# Стандартная библиотека C++ для работы с потоками

## *Заголовочные файлы и классы C++, связанные с потоками*

Ввод-вывод в программах на языке C++ рекомендуется выполнять с помощью специальных классов — *потоков ввода-вывода*, реализованных в нескольких стандартных заголовочных файлах, из которых основными являются *iostream* и *fstream*. Указанные файлы предоставляют средства для организации ввода-вывода из стандартного устройства (консольный ввод-вывод) и из файлов. Кроме того, потоки можно использовать для форматированного вывода данных в текстовые строки и чтения данных из текстовых строк; для этого предназначены строковые потоки ввода-вывода, реализованные в заголовочных файлах *sstream* и *strstream*.

Все классы-потоки связаны отношениями наследования и образуют достаточно сложную иерархию.

Базовым предком всех классов-потоков является класс *ios_base*. От этого класса порождается базовый *шаблонный* класс *basic_ios*, от которого, в свою очередь, порождается большой набор других шаблонных классов. Поскольку в программах обычно используются конкретные *специализации* шаблонов, сразу отметим, что двумя основными специализациями шаблона *basic_ios* являются *ios* (базовый класс для всех потоков, основанных на однобайтных символьных данных char) и *wios* (базовый класс для всех потоков, основанных на много-байтных символьных данных *wchar_t*). Эти специализации, наряду с набором *функций-манипуляторов*, определены в заголовочном файле *ios*.

Для простоты ограничимся иерархией, основанной на классе *ios*. От этого класса порождаются базовый поток ввода *istream* и базовый поток вывода *ostream*. Определяется также базовый поток ввода-вывода *iostream*, который порождается сразу от двух классов: *istream* и *ostream*. Классы *istream*, *ostream* и *iostream* определены в заголовочном файле *iostream*. Именно в этих классах определены все необходимые операции и функции ввода-вывода; потомки этих классов лишь изменяют реализации этих функций в зависимости от природы источника или приемника данных.

Первая группа потомков этих классов обеспечивает ввод-вывод из файлов (классы из этой группы определены в заголовочном файле *fstream*):

- *ifstream* — файловый поток ввода (потомок *istream*),
- *ofstream* — файловый поток вывода (потомок *ostream*),
- *fstream* — файловый поток ввода-вывода (потомок *iostream*).

Вторая группа потомков этих классов обеспечивает ввод-вывод из объектов типа *string* (классы из этой группы определены в заголовочном файле *sstream*):

- *istringstream* — строковый поток ввода (потомок *istream*),
- *ostringstream* — строковый поток вывода (потомок *ostream*),
- *stringstream* — строковый поток ввода-вывода (потомок *iostream*).

Подчеркнем, что все указанные классы являются *специализациями* соответствующих шаблонных классов, однако при их использовании в программе это обстоятельство никак не проявляется, и можно считать, что данные классы являются обычными потоковыми классами, ориентированными на потоки, содержащие однобайтовые символы *char*.

☑ Заметим, что имена соответствующих *шаблонных* классов получаются из имен описанных выше классов присоединением префикса *basic_* (например, *basic_iostream*, *basic_ifstream*, *basic_ostringstream* и т. д.), а классов-специализаций, ориентированных на потоки *многобайтовых символов wchar_t*, получаются из имен описанных выше классов присоединением префикса *w* (например, *wiostream*, *wifstream*, *wostringstream* и т. д.).

## *Обработка ошибок ввода-вывода*

Стандартная библиотека потокового ввода-вывода для языка C++ ориентирована, прежде всего, на следующую схему работы с потоком ввода: «читать данные из потока, пока очередная операция чтения не закончится неудачей». Поэтому никакая ошибка, связанная с потоками, не приводит к аварийному завершению программы. Вместо этого изменяется состояние соответствующего потока, *после чего все операции, связанные с этим потоком, блокируются*.

☑ Следует отметить, что подобный подход не вполне соответствует современной практике обработки ошибок, основанной на исключениях, а также другим, достаточно естественным, схемам работы с потоком, в которых конец потока распознается еще до того, как очередная операция чтения приведет к ошибке.

Для обнаружения ошибки программа должна явным образом проверять состояние потока, а для продолжения работы с потоком (если это возможно) — сбрасывать состояние ошибки.

