1. Введение в компоненты стандартной библиотеки шаблонов STL

Стандартная библиотека шаблонов (standard template library – STL) является ядром стандартной библиотеки языка C++, повлиявшим на всю ее архитектуру. STL – это библиотека универсальных компонентов для управления коллекциями (наборами) данных с помощью современных и эффективных алгоритмов. С точки зрения программиста библиотека STL содержит совокупность классов коллекций для различных целей и набор алгоритмов для работы с ними.

Как следует из названия «стандартная библиотека шаблонов», все компоненты являются шаблонами, допускающими использование любого типа, при условии, что этот тип способен выполнять требуемые операции. Таким образом, библиотека STL – яркий пример концепции обобщенного программирования. Контейнеры и алгоритмы являются обобщенными по отношению к произвольным типам и классам соответственно.

В основе библиотеки STL лежит взаимодействие разных хорошо структурированных компонентов, главными среди которых являются контейнеры, итераторы и алгоритмы.

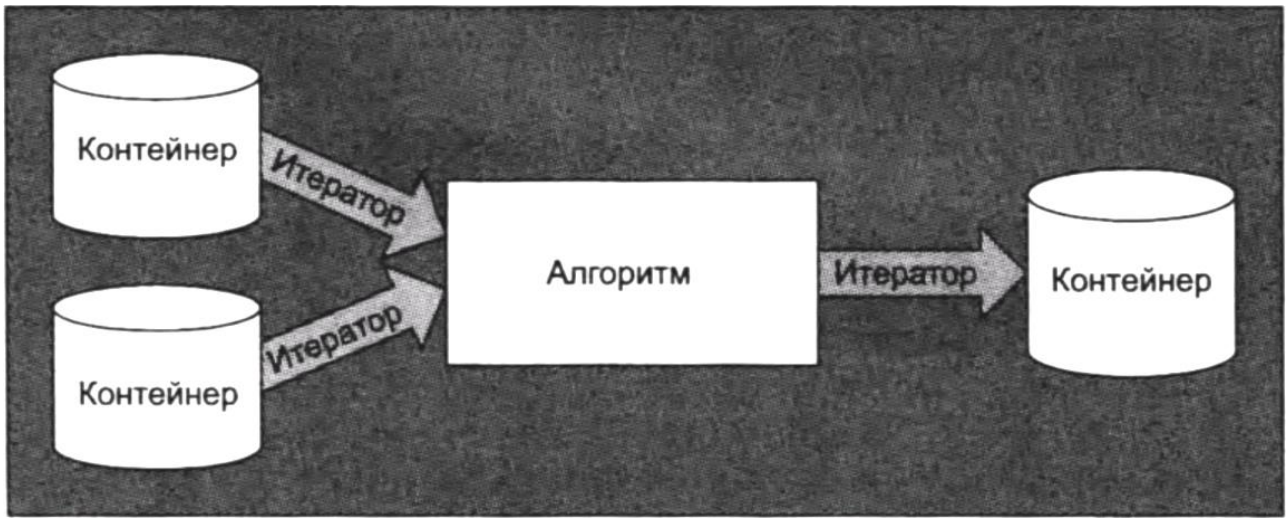
– **Контейнеры** используются для управления коллекциями объектов определенного типа. Каждый вид контейнеров имеет свои достоинства и недостатки, поэтому наличие разных контейнеров отражает разные требования к коллекциям, существующие в программах. Контейнеры могут быть реализованы как массивы или связанные списки или могут иметь специальный ключ для каждого элемента.

– **Итераторы** используются для обхода элементов в коллекциях объектов. Этими коллекциями могут быть контейнеры или подмножества контейнеров. Основное преимущество итераторов заключается в том, что они предоставляют небольшой и в то же время универсальный интерфейс для произвольного типа контейнера. Например, одна из операций этого интерфейса перемещает итератор на следующий элемент в коллекции. Выполнение этой операции не зависит от внутренней структуры коллекции. Чем бы ни была коллекция – массивом, деревом или хеш-таблицей, – эта операция работает одинаково, поскольку каждый контейнер определяет свой собственный тип итератора, который просто «правильно работает», поскольку знает внутреннюю структуру своего контейнера.

– **Алгоритмы** предназначены для обработки элементов коллекций. Например, алгоритмы могут искать, сортировать, модифицировать и просто использовать элементы для разных целей. Алгоритмы используют итераторы. Таким образом, поскольку интерфейс итераторов является общим для всех типов контейнеров, алгоритм достаточно написать один раз, и он будет работать с любым контейнером. Для того чтобы повысить гибкость алгоритмов, их можно использовать в сочетании со вспомогательными функциями, которые вызываются алгоритмами. Таким образом, универсальный алгоритм можно использо-

вать для решения своей задачи, даже если эта задача является очень специфичной или сложной. Например, программист может задать особый критерий поиска или особую операцию для объединения элементов.

Концепция библиотеки STL основана на разделении данных и операций. Данные управляются контейнерными классами, а операции определяются настраиваемыми алгоритмами. Связующим звеном между этими двумя компонентами являются итераторы. Они позволяют алгоритму взаимодействовать с любым контейнером (см. рисунок).



Отметим, что концепция библиотеки STL относится не к объектно-ориентированной, а к парадигме функционального программирования. Вместо объединения данных и операций, как это принято в объектно-ориентированном программировании, они разделяются на отдельные части, взаимодействующие с помощью определенного интерфейса. В принципе, программист может объединять любой контейнер с любым алгоритмом, так что результат будет очень гибким и в то же время компактным.

Впрочем, эта концепция имеет и недостатки: во-первых, ее использование не является интуитивным, во-вторых, некоторые сочетания структур данных и алгоритмов могут оказаться неработоспособными. Возможна еще более плохая ситуация, когда некое сочетание контейнерного типа и алгоритма может оказаться возможным, но вредным (например, снижать быстродействие программы). Таким образом, для того чтобы извлечь максимальную пользу, необходимо хорошо знать библиотеку STL со всеми ее достоинствами и недостатками.

Практически все идентификаторы стандартной библиотеки определяются в **пространстве имен std**. В соответствии с концепцией пространств имен существуют три варианта использования идентификатора из стандартной библиотеки C++.

– **Явная квалификация идентификатора.** Например, можно написать
std::cout << 3.4 << std::endl;

– **Объявление using.** Например, следующий фрагмент программы предо-

ставляет локальную возможность пропустить префикс std:: для объекта cout и модификатора формата endl:

```
using std::cout; using std::endl;
. . . . . .
cout << 3.4  << end;
```

– **Директива using.** Это простейший вариант. После выполнения директивы using для пространства имен std все идентификаторы этого пространства доступны так, будто они были объявлены глобально. Например, оператор

```
using namespace std;
```

позволяет написать

```
cout << 3.4 << endl;
```

Отметим, что в сложных программах это может привести к случайным конфликтам имен или, что еще хуже, к непредсказуемым последствиям из-за запутанных правил перегрузки. Никогда не используйте директиву using, если контекст неясен (например, в заголовочных файлах).

Наши примеры будут довольно невелики, поэтому всегда, когда это не приведет к недоразумениям, будет предполагаться использование директивы using и префикс std:: использоваться не будет.

2. Введение в строковые типы

В языке Си для представления строк используются массивы символов типа char* (С-строки). С-строки позволяют достичь высокой эффективности, но весьма неудобны и небезопасны в использовании, поскольку выход за границы строки не проверяется. Тип данных string стандартной библиотеки C++ лишен этих недостатков, но может проигрывать массивам символов в эффективности. Основные действия со строками выполняются в нем с помощью операций и методов, а длина строки изменяется динамически в соответствии с потребностями.

В заголовочном файле <string> определен класс basic_string<> как базовый класс для всех строковых типов.

```
namespace std {
template <typename charT,
typename traits = char_traits<charT>,
typename Allocator = allocator<charT>>
class basic_string;
}
```

Этот класс параметризован символьным типом, свойствами символьного типа и моделью памяти.

• Первый параметр – это тип данных отдельного символа.

• Необязательный второй параметр – это класс свойств, содержащий все операции над символами строкового класса. Если этот класс не определен, используется класс свойств, установленный по умолчанию для текущего символьного типа.

• Третий необязательный параметр определяет модель памяти, используемую строковым классом. По умолчанию используется модель памяти allocator.

В стандартной библиотеке C++ содержится несколько специализаций базового класса basic_string<>.

• Класс string является предопределенной специализацией шаблона для символов типа char:

namespace std {
typedef basic_string<char> **string**; }

• Три остальных типа определены для строк, использующих более широкий набор символов, такой как Unicode или некоторые азиатские алфавиты:

namespace std {
typedef basic_string<wchar_t> **wstring**;
typedef basic_string<char16_t> **u16string**;
typedef basic_string<char32_t> **u32string**;
}

Мы в основном будем рассматривать строки класса string.

В классе string определено несколько конструкторов. Ниже в упрощенном виде приведены заголовки наиболее употребительных:

```
string();
string(const char *);
string(const char *, int n);
string(const string &);
```

Первый конструктор создает пустой объект типа string. Второй создает объект типа string на основе C-строки, третий создает объект типа string и записывает туда n символов из C-строки, указанной первым параметром. Последний конструктор является конструктором копирования, который создает новый объект как копию объекта, переданного ему в качестве параметра.

В классе string определены три операции присваивания:

```
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
```

Как видно из заголовков, строке можно присваивать другую строку типа string, C-строку или отдельный символ, например:

```
string s1;
string s2("Иванов");
string s3(s2);
s1 = 'X';
s1 = "Иванов";
s2 = s3;
```

Для работы со строками в библиотеке предусмотрен широкий набор операций, однако, в лекциях они рассматриваться не будут.

3. Некоторые новые языковые средства C++

Ключевое слово nullptr и тип std::nullptr_t.

Стандарт C++11 позволяет использовать ключевое слово nullptr вместо 0 или NULL, чтобы отметить тот факт, что указатель не ссылается ни на один объект (в отличие от ситуации, когда указатель не имеет определенного значения). Это новое средство языка позволяет избежать ошибок, возникающих, когда нулевой указатель интерпретируется как целочисленное значение. Например:

```
void f(int);
void f(void*);
f(0); // вызов f(int)
f(NULL); // зависит от определения NULL;
                                        // если NULL = = 0, то вызывается f(int)
f(nullptr); // вызов f(void*)
```

Слово nullptr – это новое ключевое слово. Константа, заданная с помощью ключевого слова nullptr, автоматически конвертируется в любой тип указателя, но не в целочисленный тип. Она имеет тип std::nullptr_t, определенный в заголовке <cstddef>.

Автоматическое выведение типа с помощью ключевого слова auto.

В языке C++11 можно объявить переменную или объект без указания их конкретного типа, используя ключевое слово auto. Рассмотрим пример:

```
auto i = 42;  // Переменная i имеет тип int
double f();
auto d = f(); // Переменная d имеет тип double
```

Тип переменной, объявленной с помощью ключевого слова auto, выводится из ее инициализатора. Например, в следующем коде требуется инициализатор:

```
auto i;   // ОШИБКА:  невозможно вывести тип переменной i
```

Разрешается использование дополнительных квалификаторов. Например:

```
static auto vat = 0.19;
```

Использование ключевого слова auto особенно полезно в ситуациях, когда тип является очень длинным, а выражение сложным. Например:

```
vector<string> v;
. . . . .
auto pos = v.begin();
```

Здесь определен контейнер v типа vector – библиотечный аналог динамического массива, он имеет своими элементами строки типа string, v.begin() – итератор на первый элемент контейнера, его тип – vector<string>::iterator, переменная pos – это итератор того же типа.

Универсальная инициализация и списки инициализации.

До появления стандарта C++11 программисты, особенно начинающие, легко могли запутаться в вопросах инициализации переменной или объекта. Инициализация могла осуществляться с помощью круглых или фигурных скобок, а также операторов присваивания.

