ҚАЗАҚСТАН РЕСПУБЛИКАСЫНЫҢ БІЛІМ ЖӘНЕ ҒЫЛЫМ МИНИСТРЛІГІ

Коммерциялық емес жеке кәсіпорын

«АЛМАТЫ ҚАРЖЫ-ҚҰҚЫҚТЫҚ ЖӘНЕ ТЕХНОЛОГИЯЛЫҚ КОЛЛЕДЖІ»

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

Некоммерческое частное учреждение

«АЛМАТИНСКИЙ ФИНАНСОВО-ПРАВОВОЙ И ТЕХНОЛОГИЧЕСКИЙ КОЛЛЕДЖ»

«ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ»

Специальность:1304000-«Вычислительная техника и программное обеспечение»

Преподаватель: Устенов С.Ж.

Алматы 2020 г.

1. Начинаем

В этой главе представлены основные элементы языка: встроенные типы данных, определения именованных объектов, выражений и операторов, определение и использование именованных функций. Мы посмотрим на минимальную законченную C++ программу, вкратце коснемся процесса компиляции этой программы, узнаем, что такое препроцессор, и бросим самый первый взгляд на поддержку ввода и вывода. Мы увидим также ряд простых, но законченных C++ программ.

1.1. Решение задачи

Программы обычно пишутся для того, чтобы решить какую-то конкретную задачу. Например, книжный магазин ведет запись проданных книг. Регистрируется название книги и издательство, причем запись идет в том порядке, в каком книги продаются. Каждые две недели владелец магазина вручную подсчитывает количество проданных книг с одинаковым названием и количество проданных книг от каждого издателя. Этот список сортируется по издателям и используется для составления последующего заказа книг. Нас попросили написать программу для автоматизации этой деятельности.
Один из методов решения большой задачи состоит в разбиении ее на ряд задач поменьше. В идеале, с маленькими задачами легче справиться, а вместе они помогают одолеть большую. Если подзадачи все еще слишком сложны, мы, в свою очередь, разобьем их на еще меньшие, пока каждая из подзадач не будет решена. Такую стратегию называют *пошаговой детализацией* или принципом *"разделяй и властвуй"*. Задача книжного магазина делится на четыре подзадачи:
Прочитать файл с записями о продажах.
Подсчитать количество продаж по названиям и по издателям.
Отсортировать записи по издателям.
Вывести результаты.
Решения для подзадач 1, 2 и 4 известны, их не нужно делить на более мелкие подзадачи. А вот третья подзадача все еще слишком сложна. Будем дробить ее дальше.
3a. Отсортировать записи по издателям.
3b. Для каждого издателя отсортировать записи по названиям.
3c. Сравнить соседние записи в группе каждого издателя. Для каждой одинаковой пары увеличить счетчик для первой записи и удалить вторую.
Эти подзадачи решаются легко. Теперь мы знаем, как решить исходную, большую задачу. Более того, мы видим, что первоначальный список подзадач был не совсем правильным. Правильная последовательность действий такова:
Прочитать файл с записями о продажах.
Отсортировать этот файл: сначала по издателям, внутри каждого издателя - по названиям.

Удалить повторяющиеся названия, наращивая счетчик.
Вывести результат в новый файл.
Результирующая последовательность действий называется *алгоритмом*.
Следующий шаг - перевести наш алгоритм на некоторый язык программирования, в нашем случае - на C++.

1.2. Программа на языке C++

В C++ действие называется *выражением*, а выражение, заканчивающееся точкой с запятой, - *инструкцией*. Инструкция - это атомарная часть C++ программы, которой в программе на C++ соответствует предложение естественного языка. Вот примеры инструкций C++:

```
int book_count = 0;
book_count = books_on_shelf + books_on_order;
cout << "значение переменной book_count: " << book_count;
```

