21

Streams

What you see is all you get.
– Brian Kernighan

Input and output — ***ostream***s — output of built-in types — output of user-defined types — virtual output functions — ***istream***s — input of built-in types — unformatted input — stream state — input of user-defined types — I/O exceptions — tying of streams — sentries — formatting integer and floating-point output — fields and adjustments — manipulators — standard manipulators — user-defined manipulators — file streams — closing streams — string streams — stream buffers — locale — stream callbacks — ***printf***( ) — advice — exercises.

21.1 Introduction [io.intro]

Designing and implementing a general input/output facility for a programming language is notoriously difficult. Traditionally, I/O facilities have been designed exclusively to handle a few built-in data types. However, a nontrivial C++ program uses many user-defined types, and the input and output of values of those types must be handled. An I/O facility should be easy, convenient, and safe to use; efficient and flexible; and, above all, complete. Nobody has come up with a solution that pleases everyone. It should therefore be possible for a user to provide alternative I/O facilities and to extend the standard I/O facilities to cope with special applications.

C++ was designed to enable a user to define new types that are as efficient and convenient to use as built-in types. It is therefore a reasonable requirement that an I/O facility for C++ should be provided in C++ using only facilities available to every programmer. The stream I/O facilities presented here are the result of an effort to meet this challenge:

§21.2 *Output:* What the application programmer thinks of as output is really the conversion of objects of types, such as ***int***, ***char****, and ***Employee_record***, into sequences of characters. The facilities for writing built-in and user-defined types to output are described.

§21.3 *Input:* The facilities for requesting input of characters, strings, and values of other built-in and user-defined types are presented.

§21.4 *Formatting:* There are often specific requirements for the layout of the output. For example, ***int***s may have to be printed in decimal and pointers in hexadecimal or floating-point numbers must appear with exactly specified precision. Formatting controls and the programming techniques used to provide them are discussed.

§21.5 *Files and Streams:* By default, every C++ program can use standard streams, such as standard output (***cout***), standard input (***cin***), and error output (***cerr***). To use other devices or files, streams must be created and attached to those files or devices. The mechanisms for opening and closing files and for attaching streams to files and ***string***s are described.

§21.6 *Buffering:* To make I/O efficient, we must use a buffering strategy that is suitable for both the data written (read) and the destination it is written to (read from). The basic techniques for buffering streams are presented.

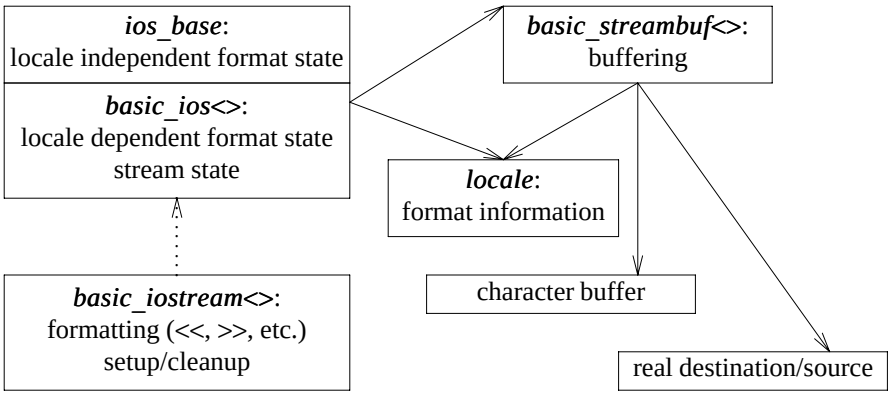
§21.7 *Locale:* A ***locale*** is an object that specifies how numbers are printed, what characters are considered letters, etc. It encapsulates many cultural differences. Locales are implicitly used by the I/O system and are only briefly described here.

§21.8 *C I/O:* The ***printf( )*** function from the C ***<stdio.h>*** library and the C library's relation to the C++ ***<iostream>*** library are discussed.

Knowledge of the techniques used to implement the stream library is not needed to use the library. Also, the techniques used for different implementations will differ. However, implementing I/O is a challenging task. An implementation contains examples of techniques that can be applied to many other programming and design tasks. Therefore, the techniques used to implement I/O are worthy of study.

