Глава 9

Дополнительные возможности ввода/вывода в C++

В этой главе продолжается изучение системы ввода/вывода C++. Здесь вы узнаете, как создать пользовательские манипуляторы ввода/вывода и как реализовать ввод/вывод в файл. Запомните, система, ввода/вывода C++ очень богата, гибка и многофункциональна. Рассмотрение всех ее возможностей выходит за рамки данной книги, мы коснемся только наиболее важных из них.

Представленная в этой главе система ввода/вывода C++ определена в стандарте Standard C++ и совместима с подавляющим большинством современных компиляторов C++. Если у вас устаревший или несовместимый с современной системой ввода/вывода компилятор, то не все описанные здесь возможности будут вам доступны.

9.1. Создание пользовательских манипуляторов

В дополнение к перегрузке операторов ввода и вывода вы можете создавать свою подсистему ввода/вывода C++, определив для этого собственные манипуляторы. Использование пользовательских манипуляторов важно и ещё двум причинам. Во-первых, можно объединить последовательность нескольких отдельных операций по вводу/выводу в один манипулятор. Например, нередки ситуации, в которых в программе встречаются одинаковые последовательности операций ввода/вывода. Для выполнения такой последовательности можно создать пользовательский манипулятор. Этим вы упрощаете исходную программу и исключаете случайные ошибки. Во-вторых, пользовательский манипулятор может понадобиться, когда необходимо выполнить ввод/вывод на нестандартном оборудовании. Например, вы могли бы воспользоваться манипулятором для посылки управляющих кодов на специальный принтер или в систему оптического распознавания.

Пользовательские манипуляторы - это те элементы языка, которые обеспечивают поддержку в C++ объектно-ориентированного программирования, но они также удобны и для обычных, не объектно-ориентированных программ. Как вы увидите, пользовательские манипуляторы могут помочь сделать любую программу ввода/вывода понятней и эффективней.

Как вы знаете, имеется два базовых типа манипуляторов: те, которые работают с потоками ввода, и те, которые работают с потоками вывода. Однако кроме этих двух категорий имеются еще две: манипуляторы с аргументами и без них. Есть несколько важных отличий в способе создания манипуляторов с параметрами и без. Более того, создание манипуляторов с параметрами является существенно более трудной задачей, чем создание манипуляторов без параметров, и в этой книге не рассматривается. С другой стороны, создать пользовательский манипулятор без параметров достаточно просто, и вы скоро узнаете, как это сделать.

Все манипуляторы без параметров для вывода имеют следующую конструкцию:

ostream ***имя_манипулятора*** **(ostream** ***&поток*****)**
{

// Код программы манипулятора
return *поток*;

Здесь ***имя_манипулятора*** - это имя создаваемого вами пользовательского манипулятора, а ***поток*** - ссылка на вызывающий поток. Возвращаемым значением функции является ссылка на поток. Это необходимо в случае, когда манипулятор является частью большого выражения ввода/вывода. Важно понимать, что хотя у манипулятора и имеется в качестве единственного аргумента ссылка на поток, с которым он работает, но, когда манипулятор используется в операции вывода, его аргумент не используется.

Все манипуляторы без параметров для ввода имеют следующую конструкцию:

istream &*имя_манипулятора* (istream &*поток*)
{
// Код программы манипулятора
return *поток*;

Манипулятор ввода получает в качестве параметра ссылку на поток, для которого он вызывается. Манипулятор должен возвращать этот поток. То, что манипулятор возвращает ссылку на вызывающий поток весьма важно. Если этого не сделать, то ваши манипуляторы нельзя будет использовать в последовательностях операций ввода или вывода.

Примеры

1. Рассмотрим первый простой пример. В следующей программе создается манипулятор setup(), который устанавливает ширину поля вывода, равную 10, точность, равную 4, и символ заполнения *.

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

ostream &setup (ostream &stream)
{
stream.width(10) ;
stream.precision(4);
stream.fill('*') ;
return stream;

int main()
{
cout << setup << 123.123456;
return 0;
```

Как можно заметить, созданный вами манипулятор setup используется в качестве части выражения ввода/вывода точно так же, как это делается с любым встраиваемым манипулятором.

2. Пользовательские манипуляторы не обязательно должны быть сложными. Например, простые манипуляторы atn() и note() обеспечивают простой и удобный способ вывода часто встречающихся слов и фраз.