Первая из приведенных инструкций является инструкцией *объявления*. book_count можно назвать *идентификатором*, *символической переменной* (или просто *переменной*) или *объектом*. Переменной соответствует область в памяти компьютера, соотнесенная с определенным именем (в данном случае book_count), в которой хранится значение типа (в нашем случае целого). 0 - это *константа*. Переменная book_count *инициализирована* значением 0.
Вторая инструкция - *присваивание*. Она помещает в область памяти, отведенную переменной book_count, результат сложения двух других переменных - books_on_shelf и books_on_order. Предполагается, что эти две целочисленные переменные определены где-то ранее в программе и им присвоены некоторые значения.
Третья инструкция является инструкцией *вывода*. cout - это выходной поток, направленный на терминал, << - оператор вывода. Эта инструкция выводит в cout - то есть на терминал - сначала символьную константу, заключенную в двойные кавычки ("значение переменной book_count: "), затем значение, содержащееся в области памяти, отведенном под переменную book_count. В результате выполнения данной инструкции мы получим на терминале сообщение:

```
значение переменной book_count: 11273
```

если значение book_count равно 11273 в данной точке выполнения программы.
Инструкции часто объединяются в именованные группы, называемые *функциями*. Так, группа инструкций, необходимых для чтения исходного файла, объединена в функцию readIn(). Аналогичным образом инструкции для выполнения оставшихся подзадач сгруппированы в функции sort(), compact() и print().

В каждой C++ программе должна быть ровно одна функция с именем main(). Вот как может выглядеть эта функция для нашего алгоритма:

```
int main()
{
  readIn();
  sort();
  compact();
  print();

  return 0;
}
```

Исполнение программы начинается с выполнения первой инструкции функции main(), в нашем случае - вызовом функции readIn(). Затем одна за другой исполняются все дальнейшие инструкции, и, выполнив последнюю инструкцию функции main(), программа заканчивает работу.
Функция состоит их четырех частей: типа возвращаемого значения, имени, списка параметров и тела функции. Первые три части составляют *прототип функции*.
Список параметров заключается в круглые скобки и может содержать ноль или более параметров, разделенных запятыми. Тело функции содержит последовательность исполняемых инструкций и ограничено фигурными скобками.
В нашем примере тело функции main() содержит *вызовы* функций readIn(), sort(), compact() и print(). Последней выполняется инструкция

```
return 0;
```

Инструкция return обеспечивает механизм завершения работы функции. Если оператор return сопровождается некоторым значением (в данном примере 0), это значение становится *возвращаемым значением функции*. В нашем примере возвращаемое значение 0 говорит об успешном выполнении функции main(). (Стандарт C++ предусматривает, что функция main() возвращает 0 по умолчанию, если оператор return не использован явно.)
Давайте закончим нашу программу, чтобы ее можно было откомпилировать и выполнить. Во-первых, мы должны определить функции readIn(), sort(), compact() и print(). Для начала вполне подойдут заглушки:

```
void readIn() { cout << "readIn()\n"; }
void sort() { cout << "sort()\n"; }
void compact() { cout << "compact()\n"; }
void print() { cout << "print ()\n"; }
```

Тип void используется, чтобы обозначить функцию, которая не возвращает никакого значения. Наши заглушки не производят никаких полезных

действий, они только выводят на терминал сообщения о том, что были вызваны. Впоследствии мы заменим их на реальные функции, выполняющие нужную нам работу.
Пошаговый метод написания программ позволяет справляться с неизбежными ошибками. Попытаться заставить работать сразу всю программу - слишком сложное занятие.
Имя файла с текстом программы, или исходного файла, как правило, состоит из двух частей: собственно имени (например, bookstore) и расширения, записываемого после точки. Расширение, в соответствии с принятыми соглашениями, служит для определения назначения файла. Файл bookstore.h является *заголовочным файлом* для C или C++ программы. (Необходимо отметить, что стандартные заголовочные файлы C++ являются исключением из правила: у них нет расширения.)
Файл bookstore.c является исходным файлом для нашей C программы. В операционной системе UNIX, где строчные и прописные буквы в именах файлов различаются, расширение .C обозначает исходный текст C++ программы, и в файле bookstore.C располагается исходный текст C++.
В других операционных системах, в частности в DOS, где строчные и прописные буквы не различаются, разные реализации могут использовать разные соглашения для обозначения исходных файлов C++. Чаще всего употребляются расширения .cpp и .cxx: bookstore.cpp, bookstore.cxx.
Заголовочные файлы C++ программ также могут иметь разные расширения в разных реализациях (и это одна из причин того, что стандартные заголовочные файлы C++ не имеют расширения). Расширения, используемые в конкретной реализации компилятора C++, указаны в поставляемой вместе с ним документации.
Итак, создадим текст законченной C++ программы (используя любой текстовый редактор):

```
#include <iostream>
using namespace std;
void readIn() { cout << "readIn()\n"; }
void sort() { cout << "sort()\n"; }
void compact() { cout << "compact()\n"; }
void print() { cout << "print ()\n"; }
int main()
{
  readIn();
  sort();
  compact();
  print();
  return 0;
}
```

