38

I/O Streams

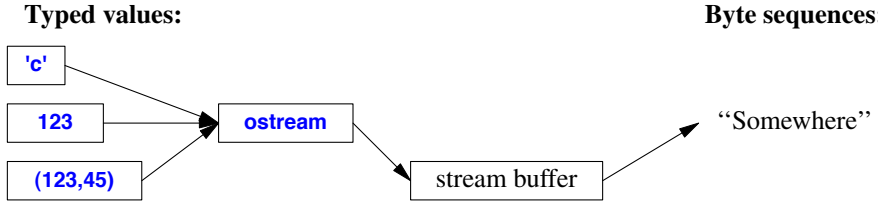
What you see is all you get.
– Brian W. Kernighan

- Introduction
- The I/O Stream Hierarchy
  File Streams; String Streams
- Error Handling
- I/O Operations
  Input Operations; Output Operations; Manipulators; Stream State; Formatting
- Stream Iterators
- Buffering
  Output Streams and Buffers; Input Streams and Buffers; Buffer Iterators
- Advice

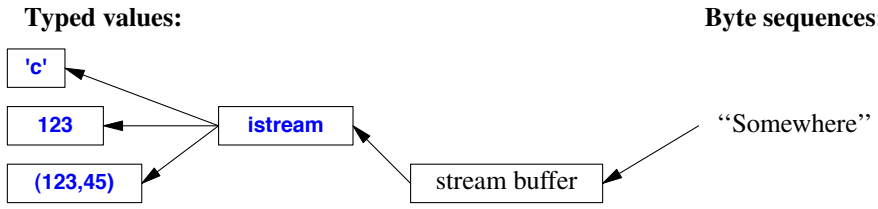
38.1 Introduction

The I/O stream library provides formatted and unformatted buffered I/O of text and numeric values. The definitions for I/O stream facilities are found in **<istream>**, **<ostream>**, etc.; see §30.2.

An **ostream** converts typed objects to a stream of characters (bytes):



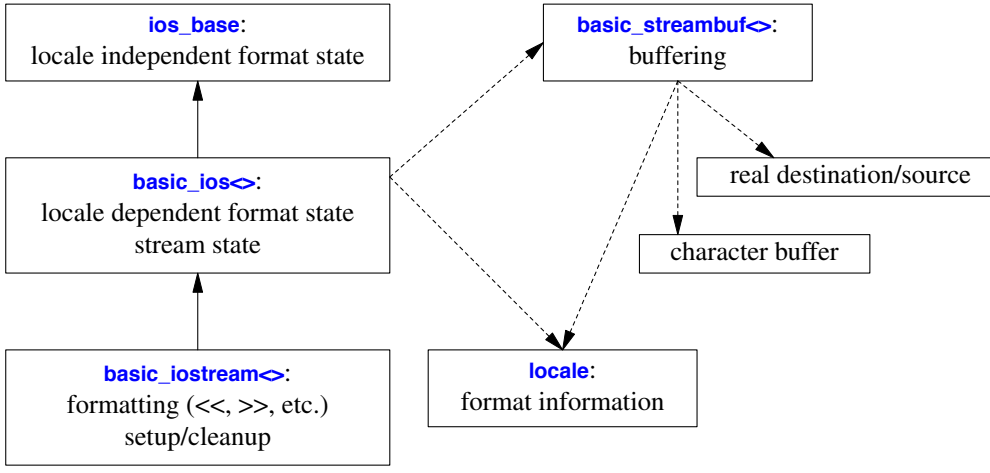
An **istream** converts a stream of characters (bytes) to typed objects:



An **iostream** is a stream that can act as both an **istream** and an **ostream**. The buffers in the diagrams are stream buffers (**streambufs**; §38.6). You need them to define a mapping from an **iostream** to a new kind of device, file, or memory. The operations on **istream**s and **ostream**s are described in §38.4.1 and §38.4.2.

Knowledge of the techniques used to implement the stream library is not needed to use the library. So I present only the general ideas needed to understand and use **iostream**s. If you need to implement the standard streams, provide a new kind of stream, or provide a new locale, you need a copy of the standard, a good systems manual, and examples of working code in addition to what is presented here.

The key components of the stream I/O system can be represented graphically like this:



The solid arrows represent "derived from." The dotted arrows represent "pointer to." The classes marked with **<>** are templates parameterized by a character type and containing a **locale**.

The I/O stream operations:

- Are type-safe and type sensitive
- Are extensible (when someone designs a new type, matching I/O stream operators can be added without modifying existing code)

- Are locale sensitive (Chapter 39)
- Are efficient (though their potential is not always fully realized)
- Are interoperable with C-style stdio (§43.3)
- Include formatted, unformatted, and character-level operations

The **basic_iostream** is defined based on **basic_istream** (§38.6.2) and **basic_ostream** (§38.6.1):

```
template<typename C, typename Tr = char_traits<C>>
class basic_iostream :
        public basic_istream<C,Tr>, public basic_ostream<C,Tr> {
public:
        using char_type = C;
        using int_type = typename Tr::int_type;
        using pos_type = typename Tr::pos_type;
        using off_type = typename Tr::off_type;
        using traits_type = Tr;

        explicit basic_iostream(basic_streambuf<C,Tr>* sb);
        virtual ~basic_iostream();
protected:
        basic_iostream(const basic_iostream& rhs) = delete;
        basic_iostream(basic_iostream&& rhs);

        basic_iostream& operator=(const basic_iostream& rhs) = delete;
        basic_iostream& operator=(basic_iostream&& rhs);
        void swap(basic_iostream& rhs);
};
```

The template parameters specify the character type and the traits used to manipulate characters (§36.2.2), respectively.

Note that no copy operations are provided: sharing or cloning the fairly complex state of a stream would be difficult to implement and expensive to use. The move operations are intended for use by derived classes and are therefore **protected**. Moving an **iostream** without moving the state of its defining derived class (e.g., an **fstream**) would lead to errors.

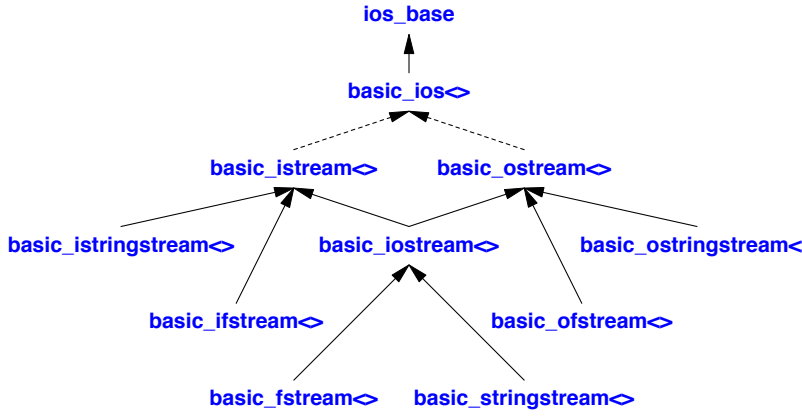
There are three standard streams:

Standard I/O Streams	
cout	The standard character output (often by default a screen)
cin	The standard character input (often by default a keyboard)
cerr	The standard character error output (unbuffered)
clog	The standard character error output (buffered)
wcin	**wistream** version of **cin**
wcout	**wostream** version of **cout**
wcerr	**wostream** version of **cerr**
wclog	**wostream** version of **clog**

Forward declarations for stream types and stream objects are provided in **<iosfwd>**.

38.2 The I/O Stream Hierarchy

An **istream** can be connected to an input device (e.g., a keyboard), a file, or a **string**. Similarly, an **ostream** can be connected to an output device (e.g., a text window or an HTML engine), a file, or a **string**. The I/O stream facilities are organized in a class hierarchy:



The classes suffixed by **<>** are templates parameterized on the character type. A dotted line indicates a virtual base class (§21.3.5).

The key class is **basic_ios** in which most of the implementation and many of the operations are defined. However, most casual (and not-so-casual) users never see it: it is mostly an implementation detail of the streams. It is described in §38.4.4. Most of its facilities are described in the context of their function (e.g., formatting; §38.4.5).

38.2.1 File Streams

In **<fstream>**, the standard library provides streams to and from a file:

- **ifstream**s for reading from a file
- **ofstream**s for writing to a file
- **fstream**s for reading from and writing to a file

The file streams follow a common pattern, so I describe only **fstream**:

```
template<typename C, typename Tr=char_traits<C>>
class basic_fstream
: public basic_iostream<C,Tr> {
public:
        using char_type = C;
        using int_type = typename Tr::int_type;
        using pos_type = typename Tr::pos_type;          // for positions in file
        using off_type = typename Tr::off_type;        // for offsets in file
        using traits_type = Tr;
        // ...
};
```

The set of fstream operations is fairly simple:

basic_fstream<C,Tr> (§iso.27.9)	
fstream fs {};	fs is a file stream not attached to a file
fstream fs {s,m};	fs is a file stream opened for a file called s with mode m; s can be a string or a C-style string
fstream fs {fs2};	Move constructor: fs2 is moved to fs; fs2 becomes unattached
fs=move(fs2)	Move assignment: fs2 is moved to fs; fs2 becomes unattached
fs.swap(fs2)	Exchange the states of fs and fs2
p=fs.rdbuf()	p is a pointer to fs's file stream buffer (basic_filebuf<C,Tr>)
fs.is_open()	Is fs open?
fs.open(s,m)	Open a file called s with mode m and have fs refer to it; sets fs's failbit if it couldn't open the file; s can be a string or a C-style string
fs.close()	Close the file associated with fs (if any)

In addition, the string streams override the basic_ios protected virtual functions underflow(), pbackfail(), overflow(), setbuf(), seekoff(), and seekpos() (§38.6).

