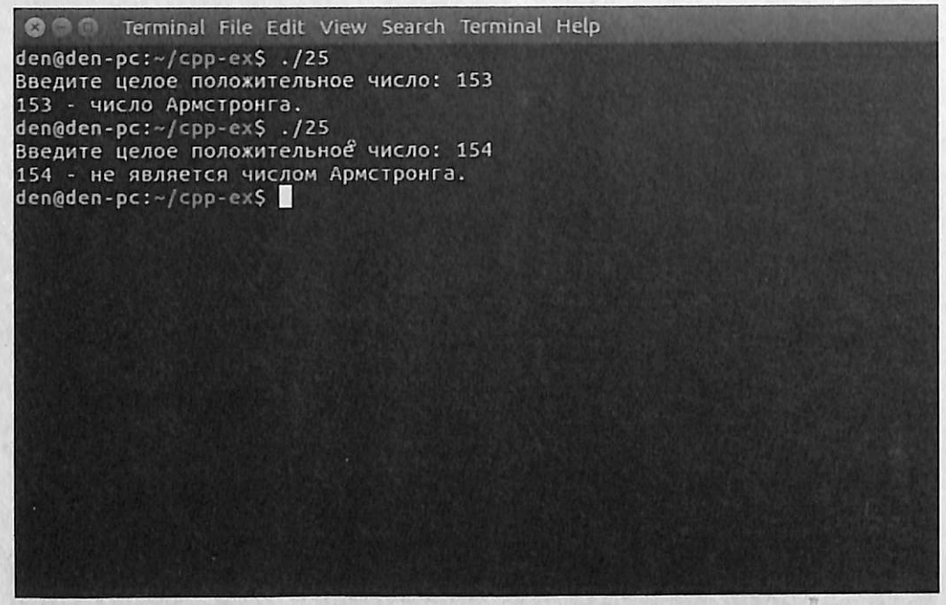


Рис. 25. Проверка на число Армстронга

Пример 26. Выводим числа Армстронга в заданном диапазоне

Ранее мы писали программу, выводящую простые числа, находящиеся в заданном диапазоне. Напишем аналогичную программу, только выводящую числа Армстронга, если таковые имеются. Работает программа просто: в цикле **for** перебираются числа заданного диапазона и каждое проверяется, является ли оно числом Армстронга.

Листинг 26. Выводим числа Армстронга в заданном диапазоне

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

int main()
{
    int low, high, i, temp1, temp2, remainder, n = 0, result =
0;
```



```
    cout << "Введите начало и конец диапазона: ";
    cin >> low >> high;
    cout << "Числа Армстронга: ";

    for(i = low + 1; i < high; ++i)
    {
        temp2 = i;
        temp1 = i;

        // к-во разрядов
        while (temp1 != 0)
        {
            temp1 /= 10;
            ++n;
        }

        // результат содержит сумму цифр, возведенных в
степень n
        while (temp2 != 0)
        {
            remainder = temp2 % 10;
            result += pow(remainder, n);
            temp2 /= 10;
        }

        // проверяем, является ли число суммой n степени
своих цифр
        if (result == i) {
            cout << i << " ";
        }

        // сброс значений перед след. итерацией
        n = 0;
        result = 0;
    }

    cout << "\n";
    return 0;
}
```

Для компиляции программы введите команду:

```
g++ 26.cpp -o 26 -lm
```

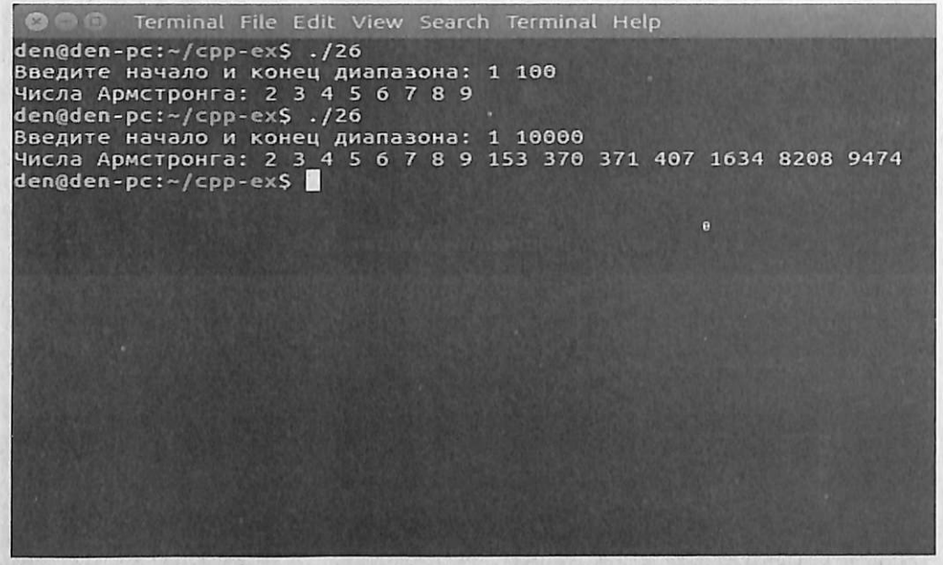


Рис. 26. Числа Армстронга в диапазоне от 1 до 1000

Пример 27. Создаем пирамиду и структуру

Чтобы отдохнуть от математики (если, конечно, вы выполняете примеры последовательно), 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 27. 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;
}
```

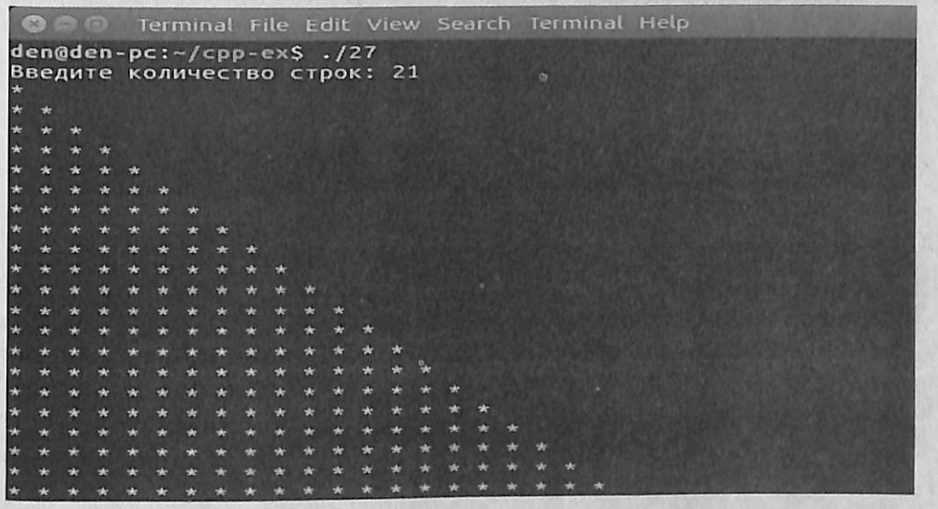


Рис. 27. Пирамида из звездочек

Пример 28. Делаем простой калькулятор с использованием switch.. case

Давайте напишем простенький калькулятор, демонстрирующий пример использования оператора **switch .. case**.

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 28. 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;
}
```

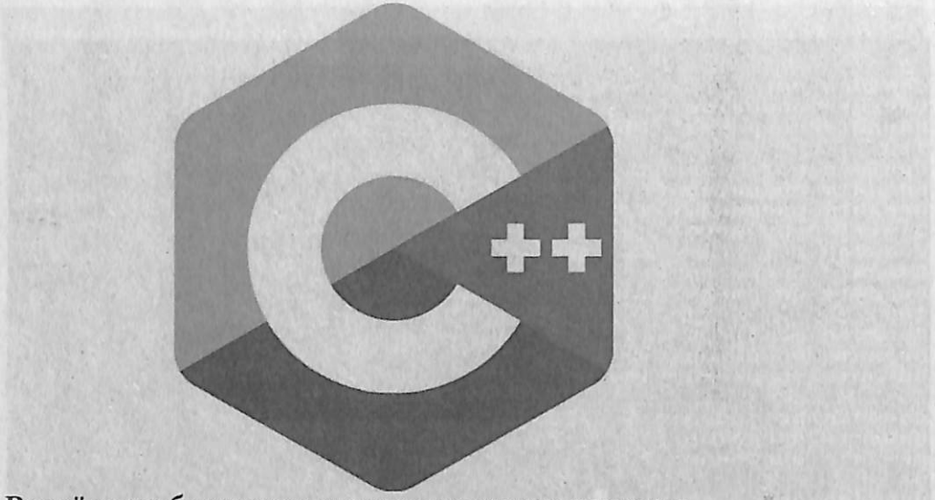


```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) +
Введите операнды: 100 20
120
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) /
Введите операнды: 9 0
Divizion by zero!
den@den-pc:~/cpp-ex$ ./28
Введите оператор (+ - * /) *
Введите операнды: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

На рис. 28 показано тестирование нашего калькулятора. Продемонстрировано, как программа реагирует на деление на **0**, на ввод некорректного оператора.

Часть 3. Пользовательские функции



В этой части будут рассмотрены следующие примеры:

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.

Пример 29. Собственная функция

Прежде, чем перейти к реальным программам, немного теории. Что такое функция? 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, то проще исправить ее один раз - в теле функции, чем искать, где он использовал ошибочный код по всей программе.

В наших небольших программах преимущества использования функций не совсем очевидно, но когда вы станете успешным программистом, вы не сможете без них обходиться - уж поверьте.

Определить функцию можно с помощью ключевого слова **function**:

```
<тип> function имя (параметры) {
   тело;
}
```

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 separator() {
  printf("**********************************");
}
```

В коде программы гораздо правильнее вызывать функцию **separator()** для разделения блоков выводимого текста, чем просто **printf()**. Если вам захочется со временем изменить символ разделителя и вместо * выводить, скажем, ~, то для этого нужно будет изменить код одной функции, а не править код всей программы и искать, где еще вы вызывали тот злосчастный **printf()**.

Пример 30. Проверка на простое число с помощью пользовательской функции

В предыдущей части книги мы писали программу, определяющую, является ли введенное пользователем число простым. Данная часть книги посвящена функциям, поэтому сейчас мы напишем функцию, определяющую это и покажем, как ее использовать в нашей программе.

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 30.

Листинг 30. Проверка на простое число

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

int checkPrimeNumber(int);

int main()
{
  int n;
  cout << "Введите положительное число: ";
  cin >> n;

  if(checkPrimeNumber(n) == true)
    cout << n << " - это простое число";
  else
    cout << n << " - не простое число";

  cout << endl;
  return 0;
}

int checkPrimeNumber(int n)
{
  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. 30.

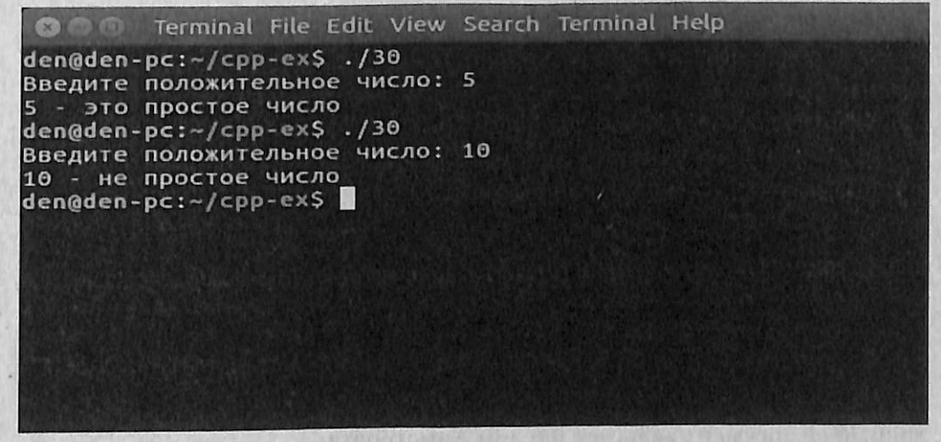


Рис. 30. Первая программа с использованием функции

Пример 31. Проверка на число Армстронга с использованием функции

На этот раз напишем программу, проверяющую, является ли введенное число числом Армстронга. Для этого будем использовать следующую функцию:

Листинг 31. Проверка на число Армстронга

```
int checkArmstrongNumber(int number)
{
    int originalNumber, remainder, result = 0, n = 0, flag;

    originalNumber = number;

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

    originalNumber = number;

    while (originalNumber != 0)
    {
        remainder = originalNumber%10;
        result += pow(remainder, n);
        originalNumber /= 10;
    }

    // Условие для числа Армстронга
    if(result == number)
        flag = 1;
    else
        flag = 0;

    return flag;
}
```

Когда программы будет аналогичен примеру 30, поэтому написать всю программу вам предстоит самостоятельно - в качестве домашнего задания. Самое сложное у вас есть - это функция, которая возвращает **true**, если переданный ей аргумент является числом Армстронга.

Пример 32. Выводим все простые числа в заданном диапазоне

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 32. 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;
}
```

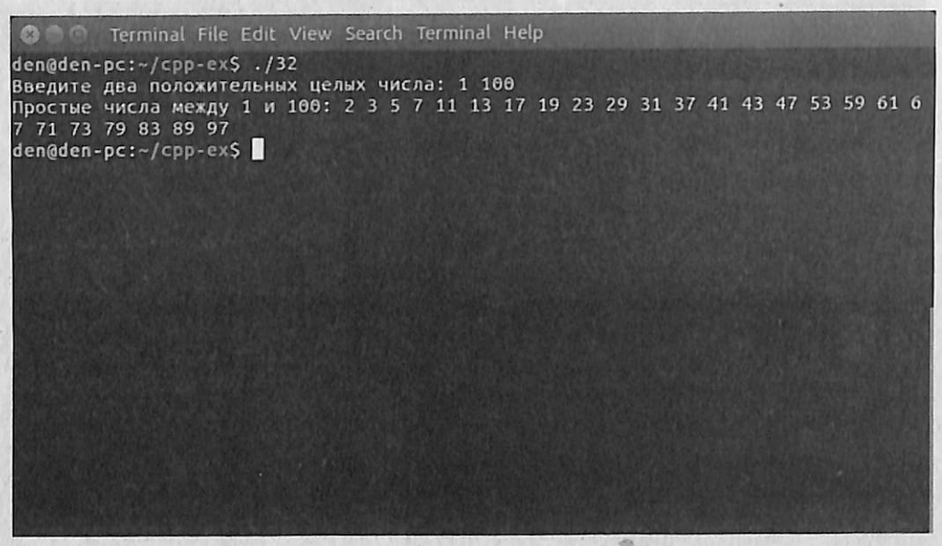


Рис. 32. Вывод простых чисел в заданном диапазоне

Пример 33. Проверяем, может ли число быть выраженным как сумма двух простых чисел

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 33. Выражаем число как сумму двух простых чисел (33.cpp)

```
#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 1;
```

Результат 1 * 2 * 3 * 4 * 5 = 120

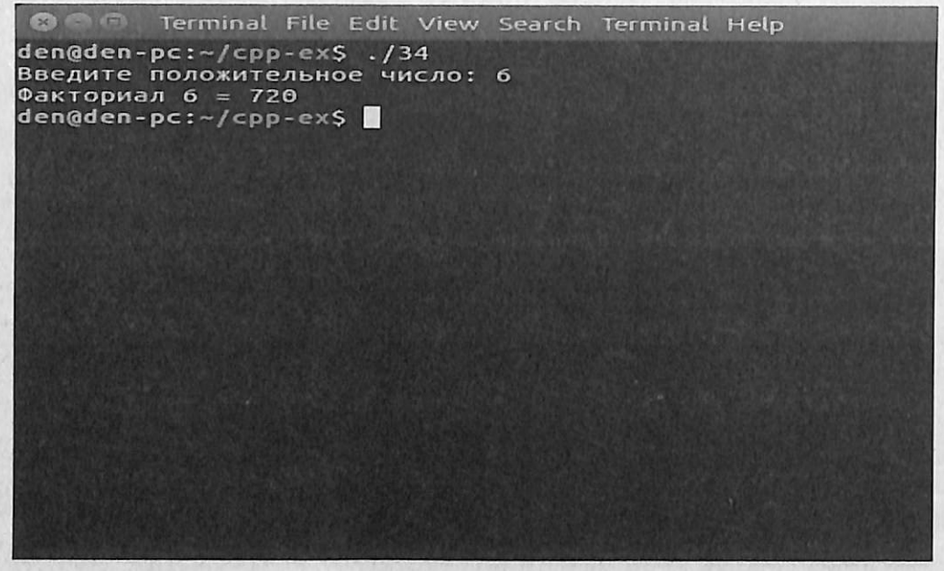


Рис. 34. Вычисление факториала

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.

Пример 35. Сумма n натуральных чисел с использованием рекурсии

Рассмотрим еще один рекурсивный алгоритм. На этот раз мы вычислим сумму **n** натуральных чисел с использованием рекурсии (лист. 35).

Листинг 35. 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. 35.

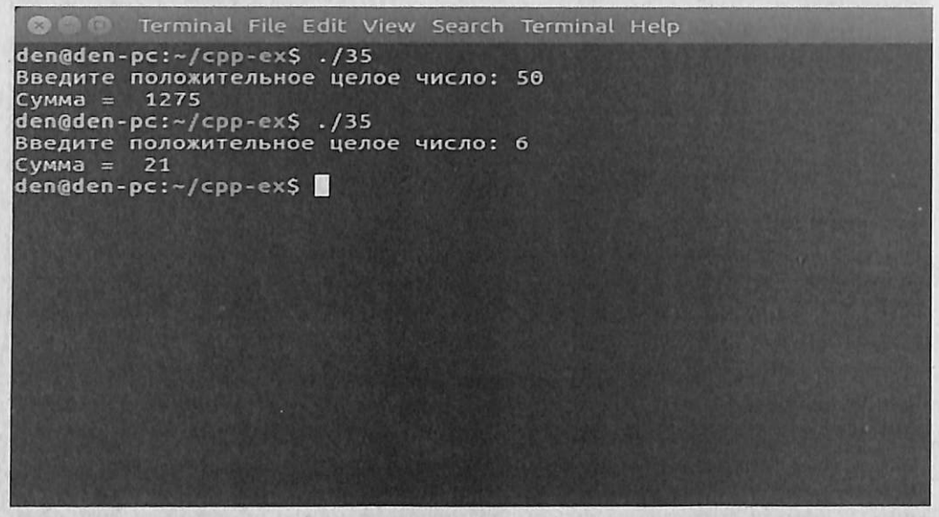


Рис. 35. Результат работы программы

Пример 36. Вычисляем НОД с использованием рекурсии

На этот раз рекурсию мы будем использовать для вычисления наибольшего общего делителя. Здесь мы используем рекурсивную функцию **gcd()**, условие выхода - n2 = 0.

Листинг 36. Вычисление НОД с помощью рекурсии (36.cpp)

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



```
int gcd(int n1, int n2);

int main()
{
   int n1, n2;

   cout << "Введите 2 целых положительных числа: ";
   cin >> n1 >> n2;

   cout << "НОД " << n1 << " и " <<  n2 << " = " << gcd(n1,
n2) << endl;

   return 0;
}

int gcd(int n1, int n2)
{
    if (n2 != 0)
       return gcd(n2, n1 % n2);
    else
       return n1;
}
```

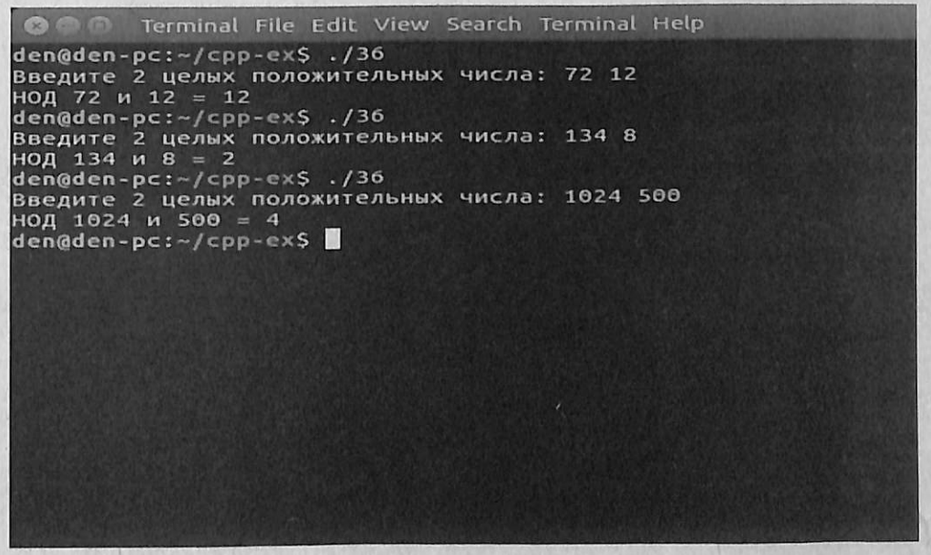


Рис. 36. НОД с использованием рекурсии

Пример 37. Конвертируем двоичные числа в десятичные и наоборот

Компьютер "думает" на языке двоичной системы счисления. Да, даже самый современный. Он оперирует только двумя числами - 0 и 1. Все, что есть в компьютере - от простого текста, фотографии до программы, видеролика, музыкального файла - можно представить как набор нулей и единиц.