This chapter discusses the stream I/O system to the point where you should be able to appreciate its structure, to use it for most common kinds of I/O, and to extend it to handle new user-defined types. 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/or examples of working code in addition to what is presented here.

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



The dotted arrow from *basic_iostream*<> indicates that *basic_ios*<> is a virtual base class; the solid arrows represent pointers. The classes marked with <> are templates parameterized by a character type and containing a *locale*.

The streams concept and the general notation it provides can be applied to a large class of communication problems. Streams have been used for transmitting objects between machines (§25.4.1), for encrypting message streams (§21.10[22]), for data compression, for persistent storage of objects, and much more. However, the discussion here is restricted to simple character-oriented input and output.

Declarations of stream I/O classes and templates (sufficient to refer to them but not to apply operations to them) and standard *typedef*s are presented in <*iosfwd*>. This header is occasionally needed when you want to include some but not all of the I/O headers.

21.2 Output [io.out]

Type-safe and uniform treatment of both built-in and user-defined types can be achieved by using a single overloaded function name for a set of output functions. For example:

```
put(cerr,"x = "); // cerr is the error output stream
put(cerr,x);
put(cerr,'\n');
```

The type of the argument determines which *put* function will be invoked for each argument. This solution is used in several languages. However, it is repetitive. Overloading the operator << to mean ''put to'' gives a better notation and lets the programmer output a sequence of objects in a single statement. For example:

```
cerr << "x = " << x << '\n';
```

If *x* is an *int* with the value *123*, this statement would print

```
x = 123
```

followed by a newline onto the standard error output stream, *cerr*. Similarly, if *x* is of type *complex* (§22.5) with the value *(1,2.4)*, the statement will print

```
x = (1,2.4)
```

on *cerr*. This style can be used as long as *x* is of a type for which operator << is defined and a user can trivially define operator << for a new type.

An output operator is needed to avoid the verbosity that would have resulted from using an output function. But why <<? It is not possible to invent a new lexical token (§11.2). The assignment operator was a candidate for both input and output, but most people seemed to prefer to use different operators for input and output. Furthermore, = binds the wrong way; that is, *cout=a=b* means *cout=(a=b)* rather than *(cout=a)=b* (§6.2). I tried the operators < and >, but the meanings ''less than'' and ''greater than'' were so firmly implanted in people's minds that the new I/O statements were for all practical purposes unreadable.

The operators << and >> are not used frequently enough for built-in types to cause that problem. They are symmetric in a way that can be used to suggest ''to'' and ''from.'' When they are

used for I/O, I refer to << as *put to* and to >> as *get from*. People who prefer more technical-sounding names call them *inserters* and *extractors*, respectively. The precedence of << is low enough to allow arithmetic expressions as operands without using parentheses. For example:

```
cout << "a*b+c=" << a*b+c << '\n';
```

Parentheses must be used to write expressions containing operators with precedence lower than <<'s. For example:

```
cout << "a^b|c=" << (a^b|c) << '\n';
```

The left shift operator (§6.2.4) can be used in an output statement, but of course it, too, must appear within parentheses:

```
cout << "a<<b=" << (a<<b) << '\n';
```

21.2.1 Output Streams [io.ostream]

An ***ostream*** is a mechanism for converting values of various types into sequences of characters. Usually, these characters are then output using lower-level output operations. There are many kinds of characters (§20.2) that can be characterized by ***char_traits*** (§20.2.1). Consequently, an ***ostream*** is a specialization for a particular kind of character of a general ***basic_ostream*** template:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
        virtual ~basic_ostream();
        // ...
};
```

This template and its associated output operations are defined in namespace ***std*** and presented by ***<ostream>***, which contains the output-related parts of ***<iostream>***.

The ***basic_ostream*** template parameters control the type of characters that is used by the implementation; they do not affect the types of values that can be output. Streams implemented using ordinary ***char***s and streams implemented using wide characters are directly supported by every implementation:

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
```

On many systems, it is possible to optimize writing of wide characters through ***wostream*** to an extent that is hard to match for streams using bytes as the unit of output.