Опишем средства, предусмотренные для проверки и изменения состояния потока; все эти средства реализованы в классе *ios* (точнее, в классе-шаблоне *basic_ios*) и поэтому доступны для всех рассматриваемых далее потоков.

ios <ios>

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

Логические функции, позволяющие проверить текущее состояние потока. Опишем для каждой функции, что означает ситуация, при которой эта функция возвращает значение *true*:

- *good*: в данный момент ошибок, связанных с потоком, не обнаружено;
- *bad*: при работе с потоком произошла *фатальная ошибка*, которую, скорее всего, не удастся исправить;
- *eof*: прочесть очередной элемент данных не удалось, так как обнаружен *конец потока*;
- *fail*: произошла какая-то ошибка; возможно, это фатальная ошибка (тогда значение *true* вернет и функция *bad*), возможно, это ошибка, связанная с концом потока (тогда значение *true* вернет и функция *eof*), возможно, это другая *ошибка ввода-вывода или преобразования данных* (в этой ситуации можно попытаться продолжить работу с потоком, сбросив состояние ошибки).

Важно подчеркнуть, что функция *eof* возвращает *true* после попытки прочесть очередной элемент за концом потока. Эту особенность следует учитывать, поскольку во многих языках программирования (например, в Паскале) функции, проверяющие конец потока (в частности, файла) ведут себя по-другому: возвращают значение *true*, как только прочитан последний элемент из потока, и поэтому механический перенос соответствующих конструкций в программу на C++ может привести к ее неверной работе (см. пример в конце данного пункта).

Следует также заметить, что фатальные ошибки при работе с потоками возникают достаточно редко; даже при попытке открыть несуществующий файл или создать файл с недопустимым именем функция *bad* соответствующего файлового потока возвращает *false* (хотя функция *fail*, естественно, возвращает *true*).

ios <ios>

```
void clear();
```

Сбрасывает состояние ошибки для данного потока (в результате функция *good* вернет значение *true*, а функции *eof*, *bad* и *fail* — значение *false*). Данную функцию следует использовать

в ситуации, когда можно рассчитывать на восстановление потока после обнаружения ошибки.

☑ На самом деле функция *clear* позволяет установить поток в любое состояние, указав комбинацию соответствующих *флагов состояния* в качестве своего параметра, однако эту возможность, как и другие возможности, связанные с использование флагов состояния, мы не будем описывать.

Для того чтобы еще более упростить проверку состояния потока, для него определены две операции.

*Операция !* («отрицания»), примененная к самому потоку, возвращает *true*, если поток находится в состоянии ошибки, и *false* в противном случае.

*Операция преобразования* (*void**) (то есть преобразования к нетипизированному указателю), примененная к потоку, возвращает нулевой указатель (то есть 0), если поток находится в состоянии ошибки, и ненулевое значение в противном случае. Для применения подобной операции достаточно указать имя потока в условии оператора *if* или *while*, например, *while (f)* … (где *f* — некоторый поток).

Приведем пример, иллюстрирующий применение описанных выше средств. Предположим, что поток *f* содержит 4 символа: «A», «B», «C», «D» и требуется сформировать строку, содержащую эти символы в том же порядке. Для чтения символов из потока будем использовать функцию *get*. В приводимых далее фрагментах программ предполагается, что поток *f* уже создан и доступен для чтения, а переменная *s* описана как *string*.

Вначале приведем ошибочный вариант решения, который, однако, представляется весьма «естественным» для программистов, знакомых с языком Паскаль или Visual Basic:

```
while (!f.eof())
    s += f.get();
```

В результате строка s получит значение "ABCD ", то есть будет содержать пять символов, последним из которых является *пробел*. Объясняется это тем, что после чтения последнего символа («D») из потока, поток еще не перейдет в состояние «обнаружен конец потока», поэтому при следующем вызове функция *eof* вернет *false*, и будет выполнена еще одна (лишняя) итерация цикла. На этой итерации будет предпринята попытка прочесть символ за концом потока. Разумеется, эта попытка закончится неудачей; в результате функция *get* вернет пробел, а поток *f* перейдет в состояние «обнаружен конец потока». При очередном вызове функции *eof* она вернет *true*, что обеспечит завершение цикла *while*.

Заметим, что аналогичный результат будет получен и при использовании следующих вариантов заголовка цикла: *while (!f.fail)* … или *while (f)* … (в последнем случае к потоку *f* будет неявно применена операция (*void**) преобразования к нетипизированному указателю).