A file stream does not have copy operations. If you want two names to refer to the same file stream, use a reference or a pointer, or carefully manipulate file streambufs (§38.6).

If an fstream fails to open, the stream is in the bad() state (§38.3).

There are six file stream aliases defined in <fstream>:

```
using ifstream = basic_ifstream<char>;
using wifstream = basic_ifstream<wchar_t>;
using ofstream = basic_ofstream<char>;
using wofstream = basic_ofstream<wchar_t>;
using fstream = basic_fstream<char>;
using wfstream = basic_fstream<wchar_t>;
```

You can open a file in one of several modes, as specified in ios_base (§38.4.4):

Stream Modes (§iso.27.5.3.1.4)	
ios_base::app	Append (i.e., add to the end of the file)
ios_base::ate	"At end" (open and seek to the end)
ios_base::binary	Binary mode; beware of system-specific behavior
ios_base::in	For reading
ios_base::out	For writing
ios_base::trunc	Truncate the file to 0 length

In each case, the exact effect of opening a file may depend on the operating system, and if an operating system cannot honor a request to open a file in a certain way, the result will be a stream that is in the bad() state (§38.3). For example:

```
ofstream ofs("target");          // "o" for "output" implying ios::out
if (!ofs)
    error("couldn't open 'target' for writing");
```

```
fstream ifs;                    // "i" for "input" implying ios::in
ifs.open("source",ios_base::in);
if (!ifs)
    error("couldn't open 'source' for reading");
```

For positioning in a file, see §38.6.1.

38.2.2 String Streams

In **<sstream>**, the standard library provides streams to and from a **string**:

- **istringstream**s for reading from a **string**
- **ostringstream**s for writing to a **string**
- **stringstream**s for reading from and writing to a **string**

The string streams follow a common pattern, so I describe only **stringstream**:

```
template<typename C, typename Tr = char_traits<C>, typename A = allocator<C>>
class basic_stringstream
    : public basic_iostream<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;      // for positions in string
    using off_type = typename Tr::off_type; // for offsets in string
    using traits_type = Tr;
    using allocator_type = A;

    // ...
```

The **stringstream** operations are:

basic_stringstream<C,Tr,A> (§iso.27.8)	
stringstream ss {m};	**ss** is an empty string stream with mode **m**
stringstream ss {};	Default constructor: **stringstream ss {ios_base::outlios_base::in};**
stringstream ss {s,m};	**ss** is a string stream with its buffer initialized from the **string s** with mode **m**
stringstream ss {s};	**stringstream ss {s,ios_base::outlios_base::in};**
stringstream ss {ss2};	Move constructor: **ss2** is moved to **ss**; **ss2** becomes empty
ss=move(ss2)	Move assignment: **ss2** is moved to **ss**; **ss2** becomes empty
p=ss.rdbuf()	**p** points to **ss**'s string stream buffer (a **basic_stringbuf<C,Tr,A>**)
s=ss.str()	**s** is a **string** copy of the characters in **ss**: **s=ss.rdbuf()–>str()**
ss.str(s)	**ss**'s buffer is initialized from the **string s**: **ss.rdbuf()–>str(s)**; if **ss**'s mode is **ios::ate** (''at end'') values written to **ss** are added after the characters from **s**; otherwise values written overwrites the characters from **s**
ss.swap(ss2)	Exchange the states of **ss** and **ss2**

The open modes are described in §38.4.4. For an **istringstream**, the default mode is **ios_base::in**. For an **ostringstream**, the default mode is **ios_base::out**.

In addition, the string streams override the **basic_ios** protected virtual functions **underflow()**, **pbackfail()**, **overflow()**, **setbuf()**, **seekoff()**, and **seekpos()** (§38.6).

A string stream does not have copy operations. If you want two names to refer to the same string stream, use a reference or a pointer.

There are six string stream aliases defined in **<sstream>**:

```
using istringstream = basic_istringstream<char>;
using wistringstream = basic_istringstream<wchar_t>;
using ostringstream = basic_ostringstream<char>;
using wostringstream = basic_ostringstream<wchar_t>;
using stringstream = basic_stringstream<char>;
using wstringstream = basic_stringstream<wchar_t>;
```

For example:

```
void test()
{
        ostringstream oss {"Label: ",ios::ate};          // write at end
        cout << oss.str() << '\n';  // writes "Label: "
        oss<<"val";
        cout << oss.str() << '\n';  // writes "Label: val" ("val" appended after "Label: ")

        ostringstream oss2 {"Label: "};                  // write at beginning
        cout << oss2.str() << '\n'; // writes "Label: "
        oss2<<"val";
        cout << oss2.str() << '\n'; // writes "valel: " (val overwrites "Label: ")
}
```

I tend to use **str()** only to read a result from an **istringstream**.

It is not possible to directly output a string stream; **str()** must be used:

```
void test2()
{
        istringstream iss;
        iss.str("Foobar");          // Fill iss

        cout << iss << '\n';        // writes 1
        cout << iss.str() << '\n';  // OK: writes "Foobar"
}
```

The reason for the probably surprising **1** is that an **iostream** converts to its state for testing:

```
if (iss) {    // the last operation of iss succeeded; iss's state is good() or eof()
        // ...
}
else {
        // handle problem
}
```

38.3 Error Handling

An **iostream** can be in one of four states, defined in **basic_ios** from **<ios>** (§38.4.4):

Stream States (§iso.27.5.5.4)	
good()	The previous **iostream** operations succeeded
eof()	We hit end-of-input (“end-of-file”)
fail()	Something unexpected happened (e.g., we looked for a digit and found **'x'**)
bad()	Something unexpected and serious happened (e.g., disk read error)

Any operation attempted on a stream that is not in the **good()** state has no effect; it is a no-op. An **iostream** can be used as a condition. In that case, the condition is true (succeeds) if the state of the **iostream** is **good()**. That is the basis for the idiom for reading a stream of values:

```
for (X x; cin>>x;) {    // read into an input buffer of type X
    // ... do something with x ...
}
// we get here when >> couldn't read another X from cin
```

After a read failure, we might be able to clear the stream and proceed:

```
int i;
if (cin>>i) {
    // ... use i ...
} else if (cin.fail()){    // possibly a formatting error
    cin.clear();
    string s;
    if (cin>>s) {    // we might be able to use a string to recover
        // ... use s ...
    }
}
```

Alternatively, errors can be handled using exceptions:

Exception Control: **basic_ios<C,Tr>** (§38.4.4, §iso.27.5.5)	
st=ios.exceptions()	**st** is the **iostate** of **ios**
ios.exceptions(st)	Set **ios**’s **iostate** to **st**

For example, we can make **cin** throw a **basic_ios::failure** when its state is set to **bad()** (e.g., by a **cin.setstate(ios_base::badbit)**):

```
cin.exceptions(cin.exceptions()|ios_base::badbit);
```

For example:

```
struct Io_guard {    // RAII class for iostream exceptions
    iostream& s;
    auto old_e = s.exceptions();
    Io_guard(iostream& ss, ios_base::iostate e) :s{ss} { s.exceptions(s.exceptions()|e); }
    ~Io_guard() { s.exceptions(old_e); }
};
```

```
void use(istream& is)
{
    Io_guard guard(is.ios_base::badbit);
    // ... use is ...
}
catch (ios_base::badbit) {
    // ... bail out! ...
}
```

I tend to use exceptions to handle **iostream** errors that I don't expect to be able to recover from. That usually means all **bad()** exceptions.

38.4 I/O Operations

The complexity of the I/O operations reflects tradition, the need for I/O performance, and the variety of human expectations. The description here is based on the conventional English small character set (ASCII). The ways in which different character sets and different natural languages are handled are described in Chapter 39.