It is possible to define streams for which the physical I/O is not done in terms of characters. However, such streams are beyond the scope of the C++ standard and beyond the scope of this book (§21.10[15]).

The ***basic_ios*** base class is presented in ***<ios>***. It controls formatting (§21.4), locale (§21.7), and access to buffers (§21.6). It also defines a few types for notational convenience:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type;    // type of integer value of character
    typedef typename Tr::pos_type pos_type;    // position in buffer
    typedef typename Tr::off_type off_type;    // offset in buffer

    // ... see also §21.3.3, §21.3.7, §21.4.4, §21.6.3, and §21.7.1 ...
};
```

The ***ios_base*** base class contains information and operations that are independent of the character type used, such as the precision used for floating-point output. It therefore doesn't need to be a template.

In addition to the ***typedef***s in ***ios_base***, the stream I/O library uses a signed integral type ***streamsize*** to represent the number of characters transferred in an I/O operation and the size of I/O buffers. Similarly, a ***typedef*** called ***streamoff*** is supplied for expressing offsets in streams and buffers.

Several standard streams are declared in **<*iostream*>**:

```
ostream cout;       // standard output stream of char
ostream cerr;       // standard unbuffered output stream for error messages
ostream clog;       // standard output stream for error messages

wostream wcout;     // wide stream corresponding to cout
wostream wcerr;     // wide stream corresponding to cerr
wostream wclog;     // wide stream corresponding to clog
```

The ***cerr*** and ***clog*** streams refer to the same output destination; they simply differ in the buffering they provide. The ***cout*** writes to the same destination as C's ***stdout*** (§21.8), while ***cerr*** and ***clog*** write to the same destination as C's ***stderr***. The programmer can create more streams as needed (see §21.5).

21.2.2 Output of Built-In Types [io.out.builtin]

The class ***ostream*** is defined with the operator **<<** (''put to'') to handle output of the built-in types:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    // ...

    basic_ostream& operator<<(short n);
    basic_ostream& operator<<(int n);
    basic_ostream& operator<<(long n);

    basic_ostream& operator<<(unsigned short n);
    basic_ostream& operator<<(unsigned int n);
    basic_ostream& operator<<(unsigned long n);
```

```
        basic_ostream& operator<<(float f);
        basic_ostream& operator<<(double f);
        basic_ostream& operator<<(long double f);

        basic_ostream& operator<<(bool n);
        basic_ostream& operator<<(const void* p);          // write pointer value

        basic_ostream& put(Ch c);       // write c
        basic_ostream& write(const Ch* p, streamsize n);   // p[0]..p[n-1]

        // ...
};
```

An *operator<<()* returns a reference to the *ostream* for which it was called so that another *operator<<()* can be applied to it. For example,

```
cerr << "x = " << x;
```

where *x* is an *int*, will be interpreted as:

```
(cerr.operator<<("x = ")).operator<<(x);
```

In particular, this implies that when several items are printed by a single output statement, they will be printed in the expected order: left to right. For example:

```
void val(char c)
{
        cout << "int('" << c << "') = " << int(c) << '\n';
}

int main()
{
        val('A');
        val('Z');
}
```

On an implementation using ASCII characters, this will print:

```
int('A') = 65
int('Z') = 90
```

Note that a character literal has type *char* (§4.3.1) so that *cout<<'Z'* will print the letter *Z* and not the integer value *90*.

A *bool* value will be output as *0* or *1* by default. If you don't like that, you can set the formatting flag *boolalpha* from *<iomanip>* (§21.4.6.2) and get *true* or *false*. For example:

```
int main()
{
        cout << true << ' ' << false << '\n';
        cout << boolalpha;                      // use symbolic representation for true and false
        cout << true << ' ' << false << '\n';
}
```

This prints:

```
1 0
true false
```

More precisely, ***boolalpha*** ensures that we get a locale-dependent representation of ***bool*** values. By setting my locale (§21.7) just right, I can get:

```
1 0
sandt falsk
```

Formatting floating-point numbers, the base used for integers, etc., are discussed in §21.4.