Здесь iostream - стандартный заголовочный файл библиотеки ввода/вывода (обратите внимание: у него нет расширения). Эта библиотека содержит информацию о потоке cout, используемом в нашей программе. #include является директивой препроцессора, заставляющей включить в нашу программу текст из заголовочного файла iostream. (Директивы препроцессора рассматриваются в разделе 1.3.)
Непосредственно за директивой препроцессора

```
#include <iostream>
```

следует инструкция

```
using namespace std;
```

Эта инструкция называется директивой using. Имена, используемые в стандартной библиотеке C++ (такие, как cout), объявлены в пространстве имен std и невидимы в нашей программе до тех пор, пока мы явно не сделаем их видимыми, для чего и применяется данная директива. (Подробнее о пространстве имен говорится в разделах 2.7 и 8.5.)
После того как исходный текст программы помещен в файл, скажем prog1.C, мы должны откомпилировать его. В UNIX для этого выполняется следующая команда:

```
$ CC prog1.C
```

Здесь $ представляет собой приглашение командной строки. CC - команда вызова компилятора C++, принятая в большинстве UNIX-систем. Команды вызова компилятора могут быть разными в разных системах.
Одной из задач, выполняемых компилятором в процессе обработки исходного файла, является проверка правильности программы. Компилятор не может обнаружить смысловые ошибки, однако он может найти формальные ошибки в тексте программы. Существует два типа формальных ошибок:
синтаксические ошибки. Программист может допустить "грамматические", с точки зрения языка C++, ошибки. Например:

```
int main( { // ошибка - пропущена ')'
  readIn(): // ошибка - недопустимый символ ':'
  sort();
  compact();
  print();
  return 0 // ошибка - пропущен символ ';' }
```

ошибки типизации. С каждой переменной и константой в C++ сопоставлен некоторый тип. Например, число 10 - целого типа. Строка "hello", заключенная в двойные кавычки, имеет символьный тип. Если функция

ожидает получить в качестве параметра целое значение, а получает символьную строку, компилятор рассматривает это как ошибку типизации. Сообщение об ошибке содержит номер строки и краткое описание. Полезно просматривать список ошибок, начиная с первой, потому что одна-единственная ошибка может вызвать цепную реакцию, появление "наведенных" ошибок. Исправление этой единственной ошибки приведет и к исчезновению остальных. После исправления синтаксических ошибок программу нужно перекомпилировать.
После проверки на правильность компилятор переводит исходный текст в объектный код, который может быть понят и исполнен компьютером. Эту фазу работы компилятора называют *генерацией кода*.
В результате успешной компиляции образуется выполняемый файл. Если запустить выполняемый файл, полученный в результате компиляции нашей программы, на терминале появится следующий текст:

```
readIn()
sort()
compact()
print()
```

В C++ набор основных типов данных - это целый и вещественный числовые типы, символьный тип и логический, или булевский. Каждый тип обозначается своим ключевым словом. Любой объект программы ассоциируется с некоторым типом. Например:

```
int age = 10;
double price = 19.99;
char delimiter = ' ';
bool found = false;
```

Здесь определены четыре объекта: age, price, delimiter, found, имеющие соответственно типы целый, вещественный с двойной точностью, символьный и логический. Каждый объект инициализирован константой - целым числом 10, вещественным числом 19.99, символом пробела и логическим значением false.
Между основными типами данных может осуществляться неявное *преобразование типов*. Если переменной age, имеющей тип int, присвоить константу типа double, например:

```
age = 33.333;
```

то значением переменной age станет целое число 33. (Стандартные преобразования типов, а также общие проблемы преобразования типов рассматриваются в разделе 4.14.)
Стандартная библиотека C++ расширяет базовый набор типов, добавляя к ним такие типы, как строка, комплексное число, вектор, список. Примеры:

```
// заголовочный файл с определением типа string
#include <string>
string current_chapter = "Начинаем";
// заголовочный файл с определением типа vector
#include <vector>
vector<string> chapter_titles(20);
```