По этой причине в стандарт C++11 включена концепция универсальной инициализации, означающая, что любая инициализация осуществляется с помощью единообразного синтаксиса. Эта конструкция использует фигурные скобки. Например:

```
int values[ ] {1, 2, 3};
vector<int> v {2, 3, 5, 1, 11, 13, 17}; //
vector<string> cities {"Berlin", "New York", "London", "Braunschweig", "Cairo", "Cologne"};
```

Список инициализации осуществляет так называемую инициализацию значениями, подразумевающую, что каждая, даже локальная переменная элементарного типа, которая обычно имеет неопределенное начальное значение, инициализируется нулем (или константой nullptr, если переменная является указателем):

```
int i;       //  Переменная i имеет неопределенное значение
int j{};   //  Переменная j  инициализируется нулем
int* p;    //  Переменная p имеет неопределенное значение
int* q{}; //  Переменная q инициализируется константой nullptr
```

Однако сужающие инициализации, т.е. уменьшающие точность или модифицирующие передаваемое значение, в фигурных скобках запрещены. Например:

```
int x1(5.3);        // OK, но x1  становится равным 5
int x2 = 5.3;     // OK, но x2 становится равным 5
int x3{5.3};     // ОШИБКА: сужение
int x4 = {5.0}; // ОШИБКА: сужение
char c1{7};      // OK: несмотря на то что 7 - целое значение, это не сужение
char c2{99999};    // ОШИБКА: сужение (если 99999 выходит
                        // за диапазон char)
vector<int> v1 {1, 2, 4, 5}; // OK
vector<int> v2   {1, 2.3, 4, 5.6}; // ОШИБКА:   сужение double в int
```

Легко видеть, что для проверки сужения могут рассматриваться даже текущие значения, доступные на этапе компиляции.

Для поддержки концепции списков инициализации для пользовательских типов стандарт C++11 предусматривает класс std::initializer_list<>. Его можно использовать для инициализации с помощью списка значений или в любом другом месте, где требуется список значений. Например:

```
void print (initializer_list<int> vals)
{
        for (auto p=vals.begin(); p!=vals.end(); ++p)
                cout << *p << "\n";  //обработка списка значений
```

```
}
. . . . . .
print ({12, 3, 5, 7, 11, 13, 17});   // передача списка значений функции print()
```

Здесь тип итератора initializer_list<int>::iterator заменен на auto, vals.begin() – итератор на начало списка инициализации, vals.end() – итератор на фиктивную позицию за его концом.

При наличии конструкторов как с конкретным количеством аргументов, так и со списком инициализации предпочтение отдается версии со списком инициализации.

```
class P
{
public:
        P(int, int);
        P(initializer_list<int>);
};
P p(77,5);         // вызов P::P(int,int)
P q{77,5};         // вызов P::P(initializer_list)
P r{77,5,42};   // вызов P::P(initializer_list)
P s = {77,5};   // вызов P::P(initializer_list)
```

Если бы конструктора со списком инициализации не было, то для инициализации объектов q и s был бы вызван конструктор, получающий два аргумента типа int, а инициализация объекта r была бы некорректной.

Благодаря спискам инициализации теперь ключевое слово explicit относится и к конструкторам, принимающим больше одного аргумента. Следовательно, теперь можно запретить автоматические преобразования типов нескольких значений, которые использовались также при инициализации с помощью синтаксиса присваивания.

```
class P
{
public:
        P(int a, int b) {
                . . . . .
        }
        explicit P(int a, int b, int c) {
                . . . . .
        }
};
P x(77, 5);                 // OK
P y{77, 5};                 // OK
P z{77, 5, 42};         // OK
P v = {77, 5};          // OK   (неявное преобразование типа допускается)
P w = {77, 5, 42}; // ОШИБКА из-за ключевого слова explicit
                        //   (неявное преобразование типа не допускается)
```

```
void fp(const P&);
fp({47,11});            // OK,  неявное преобразование {47,11} в P
fp({47,11,3});          // ОШИБКА из-за ключевого слова explicit
fp(P{47,11});           // OK,  явное преобразование {47,11} в P
fp(P{47,11,3});         // OK,  явное преобразование {47,11,3} в P
```

Аналогично в конструкторах с ключевым словом explicit, получающих список инициализации, запрещены неявные преобразования списков инициализации – как пустых, так и содержащих одно или несколько значений.

В C++17 при автоматическом выведении типа прямое использование списка инициализации (в отличие от его копирования, т.е. использования знака =) разрешено только с одиночными элементами, причем тип переменной будет определяться инициализирующим значением без использования initializer_list<>.

```
auto x{1}; // x имеет тип int
auto y{1, 2}; // ОШИБКА: некорректная конструкция
auto x = {1}; // x имеет тип std::initializer_list<int>
auto y = {1, 2}; // y имеет тип std::initializer_list<int>
```

Семантика перемещения.

Различие между конструированием с помощью *копирования* и присваиванием с помощью *копирования* с одной стороны и конструированием посредством *перемещения* и присваиванием посредством *перемещения* с другой стороны состоит в том, что операция копирования оставляет исходное значение неизменным, а операция перемещения может изменять исходное значение, возможно, перенося права владения без выполнения какого-либо копирования. Когда исходный объект является *временным*, операции перемещения могут предоставить более эффективный код, чем обычное копирование. Это особенно полезно, если копирование объекта связано с большими затратами ресурсов – например, если он представляет собой крупную коллекцию строк. В этом случае быстродействие программы значительно повышается. Перемещение возможно (но необязательно) для введенных в C++11 rvalue-ссылок, объявляемых как type &&. Далее в лекциях для rvalue-ссылок часто используется имя rv.

Например, перемещающий конструктор для строк обычно просто присваивает существующий внутренний массив символов новому объекту, а не создает новый массив и не копирует все элементы. То же самое относится и ко всем классам коллекций: вместо создания копий всех элементов мы просто присваиваем внутреннюю память новому объекту.

Кроме того, необходимо, чтобы любая модификация – особенно уничтожение – объекта, содержимое которого было перемещено, не оказывало влияния на состояние нового объекта, который стал владельцем перемещенного значения.

Все классы стандартной библиотеки C++ гарантируют, что после перемещения объект пребывает в *корректном, но неопределенном состоянии.* Иначе

говоря, вы можете впоследствии присваивать ему новые значения, но его текущее состояние не определено. Контейнеры из библиотеки STL гарантируют, что после перемещения значений они остаются пустыми.

Пример. Шаблонный класс UPtr реализует указатель настраиваемого типа, не допускающий дублирования (у объекта всегда один указатель класса UPtr):

```
template<class T> class UPtr {
        T* p;
    public:
        UPtr(T* pp): p{pp} {} // конструктор
        ~UPtr() {delete p;} // деструктор
    // передача владения (конструктор перемещения):
        UPtr(UPtr&& h): p{h.p} {h.p = nullptr;};
    // передача владения (операция перемещения):
        UPtr& operator= (UPtr&& h){
                if (&h ! = this) {
                        delete p;
                        p = h.p;
                        h.p = nullptr;
                }
                return *this;
        }
    // копирование запрещено:
        UPtr(const UPtr& ) = delete;
        UPtr& operator=(const UPtr&) = delete;
      // . . . . .
};
```

4. Библиотека потокового ввода-вывода

4.1. Потоковые классы

Классы для ввода-вывода являются важной частью стандартной библиотеки C++; программы без ввода-вывода не имеют широкого применения. Классы ввода-вывода из стандартной библиотеки C++ не ограничены файлами или экранами и клавиатурой и образуют расширяемую платформу для форматирования произвольных данных и доступа к произвольным внешним представлениям.

Библиотека IOStream, названная так по имени ввода-вывода, – единственная часть стандартной библиотеки C++, которая широко использовалась до стандартизации C++98. Ранние дистрибутивы систем C++ поставлялись вместе с рядом классов, разработанных в компании AT&T, которая установила фактический стандарт ввода-вывода. Несмотря на то, что эти классы претерпели не-

сколько изменений, чтобы соответствовать стандартной библиотеке C++ и удовлетворять новым потребностям, основные принципы библиотеки IOStream остаются неизменными.

Поток – это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки C++, в отличие от функций ввода/вывода в стиле Си, обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис.

Чтение (ввод) данных из потока иногда называется извлечением, вывод в поток – помещением или включением. Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специальную область оперативной памяти – буфер. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе – если буфер исчерпан.

По направлению обмена потоки можно разделить на входные (данные вводятся в память), выходные (данные выводятся из памяти) и двунаправленные (допускающие как извлечение, так и включение).

По виду устройств, с которыми работает поток, можно разделить потоки на стандартные, файловые и строковые.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, файловые потоки – для обмена информацией с файлами на внешних носителях данных (например, на магнитном диске), а строковые потоки – для работы с массивами символов в оперативной памяти.

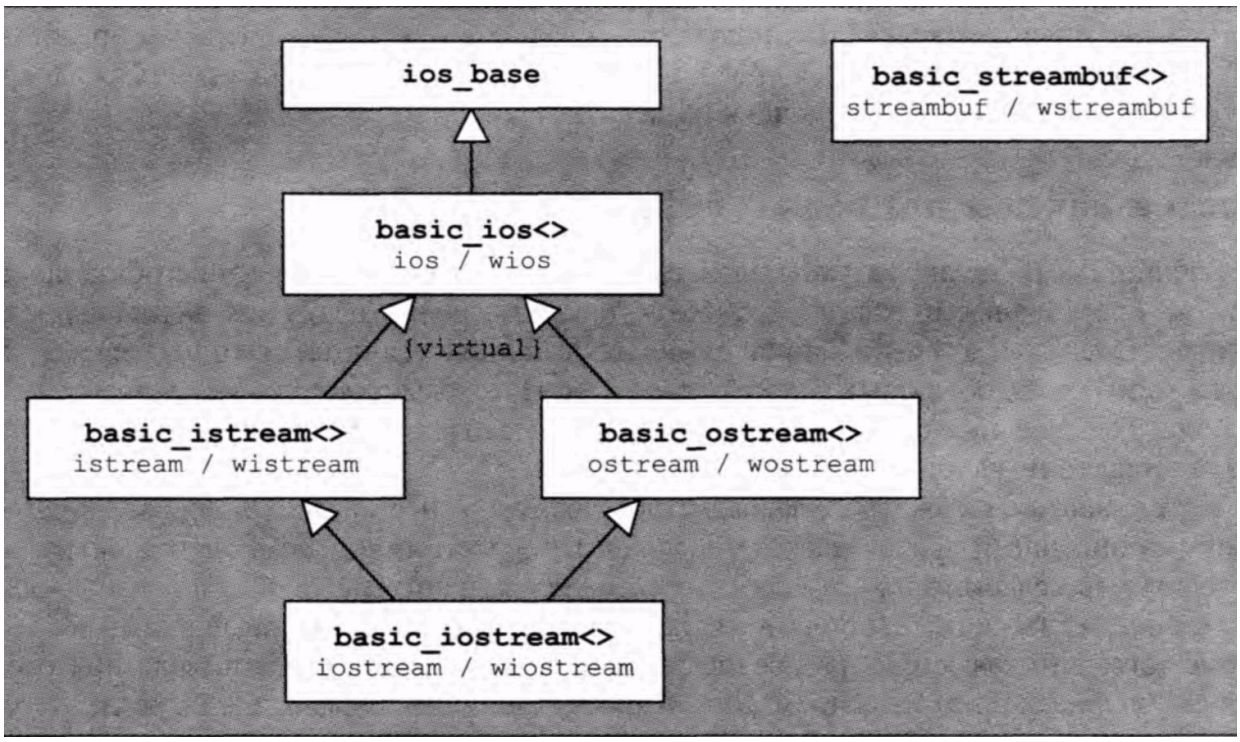
Иерархия классов.

Потоковые классы образуют иерархию, показанную на рисунке. Для шаблонных классов в верхней строке показано его имя, а в нижней строке – имена специализации для символьных типов char и wchar_t.

Классы в этой иерархии выполняют следующие задачи.

• Базовый класс ios_base определяет свойства всех потоковых классов, не зависящие от типа и соответствующих свойств символов. Этот класс в основном состоит из компонентов и функций, предназначенных для управления состоянием и флагами формата.

• Шаблонный класс basic_ios<>, производный от класса ios_base, определяет общие свойства всех потоковых классов, зависящие от типа и соответствующих свойств символов. К этим свойствам относится также определение буфера, используемого потоком данных. Буфер – это объект класса, производного от базового класса basic_streambuf<>, с соответствующей специализацией. Именно он выполняет операции чтения и/или записи.



