Оглавление

Введение

Чем отличается начинающий программист от опытного? Начинающий считает, что в килобайте 1000 байт, а опытный – что в километре 1024 метра.

Данная методичка предназначена для студентов, изучающих язык программирования C++ на практических занятиях и лабораторных работах по дисциплине «Технология программирования». В данной работе рассматриваются основные возможности языка C++, используемые в рамках структурного подхода: базовые конструкции языка, массивы, строки, структуры и функции. Также на примерах рассматриваются различные алгоритмы и приёмы написания программ, обсуждаются типичные ошибки и даются полезные практические рекомендации. Методичка не является учебником по языку C++, для этой цели лучше обратиться к книгам [1] или [3].

В работе не рассматриваются вопросы программирования для Windows, все создаваемые программы представляют собой «консольные приложения», использующие только стандартные возможности языка C++. Все примеры программ протестированы с использованием компиляторов Microsoft Visual C++ 2003 и 2005, но должны компилироваться и другими распространёнными компиляторами C++ (GCC, MinGW).

Создание консольного проекта в Microsoft Visual C++ 2005

Для создания нового проекта типа «консольное приложение» выполните следующие действия:

- Выберите в главном меню пункт **File – New – Projects…**
- В открывшемся диалоговом окне **New Project**:
  - в списке **Project types** выберите **Visual C++, Win32**;
  - в списке **Templates** выберите **Win32 Console Application**;
  - в поле **Name** введите название проекта, например **hello**;
  - в поле **Location** выберите папку, в которой будет размещаться папка проекта, например: **D:\Номер-вашей-группы**;
  - нажмите кнопку **OK**.

- После этого запускается мастер приложений, который открывает диалоговое окно **Win32 Application Wizard**:
  - нажмите на ссылку слева **Application settings**;
  - выберите тип проекта **Empty project**;
  - нажмите кнопку **Finish**.

Теперь необходимо добавить в проект новый файл с исходным текстом, для этого:

- Выберите в главном меню пункт **Project – Add New Item…** (или же в окне Solution Explorer нажмите правой кнопкой мыши на пункт Source Files, затем во всплывающем меню выберите пункт Add – New Item…);
- В открывшемся диалоговом окне **Add New Item**:
  - в списке C**ategories** выберите **Visual C++, Code**;
  - в списке **Templates** выберите **C++ File (.cpp)**;
  - в поле **Name** введите название файла, например **hello**;
  - в поле **Location** выберите папку, в которой будет размещаться файл, например: **D:\Номер-вашей-группы\hello** (обычно здесь вводить ничего не требуется, так как Visual Studio сама правильно подставляет папку);
  - нажмите кнопку **Add**.

После этого в редакторе открывается вкладка **hello.cpp**, в которой можно вводить исходный текст программы, для примера введите следующую простейшую программу:

```
// Простейшая программа на C++
#include <iostream>
using namespace std;

// выполнение программы начинается с функции main()
int main()
{
     cout << "Привет, мир!\n";
     return 0;
}
```

Завершив ввод программы, нажмите комбинацию клавиш **Ctrl+S**, чтобы сохранить на диске внесённые изменения. На рисунке 1 показан внешний вид главного окна Visual C++ после завершения ввода программы.

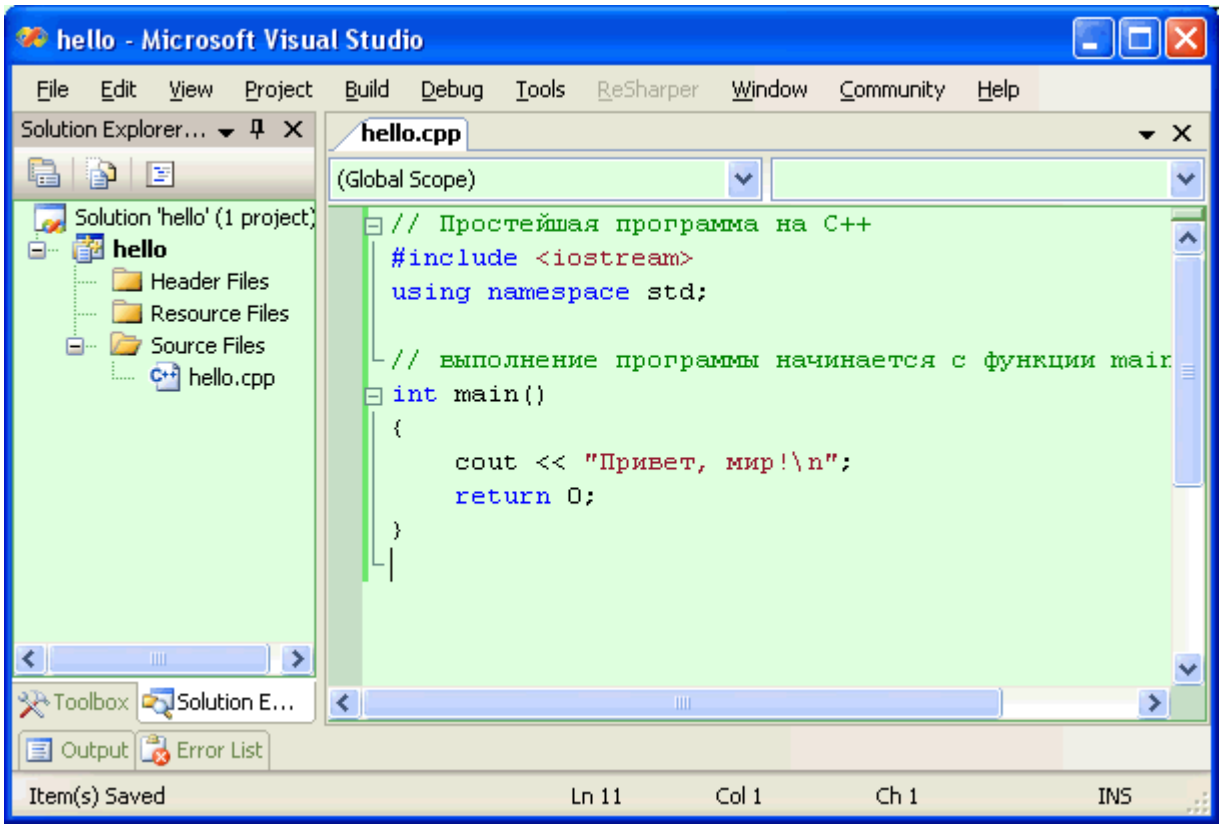


Рис. 1. Внешний вид окна после ввода программы hello

Компиляция, компоновка и запуск проекта

Для компиляции исходного текста выберите пункт меню **Build – Compile (Ctrl-F7)**. Для сборки проекта выберите пункт меню **Build – Build Solution (F6)** или **Build – Build hello**, результат можно посмотреть в окошке Output. Если сборка прошла успешно, в конце должно отображаться сообщение:

```
hello - 0 error(s), 0 warning(s)
=== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ===
```

При запуске программы на выполнение сначала производится компиляция, а потом сборка проекта. Если оба этапа завершены успешно, то происходит запуск, иначе выводится окно с сообщениями об ошибках.

Для запуска программы на исполнение в режиме отладки выберите пункт меню **Debug – Start Debugging (F5** или зелёную стрелку на панели инструментов). Для запуска программы на исполнение без отладки выберите пункт меню **Debug – Start Without Debugging (Ctrl-F5).**

Почему при выполнении данной программы не отображаются русские буквы в сообщении «Привет, мир!» читайте в теме 1.

Тема 1. Оформление программы на языке C++

В первой теме рассмотрены правила оформления программы на языке C++: функция **main()**, директива **#include**, запись операторов и комментариев.

```
// Простейшая программа на C++
#include <iostream>
using namespace std;

// выполнение программы начинается с функции main()
int main()
{
     cout << "Привет, мир!\n";
     return 0;
}
```

Любая программа на языке C++ состоит из одной или более *функций*, содержащих операции, которые должны быть выполнены. Функции в C++ подобны функциям и процедурам языка Паскаль, но не могут быть вложенными. В нашем примере это функция **main().** Обычно функциям можно давать любые имена, но **main** — это особое имя, выполнение любой программы на языке C++ начинается с функции **main().** Для выполнения своей задачи функция **main()** обычно обращается к другим функциям, часть из которых написана программистом и содержится в той же программе, а часть — в библиотеках стандартных функций.

Описание функции состоит из заголовка и тела. Заголовком (или *прототипом)* функции main является «**int main( )**». Это обозначает, что функция **main** не принимает никаких аргументов (внутри круглых скобок ничего нет) и возвращает значение типа **int** (сокращение от integer – целое). Само тело функции записывается внутри фигурных скобках **{ }** и содержит список операторов, которые необходимо выполнить при вызове функции.