```
#include <iostream>
using namespace std;
// Внимание:
ostream &atn (ostream &stream)
```

```
{
   stream << "Внимание: ";
   return stream;
}
// Пожалуйста, не забудьте:
ostream &note (ostream &stream)
{
   stream << "Пожалуйста, не забудьте: ";
   return stream;
}

int main()
{
   cout << atn << "Высокое напряжение\n";
   cout << note << "Выключить свет\n";
   return 0;
}
```

Несмотря на простоту, такие манипуляторы оградят вас от необходимости частого набора одних и тех же слов и фраз.

3. В следующей программе создается манипулятор getpass(), который вызывает гудок динамика и затем предлагает ввести пароль:

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

// Простой манипулятор ввода
istream &getpass (istream &stream)
{
   cout << '\a'; // гудок динамика
   cout << "Введите пароль: ";
   return stream;
}

int main()
{
   char pw[80];
   do {
      cin >> getpass >> pw;
   } while (strcmp(pw, "пароль"));
   cout << "Пароль введен верно\n";
   return 0;
}
```

9.2. Основы файлового ввода/вывода

Как было отмечено в предыдущей главе, файловый и консольный ввод/вывод очень близко связаны. Фактически файловый ввод/вывод поддерживается той же иерархией классов, что и консольный ввод/вывод.

Таким образом, все, что вы уже узнали о вводе/выводе, вполне применимо и к файлам. Естественно, что обработка файлов предполагает и кое-что новое.

Для реализации файлового ввода/вывода, необходимо включить в программу заголовок **<fstream>.** В нем определено несколько классов, включая классы **ifstream, ofstream** и **fstream.** Эти классы являются производными от классов **istream** и **ostream**. Вспомните, что

классы **istream** и **ostream**, в свою очередь являются производными от класса **ios**, поэтому классы **ifstream**, **ofstream** и **fstream** также имеют доступ ко всем операциям, определяемым классом **ios** (это обсуждалось в предыдущей главе).

В C++ файл открывается посредством его связывания с потоком. Имеется три типа потоков: ввода, вывода и ввода/вывода. Перед тем как открыть файл, нужно, во-первых, создать поток. Для создания потока ввода необходимо объявить объект типа **ifstream**. Для создания потока вывода – объект типа **ofstream**. Потоки, которые реализуют одновременно ввод и вывод должны объявляться как объекты типа **fstream**. Например, в следующем фрагменте создается один поток для ввода, один поток для вывода и ещё один поток одновременно для ввода и для вывода:

```
ifstream in;        // ввод
ofstream out;       // вывод
fstream io;         // ввод и вывод
```

После создания потока, одним из способов связать его с файлом является функция **open().** Эта функция является членом каждого из трех потоковых классов. Здесь показаны ее прототипы для каждого класса:

void if stream::open(const char *_имя_файла_, openmode _режим_ = ios::in) ;

void ofstream::open(const char *_имя_файла_, openmode _режим_ = ios: : out | ios: : trunc) ;

void fstream::open(const char *_имя_файла_, openmode _режим_ = ios::in | ios:: out) ;

Здесь ***имя_файла*** - имя файла, в которое может входить и спецификатор пути. Значение ***режим*** задает режим открытия файла. Оно должно быть значением типа **openmode**, которое является перечислением, определенным классе **ios**. Значение ***режим*** может быть одним из следующих:

ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc

Вы можете объединить два или более этих значения с помощью оператора OR. Рассмотрим, что означает каждое из этих значений.

Значение **ios::app** вызывает открытие файла в режиме добавления в конец файла. Это значение может применяться только к файлам, открываемым для вывода. Значение **ios::ate** задает режим поиска конца файла при его открытии. Хотя значение **ios::ate** вызывает поиск конца файла, тем не менее, операции ввода/вывода могут быть выполнены в любом месте файла.

Значение **ios::in** задает режим открытия файла для ввода. Значение **ios::out** задает режим открытия файла для вывода.

Значение **ios::binary** вызывает открытие файла в двоичном режиме. По умолчанию все файлы открываются в текстовом режиме. В текстовом режиме имеет место преобразование некоторых символов, например, последовательность символов "возврат каретки/перевод

строки" превращается в символ новой строки. Если же файл открывается в двоичном режиме, такого преобразования не выполняется. Запомните, что любой файл, независимо от того, что в нем содержится - отформатированный текст или необработанные данные - может быть открыт как в текстовом, так и в двоичном режиме. Отличие между ними, только в отсутствии или наличии упомянутого символьного преобразования.

Значение **ios::trunc** приводит к удалению содержимого ранее существовавшего файла с тем же названием и усечению его до нулевой длины. При создании потока вывода с помощью ключевого слова **ofstream** любой ранее существовавший файл с тем же именем автоматически усекается до нулевой длины.

В следующем фрагменте для вывода открывается файл **test:**

```
ofstream mystream;
mystream.open("test");
```

В этом примере параметр ***режим*** функции **open()** по умолчанию устанавливается в значение, соответствующее типу открываемого потока, поэтому нет необходимости указывать его явно.