Для правильного формирования строки можно выполнять «упреждающее» считывание символа из потока:

```
char c = f1.get();
while (f1)
{
    s += c;
    c = f1.get();
}
```

В этом варианте мы использовали наиболее краткое из возможных условий в цикле *while*. Приведенный вариант решения приводит к необходимости дублирования функции *get*, обеспечивающей считывание символа из файла, а также описания вспомогательной символьной переменной *c*.

От дублирования функции *get* можно избавиться, если использовать ее вариант с одним символьным параметром *get(c)*; в этом варианте прочитанный из потока символ записывается в параметр *c*, а функция возвращает сам поток (при попытке чтения за концом потока значение параметра *c* не изменяется):

```
char c;
while (f1.get(c))
    s += c;
```

Это, по-видимому, наиболее краткое решение, реализованное в рамках схемы «читать до первой ошибки». В этом решении нельзя избавиться от вспомогательной переменной *c*, которая, к тому же, является внешней по отношению к циклу *while*. Заметим также, что можно привести примеры задач, в которых не слишком удобно размещать операторы ввода данных в условии цикла.

В заключение приведем правильный вариант решения, реализованный в рамках схемы «читать до конца потока» (подобная схема является традиционной для многих других языков программирования, в том числе Паскаля, Visual Basic и языков платформы .NET). Для реализации этой схемы средствами стандартной библиотеки C++ приходится пользоваться функцией *peek*:

```
while (f1.peek() != -1)
    s += f1.get();
```

## *Перечисления, связанные с обработкой потоков*

В классе *ios* определено несколько перечислимых типов, связанных с обработкой потоков, а также константы, соответствующие элементам этих типов. Ниже приводится описание некоторых из перечислимых типов и связанных с ними констант.

### ios::openmode

Задает режим, в котором открывается и функционирует поток. Включает константы:

- *ios::app* — обеспечивает переход в конец потока перед каждой операцией записи;
- *ios::ate* — обеспечивает переход в конец потока при открытии потока;
- *ios::in* — включает режим чтения при открытии потока;
- *ios::out* — включает режим записи при открытии потока;
- *ios::trunc* — обеспечивает удаление прежнего содержимого потока при открытии потока;
- *ios::binary* — включает режим двоичного ввода-вывода при открытии потока (по умолчанию для потока устанавливается текстовый режим).

Константы, связанные с режимом потока, можно комбинировать с помощью операции | (побитовое ИЛИ). Например, режим двоичного ввода-вывода устанавливается комбинацией *ios::in* | *ios::out* | *ios::binary*. Как правило, режим потока указывается в качестве параметра его конструктора или другой функции, обеспечивающей открытие потока.

### ios::seekdir

Определяет позицию, от которой отсчитывается смещение потокового указателя при выполнении прямых переходов в потоке. Включает константы:

- *ios::beg* — смещение определяется относительно начала потока;
- *ios::cur* — смещение определяется относительно текущего положения указателя, связанного с потоком;
- *ios::end* — смещение определяется относительно конца потока.

Указанные константы используются в функциях *seekg* и *seekp*.

### ios::fmtflags

Устанавливает значения свойств форматирования при вводе и выводе данных. Флаги форматирования разделены на несколько групп: основание системы счисления, формат вывода вещественных чисел, выравнивание полей. Есть также набор независимых флагов. Часть флагов используется только при выводе данных, некоторые предназначены для настройки по-

токов ввода, остальные могут использоваться как при вводе, так и при выводе.

Основание системы счисления при вводе и выводе целочисленных значений задается константами:

- *ios::dec* — десятичная система счисления (используется по умолчанию);
- *ios::hex* — шестнадцатеричная система счисления;
- *ios::oct* — восьмеричная система счисления.

При установке значений флагов этой группы удобно использовать битовую маску *ios::basefield*.

При выводе вещественных чисел используются следующие флаги форматирования:

- *ios::fixed* — формат вывода с фиксированной точкой;
- *ios::scientific* — формат вывода в научной нотации (напр. 1.23e+26).

При установке значений флагов этой группы удобно использовать битовую маску *ios::floatfield*.