Оператор **return** служит для немедленного возврата из функции и передачи возвращаемого значения. В данном примере функция main должна возвращать целое значение, которое одновременно является кодом завершения программы. По соглашению, значение 0 является признаком успешного (предусмотренного) завершения программы.

Для осуществления потокового вывода в языке C++ используется операция << (помещение в поток). Вывод осуществляется в предопределённый объект **cout**, который соответствует стандартному потоку вывода. Другими предопределёнными объектами являются **cin** (стандартный поток ввода) и **cerr** (стандартный поток ошибок). В данном

случае потоковый вывод служит для отображения на экране текстовой строки «Привет, мир!», которая всегда записывается в двойных кавычках (“”). Последовательность **\n** в этой строке является обозначением для языка C++ символа новой строки (символ ‘\’ читается как «обратная косая черта» или backslash). Другие часто используемые последовательности: **\a** – звуковой сигнал, **\b –** возврат курсора на шаг, **\t** – горизонтальная табуляция, **\r** – возврат каретки, \\ – обратная косая черта, \' – апостроф, \" – кавычка, **\0** – символ с кодом 0.

Для обращения к библиотечным классам и функциям в текст программы необходимо включить соответствующие *заголовочные* файлы, для этого служит директива препроцессора **#include**. В данном примере для осуществления операций потокового ввода-вывода используется файл <**iostream**> (угловые скобки используются здесь в качестве ограничителей и не являются частью имени файла). Заголовочные файлы получили своё наименование от английского слова ‘header’ и первоначально имели расширение h/hpp/hxx, хотя теперь они часто вообще не имеют расширения. Любая директива препроцессора в языке C++ начинается с символа # и обрабатывается до начала компиляции программы.

Команда «using namespace std» служит для того, чтобы всё пространство имен **std** сделать открытым для использования. Без этой команды перед каждым объектом из стандартной библиотеки языка C++ программисту пришлось бы ставить префикс **std**. Например, писать **std::cout** вместо просто **cout.**

Разница между **cout << ‘\n’** и **cout << endl** в C++:

- **‘\n’** – вставляет в поток вывода символ перехода на новую строку;
- **endl** – делает то же самое и принудительно освобождает буфер вывода (при работе с файлом в этом случае происходит запись содержимого буфера вывода в файл).

Операция записи на диск занимает значительное время (по сравнению с записью в область памяти), поэтому при выводе больших объёмов информации обязательно надо использовать вариант **cout << ‘\n’.**

Фигурные скобки **{ }** в языке C++ являются аналогом операторов **begin – end** в Паскале и служат для обозначения границ блока. Каждый оператор в программе должен заканчиваться "**;**" (точка с запятой). "**;**" не ставится только после фигурных скобок **{ }** и в конце директив.

Существует два способа записи комментариев: однострочный — от символов // до конца строки и многострочный — между символами **/* */.**

При выполнении данной программы на экран вместо ожидаемой строки «Привет, мир!» на русском языке выводятся какие-то «крякозябы». Дело в том, что при работе консольного приложения в Windows используется режим совместимости с MS-DOS, поэтому все символы

отображаются в кодировке DOS (для России это кодовая страница 866). А в редакторе исходного текста, который используется в Visual Studio, используется кодировка Windows (для России это кодовая страница 1251). Поскольку в этих кодировках русским символам соответствуют разные коды, мы и наблюдаем столь неожиданный эффект.

Какие же существуют решения данной проблемы:

- набирать сообщения только с использованием английских букв;
- набирать сообщения по-русски, но в кодировке 866 (для этого удобно использовать текстовый редактор, который поддерживает разные русские кодировки, например, редактор, встроенный в Far Manager);
- использовать специальную функцию преобразования CharToOem() из <windows.h> для конвертации строк.

Каждый из этих вариантов имеет свои плюсы и минусы, использование конкретного подхода зависит, прежде всего, от того, на какого конечного пользователя рассчитана данная программа.

Тема 2. Условный оператор if

Во второй теме рассмотрены правила объявления переменных, оператор проверки условия **if (**который служит для создания разветвляющихся программ), операции сравнения, потоковые операции ввода-вывода.

```
/* Пример разветвляющейся программы, которая в зависимости
   от введённого значения IQ выдаёт различные заключения
*/
#include <iostream>
using namespace std;

int main()
{
     char name[80];       // массив из 80 символов
     int iq;              // целочисленная переменная

     cout << "Как тебя зовут? ";
     cin >> name;         // ввести с клавиатуры строку
     cout << "Какой у тебя IQ? ";
     cin >> iq;           // ввести целое значение
     cout << "Привет, " << name << "!\n";
     cout << "Твой IQ равен " << iq << "\n";

     if (iq > 150)
          cout << "А не врешь?";
     else if (iq > 100)
          cout << "Да ты интеллектуал!";
     else if (iq < 50)
          cout << "Маловато будет :)";
     else
          cout << "Ты – обычный человек";
     return 0;
}
```

Все переменные в языке C++ должны быть описаны до их первого использования. Описание состоит из *спецификатора типа* и следующего за ним *списка переменных*. Тип **int** обозначает 32-битовое целое число со знаком в диапазоне $\pm 2 \cdot 10^9$. Другие типы: **char** – символ (1 байт), **bool –** логическое значение (может принимать только 2 значения: **true** и **false**) , **float** и **double –** вещественные числа одинарной и двойной точности. Например, описание нескольких вещественных чисел может выглядеть следующим образом: **double x, y, z**.

Имя переменной может быть любым, не совпадающим с зарезервированными ключевыми словами языка C++. Имена составляются

из английских (латинских) букв и цифр, первый символ должен быть буквой. Символ подчеркивания "_" тоже считается буквой, это полезно для удобочитаемости длинных имен. Прописные и строчные буквы считаются различными. Все стандартные функции, типы и ключевые слова языка C++ пишутся со строчной буквы (**int, main, printf, if**).

Для выполнения ввода с клавиатуры используется объект **cin** и операция извлечения из потока >>. Если требуется ввести значения в несколько переменных, их можно сцепить: cin >> var1 >> var2 >> var3.

Условный оператор **if** служит для осуществления ветвления в программе, то есть выполнения различных действий в зависимости от условия. В общем случае, структура оператора следующая:

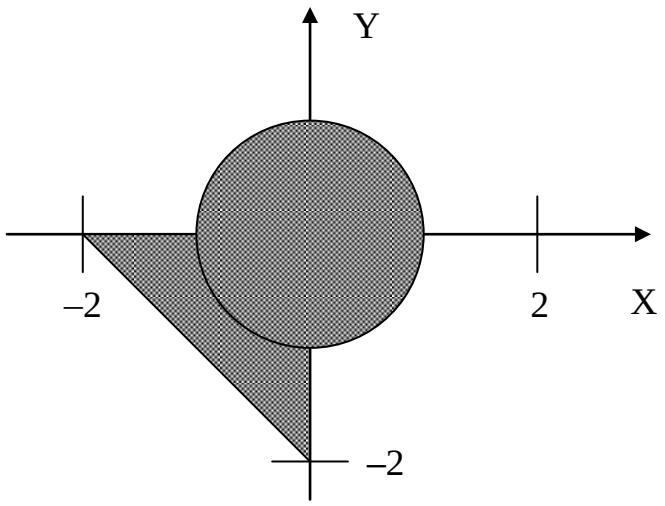
```
if (условие) {
     действие_1;
} else {
     действие_2;
}
```

Если условие, записанное в **( )** следом за **if,** истинно, то выполняется ветвь после **if**, в противном случае выполняется ветвь после **else**, которая необязательна (может отсутствовать).

В качестве выполняемых действий могут выступать любые операторы, в том числе и сам оператор if. В частности, для осуществления последовательности альтернатив может применяться каскадный оператор if (как в примере). Обратите внимание, что в этом случае порядок проверок имеет значение!

Для записи операторов, входящих в тело управляющих конструкций (условных операторов, циклов), принято использовать отступ вправо, чтобы с первого взгляда можно было видеть, какие операторы находятся внутри конструкции. Также рекомендуется писать только один оператор на строке.

Рассмотрим ещё один пример. Надо определить, попадает ли точка с координатами (x, y) внутрь заштрихованной области.