Если выполнение функции **open()** завершилось с ошибкой, в булевом выражении поток будет равен значению **false**. Этот факт можно использовать для проверки правильности открытия файла с помощью, например, такой инструкции:

```
if(!mystream) {
cout << "Файл открыть невозможно\п";
// программа обработки ошибки открытия файла
}
```

Как правило, перед тем как пытаться получить доступ к файлу, следует проверить результат выполнения функции **open().**

Проверить правильность открытия файла можно также с помощью функции **is_open(),** являющейся членом классов **ifstream, ofstream и fstream**. Ниже показан прототип этой функции:

```
bool is_open();
```

Функция возвращает истину, если поток удалось связать с открытым файлом, в противном случае функция возвращает ложь. Например, в следующем фрагменте проверяется, открыт ли файл, связанный с потоком **mystream**.

```
if(!mystream.i5_open()) {
cout.< "Файл не открыт\n";
// ...
}
```

Хотя использовать функцию **open()** для открытия файла в целом правильно, часто вы этого делать не будете, поскольку у классов **ifstream, ofstream** и **fstream** есть конструкторы, которые открывают файл автоматически. Конструкторы имеют те же параметры, в том числе и задаваемые по умолчанию, что и функция **open().** Поэтому чаще вы будете пользоваться таким способом открытия файла:

```
ifstream mystream("myfile"); // открытие файла для ввода
```

Как уже установлено, если по каким-то причинам файл не открывается, переменная, соответствующая потоку, в условной инструкции будет равна значению **false**. Поэтому, независимо от того, используете ли вы конструктор или явно вызываете функцию **open(),** вам потребуется убедиться в успешном открытии файла путем проверки значения потока.

Для закрытия файла используйте функцию-член **close()**. Например, чтобы закрыть файл, связанный с потоком **mystream**, необходима следующая инструкция:

mystream.close() ;

Функция **close()** не имеет параметров и возвращаемого значения.

С помощью функции **eof()**, являющейся членом класса **ios**, можно определить, был ли достигнут конец файла ввода. Ниже показан прототип этой функции:

bool eof() ;

Функция возвращает истину, если был достигнут конец файла; в противном случае функция возвращает ложь.

После того как файл открыт, очень легко считать из него или записать в него текстовые данные. Просто используйте операторы << и >> так же, как это делалось для консольного ввода/вывода, только замените поток **cin** или **cout** тем потоком, который связан с файлом. Так же, как и операторы << и >> для чтения из файла и записи в файл годятся функции C - **fprintf()** и **fscanf().** Вся информация в файле хранится в том же формате, как если бы она находилась на экране. Следовательно, файл, созданный с помощью оператора <<, представляет из себя файл с отформатированным текстом, и наоборот, любой файл, содержимое которого считывается с помощью оператора >>, должен быть файлом с отформатированным текстом. То есть, как правило, файлы с отформатированным текстом, которые вы будете обрабатывать, используя операторы << и >>, следует открывать в текстовом, а не в двоичном режиме. Двоичный режим больше подходит для неотформатированных файлов, о которых в этой главе будет рассказано позже.

Примеры

1. В представленной ниже программе создается файл для вывода, туда записывается информация, и файл закрывается. Затем файл снова открывается уже в качестве файла для ввода, и записанная ранее информация оттуда считывается:

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

int main()
{
   ofstream fout("test"); // создание файла для вывода
   if(!fout) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   fout << "Привет!\n";
   fout << 100 << ' ' << hex << 100 << endl;
```

```
  fout.close();

  ifstream fin("test"); // открытие файла для ввода
  if(!fin) {
    cout << "Файл открыть невозможно\n";
    return 1;
  }

  char str[80];
  int i;
  fin >> str >> i;
  cout << str << ' ' << i << endl;
  fin.close();
  return 0;
}
```

После того как программа завершится, проанализируйте содержимое файла **test**. Оно будет следующим:

```
Привет!
100 64
```

Как уже установлено, при использовании операторов << и >> для реализации файлового ввода/вывода, информация форматируется так же, как если бы она находилась на экране.

2. Рассмотрим другой пример файлового ввода/вывода. В этой программе введенные с клавиатуры строки считываются и записываются в файл. Программа завершается при вводе знака доллара **$** в качестве первого символа строки. Для использования программы в командной строке задайте имя файла для вывода.

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

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Введите <имя_файла>\n";
    return 1;
  }

  ofstream out(argv[1]); // файл для вывода

  if(!out) {
    cout << "Файл открыть невозможно\n";
    return 1;
  }

  char str[80];
  cout << "Введите строки в файл; для окончания ввода введите $\n";

  do {
    cout << ": ";
    cin >> str;
    out << str << endl;
  } while (*str != '$');
  out.close();
  return 0;
}
```