Давайте же напишем программу, позволяющую переводить числа в привычной нам, десятичной, системе счисления в двоичную. Для преобразования двоичного числа в десятичное мы будем использовать функцию **convertBinaryToDecimal()**. Конвертирование осуществляется последовательным делением на 10, при этом остаток от деления умножается на степень двойки. Думаю, вы знаете, как переводить двоичные числа в десятичные со школьного курса информатики.

Конвертирование из десятичной в двоичную осуществляется функцией **convertDecimailTo Binary()**. Данная функция выполняет последовательное деление на 2. Мы вычисляем остаток от деления, вычисляем частное и формируем двоичное число. Чтобы было нагляднее, функция выводит каждый свой шаг и вы сможете увидеть, как формируется двоичное число. Полный код программы приведен в листинге 37.

Листинг 37. Преобразование из двоичной системы в десятичную и наоборот (37.cpp)

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

int convertBinaryToDecimal(long long n);
long long convertDecimalToBinary(int n);

int main()
{
    long long n;
    cout << "Введите двоичное число: ";
    cin >> n;
    cout << n << " (binary) = " << convertBinaryToDecimal(n)
<< " (decimal)\n";

    cout << "Введите десятичное число: ";
    cin >> n;
    cout << n << " (decimal) = " << convertDecimalToBinary(n)
<< " (binary)\n";
    return 0;
}

int convertBinaryToDecimal(long long n)
{
    int decimalNumber = 0, i = 0, remainder;
    while (n!=0)
    {
        remainder = n%10;
        n /= 10;
        decimalNumber += remainder*pow(2,i);
        ++i;
    }
    return decimalNumber;
}

long long convertDecimalToBinary(int n)
{
    long long binaryNumber = 0;
    int remainder, i = 1, step = 1;

    while (n!=0)
    {
        remainder = n%2;
        cout << "Шаг: " << step++ << " " << n << "/2, Остаток
= " << remainder << ", Quotient = " << n/2 << endl;
        n /= 2;
        binaryNumber += remainder*i;
        i *= 10;
    }
    return binaryNumber;
}
```

Для компиляции программы нужно использовать опцию **-lm**:

```
g++ 37.cpp -o 37 -lm
```

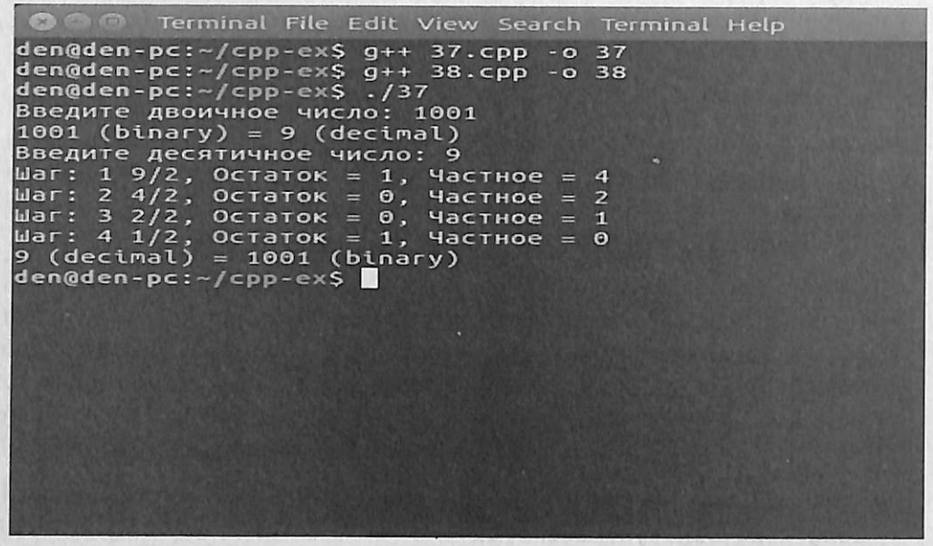


Рис. 37. Преобразование между системами счисления

Пример 38. Конвертируем восьмеричные числа в десятичные и наоборот

Кроме двоичной и десятичной системы счисления в информатике частенько используется и восьмеричная система. Для представления чисел в ней используются цифры от 0 до 7. Восьмеричная система чаще всего используется в областях, связанных с цифровыми устройствами.

Перевод в восьмеричную систему из десятичной осуществляется последовательным делением на 8. Обратное преобразование осуществляется путем умножения остатка от деления на 10 на pow(8, i). При этом 8 - это основа системы счисления, а **i** - счетчик.

Листинг 38. Конвертируем восьмеричные числа в десятичные и наоборот (38.cpp)

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



```
int octalToDecimal(int octalNumber);
int decimalToOctal(int decimalNumber);

int main()
{
   int octalNumber, decimalNumber;
   cout << "Введите восмеричное число: ";
   cin >> octalNumber;
   cout << octalNumber << " (octal) = " <<
octalToDecimal(octalNumber) << " (decimal) " << endl;

   cout << "Введите десятичное: ";
   cin >> decimalNumber;
   cout << decimalNumber << " (decimal) = " <<
decimalToOctal(decimalNumber) << " (octal)" << endl;

   return 0;
}

// Преобразуем 8-ричную систему в 10-ную
int octalToDecimal(int octalNumber)
{
    int decimalNumber = 0, i = 0, rem;
    while (octalNumber != 0)
    {
        rem = octalNumber % 10;
        octalNumber /= 10;
        decimalNumber += rem * pow(8, i);
        ++i;
    }
    return decimalNumber;
}

// преобразуем 10-ную систему в 8-ную
int decimalToOctal(int decimalNumber)
{
    int rem, i = 1, octalNumber = 0;
    while (decimalNumber != 0)
    {
        rem = decimalNumber % 8;
        decimalNumber /= 8;
        octalNumber += rem * i;
        i *= 10;
    }
    return octalNumber;
}
```

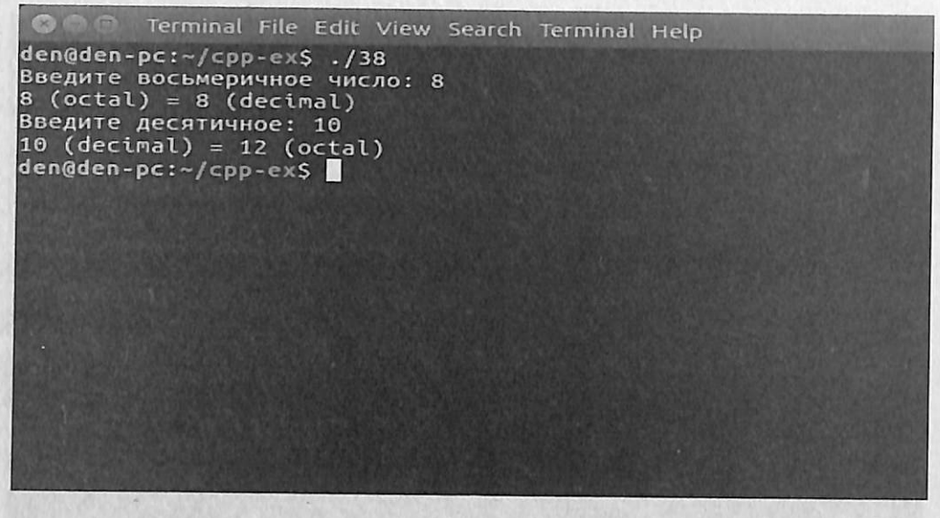


Рис. 38. Демонстрация работы программ из лист. 38

Пример 39. Конвертируем двоичные числа в восьмеричные и наоборот

Этот пример содержит код программы, конвертирующий двоичные числа в восьмеричные и наоборот (лист. 39). Результат выполнения программы приведен на рис. 39.

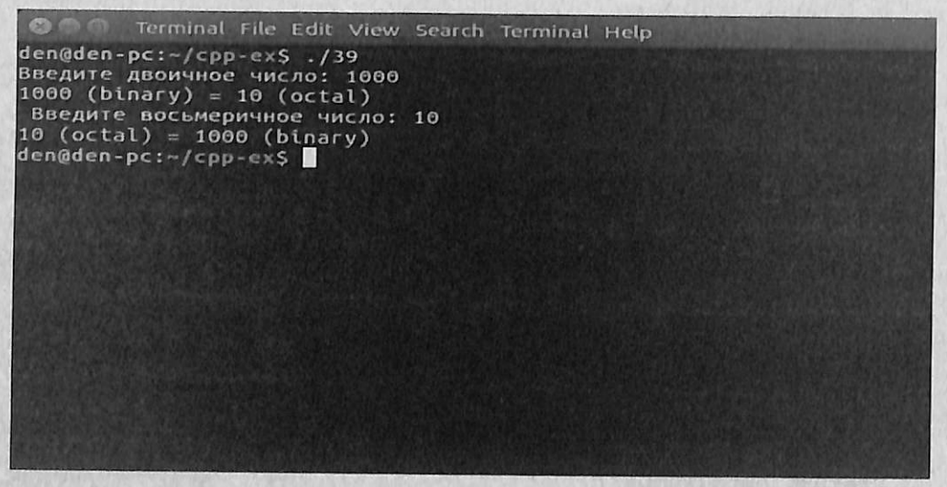


Рис. 39. Конвертирование двоичных чисел в восьмеричные и наоборот

Листинг 39. Конвертируем двоичные числа в восьмеричные и наоборот (39.cpp)

```
#include <iostream>
#include <cmath>

using namespace std;

// преобразуем двоичное число в восьмеричное
int convertBinarytoOctal(long long);
// преобразуем восьмеричное число в двоичное
long long convertOctalToBinary(int);

int main()
{
    long long binaryNumber;
    int octalNumber;

    cout << "Введите двоичное число: ";
    cin >> binaryNumber;

    cout << binaryNumber << " (binary) = " << convertBinarytoO
ctal(binaryNumber) << " (octal)\n ";

    cout << "Введите восьмеричное число: ";
    cin >> octalNumber;

    cout << octalNumber << " (octal) = " << convertOctalToBina
ry(octalNumber) << " (binary)\n";

    return 0;
}

int convertBinarytoOctal(long long binaryNumber)
{
    int octalNumber = 0, decimalNumber = 0, i = 0;

    while(binaryNumber != 0)
    {
        decimalNumber += (binaryNumber%10) * pow(2,i);
        ++i;
        binaryNumber/=10;
    }

    i = 1;
```

```
    while (decimalNumber != 0)
    {
        octalNumber += (decimalNumber % 8) * i;
        decimalNumber /= 8;
        i *= 10;
    }

    return octalNumber;
}

long long convertOctalToBinary(int octalNumber)
{
    int decimalNumber = 0, i = 0;
    long long binaryNumber = 0;

    while(octalNumber != 0)
    {
        decimalNumber += (octalNumber%10) * pow(8,i);
        ++i;
        octalNumber/=10;
    }

    i = 1;

    while (decimalNumber != 0)
    {
        binaryNumber += (decimalNumber % 2) * i;
        decimalNumber /= 2;
        i *= 10;
    }

    return binaryNumber;
}
```

Пример 40. Выводим предложение в обратном порядке

До этого мы работали только с числами. На этот раз мы напишем рекурсивную функцию **reverseSentence()**, которая будет выводить введенное пользователем предложение в обратном порядке.

Листинг 40. Вывод предложения в обратном порядке (40.cpp)

```
#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));
    }
}
```

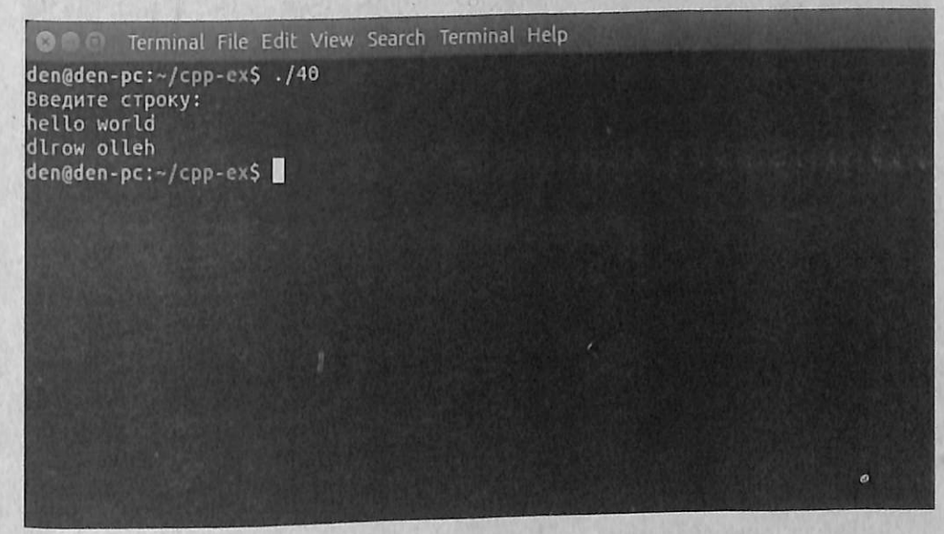


Рис. 40. Программа в действии

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.

Пример 41. Вычисляем степень с помощью рекурсии

В завершении этой части мы рассмотрим еще один пример с рекурсией. На этот раз мы попытаемся рекурсивно вычислить степень числа. Учитывая особенность рекурсии для решения нашей задачи нужно вызвать функцию calculatePower() на один раз больше, чтобы получить правильный результат.

Листинг 41. Рекурсивное вычисление степени числа

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

int calculatePower(int, int);

int main()
{
    int base, powerRaised, result;

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

    cout << "Введите степень: ";
    cin >> powerRaised;

    result = calculatePower(base, powerRaised+1);
```



```
    cout << base << "^" << powerRaised << " = " << result <<
endl;

    return 0;
}

int calculatePower(int base, int powerRaised)
{
    if (powerRaised != 1)
        return (base*calculatePower(base, powerRaised-1));
    else
        return 1;
}
```

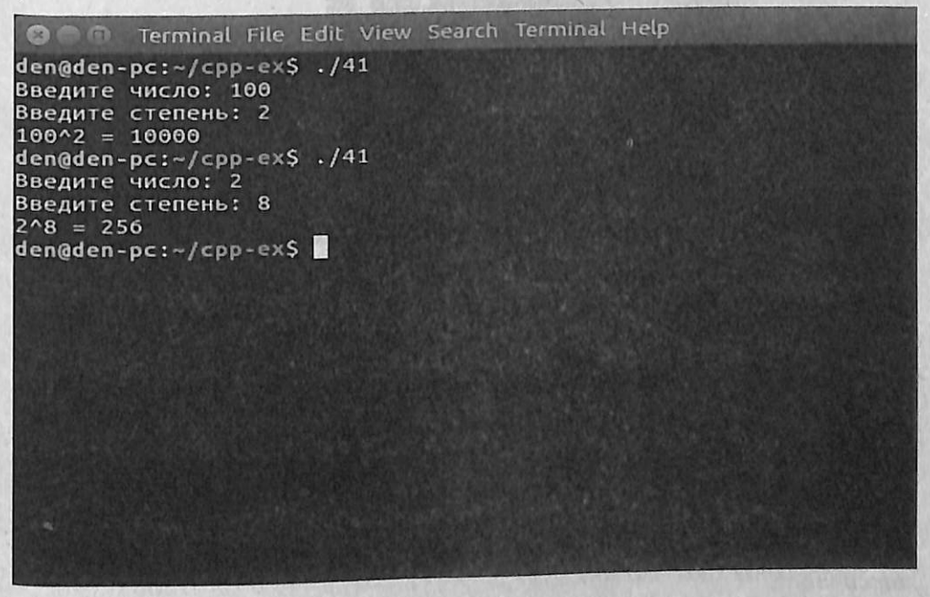
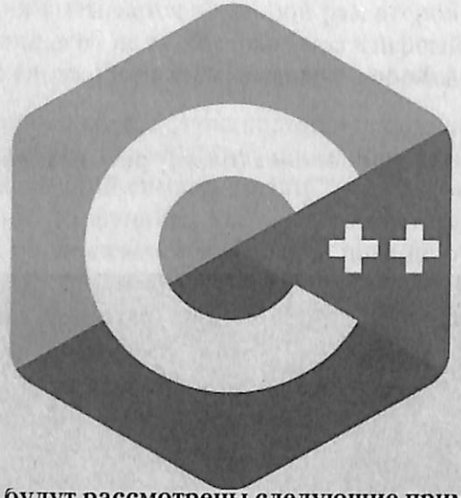


Рис. 41. Программ в действии

Часть 4. Массивы и указатели



В этой части будут рассмотрены следующие примеры:

Пример 42. Вычисляем среднее с использованием массивов

Пример 43. Вычисляем наибольший элемент массива

Пример 44. Вычисляем среднеквадратичное отклонение

Пример 45. Сложение двух матриц с использованием многомерных массивов

Пример 46. Умножение на матрицу с использованием многомерных массивов

Пример 47. Транспонированная матрица

Пример 48. Умножение двух матриц с передачей матрицы в функции

Пример 49. Доступ к элементам массива с использованием указателей

Пример 50. Своп числа в циклическом порядке с помощью вызова по ссылке

Пример 42. Вычисляем среднее с использованием массивов

Данная часть книги посвящена массивам и указателям. Начнем с массива. 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 с помощью функции **scanf()**
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 42. Вычисление среднего арифметического (42.cpp)

```
#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. 42. Результат вычисления среднего арифметического

Пример 43. Вычисляем наибольший элемент массива

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 43. 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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./43
Введите количество элементов (1-100): 5

Введите число 1 : 6
Введите число 2 : 7
Введите число 3 : 2
Введите число 4 : 9
Введите число 5 : 3
Максимум = 9
den@den-pc:~/cpp-ex$
```

Рис. 43. Поиск максимума в массиве

Пример 44. Вычисляем среднеквадратичное отклонение

Данный пример 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. 44). Результат работы программы, а также команда ее компиляции изображены на рис. 44.

Листинг 44. Передача массива функции (44.cpp)

```
#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);
}
```

Для компиляции этой программы не забудьте указать опцию **-lm**.

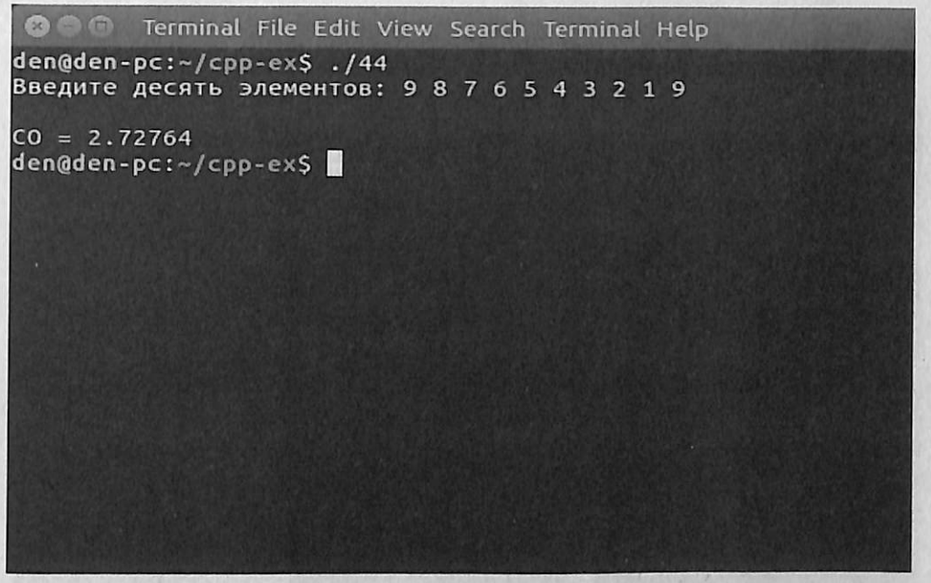


Рис. 44. Результат работы программы

Пример 45. Сложение двух матриц с использованием многомерных массивов

Довольно сложный пример - сложение двух матриц с использованием многомерных массивов. 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 45. 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. 45. Сложение двух матриц

Пример 46. Умножение на матрицу с использованием многомерных массивов

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.