Запишем условие попадания точки в область в виде формул. Область можно описать как круг, пересекающийся с треугольником. Точка может попасть либо в круг, либо в треугольник, либо в их общую часть:

$$\left\{x^2 + y^2 \le 1\right\} \quad или \quad \begin{Bmatrix} x \le 0 \\ y \le 0 \\ y \ge -x - 2 \end{Bmatrix}$$

Тогда программа будет выглядеть следующим образом:

```
// Определение, попадает ли точка внутрь закрашенной области
#include <iostream>
using namespace std;

int main()
{
     double x, y;   // две вещественные переменные
     cout << "Введите значения x и y: ";
     cin >> x >> y; // ввести координаты точки

     if ((x * x + y * y <= 1) ||
          (x <= 0 && y <= 0 && y >= -x - 2)) {
          cout << "Точка попадает в область\n";
     }
     else {
          cout << "Точка не попадает в область\n ";
     }
     return 0;
}
```

Как вы видите, в качестве условия в операторе **if** может применяться сложное выражение, составленное при помощи логических связок **&& (и)**, **|| (или)**. В языке C++ приоритет логических операций ниже, чем приоритет операций сравнения, поэтому круглые скобки **( )** для подвыражений не нужны. Приоритет операции || ниже, чем операции &&, поэтому внутренние скобки тоже не нужны, но в данном примере они используются для удобочитаемости. Логическая связка И применятся, если требуется выполнить несколько условий одновременно. Связка ИЛИ используется, если достаточно выполнить хотя бы одно из нескольких условий. Логическая операция отрицания (НЕ) в языке C++ обозначается восклицательным знаком (!). Помимо логических операций И, ИЛИ существуют ещё *побитовые* операции **& (и), | (или), ^ (исключающее или).** Например, математическое условие принадлежности диапазону 0<x<100 на языке C++ правильно записывается так: **if (0 < x && x < 100)**.

Задания к теме 2

Задание 2–1. [Треугольник]

Даны длины трех отрезков a, b, c. Определить, можно ли из этих трех отрезков построить треугольник. Исходные данные ввести с клавиатуры. Все числа считать вещественными (double).

Задание 2–2. [Високосный год]

Ввести с клавиатуры номер года (положительное целое число) и напечатать, является ли год високосным. Високосные года делятся на 400 (например, 2000) или же делятся на 4, но не делятся на 100 (2004).

Задание 2–3. [Прямоугольник и точка]

Даны координаты точки A (x, y) и координаты левого верхнего (xl, yt) и правого нижнего (xr, yb) углов прямоугольника. Определить, принадлежит ли точка A прямоугольнику. Исходные данные ввести с клавиатуры. Все числа считать целыми.

***Задание 2–4. [Прямоугольник и точка 2]**

Даны координаты точки A (x, y) и координаты двух противоположных (не сказано каких) углов прямоугольника (x1, y1) и (x2, y2). Определить, принадлежит ли точка A прямоугольнику. Исходные данные ввести с клавиатуры. Все числа считать целыми.

Задание 2–5. [График функции]

График функции y = f(x) задан точками, соединенными прямыми отрезками. Вот эти точки: (–oo, 0), (–2, 0), (–1, –1), (1, 1), (2, 0), (+oo, 0). Ввести с клавиатуры вещественное (double) значение x, напечатать соответствующее значение y.

Задание 2–6. [Квадратное уравнение]

Ввести с клавиатуры коэффициенты квадратного уравнения a, b, c. Напечатать корни x1 и x2 или сообщение о том, что корней не существует. Обработать ситуацию, когда a = 0. Все числа считать вещественными (double).

Задание 2–7. [Монеты]

Ввести с клавиатуры целое число S – сумму денег (от 1 до 100). Рассчитать и напечатать минимальное число монет достоинством 1, 2, 5 и 10 рублей, необходимое для выдачи суммы S рублей.

Тема 3. Циклы for и while

В третьей теме представлены циклы **while и for** и некоторые операции (присваивания, инкремента).

```
/* Нахождение суммы целых чисел от 1 до 100 (ответ 5050)
   Вариант с циклом while
*/
#include <iostream>
using namespace std;

int main()
{
     int sum = 0;
     int i = 1;
     while (i <= 100) {  // сокращенная форма записи для:
          sum += i;      // sum = sum + i
          ++i;           // i = i + 1
     }
     cout << "Сумма чисел от 1 до 100 = " << sum;
     return 0;
}
```

Оператор **while** служит для многократного выполнения некоторых действий в программе и называется цикл «пока». В общем случае цикл «пока» записывается следующим образом:

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

Тело цикла, записанное в **{ }** выполняется до тех пор, пока является истинным условие, записанное в **( )** после **while**. Чтобы конструкция не превратилась в «вечный» цикл, нужно следить, чтобы управляющая переменная, влияющая на завершение цикла, как-то изменялась внутри самого цикла.

Операция *присваивания* в языке C++ записывается "=", её не следует путать с операцией *сравнения*, которая обозначается двумя знаками равенства "==". Использование операции присваивания вместо операции сравнения – это одна из распространённых ошибок при переходе с языка Паскаль на C++.

Оператор **++i** демонстрирует операцию **++**, которая обозначает увеличение на единицу (*инкремент*). Можно записать **i = i + 1**, но **++i** более кратко и эффективно. Имеется соответствующая операция уменьшения на единицу **--**. Операции **++** и **--** могут быть как префиксные, так и постфиксные. По одной из версий, язык C++, как преемник языка C, получил своё наименование именно благодаря операции инкремента.

Существуют и другие сокращенные формы записи арифметических операций. В общем случае запись вида:

```
<переменная> <операция>= <выражение>
```

является сокращенной формой для:

```
<переменная> = <переменная> <операция> <выражение>
```

В качестве операции могут выступать арифметические и побитовые логические: –, +, *, /, %, &, |, <<, >> и т.п. Например, выражение prod *= 10 означает, что текущее значение переменной prod надо увеличить в 10 раз.

Переменные могут быть *инициализированы* во время описания константным выражением, например: **int sum = 0.** Вообще, давать начальное значение всем переменным – это хорошая практика. В языке C++ переменные, объявленные внутри функции, являются *локальными* и не инициализируются никаким значением по умолчанию (то есть содержат «мусор»), поэтому перед использованием им обязательно нужно присваивать начальное значение.

```
/* Нахождение суммы целых чисел от 1 до 100 (ответ 5050)
   Вариант с циклом for
*/
#include <iostream>
using namespace std;

int main()
{
     int sum = 0;
     for (int i = 1; i <= 100; ++i) {
          sum += i;
     }
     cout << "Сумма чисел от 1 до 100 = " << sum;
     return 0;
}
```

Оператор **for** является более сложным *оператором цикла* по сравнению с **while**. Он содержит в заголовке три части, разделенные "**;**", каждая из которых может быть *выражением*. Первая часть **int i = 1** выполняется один раз перед входом в сам цикл (*инициализация*). Вторая часть **i <= 100** является *условием*, если оно истинно, то выполняется тело цикла. Затем выполняется третья часть **++i** (*модификация*), после чего снова проверяется условие.

Обратите внимание, что переменная **i** доступна и существует только внутри цикла **for**, поскольку она объявлена в заголовке цикла. А область действия переменной **i**, объявленной перед циклом **while** в предыдущем примере, простирается до конца функции **main.**

В заголовке оператора **for** любая из частей может быть опущена, в предельном случае запись **for(;;)** обозначает "вечный" цикл. Другой общепринятой записью "вечного" цикла является **while(true)**.

Задания к теме 3

Задание 3–1. [Сумма чётных целых чисел от 1 до N]

Ввести с клавиатуры целое положительное число N. Посчитать сумму чётных целых чисел от 1 до N.

Задание 3–2. [Таблица y = sin(x)]

Напечатать таблицу значений функции y = sin(x), где x изменяется от 0 до 180° с шагом 15. Нарисовать разделительные линии и шапку таблицы. Замечание: библиотечная функция sin(x) принимает значение угла в радианах, а не градусах (π радиан = 180°).

Задание 3–3. [Таблица Фаренгейт–Цельсий]

Соответствие между шкалой Цельсия и шкалой Фаренгейта задается формулой: C = 5/9 * (F – 32).

Напечатать таблицу соответствия двух шкал, если градусы Фаренгейта изменяются от 0 до 300 с шагом 20. Нарисовать разделительные линии и шапку таблицы.

Задание 3–4. [Вычисление x^n для целого неотрицательного n]

Написать программу вычисления x^n, где x – вещественное число, а n – целое неотрицательное. Значения x и n ввести с клавиатуры.

***Задание 3–5. [Вычисление x^n]**

Написать программу вычисления x^n, где x – вещественное число, а n – произвольное целое число (в том числе и отрицательное). Значения x и n ввести с клавиатуры.

***Задание 3–6. [Цифры числа]**

Ввести с клавиатуры целое положительное число. Напечатать его цифры через пробел в прямом порядке (без ведущих нулей).