3. В следующей программе копируется текстовый файл и при этом пробелы превращаются в символы |. Обратите внимание, как для контроля конца файла для ввода используется функция **eof()**. Также обратите внимание, как поток ввода **fin** воспринимает сброс флага **skipws**. Это предотвращает пропуск пробелов в начале строк.

```
// Превращение пробелов в вертикальные линии |
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
   if(argc!=3) {
      cout << "Преобразование <файл_ввода> <файл_вывода>\n";
      return 1;
   }

   ifstream fin(argv[1]);  // открытие файла для ввода
   ofstream fout(argv[2]); // создание файла для вывода

   if(!out) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }
   if(!fin) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   char ch;
   fin.unsetf(ios::skipws); // не пропускать пробелы
   while(!fin.eof()) {
      fin >> ch;
      if(ch==' ') ch = '|';
      if(!fin.eof()) fout << ch;
   }
   fin.close();
   fout.close();
   return 0;
}
```

Между исходной библиотекой ввода/вывода C++ и библиотекой ввода/вывода современного стандарта Standard C++ имеются некоторые отличия, которые необходимо учитывать при модернизации старых программ. Во-первых, в исходной библиотеке ввода/вывода C++ у функции **open()** имеется третий параметр, задающий режим защиты файла. По умолчанию это режим обычного файла. В современной библиотеке C++ указанный параметр не поддерживается.

Во-вторых, при работе со старой библиотекой для открытия потока ввода/вывода **fstream** необходимо явно указать значения режима открытия файла **ios::in** и **ios::out** Значения режима открытия файла по умолчанию не поддерживаются. Это относится как к конструктору класса **fstream**, так и к функции **open().** Например, при работе со старой библиотекой ввода/вывода C++, чтобы открыть файл для ввода и вывода с помощью функции **open(),** необходимо использовать следующую конструкцию:

```
fstream mystream;
mystream.open("test", ios::in | ios::out);
```

В современной библиотеке ввода/вывода C++, если режим открытия не указан, любой объект типа **fstream** автоматически открывает файл ввода и вывода.

И последнее. При работе со старой библиотекой ввода/вывода к ошибке в выполнения функции **open()** ведет значение режима открытия файла, равное **ios::noncreate**, если указанный файл не существует, или равное **ios::noreplace**, если, наоборот, указанный файл уже существует. В стандарте Standard C++ данные значения режима открытия файла не поддерживаются.

9.3. Неформатируемый двоичный ввод/вывод

Хотя текстовые файлы (т. е. файлы, информация в которых представлена в кодах ASCII, - *примеч. пер*.) полезны во многих ситуациях, у них нет гибкости неформатированных двоичных файлов. Неформатированные файлысодержат те самые исходные или "сырые" двоичные данные, которые непосредственно используются вашей программой, а не удобный для восприятия человека текст, данные для которого транслируются операторами << и >>. Поэтому о неформатируемом вводе/выводе иногда говорят как о "сыром" (raw) вводе/выводе. В C++ для двоичных файлов поддерживается широкий диапазон функций ввода/вывода. Эти функции дают возможность точно контролировать процессы считывания из файлов и записи в файлы.

На нижнем уровне двоичного ввода/вывода находятся функции **get()** и **put().** С помощью функции-члена **put()** можно записать байт; а с помощью функции-члена **get()** - считать. Эти функции являются членами всех потоковых классов соответственно для ввода и для вывода. Функции **get()** и **put()** имеют множество форм. Ниже приведены их наиболее часто встречающиеся версии:

istream &get(char &*символ*) ;

ostream &put(char *символ*) ;

Функция **get()** считывает один символ из связанного с ней потока и передает его значение аргументу ***символ***. Ее возвращаемым значением является ссылка на поток. При считывании символа конца файла функция возвратит вызывающему потоку значение **false**. Функция **put()** записывает ***символ*** в поток и возвращает ссылку на поток.

Для считывания и записи блоков двоичных данных используются функции **read()** и **write(),** которые также являются членами потоковых классов соответственно для ввода и для вывода. Здесь показаны их прототипы:

istream &read(char **буфер*, streamsize *число_байт*);

ostream &write(const char **буфер*, streamsize *число_байт*);

Функция **read()** считывает из вызывающего потока столько байтов, сколько задано в аргументе ***число_байт*** и передает их в буфер, определенный указателем ***буфер***. Функция **write()** записывает в соответствующий поток из буфера, который определен указателем буфер, заданное в аргументе ***число_байт*** число байтов. Значения типа **streamsize** представляют собой некоторую форму целого.

Если конец файла достигнут до того, как было считано ***число_байт*** символов, выполнение функции **read()** просто прекращается, а в буфере оказывается столько символов, сколько их было в файле. Узнать, сколько символов было считано, можно с помощью другой функции-члена **gcount(),** прототип которой приведен ниже:

streamsize gcount();

Функция возвращает количество символов, считанных во время последней операции двоичного ввода.