Здесь current_chapter - объект типа string, инициализированный константой "Начинаем". Переменная chapter_titles - вектор из 20 элементов строкового типа. Несколько необычный синтаксис выражения

```
vector<string>
```

сообщает компилятору о необходимости создать вектор, содержащий объекты типа string. Для того чтобы определить вектор из 20 целых значений, необходимо написать:

```
vector<int> ivec(20);
```

Никакой язык, никакие стандартные библиотеки не способны обеспечить нас всеми типами данных, которые могут потребоваться. Взамен современные языки программирования предоставляют механизм создания новых типов данных. В C++ для этого служит механизм классов. Все расширенные типы данных из стандартной библиотеки C++, такие как строка, комплексное число, вектор, список, являются классами, написанными на C++. Классами являются и объекты из библиотеки ввода/вывода.
Механизм классов - одна из самых главных особенностей языка C++, и в главе 2 мы рассмотрим его очень подробно.

1.2.1. Порядок выполнения инструкций

По умолчанию инструкции программы выполняются одна за другой, последовательно. В программе

```
int main()
{
  readIn();
  sort();
  compact();
  print();
  return 0;
}
```

первой будет выполнена инструкция readIn(), за ней sort(), compact() и наконец print().
Однако представим себе ситуацию, когда количество продаж невелико: оно

равно 1 или даже 0. Вряд ли стоит вызывать функции sort() и compact() для такого случая. Но вывести результат все-таки нужно, поэтому функцию print() следует вызывать в любом случае. Для этого случая мы можем использовать *условную инструкцию* if. Нам придется переписать функцию readIn() так, чтобы она возвращала количество прочитанных записей:

```
// readIn() возвращает количество прочитанных записей
// возвращаемое значение имеет тип int
int readIn() { ... }

// ...

int main()
{
  int count = readIn();

  // если количество записей больше 1,
  // то вызвать sort() и compact()

  if ( count > 1 ) {
    sort();
    compact();
  }

  if ( count == 0 )
    cout << "Продаж не было\n";
  else
    print();

  return 0;
}
```

Первая инструкция if обеспечивает условное выполнение блока программы: функции sort() и compact() вызываются только в том случае, если count больше 1. Согласно второй инструкции if на терминал выводится сообщение "Продаж не было", если условие истинно, т.е. значение count равно 0. Если же это условие ложно, производится вызов функции print(). (Детальное описание инструкции if приводится в разделе 5.3.)
Другим распространенным способом непоследовательного выполнения программы является итерация, или инструкция *цикла*. Такая инструкция предписывает повторять блок программы до тех пор, пока некоторое условие не изменится с true на false. Например:

```
int main()
{
```

```
  int iterations = 0;
  bool continue_loop = true;
  while ( continue_loop != false )
  {
    iterations++;

    cout << "Цикл был выполнен " << iterations << "раз\n";

    if ( iterations == 5 )
      continue_loop = false;
  }
  return 0;
}
```

В этом надуманном примере цикл while выполняется пять раз, до тех пор пока переменная iterations не получит значение 5 и переменная continue_loop не станет равной false. Инструкция

```
iterations++;
```

увеличивает значение переменной iterations на единицу. (Инструкции цикла детально рассматриваются в главе 5.)

1.3. Директивы препроцессора

Заголовочные файлы включаются в текст программы с помощью *директивы препроцессора* #include. Директивы препроцессора начинаются со знака "диез" (#), который должен быть самым первым символом строки. Программа, которая обрабатывает эти директивы, называется *препроцессором* (в современных компиляторах препроцессор обычно является частью самого компилятора).
Директива #include включает в программу содержимое указанного файла. Имя файла может быть указано двумя способами:

```
#include <some_file.h>
#include "my_file.h"
```

Если имя файла заключено в угловые скобки (<>), считается, что нам нужен некий стандартный заголовочный файл, и компилятор ищет этот файл в предопределенных местах. (Способ определения этих мест сильно различается для разных платформ и реализаций.) Двойные кавычки означают, что заголовочный файл - пользовательский, и его поиск начинается с того каталога, где находится исходный текст программы. Заголовочный файл также может содержать директивы #include. Поэтому иногда трудно понять, какие же конкретно заголовочные файлы включены в данный исходный текст, и некоторые заголовочные файлы могут оказаться