Задание 3–7. [«Счастливые» билетики]

Подсчитать число «счастливых» шестизначных билетиков в диапазоне от 100000 до 999999. Билет считается «счастливым», если сумма его первых трёх цифр совпадает с суммой последних трёх. (Ответ: 50412)

Тема 4. Циклы с неизвестным числом повторений. Вычисление суммы ряда с заданной точностью

Циклы с неизвестным числом повторений требуют специальных действий для своего завершения. Обычно условие завершения цикла достаточно сложно и формируется в процессе вычислений в теле цикла. В этом случае иногда используют условный оператор вместе с оператором **break,** который приводит к немедленному выходу из цикла. В общем случае получается конструкция следующего вида:

```
for/while (условие_1) {
     действия_1;
     if (условие_2)
          break;
     действия_2;
}
```

Здесь условие_2 формируется в результате выполнения действий_1, а в качестве условия_1 может быть просто true (то есть «вечный» цикл).

Ещё один управляющий оператор **continue** пропускает все операторы до конца цикла и передаёт управление сразу на модификацию и перевычисление условного выражения в заголовке цикла, то есть инициирует новый виток цикла.

В качестве примера рассмотрим задачу вычисления значения функции *Ch x* (гиперболический косинус) с помощью бесконечного ряда Тейлора с заданной точностью *ε* (задача и пример кода взяты из [2]):

$$y = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots + \frac{x^{2n}}{2n!} + \cdots$$

Этот ряд сходится при любых значениях *x*. Для достижения заданной точности надо суммировать члены ряда, абсолютная величина которых больше *ε*. До выполнения программы невозможно предсказать, сколько членов ряда потребуется просуммировать. В цикле такого рода есть опасность, что он никогда не завершится – как из-за возможных ошибок в вычислениях, так и из-за ограниченной области сходимости ряда. Поэтому для надежности программы необходимо предусмотреть аварийный выход из цикла с печатью сообщения при достижении некоторого максимально допустимого количества итераций.

Прямое вычисление члена ряда по приведённой выше общей формуле, когда *x* возводится в степень, а затем делится на факториал, имеет два недостатка, которые делают этот способ непригодным. Первый недостаток: большая погрешность вычислений. При возведении в степень и вычислении факториала могут получиться очень большие числа, при делении которых друг на друга произойдёт потеря точности, поскольку

количество значащих цифр, хранимых в ячейке памяти, ограничено. Второй недостаток связан с эффективностью вычислений: при вычислении очередного члена ряда нам уже известен предыдущий, поэтому вычислять каждый член ряда «с нуля» нерационально.

Для уменьшения количества действий следует воспользоваться рекуррентной формулой получения следующего члена ряда через предыдущий: $C_{n+1} = C_n \times T$, где T – некоторый множитель. Отсюда T равно:

$$T = \frac{C_{n+1}}{C_n} = \frac{x^2}{(2n+1)(2n+2)}$$

Ниже приведён текст программы, который решает задачу.

```
// Вычисление суммы ряда с заданной точностью
#include <iostream>
#include <cmath>              // функция fabs()
using namespace std;

const int MAX_ITER = 500;     // максимальное кол-во итераций

int main()
{
     double x, eps;
     cout << "Введите аргумент x и точность: ";
     cin >> x >> eps;
     bool done = true;
     double ch = 1;           // первый член ряда
     double y = ch;           // сумма
     for (int n = 0; fabs(ch) > eps; ++n) {
          ch *= x * x / ((2*n + 1)*(2*n + 2));
          y += ch;
          if (n > MAX_ITER) {
               done = false;
               break;         // немедленный выход из цикла
          }
     }
     if (done) {
          cout << "Значение ch(" << x << ") = " << y;
     }
     else {
          cout << "Ряд расходится!";
     }
     return 0;
}
```

В данной программе используется логический флаг done, который используется для определения того, как завершился цикл. Если по

окончании цикла done = false, это означает, что ряд не сошёлся за MAX_ITER итераций (и, вероятнее всего, он вообще расходится). Если же done = true, то цикл завершился при условии fabs(ch) <= eps, здесь функция fabs() служит для вычисления модуля вещественного числа.

Вместо того, чтобы использовать в тексте программы конкретные неизменяемые числовые значения (их называют «магические числа»), лучше использовать именованные константы (в данном примере это константа MAX_ITER). Во-первых, упрощается модификация программы: при изменении значения константы достаточно его исправить в только одном месте. Во-вторых, осмысленное наименование константы имеет для программиста, читающего программу, гораздо больше смысла, чем числовое значение.

Задания к теме 4

Задание 4–1. [Вычисление квадратного корня методом Герона]

Ввести с клавиатуры вещественные положительные числа a и ε. Необходимо найти квадратный корень из числа a с указанной точностью ε при помощи формулы Герона: $x_0 = 1; \quad x_{n+1} = \left(x_n + \frac{a}{x_n} \right) \Big/ 2$. Результат сравнить со значением, вычисленным библиотечной функцией sqrt().

Задание 4–2. [Вычисление функций разложением в ряд]

Вычислить и напечатать в виде таблицы значения функции, заданной с помощью ряда Тейлора, на интервале от $x_{\text{нач}}$ до $x_{\text{кон}}$ с шагом dx и точностью ε. Каждая строка таблицы должна содержать значение аргумента, значение функции и библиотечное значение.

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \cdots; \quad |x| < \infty$$

$$\frac{\sin x}{x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \cdots; \quad |x| < \infty$$

$$arctg\ x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{2n+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots; \quad |x| \le 1$$

$$arctg\ x = \frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{(2n+1)x^{2n+1}} = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \cdots; \quad x > 1$$

$$\ln x = 2 \sum_{n=0}^{\infty} \frac{(x-1)^{2n+1}}{(2n+1)(x+1)^{2n+1}} = 2\left(\frac{x-1}{x+1} + \frac{(x-1)^3}{3(x+1)^3} + \frac{(x-1)^5}{5(x+1)^5} + \cdots \right); x > 0$$

Тема 5. Цикл do…while. Случайные числа

Цикл **do…while** напоминает цикл **while**, только условие продолжения цикла проверяется не до, а после выполнения операторов внутри цикла. Хотя цикл **do…while** применяется значительно реже цикла **while,** его можно использовать, если действия внутри цикла должен быть выполнены по крайней мере один раз. Ниже приведён пример, в котором программа не выпустит пользователя из цикла, пока он не введёт число в нужном диапазоне.

```
// Ввод с клавиатуры числа в указанном диапазоне
#include <iostream>
using namespace std;

int main()
{
     int n;
     do {
          cout << "Введите число от 1 до 100: ";
          cin >> n;
     } while (n < 1 || n > 100);
     return 0;
}
```

Для генерации *псевдослучайных* чисел в языке C++ используется функция rand() из библиотеки cstdlib. Функция rand() возвращает очередное случайное целое число в диапазоне от 0 до 32767. Чтобы получить число в более узком диапазоне, достаточно использовать над результатом операцию взятие остатка от деления нацело (%). Пример иллюстрирует получение последовательности случайных чисел в диапазоне от 0 до 999 включительно.

```
// Генерация 10 случайных целых чисел
#include <iostream>
#include <cstdlib>       // функции srand(), rand()
#include <ctime>         // функция time()
using namespace std;

int main()
{
     srand (time (0));   // инициализировать генератор
     for (int i = 0; i < 10; ++i) {
          cout << (rand() % 1000) << endl;
     }
     return 0;
}
```

Числа называются *псевдослучайными*, потому что каждое следующее число вычисляется на основании предыдущего по определённому алгоритму, то есть рано или поздно последовательность чисел повторяется. Конструкция srand (time (0)) служит для установки первого случайного числа последовательности в соответствии с текущим значением таймера. Это делается для того, чтобы при каждом новом запуске программы получалась другая последовательность случайных чисел.

Задания к теме 5

Задание 5–1. [Игра «Угадайка»]

Компьютер загадывает случайное число в диапазоне от 0 до 100. Человек вводит своё предположение. В ответ компьютер выдаёт одно из трёх возможных сообщений: «ваше число больше», «ваше число меньше» или «вы угадали». Игра продолжается до тех пор, пока число, загаданное компьютером, не будет угадано.

Задание 5–2. [Игра «Угадайка–2»]

Человек загадывает случайное число в диапазоне от 0 до 100. Компьютер выводит своё предположение. В ответ человек вводит один из 3-х символов, означающих: «моё число больше», «моё число меньше» или «компьютер угадал». Игра продолжается до тех пор, пока число, загаданное человеком, не будет угадано. Реализовать стратегию отгадывания числа компьютером: метод половинного деления или выбор случайного числа.