На рис. 46 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:

$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} = 2 \cdot 3 + 2 \cdot 3 = 6 + 6 = 12$
$c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22} = 2 \cdot 3 + 2 \cdot 3 = 6 + 6 = 12$
$c_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} = 2 \cdot 3 + 2 \cdot 3 = 6 + 6 = 12$
$c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22} = 2 \cdot 3 + 2 \cdot 3 = 6 + 6 = 12$

```
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. 46. Результат умножения двух матриц

Листинг 46. Умножение матриц (46.cpp)

```
#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;
}
```

Пример 47. Транспонированная матрица

В этом 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. 47). Напомню, что транспонированная матрица — матрица, полученная из исходной матрицы заменой строк на столбцы. В листинге 47 приводится код программы, вычисляющей транспонированную матрицу. Результат выполнения программы изображен на рис. 47: обратите внимание на исходную матрицу и результат.

Листинг 47. 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;
}
```

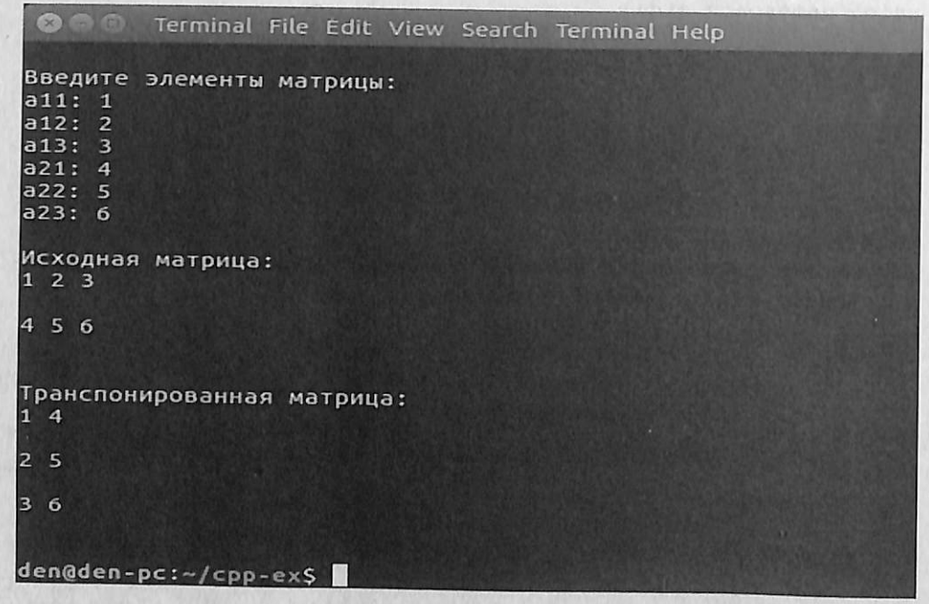


Рис. 47. Вычисление транспонированной матрицы

Пример 48. Умножение двух матриц с передачей матрицы в функции

Ранее (пример 46) 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 48. Умножение двух матриц с использованием функций (48.cpp)

```
#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. 48. Умножение 2 матриц с использованием функций

Пример 49. 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 49. Доступ к элементам массива с использованием указателей (49.cpp)

```
#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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./49
Введите элементы: 1 2 3 4 5
Вы ввели:
1
2
3
4
5
den@den-pc:~/cpp-ex$
```

Рис. 49. Результат работы программы

Пример 50. Своп числа в циклическом порядке с помощью вызова по ссылке

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 50. Своп чисел с помощью вызова по ссылке (50.cpp)

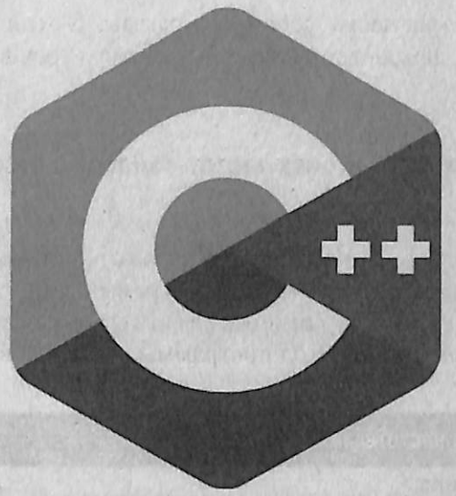
```
#include<iostream>
using namespace std;

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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./50
Введите значения a, b, c: 3 5 6
Значения перед заменой:
a, b, c соответственно: 3, 5, 6
После замены:
 a, b, c соответственно: 6, 3, 5
den@den-pc:~/cpp-ex$
```

Рис. 50. Результат замены местами чисел с помощью указателей

Часть 5. Работа со строками



В этой части будут рассмотрены следующие примеры:

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

Пример 51. Поиск частоты знаков в строке

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 51.

Листинг 51. 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;
}
```



```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./51
Символ l встречается 3 раз
den@den-pc:~/cpp-ex$
```

Рис. 51. Сколько раз встречается символ

Пример 52. Подсчет частоты символов в строке С-стиля

В языке С используется несколько иной стиль строки. Типа **string** нет, и строка представляет собой массив символов, то есть элементов типа char. Все строки в С заканчиваются знаком \0, означающим конец строки. Пример работы с такой строкой приведен в листинге 52.

Листинг 52. Посимвольный проход по строке в С-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;
}
```

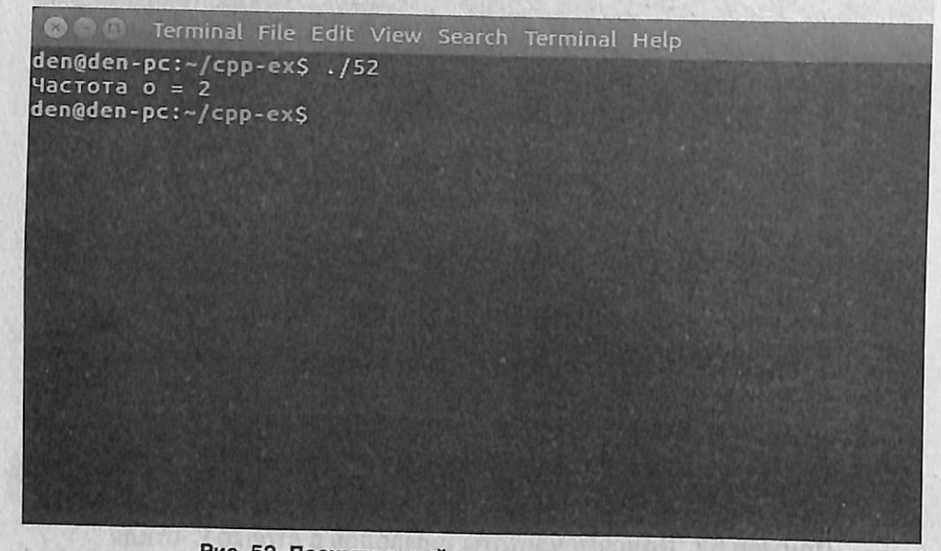


Рис. 52. Посимвольный проход по строке в С-стиле

Пример 53. Преобразование С-программы, обрабатывающую строку, в программу на C++

Мы только что рассмотрели два примера программ, выполняющих посимвольную обработку строки. Представим, что у нас есть аналогичная программа, написанная на С. Попробуем переписать ее на языке C++. Данное упражнение поможет перенести старые С-программы на более современный язык программирования.

Наша программа будет немного отличаться от примеров 51 и 52. Если в тех примерах мы использовали предопределенную строку, то в этом примере мы будем вводить строку с клавиатуры.

Программа просит пользователя ввести сначала строку, а затем символ, частоту которого нужно вычислить. Далее программа в цикле "проходится" по строке и подчитывает частоту символа. Код программы приведен в листинге 53а.

Листинг 53а. Поиск частоты знаков в строке (53.c)

```
#include <stdio.h>

int main()
{
   char str[1000], ch;
   int i, frequency = 0;

   printf("Введите строку: ");
   gets(str);

   printf("Введите символ: ");
   scanf("%c",&ch);

   for(i = 0; str[i] != '\0'; ++i)
   {
       if(ch == str[i])
           ++frequency;
   }

   printf("Частота = %d\n", frequency);

   return 0;
}
```

В этой программе мы получаем строку, введенную пользователем, в переменную **str**. Далее введенный пользователем символ мы сохраняем в переменной **ch**. После этого в цикле мы проходимся по всем символам строки **str**. Если текущий символ равен **ch**, мы увеличиваем переменную **frequency**, в которой мы и храним частоту знака. В конце программы выводится полученный результат.

Теперь перепишем программу на C++ (лист. 53б). Казалось бы, программа абсолютно такая, но, обратите внимание, как мы читаем строку и символ. На С мы используем функцию **gets()**, имя переменной, в которую будет записана прочитанная строка, указывается без каких-либо знаков:

```
gets(str);
```

В C++ для этого используется оператор:

```
cin >> str;
```

Для чтения символа мы использовали **scanf()**, указав модификатор чтения %c. Можно было бы использовать функцию **getchar()** - без разницы. Оператор **&** - это унарный оператор, возвращающий адрес операнда:

```
scanf("%c",&ch);
```

В C++ при использовании оператора >> нужно просто указать имя переменной. Оператор **&** указывать не нужно. В остальном программы похожи.

Листинг 53б. Подсчитываем частоту символа на C++

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

int main()
{
   char str[1000], ch;
   int i, frequency = 0;

   cout << "Введите строку: ";
   cin >> str;

   cout << "Введите символ: ";
   cin >> ch;

   for(i = 0; str[i] != '\0'; ++i)
   {
       if(ch == str[i])
           ++frequency;
   }

   cout << "Частота символа = " << frequency << "\n";

   return 0;
}
```

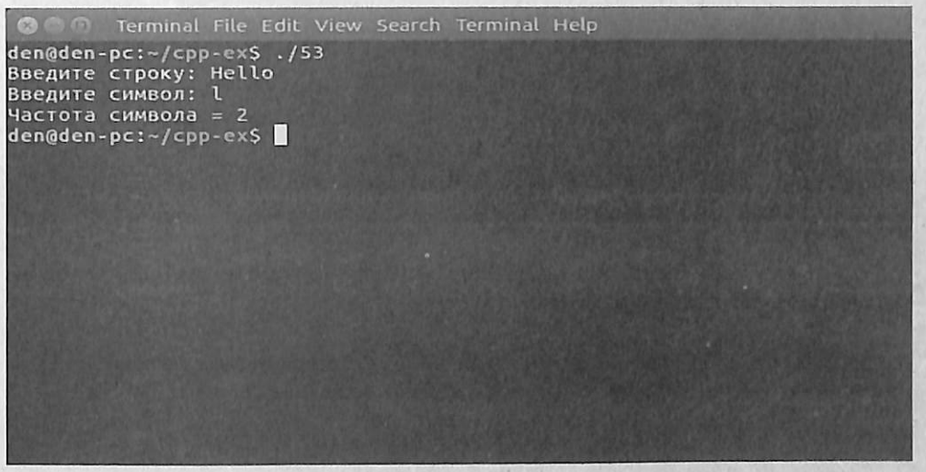



Рис. 53. Подсчет частоты символов (на C и C++)

Пример 54. Программа для подсчета 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 54.

Листинг 54. 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.

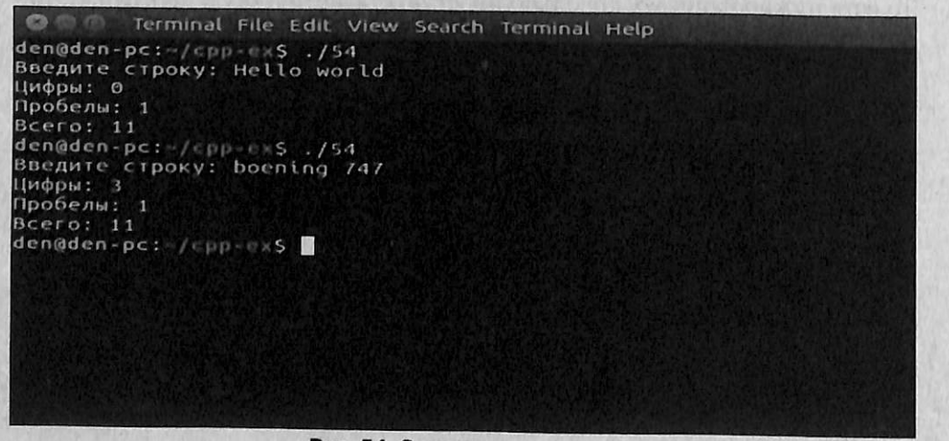


Рис. 54. Статистика о строке

Пример 55. 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 55.

Листинг 55. Удаляем все символы в строке, кроме цифровых (55.cpp)

```
#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;
}
```

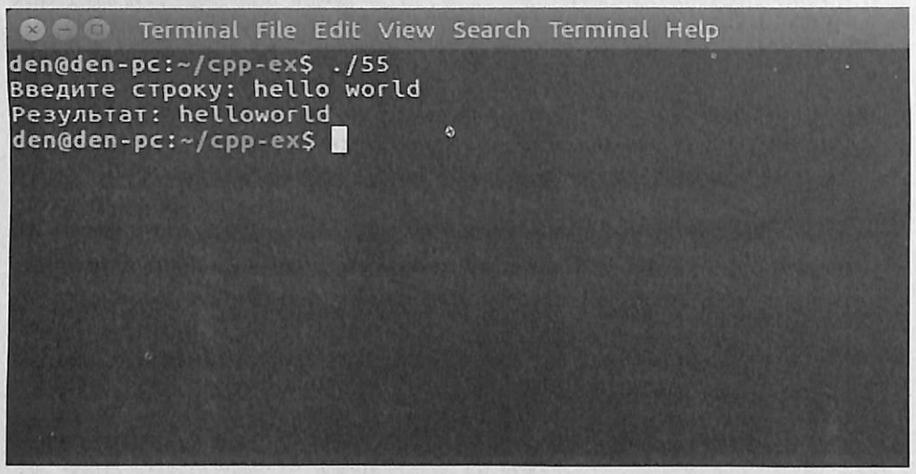


Рис. 55. Удаляем все символы, кроме цифровых

Пример 56. 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. 56).

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 56. 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;
}
```

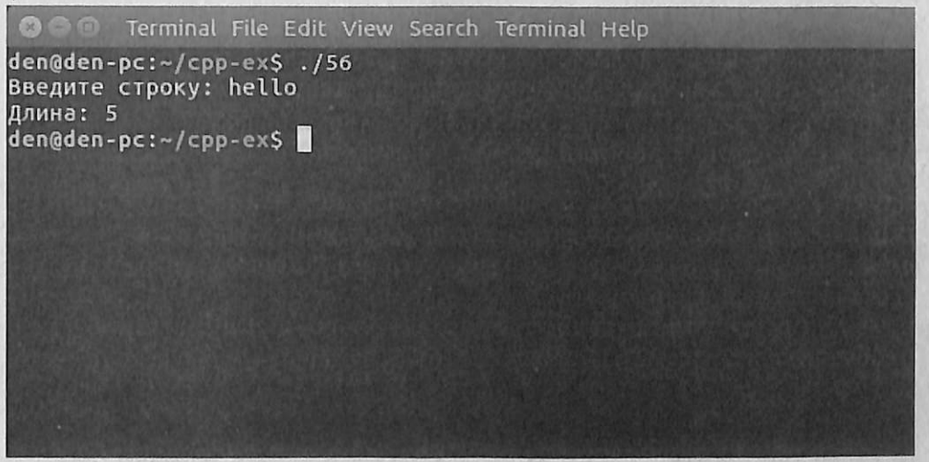


Рис. 56

Пример 57. Конкатенация двух строк

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 57. Конкатенация двух строк без функции strcat() (57.cpp)

```
#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.

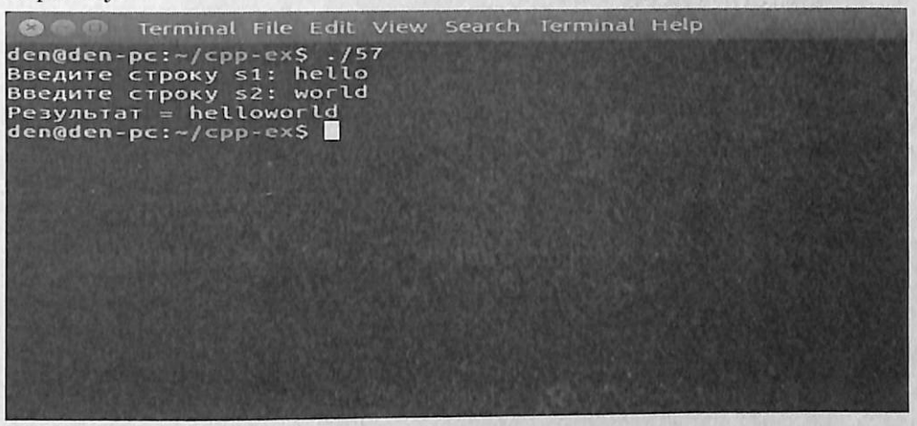


Рис. 57

Пример 58. 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. 58а и лист. 58б).

Листинг 58а. 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 58б. 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';
```

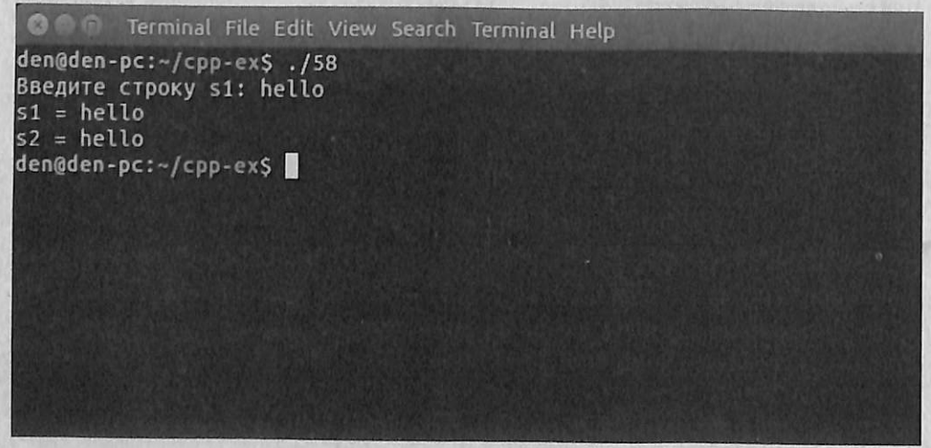


Рис. 58

Пример 59. Сортировка элементов в лексикографическом порядке

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. 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 59. Сортировка в лексикографическом порядке

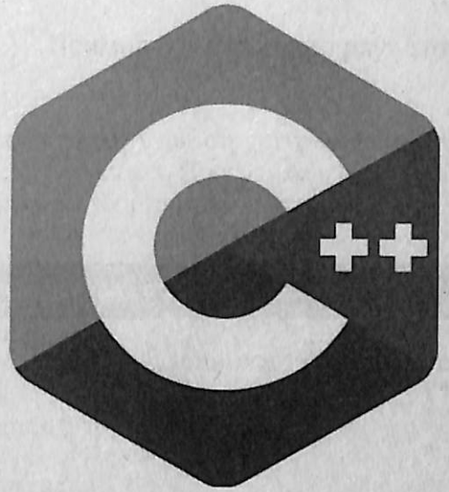
```
#include <iostream>
using namespace std;

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;
          }
    }

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

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

```
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. 59. Сортировка в лексикографическом порядкеав

Часть 6. Структуры и объединения



В этой части будут рассмотрены следующие примеры:

Пример 60. Храним информацию о студенте в структуре

Цель данного 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 60.

Листинг 60. 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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./60
Инфо о студенте
Имя: Иван
Курс: 1
Оценка: 5

Отображаем инфо
Имя: Иван
Курс: 1
Оценка: 5
den@den-pc:~/cpp-ex$
```

Рис. 60. Результат работы программы

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

Пример 61. 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 61. 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;
}
```

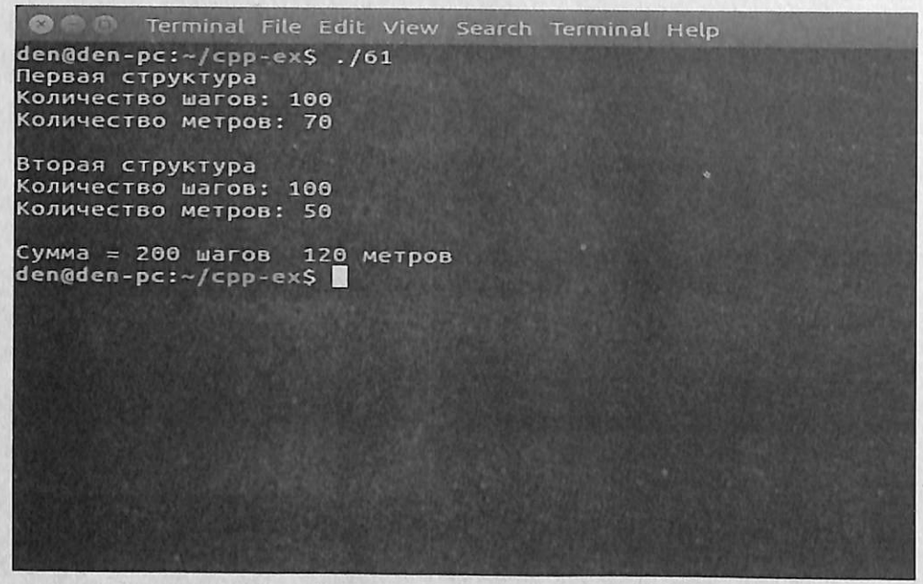


Рис. 61. Результат работы программы

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.

Пример 62. 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 62.

Листинг 62. Сложение двух структур с использованием функции (62.cpp)

```
#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. 62.