включенными несколько раз. Избежать этого позволяют *условные директивы препроцессора*. Рассмотрим пример:

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H
/* содержимое файла bookstore.h */
#endif
```

Условная директива #ifndef проверяет, не было ли значение BOOKSTORE_H определено ранее. (BOOKSTORE_H - это константа препроцессора; такие константы принято писать заглавными буквами.) Препроцессор обрабатывает следующие строки вплоть до директивы #endif. В противном случае он пропускает строки от #ifndef до # endif.
Директива

```
#define BOOKSTORE_H
```

определяет константу препроцессора BOOKSTORE_H. Поместив эту директиву непосредственно после директивы #ifndef, мы можем гарантировать, что содержательная часть заголовочного файла bookstore.h будет включена в исходный текст только один раз, сколько бы раз ни включался в текст сам этот файл.
Другим распространенным примером применения условных директив препроцессора является включение в текст программы отладочной информации. Например:

```
int main()
{
#ifdef DEBUG
  cout << "Начало выполнения main()\n";
#endif

  string word;
  vector<string> text;

  while ( cin >> word )
  {
#ifdef DEBUG
  cout << "Прочитано слово: " << word << "\n";
#endif
  text.push_back(word);
  }
  // ...
}
```

Если константа DEBUG не определена, результирующий текст программы будет выглядеть так:

```
int main()
{

  string word;
  vector<string> text;

  while ( cin >> word )
  {
    text.push_back(word);
  }
  // ...
}
```

В противном случае мы получим:

```
int main()
{
  cout << "Начало выполнения main()\n";

  string word;
  vector<string> text;

  while ( cin >> word )
  {
    cout << "Прочитано слово: " << word << "\n";
    text.push_back(word);
  }
  // ...
}
```

Константа препроцессора может быть определена в командной строке при вызове компилятора с помощью опции -D (в различных реализациях эта опция может называться по-разному). Для UNIX-систем вызов компилятора с определением препроцессорной константы DEBUG выглядит следующим образом:

```
$ CC -DDEBUG main.C
```

Есть константы, которые автоматически определяются компилятором. Например, мы можем узнать, компилируем ли мы C++ или C программу. Для C++ программы автоматически определяется константа __cplusplus (два подчеркивания). Для стандартного C определяется __STDC__. Естественно, обе константы не могут быть определены одновременно. Пример:

```
#idfef __cplusplus
  // компиляция C++ программы

  extern "C";
  // extern "C" объясняется в главе 7
#endif

int main(int,int);
```

Другими полезными предопределенными константами (в данном случае лучше сказать переменными) препроцессора являются __LINE__ и __FILE__. Переменная __LINE__ содержит номер текущей компилируемой строки, а __FILE__ - имя компилируемого файла. Вот пример их использования:

```
if ( element_count == 0 )
  cerr << "Ошибка. Файл: " << __FILE__
       << " Строка: " << __LINE__
       << "element_count не может быть 0";
```

Две константы __DATE__ и __TIME__ содержат дату и время компиляции. Стандартная библиотека C предоставляет полезный макрос assert(), который проверяет некоторое условие и в случае, если оно не выполняется, выдает диагностическое сообщение и аварийно завершает программу. Мы будем часто пользоваться этим полезным макросом в последующих примерах программ. Для его применения следует включить в программу директиву

```
#include <assert.h>
```

assert.h - это заголовочный файл стандартной библиотеки C. Программа на C++ может ссылаться на заголовочный файл как по его имени, принятому в C, так и по имени, принятому в C++. В стандартной библиотеке C++ этот файл носит имя cassert. Имя заголовочного файла в библиотеке C++ отличается от имени соответствующего файла для C отсутствием расширения .h и подставленной спереди буквой c (выше уже упоминалось, что в заголовочных файлах для C++ расширения не употребляются, поскольку они могут зависеть от реализации).
Эффект от использования директивы препроцессора #include зависит от типа заголовочного файла. Инструкция