Выравнивание полей вывода определяется константами:

- *ios::internal* — поле вывода заполняется *символами-заполнителями* в указанной позиции;
- *ios::left* — поле вывода заполняется *символами-заполнителями* справа (то есть выводимое значение выравнивается по левому краю);
- *ios::right* — поле вывода заполняется *символами-заполнителями* слева (то есть выводимое значение выравнивается по правому краю).

При установке значений флагов этой группы удобно использовать битовую маску *ios::adjustfield*.

К независимым флагам форматирования относятся:

- *ios::boolalpha* — ввод и вывод логических значений в символьной форме *true* и *false* (по умолчанию используется значение 0 для *false* и 1 для *true*);
- *ios::showbase* — вывод целочисленных значений с указанием соответствующей системы счисления;
- *ios::showpoint* — вывод вещественных значений с обязательным указанием десятичной точки (по умолчанию, если вещественное число имеет нулевую дробную часть, она не выводится);
- *ios::showpos* — вывод неотрицательных значений с явным указанием знака «+»;
- *ios::skipws* — автоматический пропуск пробельных символов при чтении из потока ввода операцией >>;
- *ios::uppercase* — вывод всех символов в верхнем регистре;
- *ios::unitbuf* — автоматическое сбрасывание (*flush*) выходного буфера после каждой операции записи в поток.

Способы установки и снятия флагов форматирования описаны в разделе «Форматирование данных. Манипуляторы».

## *Базовый поток ввода: класс istream*

Базовым предком потоков ввода является класс *istream*, порожденный от класса *ios* и реализованный в заголовочном файле *istream*. Среди его потомков следует отметить классы *ifstream* (файловый поток ввода) и *istringstream* (строковый поток ввода).

Для ввода данных из потока *istream* (и всех его потомков) предусмотрена операция >> и набор функций, основные из которых описываются в данном пункте.

istream <istream>

```
операция >>
```

Данная операция обеспечивает форматированный ввод данных из потока ввода *f* в указанную переменную *v* и имеет вид *f >> v*. Она перегружена для всех базовых типов C++, в том числе типов *char, int, double*. Кроме того, для ввода строковых данных в качестве переменной *v* можно указывать символьный массив и переменную типа *string*.

Возвращаемым значением операции >> является поток, для которого она вызвана, что позволяет использовать для ввода нескольких данных выражения, содержащие несколько операций >>, например:

```
f >> v1 >> v2 >> v3;
```

В указанном операторе вначале выполняется операция *f >> v1*, считывающая элемент данных в переменную *v1*. Эта операция возвращает тот же поток *f*, который, таким образом, оказывается левым операндом следующей операции >>, вводящей данные в переменную *v2*. Эта операция, в свою очередь, возвращает поток *f*, который оказывается операндом последней операции >>. Данный процесс можно сделать более наглядным, если явным образом расставить скобки в приведенном выражении:

```
((f >> v1) >> v2) >> v3;
```

В качестве правого операнда операции >> можно также использовать *манипуляторы* (см. соответствующий раздел).

Целый ряд функций обеспечивает *неформатированный* ввод. Это означает, что во всех таких функциях предоставляется буфер для заполнения данными, считываемыми из потока. В качестве буфера могут использоваться символы (то есть, фактически, байты), символьные массивы и строки, а с учетом операций преобразования типа буфером может стать любой массив примитивных типов. Первая часть таких функций ориентирована на ввод символьных данных.

istream <istream>

```
int get();
istream& get(char& c);
istream& get(char* cstr, streamsize count);
istream& get(char* cstr, streamsize count, char
    delim);
istream& getline(char* cstr, streamsize count);
istream& getline(char* cstr, streamsize count,
    char delim);
```

Функция *get* возвращает код символа, прочитанного из потока ввода. При попытке чтения за концом потока поток приходит в ошибочное состояние. Вторая форма функции *get* принимает ссылку на символ *c* в качестве параметра и присваивает ему введенное значение. Третья и четвертая формы предназначены для чтения из потока ввода последовательности символов вплоть до обнаружения символа-разделителя, в качестве которого используется либо символ перевода строки (в третьей форме), либо символ *delim* (в четвертой форме). Сам символ-разделитель *остается в потоке* и в строку не попадает. В конец строки автоматически дописывается символ завершения строки '\0'. Функции *getline* действуют похожим образом, за тем исключением, что символ-разделитель *считывается из потока*, хотя в результирующую строку не попадает. Параметр *count* ограничивает максимальное количество вводимых символов. Программист должен обеспечить соответствующий размер символьного буфера с учетом добавляемого терминального символа '\0'. Отметим, что функции *getline* являются предпочтительным способом для чтения С-строк из потока, поскольку они ограничивают максимальное количество вводимых символов. При использовании операции >> таких ограничений нет, что может привести к переполнению буфера и неопределенным последствиям.