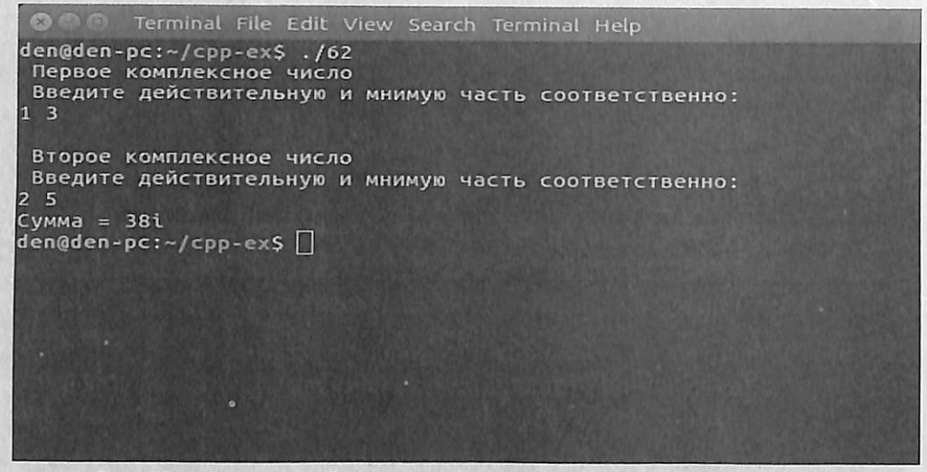


Рис. 62. Результат работы программы: сложение комплексных чисел

Пример 63. Вычислением разницы между двумя периодами времени

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 63. Вычисление разницы между двумя периодами времени (63.cpp)

```
#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;
    }

    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;
}
```

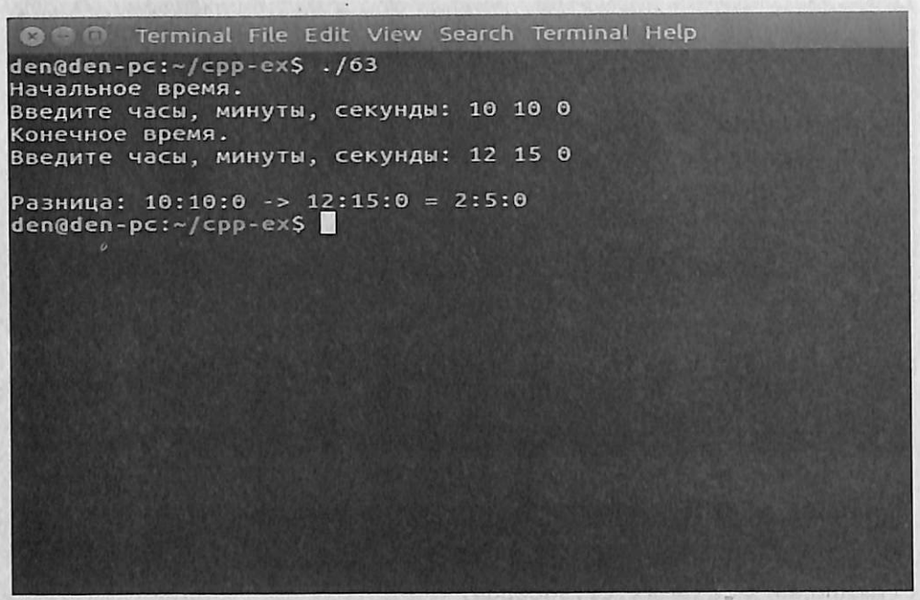


Рис. 63. Разница между двумя периодами времени

Пример 64. Работа с массивом структур

В примере 60 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 64 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.

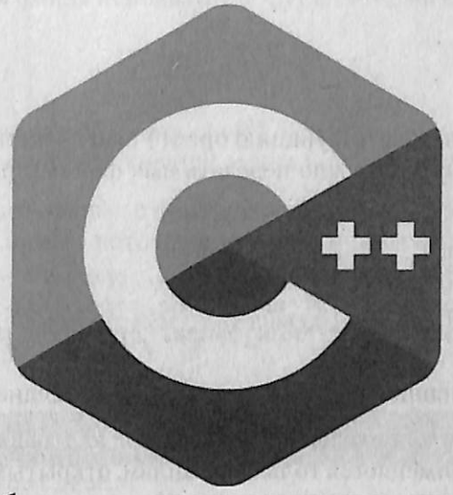
Листинг 64. Работа с массивом структур

```
#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;
    }

    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;
}
```

```
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. 64. Работа с массивом структур

Часть 7. Работаем с файлами



В этой части будут рассмотрены следующие примеры:

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.

Пример 65. Запись в файл

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. 65).

Листинг 65. 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.

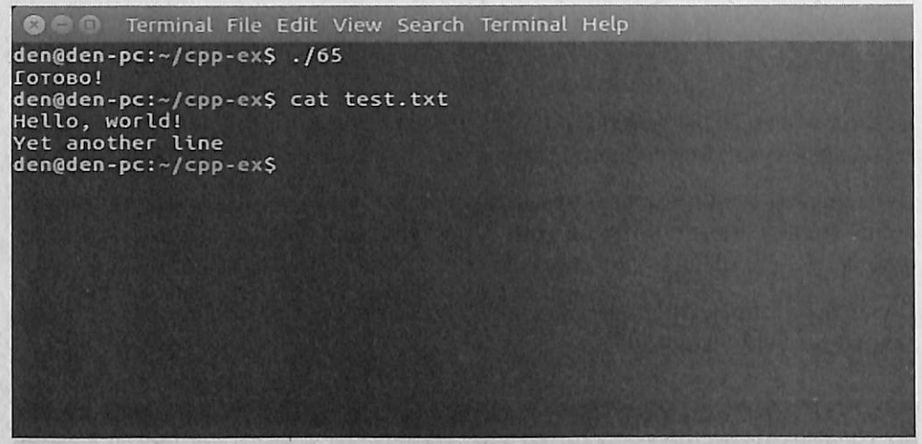


Рис. 65. Результат работы программы

Пример 66. 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 66. 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. 66. На нем продемонстрирована работа следующего оператора:

```
fin.unsetf(ios::skipws);
```

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.

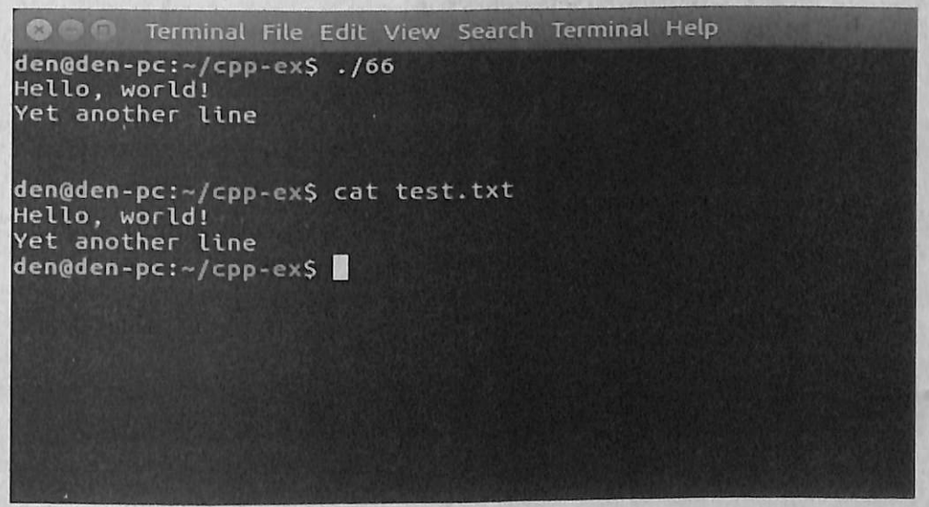


Рис. 66. Результат работы программы из листинга 66

Пример 67. 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 67. 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 67. 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;
}
```

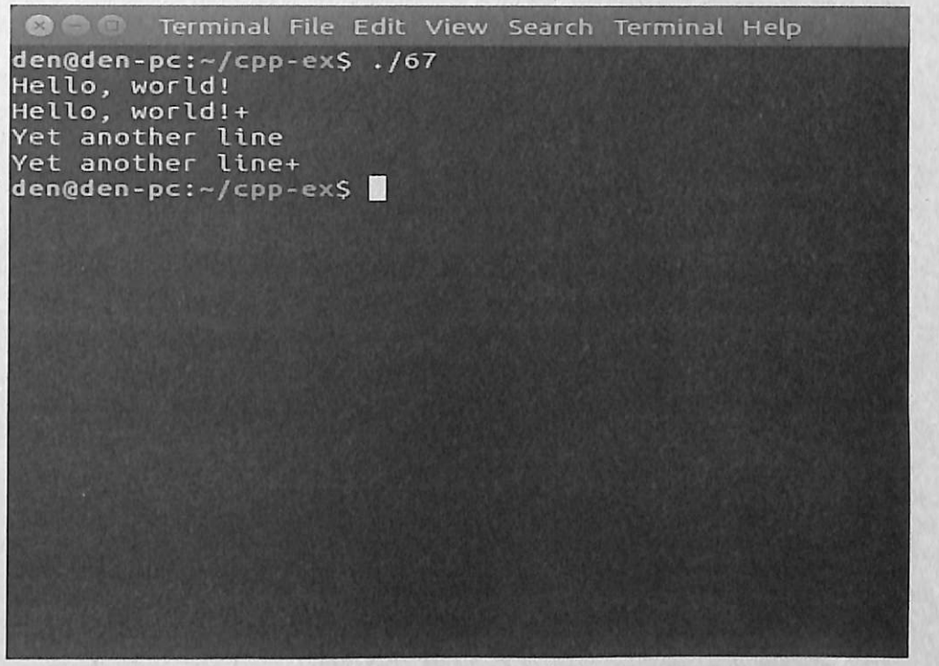



Рис. 67. Результат работы программ

Пример 68. Перегрузка операторов << и >>

В C++ допускается перегрузка любых операторов, в том числе и операторов ввода/вывода, что и будет показано в этом примере.

В программе мы создаем класс **coord**, для которого перегружаем операторы ввода/вывода. Оператор вывода будет записывать в поток сначала координату **x**, затем пробел, а после - координату **y** и символ новой строки. Оператор ввода будет читать координаты из потока.

Листинг 68. Перегрузка операторов ввода/вывода

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

```
class coord {

public:
  int x, y;
  coord (int i, int j) { x = i; y = j; }
  friend ostream &operator<< (ostream &stream, coord ob);
  friend istream &operator>> (istream &stream, coord &ob);
};

ostream &operator<< (ostream &stream, coord ob) {
  stream << ob.x << " " << ob.y << '\n';
  return stream;
}

istream &operator>> (istream &stream, coord &ob) {
  stream >> ob.x >> ob.y;
  return stream;
}

int main() {
  // создаем два объекта coord с координатами
  // 100,200 и 300,400
  coord o1(100, 200), o2(300, 400);
  // Поток вывода, в него будем записывать координаты
  ofstream out("coords.txt");

  if (!out) {
    cout << "Ошибка!";
    return 1;
  }

  out << o1 << o2;              // Записываем координаты в файл
  out.close();            // Закрываем файл

  // Поток для чтения (ввода)
  ifstream in("coords.txt");
  coord o3(0, 0), o4(0, 0);           // Координаты 0,0 и 0,0
  // Читаем из файла ранее записанные координаты
  in >> o3 >> o4;
  // Выводим координаты на консоль
  cout << o3 << o4;
  // Закрываем файл
  in.close();

  return 0;
}
```

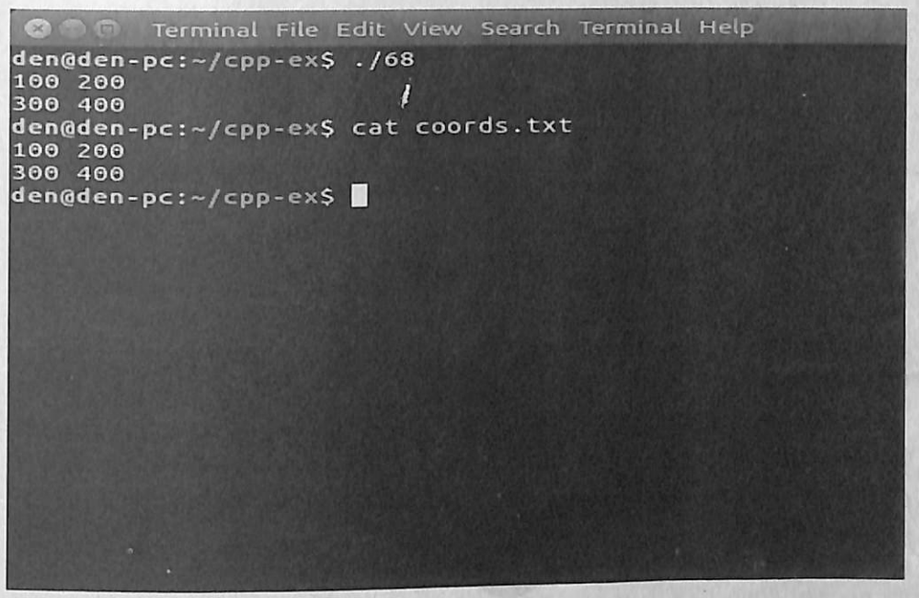
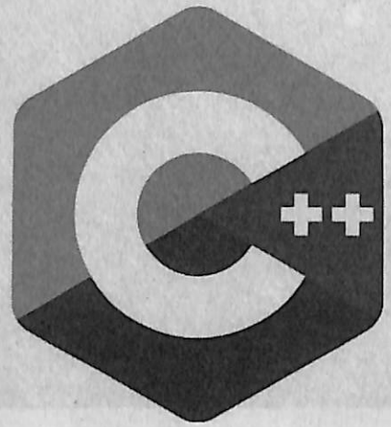



Рис. 68. Результат работы программы

Как видите, все достаточно просто и мы можем сохранять данные пользовательского класса в файл и читать их из файла с помощью перегрузки операторов ввода/вывода.

Часть 8. Объектно-ориентированное программирование



В этой части будут рассмотрены следующие примеры:

Пример 69. Пример класса

Пример 70. Конструкторы и деструкторы

Пример 71. Массивы объектов

Пример 72. Наследование

Пример 73. Перегрузка операторов

В данной части книги рассматриваются несколько примеров относящихся к объектно-ориентированному программированию. Напомню, что язык C не поддерживает ООП, поэтому все примеры в этой части книги будут написаны на C++.

Пример 69. Пример класса

Сейчас мы разработаем собственный класс. Чтобы пример имел практическую ценность, мы разработаем класс стека, который можно использовать для хранения символов. 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 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.

В листинге 69 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"):


```
g++ 69.cpp -o 69 -pedantic
```

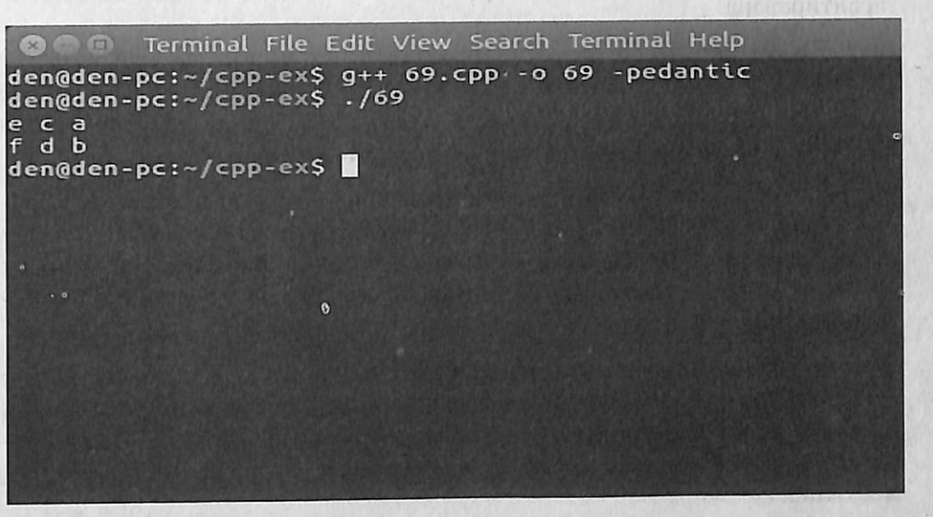


Рис. 69. Результат работы программы из листинга 69

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.