Естественно, что при использовании функций, предназначенных для работы с двоичными файлами, файлы обычно открывают в двоичном, а не в текстовом режиме. Смысл этого понять легко, значение режима открытия файла **ios::binary** предотвращает какое бы то ни было преобразование символов. Это важно, когда в файле хранятся двоичные данные, например, целые, вещественные или указатели. Тем не менее, для файла, открытого в текстовом режиме, хотя в нем содержится только текст, двоичные функции также вполне доступны, но при этом помните о возможности нежелательного преобразования символов.

Примеры

1. В следующей программе на экран выводится содержимое файла. Используется функция **get().**

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

int main(int argc, char *argv[])
{
   char ch;
   if(argc!=2) {
      cout << "Содержимое: <имя_файла>\n";
      return 1;
   }
   ifstream in(argv[1], ios::in | ios::binary);
   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   while(!in.eof()) {
      in.get(ch);
      cout << ch;
   }
   in.close();
   return 0;
}
```

2. В данной программе для записи в файл вводимых пользователем символов используется функция **put().** Программа завершается при вводе знака доллара **$**.

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

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

```
    char ch;

    if(argc!=2) {
        cout << "Запись: <имя_файла>\n";
        return 1;
    }

    ofstream out(argv[1], ios::out | ios::binary);
    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    cout << "Для остановки введите символ $\n";
    do {
        cout << ": ";
        cin.get(ch);
        out.put(ch);
    } while (ch!='$');

    out.close();
    return 0;
}
```

Обратите внимание, что для считывания символов из потока **cin** в программе используется функция **get().** Это предотвращает игнорирование начальных пробелов.

3. В следующей программе для записи строки и числа типа **double** в файл **test** используется функция **write()**:

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

int main()
{
    ofstream out("test", ios::out | ios::binary);
    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    double num = 100.45;
    char str[] = "Это проверка";
    out.write((char *) &num, sizeof(double));
    out.write(str, strlen(str));
    out.close();
    return 0;
}
```

Приведение типа к **(char*)** при вызове функции **write()** необходимо, ее буфер вывода не определен как символьный массив. Поскольку в C++ осуществляется строгий контроль типов, указатель на один тип не преобразуется автоматически в указатель на другой тип.

4. В следующей программе для считывания из файла, созданного в программе примера 3, используется функция **read()**:

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

```
int main()
{

   ifstream in("test", ios::in | ios::binary);

   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   double num;
   char str[80];
   in.read((char *) &num, sizeof(double));
   in.read(str, 15);
   str[14] = '\0';
   cout << num << ' ' << str;
   in.close();
   return 0;
}
```

Как и в программе из предыдущего примера, приведение типов внутри функции **read()** необходимо, поскольку в C++ указатель одного типа автоматически не преобразуется в указатель другого типа.

5. В следующей программе сначала массив **double** записывается в файл, а затем считывается обратно. Кроме того, отображается число считанных символов.

```
// Демонстрация работы функции gcount()
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   ofstream out("test", ios::out | ios::binary);

   if(!out) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   double nums[4] = {1.1, 2.2, 3.3, 4.4};
   out.write((char *) nums, sizeof(nums));
   out.close();

   ifstream in("test", ios::in | ios::binary);
   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   in.read((char *) &nums, sizeof(nums));
   int i;
   for(i=0; i<4; i++)
      cout << nums[i] << ' ';
   cout << '\n';
   cout << in.gcount() << " символов считано\n";
   in.close();
   return 0;
}
```

9.4. Дополнительная информация о функциях двоичного ввода/вывода

Кроме представленной ранее формы, функцию **get()** можно перегрузить ещё несколькими способами. Здесь показаны прототипы трех наиболее часто перегружаемых форм:

istream &get(char *_буфер_, streamsize _число_байт_);

istream &get(char *_буфер_, streamsize _число_байт_, char _ограничитель_) ;

int get ();

Первая функция **get()** считывает символы в массив, определенный указателем ***буфер***, до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт - 1***, либо не встретился символ конца файла. В конце массива, заданного указателем ***буфер***, функция **get()** помещает ноль. Если в потоке ввода встретится символ новой строки, он не извлекается, а остается в потоке до следующей операции ввода.

Вторая функция **get()** считывает символы в массив, определенный указателем ***буфер***, до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт - 1***, либо не встретился символ, заданный параметром ***ограничитель***, либо не встретился символ конца файла. В конце массива, заданного указателем ***буфер***, функция **get()** помещает ноль. Если в потоке ввода встретится символ ***ограничитель***, он не извлекается, а остается в потоке до следующей операции ввода.

Третья функция **get()** возвращает из потока следующий символ. Она возвращает символ EOF, если достигнут конец файла. Эта форма функции **get()** напоминает функцию **getc()** языка С.