istream <istream>

```
istream& getline(istream& strm, string& str);
istream& getline(istream& strm, string& str, char
    delim);
```

Эти функции используются для чтения из потока строк типа *string*. Строка читается вплоть до разделителя (символ перевода строки или *delim*), сам разделитель считывается из потока, но в строку не попадает. Эти функции удобно использовать для построчного чтения потока:

```
string s;
while (getline(cin, s))
```

```
{
  // Обработка очередной строки s
}
```

istream <istream>

```
istream& read(char* cstr, streamsize count);
streamsize readsome(char* cstr, streamsize count);
```

Функции *read* и *readsome* предназначены для чтения символов из потока без учета разделителей. Параметр *count* ограничивает максимальное количество считываемых символов. При отсутствии в потоке требуемого количества символов функция *read* переводит его в ошибочное состояние. В противоположность этому функция *readsome* читает из потока столько символов, сколько возможно, и возвращает количество реально прочитанных символов. Вместо символьного буфера можно использовать любой другой буфер, преобразовав указатель на него к типу *char**, при этом важно не ошибиться в указании размера буфера:

```
double* p = new double[100];
is.read(static_cast<char*>(p), sizeof(double)*100);
```

Функции *read* и *readsome* считают символы перевода строки обычными символами, они также *не* добавляют к результирующему буферу терминальные символы.

istream <istream>

```
istream& ignore();
istream& ignore(streamsize count);
istream& ignore(streamsize count, int delim);
```

Функции *ignore* позволяют пропустить (извлечь из потока, но никуда не сохранить) символы, находящиеся в потоке. В первой форме пропускается один символ, во второй — *count* символов, а в третьей пропускаются все символы вплоть до символа-разделителя *delim*, включая его, причем пропускается не более чем *count* символов.

istream <istream>

```
int peek();
istream& unget();
istream& putback(char c);
```

Функция *peek* возвращает символ, который должен быть прочитан следующей функцией чтения потока. В частности, это позволяет распознать конец потока (см. раздел «Обработка ошибок ввода-вывода»). Функция *unget* помещает назад в поток последний прочитанный из него символ. Функция *putback* позволяет поместить в поток ввода произвольный символ *c*. Отметим, что не все потоки позволяют выполнять эти операции. Если поток их не поддерживает, то при попытке их вызова он переходит в состояние *фатальной* ошибки.

istream <istream>

```
istream& seekg(streampos pos);
istream& seekg(streamoff off, ios_base::seekdir dir);
streampos tellg();
```

Функции *seekg*(*pos*) и *seekp*(*pos*) перемещают файловый указатель на байт файла с номером *pos* (нумерация начинается от нуля), а функции *tellg* и *tellp* возвращают номер текущего байта (то есть байта, на котором расположен файловый указатель). В варианте с двумя параметрами первый параметр *off* задает относительное смещение от базовой позиции (и может быть как положительным, так и отрицательным), а второй параметр определяет базовую позицию и может принимать три значения: *ios*::*beg* (базовая позиция совпадает с началом файла), *ios*::*end* (базовая позиция совпадает с концом файла) и *ios*::*cur* (базовая позиция совпадает с текущей позицией файлового указателя).

☑ Функции с суффиксом *g* реализованы для потоков, открытых на чтение (**g**et), а функции с суффиксом *p* — для потоков, открытых на запись (**p**ut). Однако если файл открыт одновременно и на запись, и на чтение, то соответствующие потоки всегда синхронизированы, поэтому для управления файловым указателем можно использовать любую из этих функций.

Все функции, определенные в классе *istream*, надлежащим образом переопределяются в любом из его потомков, таким образом, их можно использовать также для файловых и строковых потоков ввода (см. далее).

### *Базовый поток вывода: класс ostream*

Базовым предком потоков вывода является класс *ostream*, порожденный от класса *ios* и реализованный в заголовочном файле *ostream*. Среди его потомков следует отметить классы *ofstream* (файловый поток ввода) и *ostringstream* (строковый поток ввода).