• Шаблонные классы basic_istream<> и basic_ostream<>, виртуально наследующие от класса basic_ios<>, определяют объекты, которые могут использоваться для чтения и записи соответственно. Эти классы, как и классы basic_ios<>, представляют собой шаблоны, параметризованные символьным типом и его свойствами. Если проблемы интернационализации не имеют значения, используются специализации этих классов для символьного типа char – классы istream и ostream.

• Шаблонный класс basic_iostream<> является производным от двух шаблонных классов – basic_istream<> и basic_ostream<>. Он определяет объекты, которые могут использоваться как для чтения, так и для записи.

• Шаблонный класс basic_streambuf<> является ядром библиотеки IOStream. Он определяет интерфейс всех представлений, которые могут быть записаны в потоки или считаны из потоков, и используется другими потоковыми классами для чтения или записи символов. Для получения доступа к некоторым внешним представлениям классы объявляются производными от класса basic_streambuf<>.

Библиотека IOStream разрабатывалась на основе строгого разделения обязанностей. Классы, производные от класса basic_ios, предназначены только для форматирования данных. Операции чтения и записи символов выполняются потоковыми буферами, поддерживаемыми подобъектами класса basic_ios. Потоковые буферы обеспечивают чтение и запись в символьных буферах. Кроме того, они позволяют абстрагироваться от внешнего представления, например, файлов или строк.

Подробные определения классов.

Как и все шаблонные классы в библиотеке IOStream, шаблонный класс

basic_ios<> параметризован двумя аргументами и определен следующим образом:

```
namespace std  {
template <typename charT,
typename traits = char_traits<charT>> class basic_ios; }
```

Шаблонными аргументами являются символьный тип, используемый потоковыми классами, и класс, описывающий свойства этого символьного типа.

Примерами свойств, определенными в классе свойств, являются значения, используемые как признак файла, и инструкции о том, как копировать или перемещать последовательности символов. Как правило, свойства символьного типа ассоциируются с определенным символьным типом, поэтому целесообразно определить шаблонный класс, специализируемый для конкретных символьных типов. Следовательно, класс свойств по умолчанию является классом char_traits<charT>, если аргумент задает символьный тип charT. Стандартная библиотека C++ содержит специализации класса char_traits для символьных типов char, char16_t, char32_t и wchar_t (типы char16_t и char32_t введены стандартном C++11).

Существуют две специализации класса basic_ios<> для двух наиболее распространенных символьных типов:

```
namespace std {
typedef basic_ios<char> ios;
typedef basic_ios<wchar_t> wios;
}
```

Тип ios соответствует базовому классу старой библиотеки IOStream, разработанной компанией AT&T, и может использоваться для обеспечения совместимости со старыми программами на языке C++.

Класс потокового буфера, используемый классом basic_ios, определяется аналогично.

```
namespace std {
template <typename charT,
typename traits = char_traits<charT>> class basic_streambuf;
typedef basic_streambuf<char> streambuf;
typedef basic_streambuf<wchar_t> wstreambuf;
}
```

Разумеется, шаблонные классы basic_istream<>, basic_ostream<> и basic_iostream<> также параметризуются символьным типом и классом свойств.

```
namespace std {
template <typename charT,
typename traits = char_traits<charT>> class basic_istream;
template <typename charT,
typename traits = char_traits<charT>> class basic_ostream;
template <typename charT,
typename traits = char_traits<charT>> class basic_iostream;
}
```

Аналогично другим классам, существуют специализации двух наиболее важных символьных типов:

```
namespace std {
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
}
```

Типы istream и ostream обычно используются в Западном полушарии, где достаточно использовать наборы восьмибитовых символов. Тип wchar_t позволяет использовать наборы символов, состоящих из более чем восьми битов. Для типов char16_t и char32_t в стандартной библиотеке C++ нет соответствующих специализаций.

Глобальные потоковые объекты (стандартные потоки).

В потоковых классах определено несколько глобальных потоковых объектов, предназначенных для обеспечения доступа к стандартным каналам ввода-вывода с символьными типами char и wchar_t.

Тип	Имя	Назначение
istream	cin	Считывает данные из стандартного канала ввода
ostream	cout	Записывает обычные данные в стандартный канал вывода
ostream	cerr	Записывает сообщения об ошибках в стандартный канал ошибок
ostream	clog	Записывает регистрационные сообщения в стандартный канал регистрации
wistream	wcin	Считывает символы в расширенной кодировке из стандартного канала ввода
wostream	wcout	Записывает обычные символы в расширенной кодировке в стандартный канал вывода
wostream	wcerr	Записывает сообщение об ошибках, состоящее из символов

		в расширенной кодировке, в стандартный канал ошибок
wostream	wclog	Записывает сообщение регистрации, состоящее из символов в расширенной кодировке, в стандартный канал регистрации

Стандартным каналом ввода по умолчанию является клавиатура, остальными – экран. Имена объектов можно переназначить на файлы или символьные буферы. Глобальные потоковые объекты создаются при включении в программу заголовочного файла <iostream>, при этом становятся доступными связанные с ними средства ввода/вывода.

По умолчанию эти стандартные потоки синхронизированы со стандартными потоками в языке Си. Иначе говоря, стандартная библиотека C++ гарантирует сохранение порядка смешанного вывода в потоки C++ и Си. Прежде чем записать в них данные, каждый буфер стандартных потоков C++ очищает соответствующий буфер Си, и наоборот. Разумеется, эта синхронизация требует определенных затрат времени. Если она не нужна, то ее можно отключить с помощью вызова sync_with_stdio (false) перед вводом или выводом.

Заголовочные файлы.

Определения потоковых классов разбросаны по нескольким заголовочным файлам.

• Файл <iosfwd> содержит опережающие объявления потоковых классов.

• Файл <streambuf> содержит определения базового класса потокового буфера (basic_streambuf<>).

• Файл <istream> содержит определения классов, поддерживающих только ввод (basic_istream<>) или и ввод, и вывод (basic_iostream<>).

• Файл <ostream> содержит определения для потокового класса вывода (basic_ostream<>).

• Файл <iostream> содержит объявления глобальных потоковых объектов, таких как cin и cout.

Большинство заголовочных файлов предназначено для внутренней организации стандартной библиотеки C++. Прикладному программисту достаточно включить файл <iosfwd> в объявление потоковых классов и файл <istream> или <ostream> при непосредственном использовании функций ввода или вывода соответственно. Заголовочный файл <iostream> следует включать только при использовании стандартных потоковых объектов. В некоторых реализациях при выполнении каждого модуля, включающего этот заголовочный файл, происходит инициализация. Этот код не связан с большими затратами, но при этом приходится загружать соответствующие страницы исполняемого файла, а эта операция может быть затратной. Как правило, в программу следует включать только совершенно необходимые заголовочные файлы. В частности, заголовочные файлы должны включать только заголовок <iosfwd>, а соответствующие файлы реализации – заголовок с полным определением.

4.2. Стандартные потоковые операции << и >>

В языках Си и C++ операции << и >> используются для сдвига битов целых чисел вправо и влево соответственно. Классы basic_istream и basic_ostream перегружают операции >> и << для выполнения стандартного ввода-вывода. Таким образом, в языке C++ операции сдвига становятся операциями ввода-вывода.

Пример.

```
#include <iostream>
int main(){
        int i;
        cin >> i;
        cout << "Вы ввели " << i;
        return 0;
}
```

Операции ввода и вывода в качестве результата своего выполнения формируют ссылку на объект типа istream для ввода и ссылку на ostream – для вывода. Это позволяет формировать цепочки операций, что проиллюстрировано последним оператором приведенного примера. Вывод при этом выполняется слева направо.

Как и для других перегруженных операций, для ввода и вывода невозможно изменить приоритеты, поэтому в необходимых случаях используются скобки:

```
// Скобки не требуются - приоритет сложения больше, чем <<:
cout << i + j;
// Скобки необходимы - приоритет операции отношения меньше, чем <<:
cout << (i < j);
cout << (i << j ); // Правая операция << означает сдвиг
```

Величины при вводе должны разделяться пробельными символами (пробелами, знаками табуляции или перевода строки). Ввод прекращается, если очередной символ оказался недопустимым.

Если в операции вывода в поток встречается выражение, изменяющее некоторую переменную, то она не должна присутствовать в цепочке операций более одного раза, поскольку в таком случае результат может зависеть от реализации компилятора.

Операции << и >> перегружены для почти всех встроенных типов данных, что позволяет автоматически выполнять ввод и вывод в соответствии с типом величин. Это означает, что при вводе последовательность символов преобразуется во внутреннее представление величины, стоящей справа от знака ввода, а при выводе выполняется обратное преобразование, например:

```
#include <iostream>
```

```
int main(){
        int i = 0xD;
        double d;
        // Символы из потока ввода преобразуются в double:
        cin >> d;
        // int и double преобразуются в строку символов:
        cout << i << ' ' << d;
        return 0;
}
```

Рассмотрим, как обрабатываются с помощью этих операций данные различных типов.

Числовые значения можно вводить в десятичной или шестнадцатеричной системе счисления (с префиксом 0x) со знаком или без знака. Вещественные числа представляются в форме с фиксированной точкой или с порядком. Например, если для предыдущего примера с клавиатуры вводится последовательность символов 1.53e-2, она интерпретируется как вещественное число с порядком и преобразуется во внутреннее представление, соответствующее типу double. При выводе выполняется обратное преобразование, и на экран выводятся символы:

13 0.0153

Поскольку ввод буферизован, помещение в буфер ввода происходит после нажатия клавиши перевода строки, после чего из буфера выполняется операция извлечения из потока. Это дает возможность исправлять введенные символы до того, как нажата клавиша Enter.

При вводе строк извлечение происходит до ближайшего пробела (вместо него в строку заносится нуль-символ):

```
char str1[100], str2[100];
cin >> strl >> str2;
```

Если с клавиатуры вводится строка "раз два три четыре пять", переменные strl и str2 примут значения "раз" и "два" соответственно, а остаток строки воспринят не будет. При необходимости ввести из входного потока строку целиком (до символа '\n') пользуются методами get или getline (см. ниже).

Значения указателей выводятся в шестнадцатеричной системе счисления. Под любую величину при выводе отводится столько позиций, сколько требуется для ее представления. Чтобы отделить одну величину от другой, используются пробелы:

```
cout << i << ' ' << d << " " << j;
```

По умолчанию булевы величины выводятся и считываются как числа: значение false преобразовывается в 0 или из 0, а значение true преобразовывается в 1 или из 1. При чтении значения, отличающиеся от 0 и 1, считаются ошибочными. Для потока можно также задать режим форматирования, при котором булевы величины вводятся и выводятся как символьные строки.

Если формат вывода, используемый по умолчанию, не устраивает программиста, он может скорректировать его с помощью методов классов ввода/вывода, флагов форматирования и так называемых манипуляторов. Об этом рассказывается ниже.

4.3. Файловые потоки

Под файлом обычно подразумевается именованная информация на внешнем носителе, например, на жестком магнитном диске. Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатуру и принтер также можно рассматривать как частные случаи файлов.

По способу доступа файлы можно разделить на последовательные, чтение и запись в которых производятся с начала байт за байтом, и файлы с произвольным доступом, допускающие чтение и запись в указанную позицию.

Потоки можно использовать для доступа к файлам. Стандартная библиотека C++ содержит четыре шаблонных класса, для которых заранее определены следующие стандартные специализации.

1. Шаблонный класс basic_ifstream<> со специализациями ifstream и wifstream, обеспечивающими чтение файлов ("файловый поток ввода").

2. Шаблонный класс basic_ofstream<> со специализациями ofstream и wofstream, обеспечивающими вывод в файлы ("файловый поток вывода").