***Задание 5–3. [Игра «Быки и коровы»]**

Компьютер загадывает случайное 4-разрядное число, все цифры которого различны, а человек пытается его угадать. В ответ на ввод человека, компьютер сообщает, сколько во введённом числе цифр совпадает и стоит на своих местах («быки») и сколько цифр совпадает и не стоит на своих местах («коровы»). Игра заканчивается, когда человек отгадает число полностью (получит 4 «быка»). Например: компьютер загадал число 3270, человек ввел 1207, ответ: 1 бык, 2 коровы.

Тема 6. Использование массивов

В шестой теме рассмотрены правила объявления, инициализации и использования массивов. Массивы – это самая часто используемая программистами неэлементарная структура данных. Поскольку доступ к элементам массива осуществляется через целочисленные индексы, это допускает систематическую обработку всего массива при помощи циклов (особенно цикла for).

```
/* Заполнение массива случайными числами в диапазоне [0..99]
   и поиск минимального и максимального значения в нём
*/
#include <iostream>
#include <cstdlib>       // функции srand(), rand()
#include <ctime>         // функция time()
using namespace std;

const int MSIZE = 10;    // кол-во элементов в массиве

int main()
{
     int mas[MSIZE];     // массив на MSIZE элементов
     srand (time (0));   // инициализировать ГСЧ
     for (int i = 0; i < MSIZE; ++i) {
          mas[i] = rand() % 100;
     }

     // поиск минимального элемента в массиве
     int min = mas[0];   // min – текущий минимум
     for (int i = 1; i < MSIZE; ++i) {
          if (mas[i] < min) {
               min = mas[i];
          }
     }

     // для разнообразия, используем другой подход
     // и пойдем по массиву в обратном направлении
     int max = -1000000; // заведомо малое число
     for (int i = MSIZE - 1; i >= 0; --i) {
          if (mas[i] > max) {
               max = mas[i];
          }
     }
     cout << "Минимальный элемент в массиве " << min;
     cout << "\nМаксимальный элемент в массиве " << max;
     return 0;
}
```

В качестве примера рассмотрим программу поиска минимального и максимального значения в целочисленном массиве. Обратите внимание, что для решения этих двух однотипных задач используются слегка отличающиеся подходы.

Запись **int mas[10]** объявляет массив на 10 целых чисел. Размерность массива записывается после имени переменной в квадратных скобках **[ ]**. Допускаются многомерные массивы, в этом случае каждая следующая размерность записывается в своих **[ ]**. В языке C++ *индексы массива* всегда начинаются с **0**, так что элементами массива являются mas[0], mas[1], ... mas[9]. Выражение **mas[10]** является ошибкой, потому что обозначает обращение к *одиннадцатому* элементу массива, который не существует (точнее, принадлежит совершенно другим данным)!

Так же, как и переменные, массивы могут быть инициализированы при описании списком констант, заключенным в **{ }**. Например, так:

```
int mas[5] = {1, 3, 5, 7, 9};
```

Локальные массивы (объявленные внутри функций) по-умолчанию содержат «мусор» и перед использованием должны быть инициализированы каким-либо значением. Обычно для этой цели используются циклы. Глобальные массивы инициализируются нулём.

При работе с массивами часто встречается операция упорядочения элементов массива по возрастанию или убыванию (сортировка). Для выполнения этой задачи используют функции из <algorithm>. Вот как решается задача сортировки числового массива из n элементов:

```
int mas[n];
sort (mas, mas + n);
```

По умолчанию, числовые массивы сортируется по возрастанию, а строковые – в лексикографическом порядке. Если требуется выполнить сортировку по убыванию, используют конструкцию:

```
sort (mas, mas + n, greater<int>());
```

Третий аргумент – предопределённая функция сравнения по убыванию для массива целочисленного типа. Можно написать и свою собственную функцию сравнения (это делают, например, для массивов структур), которая в случае с целочисленным массивом будет такой:

```
// true, если первый элемент должен стоять в
// упорядоченной посл-ти раньше второго
bool my_greater (const int left, const int right) {
     return left > right;
}
sort (mas, mas + n, my_greater);
```

Задания к теме 6

Задание 6–1. [Работа с массивом]

а) Заполнить массив из 10 элементов случайными целыми числами в диапазоне [–30; 30] и напечатать их в одну строку.

б) Подсчитать количество положительных и отрицательных элементов в массиве.

в) Найти среднее арифметическое значений массива и определить в массиве значение, ближайшее к среднему арифметическому.

г) Подсчитать сумму элементов в массиве между максимальным и минимальным (не включая сами границы).

д) Построить горизонтальную гистограмму: ось нулевых значений проходит в центре экрана сверху вниз, отрицательные значения откладывать влево, положительные – вправо:

```
Значения элементов массива = {-5, -6, 6, -4, 8, -2}
            *****|
           ******|
                 |******
             ****|
                 |********
               **|
```

Задание 6–2. [Упорядочение массива]

а) Заполнить массив из 15 элементов случайными целыми числами в диапазоне [–90; 90] и напечатать их в одну строку.

б) Упорядочить массив так, чтобы вначале шли все отрицательные элементы, затем нули (если они есть), а потом все положительные.

в) Упорядочить массив по возрастанию модулей элементов.

г) Упорядочить массив так, чтобы вначале шли чётные элементы, а потом все нечётные.

***Задание 6–3. [Преобразование «Дата => День недели»]**

Ввести дату в формате "dd mm" (dd от 1 до 31, mm от 1 до 12,). Вывести соответствующий этой дате день недели или выдать сообщение об ошибочных исходных данных (например, даты 30 02 и 10 18 некорректны). 1 января считать понедельником, как в 2007 году, год считать невисокосным. (Подсказка: первоначально надо получить по исходной дате соответствующий номер дня в году.)

Тема 7. Работа со строками

В седьмой теме рассмотрены правила хранения, описания и работы со строками. Хотя в языке C++ для работы со строками рекомендуется использовать специальный класс **string**, большое количество программ (для совместимости) по-прежнему использует представление строк, принятое в языке C: строка представляет собой *массив*, элементами которого являются отдельные символы (типа **char**).

```
/* Определение количества разных типов символов в строке
   Использование массива символов для хранения строки
*/
#include <iostream>
#include <cctype>   // функции is…()
#include <cstring>  // функция strlen()
#include <cstdio>   // функция gets()
using namespace std;

int main()
{
     char s[100];   // массив символов под строку
     gets (s);      // прочесть строку

     int alpha, space, digit;
     alpha = space = digit = 0;
     int len = strlen (s);     // реальная длина строки
     for (int i = 0; i < len; ++i) {
          if (isalpha (s[i]))
               ++alpha;
          else if (isspace (s[i]))
               ++space;
          else if (isdigit (s[i]))
               ++digit;
     }
     cout << "Букв " << alpha;
     cout << "\nПробелов " << space;
     cout << "\nЦифр " << digit;
     return 0;
}
```

Признаком конца строки является *нулевой символ* **'\0'**, поэтому не существует формального ограничения на длину строки. При этом для физического размещения строки требуется на одну ячейку памяти больше, чем планируемое число символов. Для определения количества символов в строке (без учёта нулевого символа) используется функция **strlen(s)** из библиотеки cstring**.**

Не следует путать символьную константу со строкой: **'X'** – это отдельный символ, служащий для получения численного значения; **"X"** – это символьная строка, состоящая из символов **'X'** и **'\0'**.

Основными проблемами, с которыми приходится сталкиваться программистам при использовании этого классического подхода представления строк, являются: необходимость выделения должного количества памяти для хранения строки и правильное размещение *нулевого символа*, ограничивающего строку.

Рекомендуемый класс **string** языка C++ представляет собой *динамическую* строку, которая сама реализует выделение дополнительной памяти при необходимости, и в этом смысле является более удобной и безопасной для программиста. Доступ к отдельным символам строки **string** осуществляется так же, как и к элементам массива, а текущая длина строки определяется через метод **size()**.

Ниже представлен текст программы, который решает ту же самую задачу, но использует класс **string** для хранения строки.

```
/* Определение количества разных типов символов в строке
   Использование класса string для хранения строки
*/
#include <iostream>
#include <cctype>   // функции is…()
#include <string>   // класс string
using namespace std;

int main()
{
     string s;
     getline (cin, s);

     int alpha, space, digit;
     alpha = space = digit = 0;

     for (int i = 0; i < (int) s.size(); ++i) {
          if (isalpha (s[i]))
               ++alpha;
          else if (isspace (s[i]))
               ++space;
          else if (isdigit (s[i]))
               ++digit;
     }

     cout << "Букв " << alpha;
     cout << "\nПробелов " << space;
     cout << "\nЦифр " << digit;
     return 0;
}
```

Для определения типа символа удобно использовать не коды отдельных символов (как это делается в Паскале), а специальные функции из библиотеки cctype. Например, функция **isalpha (ch)** возвращает истину, если символ ch является латинской буквой, другие функции с префиксом is распознают другие типы символов.