Другой функцией для реализации ввода является функция **getline()**. Эта функция - член всех потоковых классов ввода. Ниже показаны ее прототипы:

istream &getline(char *_буфер_, streamsize _число_байт_) ;

istream &getline(char *буфер, streamsize _число_байт_, char _ограничитель_) ;

Первая функция считывает символы в массив, обозначенный указателем ***буфер***, до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт - 1***, либо не встретился символ новой строки, либо не встретился символ конца файла. В конце массива, заданного указателем ***буфер***, функция **getline()** помещает ноль. Если в потоке ввода встретится символ новой строки, он извлекается, но не помещается в массив.

Вторая функция считывает символы в массив, обозначенный указателем ***буфер***, до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт - 1***, либо не встретился символ ***ограничитель***, либо не встретился символ конца файла. В конце массива, заданного указателем ***буфер***, функция **getline()** помещает ноль. Если в потоке ввода встретится символ ***ограничитель***, он извлекается, но не помещается в массив.

Как можно заметить, обе версии функции **getline()** фактически тождественны версиям **get*(буфер, число_байт)*** и **get*(буфер, число_байт, ограничитель)*** функции **get().** Обе считывают символы из потока ввода и помещают их в массив, обозначенный указателем

буфер до тех пор, пока либо не считано ***число_байт - 1*** символов, либо не встретился символ ограничитель или символ конца файла. Отличие между функциями **get()** и **getlihe()** в том, что функция **getline()** считывает и удаляет из потока ввода символ ***ограничитель***, а функция **get()** - нет.

Используя функцию **peek(),** можно получить следующий символ из потока ввода без его удаления из потока. Функция является членом потоковых классов ввода и имеет следующий прототип:

int peek() ;

Функция возвращает следующий символ из потока или, если достигнут конец файла, символ EOF.

С помощью функции **putback(),** являющейся членом потоковых классов ввода, можно возвратить последний считанный из потока символ обратно в поток. Ниже показан прототип этой функции:

istream &putback (char *c*) ;

Здесь ***c*** - это последний считанный из потока символ.

При выполнении вывода данные не сразу записываются на связанное с потоком физическое устройство, а информация временно сохраняется во внутреннем буфере. Только после заполнения буфера его содержимое переписывается на диск. Однако вызов функции **flush()** вызывает физическую запись информации на диск до заполнения буфера. Ниже показан прототип функции **flush()**, являющейся членом потоковых классов вывода:

ostream &flush() ;

Вызовы функции **flush()** оправданы при работе в неблагоприятной обстановке (например, в ситуациях, когда часто случаются сбои по питанию).

Примеры

1. Как вы знаете, при использовании для считывания строки оператора >> считывание прекращается при встрече первого разделительного символа. При наличии в строке пробелов такое считывание становится невозможным. Однако, как показано в программе, с помощью функции **getline()** можно решить эту проблему:

```
// Использование функции getline() для считывания строки с пробелами
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   char str[80];
   cout << "Введите ваше имя: ";
   cin.getline(str, 79);
   cout << str << '\n';
   return 0;
}
```

В данном случае ограничителем для функции **getline()** является символ новой строки. Это делает выполнение функции **getline()** очень похожей на выполнение стандартной функции **gets().**

2. В реальном программировании особенно полезны функции **peek()** и **putback()**. Они позволяют упростить управление, когда неизвестен тип вводимой в каждый конкретный момент времени информации. Следующая программа иллюстрирует это. В ней из файла считываются строки либо целые. Строки и целые могут следовать в любом порядке.

```
// Демонстрация работы функции peek()
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main()
{
   char ch;
   ofstream out("test", ios::out | ios::binary);

   if(!out) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   char str[80], *p;

   out << 123 << "this is a test" << 23;
   out << "Hello there!" << 99 << "sdf" << endl;
   out.close();

   ifstream in("test", ios::in | ios::binary);

   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   do {
      p = str;
      ch = in.peek(); // выяснение типа следующего символа
      if(isdigit(ch)) {
         while(isdigit(*p=in.get())) p++; // считывание целого
         in.putback(*p); // возврат символа в поток
         *p = '\0'; // заканчиваем строку нулем
         cout << "Целое: " << atoi(str);
      }
      else if(isalpha(ch)) { // считывание строки
         while(isalpha(*p=in.get())) p++;
         in.putback(*p); // возврат символа в поток
         *p = '\0'; // заканчиваем строку нулем
         cout << "Строка: " << str;
      }
      else in.get(); // пропуск
      cout << '\n';
   } while(!in.eof());

   in.close();
   return 0;
}
```