Для вывода данных в поток *ostream* (и все его потомки) предусмотрена операция << и набор функций, некоторые из которых описываются в данном пункте.

ostream <ostream>

```
операция <<
```

Операция << является основным механизмом форматированного вывода данных примитивных типов и типов, определенных пользователем. Ее использование аналогично использованию операции >> для потоков ввода:

```
int a = 7;
double b = 12.43;
string s = "Hello";
os << a << b << s;
```

Здесь в поток вывода *os* выводятся целое число, вещественное число и строка. Управление правилами вывода осуществляется с помощью функций форматирования и манипуляторов (см. раздел «Манипуляторы»).

ostream <ostream>

```
ostream& put(char c);
ostream& write(const char* str, streamsize count);
```

С помощью этих функций можно вывести в поток символ (функция *put*) или массив символов (функция *write*). С помощью преобразования типов функция *write* позволяет также записать в поток буфер с данными любых типов. Параметр *count* определяет размер выводимого буфера в байтах. Функция *write* обычно используется для работы с двоичными файлами.

ostream <ostream>

```
ostream& flush();
```

Данные, выводимые в поток несколькими операциями, обычно накапливаются в некотором буфере в памяти, и только после этого выводятся на диск или на терминал, то есть на то устройство, с которым связан поток. При необходимости сбросить все содержимое буфера следует вызвать функцию *flush*. Отметим, что при закрытии файловых потоков сбрасывание буфера на диск происходит автоматически.

ostream <ostream>

```
ostream& seekp(streampos pos);
ostream& seekp(streamoff off, ios_base::seekdir dir);
streampos tellp();
```

Эти функции описаны совместно с функциями seekg и tellg.

### *Форматирование данных. Манипуляторы*

Библиотека ввода/вывода языка C++ предоставляет две основные возможности для управления форматом вводимых и выводимых данных: флаги форматирования и манипуляторы. В качестве конкретных примеров правил форматирования можно привести регулировку количества знаков, выводимых после десятичной точки, или установку общей ширины поля вывода и его выравнивания.

#### Чтение и установка флагов форматирования

Флаги форматирования управляют правилами ввода и вывода значений разных типов, они объявлены в перечислении *ios::fmtflags* (см. выше его описание). Для установки и снятия этих флагов предусмотрен целый ряд функций.

Первая группа таких функций позволяет добавить и снять значения конкретных флагов.

ios_base <ios>

```
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);
```

Функция *setf* добавляет к набору флагов потока указанный флаг или их комбинацию:

```
cout.setf(ios::showpos | ios::uppercase);
```

Для установки флагов, принадлежащих одной группе (см. описание *ios::fmtflags*), следует пользоваться второй формой функции *setf* с указанием битовой маски. Так, в следующей строке устанавливается шестнадцатеричный формат для вводимых целых чисел:

```
cin.setf(ios::hex, ios::basefield);
```

Обе формы функции *setf* возвращают общий набор флагов, установленных для потока.

Функция *unsetf* снимает значение переданного ей флага:

```
cout.unsetf(ios::uppercase);
```

Второй способ задания флагов форматирования позволяет заменить весь набор новым значением.

ios_base <ios>

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

Функция *flags* возвращает весь набор флагов, установленных для потока. Эта же функция во второй форме позволяет установить полностью новый набор. Пара этих функций обычно используется для сохранения значения флагов (перед их временным изменением) и восстановления сохраненных значений:

```
ios::fmtflags oldFlags = cout.flags();
cout.setf(ios::showpos | ios::showbase |
    ios::uppercase);
cout.setf(ios::internal, ios::adjustfield);
cout << hex << x << endl;
cout.flags(oldFlags);
```

Эти же функции можно использовать для задания флагов форматирования, которые уже были установлены для другого потока. Однако ту же задачу можно решить проще с помощью функции *copyfmt*, специально предназначенной для копирования форматных настроек.

ios <ios>

```
ios& copyfmt(const ios& rhs);
```

Функция *copyfmt* позволяет скопировать значения всех флагов форматирования, установленных для другого потока, передаваемого ей в качестве параметра.