Пример 70. Конструкторы и деструкторы

Обратите внимание на программу из листинга 70. 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 70. 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 70. 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;
}
```

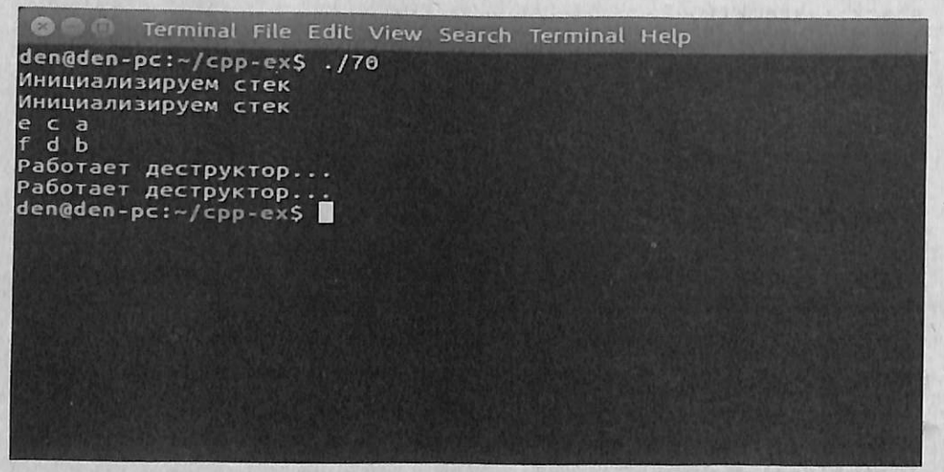


Рис. 70. Конструктор и деструктор

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 70б. Результат выполнения изображен на рис. 70б.

Листинг 70б. 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;
}
```

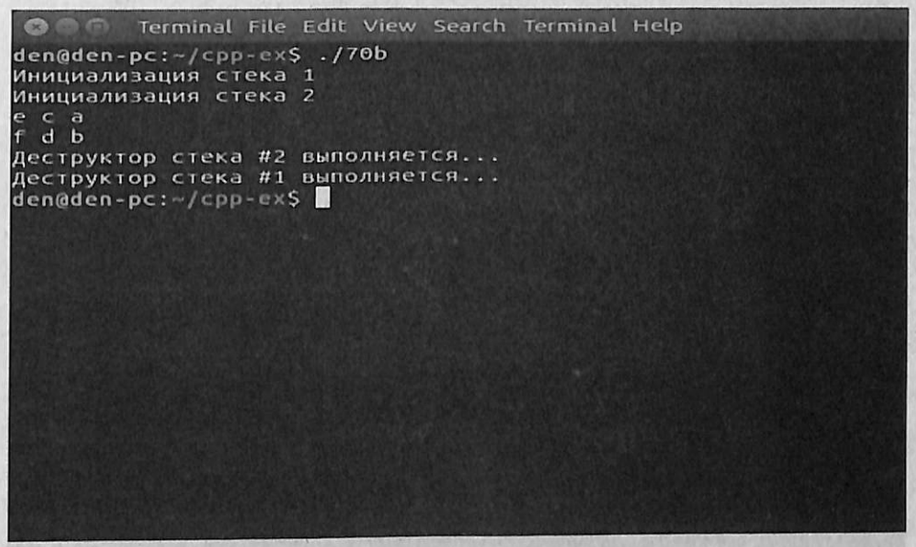


Рис. 70б. Результат идентификации стеков

Пример 71. Массивы объектов

Объекты - это обычные переменные. Следовательно, мы можем создать массив таких переменных, то есть массив объектов. В листинге 71а 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 71а. 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;
}
```

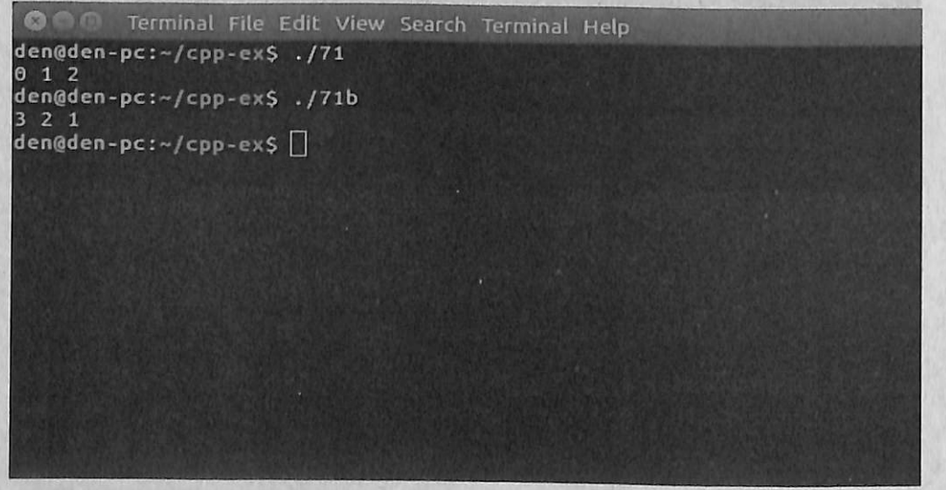


Рис. 71. Результат работы программ из лист. 71а и 71б

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. 71б).

Листинг 71б. 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 (рис. 71).

Пример 72. 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 72.

Листинг 72. 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
```

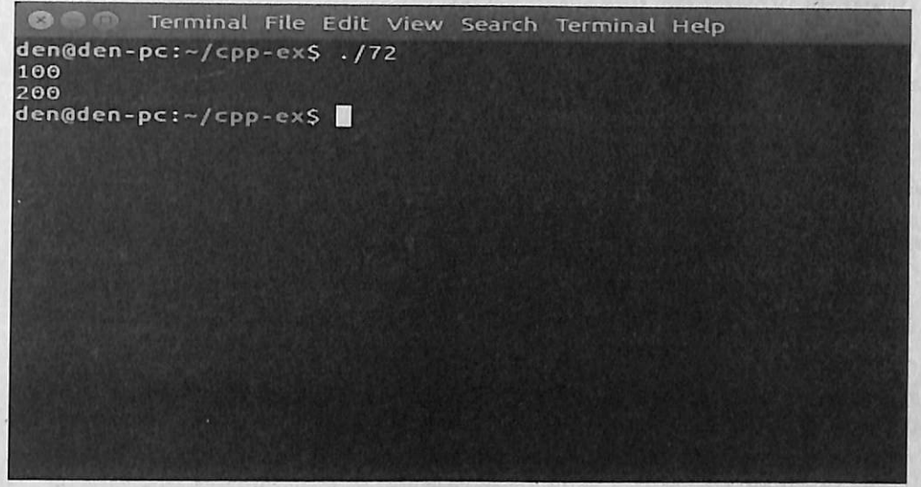


Рис. 72

Пример 73. 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 73 мы перегрузили операторы + и - для класса **coord**, содержащего координаты точки в двухмерном пространстве (**x** и **y**).

Листинг 73. 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**.

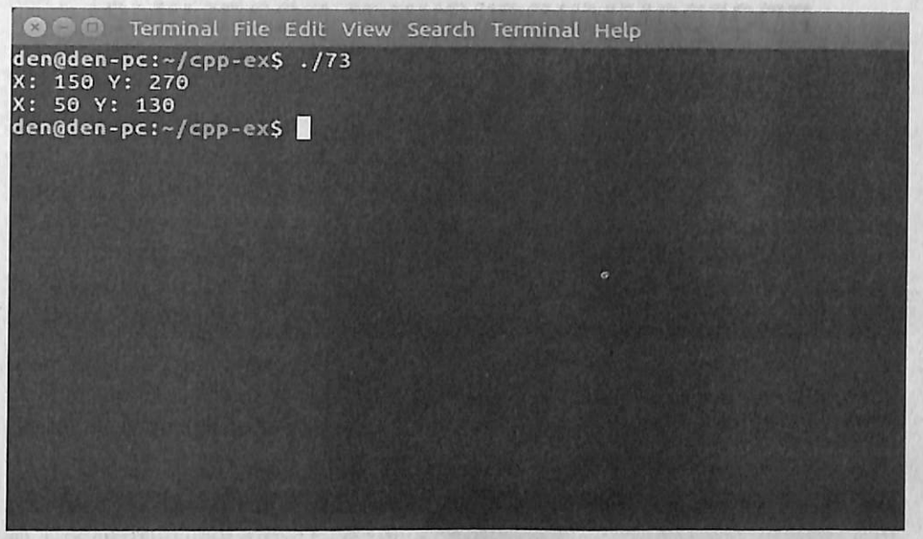
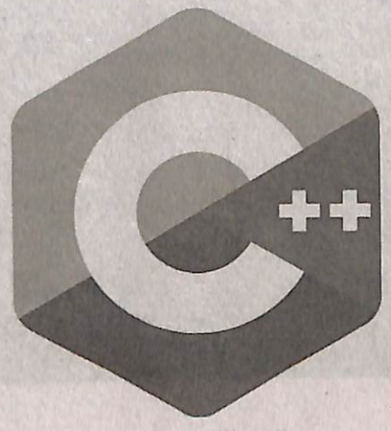


Рис. 73

Часть 9. Практика сетевого программирования: реальные приложения



В этой части будут рассмотрены следующие примеры:

Пример 74. Приложение-клиент

Пример 75. Приложение-сервер

*Пример 76. Используем команду **make** для сборки сложного проекта. Собираем все воедино*

Приведенные ранее примеры были больше теоретическими, нежели практическими. Но теория без практики - ничто, поэтому в данной главе будут разработано практическое приложение, а именно приложение Клиент-сервер, позволяющее обмениваться сообщениями по сети. Теория - это хорошо, но иногда нужно от нее отдыхать и делать "практические перерывы".

Пример 74. Приложение-клиент

В этой главе мы разработаем простое приложение клиент/сервер. 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.

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 74а. 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 74б 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 74б. 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 74в.

Листинг 74в. 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. 74. 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. 74. Клиент в действии

Небольшой итог. Мы только что разработали приложение-клиент, состоящее из трех файлов: client.cpp, TCPClient.cpp, TCPClient.h.

Пример 75. Приложение-сервер

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 75а.

Листинг 75а. Приложение-сервер (server.cpp)

```
#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 75б.

Листинг 75б. 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 75в.

Листинг 75в. 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*)&clientAddres
s,&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. 75 показан вывод сервера и двух подключенных клиентов. Наш сервер хоть и простой, но многопотоковый.

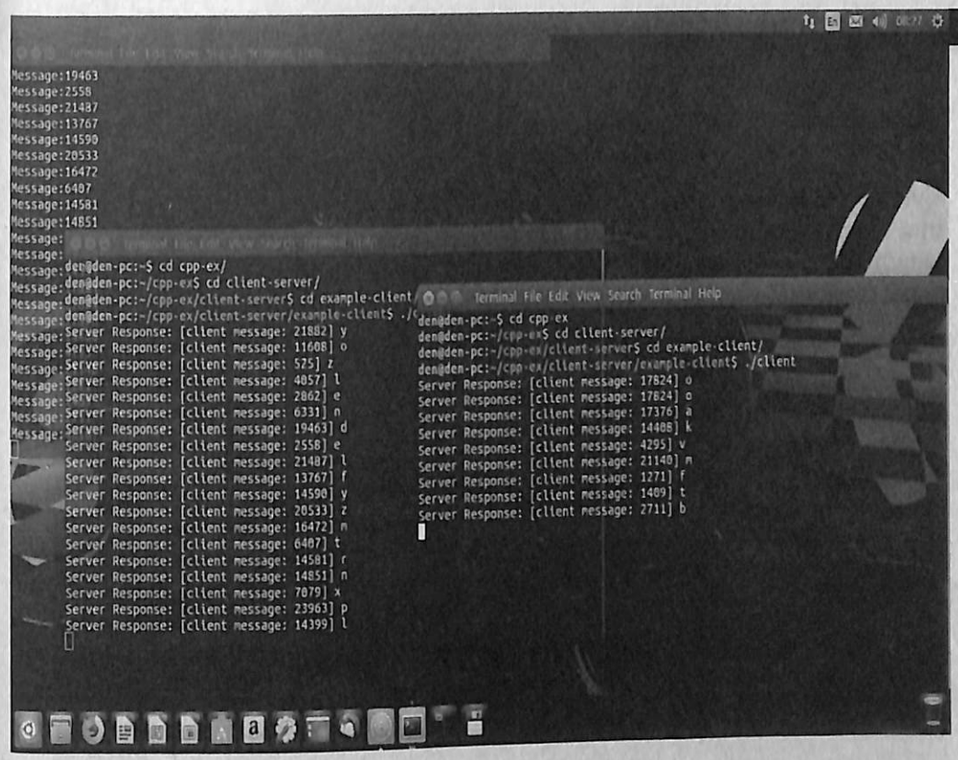


Рис. 75. Сервер и два клиента

Пример 76. Используем команду make для сборки сложного проекта. Собираем все воедино

Для сборки нашего клиента нужно ввести команду:

```
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).

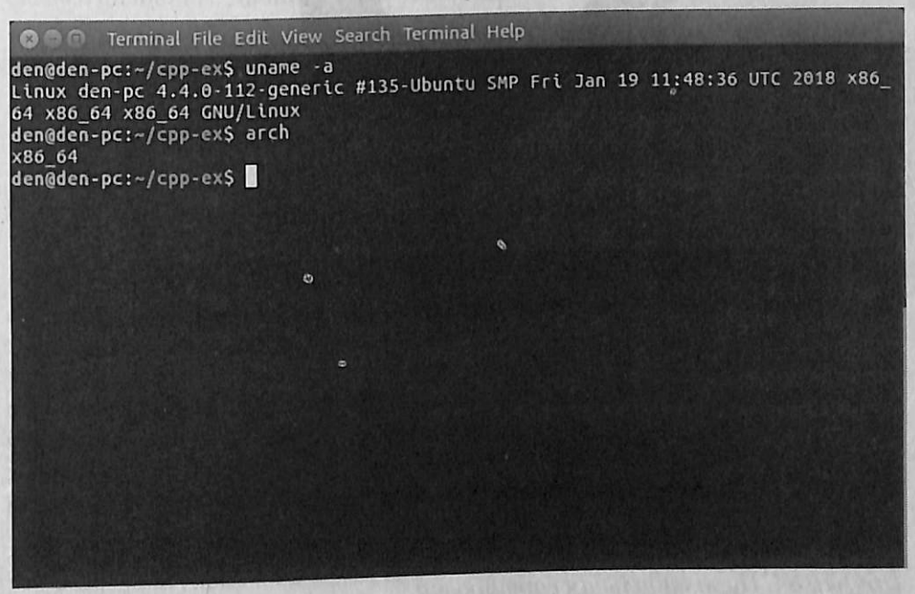
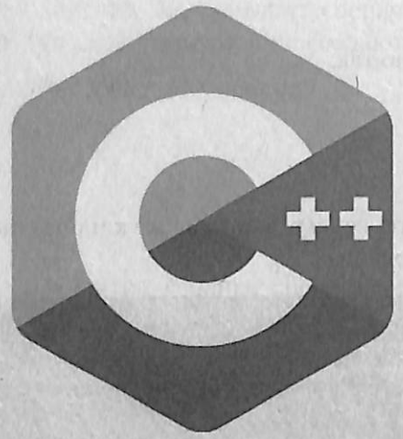


Рис. 76. Параметры системы, на которой осуществлялась сборка проекта

Часть 10. Алгоритмы поиска и сортировки



В этой части будут рассмотрены следующие примеры:

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 части мы рассмотрим множество примеров, которые, я надеюсь, помогут вам при написании лабораторной, курсовой работы или даже собственного реального приложения.

Пример 77. Бинарный поиск в целочисленном массиве

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 77. Двоичный поиск в целом (int) массиве (77.cpp)

```
#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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ g++ 77.cpp -o 77 -std=c++11
den@den-pc:~/cpp-ex$ ./77
Массив: 0 1 2 3 4 6 7 8 9 10
Ищем число: 5
-- Не найдено --
den@den-pc:~/cpp-ex$ ./77
Массив: 0 1 2 3 4 6 7 8 9 10
Ищем число: 6
** Найдено! **
den@den-pc:~/cpp-ex$
```

Рис. 77. Результат двоичного поиска

Посмотрите на рис. 77 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**

Пример 78. Бинарный поиск по массиву указателей строк

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 78, а результат его работы, как обычно, показан на рис. 78.

Листинг 78. Бинарный поиск по массиву строк (78.cpp)

```
#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;
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./78
0 audi
1 bentley
2 bmw
3 cadillac
4 ford

`bmw' найдено на позиции: 2
`mercedes' не найдено..
den@den-pc:~/cpp-ex$
```

Рис. 78. Результат двоичного поиска строки

Если понадобится переписать данную программу на C, то подключите следующие заголовочные файлы:

```
#include <stdio.h>
#include <string.h>
```

Далее измените все операторы << на функцию **printf()**.

Пример 79. Сортировка пузырьком

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 79. 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;
  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;
   }
 }
}
```

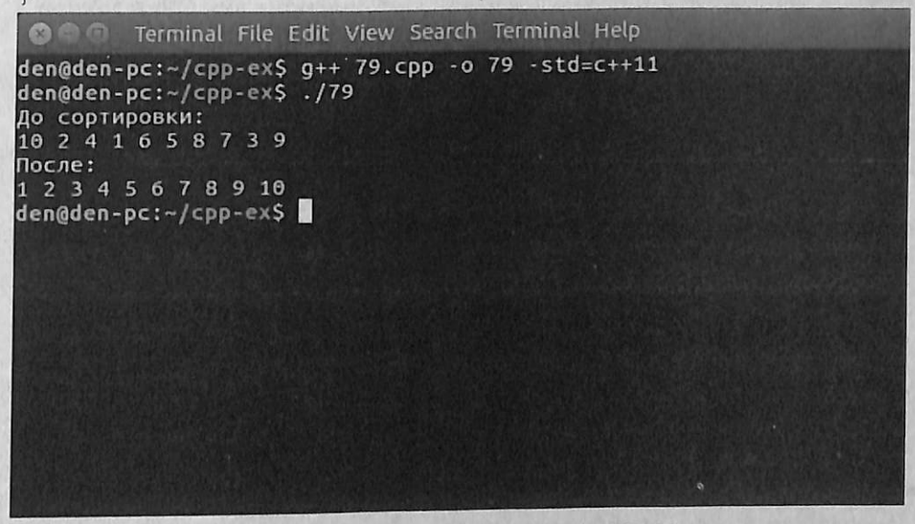


Рис. 79. Пузырьковая сортировка массива

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**

Пример 80. Быстрая сортировка массива

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 C++.

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 80.

Листинг 80. 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);
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./80
До сортировки: 83 86 77 15 93 35 86 92 49 21
После: 15 21 35 49 77 83 86 86 92 93
den@den-pc:~/cpp-ex$
```

Рис. 80. Сортировка массива методом быстрой сортировки

Пример 81. 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. 81. Как обычно, результат выполнения программы приводится на рис. 81.

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ g++ 81.cpp -o 81 -std=c++11 -fpermissive
81.cpp: In function 'void llist_add(lnode**, int)':
81.cpp:47:14: warning: invalid conversion from 'void*' to 'lnode*' [-fpermissive
]
   *q = malloc(sizeof(struct lnode));
              ^
81.cpp:55:23: warning: invalid conversion from 'void*' to 'lnode*' [-fpermissive
]
    temp->next = malloc(sizeof(struct lnode));
                       ^
den@den-pc:~/cpp-ex$ ./81
До:
83 86 77 15 93 35 86 92 49 21
После:
15 21 35 49 77 83 86 86 92 93
den@den-pc:~/cpp-ex$
```

Рис. 81. Сортировка выбором

Команда компиляции примера:

```
g++ 81.cpp -o 81 -std=c++11 -fpermissive
```

Листинг 81. 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;
  }
 }
```

Пример 82. Сортировка вставкой связного списка

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 82. 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;
}
```

Команда компиляции примера:

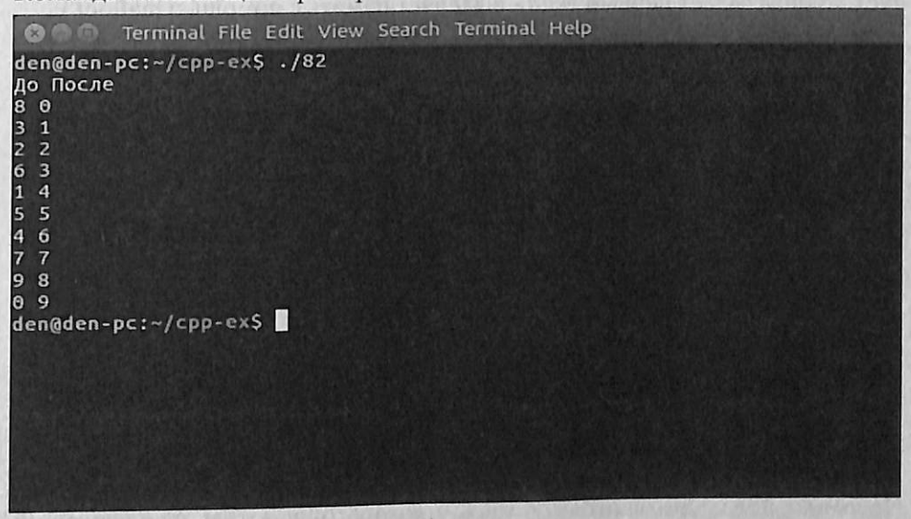


Рис. 82. Вывод программы

Пример 83. Пузырьковая сортировка связного списка

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 83, а результат ее выполнения - на рис. 83. В программе мы будем генерировать случайные числа, и ними же будем заполнять наш список - чтобы избавить вас от ввода чисел вручную.

Листинг 83. 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;
  }
 }
}
```

Команда компиляции примера:

```
g++ 83.cpp -o 83 -std=c++1 -fpermissive
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ g++ 83.cpp -o 83 -std=c++11 -fpermissive
83.cpp: In function 'void llist_add(lnode**, int)':
83.cpp:48:14: warning: invalid conversion from 'void*' to 'lnode*' [-fpermissive
]
   *q = malloc(sizeof(struct lnode));
              ^
83.cpp:56:22: warning: invalid conversion from 'void*' to 'lnode*' [-fpermissive
]
    tmp->next = malloc(sizeof(struct lnode));
                      ^
den@den-pc:~/cpp-ex$ ./83
До сортировки:
83 86 77 15 93 35 86 92 49 21
После:
15 21 35 49 77 83 86 86 92 93
den@den-pc:~/cpp-ex$
```