Функции **getline (cin, s)** или **gets (s)** используются для чтения строки с клавиатуры целиком, вместе с пробельными символами. **cin >> s** для этой цели не подходит, так как читает строку только до первого пробельного символа (то есть читает слово).

В реальном программировании вместо функции gets (s) используют fgets (s, MAXLEN, stdin) или cin.getline (s, MAXLEN), обе из которых ограничивают количество введённых символов и не допускают переполнение массива s. В данном примере функция gets() используется в силу её простоты.

Строки можно инициализировать при объявлении:

```
char s[100] = "Hasta la vista";
char s[ ] = "Hasta la vista";
string s ("Hasta la vista");  // вызов конструктора
string s = "Hasta la vista";  // присваивание
```

Во втором случае размер массива вычисляется компилятором автоматически, чтобы разместить строку и нуль-терминатор (в данном случае это 15 байт).

Задания к теме 7

Задание 7–1. [Подсчёт английских букв]

Ввести с клавиатуры строку. Определить, сколько в этой строке гласных, согласных, строчных и прописных английских букв (использовать <cctype>).

Задание 7–2. [Подсчёт русских букв]

Ввести с клавиатуры строку. Определить, сколько в этой строке гласных, согласных, строчных и прописных русских букв (использовать голову ☺).

Задание 7–3. [Подсчёт слов]

Написать программу, которая подсчитывает, сколько во введённой с клавиатуры строке содержится слов (словом считается последовательность любых символов, разделенных одним или несколькими пробельными символами: ‘ ’, ‘\t’, ‘\n’).

Тема 8. Функции

В восьмой теме рассмотрены правила объявления вспомогательных функций, передача параметров, рекурсивные и нерекурсивные функции.

Функция – это группа операторов, выполняющая законченное действие. К функции можно обратиться по имени, передать ей значения для обработки и получить результат. Чтобы использовать функцию, не требуется знать, как она работает – достаточно знать, как её вызвать!

Функции служат для упрощения структуры программы: разбив задачу на подзадачи и оформив каждую из них в виде функций, мы действуем в соответствии с основным принципом борьбы со сложностью – «разделяй и властвуй». Передача в функцию параметров делает её универсальной и удобной для многократного использования.

Для вызова функции надо знать только её *интерфейс* (другие названия: *прототип, заголовок*), который состоит из имени функции, типа возвращаемого значения и типов передаваемых аргументов.

В общем случае прототип функции следующий:

```
тип_возвр_значения  имя_функции ( [список_параметров] );
```

Список параметров является необязательным и может быть опущен.

Например, заголовок функции main обычно имеет вид:

```
int main();
```

Это означает, что функция не получает извне никаких аргументов и возвращает значение типа int (целое число).

Другой допустимый вариант заголовка функции main, который используется для работы с аргументами командной строки, таков:

```
int main (int argc, char* argv[]);
```

В этом случае функция получает два параметра: первый – целое число – содержит количество аргументов в командной строке, второй – массив строк – содержит сами эти аргументы (записано как char*, то есть указатель на символы, что в языке С эквивалентно строке).

По соглашению, если программа вызвана без аргументов, то argc == 1 и argv[0] является именем самой программы. Разделителем аргументов в командной строке являются пробельные символы. Все аргументы, полученные функцией main(), являются строками, поэтому перед использованием они могут потребовать преобразований.

Например, если некоторая программа была вызвана так:

```
>prog.exe –x name 123
```

то argc == 4, argv[0] == "prog.exe", argv[1] == "–x", argv[2] == "name", argv[3] == "123".

Заголовок задаёт *объявление* функции. Именно объявления функций содержатся в заголовочных файлах, подключаемых директивой #include. *Определение* функции, кроме заголовка, включает её тело, то есть операторы, заключённые в {}, например:

```
int sum (int a, int b) { // нахождение суммы двух чисел
     return a + b;
}
```

Для возврата результата, вычисленного в функции, служит оператор return. После него указывается выражение, результат вычисления которого и передаётся в точку вызова функции. Функция может содержать несколько операторов возврата, это определяется алгоритмом.

Рассмотрим следующую простейшую функцию:

```
void fun (int x) {
     x = x + 10;
}
```

Если функция не возвращает никакого значения (то есть имеет тип void), то оператор return может в ней отсутствовать. По-умолчанию, параметры передаются в функцию по значению, то есть в стеке создается копия исходного значения, с которым и работает функция. Понятно, что те изменения, которые производятся с переменными-параметрами внутри функции, ни коим образом не могут отразиться на исходных переменных. Это можно проиллюстрировать следующим фрагментом кода:

```
int a = 5;
cout << "before: a = " << a;  // печатает 5
fun (a);
cout << "\nafter: a = " << a; // печатает 5
...
```

Если же мы хотим, чтобы функция изменяла внешнее значение, мы должны передавать параметры по ссылке (обратите внимание на единственное отличие – знак & после типа параметра). В этом случае в стек помещается не значение, а адрес внешней переменной. Вот фрагмент кода:

```
void fun2 (int& x) {
     x = x + 10;
}
...
int a = 5;
cout << "before: a = " << a;  // печатает 5
fun2 (a);
cout << "\nafter: a = " << a; // печатает 15
...
```

Передача по ссылке также применяется в случае, когда функция должна возвращать больше одного значения.

Передача массивов в функцию всегда происходит по ссылке, то есть передаётся адрес первого элемента массива. Это означает, что внутри

функции можно модифицировать элементы внешнего массива. Но задача определения фактического количества элементов в массиве целиком возлагается на программиста. Обычно для этого размер массива передаётся в функцию отдельным параметром:

```
void mas_fun (int mas[], int size);
```

Записывать какое-либо числовое значение внутри [ ] не имеет смысла, потому что компилятор всё равно проигнорирует эту информацию.

Поскольку адрес объекта в языке С можно хранить в указателе, запись int* mas является эквивалентной записи int mas[ ].

В качестве законченного примера рассмотрим программу, которая печатает на экран простые числа, не превосходящие заданного числа N. (Напомним, что простым в математике называется целое число, которое делится без остатка только на себя и на 1, вот начало этого ряда: 2, 3, 5, 7, 11, 13…) Поскольку проверка на простоту числа может встретиться в программе многократно, имеет смысл вынести её в отдельную функцию.

```
// Нахождение простых чисел (неэффективное!)
#include <iostream>
using namespace std;

const int MAXN = 10000;

// функция возвращает true, если число cand простое
bool isprime (const int cand)
{
     // проверяем всевозможные делители
     for (int i = 2; i <= cand - 1; ++i) {
          if (cand % i == 0) { // делится нацело
               return false;
          }
     }
     return true;
}

int main()
{
     // проверяем всех возможных кандидатов
     for (int cand = 2; cand <= MAXN; ++cand) {
          if (isprime (cand)) {
               cout << cand << " ";
          }
     }
     return 0;
}
```

Хотя функция isprime( ) получилась достаточно простой и интуитивно понятной, тем не менее она достаточна неэффективна. Поскольку для расчёта применяется двойной цикл до N, полученная программа обладает квадратичной сложностью (N^2). То есть при MAXN = 10^5, ей понадобится порядка 10^{10} действий, что займёт 3-5 секунд на современных машинах (с тактовой частотой 2-3 ГГц).

Понятно, как можно ускорить программу в 2 раза: достаточно проверять только нечётные числа (среди чётных только число 2 является простым). Но это ещё не всё, оказывается, при проверке делимости достаточно проверять числа не до $cand - 1$, а до $\sqrt{cand}$. Ниже представлен пример программы, которая стала чуть сложнее, но обладает логарифмической сложностью выполнения (NlogN).

```
// Нахождение простых чисел (эффективное!)
#include <iostream>
#include <cmath>          // функция sqrt()
using namespace std;

const int MAXN = 10000;

// функция возвращает true, если число cand простое
bool isprime (const int cand)
{
     if (cand == 2)       // разбираемся с чётными числами
          return true;
     else if (cand % 2 == 0)
          return false;

     int limit = sqrt (double (cand));
     // проверяем только нечётные делители
     for (int i = 3; i <= limit; i += 2) {
          if (cand % i == 0) { // делится нацело
               return false;
          }
     }
     return true;
}

int main()
{
     for (int cand = 2; cand <= MAXN; ++cand) {
          if (isprime (cand)) {
               cout << cand << " ";
          }
     }
     return 0;
}
```

Функция называется *рекурсивной*, если она обращается к самой себе для получения частичного результата. Некоторые алгоритмы реализуются очень кратко и понятно именно в своём рекурсивном варианте. Пример – алгоритм быстрой сортировки (QuickSort).