### Манипуляторы

Манипуляторы представляют собой функции специального типа, которые можно «выводить» в поток и «вводить» из него. Они позволяют выполнять основные операции форматирования проще, чем средствами флагов форматирования. Простейшие манипуляторы, не принимающие никаких параметров, объявляются вместе с классами потоков, с которыми они должны использоваться. Таким образом, никакие дополнительные заголовочные файлы не нужны. Все манипуляторы, принимающие параметры, объявлены в заголовочном файле *iomanip*.

Приведем примеры некоторых простейших манипуляторов.

<ostream>

```
endl
ends
flush
```

Манипулятор *endl* выводит в поток вывода символ перевода строки. Манипулятор *ends* позволяет вывести в поток терминальный символ '\0' (это имеет смысл при формировании символьных буферов средствами потоков). Манипулятор *flush* выполняет задачу, аналогичную функции *flush*, то есть принудительно сбрасывает на устройство, связанное с потоком, буфер памяти, в котором накапливалась выводимая информация.

С помощью манипуляторов можно также выполнять операции установки и снятия значений флагов форматирования.

<iomanip>

```
setiosflags(ios_base::fmtflags mask)
resetiosflags(ios_base::fmtflags mask)
```

Манипулятор *setiosflags* устанавливает значения переданных ему флагов для того потока, в который он выводится или из которого вводится, а манипулятор *resetiosflags* снимает их:

```
cout << resetiosflags(ios::adjustfield)
    << setiosflags(ios::left);
```

Для большинства задач форматирования существуют специализированные манипуляторы, пользоваться которыми гораздо удобнее.

### Форматирование полей вывода

Поле для вывода значения любого типа может быть охарактеризовано следующими достаточно общими параметрами: шириной (количеством символов, отводимых на вывод этого значения), символом-заполнителем (это символ, который должен выводиться в позиции, не заполненные самим значением) и выравниванием (размещением выводимого значения относительно границ поля вывода). Для управления этими параметрами могут использоваться как специальные функции-члены потоковых классов, так и манипуляторы.

ios_base <ios>

```
streamsize width() const;
streamsize width(streamsize wide);
```

Функция *width* позволяет узнать (в первой форме) и изменить (во второй форме) ширину поля вывода. Вторая форма возвращает значение ширины, которое было задано до вызова этой функции. Заметим, что функция *width* устанавливает *минимальный* размер поля вывода; если выводимое значение оказывается шире, то этот размер игнорируется. По умолчанию значение ширины поля вывода равно 0, что означает, что минимальный размер поля не ограничен. После осуществления операции вывода, перед которой было вызвано изменение ширины, значение по умолчанию восстанавливается. Таким образом, функция *width* влияет только на одно значение, выводимое после ее вызова.

ios <ios>

```
char fill() const;
char fill(char fillch);
```

Функция *fill* позволяет установить символ-заполнитель для полей вывода. В отличие от функции *width*, ее действие продолжается вплоть до установки другого символа-заполнителя.

<iomanip>

```
setw (int n)
setfill (char c)
```

Манипуляторы *setw* и *setfill* можно использовать в качестве полного аналога соответствующих функций:

```
cout << setw(5) << 12 << 6 << setw(10) << 6.7 << endl;
```

Здесь на вывод числа 12 отводится 5 позиций, а на вывод числа 6.7 — 10 позиций. Заметим, что число 6 выводится в поле, ширина которого не ограничивается (то есть для него действует значение по умолчанию).

Выравнивание поля вывода может устанавливаться с помощью флагов форматирования из группы *ios::adjustfield*, а также посредством манипуляторов.

<ios>

```
left
right
```

Эти манипуляторы предназначены для установки выравнивания поля вывода по левому краю или по правому краю.

### Форматирование целых и вещественных чисел

Среди средств форматирования числовых данных можно отметить ввод и вывод целых чисел в разных системах счисления или регулировку количества символов, выводимых в вещественном числе после десятичной точки. Управление этими средствами может осуществляться флагами форматирования из групп *ios::basefield* и *ios::floatfield*, а также специальными функциями и манипуляторами.

<ios>

```
dec
oct
hex
showbase
noshowbase
```

Манипуляторы *dec*, *oct* и *hex* переводят поток в режим ввода или вывода целых чисел, записанных в десятичной, восьмеричной и шестнадцатеричной системах счисления. По умолчанию все целые числа выводятся в десятичной системе счисления. Манипуляторы *showbase* и *noshowbase* позволяют соответственно выводить и не выводить признак системы счисления ('0x' для шестнадцатеричной системы счисления и '0' для восьмеричной).

ios_base <ios>

```
streamsize precision() const;
streamsize precision(streamsize prec);
```

Функция *precision* устанавливает количество символов, выводимых после десятичной точки, возвращая значение, установленное ранее.