Рис. 83. Программа в действии. Пузырьковая сортировка связного списка

Пример 84. 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 84. 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);
 }
}
```

Как видно на рис. 84, программа сгенерировала случайные значения, поместила их в массив и выполнила сортировку этого массива.

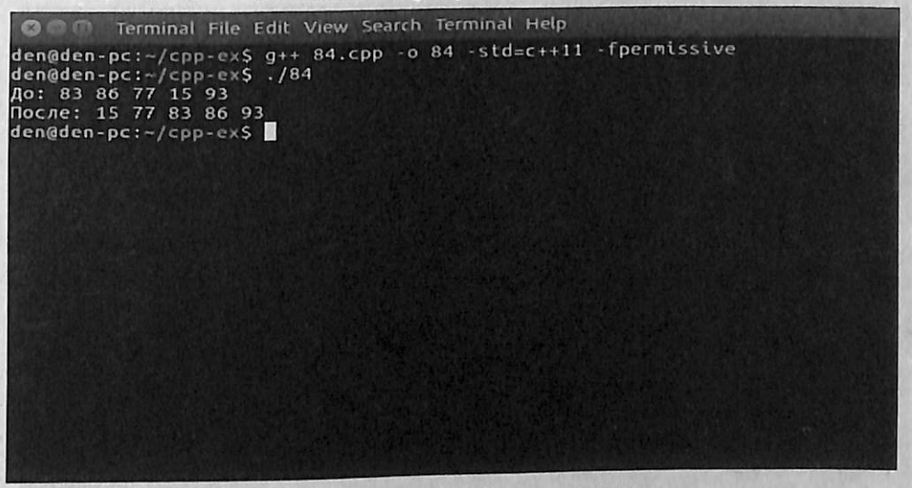


Рис. 84. Результат работы программы

Команда компиляции:

```
g++ 84.cpp -o 84 -std=c++11
```

Пример 85. 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 85. 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];
  arr[w] = arr[s];
  arr[s] = sm;
 }
}
```

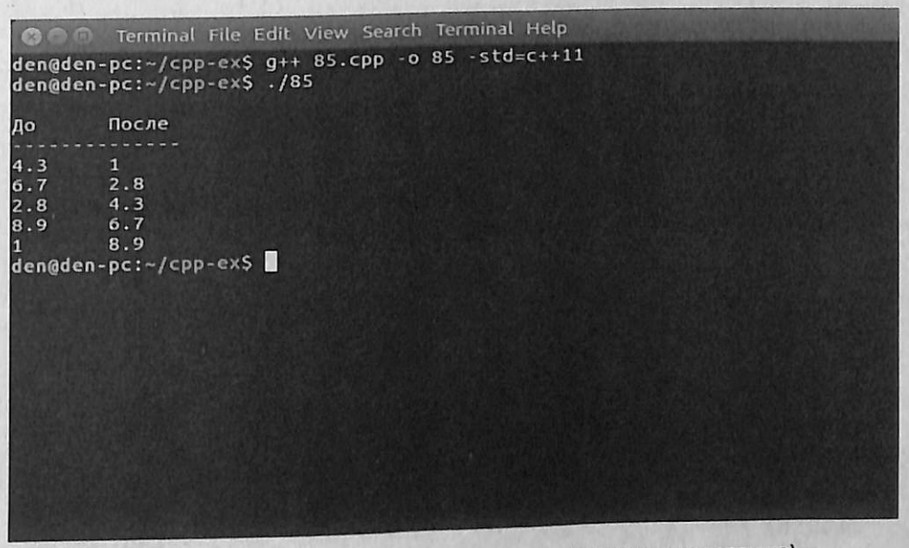


Рис. 85. Сортировка массива чисел (по убыванию и возрастанию)

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;
}
```

При компиляции программы укажите опцию -std=c++11.

Пример 86. Сортировка слиянием 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 86. Результат сортировки массива показан на рис. 86.

Листинг 86. Сортировка слиянием массива (86.cpp)

```
#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++];
 }
}
```

```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./86
До сортировки:83 86 77 15 93 35 86 92 49 21
После:15 21 35 49 77 83 86 86 92 93
den@den-pc:~/cpp-ex$
```

Рис. 86. Сортировка массива слиянием

Пример 87. 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 возвраща-

ем **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 87.

Листинг 87. 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);
 }

 return head_three;
}
```

При компиляции программы укажите опцию -std=c++11.

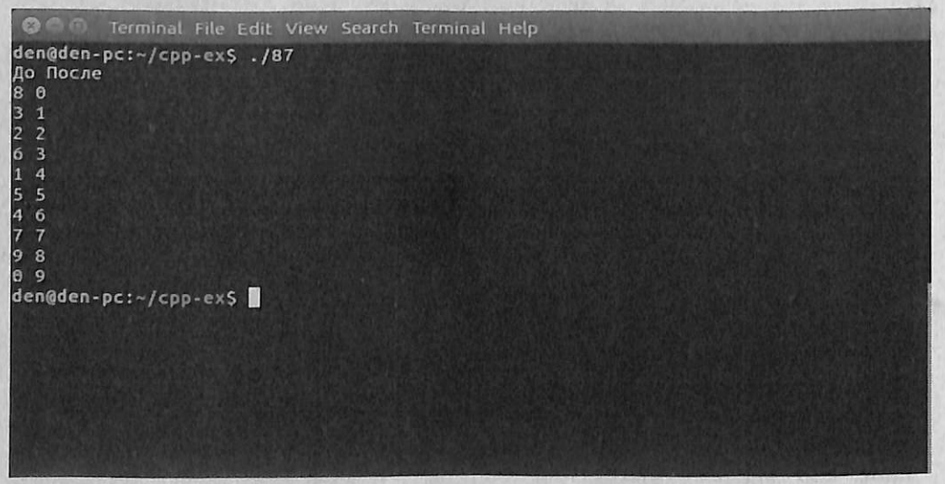


Рис. 87. Результат сортировки слиянием

Пример 88. 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 88.

Листинг 88. 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;

    return 0;
}
```

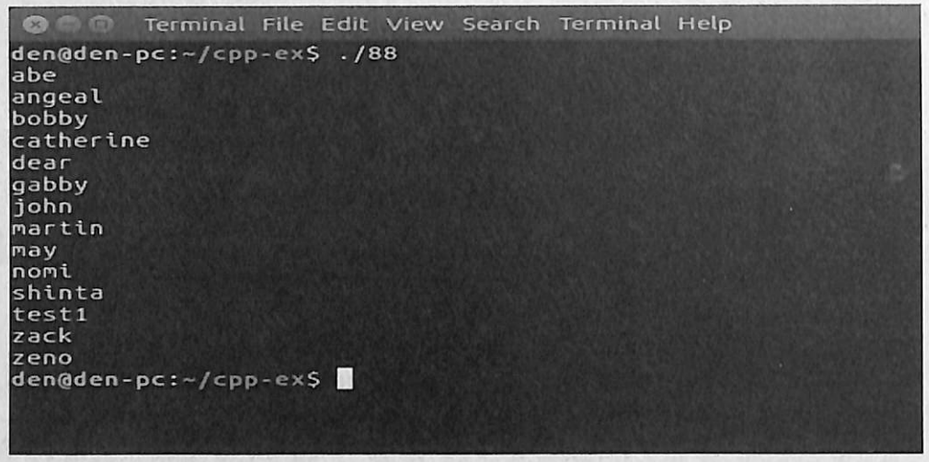


Рис. 88. Сортировка с использованием функции sort()

Пример 89. Использование итераторов begin() и end() для сортировки

Приведенный в предыдущем примере код не очень логичен и понятен. У новичков возникает сразу множество вопросов - не очень понятно, как мы получили последний элемент массива. К счастью, для решения этой проблемы, чтобы ваш код выглядел наглядно и современно, можно использовать итераторы **begin()** и **end()**, указывающие на первый и последний элементы массива соответственно. Пример кода приведен в листинге 89. Также в этом примере показано, как можно инициализировать массив с использованием vector. Компиляция программы требует наличия параметра -std=c++11.

Листинг 89. Использование итераторов для работы с массивом

```
#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;
}
```

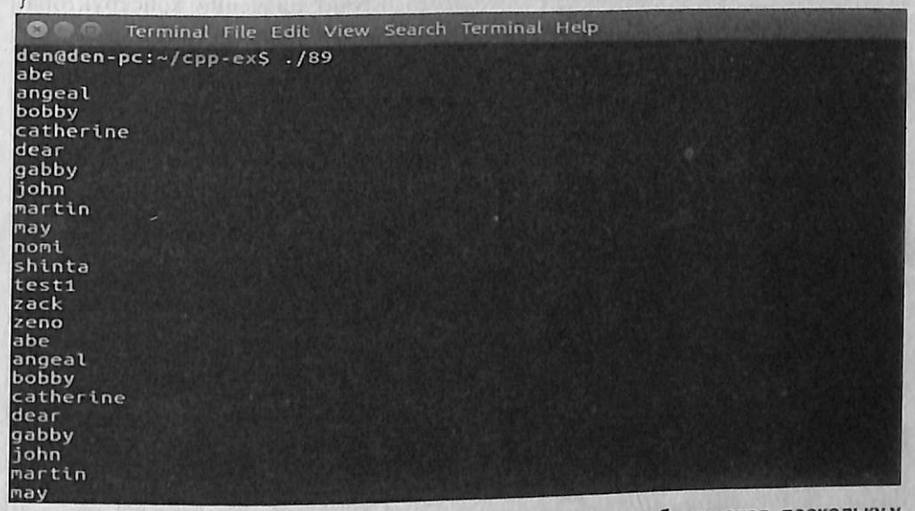


Рис. 89. Сортировка с использованием итераторов. Имена дублируются, поскольку у нас есть два цикла вывода - один требует C++11, другой - нет

Стандартный шаблон std::vector<T> — реализация динамического массива переменного размера.

Шаблон vector расположен в заголовочном файле <vector>, который расположен в пространстве имен **std**. Данный интерфейс эмулирует работу стандартного массива C (например, быстрый произвольный доступ к элементам), а также некоторые дополнительные возможности, вроде автоматического изменения размера вектора при вставке или удалении элементов.

Методы шаблона **vector** приведены в таблице 10.1.

Таблица 10.1. *Методы шаблона* ***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

Часть 11. Еще немного практики

В этой части будут рассмотрены следующие примеры:

Прошлая часть была больше теоретической, чем практической. Мы рассмотрели множество алгоритмов поиска и сортировки. В части книги будет рассмотрен ряд сугубо практических и интересных примеров.

Пример 90. Мини-игра Эволюция

В этом примере мы запрограммируем игру "Эволюция", она же "Жизнь". Оригинальное название этой игры Conway's Game of Life - игру придумал английский математик Джон Конвей еще в 1970 году.

Полное описание игры доступно в Википедии по адресу http://bit.ly/2E5F9bt. Основные правила такие:

1. Место действия игры - размеченная на клетки поверхность aka "вселенная". Вселенная может быть безграничная, ограниченная или замкнутая.
2. Каждая клетка на поверхности может находиться в двух состояниях - живая или мертвая. Живой считается заполненная клетка, мертвой - пустая клетка.

3. Распределение живых клеток в самом начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего по таким правилам:
   - В пустой клетке, рядом с которой есть ровно три живые клетки, зарождается жизнь.
   - Если у живой клетки есть 2 или 3 живые клетки по соседству, эта клетка продолжает жить, в противном случае (если соседей меньше 2 или больше 3) - клетка умирает (или от одиночества или от перенаселенности).
4. Игра прекращается, когда на поле не остается ни одной живой точки или при следующем шаге ни одна из клеток не изменит своего состояния, то есть складывается стабильная конфигурация клеток.

Для хранения информации о поле (то есть нашей "вселенной") мы будем использовать двумерный массив побитовых структур. Количеств строк этого массива будет высотой поля, а столбцов - шириной. Массив будет выглядеть вот так:

```
struct cell {
    unsigned is_live:1;
};

/* Ширина мира */
#define __WORLD_HEIGHT__ 10

/* Высота мира */
#define __WORLD_WIDTH__ 10

// Мир 10x10
cell world[__WORLD_WIDTH__][__WORLD_HEIGHT__];
```

Первое поколение клеток мы будем генерировать случайным образом. Для более качественных случайных чисел будет использоваться библиотека <random> из C++. Поэтому не забудьте при компиляции указать опцию -std=c++11.

Функция создания мира будет выглядеть так:

```
void make_world(cell world[][__WORLD_HEIGHT__])
{
    std::random_device rd;
```

```
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 10000);

    unsigned int i, j;

    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            unsigned int num = dis(gen);
            if (num % 2 == 0) {
                world[i][j].is_live = 1;
            } else {
                world[i][j].is_live = 0;
            }
        }
    }
}
```

Как только мы сформируем первое поколение, мы начнем игровой цикл, который завершится, когда все клетки умрут или мы найдем стабильную конфигурацию.

Функция **get_live_cells()** будет заниматься подсчетом живых клеток. Она будет возвращать счетчик всех живых клеток в мире:

```
unsigned int get_live_cells(cell world[][__WORLD_HEIGHT__])
{
    unsigned int count = 0;
    unsigned i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            if (world[i][j].is_live == 1) {
                count++;
            }
        }
    }
    return count;
}
```

Для каждого шага цикла нужно сгенерировать новое поколение, в соответствии с правилами игры. Функция, определяющая координаты соседей клетки, будет называться **get_cell_neighbors()**:

```
void get_cell_neighbors(signed int nb[][2], unsigned int x,
unsigned int y)
{
```



```
    unsigned int i, j;
    unsigned int k = 0;

    for (i = x - 1; i <= x + 1; i++) {
        for (j = y - 1; j <= y + 1; j++) {
            if (i == x && j == y) {
                continue;
            }
            nb[k][0] = i;
            nb[k][1] = j;
            k++;
        }
    }
}
```

Данная функция принимает указатель на двумерный массив (сюда будет записан результат) и координаты точки **x**, **y** - для этой точки мы ищем соседей.

Количество живых соседей клетки возвращает функция **get_live_neighbors()**. Ей нужно передать массив мира и координаты клетки.

```
unsigned int get_live_neighbors(cell world[][__WORLD_
HEIGHT__], unsigned int x, unsigned int y)
{
    unsigned int count = 0;
    unsigned int i;
    signed int nb[8][2];
    signed int _x, _y;

    read_point_neighbors(nb, x, y);

    for (i = 0; i < 8; i++) {
        _x = nb[i][0];
        _y = nb[i][1];

        if (_x < 0 || _y < 0) {
            continue;
        }
        if (_x >= __WORLD_WIDTH__ || _y >= __WORLD_HEIGHT__) {
            continue;
        }

        if (world[_x][_y].is_live == 1) {
            count++;
        }
```

```
    }

    return count;
}
```

Теперь мы подходим к самому интересному - к смене поколений. Мы подготовили все необходимые функции, чтобы построить следующее поколение клеток. Для этого у нас будет использоваться функция **next_generation()**:

```
void next_generation(cell world[][__WORLD_HEIGHT__], cell
prev_world[][__WORLD_HEIGHT__])
{
    unsigned int i, j;
    unsigned int live_nb;
    cell p;

    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            p = prev_world[i][j];
            live_nb = get_live_neighbors(prev_world, i, j);

            if (p.is_live == 0) {
                if (live_nb == 3) {
                    world[i][j].is_live = 1;
                }
            } else {
                if (live_nb < 2 || live_nb > 3) {
                    world[i][j].is_live = 0;
                }
            }
        }
    }
}
```

Нужно помнить о еще одном условии игры: если у двух поколений, идущих двух за другом идентичное состояние клеток. Следовательно, мы нам понадобится функция сравнения двух состояний мира - предыдущего и текущего. Эта функция будет называться **compare_world()**. А вот, чтобы старое состояние можно было сравнить с новым, мы должны старое состояние скопировать во временную переменную. Для этого мы будем использовать функцию **copy_world()**. Напишем эти функции:

```
// Копирование мира
void copy_world(cell src[][__WORLD_HEIGHT__], cell dest[][__
WORLD_HEIGHT__])
{
```



```
    unsigned int i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            dest[i][j] = src[i][j];
        }
    }
}

// Сравнение мира
int compare_world(cell w1[][__WORLD_HEIGHT__], cell w2[][__
WORLD_HEIGHT__])
{
    unsigned int i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            if (w1[i][j].is_live != w2[i][j].is_live) {
                return -1;
            }
        }
    }
    return 0;
}
```

За вывод на экран отвечает функция **show_world()**:

```
void show_world(cell world[][__WORLD_HEIGHT__])
{
    unsigned int i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            if (world[i][j].is_live == 1) {
                cout << '*';
            } else {
                cout << ' ';
            }
            cout << ' ';
        }
        cout << endl;
    }
}
```

Осталось написать только функцию **main()**, которая запустит игровой процесс:

```
int main()
{
    cell world[__WORLD_WIDTH__][__WORLD_HEIGHT__];
    cell temp[__WORLD_WIDTH__][__WORLD_HEIGHT__];
```

```
    make_world(world);
    unsigned int live_points = -1;
    bool is_stab = false;

    do {
        show_world(world);
        copy_world(world, temp);
        next_generation(world, temp);

        is_stab = compare_world(world, temp) == 0;
        live_points = get_live_cells(world);

        if (is_stab) {
            cout << "Найдена стабильная конфигурация" << endl;
        }

        if (live_points == 0) {
            cout << "Все клетки умерли" << endl;
        }
        msleep(1000);
    } while (live_points != 0 && !is_stab);

    return 0;
}
```

Полный исходный код игры приведен в листинге 90.

Листинг 90. Полный исходный код

```
#include <iostream>
#include <random>

#ifdef _WIN32
    #include <windows.h>
    #define msleep(x) Sleep(x)
#else
    #include <unistd.h>
    #define msleep(x) usleep(x * 1000)
#endif


using std::cout;
using std::cin;
using std::cerr;
using std::endl;
```



```
/* Ширина мира */
#define __WORLD_HEIGHT__ 20

/* Высота мира */
#define __WORLD_WIDTH__ 20

struct cell {
    unsigned is_live:1;
};

void make_world(cell world[][__WORLD_HEIGHT__])
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 10000);

    unsigned int i, j;

    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            unsigned int num = dis(gen);
            if (num % 2 == 0) {
                world[i][j].is_live = 1;
            } else {
                world[i][j].is_live = 0;
            }
        }
    }
}

void show_world(cell world[][__WORLD_HEIGHT__])
{
    unsigned int i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            if (world[i][j].is_live == 1) {
                cout << '*';
            } else {
                cout << ' ';
            }
            cout << ' ';
        }
        cout << endl;
    }
}
```

```
unsigned int get_live_cells(cell world[][__WORLD_HEIGHT__])
{
    unsigned int count = 0;
    unsigned i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            if (world[i][j].is_live == 1) {
                count++;
            }
        }
    }
    return count;
}

void get_cell_neighbors(signed int nb[][2], unsigned int x,
unsigned int y)
{
    unsigned int i, j;
    unsigned int k = 0;

    for (i = x - 1; i <= x + 1; i++) {
        for (j = y - 1; j <= y + 1; j++) {
            if (i == x && j == y) {
                continue;
            }
            nb[k][0] = i;
            nb[k][1] = j;
            k++;
        }
    }
}

unsigned int get_live_neighbors(cell world[][__WORLD_
HEIGHT__], unsigned int x, unsigned int y)
{
    unsigned int count = 0;
    unsigned int i;
    signed int nb[8][2];
    signed int _x, _y;

    get_cell_neighbors(nb, x, y);

    for (i = 0; i < 8; i++) {
        _x = nb[i][0];
        _y = nb[i][1];
        if (_x < 0 || _y < 0) {
```



```
            continue;
        }
        if (_x >= __WORLD_WIDTH__ || _y >= __WORLD_HEIGHT__) {
            continue;
        }

        if (world[_x][_y].is_live == 1) {
            count++;
        }
    }

    return count;
}

void next_generation(cell world[][__WORLD_HEIGHT__], cell
prev_world[][__WORLD_HEIGHT__])
{
    unsigned int i, j;
    unsigned int live_nb;
    cell p;

    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            p = prev_world[i][j];
            live_nb = get_live_neighbors(prev_world, i, j);

            if (p.is_live == 0) {
                if (live_nb == 3) {
                    world[i][j].is_live = 1;
                }
            } else {
                if (live_nb < 2 || live_nb > 3) {
                    world[i][j].is_live = 0;
                }
            }
        }
    }
}

void copy_world(cell src[][__WORLD_HEIGHT__], cell dest[][__
WORLD_HEIGHT__])
{
    unsigned int i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            dest[i][j] = src[i][j];
```