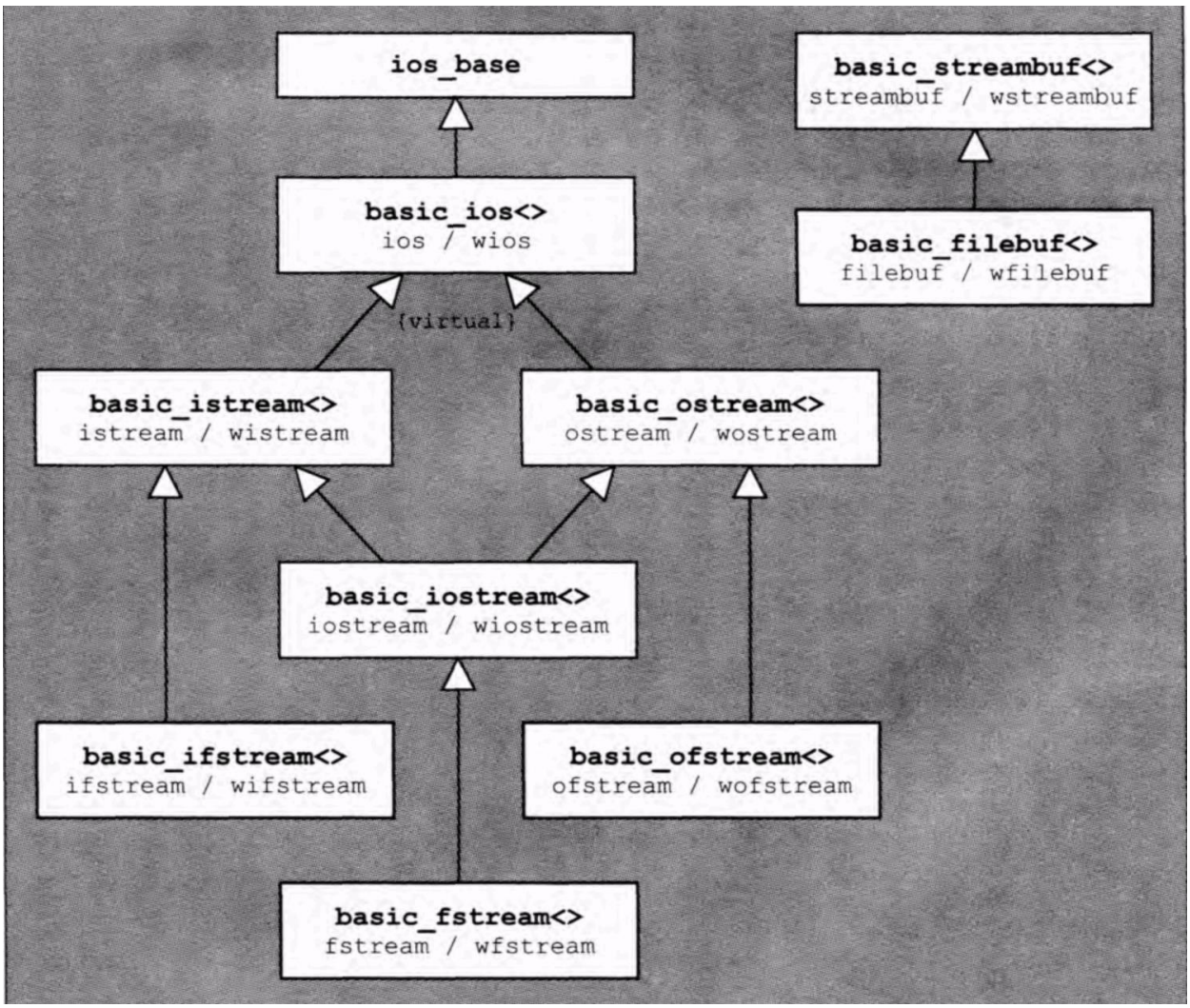
3. Шаблонный класс basic_fstream<> со специализациями fstream и wfstream, обеспечивающими чтение и запись файлов.

4. Шаблонный класс basic_filebuf<> со специализациями filebuf и wfilebuf, которые используются другими классами файловых потоков для фактического чтения и записи символов.

Как показано на рисунке, эти классы связаны с базовыми классами потоков и объявлены в заголовочном файле <fstream> следующим образом:

```
namespace std {
    template < typename charT,
               typename traits = char_traits<charT>>
               class basic_ifstream;
    typedef basic_ifstream<char> ifstream;
    typedef basic_ifstream<wchar_t> wifstream;

    template < typename charT,
               typename traits = char_traits<charT>>
               class basic_ofstream;
    typedef basic_ofstream<char> ofstream;
    typedef basic_ofstream<wchar_t> wofstream;
```



template < typename charT,
typename traits = char_traits<charT>>
class **basic_fstream**;
typedef basic_fstream<char> **fstream**;
typedef basic_fstream<wchar_t> **wfstream**;
template <typename charT,
typename traits = char_traits<charT>>
class **basic_filebuf**;
typedef basic_filebuf<char> **filebuf**;
typedef basic_filebuf<wchar_t> **wfilebuf**;
}

Использование файлов в программе предполагает следующие операции:
• создание потока;
• открытие потока и связывание его с файлом;
• обмен (ввод/вывод);
• уничтожение потока;
• закрытие файла.

Каждый класс файловых потоков содержит конструкторы, с помощью которых можно создавать объекты этих классов различными способами.

• Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом:

ifstream();

ofstream();

fstream();

• Конструкторы с параметрами создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

ifstream(const char *name, int mode = ios::in);

ofstream(const char *name, int mode = ios::out);

fstream(const char *name, int mode = ios::in | ios::out);

Начиная со стандарта C++11, задавать имя файла можно не только строкой языка Си (C-строкой), но и строкой библиотечного класса string.

Вторым параметром конструктора является режим открытия файла. Если установленное по умолчанию значение не устраивает программиста, можно указать другое, составив его из флагов (битовых масок), определенных в классе ios_base (программисты предпочитают обращаться к ним не через класс ios_base, а через класс ios, что короче и соответствует старой версии библиотеки).

Флаг	**Описание**
in	Открыть для ввода (по умолчанию для ifstream)
out	Открыть для вывода (по умолчанию для ofstream)
app	Всегда добавлять в конец при выводе
ate	Установить курсор на конец файла после открытия ("at end")
trunc	Удалить предыдущее содержимое файла
binary	Не заменять специальные символы

Флаг binary конфигурирует поток так, чтобы подавить преобразование специальных символов или символьных последовательностей, например конца строки или конца файла. В операционных системах, таких как Windows и OS/2, конец строки в текстовом файле представлен двумя символами (CR и LF). В обычном текстовом режиме (когда флаг binary не установлен) при вводе и выводе символ перехода на новую строку заменяется двухсимвольной последовательностью, и наоборот. В бинарном режиме (когда флаг binary установлен) эти преобразования не выполняются.

Флаг binary необходимо использовать всегда, если файл не содержит специальных символов, а его содержимое обрабатывается как двоичные данные. Примером является копирование файла путем ввода символ за символом и их вывода без модификации. Если файл обрабатывается как текстовый, флаг устанавливать не следует, потому что при этом требуется специальная обработка перехода на новую строку. Например, переход на новую строку может кодироваться двумя символами.

В таблице перечислены разные комбинации флагов в интерфейсе функции fopen() из языка Си, предназначенной для открытия файлов. Комбинации флагов binary и ate не указаны. Установка флага binary означает, что к строке приписывается символ b, а установка флага ate означает переход в конец файла сразу после его открытия. Остальные комбинации флагов, не приведенные в таблице, такие как trunc | app, не разрешаются. До принятия стандарта C++11 флаги app, in | app и in | out | app не были определены.

Флаги ios_base	**Описание**	**Обозначение режима в языке C**
in	Читать (файл должен существовать)	"r"
out	Стереть и записать (создать файл при необходимости)	"w"
out | trunc	Стереть и записать (создать файл при необходимости)	"w"
out | app	Добавить (создать файл при необходимости)	"a"
app	Добавить (создать файл при необходимости)	"a"
in | out	Читать и записывать; исходная позиция расположена в начале (файл должен существовать)	"r+"
in | out | trunc	Стереть, читать и записать (создать файл при необходимости)	"w+"
in | app	Дописать в конец (создать файл при необходимости)	"a+"
in | out | app	Дописать в конец (создать файл при необходимости)	"a+"

Открыть файл в программе можно с использованием либо конструкторов, либо метода open, имеющего такие же параметры, как и в соответствующем конструкторе, например:

```
ifstream inpf ("input.txt"); //Использование конструктора
if(!inpf){
cout << "Невозможно открыть файл для чтения";
return 1;
}
ofstream f;
f.open("output.txt"); // Использование метода open
if (!f){
cout << "Невозможно открыть файл для записи";
return 1;
}
```

Для закрытия потока определен метод close(), но поскольку он неявно выполняется деструктором, явный вызов необходим только тогда, когда требуется закрыть поток раньше конца его области видимости.

Ввод и вывод выполняются либо с помощью операций << и >>, либо с помощью методов классов (см. ниже).

4.4. Строковые потоки

Механизмы потоковых классов можно также использовать для чтения данных из строк и записи данных в строки. Строковые потоки имеют буфер, но не связаны с каналами ввода-вывода. Буфером и строкой можно манипулировать с помощью специальных функций. В основном эта возможность используется для обеспечения независимости фактического ввода-вывода. Например, текст для вывода можно форматировать в строке, а затем послать в канал вывода или прочитать данные строка за строкой и обработать их с помощью строковых потоков.

До появления стандарта C++98 классы строковых потоков использовали для представления строк тип char* (C-строк). В настоящее время используется библиотечный тип string (или, более широко, тип basic_string<>). Старые строковые потоковые классы также являются частью стандартной библиотеки C++, но объявлены **нежелательными средствами**. Таким образом, их не следует использовать в новых программах и необходимо постепенно заменять их в унаследованном коде.

Для строк типа string определены следующие потоковые классы, аналогичные потоковым классам для файлов.

• Шаблонный класс basic_istringstream<> со специализациями istringstream и wistringstream для ввода данных из строк ("строковый поток ввода").

• Шаблонный класс basic_ostringstream<> со специализациями ostringstream и wostringstream для вывода данных в строки ("строковый поток вывода").

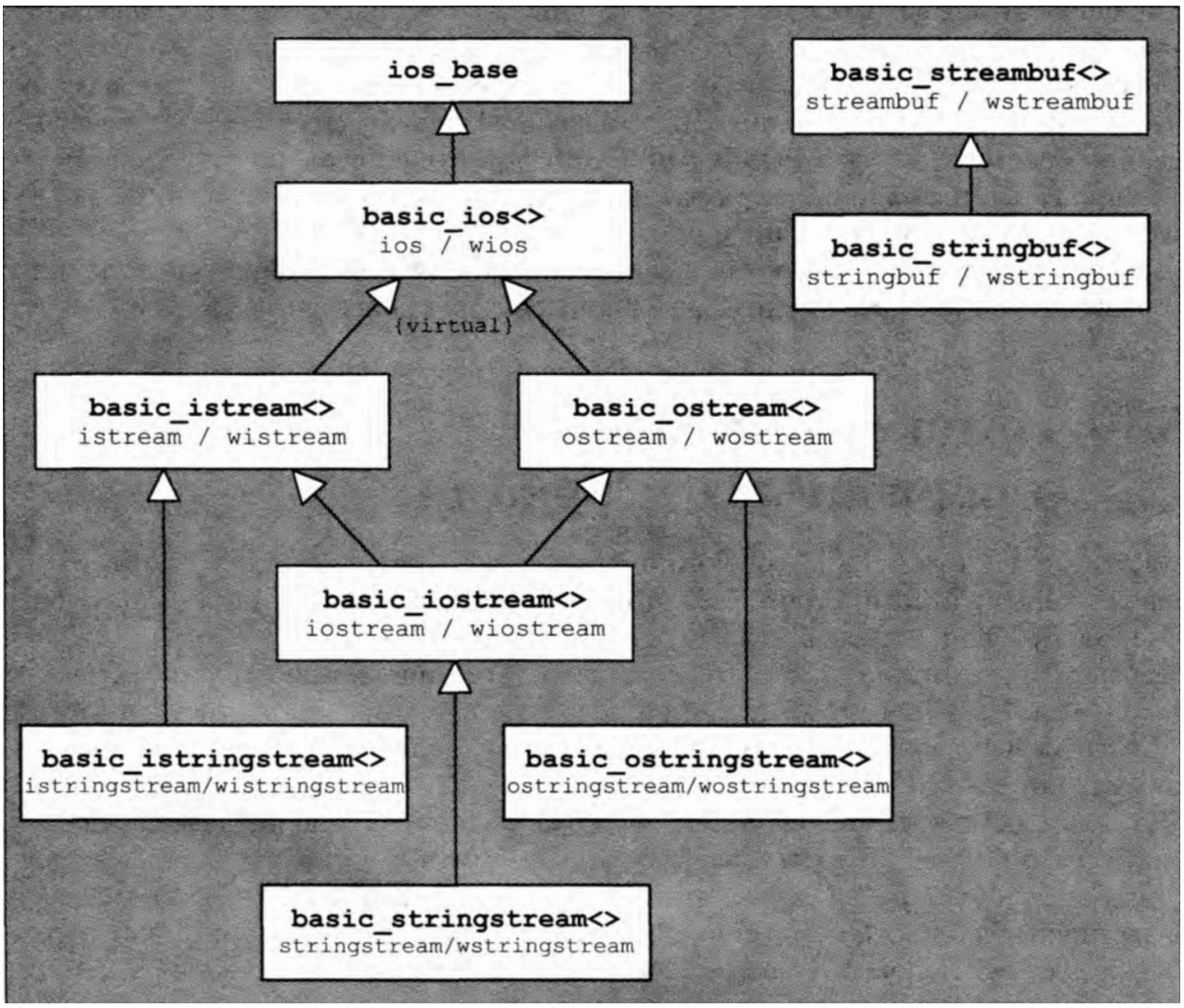
• Шаблонный класс basic_stringstream<> со специализациями stringstream и wstringstream для ввода данных из строк и вывода данных в строки.