9.5. Произвольный доступ

В системе ввода/вывода C++ *произвольный доступ (random access)* реализуется с помощью функций **seekg()** и **seekp()**, являющихся соответственно потоковыми функциями ввода и вывода. Здесь показаны их основные формы:

istream &seekg(off_type *смещение*, seekdir *задание*);

ostream &seekp(off_type *смещение*, seekdir *задание*);

Здесь **off_type** - это целый тип данных, определенный в классе **ios** и совместимый с максимальным правильным значением, которое способен хранить параметр ***смещение***. Тип **seekdir** - это перечисление, определенное в классе **ios** и содержащее следующие значения:

Значение	Смысл
ios::beg	Поиск с начала файла
ios:: cur	Поиск от текущей позиции в файле
ios::end	Поиск с конца файла

Система ввода/вывода C++ управляет двумя указателями, связанными с файлом. Первый - это *указатель считывания (get pointer*), который задаёт следующее место в файле, откуда будет вводиться информация. Второй - это *указатель записи (put pointer),* который задает следующее место в файле, куда будет выводиться информация. При каждом вводе или выводе соответствующий указатель последовательно продвигается дальше. Однако с помощью функций **seekg()** и **seekp()** возможен непоследовательный доступ к файлу.

Функция **seekg()** устанавливает указатель считывания соответствующего файла в позицию, отстоящую на величину ***смещение*** от заданного места ***задание***. Функция **seekp()** устанавливает указатель записи соответствующего файла в позицию, отстоящую на величину ***смещение*** от заданного места ***задание***.

Как правило, файлы, доступные для функций **seekg()** и **seekp()**, должны открываться в режиме-операции для двоичных файлов. Таким образом предотвращается возможное неожиданное преобразование символов внутри файла.

Определить текущую позицию каждого из двух указателей можно с помощью функций:

pos_type tellg();

pos_type tellp();

Здесь **pos_type** - это целый тип данных, определенный в классе **ios** и способный хранить наибольшее возможное значение указателя.

Для перемещения файловых указателей считывания и записи на позицию, заданную возвращаемыми значениями функций **tellg()** и **tellp(),** используются перегруженные версии функций **seekg()** и **seekp().** Прототипы этих функций представлены ниже:

istream &seekg(pos_type *позиция*);

ostream &seekp(pos type *позиция*);

Примеры

1. В следующей программе показана работа функции **seekp().** Она позволяет заменить в файле заданный символ. Укажите в командной строке имя файла, затем номер байта в файле, который вы хотите изменить, и, наконец, новый символ для замены. Обратите внимание: файл открывается для операций чтения и записи.

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

int main(int argc, char *argv[])
{

   if(argc!=4) {
      cout << "Замена: <файл> <байт> <символ>\n";
      return 1;
   }

   fstream out(argv[1], ios::in | ios::out | ios::binary);
   if(!out) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   out.seekp(atoi(argv[2]), ios::beg);
   out.put(*argv[3]);
   out.close();
   return 0;
}
```

2. В следующей программе функция **seekg()** используется для установки указателя считывания в заданную позицию внутри файла и для вывода содержимого файла, начиная с этой позиции. Имя файла и позиция начала считывания задаются в командной строке.

```
// Демонстрация работы функции seekg()
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

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

   if(argc!=3) {
      cout << "Поиск: <файл> <позиция>\n";
      return 1;
   }

   ifstream in(argv[1] , ios::in | ios::binary);
   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }
```

```
    in.seekg(atoi(argv[2]), ios::beg);