```
        }
    }
}
int compare_world(cell w1[][__WORLD_HEIGHT__], cell w2[][__
WORLD_HEIGHT__])
{
    unsigned int i, j;
    for (i = 0; i < __WORLD_WIDTH__; i++) {
        for (j = 0; j < __WORLD_HEIGHT__; j++) {
            if (w1[i][j].is_live != w2[i][j].is_live) {
                return -1;
            }
        }
    }
    return 0;
}

int main()
{
    cell world[__WORLD_WIDTH__][__WORLD_HEIGHT__];
    cell temp[__WORLD_WIDTH__][__WORLD_HEIGHT__];

    make_world(world);
    unsigned int live_cells = -1;
    bool is_stab = false;

    do {
        show_world(world);
        copy_world(world, temp);
        next_generation(world, temp);

        is_stab = compare_world(world, temp) == 0;
        live_cells = get_live_cells(world);

        if (is_stab) {
            cout << "Найдена стабильная конфигурация" << endl;
        }

        if (live_cells == 0) {
            cout << "All cells died" << endl;
        }
        msleep(1000);
    } while (live_cells != 0 && !is_stab);

    return 0;
}
```

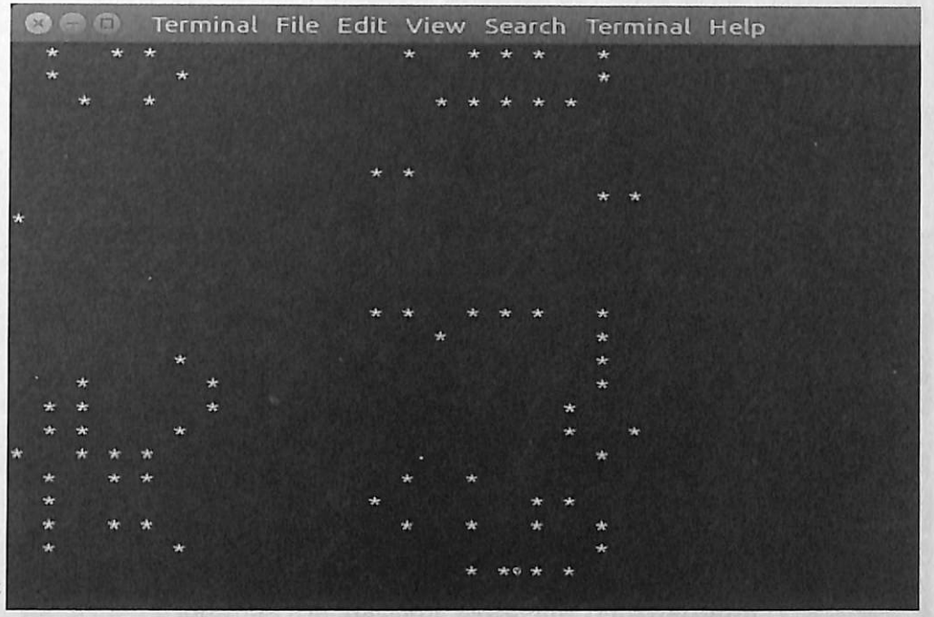



Рис. 90. Игра в действии

Пример 91. Получение информации о системе

Напишем простенькую программу, выводящую информацию о системе - имя системы, архитектуру системы, имя узла (**hostname**), релиз и версию ядра.

Листинг 91. Получение информации о системе на C++

```
#include <iostream>
#include <sys/utsname.h>
using namespace std;
int main(){
struct utsname sysinfo;
uname(&sysinfo);
cout << "System Name:                "<<sysinfo.sysname<<endl;
cout << "Machine Arch:               "<<sysinfo.machine<<endl;
cout << "Host Name:                  "<<sysinfo.
nodename<<endl;
cout << "Release(Kernel) Version: "<<sysinfo.release<<endl;
cout << "Kernel Build Timestamp:  "<<sysinfo.version<<endl;
cout << "Domain Name:                "<<sysinfo.
domainname<<endl;
```

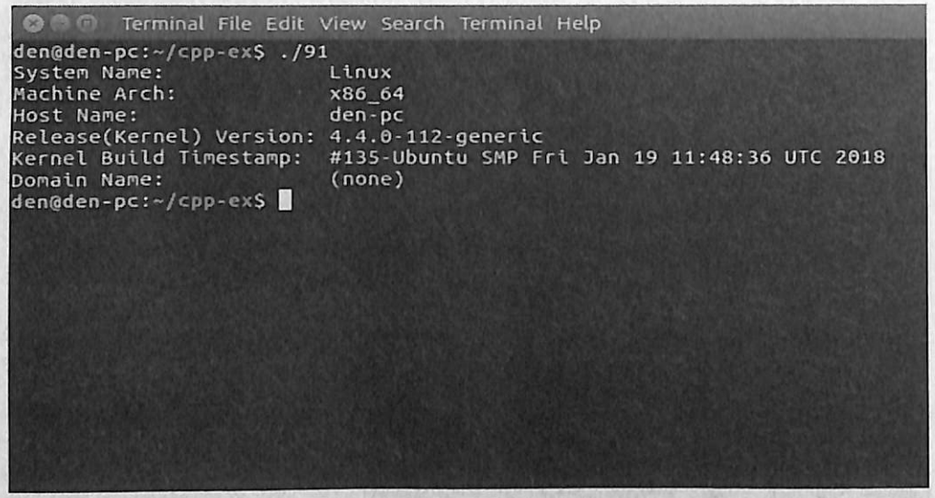


Рис. 91. Информация о системе

Пример 92. Обработка полученного сигнала

Напишем программу, обрабатывающую сигнал. В части 9 шла речь об обработке сигналов, но конкретного примера не было. Попробуем обработать сигнал. Для этого мы используем системный вызов **signal()** и устанавливаем обработчик сигнала - функцию **ex_program**. Тип сигнала SIGINT - прерывание программы с помощью Ctrl + C. Результат выполнения программы и перехвата сигнала приведен на рис. 92.

Листинг 92. Обработка сигнала

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

namespace
{
  volatile std::sig_atomic_t gSignalStatus;
}

void signal_handler(int signal)
{
  gSignalStatus = signal;
  std::cout << "Подъем!!!";
  exit(1);
}

int main()
{
  // Установка обработчика сигнала
  std::signal(SIGINT, signal_handler);

  std::cout << "Спим...\n";
  while (1) ;
}
```

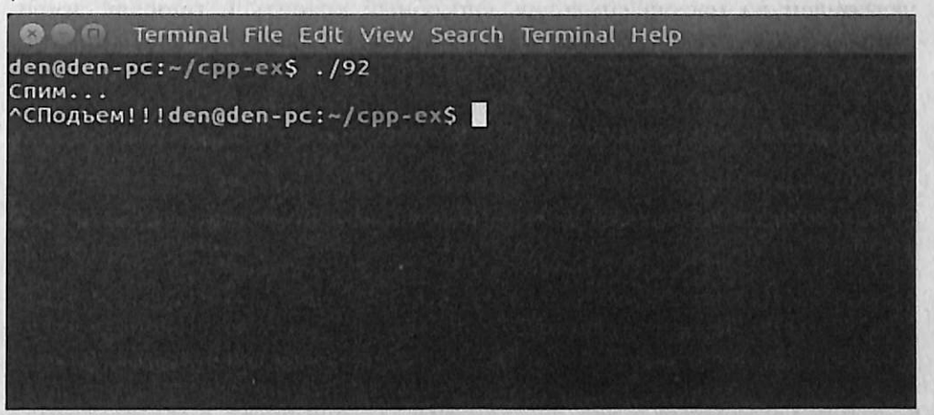


Рис. 92. Пример обработки сигнала

Пример 93. Создание временного файла

В процессе обработки различных данных часто появляется необходимость создать временный файл. Сначала мы получаем имя временного файла с помощью **tmpnam()** - оно может понадобиться в некоторых случаях.

Далее функцией **tmpfile()** создаем временный файл. После этого мы можем работать с ним, как с обычным файлом. Файл открывается в режиме **wb+** (запись, бинарный режим).

Листинг 93. Создание временного файла

```
#include <iostream>
#include <cstdio>
#include <string>

int main()
{
```

```
    std::string name1 = std::tmpnam(nullptr);
    std::cout << "имя временного файла: " << name1 << '\n';

    char name2[L_tmpnam];
    if (std::tmpnam(name2)) {
        std::cout << "имя временного файла: " << name2 <<
'\n';
    }
}
```

Функция **tmpnam()** принимает параметр **filename**, который должен быть указателем на массив символов, способный хранить **L_tmpnam** символов, но гораздо проще использовать тип **string**. Обратите внимание, что в программе мы не использовали подключение пространства имен (**using namespace std**) и нам пришлось везде указывать std:: - это очень неудобно, поэтому проще один раз указать оператор **using namespace std**. Однако такие программы встречаются - чтобы вы не удивлялись.

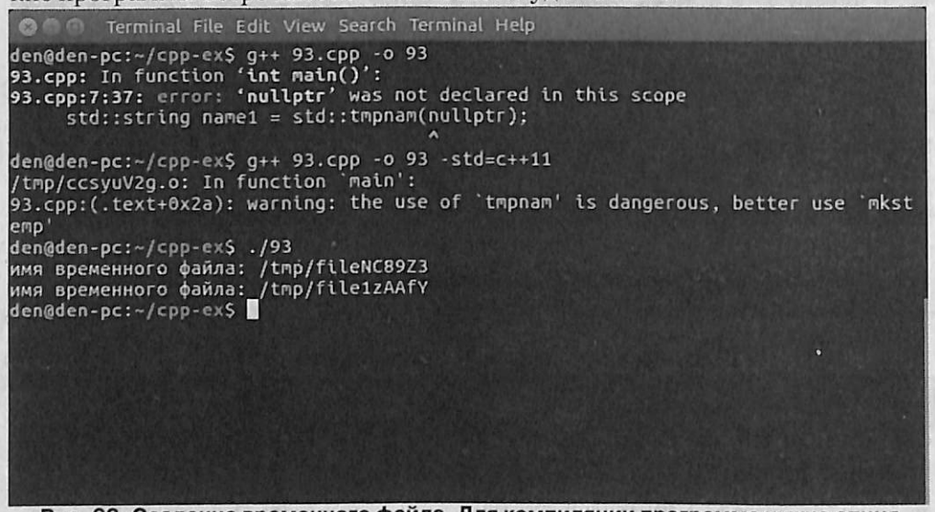


Рис. 93. Создание временного файла. Для компиляции программа нужна опция -std=c++11!

Пример 94. Простейшее шифрование файлов

Данный пример показывает, как можно зашифровать и расшифровать файлы. Конечно, шифрование у нас будет самое простое и его нельзя использовать в реальных приложениях, поскольку расшифровать такой файл будет очень просто. Тем не менее, программа демонстрирует сам процесс шифрования и расшифровки (а функции шифрования/расшифровки вы можете написать другие).

В программе мы будем использовать следующий класс:

```
class passkey
{
private:
    int password, key, k;
public:
    passkey();
    void getkey();
    int checkkey(int );
    friend void encrypt(passkey);
    friend void decrypt(passkey);
} file1;
```

Он содержит всю информацию относительно шифрования - пароль, ключ, а также различные методы. Основных метода два - **encrypt()** и **decrypt()**, которые, как понятно из названия, используются для шифрования и расшифровки.

Метод **getkey()** запрашивает у пользователя ключ шифрования:

```
void passkey::getkey()
{
    cout << "Введите ключ: ";
    cin >> key;
}
```

Метод **checkkey()** проверяет ключ:

```
int passkey::checkkey(int a)
{
    if(a==key)
        return 1;
    else
        return 0;
}
```

А вот как выглядит функция шифрования:

```
void encrypt(passkey file)
{
    char encfilenaam[50] = "Encrypted_";
    strcat(encfilenaam,filenaam);

    int n;

    srand(file.key);
```

```
    ifstream OutputData(filenaam);

    char ch[50] = "";

    ofstream InputData(encfilenaam);

    OutputData.getline(ch,49);
    do
    {
        n= 1 + rand()%9;
        for(int i=0; i<strlen(ch); i++)
        {
            ch[i] = ch[i] + n;
        }
        InputData << ch;
    }
    while(OutputData.getline(ch,49));

    OutputData.close();
    InputData.close();

    cout << "Файл успешно зашифрован"  << endl;
    cout << "Зашифрованные данные сохранены в файле: " <<
encfilenaam << endl;
}
```

Алгоритм шифрования очень простой. Введенным ключом мы инициализируем генератор случайных чисел. Затем получаем псевдослучайные числа и делаем смещение каждого символа на псевдослучайное число. При расшифровке мы аналогично инициализируем генератор тем же ключом и выполняем обратное смещение. Если ключ неправильный, то в результирующий файл будет записано что угодно, кроме исходного текста. По сути, мы просто модифицируем символы введенного текста. Не нужно слишком полагаться на наш алгоритм.

После шифрования зашифрованный текст помещается в файл с именем имя.txt. Расширение .txt мы автоматически будем добавлять в функции **main()**, чтобы меньше вводить с клавиатуры.

Аналогично, вот функция расшифровки. Она проделывает обратное преобразование. Если до этого мы добавляли смещение кода символа, то сейчас убираем его.

```
void decrypt(passkey file)
{
    char decfilenaam[50] = "Decrypted_";
```



```
    strcat(decfilenaam,filenaam);

    int m;

    srand(file.key);

    ifstream OutputData(filenaam);

    char ch[50] = "";

    ofstream InputData(decfilenaam);

    OutputData.getline(ch,49);
    do
    {
        m= 1 + rand()%9;
        for(int i=0; i<strlen(ch); i++)
        {
            ch[i] = ch[i] - m;
        }
        InputData << ch;
    }
    while(OutputData.getline(ch,49));

    OutputData.close();
    InputData.close();

    cout << "Файл успешно дешифрован" << endl;
    cout << "Дешифрованный текст сохранен в " << decfilenaam <<
endl;
}
```

В функции **main()** мы запрашиваем пароль, чтобы никто кроме нас не смог использовать программу. Поскольку это пример, то пароль мы выводим на экран в виде подсказки, чтобы самому не забыть на время тестирования программы. Вы можете перекомпилировать программу и указать другой пароль (разумеется, и отключить его подсказку):

```
int main()
{
     int a, pw, x, k;
     string c;

      cout << "Введите пароль (88): ";
      cin>>pw;
```

```
    if(pw!=88)
      {
          cout << "Неправильный пароль" << endl;

          return 0;
      }

    cout << "1 - шифрование\n2 - расшифровка\n3 - выход"
<< endl;
    cin>> a;
```

Мы просим пользователя выбрать действие. Если пользователь выберет шифрование, то мы запросим у него ключ, попросим ввести имя файла, а также текст для шифровки:

```
      file1.getkey();          // вводим ключ

    // вводим имя файла
      cout << "Имя файла\nнапр: \"Input_Data\"" << endl;
      cin >> filenaam;

    // добавляем расширение, чтобы не вводить его
      strcat(filenaam,".txt");

      cout << "Данные для шифровки (Ctrl + D для завершения
ввода): " << endl;

      ofstream InputData(filenaam);

      while(cin >> c)
      {
          InputData << c << " ";
      }

      InputData.close();

      encrypt(file1);

      return 0;
```

Когда все данные собраны, мы вызываем метод шифрования текста. Расшифровка проходит аналогично. Мы запрашиваем ключ, имя файла и расшифровываем сообщение. Полный код программы приведен в листинге 94.

Листинг 94. Программа для шифрования и расшифровки файла

```
#include<iostream>
#include<string>
#include<string.h>
#include<fstream>
#include<cstdlib>

using namespace std;

class passkey
{
private:
    int password, key, k;
public:
    passkey();
    void getkey();
    int checkkey(int );
    friend void encrypt(passkey);
    friend void decrypt(passkey);
} file1;

passkey::passkey()
{
    password = 88;
}

void passkey::getkey()
{
    cout << "Введите ключ: ";
    cin >> key;
}

int passkey::checkkey(int a)
{
    if(a==key)
        return 1;
    else
        return 0;
}

char filenaam[40];

int main()
{
    int a, pw, x, k;
```

```
    string c;

        cout << "Введите пароль (88): ";
    cin>>pw;
       if(pw!=88)
       {
           cout << "Неправильный пароль" << endl;

           return 0;
       }

    cout << "1 - шифрование\n2 - расшифровка\n3 - выход" <<
endl;
    cin>> a;
    if(a==1)
    {
        file1.getkey();

        cout << "Имя файла\nнапр: \"Input_Data\"" << endl;
        cin >> filenaam;

        strcat(filenaam,".txt");

       cout << "Данные для шифровки (Ctrl + D для завершения
ввода): " << endl;

       ofstream InputData(filenaam);

       while(cin >> c)
       {
           InputData << c << " ";
       }

       InputData.close();

       encrypt(file1);

       return 0;

    }
    else if(a==2)
    {
        file1.getkey();

        cout << "Введите имя файла\nнапр: Input_Data" <<
endl;
```



```
        cin >> filenaam;

        strcat(filenaam,".txt");

       decrypt(file1);

       return 0;

    }

    else if (a==3)
    {
        return 0;
    }
}

void encrypt(passkey file)
{
    char encfilenaam[50] = "Encrypted_";
    strcat(encfilenaam,filenaam);

    int n;
    srand(file.key);
    ifstream OutputData(filenaam);
    char ch[50] = "";
    ofstream InputData(encfilenaam);
    OutputData.getline(ch,49);
    do
    {
        n= 1 + rand()%9;
        for(int i=0; i<strlen(ch); i++)
        {
            ch[i] = ch[i] + n;
        }
        InputData << ch;
    }
    while(OutputData.getline(ch,49));

    OutputData.close();
    InputData.close();

    cout << "Файл успешно зашифрован"  << endl;
    cout << "Зашифрованные данные сохранены в файле: " <<
encfilenaam << endl;
}
```

```
void decrypt(passkey file)
{
    char decfilenaam[50] = "Decrypted_";
    strcat(decfilenaam,filenaam);

    int m;

    srand(file.key);

    ifstream OutputData(filenaam);

    char ch[50] = "";

    ofstream InputData(decfilenaam);

    OutputData.getline(ch,49);
    do
    {
        m= 1 + rand()%9;
        for(int i=0; i<strlen(ch); i++)
        {
            ch[i] = ch[i] - m;
        }
        InputData << ch;
    }
    while(OutputData.getline(ch,49));

    OutputData.close();
    InputData.close();

    cout << "Файл успешно дешифрован" << endl;
    cout << "Дешифрованный текст сохранен в " << decfilenaam <<
endl;
}
```

Теперь посмотрим на нее в действии (рис. 94). Мы ввели ключ (11), имя файла **mmm** и текст "привет мир". Программа зашифровала текст и сохранила его в файле Encrypted_mmm.txt.

Далее мы выводим этот файл командой **cat**. Как видите, сообщение зашифровано. Далее мы опять запускаем программу и выбираем расшифровку файла. Вводим ключ и имя файла Ecrypted_mmm. Программа выполнила расшифровку и сохранила расшифрованный текст в файле Decrypted_Encrypted_mmm.txt. Выводим содержимое этого файла командой **cat** и видим нашу строчку "привет мир". Наша программа, хоть и простая, но корректно работает с русским текстом.


```
Terminal File Edit View Search Terminal Help
den@den-pc:~/cpp-ex$ ./94
Введите пароль (88): 88
1 - шифрование
2 - расшифровка
3 - выход
1
Введите ключ: 11
Имя файла
напр: "Input_Data"
mmm
Данные для шифровки (Ctrl + D для завершения ввода):
привет мир
Файл успешно зашифрован
Зашифрованные данные сохранены в файле: Encrypted_mmm.txt
den@den-pc:~/cpp-ex$ cat Encrypted_mmm.txt
den@den-pc:~/cpp-ex$ ./94
Введите пароль (88): 88
1 - шифрование
2 - расшифровка
3 - выход
2
Введите ключ: 11
Введите имя файла
напр: Input_Data
Encrypted_mmm
Файл успешно дешифрован
Дешифрованный текст сохранен в Decrypted_Encrypted_mmm.txt
den@den-pc:~/cpp-ex$ cat Decrypted_Encrypted_mmm.txt
привет мир den@den-pc:~/cpp-ex$
```

Рис. 94. Шифрование и расшифровка файла

Пример 95. Простая программа копирования файла. Получаем аргументы командной строки

Как скопировать файл? Если бы мы программировали на C, программа была бы довольно большой. Мы бы посимвольно копировали файл, пока бы не достигли его конца. При использовании C++ нам достаточно использовать потоки, а сама процедура копирования занимает три строчки кода. Нам нужно открыть исходный поток, поток-назначение и из первого все скопировать во второй (лист. 95).