• Шаблонный класс basic_stringbuf<> со специализациями stringbuf и wstringbuf, используемыми другими строковыми потоковыми классами для ввода и вывода символов

Эти классы имеют такое же отношение к базовым потоковым классам, как и файловые потоковые классы. Эта иерархия классов изображена на рисунке.

Классы объявлены в заголовочном файле <sstream> следующим образом:

```
namespace std {
template <typename charT, typename traits = char_traits<charT>,
typename Allocator = allocator<charT>> class basic_istringstream;
typedef basic_istringstream<char> istringstrearn;
typedef basic_istringstream<wchar_t> wistringstream;
template <typename charT, typename traits = char_traits<charT>,
typename Allocator = allocator<charT>> class basic_ostringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
```



template <typename charT, typename traits = char_traits<charT>,
typename Allocator = allocator<charT>> class **basic_stringstream**;
typedef basic_stringstream<char> **stringstream**;
typedef basic_stringstream<wchar_t> **wstringstream**;
template <typename charT, typename traits = char_traits<charT>,
typename Allocator = allocator<charT>> class **basic_stringbuf**;
typedef basic_stringbuf<char> **stringbuf**;
typedef basic_stringbuf<wchar_t> **wstringbuf**;
}

Строковые потоки (здесь и далее – для класса string) создаются и связываются со строками с помощью конструкторов:

explicit istringstream(int mode = ios::in);
explicit istringstream(const string& name, int mode = ios::in);
explicit ostringstream(int mode = ios::out);
explicit ostringstream(const string& name, int mode = ios::out);
explicit stringstream(int mode = ios::in | ios::out);
explicit stringstream(const strings name, int mode = ios::in | ios::out);

Применение строковых потоков аналогично применению файловых потоков. Отличие заключается в том, что физически информация потока размещается в оперативной памяти, а не в файле на диске.

В интерфейсе строковых потоковых классов описана функция-член str(), которая используется для управления буфером строкового потокового класса:

string str() const – возвращает буфер в виде строки;

void str(const string& s) – задает содержимое буфера с помощью аргумента string.

Пример. Воспользуемся строковым потоком для формирования сообщения, включающего текущее время и передаваемый в качестве параметра номер:

```
#include <sstream>
#include <string>
#include <iostream>
#include <ctime>
string message(int i){
        ostringstream s;
        time_t t;
        time(&t);
        s << "time: " << ctime(&t) << " number: " << i << endl;
        return s.str():
}
int main(){
        cout << message(22);
        return 0;
}
```

Запись данных в строковый поток вывода часто используется пользовательскими операциями вывода (см. ниже).

Строковые потоки ввода используются в основном для форматированного чтения данных из существующих строк. Например, часто проще прочитать данные строка за строкой, а затем анализировать каждую строку отдельно. Следующие операторы считывают целочисленную переменную x со значением 3 и переменную с плавающей точкой f со значением 0.7 из строки s:

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

Для того чтобы удалить текущее содержимое из потока, можно использовать функцию str (), записывающую в буфер новое содержимое:

```
is.str("");
```

4.5. Форматирование данных

В потоковых классах форматирование выполняется тремя способами с помощью флагов, форматирующих методов и манипуляторов.

Флаги и форматирующие методы.

Класс ios_base имеет набор флагов (типа fmtflags), управляющих различными аспектами форматирования, такими как основание системы счисления и признак необходимости отображения завершающих нулей. Включение флага называется установкой флага (или бита) и означает установку его значения в 1.

Флаг	Описание действия при установленном бите
skipws	При вводе символы-разделители игнорируются
left	Выравнивание по левому краю поля (символы-заполнители справа)
right	Выравнивание по правому краю поля (символы-заполнители слева)
internal	Знак числа выводится по левому краю, число – по правому. Между ними выводятся символы-заполнители.
boolalpha	Использование символьного представление для true и false
dec	Десятичная система счисления
hex	Шестнадцатеричная система счисления
oct	Восьмеричная система счисления
scientific	Печатать вещественные числа в форме с мантиссой и порядком
fixed	Печатать вещественные числа в форме с фиксированной точкой
showbase	Выводится основание системы счисления (0x для шестнадцатеричных чисел и 0 для восьмеричных)
showpoint	При печати вещественных чисел выводятся незначащие нули
showpos	Печатается знак при выводе положительных чисел
uppercase	При выводе чисел используются символы верхнего регистра (1.2E20, 0X1A2)
unitbuf	Очистка буфера после каждой операции вывода
adjustfleld	left | right | internal (флаги для выравнивания полей)
basefield	dec | oct | hex (флаги системы счисления целых)
floatfield	scientific | fixed (флаги вывода чисел с плавающей запятой)

Для управления флагами используются методы flags, setf и unsetf:
fmtflags flags() const; // читать флаги
fmtflags flags(fmtflags f); //установить флаги, возвращает прежние флаги

```
fmtflags setf(fmtflags f) {return flags(flags() | f);}//добавить флаг
fmtflags setf(fmtflags f, fmtflags mask) {return flags(flags() | (f&mask));}// добав-
                                                                    // ление флага
void unsetf(fmtflags mask) {flags(flags() &~mask);} // очистить флаги
```

Значения флагов зависят от реализации. Используйте только их символические имена, а не соответствующие им числовые значения, даже если вы уверены в этих значениях для вашей конкретной реализации.

```
const ios::fmtflags my_opt = ios::left | ios::oct | ios::fixed;
```

Предоставление интерфейса управления в виде набора флагов вкупе с функциями для их установки и сброса хоть и выглядит несколько старомодно, зато проверено временем. Главное достоинство такого подхода заключается в том, что пользователь может легко комбинировать имеющиеся опции. Например:

```
void your_function(ios::fmtflags opt)
{
        ios::fmtflags old_options = cout.flags(opt); //установить новые
        // и сохранить старые опции
        . . . . .
        cout.flags(old_options); //восстановить старые опции
}
void my_function ()
{
        your_function (my_opt);
}
```

После установки флаги сохраняют свое состояние до тех пор, пока не будут сброшены.

С помощью функции

```
copyfmt(stream)
```

можно копировать всю информацию о форматировании из одного потока в другой.

Перед установкой некоторых флагов требуется сбросить флаги, которые не могут быть установлены одновременно с ними. Для этого удобно использовать вторым параметром метода setf флаги adjustfleld, basefield, floatfield.

Кроме флагов, для форматирования используются следующие свойства: минимальная ширина поля вывода; количество цифр в дробной части при выводе вещественных чисел с фиксированной точкой или общее количество значащих цифр при выводе в форме с мантиссой и порядком; символ заполнения поля вывода.

Для управления этими свойствами используются методы width, precision и fill:

streamsize ios::width() – возвращает значение ширины поля вывода;

streamsize ios::width(streamsize) – устанавливает ширину поля вывода в соответствии со значением параметра;

streamsize ios::precision() – возвращает значение точности представления при выводе вещественных чисел;

streamsize ios::precision(streamsize) – устанавливает значение точности представления при выводе вещественных чисел, возвращает старое значение точности;

Ch fill() – возвращает текущий символ заполнения;

Ch fill(Ch) – устанавливает значение текущего символа заполнения, возвращает старое значение символа.

Тип streamsize – размер потока (целое число со знаком, например, это может быть int). Ch – используемый символьный тип (например, char).

По умолчанию символом-заполнителем является пробел, ширина поля вывода по умолчанию равна нулю, что трактуется как «столько символов, сколько нужно». Значение по умолчанию ширины поля вывода можно восстановить с помощью вызова

cout.width(0);

Вызов width(n) устанавливает минимальное число символов равным n. При наличии большего числа символов все они будут выведены. Например:

cout.width(4); cout << "abcdef";

выводит abcdef, а не abcd. Лучше получить некрасивый, но правильный вывод, чем прекрасный с виду, но неправильный.

Вызов width(n) влияет только на одну, непосредственно следующую операцию вывода <<:

cout.width (4); cout.fill ('#'); cout << 12 << ':' << 13;

Данный код выведет ##12:13, а не ##12###:##13, что имело бы место в случае, если вызов width (4) влиял бы на все последующие операции вывода. В последнем случае нам пришлось бы то и дело вызывать width() практически для всех выводимых величин.

Манипуляторы.

Манипуляторами называются функции или объекты, которые можно включать в цепочку операций ввода и вывода для форматирования данных. Манипуляторы делятся на *простые,* не требующие указания аргументов, и *параметризованные.* Пользоваться манипуляторами более удобно, чем методами установки флагов форматирования.

Как и операции ввода и вывода, манипулятор возвращает потоковый объект, к которому он применяется; таким образом, можно объединить манипуляторы и данные в один оператор.

Рассмотрим простые манипуляторы (знак * означает стандартное состояние потока).

Манипулятор	**Значение**
boolalpha	Отображать значения true и false как строки
*noboolalpha	Отображать значения true и false как 0 и 1
showbase	Создавать префикс, означающий базу целочисленных

	значений
*noshowbase	Не создавать префикс базы чисел
showpoint	Всегда отображать десятичную точку для значений с плавающей запятой
*noshowpoint	Отображать десятичную точку, только если у значения есть дробная часть
showpos	Отображать + для положительных чисел
*noshowpos	Не отображать + в неотрицательных числах
uppercase	Выводить "OX" в шестнадцатеричной и "E" в экспоненциальной формах записи
*nouppercase	Выводить "Ox" в шестнадцатеричной и "e" в экспоненциальной формах записи
*dec	Отображать целочисленные значения с десятичной базой числа
hex	Отображать целочисленные значения с шестнадцатеричной базой числа
oct	Отображать целочисленные значения с восьмеричной базой числа
left	Добавлять заполняющие символы справа от значения
*right	Добавлять заполняющие символы слева от значения
internal	Добавлять заполняющие символы между знаком и значением
fixed	Отображать значения с плавающей точкой в десятичном представлении
scientific	Отображать значения с плавающей точкой в экспоненциальном представлении
hexfloat	Отображать значения с плавающей точкой в шестнадцатеричном представлении (C++11)
*defaultfloat	Отображать значения с плавающей точкой в наиболее удобочитаемом виде (C++11)
unitbuf	Сбрасывать буфер после каждой операции вывода
*nounitbuf	Восстановить обычный сброс буфера
*skipws	Пропускать символы-разделители в операторах ввода
noskipws	Не пропускать символы-разделители в операторах ввода
flush	Сбросить буфер объекта ostream
ends	Вставить нулевой символ '\0', а затем сбросить буфер объекта ostream
endl	Вставить символ перехода новую строку '\n', а затем сбросить буфер объекта ostream

Рассмотрим некоторые манипуляторы, требующие указания аргумента, т.е. параметризированные манипуляторы. Для их использования требуется подключить к программе заголовочный файл <iomanip>.