Классическим примером неэффективного рекурсивного алгоритма является вычисление факториала. Напомним, что факториал положительного числа определяется так:

$n! = 1 \cdot 2 \cdot \ldots \cdot n = n \cdot (n–1)!$

Вот рекурсивная реализация этого алгоритма:

```
int fact (const int n) {
     if (n <= 1)    // защита и от n <= 0
          return 1;
     else
          return n * fact (n - 1);
}
```

При рекурсии надо следить, чтобы проверка граничных случаев предшествовала общему случаю, иначе возможно зацикливание. То же самое можно вычисление можно реализовать более эффективно (нет копирования параметров в стек и вызова функции) через обычный цикл:

```
int fact (const int n) {
     int res = 1;
     for (int i = 2; i <= n; ++i) {
          res *= i;
     }
     return res;
}
```

Задания к теме 8

Задание 8–1. [Работа с массивом через функции]

Выполнить задание 6–1, реализовав все пункты задания через вспомогательные универсальные функции, получающие в качестве параметра имя массива и не зависящие от количества элементов в нём.

Задание 8–2. [Числа Фибоначчи]

Ввести с клавиатуры целое число N (N>0). Написать функции fib_rec() и fib() для вычисления N-го числа Фибоначчи рекурсивным и нерекурсивным методом. Определить максимальное число N, для которого можно правильно вычислить результат, помещающийся в типе int.

Числа Фибоначчи определяются следующим образом:

$f(1) = f(2) = 1;$

$f(n+2) = f(n) + f(n+1),$

вот начало этого ряда: 1, 1, 2, 3, 5, 8, 13…

Тема 9. Работа с файлами

В девятой теме рассмотрены правила работы с файлами в потоковом режиме. Для этого, прежде всего, нужно создать объект соответствующего класса, через который в дальнейшем будет идти работа с файлом. Существует три основных класса для работы с файлами:

- **ifstream** – класс, предназначенный для чтения из файла,
- **ofstream** – класс, предназначенный для записи в файл,
- **fstream** – класс для операций и чтения и записи.

Все потоковые классы описаны в заголовочном файле <fstream>.

Ниже представлен пример посимвольного чтения текстового файла:

```
// Посимвольное чтение файла и вывод его на экран
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
     ifstream fin ("test1.txt");
     if (!fin) {
          cout << "Не открыть файл test1.txt для чтения!";
          return 1;
     }

     char ch;
     while (!fin.feof()) {
          ch = fin.get();
          cout << ch;
     }
     fin.close();
     return 0;
}
```

Прежде чем начать работу с файлом, его необходимо открыть. Это можно сделать либо в конструкторе соответствующего класса, либо при помощи метода **open().** В обоих случаях синтаксис похожий: **ifstream file (path, mode)** или **file.open (path, mode)**. Здесь file – это файловая переменная (имя выбирается программистом, в примере это fin), которая в дальнейшем будет использоваться для работы с файлом.

Аргумент path обозначает имя файла (возможно с путём) в соответствии с правилами Windows. Если указывается путь, то для разделения каталогов надо применять удвоенный ‘\’, например: "C:\\temp\\test1.txt". Если путь не указан, подразумевается текущий каталог (откуда запускается программа).

Аргумент mode – режим открытия файла, указывает, как этот файл можно будет использовать. Этот аргумент – побитовое ИЛИ следующих величин:

- **ios::app** – при записи данные добавляются в конец файла, даже если текущая позиция была перед этим перемещена;
- **ios::ate** – при создании потока текущая позиция помещается в конец файла; однако, в отличие от режима app, запись ведется в текущую позицию;
- **ios::in** – поток создается для ввода; если файл уже существует, он сохраняется;
- **ios::out** – поток создается для вывода;
- **ios::trunc** – если файл уже существует, его прежнее содержимое уничтожается, и длина файла становится равной нулю;
- **ios::binary** – ввод-вывод будет происходить в двоичном виде, по-умолчанию используется текстовое представление данных.

Например, если мы хотим создать новый файл для записи, следующие два способа эквивалентны:

```
ofstream fout ("filename.txt");

fstream  fout ("filename.txt", ios::out | ios::trunc);
```

При попытке открытия файла может произойти ошибка, например, файл может отсутствовать, находиться в другой папке, быть защищённым от записи или заблокированным другой программой, имя файла может быть неправильно написано и т.п. Поэтому проверку успешности открытия надо выполнять обязательно! Простая операция **if (!file)** возвращает **true**, если файл не был открыт или при его открытии произошла ошибка. Наличие в программе подобных проверок сберегает время при поиске ошибок и является признаком хорошего стиля программирования.

Для чтения/записи данных из потоков в основном используются операции << и >>. Нередко применяются и некоторые другие операции:

bool file.feof() – возвращает true, если достигнут конец файла;

int file.get() – читает из файла очередной символ;

getline (file, string s) – прочитать из файла строку целиком в объект типа string (включая и символы перевода строки ‘\n’);

file.getline (char s[ ], int count) – прочитать из файла строку целиком в массив символов, но не более, чем count символов;

file.seekg (int offset, int org) – перемещает текущую позицию чтения/записи на offset байт относительно org; параметр org может принимать одно из следующих значений: ios::beg (от начала файла), ios::cur (от текущей позиции), ios::end (от конца файла);

file.close() – записывает данные на диск и закрывает файл.

Рассмотрим ещё один пример: написать программу, которая определяет, встречается ли в данном текстовом файле заданная последовательность символов (без пробелов). Длина строки текста не превышает 80 символов, текст не содержит переносов слов.

Хотя размер файла заранее неизвестен, но поскольку переносов слов нет, можно ограничиться поиском последовательности в каждой строке отдельно. Для хранения строки из файла достаточно выделить буфер размером в 81 символ (дополнительный символ для завершающего нуля). Для поиска подстроки word в строке line используется функция strstr (line, word), которая возвращает NULL (0), если подстрока отсутствует, или же адрес найденной подстроки.

```
// Поиск подстроки в текстовом файле
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main()
{
     const int LEN = 81;
     char word[LEN], line[LEN];
     cout << "Введите слово: ";
     cin >> word;

     ifstream fin ("test1.txt");
     if (!fin) {
          cout << "Ошибка открытия файла!\n";
          return 1;
     }

     bool found = false;
     while (fin.getline (line, LEN)) {
          if (strstr (line, word)) {
               found = true;
               break;
          }
     }
     if (found) {
          cout << "Слово присутствует!\n";
     }
     else {
          cout << "Слово отсутствует!\n";
     }
     fin.close();
     return 0;
}
```

Задания к теме 9

Во всех заданиях ниже *словом* считается последовательность любых символов, разделённых пробельными символами (‘ ‘, ‘\t’, ‘\n’).

Задание 9–1. [Создание копии файла]

Написать программу, которая создаёт копию содержимого текстового файла «test1.txt». Имя выходного файла ввести с клавиатуры.

Задание 9–2. [Поиск строк с заданным словом]

Написать программу, которая считывает текст из файла и выводит в новый файл только те строки, в которых содержится заданное слово.

Задание 9–3. [Поиск самого длинного слова в файле]

Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в файле.

Задание 9–4. [Перестановка слов в файле]

Написать программу, которая считывает текст из файла и выводит его в новый файл, меняя местами каждые два соседние слова.

Задание 9–5. [Замена пробельных символов]

Написать программу, которая считывает текст из файла и выводит его в новый файл, заменяя последовательность из нескольких пробелов или табуляций ровно на один пробел.

Задание 9–6. [Объединение двух файлов в один]

Написать программу, которая:

а) копирует в выходной файл 2 файла: «test1.txt» и «test2.txt»;

б) нумерует строки в выходном файле – для каждого исходного файла отдельно, начиная с 1;

в) отображает процесс копирования в виде индикатора.

Рекомендуется написать вспомогательную функцию для копирования одного файла.

Задание 9–7. [Перемешивание слов в файле]

Написать программу, которая формирует текстовый файл на основе двух заданных путем "перемешивания" слов этих файлов, то есть помещает в результирующий файл поочередно слова то из первого, то из второго файла.

Тема 10. Использование структур

В десятой теме рассмотрены правила описания и определения структур, обращение к компонентам структур.

Структуры применяют для логического объединения связанных между собой данных. В отличие от массивов, в структуру можно объединять данные различных типов. В языке Паскаль структуры известны под именем записей (record).