```
#include <cassert>
```

включает в текст программы содержимое файла cassert. Но поскольку все имена, используемые в стандартной библиотеке C++, определены в пространстве std, имя assert() будет невидимо до тех пор, пока мы явно не сделаем его видимым с помощью следующей using-директивы:

```
using namespace std;
```

Если же мы включаем в программу заголовочный файл для библиотеки C

```
#include <assert.h>
```

то надобность в using-директиве отпадает: имя assert() будет видно и так . (Пространства имен используются разработчиками библиотек для предотвращения засорения глобального пространства имен. В разделе 8.5 эта тема рассматривается более подробно.)

1.4. Немного о комментариях

Комментарии помогают человеку читать текст программы; писать их грамотно считается правилом хорошего тона. Комментарии могут характеризовать используемый алгоритм, пояснять назначение тех или иных переменных, разъяснять непонятные места. При компиляции комментарии выкидываются из текста программы поэтому размер получающегося исполняемого модуля не увеличивается.
В C++ есть два типа комментариев. Один – такой же, как и в C, использующий символы /* для обозначения начала и */ для обозначения конца комментария. Между этими парами символов может находиться любой текст, занимающий одну или несколько строк: вся последовательность между /* и */ считается комментарием. Например:

```
/*
  * Это первое знакомство с определением класса в C++.
  * Классы используются как в объектном, так и в
  * объектно-ориентированном программировании. Реализация
  * класса Screen представлена в главе 13.
*/
  class Screen {
    /* Это называется телом класса */
    public:
      void home(); /* переместить курсор в позицию 0,0 */
      void refresh ();/* перерисовать экран */
    private:
      /* Классы поддерживают "сокрытие информации" */
      /* Сокрытие информации ограничивает доступ из */
      /* программы к внутреннему представлению класса */
      /* (его данным). Для этого используется метка */
      /* "private:" */
      int height, width;
  }
```

Слишком большое число комментариев, перемежающихся с кодом программы, может ухудшить читаемость текста. Например, объявления переменных width и height в данном тексте окружены комментариями и почти не заметны. Рекомендуется писать развернутое объяснение перед блоком текста. Как и любая программная документация, комментарии должны обновляться в процессе модификации кода. Увы, нередко случается, что они относятся к устаревшей версии.
Комментарии в стиле C не могут быть вложенными. Попробуйте откомпилировать нижеследующую программу в своей системе. Большинство компиляторов посчитают ее ошибочной:

```
#include <iostream>

  /* комментарии /* */ не могут быть вложенными.
  * Строку "не вкладываются" компилятор рассматривает,
  * как часть программы. Это же относится к данной и следующей строкам
  */
  int main() {
    cout << "Здравствуй, мир\n";
  }
```

Один из способов решить проблему вложенных комментариев – поставить пробел между звездочкой и косой чертой:

```
/* * /
```

Последовательность символов */ считается концом комментария только в том случае, если между ними нет пробела.
Второй тип комментариев – однострочный. Он начинается последовательностью символов // и ограничен концом строки. Часть строки вправо от двух косых черт игнорируется компилятором. Вот пример нашего класса Screen с использованием двух строчных комментариев:

```
 /*
   * Первое знакомство с определением класса в C++.
   * Классы используются как в объектном, так и в
   * объектно-ориентированном программировании. Реализация
   * класса Screen представлена в главе 13.
 */
class Screen {
  // Это называется телом класса
  public:
    void home(); // переместить курсор в позицию 0,0
    void refresh (); // перерисовать экран
  private:
    /* Классы поддерживают "сокрытие информации". */
    /* Сокрытие информации ограничивает доступ из */
    /* программы к внутреннему представлению класса */
```

```
    /* (его данным). Для этого используется метка */
    /* "private:" */
    int height, width;
}
```

Обычно в программе употребляют сразу оба типа комментариев. Строчные комментарии удобны для кратких пояснений – в одну или полстроки, а комментарии, ограниченные /* и */, лучше подходят для развернутых многострочных пояснений.

1.5. Первый взгляд на ввод/вывод

Частью стандартной библиотеки C++ является библиотека iostream, которая реализована как иерархия классов и обеспечивает базовые возможности ввода/вывода.
Ввод с терминала, называемый стандартным вводом, “привязан” к предопределенному объекту cin. Вывод на терминал, или стандартный вывод, привязан к объекту cout. Третий предопределенный объект, cerr, представляет собой стандартный вывод для ошибок. Обычно он используется для вывода сообщений об ошибках и предупреждений.
Для использования библиотеки ввода/вывода необходимо включить соответствующий заголовочный файл:

```
#include <iostream>
```

Чтобы значение поступило в стандартный вывод или в стандартный вывод для ошибок используется оператор <<:

```
int v1, v2;
// ...
cout << "сумма v1 и v2 = ";
cout << v1 + v2;
cout << "\n";
```

Последовательность "\n" представляет собой символ перехода на новую строку. Вместо "\n" мы можем использовать
предопределенный *манипулятор* endl.

```
cout << endl;
```

Манипулятор endl не просто выводит данные (символ перехода на новую строку), но и производит сброс буфера вывода. (Предопределенные манипуляторы рассматриваются в главе 20.) Операторы вывода можно сцеплять. Так, три строки в предыдущем примере заменяются одной:

```
cout << "сумма v1 и v2 = " << v1 + v2 << "\n";
```

Для чтения значения из стандартного ввода применяется оператор ввода (>>):

```
string file_name;
// ...
cout << "Введите имя файла: ";
cin >> file_name;
```

Операторы ввода, как и операторы вывода, можно сцеплять:

```
string ifile, ofile;
// ...
cout << "Введите имя входного и выходного файлов: ";
cin >> ifile >> ofile;
```

Каким образом ввести заранее неизвестное число значений? Мы вернемся к этому вопросу в конце раздела 2.2, а пока скажем, что последовательность инструкций

```
string word;
while ( cin >> word )
// ...
```

считывает по одному слову из стандартного ввода до тех пор, пока не считаны все слова. Выражение

```
( cin >> word )
```

возвращает false, когда достигнут конец файла. (Подробнее об этом – в главе 20.) Вот пример простой законченной программы, считывающей по одному слову из cin и выводящей их в cout:

```
#include <iostream>
#include <string>
int main ()
{
  string word;
  while ( cin >> word )
     cout << "Прочитано слово: " << word << "\n";
  cout << "Все слова прочитаны!";
}
```

Вот первое предложение из произведения Джеймса Джойса “Пробуждение Финнегана”:

```
riverrun, past Eve and Adam's
```

Если запустить приведенную выше программу и набрать с клавиатуры данное предложение, мы увидим на экране терминала следующее:

```
Прочитано слово: riverrun,
Прочитано слово: past
Прочитано слово: Eve,
Прочитано слово: and
Прочитано слово: Adam's
Все слова прочитаны!
```

(В главе 6 мы рассмотрим вопрос о том, как убрать знаки препинания из вводимых слов.)

1.5.1. Файловый ввод/вывод

Библиотека iostream поддерживает и файловый ввод/вывод. Все операции, применимые в стандартному вводу и выводу, могут быть также применены к файлам. Чтобы использовать файл для ввода или вывода, мы должны включить еще один заголовочный файл:

```
#include <fstream>
```

Перед тем как открыть файл для вывода, необходимо объявить объект типа ofstream:

```
ofstream outfile("name-of-file");
```

Проверить, удалось ли нам открыть файл, можно следующим образом:

```
if ( ! outfile ) // false, если файл не открыт
cerr << "Ошибка открытия файла.\n"
```

Так же открывается файл и для ввода, только он имеет тип ifstream:

```
ifstream infile("name-of-file");
if ( ! infile ) // false, если файл не открыт
   cerr << "Ошибка открытия файла.\n"
```

Ниже приводится текст простой программы, которая читает файл с именем in_file и выводит все прочитанные из этого файла слова, разделяя их пробелом, в другой файл, названный out_file.

```
#include <iostream>
#include <fstream>
#include <string>
int main()
```

```
{
 ifstream infile("in_file");
 ofstream outfile("out_file");

 if ( ! infile ) {
   cerr << "Ошибка открытия входного файла.\n";
   return -1;
 }

 if ( ! outfile ) {
   cerr << "Ошибка открытия выходного файла.\n";
   return -2;
 }
 string word;
 while ( infile >> word )
    outfile << word << ' ';
 return 0;
}
```