Манипулятор	**Значение**
setfill(ch)	Определить символ ch как символ-заполнитель
setprecision(n)	Установить точность n числа с плавающей точкой
setw(w)	Читать или писать значение в w символов
setbase(b)	Задает основание системы счисления b (8, 10, 16)
resetiosflags(flags)	Сбросить флаги состояния потока, биты которых установлены в параметре
setiosflags(flags)	Установить флаги состояния потока, биты которых в параметре равны 1

Большинство манипуляторов изменяет флаг формата. Манипуляторы используются для двух общих категорий управления выводом: контроль представления числовых значений, а также контроль количества и расположения заполнителей. Большинство манипуляторов, изменяющих флаг формата, предоставлены парами для установки и сброса; один манипулятор устанавливает флаг формата в новое значение, а другой сбрасывает его, восстанавливая стандартное значение.

Манипуляторы, изменяющие флаг формата потока, обычно оставляют флаг формата измененным для всего последующего ввода-вывода. Тот факт, что манипулятор вносит постоянное изменение во флаг формата, может оказаться полезным, когда имеется ряд операций ввода-вывода, использующих одинаковое форматирование. Действительно, некоторые программы используют эту особенность манипуляторов для изменения поведения одного или нескольких правил форматирования ввода или вывода. В таких случаях факт изменения потока является желательным.

Но большинство программ (и что еще важней, разработчиков) ожидают, что состояние потока будет соответствовать стандартным библиотечным значениям. В этих случаях оставленный в нестандартном состоянии поток может привести к ошибке. В результате обычно лучше отменить изменение состояния, как только оно больше не нужно.

Пример. Управление форматом логических значений. Манипулятор boolalpha изменяет состояние формата вывода логических значений. По умолчанию значение типа bool выводится как 1 или 0. Значение true выводится как целое число 1, а значение false как 0. Это поведение можно переопределить, применив к потоку манипулятор boolalpha:

```
cout << "default bool values: " << true << " " << false << "\nalpha bool values: " << boolalpha << true << " " << false << endl;
```

Эта программа выводит следующее:
```
default bool values: 1 0
alpha bool values: true false
```

Как только манипулятор boolalpha «записан» в поток cout, способ вывода

логических значений изменяется. Последующие операции вывода логических значений отобразят их как «true» или «false».

Чтобы отменить изменение флага формата потока cout, применяется манипулятор noboolalpha:

```
bool bool_val = get_status();
cout << boolalpha    // устанавливает внутреннее состояние cout
<< bool_val
<< noboolalpha; // возвращает стандартное внутреннее состояние
```

Здесь формат вывода логических значений изменен только для вывода значения bool_val. Как только это значение будет выведено, поток немедленно возвращается в первоначальное состояние.

Пример. Использование параметризованных манипуляторов:

```
#include <iostream>
#include <iomanip>
int main(){
        double d[ ] = {1.234, -12.34567, 123.456789, -1.234, 0.00001};
        cout   << setfill('.') << setprecision(4)
                << setiosflags(ios::showpoint | ios::fixed);
        for (int i = 0; i < 5: i++)
        cout << setw(12) << d[i] << endl;
        return 0;
}
```

Результат работы программы:

```
......1.2340
....-12.3457
....123.4568
.....-1.2340
......0.0000
```

Вывод вещественных чисел.

Контролировать можно три аспекта вывода вещественного числа.

• Количество выводимых цифр точности.

• Выводится ли число в шестнадцатеричном формате, как десятичное число с фиксированной точкой или в экспоненциальном представлении.

• Выводится ли десятичная точка для значений, не имеющих дробной части.

По умолчанию вещественные значения выводятся с шестью цифрами точности; десятичная точка не отображается при отсутствии дробной части; в зависимости от величины числа используется формат с фиксированной точкой или экспоненциальная форма. Библиотека выбирает формат, увеличивающий удобочитаемость числа (аналог формата %g в языке Си). Очень большие и очень маленькие значения выводятся в экспоненциальном представлении. Другие значения выводятся в формате с фиксированной точкой.

По умолчанию точность контролирует общее количество отображаемых цифр. При выводе значение с плавающей запятой округляется (а не усекается) до текущей точности. Таким образом, если текущая точность четыре, то число 3.14159 становится 3.142; если точность три, то оно выводится как 3.14.

Для изменения точности можно воспользоваться функцией precision() или манипулятором setprecision. Функция precision() перегружена. Одна ее версия получает значение типа int и устанавливает точность в это новое значение. Она возвращает предыдущее значение точности. Другая версия не получает никаких аргументов и возвращает текущее значение точности. Манипулятор setprecision получает аргумент, который и использует для установки точности.

Следующая программа иллюстрирует различные способы контроля точности при выводе значения с плавающей точкой:

```
// cout.precision() сообщает текущее значение точности
cout << "Precision: " << cout.precision() << ", Value: " << sqrt(2.0) << endl;
// cout.precision(12) запрашивает вывод 12 цифр точности
cout.precision(12);
cout << "Precision: " << cout.precision() << ", Value: " << sqrt(2.0) << endl;
// альтернативный способ установки точности с использованием
// манипулятора setprecision
cout << setprecision(3);
cout << "Precision: " << cout.precision() << ", Value: " << sqrt(2.0) << endl;
```

Эта программа выводит следующее:

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

Используя соответствующий манипулятор, можно заставить поток использовать научную, фиксированную или шестнадцатеричную форму записи. Манипулятор scientific задает использование экспоненциального представления. Манипулятор fixed задает использование фиксированных десятичных чисел.

В стандарте C++11 можно выводить значения с плавающей точкой в шестнадцатеричном формате (со степенью 2) при помощи манипулятора hexfloat. Например, число 234.5 записывается в виде 0x1.d5p+7 (0x1.d5 умножить на 2^7, т.е. 1*128 + 13*128/16 + 5*128/256).

Стандарт C++11 предоставляет еще один манипулятор, defaultfloat. Он возвращает поток в стандартное состояние, при котором выбор формы записи осуществляется на основании выводимого значения.

Манипуляторы изменяют также заданное для потока по умолчанию значение точности. После применения манипуляторов scientific, fixed или hexfloat значение точности контролирует количество цифр после десятичной точки. По умолчанию точность определяет общее количество цифр до и после десятичной точки. Манипуляторы fixed и scientific позволяют выводить числа, выстроенные в столбцы, с десятичной точкой в фиксированной позиции относительно дробной части.

Пример. Вывод числа 100*sqrt(2.0) в разных форматах.

```
cout << "default format: " << 100*sqrt(2.0) << '\n'
<< "scientific: " << scientific << 100*sqrt(2.0) << '\n'
<< "fixed decimal: " << fixed << 100*sqrt(2.0) << '\n'
<< "hexadecimal: " << hexfloat << 100*sqrt(2.0) << '\n'
<< "use defaults: " << defaultfloat << 100*sqrt(2.0)
<< "\n\n";
```

Получается следующий вывод:

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcp+7
use defaults: 141.421
```

По умолчанию шестнадцатеричные цифры и символ "e", используемый в экспоненциальном представлении, выводятся в нижнем регистре. Манипулятор uppercase позволяет выводить эти значения в верхнем регистре.

По умолчанию, когда дробная часть значения с плавающей точкой равна нулю, десятичная точка не отображается. Манипулятор showpoint требует отображать десятичную точку всегда:

```
cout << 10.0 << endl;        //  выводит 10
cout << showpoint << 10.0    //  выводит 10.0000
<< noshowpoint << endl; //  возвращает стандартный формат десятичной
                        //  точки
```

Манипулятор noshowpoint восстанавливает стандартное поведение. У вывода следующих выражений будет стандартное поведение, подразумевающее отсутствие десятичной точки, если дробная часть значения с плавающей точкой отсутствует.

4.6. Методы обмена с потоками

В потоковых классах наряду с операциями ввода >> и вывода << определены методы для неформатированного чтения и записи в поток (при этом преобразования данных не выполняются).

Ниже приведены функции чтения, определенные в классе istream.

gcount() – возвращает количество символов, считанных с помощью последней функции неформатированного ввода;

get() – возвращает код извлеченного из потока символа или EOF (end of file – конец файла);

get(C) – возвращает ссылку на поток, из которого выполнялось чтение, и записывает извлеченный символ в C;

get(buf, num, lim='\n') – считывает num-1 символов (или пока не встретится символ lim) и копирует их в символьную строку buf. Вместо символа lim в стро-

ку записывается признак конца строки ('\0'). Символ lim остается в потоке. Возвращает ссылку на текущий поток;

getline(buf, num, lim='\n') – аналогична функции get, но копирует в buf и символ lim;

ignore(num = 1, lim = EOF) – считывает и пропускает символы до тех пор, пока не будет прочитано num символов или не встретится разделитель, заданный параметром lim. Возвращает ссылку на текущий поток;

peek() – возвращает следующий символ без удаления его из потока или EOF, если достигнут конец файла;

putback(C) – помещает в поток символ C, который становится текущим при извлечении из потока;

read(buf, num) – считывает num символов (или все символы до конца файла, если их меньше num) в символьный массив buf и возвращает ссылку на текущий поток;

readsome(buf, num) – считывает num символов (или все символы до конца файла, если их меньше num) в символьный массив buf и возвращает количество считанных символов;

seekg(pos) – устанавливает текущую позицию чтения в значение pos;

seekg(offs, org) – перемещает текущую позицию чтения на offs байтов, считая от одной из трех позиций, определяемых параметром org: ios::beg (от начала потока), ios::cur (от текущей позиции) или ios::end (от конца потока);

tellg() – возвращает текущую позицию чтения потока;

unget() – помещает последний прочитанный символ в поток и возвращает ссылку на текущий поток.

В классе ostream определены аналогичные функции для неформатированного вывода:

flush() – записывает содержимое потока вывода на физическое устройство;

put(C) – выводит в поток символ C и возвращает ссылку на поток;

seekg(pos) – устанавливает текущую позицию записи в значение pos;

seekg (offs, org) – перемещает текущую позицию записи на offs байтов, считая от одной из трех позиций, определяемых параметром org: ios::beg (от начала потока), ios::cur (от текущей позиции) или ios::end (от конца потока);

tellg() – возвращает текущую позицию записи потока;

write(buf, num) – записывает в поток num символов из массива buf и возвращает ссылку на поток.

Пример. Программа считывает строки из входного потока в символьный массив.

```
#include <iostream>
int main(){
    const int N = 20, Len = 100;
    char str[N][Len];
    int i = 0;
```

```
        while (cin.getline(str[i], Len,  '\n') && i<N){
                . . . . .
                i++;
        }
        return 0;
}
```

Пример. В приведенной ниже программе формируется файл test, в который выводится три строки.

```
#include <fstream>
#include <string.h>
int main(){
// Запись в файл
        ofstream out("test");
        if(!out) {
                cout << "Cannot open file 'test' for writing" << endl;
                return 1;
        }
        char *str[ ] = {"This is the first line.", "This is the second line.",
        "This is the third line."};
        for (int i = 0; i<3; ++i){
                out.write(str[i], strlen(str[i]));
                out.put('\n');
        }
        out.close();
// Чтение из файла
        ifstream in("test");
        if(!in){
                cout << "Cannot open file 'test' for reading" << endl;
                return 1;
        }
        char check_str[3][60];
        for (int i = 0; i<3; ++i){
                in.get(check_str[i], 60);
                in.get();
        }
// Контрольный вывод
        for (i = 0; i<3; ++i) cout << check_str[i] << endl;
        in.close();
        return 0;
}
```

После выполнения функции get(check_str[i], 60) символ-разделитель строк '\n' остается во входном потоке, поэтому необходим вызов get() для пропуска одного символа. Альтернативным способом является использование вместо

функции get функции getline, которая извлекает символ-ограничитель из входного потока.

Пример. Функции peek() и putback() позволяют упростить управление, когда неизвестен тип вводимой в каждый конкретный момент времени информации. Следующая программа иллюстрирует это. В ней из файла считываются либо строки, либо целые. Строки и целые могут следовать в любом порядке.