Листинг 95. Копирование файла с помощью потоков

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

int main(int argc, char* argv[])
{
  if(3 != argc)
  {
    return 1;
  }
  ifstream input(argv[1], ios_base::binary | ios_base::in);
  ofstream output(argv[2], ios_base::binary | ios_base::out);
  input >> output.rdbuf();
  return 0;
}
```

Мы не производим каких-либо проверок на существование файла дабы не усложнять нашу программу. Конечно, в реальном мире вы должны будете проверить, существует ли файл, заданный аргументом argv[1] (исходный файл): если нет, вывести соответствующее сообщение и прекратить работу. Также нужно проверить существует ли результирующий файл: если да, то выдать подтверждение перезаписи этого файла. Мы же только проверяем количество аргументов: если оно не равно 3 (сама программа - argv[0] и два переданных ей имени файла), то завершаем работу без каких-либо сообщений.

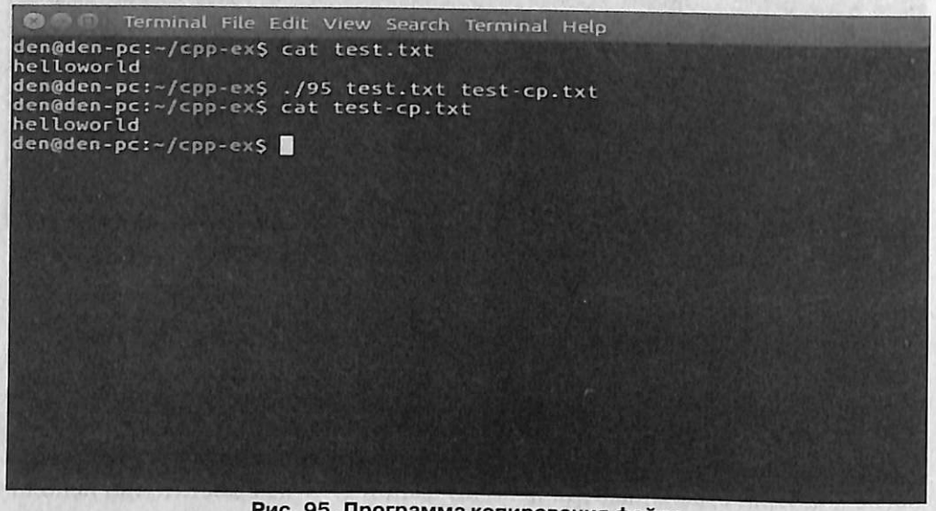


Рис. 95. Программа копирования файла

На рис. 95 показана программа в действии. Она скопировала файл test.txt в test-cp.txt. Помимо всего прочего программа демонстрирует, как получить доступ к параметрам командной строки.

Пример 96. Генератор паролей с записью в файл

Думаю, не нужно говорить о том, что надежный пароль - залог безопасности. В этом примере мы рассмотрим генератор надежных паролей. Конечно, это не просто программа для генерирования случайной последовательности символов заданной длины. Так было бы не интересно. Наш генератор будет запрашивать у пользователя нужную длину и нужное количество паролей, а потом все сгенерированные пароли запишет в файл, имя которого он также запросит у пользователя.

Листинг 96. Генератор паролей

```
#include <iostream>
```



```
#include <algorithm>
#include <time.h>
#include <stdlib.h>
#include <fstream>

using std::cout;
using std::cin;
using std::endl;

class PassGen {

public:
    void displayMessage()
    {
        int passLenght;
        int numOfPasswords;
        char * filename = new char;

        cout << "Введите длину пароля для генерации: ";
        cin >> passLenght;
        cout << "Введите количество паролей для генерации: ";
        cin >> numOfPasswords;
        cout << "Будет сгенерировано паролей: " <<
numOfPasswords << "." << endl;
        cout << endl;
        cout << "Введите имя файла для записи: ";
        cin >> filename;

        std::ofstream outFile(filename);

    // Генерируем numofPasswords паролей
        for (int k = 0; k < numOfPasswords; k++) {
            for (int i = 0; i < passLenght; ++i) {
                numOfChars(passLenght);
    // генерируем пароль
                passGenerator(passLenght);
             // записываем пароль в файл
                outFile << password [i];
            }
            outFile << endl;
        }
        outFile.close();

        cout << "Пароли успешно сгенерированы и записаны в
файл " << filename << "" << endl;
    }
```

```
    // генерирует пароль заданной длины
    void passGenerator(int passLenght)
    {
        password = new char [passLenght];

        for (int i = 0; i < numOfNumbers; ++i) {
            password [i] = char(rand() % 10 + 48);
        }
        for (int i = numOfNumbers; i < numOfNumbers +
numOfBigChars; ++i) {
            password [i] = char(rand() % 26 + 65);
        }
        for (int i = numOfNumbers + numOfBigChars; i <
passLenght; ++i) {
            password [i] = char(rand() % 26 + 97);
        }
         std::random_shuffle(password, password + passLenght);
    }

    void numOfChars(int passLenght)
    {
        numOfSmallChars = rand() % passLenght;
        int charRandEnd = passLenght - numOfSmallChars;
        numOfBigChars = rand() % charRandEnd;
        numOfNumbers = passLenght - numOfSmallChars -
numOfBigChars;
    }

private:
    int numOfSmallChars;
    int numOfBigChars;
    int numOfNumbers;
    char * password;
};

int main()
{
    srand(time(NULL));
    PassGen * pass = new PassGen;
    pass->displayMessage();           // запрашиваем
информацию
    return 0;
}
```


На рис. 96 показано, как работает генератор паролей. Он сгенерировал 3 пароля по 8 символов каждый.

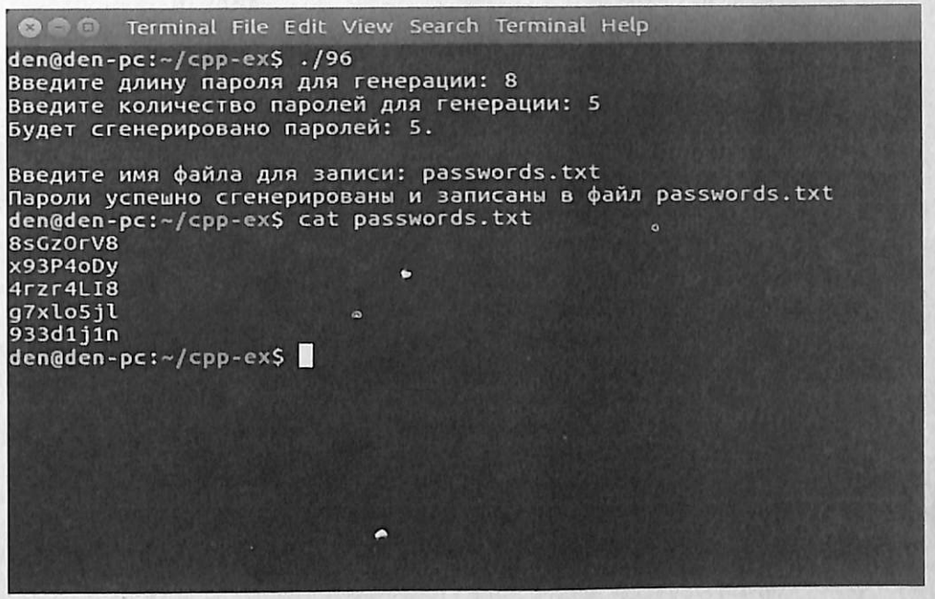


Рис. 96. Генератор в действии

Пример 97. Рекурсивный обход каталога. Команда ls своими руками

Довольно часто в качестве тестового задания (или какой-то практической работы по программированию) встречается рекурсивный обход каталога. В этом примере мы создадим некий аналог команды **ls** в Linux: мы не только выведем имена файлов, но и информацию о них - права доступа и информацию о владельце - так, как это делает команда **ls**. Вы же можете использовать этот пример не только для вывода информации о каталоге (ведь если нам нужно было только вывести содержимое каталога, можно довольно просто вызвать команду **ls** и отобразить результат), но и для рекурсивной обработки всех файлов в каталоге и его подкаталогах, например, для шифрования этих файлов.

Пример 97. Рекурсивный обход каталога с отображением информации о файлах и каталогах

```
#ifndef _VIEW_H
#define  _VIEW_H

#include <stdio.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <iostream>
#include <pwd.h>
#include <grp.h>
#include <errno.h>
#include <unistd.h>

using namespace std;

int myLS(bool l,bool r,const char* path)
{
  DIR* dir = opendir(path);
  if(!dir)
    return 1;

  struct dirent *rd;

  while((rd = readdir(dir)))
  {
    if(!strcmp(rd->d_name,".") || !strcmp(rd->d_name,".."))
    {
      continue;
    }
    struct stat entryInfo;
    char pathName[PATH_MAX+1];

    strncpy(pathName,path,PATH_MAX);
    strncat(pathName,"/",PATH_MAX);
    strncat(pathName,rd->d_name,PATH_MAX);

    if(!lstat(pathName,&entryInfo))
    {
      if(l)
      {
        switch(entryInfo.st_mode & S_IFMT)
        {
          case S_IFDIR:
          {
            cout << "d";        // это каталог
            break;
          }
          case S_IFIFO:
```



```
          {
            cout << "p";        // это канал (pipe)
            break;
          }
          case S_IFSOCK:
          {
            cout << "s";        // сокет
            break;
          }
          case S_IFLNK:
          {
            cout << "l";        // ссылка
            break;
          }
          default:
            cout << "-";        // обычный файл
        }

        //Владелец
        //read
        if(entryInfo.st_mode & S_IRUSR)
          cout << "r";
        else
          cout << "-";
        //write
        if(entryInfo.st_mode & S_IWUSR)
          cout << "w";
        else
          cout << "-";
        //execute
        if(entryInfo.st_mode & S_IXUSR)
          cout << "x";
        else
          cout << "-";

        //Группа
        //read
        if(entryInfo.st_mode & S_IRGRP)
          cout << "r";
        else
          cout << "-";
        //write
        if(entryInfo.st_mode & S_IWGRP)
          cout << "w";
        else
          cout << "-";
```

```
        //execute
        if(entryInfo.st_mode & S_IXGRP)
          cout << "x";
        else
          cout << "-";

        //Прочие
        //read
        if(entryInfo.st_mode & S_IROTH)
          cout << "r";
        else
          cout << "-";
        //write
        if(entryInfo.st_mode & S_IWOTH)
          cout << "w";
        else
          cout << "-";
        //execute
        if(entryInfo.st_mode & S_IXOTH)
          cout << "x";
        else
          cout << "-";

        cout << " ";
        struct passwd* pwd = getpwuid(getuid());
        struct group* grp = getgrgid(pwd->pw_gid);
        cout << " " << grp->gr_name << " " << pwd->pw_name <<
" ";
      }

      if(S_ISDIR(entryInfo.st_mode))
      {
        cout << rd->d_name << endl;
        if(r)
        {
          myLS(l,r,pathName);
        }
      }else
      if(S_ISREG(entryInfo.st_mode))
      {
        cout << rd->d_name;
      }else
      if(S_ISLNK(entryInfo.st_mode))
      {
        char targetName[PATH_MAX+1];
        if(readlink(pathName,targetName,PATH_MAX))
```



```
        {
          cout << rd->d_name << " link: "
              << targetName;
        }
      }else
      if(S_ISSOCK(entryInfo.st_mode))
      {
        cout << rd->d_name;
      }
      cout << endl;
    }
    else
    {
      cout << rd->d_name << " " << strerror(errno) << endl;
      exit(1);
    }
  }
  closedir(dir);
}

int main() {

  myLS(true, false, ".");

  return 0;

}
#endif  /* _VIEW_H */
```

Основная здесь функция **myLS()**. Ей нужно передать три параметра:

1. Показывать (*true*) или нет (*false*) информацию о файле
2. Выполнять (*true*) или нет (*false*) рекурсивный обход
3. Начальный каталог.

В нашем случае программа выводит информацию о файлах текущего каталога (".") и не производит рекурсивный обход (файлов много и все равно все на экран не поместятся).

Рис. 97. Вывод информации о файлах

Пример 98. Программа для объединения двух файлов

Ранее мы писали программу копирования файла. Сейчас мы ее модифицируем так, что она будет объединять два файла в один. Принцип останется тем же, просто программа будет копировать два файла в один. Также программа продемонстрирует обработку ошибок при открытии файлов.

Листинг 98. Объединение двух файлов в один

```
#include<iostream>
#include<fstream>
#include<stdio.h>
#include<stdlib.h>

using namespace std;

int main()
{
   ifstream ifiles1, ifiles2;
   ofstream ifilet;
   char ch, fname1[20], fname2[20], fname3[30];
   cout<<"Введите имя первого файла: ";
   cin >> fname1;
   cout<<"Введите имя второго файла: ";
   cin >> fname2;
```



```
   cout<<"Введите имя результирующего файла: ";
   cin >> fname3;
   ifiles1.open(fname1, ios_base::binary | ios_base::in);
   ifiles2.open(fname2, ios_base::binary | ios_base::in);

      // Если не получилось открыть один из файлов
   if(ifiles1==NULL || ifiles2==NULL)
   {
     perror("Произошла ошибка при открытии исходных
файлов\n");
     exit(EXIT_FAILURE);
   }
   ifilet.open(fname3, ios_base::binary | ios_base::out);
      // Если не получилось открыть результирующий файл
   if(!ifilet)
   {
     perror("Ошибка при создании результирующего файла\n");
     exit(EXIT_FAILURE);
   }

   ifiles1 >> ifilet.rdbuf();
   ifiles2 >> ifilet.rdbuf();

   cout<<"Файлы были объединены в "<<fname3<<" успешно!\n";
   ifiles1.close();
   ifiles2.close();
   ifilet.close();

}
```

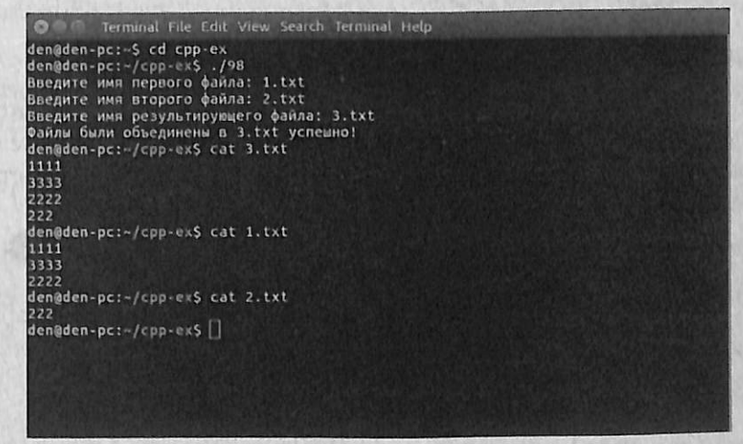


Рис. 98. Объединение двух файлов

Результат работы программы показан на рис. 98. Обратите внимание на содержимое исходных файлов и результирующего.

Пример 99. Сортировка файла, содержащего числовые значения

Пусть у нас есть файл, содержащий числовые значения. Нам нужно отсортировать эти значения по возрастанию или по убыванию. Если бы мы писали на C, то код вряд ли можно было бы назвать компактным. А вот в составе C++ есть библиотечные решения, которые помогут все автоматизировать.

Листинг 99. Сортировка файла с числовыми значениями

```
#include <algorithm>  // для sort
#include <iostream>
#include <fstream>
#include <iterator>  // для итераторв
#include <vector>   // для vector

int main()
{
    std::fstream myfile("1.txt"); // нужно проверить на
существование
    std::vector<double> x(std::istream_
iterator<double>(myfile), {});
    std::sort(x.begin(), x.end());

    for(const auto& elem: x)
        std::cout << elem << "\n";
}
```

Что мы делаем:

1. Открываем файл 1.txt. Это демонстрационная программа, при желании можно сделать, чтобы программа получала имя файла из командной строки (ранее приводился пример обращения к командной строке - пример 99).
2. Читает содержимое файла в вектор x.
3. Выполняет сортировку вектора x.
4. Выводит результат.

Порядок сортировки задается так:

```
std::sort(x.begin(), x.end(), std::greater<double>()); // по
убыванию
```



```
std::sort(x.begin(), x.end(), std::less<double>()); // по
возрастанию
```

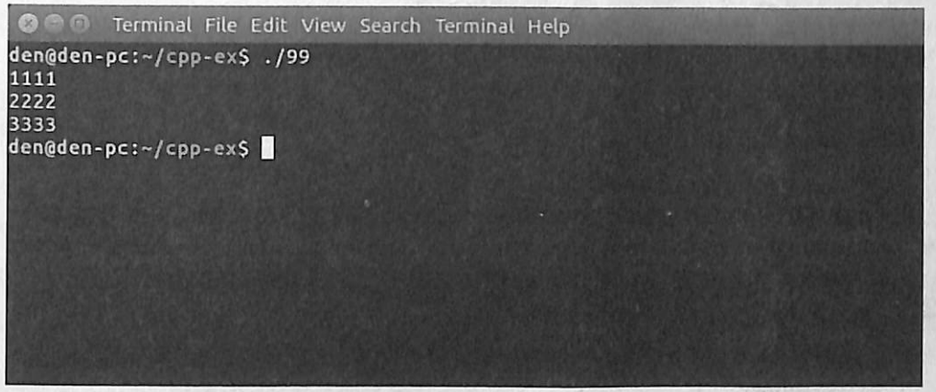


Рис. 99. Программа в действии

Программа производит вывод на **cout**, а это означает, что ее вывод может быть перенаправлен в файл, например:

```
./99 > sorted.txt
```

Пример 100. Подсчет слов или word count на C++

В Linux есть полезная программа, подсчитывающая количество строк в файле. Сейчас мы реализуем аналогичную программу, выполняющую подсчет слов в файле, переданном в качестве первого и единственного параметра командной строки.

Мы будем использовать итераторы C++11 - ведь на дворе 2018-ый год и не очень хочется использовать старые конструкции. К слову, если бы мы писали на диалекте древних, то код был бы похож примерно на это:

```
    inFile.open(fileName.c_str());

    while(!inFile.eof())
    {
        inFile >> word;
        count++;
}
```

Открываем файл, с помощью >> читаем слово в переменную word и увеличиваем счетчик. Как-то так. Но такая программа в 2018 году выглядит как

представитель Мезозойской эры. Именно поэтому мы будем использовать итераторы. Благодаря ним программа станет гораздо компактнее:

```
std::istream_iterator<std::string> in{ infile }, end;
```

Собственно, на этом все. До этого нам нужно прочитать входной файл, а после этого - вывести полученный результат.

Переменная infile() - это поток открытого файла:

```
std::ifstream infile(argv[1]);
```

Для упрощения кода мы не проводим проверку на существование файла, а сразу передаем его имя в конструктор для открытия. Вывести разницу между первым и последним элементом можно с помощью функции distance(). Это и будет количество слов в файле:

```
std::cout << "Количество слов: " << std::distance(in, end)
<< std::endl;
```

Полный код программы приведен в листинге 100.

Листинг 100. Подсчет количества слов в файле

```
#include <fstream>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>
#include <cstdlib>
int main(int argc, char **argv) {
    if (argc < 2) {
        std::cerr << "Использование: count_words <имя_
файла>\n";
        return EXIT_FAILURE;
    }
    std::ifstream infile(argv[1]);
    std::istream_iterator<std::string> in{ infile }, end;
    std::cout << "Количество слов: " << std::distance(in, end)
<< std::endl;
}
```

Обратите внимание, что мы не использовали пространство имен, поэтому явно указываем std:: перед каждым идентификатором, который мы используем из стандартного пространства имен.

Для компиляции программы используйте опцию -std=c++11:

```
g++ 100.cpp -o 100 -std=c++11
```

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.

198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.

Подписано в печать 30.01.2018. Формат 70x100 1/16.

Бумага газетная. Печать офсетная. Объем 16 п. л.

Тираж 1750. Заказ 1465.

Отпечатано с готовых файлов заказчика
в АО "Первая Образцовая типография"
филиал "УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ"
432980, г. Ульяновск, ул. Гончарова, 14.