38.4.1 Input Operations

Input operations are provided by **istream** (§38.6.2), found in **<istream>** except for the ones reading into a **string**; those are found in **<string>**. The **basic_istream** is primarily intended as a base class for more specific input classes, such as **istream** and **istringstream**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_istream : virtual public basic_ios<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;
    using off_type = typename Tr::off_type;
    using traits_type = Tr;

    explicit basic_istream(basic_streambuf<C,Tr>* sb);
    virtual ~basic_istream();  // release all resources

    class sentry;
    // ...
protected:
    // move but no copy:
    basic_istream(const basic_istream& rhs) = delete;
    basic_istream(basic_istream&& rhs);
    basic_istream& operator=(const basic_istream& rhs) = delete;
    basic_istream& operator=(basic_istream&& rhs);
    // ...
};
```

To users of an **istream**, the **sentry** class is an implementation detail. It provides common code for standard-library and user-defined input operations. Code that needs to be executed first (the "prefix code") – such as flushing a tied stream – is provided as the **sentry**'s constructor. For example:

```
template<typename C, typename Tr = char_traits<C>>
basic_ostream<C,Tr>& basic_ostream<C,Tr>::operator<<(int i)
{
    sentry s {*this};
    if (!s) {                // check whether all is well for output to start
        setstate(failbit);
        return *this;
    }

    // ... output the int ...
    return *this;
}
```

A **sentry** is used by implementers of input operations rather than by their users.

38.4.1.1 Formatted Input

Formatted input is primarily supplied by the **>>** ("input," "get," or "extraction") operator:

Formatted Input (§iso.27.7.2.2, §iso.21.4.8.9)	
in>>x	Read from **in** into **x** according to **x**'s type; **x** can be an arithmetic type, a pointer, a **basic_string**, a **valarray**, a **basic_streambuf**, or any type for which the user has supplied a suitable **operator>>()**
getline(in,s)	Read a line from **in** into the **string s**

Built-in types are "known" to **istream** (and **ostream**), so if **x** is a built-in type, **cin>>x** means **cin.operator>>(x)**. If **x** is a user-defined type, **cin>>x**, means **operator>>(cin,x)** (§18.2.5). That is, **iostream** input is type sensitive, inherently type-safe, and extensible. A designer of a new type can provide I/O operations without direct access to the implementation of **iostream**.

If a pointer to a function is the target of **>>**, that function will be invoked with the **istream** as its argument. For example, **cin>>pf** yields **pf(cin)**. This is the basis for the input manipulators, such as **skipws** (§38.4.5.2). Output stream manipulators are more common than input stream manipulators, so the technique is explained further in §38.4.3.

Unless otherwise stated, an **istream** operation returns a reference to its **istream**, so that we can "chain" operations. For example:

```
template<typename T1, typename T2>
void read_pair(T1& x, T2& y)
{
    cin >> c1 >> x >> c2 >> y >> c3;
    if (c1!='{' || c2!=',' || c3!='}') {          // unrecoverable input format error
        cin.setstate(ios_base::badbit);           // set badbit
        throw runtime_error("bad read of pair");
    }
}
```

By default >> skips whitespace. For example:

```
for (int i; cin>>i && 0<i;)
    cout << i << '\n';
```

This will take a sequence of whitespace-separated positive integers and print them one to a line.

Skipping of whitespace can be suppressed using noskipws (§38.4.5.2).

The input operations are not virtual. That is, a user cannot do an in>>base where base is a class hierarchy and automatically have the >> resolved to an operation on the appropriate derived class. However, a simple technique can deliver that behavior; see §38.4.2.1. Furthermore, it is possible to extend such a scheme to be able to read objects of essentially arbitrary types from an input stream; see §22.2.4.

38.4.1.2 Unformatted Input

Unformatted input can be used for finer control of reading and potentially for improved performance. One use of unformatted input is the implementation of formatted input:

Unformatted Input (§iso.27.7.2.3, §iso.27.7.2.3)	
x=in.get()	Read one character from in and return its integer value; return EOF for end-of-file
in.get(c)	Read a character from in into c
in.get(p,n,t)	Read at most n characters from in into [p:...); consider t a terminator
in.get(p,n)	in.get(p,n,'\n')
in.getline(p,n,t)	Read at most n characters from in into [p:...); consider t a terminator; remove terminator from in
in.getline(p,n)	in.getline(p,n,'\n')
in.read(p,n)	read at most n characters from in into [p:...)
x=in.gcount()	x is the number of characters read by the most recent unformatted input operation on in
in.putback(c)	Put c back into in's stream buffer
in.unget()	Back up in's stream buffer by one, so that the next character read is the same as the previous character
in.ignore(n,d)	Extract characters from in and discard them until either n characters have been discarded or d is found (and discarded)
in.ignore(n)	in.ignore(n,traits::eof())
in.ignore()	in.ignore(1,traits::eof())
in.swap(in2)	Exchange the values of in and in2

If you have a choice, use formatted input (§38.4.1.1) instead these low-level input functions.

The simple get(c) is useful when you need to compose your values out of characters. The other get() function and getline() read sequences of characters into a fixed-size area [p:...). They read until they reach the maximum number of characters or find their terminator character (by default '\n'). They place a 0 at the end of the characters (if any) written to; getline() removes its terminator from the input, if found, whereas get() does not. For example:

```
void f()    // low-level, old-style line read
{
    char word[MAX_WORD][MAX_LINE];          // MAX_WORD arrays of MAX_LINE char each
    int i = 0;
    while(cin.getline(word[i++],MAX_LINE,'\n') && i<MAX_WORD)
        /* do nothing */ ;
    // ...
}
```

For these functions, it is not immediately obvious what terminated the read:

- We found the terminator.
- We read the maximum number of characters.
- We hit end-of-file.
- There was a non-format input error.

The last two alternatives are handled by looking at the file state (§38.3). Typically, the appropriate actions are quite different for these cases.

A `read(p,n)` does not write a `0` to the array after the characters read. Obviously, the formatted input operators are simpler to use and less error-prone than the unformatted ones.

The following functions depend on the detailed interaction between the stream buffer (§38.6) and the real data source and should be used only if necessary and then very carefully:

Unformatted Input (§iso.27.7.2.3)	
`x=in.peek()`	`x` is the current input character; `x` is not extracted from `in`'s stream buffer and will be the next character read
`n=in.readsome(p,n)`	If `rdbuf()->in_avail()==-1`, call `setstate(eofbit)`; otherwise read at `min(n,most rdbuf()->in_avail())` characters into [`p`:...); `n` is the number of characters read
`x=in.sync()`	Synchronize buffers: `in.rdbuf()->pubsync()`
`pos=in.tellg()`	`pos` is the position of `in`'s get pointer
`in.seekg(pos)`	Place `in`'s get pointer at position `pos`
`in.seekg(off,dir)`	Place `in`'s get pointer at the offset `off` in the direction `dir`

38.4.2 Output Operations

Output operations are provided by `ostream` (§38.6.1), found in `<ostream>` except for the ones writing out a `string`; those are found in `<string>`:

```
template<typename C, typename Tr = char_traits<C>>
class basic_ostream : virtual public basic_ios<C,Tr> {
public:
    using char_type = C;
    using int_type = typename Tr::int_type;
    using pos_type = typename Tr::pos_type;
    using off_type = typename Tr::off_type;
    using traits_type = Tr;
```

```
        explicit basic_ostream(basic_streambuf<char_type,Tr>* sb);
        virtual ~basic_ostream(); // release all resources

        class sentry;   // see §38.4.1
        // ...
    protected:
        // move but no copy:
        basic_ostream(const basic_ostream& rhs) = delete;
        basic_ostream(basic_ostream&& rhs);
        basic_ostream& operator=(basic_ostream& rhs) = delete;
        basic_ostream& operator=(const basic_ostream&& rhs);
        // ...
    };
```

An **ostream** offers formatted output, unformatted output (output of characters), and simple operations on its **streambuf** (§38.6):

Output Operations (§iso.27.7.3.6, §iso.27.7.3.7, §iso.21.4.8.9)	
out<<x	Write **x** to **out** according to **x**'s type; **x** can be an arithmetic type, a pointer, a **basic_string**, a **bitset**, a **complex**, a **valarray**, or any type for which a user has defined a suitable **operator<<()**
out.put(c)	Write the character **c** to **out**
out.write(p,n)	Write the characters [**p**:**p+n**) to **out**
out.flush()	Empty the character buffer to the destination
pos=out.tellp()	**pos** is the position of **out**'s put pointer
out.seekp(pos)	Place **out**'s put pointer at position **pos**
out.seekp(off,dir)	Place **out**'s put pointer at the offset **off** in the direction **dir**

Unless otherwise stated, an **ostream** operation returns a reference to its **ostream**, so that we can "chain" operations. For example:

```
cout << "The value of x is " << x << '\n';
```

Note that **char** values are output as characters rather than small integers. For example:

```
void print_val(char ch)
{
    cout << "the value of '" << ch << "' is " << int{ch} << '\n';
}

void test()
{
    print_val('a');
    print_val('A');
}
```

This prints:

```
the value of 'a' is 97
the value of 'A' is 65
```

Versions of operator << for user-defined types are usually trivial to write:

```
template<typename T>
struct Named_val {
        string name;
        T value;
};

ostream& operator<<(ostream& os, const Named_val& nv)
{
        return os << '{' << nv.name << ':' << nv.value << '}';
}
```

This will work for every `Named_val<X>` where `X` has a << defined. For full generality, << must be defined for `basic_string<C,Tr>`.

38.4.2.1 Virtual Output Functions

The `ostream` members are not `virtual`. The output operations that a programmer can add are not members, so they cannot be `virtual` either. One reason for this is to achieve close to optimal performance for simple operations such as putting a character into a buffer. This is a place where run-time efficiency is often crucial so that inlining is a must. Virtual functions are used to achieve flexibility for the operations dealing with buffer overflow and underflow only (§38.6).

However, a programmer sometimes wants to output an object for which only a base class is known. Since the exact type isn't known, correct output cannot be achieved simply by defining a << for each new type. Instead, a virtual output function can be provided in an abstract base:

```
class My_base {
public:
        // ...
        virtual ostream& put(ostream& s) const = 0;  // write *this to s
};

ostream& operator<<(ostream& s, const My_base& r)
{
        return r.put(s); // use the right put()
}
```

That is, `put()` is a virtual function that ensures that the right output operation is used in <<.

Given that, we can write:

```
class Sometype : public My_base {
public:
        // ...
        ostream& put(ostream& s) const override;     // the real output function
};

void f(const My_base& r, Sometype& s)  // use << which calls the right put()
{
        cout << r << s;
}
```

This integrates the virtual **put()** into the framework provided by **ostream** and **<<**. The technique is generally useful to provide operations that act like virtual functions, but with the run-time selection based on their second argument. This is similar to the technique that under the name *double dispatch* is often used to select an operation based on two dynamic types (§22.3.1). A similar technique can be used to make input operations **virtual** (§22.2.4).

38.4.3 Manipulators

If a pointer to function is given as the second argument to **<<**, the function pointed to is called. For example, **cout<<pf** means **pf(cout)**. Such a function is called a *manipulator*. Manipulators that take arguments can be useful. For example:

```
cout << setprecision(4) << angle;
```

This prints the value of the floating-point variable **angle** with four digits.

To do this, **setprecision** returns an object that is initialized by **4** and calls **cout.precision(4)** when invoked. Such a manipulator is a function object that is invoked by **<<** rather than by **()**. The exact type of that function object is implementation-defined, but it might be defined like this:

```
struct smanip {
        ios_base& (*f)(ios_base&,int);      // function to be called
        int i;                              // value to be used
        smanip(ios_base&(*ff)(ios_base&,int), int ii) :f{ff}, i{ii} { }
};

template<typename C, typename Tr>
basic_ostream<C,Tr>& operator<<(basic_ostream<C,Tr>& os, const smanip& m)
{
        m.f(os,m.i);    // call m's f with m's stored value
        return os;
}
```

We can now define **setprecision()** like this:

```
inline smanip setprecision(int n)
{
        auto h = [](ios_base& s, int x) -> ios_base& { s.precision(x); return s; };
        return smanip(h,n);         // make the function object
}
```

The explicit specification of the return type for the lambda is needed to return a reference. An **ios_base** cannot be copied by a user.

We can now write:

```
cout << setprecision(4) << angle;
```

A programmer can define new manipulators in the style of **smanip** as needed. Doing this does not require modification of the definitions of standard-library templates and classes.

The standard-library manipulators are described in §38.4.5.2.

38.4.4 Stream State

In <ios>, the standard library defines the base class ios_base defining most of the interface to a stream class:

```
template<typename C, typename Tr = char_traits<C>>
class basic_ios : public ios_base {
public:
        using char_type = C;
        using int_type = typename Tr::int_type;
        using pos_type = typename Tr::pos_type;
        using off_type = Tr::off_type;
        using traits_type = Tr;
        // ...
};
```

The basic_ios class manages the state of a stream:

- The mapping between a stream and its buffers (§38.6)
- The formatting options (§38.4.5.1)
- The use of locales (Chapter 39)
- Error handling (§38.3)
- Connections to other streams and stdio (§38.4.4)

It might be the most complicated class in the standard library.

The ios_base holds information that does not depend on template arguments:

```
class ios_base {
public:
        using fmtflags = /* implementation-defined type */;
        using iostate = /* implementation-defined type */;
        using openmode = /* implementation-defined type */;
        using seekdir = /* implementation-defined type */;

        class failure;          // exception class
        class Init;             // initialize standard iostreams
};
```

The implementation-defined types are all *bitmask types*; that is, they support bitwise logical operations, such as & and |. Examples are int (§11.1.2) and bitset (§34.2.2).

The ios_base controls an iostream's connection (or lack thereof) to stdio (§43.3):

Fundamental ios_base Operations (§iso.27.5.3.4)	
ios_base b {};	Default constructor; protected
ios.~ios_base()	Destructor; virtual
b2=sync_with_stdio(b)	If b==true synchronize ios with stdio; otherwise shared buffers might be corrupted; b is the previous synchronization state; static
b=sync_with_stdio()	b=sync_with_stdio(true)

A call of sync_with_stdio(true) before the first iostream operation in the execution of a program

guarantees that the iostream and stdio (§43.3) I/O operations share buffers. A call of sync_with_stdio(false) before the first stream I/O operation prevents buffer sharing and can improve I/O performance significantly on some implementations.

Note that ios_base has no copy or move operations.

ios_base Stream State iostate Member Constants (§iso.27.5.3.1.3)	
badbit	Something unexpected and serious happened (e.g., a disk read error)
failbit	Something unexpected happened (e.g., we looked for a digit and found 'x')
eofbit	We hit end-of-input (e.g., end-of-file)
goodbit	All is well

Functions for reading these bits (good(), fail(), etc.) in a stream are provided by basic_ios.

ios_base Mode openmode Member Constants (§iso.27.5.3.1.4)	
app	Append (insert output at end-of-stream)
ate	At end (position to end-of-stream)
binary	Don't apply formatting to characters
in	Input stream
out	Output stream
trunc	Truncate stream before use (set the stream's size to zero)

The exact meaning of ios_base::binary is implementation-dependent. However, the usual meaning is that a character gets mapped to a byte. For example:

```
template<typename T>
char* as_bytes(T& i)
{
    return static_cast<char*>(&i); // treat that memory as bytes
}

void test()
{
    ifstream ifs("source",ios_base::binary);      // stream mode is binary
    ofstream ofs("target",ios_base::binary);      // stream mode is binary

    vector<int> v;

    for (int i; ifs.read(as_bytes(i),sizeof(i));)      // read bytes from binary file
        v.push_back(i);

    // ... do something with v ...

    for (auto i : v)                              // write bytes to binary file:
        ofs.write(as_bytes(i),sizeof(i));
}
```

Use binary I/O when dealing with objects that are "just bags of bits" and do not have an obvious and reasonable character string representation. Images and sound/video streams are examples.

The seekg() (§38.6.2) and seekp() (§38.6.2) operations require a direction:

ios_base Direction seekdir Member Constants (§iso.27.5.3.1.5)	
beg	Seek from beginning of current file
cur	Seek from current position
end	Seek backward from end of current file

Classes derived from basic_ios format output and extract objects based on the information stored in their basic_io.

The ios_base operations can be summarized:

basic_ios<C,Tr> (§iso.27.5.5)	
basic_ios ios {p};	Construct ios given the stream buffer pointed to by p
ios.~basic_ios()	Destroy ios: release all of ios's resources
bool b {ios};	Conversion to bool: b is initialized to !ios.fail(); explicit
b=!ios	b=ios.fail()
st=ios.rdstate()	st is the iostate of ios
ios.clear(st)	Set the iostate of ios to st
ios.clear()	Set the iostate of ios to good
ios.setstate(st)	Add st to ios's iostate
ios.good()	Is the state of ios good (is goodbit set)?
ios.eof()	Is the state of ios end-of-file?
ios.fail()	Is the state of ios fail?
ios.bad()	Is the state of ios bad?
st=ios.exceptions()	st is the exceptions bits of the iostate of ios
ios.exceptions(st)	Set the exceptions bits of ios's iostate to st
p=ios.tie()	p is a pointer to a tied stream or nullptr
p=ios.tie(os)	Tie output stream os to ios; p is a pointer to the previously tied stream or nullptr
p=ios.rdbuf()	p is a pointer to ios's stream buffer
p=ios.rdbuf(p2)	Set ios's stream buffer to the one pointed to by p2; p is a pointer to the previous stream buffer
ios3=ios.copyfmt(ios2)	Copy the parts of io2's state related to formatting to ios; call any ios2 callback of type copyfmt_event; copy the values pointed to by ios2.pword and ios2.iword; ios3 is the previous format state
c=ios.fill()	c is the fill character of ios
c2=ios.fill(c)	Set c to be the fill character of ios; c2 is the previous fill character
loc2=ios.imbue(loc)	Set ios's locale to loc; loc2 is the previous locale
c2=narrow(c,d)	c2 is a char value obtained by converting c of char_type, d is a default value: use_facet<ctype<char_type>>(getloc()).narrow(c,d))

basic_ios<C,Tr> (continued) (§iso.27.5.5)	
c2=widen(c)	c2 is a char_type value obtained by converting c of char type: use_facet<ctype<char_type>>(getloc()).widen(c))
ios.init(p)	Set ios to the default state and use the stream buffer pointed to by p; protected
ios.set_rdbuf(p)	Make ios use the stream buffer pointed to by p; protected
ios.move(ios2)	Copy and move operation; protected
ios.swap(ios2)	Exchange the states of ios and ios2; protected; noexcept

The conversion of an ios (including istreams and ostreams) to bool is essential to the usual idiom for reading many values:

```
for (X x; cin>>x;) {
    // ...
}
```

Here, the return value of cin>>x is a reference to cin's ios. This ios is implicitly converted to a bool representing the state of cin. Thus, we could equivalently have written:

```
for (X x; !(cin>>x).fail();) {
    // ...
}
```

The tie() is used to ensure that output from a tied stream appears before an input from the stream to which it is tied. For example, cout is tied to cin:

```
cout << "Please enter a number: ";
int num;
cin >> num;
```

This code does not explicitly call cout.flush(), so had cout not been tied to cin, the user would see the request for input.

ios_base Operations (§iso.27.5.3.5, §iso.27.5.3.6)	
i=xalloc()	i is the index of a new (iword,pword) pair; static
r=iob.iword(i)	r is a reference to the ith long
r=iob.pword(i)	r is a reference to the ith void*
iob.register_callback(fn,i)	Register callback fn to iword(i)

Sometimes, people want to add to the state of a stream. For example, one might want a stream to "know" whether a complex should be output in polar or Cartesian coordinates. Class ios_base provides a function xalloc() to allocate space for such simple state information. The value returned by xalloc() identifies a pair of locations that can be accessed by iword() and pword().

Sometimes, an implementer or a user needs to be notified about a change in a stream's state. The register_callback() function "registers" a function to be called when its "event" occurs. Thus, a call of imbue(), copyfmt(), or ~ios_base() will call a function "registered" for an imbue_event, copyfmt_event, or erase_event, respectively. When the state changes, registered functions are called with the argument i supplied by their register_callback().

The event and event_callback types are defined in ios_base:

```
enum event {
        erase_event,
        imbue_event,
        copyfmt_event
};
using event_callback = void (*)(event, ios_base&, int index);
```

38.4.5 Formatting

The format of stream I/O is controlled by a combination of object type, stream state (§38.4.4), format state (§38.4.5.1), locale information (Chapter 39), and explicit operations (e.g., manipulators; §38.4.5.2).

38.4.5.1 Formatting State

In **<ios>**, the standard library defines a set of formatting constants of an implementation-defined bitmask type **fmtflags** as members of class **ios_base**:

ios_base Formatting fmtflags Constants (§iso.27.5.3.1.2)	
boolalpha	Use symbolic representation of **true** and **false**
dec	Integer base is 10
hex	Integer base is 16
oct	Integer base is 8
fixed	Floating-point format dddd.dd
scientific	Scientific format d.ddd**E**dd
internal	Pad between a prefix (such as **+**) and the number
left	Pad after the value
right	Pad before the value
showbase	On output, prefix octal numbers by **0** and hexadecimal numbers by **0x**
showpoint	Always show the decimal point (e.g., **123.**)
showpos	Show **+** for positive numbers (e.g., **+123**)
skipws	Skip whitespace on input
unitbuf	Flush after each output operation
uppercase	Use uppercase in numeric output, e.g., **1.2E10** and **0X1A2**
adjustfield	Set a value's placement in its field: **left**, **right**, or **internal**
basefield	Set the integer's base: **dec**, **oct**, or **hex**
floatfield	Set the floating-point format: **scientific** or **fixed**

Curiously, there are no **defaultfloat** or **hexfloat** flags. To get the equivalent, use manipulators **defaultfloat** and **hexfloat** (§38.4.5.2), or manipulate the **ios_base** directly:

```
ios.unsetf(ios_base::floatfield);                    // use the default floating-point format
ios.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield);    // use hexadecimal floats
```

An **iostream**'s format state can be read and written (set) by operations provided in its **ios_base**:

ios_base Formatting fmtflags Operations (§iso.27.5.3.2)	
f=ios.flags()	f is ios's formatting flags
f2=ios.flags(f)	Set ios's formatting flags to f; f2 is the old value of the flags
f2=ios.setf(f)	Set ios's formatting flags to f; f2 is the old value of the flags
f2=ios.setf(f,m)	f2=ios.setf(f&m)
ios.unsetf(f)	Clear the flags f in ios
n=ios.precision()	n is ios's precision
n2=ios.precision(n)	Set ios's precision to n; n2 is the old precision
n=ios.width()	n is ios's width
n2=ios.width(n)	Set ios's width to n; n2 is the old width

Precision is an integer that determines the number of digits used to display a floating-point number:

- The *general* format (**defaultfloat**) lets the implementation choose a format that presents a value in the style that best preserves the value in the space available. The precision specifies the maximum number of digits.
- The *scientific* format (**scientific**) presents a value with one digit before a decimal point and an exponent. The precision specifies the maximum number of digits after the decimal point.
- The *fixed* format (**fixed**) presents a value as an integer part followed by a decimal point and a fractional part. The precision specifies the maximum number of digits after the decimal point. For example, see §38.4.5.2.

Floating-point values are rounded rather than just truncated, and **precision()** doesn't affect integer output. For example:

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

This produces:

```
1234.5679 1234.5679 123456
1235 1235 123456
```

The **width()** function specifies the minimum number of characters to be used for the next standard-library **<<** output operation of a numeric value, **bool**, C-style string, character, pointer, **string**, and **bitset** (§34.2.2). For example:

```
cout.width(4);
cout << 12;     // print 12 preceded by two spaces
```

The "padding" or "filler" character can be specified by the **fill()** function. For example:

```
cout.width(4);
cout.fill('#');
cout << "ab";        // print ##ab
```

The default fill character is the space character, and the default field size is **0**, meaning "as many characters as needed." The field size can be reset to its default value like this:

```
cout.width(0);  // "as many characters as needed"
```

A call **width(n)** sets the minimum number of characters to **n**. If more characters are provided, they will all be printed. For example:

```
cout.width(4);
cout << "abcdef";   // print abcdef
```

It does not truncate the output to **abcd**. It is usually better to get the right output looking ugly than to get the wrong output looking just fine.

A **width(n)** call affects only the immediately following **<<** output operation. For example:

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

This produces **##12:13**, rather than **##12###:##13**.

If the explicit control of formatting options through many separate operations becomes tedious, we can combine them using a user-defined manipulator (§38.4.5.3).

An **ios_base** also allows the programmer to set an **iostream**'s **locale** (Chapter 39):

ios_base locale Operations (§iso.27.5.3.3)	
loc2=ios.imbue(loc)	Set **ios**'s locale to **loc**; **loc2** is the old value of the locale
loc=ios.getloc()	**loc** is **ios**'s locale

38.4.5.2 Standard Manipulators

The standard library provides manipulators corresponding to the various format states and state changes. The standard manipulators are defined in **<ios>**, **<istream>**, **<ostream>**, and **<iomanip>** (for manipulators that take arguments):

I/O Manipulators from <ios> (§iso.27.5.6, §iso.27.7.4) (continues)	
s<<boolalpha	Use symbolic representation of **true** and **false** (input and output)
s<<noboolalpha	**s.unsetf(ios_base::boolalpha)**
s<<showbase	On output prefix octal numbers by **0** and hexadecimal numbers by **0x**
s<<noshowbase	**s.unsetf(ios_base::showbase)**
s<<showpoint	Always show decimal point
s<<noshowpoint	**s.unsetf(ios_base::showpoint)**
s<<showpos	Show **+** for positive numbers
s<<noshowpos	**s.unsetf(ios_base::showpos)**
s<<uppercase	Use uppercase in numeric output, e.g., **1.2E10** and **0X1A2**
s<<nouppercase	Use lowercase in numeric output, e.g., **1.2e10** and **0x1a2**
s<<unitbuf	Flush after each output operation
s<<nounitbuf	Do not flush after each output operation

I/O Manipulators from <ios> (continued) (§iso.27.5.6, §iso.27.7.4)	
s<<internal	Pad where marked in formatting pattern
s<<left	Pad after value
s<<right	Pad before value
s<<dec	Integer base is 10
s<<hex	Integer base is 16
s<<oct	Integer base is 8
s<<fixed	Floating-point format dddd.dd
s<<scientific	Scientific format d.ddddEdd
s<<hexfloat	Use base 16 for mantissa and exponent, using p to start an exponent, e.g., A.1BEp–C and a.bcdef
s<<defaultfloat	Use the default floating point format
s>>skipws	Skip whitespace
s>>noskipws	s.unsetf(ios_base::skipws)

Each of these operations returns a reference to its first (stream) operand, s. For example:

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n';          // print 1234,4d2,2322
```

We can explicitly set the output format for floating-point numbers:

```
constexpr double d = 123.456;

cout << d << "; "
     << scientific <<  d << "; "
     << hexfloat <<  d << "; "
     << fixed << d << "; "
     << defaultfloat << d << '\n';
```

This produces:

```
123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456
```

The floating-point format is "sticky"; that is, it persists for subsequent floating-point operations.

I/O Manipulators from <ostream> (§iso.27.5.6, §iso.27.7.4)	
os<<endl	Put '\n' and flush
os<<ends	Put '\0'
os<<flush	Flush stream

An ostream is flushed when it is destroyed, when a tie()d istream needs input (§38.4.4), and when the implementation finds it advantageous. Explicitly flushing a stream is very rarely necessary. Similarly, <<endl can be considered equivalent to <<'\n', but the latter is probably a bit faster. I find

```
cout << "Hello, World!\n";
```

easier to read and write than

```
cout << "Hello, World!" << endl;
```

If you have a genuine need for frequent flushing, consider cerr and unitbuf.