<ios>

```
showpoint
noshowpoint
fixed
scientific
```

Манипуляторы *showpoint* и *noshowpoint* управляют отображением десятичной точки в вещественных числах с нулевой дробной частью. Манипулятор *fixed* устанавливает формат вывода вещественных чисел с фиксированной точкой, а манипулятор *scientific* позволяет выводить числа в научной нотации (например, 1.87e+21).

<iomanip>

```
setprecision(int n)
```

Этот манипулятор аналогичен действию функции *precision*.

<ios>

```
showpos
noshowpos
```

С помощью этих манипуляторов можно включать (*showpos*) и выключать (*noshowpos*) вывод знака '+' для неотрицательных чисел. По умолчанию знак '+' не выводится.

### *Файловые потоки ввода-вывода*

Файловые потоки являются производными от общих потоковых классов *istream*, *ostream* и *iostream*, поэтому они наследуют все их возможности. Укажем здесь только то, что их отличает. Здесь приводится описание нескольких функций класса *fstream*, функции классов *ifstream* и *ofstream* почти полностью аналогичны.

fstream <fstream>

```
explicit fstream(const char* filename,
    ios_base::openmode mode = ios_base::in |
    ios_base::out);
void open(const char* filename, ios_base::openmode mode
    = ios_base::in | ios_base::out);
void close();
bool is_open();
```

Конструктор класса *fstream* позволяет указать имя файла, с которым связывается поток. Если имя файла содержится в переменной типа *string*, то необходимо преобразовать его в форму *char** (в стандарте C++11 это требование отменено):

```
string filename = "file.dat";
fstream f(filename.c_str());
```

Функция *open* с теми же параметрами позволяет открыть файл после создания объекта потока. Файловый поток должен закрываться функцией *close*. Проверить, открыт ли файл в настоящее время, можно с помощью функции *is_open*.

После открытия файловый поток *fstream* может выполнять как операции чтения, так и операции записи. Файловый поток ввода *ifstream* открывается в режиме чтения, а поток вывода *ofstream* в режиме записи.

### *Строковые потоки ввода-вывода*

Строковые потоки предназначены для чтения и записи данных в строки. Фактически строки играют здесь роль буфера для хранения данных. Обычно строковые потоки используют для предварительной подготовки данных перед их выводом или для реализации отложенного ввода. Еще один вариант применения — преобразование значений разных типов в строковое представление. Так как строковые потоки являются наследниками общих потоковых классов, для работы с ними можно использовать все рассмотренные ранее приемы, в том числе операции >> и <<, флаги форматирования и манипуляторы. Специальные функции строковых потоков позволяют указать строку, используемую в качестве буфера данных, а также прочесть эту строку с накопленными в ней данными.

istringstream <sstream>

```
explicit istringstream(openmode which = ios_base::in);
explicit istringstream(const string & str,
    openmode which = ios_base::in);
```

Эти конструкторы создают объект класса *istringstream*. Вторая форма конструктора позволяет указать строку, используемую в качестве источника ввода.

Поток *istringstream* может использоваться для преобразования строкового представления любого примитивного типа в соответствующее значение:

```
float f;
string s = "3.7";
istringstream is(s);
is >> f;
```

istringstream <sstream>

```
string str() const;
void str(const string & s);
```

Функция *str* предназначена для чтения строки, используемой потоком в качестве буфера, а также для ее установки.

ostringstream <sstream>

```
explicit ostringstream(openmode which = ios_base::out);
explicit ostringstream(const string & str,
    openmode which = ios_base::out);
```

Эти конструкторы создают объект класса *ostringstream*. Вторая форма конструктора позволяет указать строку, используемую в качестве приемника выводимых данных.

ostringstream <sstream>

```
string str() const;
void str(const string & s);
```

Функция *str* предназначена для чтения строки, используемой потоком в качестве буфера, а также для ее установки.

Поток *ostringstream* может использоваться для преобразования значения примитивного типа в строковое представление:

```
float f = 3.7;
ostringstream os;
os << f;
string s = os.str();
```

Отметим, что перед выводом числа в строковый поток можно настроить любые свойства форматирования.