```
#include <fstream>
#include <ctype.h>
#include <stdlib.h>
int main(){
        char ch;
// Подготовка файла
        ofstream out("test");
        if(!out) {
                cout << "Cannot open file 'test' for writing" << endl;
                return 1;
        }
        char str[80], *p;
        out << 123 << "this is a test" << 23;
        out << "Hello there!" << 99 << "bye" << endl;
        out.close();
// Чтение файла
        ifstream in("test");
        if(!in){
                cout << "Cannot open file 'test' for reading" << endl;
                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 << "Number: " << 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 << "String: " << str;
                }
```

```
            else in.get();        // пропуск
            cout << endl;
        }while(!in.eof()); // пока не конец файла (in.eof() - см. ниже)
        in.close();
        return 0;
}
```

Результат работы программы:

```
Number:  123
String: this

String: is

String:  a

String: test
Number:  23
String: Hello

String: there

Number: 99
String: bye
```

При организации диалогов с пользователем программы при помощи потоков необходимо учитывать буферизацию. Например, при выводе приглашения к вводу мы не можем гарантировать, что оно появится раньше, чем будут считаны данные из входного потока, поскольку приглашение появится на экране только при заполнении буфера вывода:

```
cout << "Введите x";
cin >> x;
```

Для решения этой проблемы в basic_ios определена функция tie(), которая связывает потоки istream и ostream с помощью вызова вида cin.tie(&cout). После этого вывод очищается (то есть выполняется функция cout.flush()) каждый раз, когда требуется новый символ из потока ввода.

4.7. Состояние потока

Общее состояние потоков определяется несколькими константами типа iostate, представляющими собой флаги, они определены в классе ios_base.

Рассмотрим эти константы и их значения:

goodbit – нет ошибок;

eofbit – достигнут конец файла;

failbit – ошибка форматирования или преобразования;

badbit – серьезная ошибка, после которой пользоваться потоком невозможно.

По определению флаг goodbit равен 0. Таким образом, установка флага goodbit означает, что все остальные биты сброшены. Имя goodbit может в определенной степени вводить в заблуждение, потому что оно на самом деле означает, что все остальные биты сброшены.

Разница между флагами failbit и badbit состоит в том, что флаг badbit служит индикатором более серьезной ошибки.

• Бит failbit устанавливается, если операция не была выполнена правильно, но поток остался в исправном состоянии. Обычно этот флаг является результатом ошибок форматирования при чтении данных. Например, этот флаг устанавливается, если должно считываться целое число, а следующим символом является буква.

• Бит badbit устанавливается при повреждении потока или потере данных. Например, этот флаг устанавливается, когда указатель в файловом потоке ссылается на позицию, предшествующую началу файла.

Бит eofbit обычно устанавливается одновременно с битом failbit, потому что конец файла проверяется и обнаруживается при попытке чтения за концом файла. После считывания последнего символа флаг eofbit еще не установлен. Следующая попытка считать символ установит биты eofbit и failbit, потому что чтение невозможно.

Текущее состояние флагов можно определить с помощью следующих функций.

Функция	**Описание**
good()	Возвращает true, если поток находится в работоспособном состоянии (установлен бит goodbit)
eof()	Возвращает true, если обнаружен конец файла (установлен бит eofbit)
fail()	Возвращает true, если обнаружена ошибка (установлен бит failbit или badbit)
bad()	Возвращает true, если обнаружена фатальная ошибка (установлен бит badbit)
rdstate()	Возвращает флаги, установленные в данный момент
clear()	Сбрасывает все флаги
clear(state)	Сбрасывает и устанавливает флаги состояния state
setstate(state)	Устанавливает дополнительные флаги состояния state

Первые четыре функции выясняют определенные состояния и возвращают булево значение. Функция fail() возвращает признак того, что установлен бит failbit или badbit. Преимущество этих функций заключается в том, что всего лишь одна проверка позволяет определить, возникла ли ошибка.

Кроме того, состояние флагов можно определить и изменить с помощью более общих функций. Когда функция clear() вызывается без параметров, все флаги ошибок, включая eofbit, сбрасываются (именно это означает слово clear).

Если функция clear() получает параметр, то состояние потока уточняется по этому параметру. Другими словами, для потока устанавливается набор флагов, заданных параметром, а остальные флаги сбрасываются.

Далее приведены часто используемые операции с флагами состояния потока.

```
// Проверить, установлен ли флаг flag:
if(strm.rdstate() & ios::flag);
// Сбросить флаг flag:
strm.clear(rdstate() & ~ios::flag);
// Установить флаг flag:
strm.clear(rdstate() | ios::flag);
// Установить флаг flag и сбросить все остальные:
strm.clear(ios::flag);
// Сбросить все флаги:
strm.clear();
```

Все биты всегда следует сбрасывать явным образом. В языке Си было возможным читать символы после обнаружения ошибки форматирования. Например, если функция scanf() не могла прочитать целое число, можно было прочитать остальные символы. Таким образом, операция чтения выполнялась неправильно, но поток ввода оставался в корректном состоянии. В языке C++ ситуация иная. Если установлен бит failbit, то каждая следующая потоковая операция становится фиктивной, пока флаг failbit не будет сброшен явным образом.

Пример. Выполним проверку флага failbit и его сбрасывание (по необходимости):

```
if (strm.rdstate() & ios::failbit) {// проверяем, установлен ли флаг failbit
    cout << "failbit was set" << endl;
    // сбрасываем только флаг failbit
    strm.clear (strm.rdstate() & ~ios::failbit);
}
```

Состояние потока и булевы условия.

Для использования потоков в логических выражениях используются две функции

operator bool()	проверяет, находится ли поток в нормальном состоянии (соответствует !fail());
operator !()	проверяет, находится ли поток в ошибочном состоянии (соответствует fail()).

Функция operator bool() позволяет лаконично проверять текущее состояние потоков в управляющих конструкциях.

```
while (cin) { // проверяем,   находится стандартный поток ввода
. . . . .                    // в нормальном состоянии
}
```

Для проверки логического условия в управляющей конструкции не обязательно непосредственно преобразовывать тип в bool. Достаточно одного преобразования в целочисленный тип, например int или char, или в тип указателя. Преобразование в тип bool часто используется для чтения объектов и проверки его успешности в одном и том же выражении.

```
if (cin >> x)   { // чтение x было успешным
. . . . .
}
```

Как мы уже говорили, следующее выражение возвращает объект cin:

```
cin >> x
```

Таким образом, после того как будет прочитан объект x, проверка выглядит как

```
if (cin)   {
. . . . .
}
```

Поскольку объект cin используется в контексте проверки условия, вызывается функция operator bool(), проверяющая, находится ли поток в нормальном состоянии.

Типичное применение этого способа – цикл, в котором выполняются чтение и обработка объектов:

```
while (cin >> obj)   {// пока возможно чтение объекта obj
// обрабатываем объект obj   (в данном случае просто выводим его на печать)
cout << obj  << endl;
}
```

Цикл заканчивается, если устанавливается флаг failbit или badbit. Это происходит, если возникает ошибка или обнаруживается конец файла (напомним, что попытка прочитать данные за концом файла приводит к установке флагов eofbit и failbit).

Операция operator !() позволяет выполнить обратную проверку. Она проверяет, находится ли поток в ошибочном состоянии, т.е. операция возвращает значение true, если установлен флаг failbit или badbit. Операцию можно использовать следующим образом:

```
if (!cin) {// поток cin не находится в нормальном состоянии
. . . . .
}
```

Аналогично неявным преобразованиям к булевому значению, этот оператор часто используется для проверки успеха одновременно с чтением объекта в одном выражении.

```
if (!(cin >> x)) { // сбой при чтении
. . . . .
}
```

Здесь следующее выражение возвращает объект cin, к которому применяется оператор !:

```
cin >> x
```

Выражение после оператора ! должно быть заключено в круглые скобки из-за правил приоритета операторов: без круглых скобок оператор ! выполнялся бы первым. Иначе говоря, выражение

```
!cin >> x
```

эквивалентно выражению

```
(!cin) >> x
```

Вероятно, это не то, к чему стремился программист.

Несмотря на то, что эти операторы очень удобно использовать в булевых выражениях, можно заметить одну особенность: двойное отрицание не возвращает исходный объект:

• cin является объектом потока класса istream.

• !!cin – это булево значение, описывающее состояние потока cin.

Использование функций-членов, таких как fail(), обычно повышает читабельность программ.

```
cin >> x;
if (cin.fail()){
. . . . .
}
```

4.8. Потоки и типы, определенные пользователем

Как упоминалось ранее, главным преимуществом потоков над старыми средствами ввода-вывода в языке Си является возможность расширения потокового механизма на пользовательские типы. Для этого необходимо перегрузить операции << и >>.

Реализация операций вывода.

В выражениях, содержащих операцию вывода <<, левый операнд определяет поток данных, а правый – объект, который записывается в этот поток:

поток << объект

В соответствии с правилами языка C++ это можно интерпретировать следующим образом.

1. Как *поток*.*operator << (объект)*.
2. Как *operator << (поток, объект)*.

Первый способ используется для встроенных типов. Для пользовательских типов необходимо применять второй способ, потому что потоковые классы закрыты для расширения. Все, что нужно сделать для этого – реализовать глобальный оператор << для пользовательского типа. Это довольно просто, если не требуется получать доступ к закрытым членам объектов.

Пример. Опишем класс Complex.

```
class Complex {
        double re;
        double im;
public:
        Complex (double r=0, double i=0){re=r; im=i;}
        double get_re() const {return re;}
        void set_re(double r){re=r;}
        double get_im() const {return im;}
        void set_im(double i){im=i;}
};
```

Чтобы вывести на печать объект класса Complex, можно написать следующую функцию.

```
inline ostream& operator << (ostream& out, const Complex& c)
{
        out << "(" << c.get_re() << "," << c.get_im() << ")";
        return out;
}
```

Эта функция выводит комплексное число в виде (re,im) в поток данных, передаваемый в виде аргумента. Поток может быть стандартным, файловым, строковым или каким-то еще. Для создания цепочек операций вывода и проверки состояния потока одновременно с выводом функция должна возвращать ссылку на поток.

У этой простой формы есть недостатки. Поскольку в сигнатуре функции используется класс ostream, ее можно применять только к потокам символов char. Если функция предназначается только для таких потоков, проблемы не возникают. С другой стороны, создать более универсальную версию несложно, поэтому следует, по крайней мере, рассмотреть такую возможность.

Другая проблема возникает при задании ширины поля. В данном случае результат окажется не тем, который можно было бы ожидать. Ширина поля будет относиться только к ближайшей операции вывода, в данном случае – к выводу открывающейся скобки. Таким образом, операторы

```
        Complex c(3,4);
        cout<<left; cout.fill('.'); // заполнитель точка используется для наглядности
        cout.width(8);
        cout << c << endl;
```

выведут

```
(.....3,4)
```

Следующая версия решает обе проблемы:

```
template <typename charT, typename traits> inline basic_ostream<charT, traits>& operator << (basic_ostream<charT, traits>& out, const Complex& c)
{
        basic_ostringstream<charT, traits> s; // создаем строковый поток s
        s.copyfmt(out); // копируем форматы потока out в форматы потока s
```

```
    s.width(0); // устанавливаем "нулевую" ширину для потока s
    s << "(" << c.get_re() << "," << c.get_im() << ")"; // выводим в строку
    out << s.str(); // выводим строку в заданный поток
    return out;
}
```

Оператор превратился в шаблонную функцию, параметризованную для всех разновидностей потоков.

Проблема с шириной поля решается с помощью предварительной записи комплексного числа в строковый поток без указания конкретной ширины. Созданная строка затем передается в поток, заданный аргументом. В результате символьное представление комплексного числа выводится с помощью одной операции вывода, к которой и применяется заданная ширина поля. В результате операторы

```
    Complex c(3,4);
    cout<<left; cout.fill('.');
    cout.width(8);
    cout << c << endl;
```

выведут

(3,4)...

Реализация операций ввода.

Операции ввода реализуются в соответствии с теми же принципами, что и операции вывода. Однако при вводе могут возникнуть проблемы чтения. Как правило, функции чтения должны особым образом обрабатывать неудачный ввод.

При реализации функции чтения можно выбирать между простотой и гибкостью. Например, в следующей функции используется простой подход: комплексное число считывается в виде (re,im) без проверки возможных ошибок:

```
inline istream& operator >> (istream& in, Complex& c)
{
    double r, i;
    char ch1, ch2, ch3 ;
    in >> ch1 >> r >> ch2 >> i >> ch3;
    c = Complex(r,i);
    return in;
}
```

Проблема заключается в том, что такая реализация подходит только для потоков данных с символьным типом char. Кроме того, она не проверяет, действительно ли комплексные числа в потоке находятся в нужном формате.

Поскольку на практике ошибки форматирования обычно регистрируются на уровне потоков данных, лучше в этом случае установить флаг ios::failbit.

В заключение объект, переданный по ссылке, можно модифицировать даже при неудачном вводе. Это может произойти, например, когда действи-

тельная часть вводится успешно, а при чтении мнимой части возникает сбой. Такое поведение противоречит общепринятым правилам, установленным стандартными операциями ввода, и его лучше всего избегать. **Операция чтения должна либо завершаться успешно, либо не иметь последствий.**

Рассмотрим улучшенную реализацию программы, в которой указанные проблемы не возникают.