I/O Manipulators from <iomanip> (§iso.27.5.6, §iso.27.7.4)	
s<<resetiosflags(f)	Clear flags f
s<<setiosflags(f)	Set flags f
s<<setbase(b)	Output integers in base b
s<<setfill(int c)	Make c the fill character
s<<setprecision(n)	Precision is n digits
s<<setw(n)	Next field width is n char
is>>get_money(m,intl)	Read from is using is's money_get facet; m is a long double or a basic_string; if intl==true, use standard three-letter currency names
is>>get_money(m)	s>>get_money(m,false)
os<<put_money(m,intl)	Write m to os using os's money_put facet; that money_put determines which types are acceptable for m; if intl==true, use standard three-letter currency names
os<<put_money(m)	s<<put_money(m,false)
is>>get_time(tmp,fmt)	Read into *tm according to the format fmt, using is's time_get facet
os<<put_time(tmp,fmt)	Write *tm to os according to the format fmt, using os's time_put facet

The time facets are found in §39.4.4 and the time formats in §43.6.

For example:

```
cout << '(' << setw(4) << setfill('#') << 12 << ") (" << 12 << ")\n";          // print (##12) (12)
```

istream Manipulators (§iso.27.5.6, §iso.27.7.4)	
s>>skipws	Skip whitespace (in <ios>)
s>>noskipws	s.unsetf(ios_base::skipws) (in <ios>)
is>>ws	Eat whitespace (in <istream>)

By default >> skips whitespace (§38.4.1). This default can be modified by >>skipws and >>noskipws. For example:

```
string input {"0 1 2 3 4"};
istringstream iss {input};
string s;
for (char ch; iss>>ch;)
        s += ch;
cout << s;                  // print "01234"

istringstream iss2 {input};
iss>>noskipws;
for (char ch; iss2>>ch;)
        s += ch;
cout << s;                  // print "0 1 2 3 4"
}
```

If you want to explicitly deal with whitespace (e.g., to make a newline significant) and still use **>>**, **noskipws** and **>>ws** become a convenience.

38.4.5.3 User-Defined Manipulators

A programmer can add manipulators in the style of the standard ones. Here, I present an additional style that I have found useful for formatting floating-point numbers.

Formatting is controlled by a confusing multitude of separate functions (§38.4.5.1). For example, a **precision()** persists for all output operations, but a **width()** applies to the next numeric output operation only. What I want is something that makes it simple to output a floating-point number in a predefined format without affecting future output operations on the stream. The basic idea is to define a class that represents formats, another that represents a format plus a value to be formatted, and then an operator **<<** that outputs the value to an **ostream** according to the format. For example:

```
Form gen4 {4};      // general format, precision 4

void f(double d)
{
    Form sci8;
    sci8.scientific().precision(8);      // scientific format, precision 8
    cout << d << ' ' << gen4(d) << ' ' << sci8(d) << ' ' << d << '\n';

    Form sci {10,ios_base::scientific}; // scientific format, precision 10
    cout << d << ' ' << gen4(d) << ' ' << sci(d) << ' ' << d << '\n';
}
```

A call **f(1234.56789)** writes:

```
1234.57 1235 1.23456789e+003 1234.57
1234.57 1235 1.2345678900e+003 1234.57
```

Note how the use of a **Form** doesn't affect the state of the stream, so that the last output of **d** has the same default format as the first.

Here is a simplified implementation:

```
class Form;          // our formatting type

struct Bound_form {      // Form plus value
    const Form& f;
    double val;
};

class Form {
    friend ostream& operator<<(ostream&, const Bound_form&);

    int prc;    // precision
    int wdt;    // width 0 means "as wide as necessary"
    int fmt;    // general, scientific, or fixed (§38.4.5.1)
    // ...
```

```
public:
    explicit Form(int p =6, ios_base::fmtflags f =0, int w =0) : prc{p}, fmt{f}, wdt{w} {}

    Bound_form Form::operator()(double d) const      // make a Bound_form for *this and d
    {
        return Bound_form{*this,d};
    }

    Form& scientific() { fmt = ios_base::scientific; return *this; }
    Form& fixed() { fmt = ios_base::fixed; return *this; }
    Form& general() { fmt = 0; return *this; }

    Form& uppercase();
    Form& lowercase();
    Form& precision(int p) { prc = p; return *this; }

    Form& width(int w) { wdt = w; return *this; }        // applies to all types
    Form& fill(char);

    Form& plus(bool b = true);                           // explicit plus
    Form& trailing_zeros(bool b = true);                 // print trailing zeros
    // ...
};
```

The idea is that a `Form` holds all the information needed to format one data item. The default is chosen to be reasonable for many uses, and the various member functions can be used to reset individual aspects of formatting. The `()` operator is used to bind a value with the format to be used to output it. A `Bound_form` (that is, a `Form` plus a value) can then be output to a given stream by a suitable `<<` function:

```
ostream& operator<<(ostream& os, const Bound_form& bf)
{
    ostringstream s;                 // §38.2.2
    s.precision(bf.f.prc);
    s.setf(bf.f.fmt,ios_base::floatfield);
    s << bf.val;                 // compose string in s
    return os << s.str();        // output s to os
}
```

Writing a less simplistic implementation of `<<` is left as an exercise.

Note that these declarations make the combination of `<<` and `()` into a ternary operator; `cout<<sci4{d}` collects the `ostream`, the format, and the value into a single function before doing any real computation.

38.5 Stream Iterators

In `<iterator>`, the standard library provides iterators to allow input and output streams to be viewed as sequences [input-begin:end-of-input) and [output-begin:end-of-output):

```
template<typename T,
         typename C = char,
         typename Tr = char_traits<C>,
         typename Distance = ptrdiff_t>
class istream_iterator
    :public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
    using char_type = C;
    using traits_type = Tr;
    using istream_type = basic_istream<C,Tr>;
    // ...
};

template<typename T, typename C = char, typename Tr = char_traits<C>>
class ostream_iterator: public iterator<output_iterator_tag, void, void, void, void> {
    using char_type = C;
    using traits_type = Tr;
    using ostream_type = basic_ostream<C,Tr>;
    // ...
};
```

For example:

```
copy(istream_iterator<double>{cin}, istream_iterator<double,char>{},
    ostream_iterator<double>{cout,";\n"});
```

When an `ostream_iterator` is constructed with a second (`string`) argument, that string is output as a terminator after every element value. So, if you enter `1 2 3` to that call of `copy()`, the output is:

```
1;
2;
3;
```

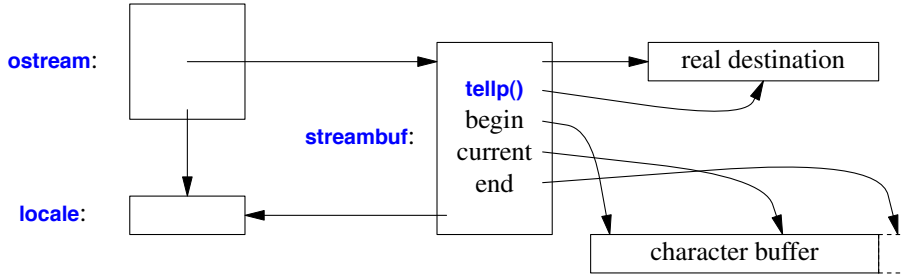
The operators provided for a `stream_iterator` are the same as for other iterator adaptors (§33.2.2):

Stream Iterator Operations (§iso.24.6)	
`istream_iterator p {st};`	Iterator for input stream `st`
`istream_iterator p {p2};`	Copy constructor: `p` is a copy of the `istream_iterator p2`
`ostream_iterator p {st};`	Iterator for output stream `st`
`ostream_iterator p {p2};`	Copy constructor: `p` is a copy of the `ostream_iterator p2`
`ostream_iterator p {st,s};`	Iterator for output stream `st`; use the C-style string `s` as the separator between output elements
`p=p2`	`p` is a copy of `p2`
`p2=++p`	`p` and `p2` point to the next element
`p2=p++`	`p2=p,++p`
`*p=x`	Insert `x` before `p`
`*p++=x`	Insert `x` before `p`, then increment `p`

Except for the constructors, these operations are typically used by general algorithms, such as `copy()`, rather than directly.

38.6 Buffering

Conceptually, an output stream puts characters into a buffer. Sometime later, the characters are then written to ("flushed to") wherever they are supposed to go. Such a buffer is called a **streambuf**. Its definition is found in **<streambuf>**. Different types of **streambuf**s implement different buffering strategies. Typically, the **streambuf** stores characters in an array until an overflow forces it to write the characters to their real destination. Thus, an **ostream** can be represented graphically like this:



The set of template arguments for an **ostream** and its **streambuf** must be the same, and they determine the type of character used in the character buffer.

An **istream** is similar, except that the characters flow the other way.

Unbuffered I/O is simply I/O where the **streambuf** immediately transfers each character, rather than holding on to characters until enough have been gathered for efficient transfer.