    while(!in.eof()) {
        in.get(ch);
        cout << ch;
    }

    in.close();
    return 0;
}
```

9.6. Контроль состояния ввода/вывода

В системе ввода/вывода C++ поддерживается информация о состоянии после каждой операции ввода/вывода. Текущее состояние потока ввода/вывода, которое хранится в объекте типа **iostate**, является перечислением, определенным в классе **ios** и содержащим следующие члены:

Название	Значение
goodbit	Ошибок нет
eofbit	Достигнут конец файла
failbit	Имеет место нефатальная ошибка
badbit	Имеет место фатальная ошибка

В устаревших компиляторах флаги состояния ввода/вывода хранятся как целые, а не как объекты типа **iostate**.

Имеются два способа получения информации о состоянии ввода/вывода. Во-первых, можно вызвать функцию **rdstate(),** являющуюся членом класса **ios**. Прототип этой функции:

iostate rdstate();

Функция возвращает текущее состояние флагов ошибки. Как вы, вероятно, догадываетесь, глядя на приведенный выше список флагов, функция **rdstate()** возвращает флаг **goodbit** при отсутствии какой бы то ни было ошибки. В противном случае она возвращает флаг ошибки.

Другим способом определения того, имела ли место ошибка, является использование одной или нескольких следующих функций - членов класса **ios**:

bool bad() ;
bool eof() ;
bool fail();
bool good() ;

Функция **eof()** уже обсуждалась. Функция **bad()** возвращает истину, если установлен флаг **badbit**. Функция **fail()** возвращает истину, если установлен флаг **failbit**. Функция **good()** возвращает истину при отсутствии ошибок. В противном случае функции возвращают ложь.

После появления ошибки может возникнуть необходимость сбросить это состояние перед тем, как продолжить выполнение программы. Для этого используется функция **c1ear(),** являющаяся членом класса **ios**. Ниже приведен прототип этой функции:

void clear(iostate *флаги* = ios::goodbit);

Если параметр ***флаги*** равен **goodbit** (значение по умолчанию), то сбрасываются флаги всех ошибок. В противном случае переменной ***флаги*** присваиваются значения тех флагов, которые вы хотите сбросить.

Примеры

1. В следующей программе иллюстрируется выполнение функции **rdstate().** Программа выводит на экран содержимое текстового файла. При наличии ошибки функция сообщает об этом с помощью функции **checkstatus().**

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

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
   if(argc!=2) {
      cout << "Содержимое: <имя_файла>\n";
      return 1;
   }

   ifstream in(argv[1]);

   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   char c;
   while(in.get(c)) {
      cout << c;
      checkstatus(in);
   }

   checkstatus(in); // контроль финального состояния
   in.close();
   return 0;
}

void checkstatus(ifstream &in)
{
   ios::iostate i;
   i = in.rdstate();

   if(i & ios::eofbit)
      cout << "Достигнут EOF\n";
   else if(i & ios::failbit)
      cout << "Нефатальная ошибка ввода/вывода\n";
   else if(i & ios::badbit)
      cout << "Фатальная ошибка ввода/вывода\n";
```

```
}
```

Эта программа всегда будет выводить сообщение по крайней мере об одной ошибке. После окончания цикла **while** последний вызов функции **checkstatus(),** как и ожидается, выдаст сообщение о достижении конца файла (символа EOF).

2. В следующей программе с помощью функции **good()** файл проверяется на наличие ошибки:

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

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

   if(argc!=2) {
      cout << "Содержимое: <имя_файла>\n";
      return 1;
   }

   ifstream in(argv[1]);

   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   while(!in.eof()) {
      in.get(ch);
      // контроль ошибки
      if(!in.good() && !in.eof()) {
         cout << "Ошибка ввода/вывода ... прерывание работы\n";
         return 1;
      }
      cout << ch;
   }

   in.close();
   return 0;
}
```

9.7. Пользовательский ввод/вывод и файлы

В предыдущей главе вы изучили перегрузку операторов ввода и вывода для создаваемых вами классов. При этом рассматривался только консольный ввод/вывод. Однако поскольку все потоки C++ одинаковы, то одинаково перегруженная, например, функция вывода, может использоваться без каких-либо изменений для вывода как на экран, так и в файл. Это одна из наиболее важных и полезных возможностей ввода/вывода в C++.

Как установлено в предыдущей главе, перегруженные функции ввода/вывода так же, как и манипуляторы ввода/вывода могут использоваться с любым потоком. Если вы "жестко" зададите конкретный поток в функции ввода/вывода, область ее применения, несомненно, будет ограничена только этим потоком. Следует, по возможности, разрабатывать такие функции ввода/вывода, чтобы они могли одинаково работать с любыми потоками.

Примеры

1. В следующей программе относительно класса **coord** перегружаются операторы << и >>. Обратите внимание, что одни и те же оператор-функции можно использовать для вывода как на экран, так и в файл.

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

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

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

   return stream;
}

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

   return stream;
}

int main()
{
   coord o1(1, 2), o2(3, 4);
   ofstream out("test");

   if(!out) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   out << o1 << o2;

   out.close();

   ifstream in("test");

   if(!in) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   coord o3(0, 0), o4(0, 0);
   in >> o3 >> o4;
   cout << o3 << o4;
   in.close();
   return 0;
}
```

2. Все манипуляторы ввода/вывода подходят и для файлового ввода/вывода. Например, в представленной ниже переработанной версии одной из программ этой главы, тот манипулятор, который выводит информацию на экран, используется и для ее записи в файл.

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

// Внимание:
ostream &atn(ostream &stream)
{
   stream << "Внимание: ";
   return stream;
}

// Пожалуйста, не забудьте:
ostream &note (ostream &stream)
{
   stream << "Пожалуйста, не забудьте: ";
   return stream;
}

int main()
{
   ofstream out("test");

   if(!out) {
      cout << "Файл открыть невозможно\n";
      return 1;
   }

   // вывод на экран
   cout << atn << "Высокое напряжение \n";
   cout << note << "Выключить свет\n";

   // вывод в файл
   out << atn << "Высокое напряжение\n";
   out << note << "Выключить свет\n";

   out.close();
   return 0;
}
```