```
template <typename charT, typename traits> inline basic_istream<charT, traits>&
operator >> (basic_istream<charT, traits>& in, Complex& c)
{
        double r, i;
        char ch1, ch2, ch3;
        in >> ch1 >> r >> ch2 >> i >> ch3;
        if (!in) return in;
        if ( ch1 != '(' || ch2 != ',' || ch3 != ')' ){
        // Ошибка формата
                in.clear (ios::failbit);
                return in;
        }
        c = Complex(r,i);
        return in;
}
```

В функции проверяется состояние потока данных после чтения, а также соответствие считанных элементов формату. При несоответствии формату устанавливается флаг ios::failbit. Новое значение присваивается комплексному числу, только если ввод был выполнен без ошибок.

Ввод и вывод с помощью вспомогательных функций.

Если реализация операции ввода-вывода требует доступа к закрытым данным объекта, то стандартные операции должны делегировать фактическую работу вспомогательным функциям-членам классов. Этот подход позволяет создавать полиморфные функции чтения и записи. Рассмотрим пример:

```
class Complex {
. . . . .
public:
        virtual void printOn (ostream& out) const;  // вывод
        virtual void scanFrom (istream& in);      // ввод
. . . . .
};
ostream& operator << (ostream& out, const Complex& c)
{
        c.printOn (out);
        return out;
}
```

```
istream& operator >> (istream& in, Complex& c)
{
        c.scanFrom (in);
        return in;
}
```

Типичным примером является прямой доступ к действительной и мнимой частям во время ввода.

```
void Complex::scanFrom (istream& in) {
. . . . .
// присваиваем значения непосредственно компонентам
re = r; im = i;
}
```

Если класс не будет использоваться в качестве базового, то операции ввода-вывода можно объявить дружественными для этого класса. Однако такой подход значительно ограничивает возможности при наследовании. Дружественные функции не могут быть виртуальными, поэтому могут быть вызваны неправильные функции. Например, если в аргументе операции ввода передается ссылка на базовый класс, которая на самом деле ссылается на объект производного класса, то для нее будет вызвана операция из базового класса.

Таким образом, описанная выше реализация является более универсальной, чем реализация на основе дружественных функций. Она должна рассматриваться как стандартный подход, хотя дружественные функции тоже часто применяются.

Соглашения по созданию пользовательских операций ввода-вывода.

Ниже перечислены некоторые соглашения, которые должны соблюдаться при реализации пользовательских операторов ввода-вывода. Эти соглашения соответствуют типичному поведению стандартных операторов.

Формат **вывода** должен допускать определение оператора **ввода**, читающего данные без потери информации. Для строк эта задача практически невыполнима из-за проблем с пробелами. Пробел внутри строки невозможно отличить от пробела, разделяющего две строки.

При вводе-выводе должна учитываться текущая спецификация формата потока. Прежде всего, это относится к ширине поля при выводе.

При возникновении ошибок должен быть установлен соответствующий флаг состояния.

Ошибки не должны изменять состояние объекта. Если операция читает несколько объектов данных, промежуточные результаты сохраняются во вспомогательных объектах до окончательного принятия значения.

Вывод не должен завершаться символом перехода на новую строку, в основном из-за того, что это не позволяет выводить другие объекты в той же строке.

Даже слишком большие данные должны считываться полностью. После

чтения необходимо установить соответствующий флаг ошибки, а возвращаемое значение должно содержать полезную информацию, например, максимальное значение.

При обнаружении ошибки форматирования по возможности никакие символы не должны быть считаны.

4.9. Как работают манипуляторы

Манипуляторы реализуются с помощью очень простого приема, который не только позволяет легко управлять потоками, но и демонстрирует мощь механизма перегрузки функций. Манипуляторы – это всего лишь функции, передаваемые операторам ввода-вывода как аргументы. Затем оператор вызывает эти функции. Например, оператор вывода в классе ostream обычно перегружается следующим образом:

```
ostream& ostream::operator << (ostream& (*op)(ostream&))
{
// вызываем функцию, переданную как параметр и получающую поток
//как аргумент
return (*op)(*this);
}
```

Аргумент op – это указатель на функцию, которая получает поток ostream в качестве аргумента и возвращает его обратно. Если второй операнд оператора << является такой функцией, то она вызывается, получая в качестве аргумента первый операнд оператора <<.

Пример. Манипулятор endl (т.е. функция, endl()) для потока ostream реализуется примерно так:

```
ostream& endl (ostream& out)
{
// записываем символ перехода на новую строку
out.put('\n');
// выгружаем буфер вывода
out.flush();
// возвращаем out, чтобы разрешить образование цепочек
return out;
}
```

Этот манипулятор можно использовать в выражениях следующим образом:

```
cout << endl;
```

Здесь оператор << применяется к потоку cout с функцией endl() в качестве второго операнда. Реализация оператора << преобразует это выполнение в вызов переданной функции с аргументом, представляющим собой поток:

```
endl(cout);
```

Того же самого эффекта можно достигнуть, выполнив это выражение непосредственно.

Поскольку потоковые классы являются шаблонными классами, параметризованными символьным типом, настоящая реализация манипулятора endl() выглядит следующим образом:

```
template <typename charT, typename traits>
basic_ostream<charT, traits>& endl (basic_ostream<charT, traits>& out)
{
out.put(out.widen('\n'));
out.flush();
return out;
}
```

Функция widen() используется для преобразования символа перехода на новую строку в набор символов, используемый потоком в настоящее время.

Точный способ обработки аргументов манипулятора зависит от реализации, и не существует стандартного способа реализации пользовательских манипуляторов с аргументами (пример приведен в следующем разделе).

4.10. Пользовательские манипуляторы

Для определения собственного манипулятора необходимо написать функцию наподобие endl(). Например, следующая функция определяет манипулятор, игнорирующий все символы вплоть до конца строки:

```
#include <istream>
#include <limits>
template <typename charT, typename traits>
inline basic_istream<charT, traits>&
ignoreLine (basic_istream<charT, traits>& in)
{
    // пропускаем до конца строки
    in.ignore(numeric_limits<streamsize>::max(),in.widen('\n'));
    // возвращаем поток
    return in;
}
```

Этот манипулятор делегирует работу функции ignore(), игнорирующей все символы до конца строки. Выражением

```
numeric_limits<streamsize>::max()
```

определяется максимальная длина строки. Вызов функции ignore(max, c) игнорирует все символы, пока в потоке ввода не будет обнаружен символ ‘c’, не будет считано max символов или не будет достигнут конец потока. Однако, прежде чем функция вернет значение, этот символ также будет проигнорирован.

Применение этого манипулятора очень простое:

```
// игнорируем оставшуюся часть строки
cin >> ignoreLine;
```

Применяя манипулятор несколько раз, можно пропустить несколько строк:

```
// игнорируем две строки
cin >> ignoreLine >> ignoreLine;
```

Данный манипулятор может быть удобным использовать для обработки ошибок формата при консольном вводе:

```
while(!(cin>>c))
{
    if(ios::failbit){
        cin.clear();
        cin >> ignoreLine;
        cout << "Ошибка ввода." << endl;
        cout << "Введите еще раз:" << endl;
    }
    else . . . . .
}
cout << "Вы ввели:" << endl;
cout << c << endl;
```

Как уже было сказано, существует несколько способов определить собственный манипулятор с аргументами. Например, следующий манипулятор игнорирует n строк:

```
#include <istream>
#include <limits>
class ignoreLine
{
private:
    int num;
public:
    explicit ignoreLine (int n=1) : num(n) {}
    template <typename charT, typename traits> friend
    basic_istream<charT, traits>& operator >> (basic_istream<charT, traits>& in,
    const ignoreLine& ign)
    {
    // пропускаем символы, пока символ конца строки
    //не будет обнаружен num раз
        for (int i=0; i<ign.num; ++i)
        in.ignore(numeric_limits< streamsize>::max(), in.widen('\n'));
    // возвращаем поток
    return in;
    }
};
```

Здесь манипулятор ignoreLine – это класс, получающий аргумент для инициализации, а операция ввода перегружена для объектов этого класса. Отметим, что перегруженная операция ввода реализована как дружественная.

Теперь пропустить две строки можно следующим образом:

```
// игнорируем две строки
cin >> ignoreLine(2);
```

4.11. Пользовательские флаги форматов

При создании пользовательских операций ввода-вывода часто желательно иметь специальные флаги форматирования, соответствующие этим операциям и устанавливаемые соответствующими манипуляторами. Например, было бы хорошо, если бы в операции вывода для комплексного числа в зависимости от желания пользователя использовались разные форматы: например, число выдавалось или в виде пары (4,5), или в алгебраической форме 4+5i.

Эту возможность предоставляют объекты потоков – в них предусмотрен механизм связывания данных с потоком. Он позволяет задать нужные значения, например, с помощью манипулятора, и прочитать их позднее. В классе ios_base определены две функции, iword() и pword(), которые при вызове получают индекс типа int и возвращают по нему соответствующее значение long& или void*&. Предполагается, что функции iword() и pword() обращаются к объектам типа long или void* в массиве произвольного размера, хранящемся в объекте потока. Флаги форматирования, сохраняемые для потока, располагаются по одному и тому же индексу для всех потоков. Статическая функция-член xalloc() класса ios_base используется для получения индекса, который еще не применялся для этой цели.

В исходном состоянии объекты, доступ к которым осуществляется функциями iword() и pword(), равны 0. Это значение может использоваться для форматирования по умолчанию или как признак того, что к данным еще не обращались. Рассмотрим пример.

```
// получаем индекс для новых данных в потоке ostream
const int complex_index = ios::xalloc();
// определяем манипуляторы для установки этих данных
template <typename charT, typename traits> basic_ostream <charT, traits>&
complex_algebraic (basic_ostream<charT, traits>& out)
{
        out.iword(complex_index) = true;
        return out;
}
template <typename charT, typename traits> basic_ostream<charT, traits>&
complex_noalgebraic (basic_ostream<charT, traits>& out)
{
        out.iword(complex_index) = false;
        return out;
}
template <typename charT, typename traits> basic_ostream<charT, traits>&
operator << (basic_ostream<charT, traits>& out, Complex& c)
```

```
{
	basic_ostringstream <charT, traits> s;
	s.copyfmt(out);
	s.width(0);
	if (out.iword(complex_index)) {
		double r=c.get_re(), i=c.get_im();
		if(!r&&!i) s << 0;
		else
			if(r){ s << r;
				if(i>0) s << "+";
			}
		if(i) s<<i<<"i";
	}
	else
		s << "("<< c.get_re() << "," <<c.get_im()<<")";
	out << s.str();
	return out;
}
```

В этом примере используется простой подход к реализации операции вывода, потому что его основная цель – демонстрация функции iword(). Флаг форматирования считается булевым значением, определяющим, следует ли при выводе использовать алгебраическую форму комплексного числа. В первой строке функция ios::xalloc() возвращает индекс, который может использоваться для хранения флага форматирования. Результат вызова сохраняется в константе, поскольку это значение никогда не изменяется. Функции complex_algebraic и complex_noalgebraic – это манипуляторы присваивания значения true или false переменной типа long, хранящейся по индексу complex_index в целочисленном массиве, связанном с потоком out. Операция вывода извлекает это значение и выводит комплексное число в соответствии с состоянием флага.

Функции iword () и pword() возвращают ссылки на объекты типа long или void*. Эти ссылки остаются корректными только до следующего вызова функции iword() или pword() с соответствующим объектом потока данных или до уничтожения объекта потока. Обычно результаты функций iword() и pword() сохраняться не должны. Предполагается, что доступ происходит достаточно быстро, хотя хранение данных в массиве не обязательно.

Выведем комплексное число в обоих форматах.

```
void main ()
{
	Complex c(2,3);
	cout << complex_algebraic << c << endl;
	cout << complex_noalgebraic << c << endl;
}
```

Программа выводит следующий результат.

2+3i
(2,3)