The key class in the buffering mechanisms is **basic_streambuf**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_streambuf {
public:
    using char_type = C;                    // the type of a character
    using int_type = typename Tr::int_type; // the integer type to which
                                            // a character can be converted
    using pos_type = typename Tr::pos_type; // type of position in buffer
    using off_type = typename Tr::off_type; // type of offset from position in buffer
    using traits_type = Tr;
    // ...
    virtual ~basic_streambuf();
};
```

As usual, a couple of aliases are provided for the (supposedly) most common cases:

```
using streambuf = basic_streambuf<char>;
using wstreambuf = basic_streambuf<wchar_t>;
```

The **basic_streambuf** has a host of operations. Many of the **public** operations simply call a **protected** virtual function that ensures that a function from a derived class implemented the operation appropriately for the particular kind of buffer:

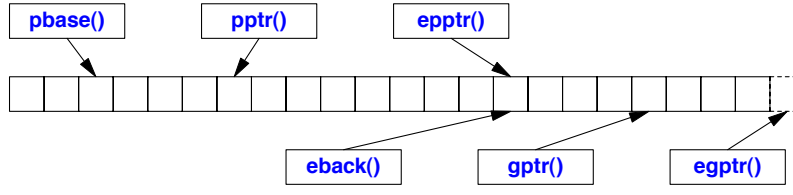
public basic_streambuf<C,Tr> Operations (§iso.27.6.3)	
sb.~basic_streambuf()	Destructor: release all resources; virtual
loc=sb.getloc()	loc is sb's locale
loc2=sb.pubimbue(loc)	sb.imbue(loc); loc2 is a pointer to the previous locale
psb=sb.pubsetbuf(s,n)	psb=sb.setbuf(s,n)
pos=sb.pubseekoff(n,w,m)	pos=sb.seekoff(n,w,m)
pos=sb.pubseekoff(n,w)	pos=sb.seekoff(n,w)
pos=sb.pubseekpos(n,m)	pos=sb.seekpos(n,m)
pos=sb.pubseekpos(n)	pos=sb.seekpos(n,ios_base::inlios_base::out)
sb.pubsync()	sb.sync()

All constructors are **protected** because **basic_streambuf** is designed as a base class.

protected basic_streambuf<C,Tr> Operations (§iso.27.6.3)	
basic_streambuf sb {};	Construct sb with no character buffer and the global locale
basic_streambuf sb {sb2};	sb is a copy of sb2 (they share a character buffer)
sb=sb2	sb is a copy of sb2 (they share a character buffer); sb's old resources are released
sb.swap(sb2)	Exchange the states of sb and sb2
sb.imbue(loc)	loc becomes sb's locale; virtual
psb=sb.setbuf(s,n)	Set sb's buffer; psb=&sb; s is a const char* and n is a streamsize; virtual
pos=sb.seekoff(n,w,m)	Seek with an offset n, a direction w, and a mode m; pos is the resulting position or pos_type(off_type(−1)), indicating an error; virtual
pos=sb.seekoff(n,w)	pos=sb.seekoff(n,way,ios_base::inlios_base::out)
pos=sb.seekpos(n,m)	Seek to position n and a mode m; pos is the resulting position or pos_type(off_type(−1)), indicating an error; virtual
n=sb.sync()	Synchronize the character buffers with the real destination or source; virtual

The exact meaning of the virtual functions are determined by derived classes.

A **streambuf** has a *put area* into which **<<** and other output operations write (§38.4.2), and a *get area* from which **>>** and other input operations read (§38.4.1). Each area is described by a beginning pointer, current pointer, and one-past-the-end pointer:



Overflow are handled by the virtual functions **overflow()**, **underflow()**, and **uflow()**.

For a use of positioning, see §38.6.1.

The put-and-get interface is separated into a **public** and a **protected** one:

public Put and Get basic_streambuf<C,Tr> Operations (§iso.27.6.3) (continues)	
n=sb.in_avail()	If a read position is available, **n=sb.egptr()−sb.gptr()**; otherwise returns **sb.showmanyc()**
c=sb.snextc()	Increase **sb**'s get pointer, then **c=∗sb.gptr()**
n=sb.sbumpc()	Increase **sb**'s get pointer
c=sb.sgetc()	If there is no character left to get, **c=sb.underflow()**; otherwise **c=∗sb.gptr()**
n=sb.sgetn(p,n)	**n=sb.xsgetn(p,n)**; **p** is a **char∗**
n=sb.sputbackc(c)	Put **c** back into the get area and decrease the **gptr**; **n=Tr::to_int_type(∗sb.gptr())** if the putback succeeded; otherwise **n=sb.pbackfail(Tr::to_int_type(c))**
n=sb.sungetc()	Decrease the get pointer; **n=Tr::to_int_type(∗sb.gptr())** if the unget succeeded; otherwise **n=sb.pbackfail(Tr::to_int_type())**
n=sb.sputc(c)	If there is no character left to put into, **n=sb.overflow(Tr::to_int_type(c))**; otherwise **∗sb.sptr()=c**; **n=Tr::to_int_type(c)**
n=sb.sputn(s,n)	**n=sb.xsputn(s,n)**; **s** is a **const char∗**

The **protected** interface provides simple, efficient, and typically inlined functions manipulating the put and get pointers. In addition, there are virtual functions to be overridden by derived classes.

protected Put and Get basic_streambuf<C,Tr> Operations (continued) (§iso.27.6.3)	
sb.setg(b,n,end)	The get area is [**b**,**e**); the current get pointer is **n**
pc=sb.eback()	[**pc**:**sb.egptr()**) is the get area
pc=sb.gptr()	**pc** is the get pointer
pc=sb.egptr()	[**sb.eback()**:**pc**) is the get area
sb.gbump(n)	Increase **sb**'s get pointer
n=sb.showmanyc()	"Show how many characters"; **n** is an estimate of how many characters can be read without calling **sb.underflow()** or **n=−1** indicating that no characters are ready to be read; virtual
n=sb.underflow()	No more characters in the get area; replenish the get area; **n=Tr::to_int_type(c)** where **c** is the new current get character; virtual
n=sb.uflow()	Like **sb.underflow()**, but advance the get pointer after reading the new current get character; virtual
n=sb.pbackfail(c)	A putback operation failed; **n=Tr::eof()** if an overriding **pbackfail()** could not put back; virtual
n=sb.pbackfail()	**n=sb.pbackfail(Tr::eof())**

protected Put and Get basic_streambuf<C,Tr> Operations (continued) (§iso.27.6.3)	
sb.setp(b,e)	The put area is [b,e) the current put pointer is b
pc=sb.pbase()	[pc:sb.epptr()) is the put area
pc=sb.pptr()	pc is the put pointer
pc=sb.epptr()	[sb.pbase(),pc) is the put area
sb.pbump(n)	Add one to the put pointer
n2=sb.xsgetn(s,n)	s is a const char*; do sb.sgetc(*p) for each p in [s:s+n); n2 is the number of characters read; virtual
n2=sb.xsputn(s,n)	s is a [const char*; sb.sputc(*p) for each p in [s:s+n); n2 is the number of character written; virtual
n=sb.overflow(c)	Replenish the put area, then n=sb.sputc(c); virtual
n=sb.overflow()	n=sb.overflow(Tr::eof())

The showmanyc() (“show how many characters”) function is an odd function intended to allow a user to learn something about the state of a machine’s input system. It returns an estimate of how many characters can be read “soon,” say, by emptying the operating system’s buffers rather than waiting for a disk read. A call to showmanyc() returns –1 if it cannot promise that any character can be read without encountering end-of-file. This is (necessarily) rather low-level and highly implementation-dependent. Don’t use showmanyc() without a careful reading of your system documentation and conducting a few experiments.

38.6.1 Output Streams and Buffers

An ostream provides operations for converting values of various types into character sequences according to conventions (§38.4.2) and explicit formatting directives (§38.4.5). In addition, an ostream provides operations that deal directly with its streambuf:

```
template<typename C, typename Tr = char_traits<C>>
class basic_ostream : virtual public basic_ios<C,Tr> {
public:
    // ...
    explicit basic_ostream(basic_streambuf<C,Tr>* b);

    pos_type tellp();                                   // get current position
    basic_ostream& seekp(pos_type);                     // set current position
    basic_ostream& seekp(off_type, ios_base::seekdir);  // set current position

    basic_ostream& flush();                             // empty buffer (to real destination)

    basic_ostream& operator<<(basic_streambuf<C,Tr>* b);    // write from b
};
```

The basic_ostream functions override their equivalents in the basic_ostream’s basic_ios base.

An ostream is constructed with a streambuf argument, which determines how the characters written are handled and where they eventually go. For example, an ostringstream (§38.2.2) or an ofstream (§38.2.1) is created by initializing an ostream with a suitable streambuf (§38.6).

The **seekp()** functions are used to position an **ostream** for writing. The **p** suffix indicates that it is the position used for *putting* characters into the stream. These functions have no effect unless the stream is attached to something for which positioning is meaningful, such as a file. The **pos_type** represents a character position in a file, and the **off_type** represents an offset from a point indicated by an **ios_base::seekdir**.

Stream positions start at **0**, so we can think of a file as an array of **n** characters. For example:

```
int f(ofstream& fout)// fout refers to some file
{
        fout << "0123456789";
        fout.seekp(8);                  // 8 from beginning
        fout << '#';                    // add '#' and move position (+1)
        fout.seekp(–4,ios_base::cur);   // 4 backward
        fout << '*';                    // add '*' and move position (+1)
}
```

If the file was initially empty, we get:

```
01234*67#9
```

There is no similar way to do random access on elements of a plain **istream** or **ostream**. Attempting to seek beyond the beginning or the end of a file typically puts the stream into the **bad()** state (§38.4.4). However, some operating systems have operating modes where the behavior differs (e.g., positioning might resize a file).

The **flush()** operation allows the user to empty the buffer without waiting for an overflow.

It is possible to use **<<** to write a **streambuf** directly into an **ostream**. This is primarily handy for implementers of I/O mechanisms.

38.6.2 Input Streams and Buffers

An **istream** provides operations for reading characters and converting them into values of various types (§38.4.1). In addition, an **istream** provides operations that deal directly with its **streambuf**:

```
template<typename C, typename Tr = char_traits<C>>
class basic_istream : virtual public basic_ios<C,Tr> {
public:
        // ...
        explicit basic_istream(basic_streambuf<C,Tr>* b);
        pos_type tellg();                                   // get current position
        basic_istream& seekg(pos_type);                     // set current position
        basic_istream& seekg(off_type, ios_base::seekdir);  // set current position

        basic_istream& putback(C c);    // put c back into the buffer
        basic_istream& unget();         // put back most recent char read
        int_type peek();                // look at next character to be read

        int sync();                     // clear buffer (flush)

        basic_istream& operator>>(basic_streambuf<C,Tr>* b);    // read into b
        basic_istream& get(basic_streambuf<C,Tr>& b, C t = Tr::newline());
```

```
        streamsize readsome(C* p, streamsize n);     // read at most n char
};
```

The basic_istream functions override their equivalents in the basic_istream's basic_ios base.

The positioning functions work like their ostream counterparts (§38.6.1). The g suffix indicates that it is the position used for *getting* characters from the stream. The p and g suffixes are needed because we can create an iostream derived from both istream and ostream, and such a stream needs to keep track of both a get position and a put position.

The putback() function allows a program to put a character "back" into an istream to be the next character read. The unget() function puts the most recently read character back. Unfortunately, backing up an input stream is not always possible. For example, trying to back up past the first character read will set ios_base::failbit. What is guaranteed is that you can back up one character after a successful read. The peek() function reads the next character and also leaves that character in the streambuf so that it can be read again. Thus, c=peek() is logically equivalent to (c=get(),unget(),c). Setting failbit might trigger an exception (§38.3).

Flushing an istream is done using sync(). This cannot always be done right. For some kinds of streams, we would have to reread characters from the real source – and that is not always possible or desirable (e.g., for a stream attached to a network). Consequently, sync() returns 0 if it succeeded. If it failed, it sets ios_base::badbit (§38.4.4) and returns –1. Setting badbit might trigger an exception (§38.3). A sync() on a buffer attached to an ostream flushes the buffer to output.

The >> and get() operations that directly reads from a streambuf are primarily useful for implementers of I/O facilities.

The readsome() function is a low-level operation that allows a user to peek at a stream to see if there are any characters available to read. This can be most useful when it is undesirable to wait for input, say, from a keyboard. See also in_avail() (§38.6).

38.6.3 Buffer Iterators

In <iterator>, the standard library provides istreambuf_iterator and ostreambuf_iterator to allow a user (mostly an implementer of a new kind of iostream) to iterate over the contents of a stream buffer. In particular, these iterators are widely used by locale facets (Chapter 39).

38.6.3.1 istreambuf_iterator

An istreambuf_iterator reads a stream of characters from an istream_buffer:

```
template<typename C, typename Tr = char_traits<C>>   // §iso.24.6.3
class istreambuf_iterator
        :public iterator<input_iterator_tag, C, typename Tr::off_type, /*unspecified*/, C> {
public:
        using char_type = C;
        using traits_type = Tr;
        using int_type = typename Tr::int_type;
        using streambuf_type = basic_streambuf<C,Tr>;
        using istream_type = basic_istream<C,Tr>;
        // ...
};
```

The reference member of the iterator base is not used and is consequently left unspecified.

If you use an istreambuf_iterator as an input iterator, its effect is like that of other input iterators: a stream of characters can be read from input using c=*p++:

istreambuf_iterator<C,Tr> (§iso.24.6.3)	
istreambuf_iterator p {};	p is an end-of-stream iterator; noexcept; constexpr
istreambuf_iterator p {p2};	Copy constructor; noexcept
istreambuf_iterator p {is};	p is an iterator for is.rdbuf(); noexcept
istreambuf_iterator p {psb};	p is an iterator to the istreambuf *psb; noexcept
istreambuf_iterator p {nullptr};	p is an end-of-stream iterator
istreambuf_iterator p {prox};	p points to the istreambuf designated by prox; noexcept
p.~istreambuf_iterator()	Destructor
c=*p	c is the character returned by the streambuf's sgetc()
p->m	The member m of *p, if that is a class object
p=++p	The streambuf's sbumpc()
prox=p++	Let prox designate the same position as p; then ++p
p.equal(p2)	Are both p and p2, or neither, at end-of-stream
p==p2	p.equal(p2)
p!=p2	!p.equal(p2)

Note that any attempt to be clever when comparing istreambuf_iterators will fail: you cannot rely on two iterators referring to the same character while input is going on.

38.6.3.2 ostreambuf_iterator

An ostreambuf_iterator writes a stream of characters to an ostream_buffer:

```
template<typename C, typename Tr = char_traits<C>>  // §iso.24.6.4
class ostreambuf_iterator
        :public iterator<output_iterator_tag, void, void, void, void> {
public:
        using char_type = C;
        using traits_type = Tr;
        using streambuf_type = basic_streambuf<C,Tr>;
        using ostream_type = basic_ostream<C,Tr>;
        // ...
};
```

By most measures, ostreambuf_iterator's operations are odd, but the net effect is that if you use it as an output iterator, its effect is like that of other output iterators: a stream of characters can be written to output using *p++=c:

ostreambuf_iterator<C,Tr> (§iso.24.6.4) (continues)	
ostreambuf_iterator p {os};	p is an iterator for os.rdbuf(); noexcept
ostreambuf_iterator p {psb};	p is an iterator for the istreambuf *psb; noexcept

ostreambuf_iterator<C,Tr> (continued) (§iso.24.6.4)	
p=c	If !p.failed() call the streambuf's sputc(c)
*p	Do nothing
++p	Do nothing
p++	Do nothing
p.failed()	Has a sputc() on p's streambuf reached eof? noexcept

38.7 Advice

[1] Define << and >> for user-defined types with values that have meaningful textual representations; §38.1, §38.4.1, §38.4.2.
[2] Use cout for normal output and cerr for errors; §38.1.
[3] There are iostreams for ordinary characters and wide characters, and you can define an iostream for any kind of character; §38.1.
[4] There are standard iostreams for standard I/O streams, files, and strings; §38.2.
[5] Don't try to copy a file stream; §38.2.1.
[6] Binary I/O is system specific; §38.2.1.
[7] Remember to check that a file stream is attached to a file before using it; §38.2.1.
[8] Prefer ifstreams and ofstreams over the generic fstream; §38.2.1.
[9] Use stringstreams for in-memory formatting; §38.2.2.
[10] Use exceptions to catch rare bad() I/O errors; §38.3.
[11] Use the stream state fail to handle potentially recoverable I/O errors; §38.3.
[12] You don't need to modify istream or ostream to add new << and >> operators; §38.4.1.
[13] When implementing a iostream primitive operation, use sentry; §38.4.1.
[14] Prefer formatted input over unformatted, low-level input; §38.4.1.
[15] Input into strings does not overflow; §38.4.1.
[16] Be careful with the termination criteria when using get(), getline(), and read(); §38.4.1.
[17] By default >> skips whitespace; §38.4.1.
[18] You can define a << (or a >>) so that it behaves as a virtual function based on its second operand; §38.4.2.1.
[19] Prefer manipulators to state flags for controlling I/O; §38.4.3.
[20] Use sync_with_stdio(true) if you want to mix C-style and iostream I/O; §38.4.4.
[21] Use sync_with_stdio(false) to optimize iostreams; §38.4.4.
[22] Tie streams used for interactive I/O; §38.4.4.
[23] Use imbue() to make an iostream reflect "cultural differences" of a locale; §38.4.4.
[24] width() specifications apply to the immediately following I/O operation only; §38.4.5.1.
[25] precision() specifications apply to all following floating-point output operations; §38.4.5.1.
[26] Floating-point format specifications (e.g., scientific) apply to all following floating-point output operations; §38.4.5.2.
[27] #include <iomanip> when using standard manipulators taking arguments; §38.4.5.2.
[28] You hardly ever need to flush(); §38.4.5.2.

[29] Don't use **endl** except possibly for aesthetic reasons; §38.4.5.2.
[30] If **iostream** formatting gets too tedious, write your own manipulators; §38.4.5.3.
[31] You can achieve the effect (and efficiency) of a ternary operator by defining a simple function object; §38.4.5.3.