Структуры – это механизм для создания новых (пользовательских) типов данных. Например, требуется обработать информацию о расписании работы конференции, и для каждого мероприятия надо знать время, тему, фамилию организатора и количество участников. Поскольку вся эта информация относится к одному событию, логично дать ему имя, чтобы впоследствии можно было к нему обращаться.

```
struct Event {
     int hour, min;
     char theme[100], name[100];
     int num;
};
```

Описание структуры состоит из ключевого слова **struct**, за которым следует имя структуры (**tag**), а затем в **{ }** расположен список описаний компонентов структуры. Описание структуры должно заканчиваться "**;**". В данном случае имя нового типа данных Event, теперь можно создавать переменные этого типа так же, как создаются переменных встроенных типов:

```
Event e1, e2[10];         // структура и массив структур
```

Если структура используется только в одном месте программы, можно совместить описание типа и определение переменных, при этом имя типа можно не указывать:

```
struct {
     int hour, min;
     char theme[100], name[100];
     int num;
} e1, e2[10];
```

Элементы структуры называются *полями*. Поля могут быть любого основного типа, массивом, указателем или другой структурой. Для обращения к полю используется операция «точка»:

```
e1.hour = 12; e1.min++;
strcpy (e1.name, "Билл Гейтс");
```

```
e2[0].num = e1.num + 10;
```

Структуры одного типа можно присваивать друг другу:

```
e1 = e[3]; e[3] = e[0]; e[0] = e1; // Обмен e[0] и e[3]
```

Структуры можно инициализировать перечислением значений её элементов:

```
Event e3 = {11, 30, "Иные и мы", "Неземной И.А.", 25};
```

В объектно-ориентированном программировании внутри структур помимо данных можно хранить и процедуры (они называются методы) их обработки. Фактически классы и структуры в ООП обладают равными возможностями, но у структур все компоненты открытые (public).

Пример: написать программу, которая вводит с клавиатуры слова и подсчитывает, сколько раз каждое из них встретилось, а затем выводит эти слова в порядке убывания частоты их появления. Прекращение приёма слов происходит, когда пользователь введёт слово “quit”.

Рассмотрим два варианта решения задачи:

1) подход в стиле языка C: количество хранимых слов и длина каждого слова ограничены размерами соответствующих массивов;

2) подход в стиле языка C++: количество хранимых слов неограничено (используется динамический массив – vector), длина слова тоже неограничена (используется string).

```
// Частотный словарь (подход в стиле языка C)
#include <algorithm>          // sort()
#include <iostream>
#include <cstring>            // strcmp()
using namespace std;

const int WORDLEN = 80;
const int MAXWORDS = 100;
const int NOTFOUND = -1;

struct Elem {             // Описание структуры из 2-х компонент:
     char str[WORDLEN+1];//    строка для хранения слова
     int cnt;             //    счетчик кол-ва появлений слова
};

Elem list[MAXWORDS];      // Объявление массива структур List
int last = 0;             // Индекс посл. свободного эл-та

// Функция сравнения двух элементов структур
// по убыванию значения поля cnt (для sort())
bool cmp_by_cnt (const Elem& lh, const Elem& rh)
{
     return lh.cnt > rh.cnt;
}
```

```
// Получить очередное слово с клавиатуры и записать в str
// Возвращает true, если это слово не "quit"
bool GetWord (char *str)
{
     cin >> str;
     return strcmp (str, "quit");
}

// Поиск слова str в массиве list
// Возвращает индекс найденного элемента или NOTFOUND
int Search (char *str)
{
     for (int i = 0; i < last; ++i) {
          if (strncmp (str, list[i].str, WORDLEN) == 0)
               return i;
     }
     return NOTFOUND;
}

void PrintList()
{
     for (int i = 0; i < last; ++i)
          cout << "\nСлово <" << list[i].str <<
               "> встретилось " << list[i].cnt << " раз";
}

int main()
{
     char s[200];   // Буфер для хранения очередного слова
     while (GetWord (s)) {
          int pos = Search (s);
          if(pos != NOTFOUND) { // Слово уже встречалось
               list[pos].cnt++;
          }
          else if (last == MAXWORDS) {
               cout << "Список уже полон!";
               break;          // досрочный выход
          }
          else {          // Заносим в список новое слово
               strncpy (list[last].str, s, WORDLEN);
               list[last].str[WORDLEN] = '\0';
               list[last].cnt = 1;
               last++;
          }
     }
     sort (list, list + last, cmp_by_cnt);
     PrintList();
     return 0;
}
```

```
// Частотный словарь (с использованием vector и string)
// Преимущества: список и строка не могут переполниться!
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

const int NOTFOUND = -1;

struct Elem {            // Описание структуры из 2-х компонент:
     string str;         //    строка для хранения слова
     int cnt;            //    счетчик кол-ва появлений слова
};

vector <Elem> list;      // Динамический массив структур

// Функция сравнения двух элементов структур
// по убыванию значения поля cnt (для sort())
bool cmp_by_cnt (const Elem& lh, const Elem& rh)
{
     return lh.cnt > rh.cnt;
}

// Получить очередное слово с клавиатуры и записать в str
// Возвращает true, если это слово не "quit"
// (string& - строка передаётся по ссылке!)
bool GetWord (string& str) {
     cin >> str;
     return str != "quit";
}

// Поиск слова Str в массиве List
// Возвращает индекс найденного элемента или NOTFOUND
// (const string& - строку нельзя модифицировать)
int Search (const string& str)
{
     for (int i = 0; i < list.size(); ++i) {
          if (str == list[i].str)
               return i;
     }
     return NOTFOUND;
}

void PrintList()
{
     for (int i = 0; i < list.size(); ++i)
          cout << "\nСлово <" << list[i].str <<
               "> встретилось " << list[i].cnt << " раз";
}
```

```
int main()
{
     string s;       // Буфер для хранения очередного слова
     while (GetWord (s)) {
          int pos = Search (s);
          if(pos != NOTFOUND) { // Слово уже встречалось
               list[pos].cnt++;
          }
          else {
               Elem tmp;
               tmp.str = s;
               tmp.cnt = 1;
               list.push_back (tmp);
          }
     }
     sort (list.begin(), list.end(), cmp_by_cnt);
     PrintList();
     return 0;
}
```

Несколько дополнительных замечаний:

а) Функция сравнения двух элементов структур (для сортировки) должна быть написана таким образом, чтобы возвращать значение true, если первый элемент должен стоять в упорядоченной последовательности раньше второго. Сами элементы должны передаваться в функцию сравнения как константные ссылки.

```
bool cmp_by_cnt (const Elem& lh, const Elem& rh)
{
     return lh.cnt > rh.cnt;
}
```

б) Для занесения нового слова в список используется функция strncpy (dest_s, src_s, MAXLEN), которая ограничивает число копируемых символов и не допускает переполнения массива dest_s, если строка src_s слишком длинная. Аналогично, для сравнения используется strncmp().

в) В первом варианте для хранения количества слов используется глобальная переменная last, во втором случае эта переменная не нужна, так как vector содержит метод size(), который возвращает число элементов.

Задания к теме 10

Задание 10–1. [Комплексная сортировка]

Модифицировать программу «Частотный словарь» так, чтобы она выводила слова в порядке убывания частоты их появления, а при одинаковой частоте – в алфавитном порядке.

Задание 10–2. [Работа со структурой «Товары»]

Описать структуру PRICE, содержащую следующие поля:

а) название товара;

б) название магазина, в котором продаётся товар;

в) стоимость товара.

Написать программу, выполняющую следующие действия:

а) чтение информации о 10 товарах из текстового файла;

б) вывод полной информации об имеющихся товарах на экран, записи должны быть упорядоченны в алфавитном порядке по названию товара;

в) вывод на экран информации о товарах, стоимость которых превышает заданную (или сообщение об их отсутствии).

Задание 10–3. [Работа со структурой «Студент»]

Описать структуру STUDENT, содержащую следующие поля:

а) фамилия и инициалы студента;

б) номер группы;

в) результаты последней сессии (массив из 5 элементов).

Написать программу, выполняющую следующие действия:

а) чтение информации о 10 студентах из текстового файла;

б) вывод полной информации о студентах на экран, записи должны быть упорядочены по возрастанию среднего балла;

в) вывод на экран информации о студентах, имеющих хотя бы одну оценку выше заданной (или сообщение об их отсутствии).

Список рекомендуемой литературы

1) Павловская Т.А. C/C++. Программирование на языке высокого уровня: Учебник для вузов. СПб.: Питер, 2003. — 464 с.

2) Павловская Т.А., Щупак Ю.А. C/C++. Структурное программирование. Практикум. — СПб.: Питер, 2003. — 240 с.

3) Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином, 2007. — 1099 с.

4) Шилдт Г. Полный справочник по C++, 4-е издание. — М.: Вильямс, 2003. — 800 с.

5) Шилдт Г. Полный справочник по C, 4-е издание. — М.: Вильямс, 2007. — 678 с.

6) Шилдт Г. Справочник программиста по C/C++. — М.: Вильямс, 2006. — 432 с.