The function ***ostream*::*operator*<<(*const void**)** prints a pointer value in a form appropriate to the architecture of the machine used. For example,

```
int main()
{
    int i = 0;
    int* p = new int;
    cout << "local " << &i << ", free store " << p << '\n';
}
```

printed

```
local 0x7fffead0, free store 0x500c
```

on my machine. Other systems have different conventions for printing pointer values.

The ***put*()** and ***write*()** functions simply write characters. Consequently, the << for outputting characters need not be a member. The ***operator*<<()** functions that take a character operand can be implemented as nonmembers using ***put*()**:

```
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, Ch);
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, char);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, char);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, signed char);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, unsigned char);
```

Similarly, << is provided for writing out zero-terminated character arrays:

```
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const Ch*);
template<class Ch, class Tr>
    basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const char*);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const char*);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const signed char*);
template<class Tr>
    basic_ostream<char,Tr>& operator<<(basic_ostream<char,Tr>&, const unsigned char*);
```

21.2.3 Output of User-Defined Types [io.out.udt]

Consider a user-defined type *complex* (§11.3):

```
class complex {
public:
        double real() const { return re; }
        double imag() const { return im; }
        // ...
};
```

Operator << can be defined for the new type *complex* like this:

```
ostream& operator<<(ostream&s, complex z)
{
        return s << '(' << z.real() << ',' << z.imag() << ')';
}
```

This << can then be used exactly like << for a built-in type. For example,

```
int main()
{
        complex x(1,2);
        cout << "x = " << x << '\n';
}
```

produces

```
x = (1,2)
```

Defining an output operation for a user-defined type does not require modification of the declaration of class *ostream*. This is fortunate because *ostream* is defined in *<iostream>*, which users cannot and should not modify. Not allowing additions to *ostream* also provides protection against accidental corruption of that data structure and makes it possible to change the implementation of an *ostream* without affecting user programs.

21.2.3.1 Virtual Output Functions [io.virtual]

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 crucial and where inlining is a must. Virtual functions are used to achieve flexibility for the operations dealing with buffer overflow and underflow only (§21.6.4).

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 the 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;      // the real output function: override My_base::put()
};

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.

21.3 Input [io.in]

Input is handled similarly to output. There is a class ***istream*** that provides an input operator **>>** (''get from'') for a small set of standard types. An ***operator>>()*** can then be defined for a user-defined type.

21.3.1 Input Streams [io.istream]

In parallel to ***basic_ostream*** (§21.2.1), ***basic_istream*** is defined in **<*istream*>**, which contains the input-related parts of **<*iostream*>**, like this:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    virtual ~basic_istream();

    // ...
};
```

The base class ***basic_ios*** is described in §21.2.1.

Two standard input streams ***cin*** and ***wcin*** are provided in <***iostream***>:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

istream cin;      // standard input stream of char
wistream wcin;    // standard input stream of wchar_t
```

The ***cin*** stream reads from the same source as C's ***stdin*** (§21.8).

21.3.2 Input of Built-In Types [io.in.builtin]

An ***istream*** provides operator >> for the built-in types:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    // formatted input:

    basic_istream& operator>>(short& n);              // read into n
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(long& n);

    basic_istream& operator>>(unsigned short& u); // read into u
    basic_istream& operator>>(unsigned int& u);
    basic_istream& operator>>(unsigned long& u);

    basic_istream& operator>>(float& f);              // read into f
    basic_istream& operator>>(double& f);
    basic_istream& operator>>(long double& f);

    basic_istream& operator>>(bool& b);               // read into b
    basic_istream& operator>>(void*& p);              // read pointer value into p

    // ...
};
```

The ***operator>>()*** input functions are defined in this style:

```
istream& istream::operator>>(T& tvar)    // T is a type for which istream::operator>> is declared
{
    // skip whitespace, then somehow read a T into 'tvar'
    return *this;
}
```

Because >> skips whitespace, you can read a sequence of whitespace-separated integers like this:

```
int read_ints(vector<int>& v)    // fill v, return number of ints read
{
    int i = 0;
    while (i<v.size() && cin>>v[i]) i++;
    return i;
}
```

A non-***int*** on the input will cause the input operation to fail and thus terminate the input loop. For example, the input:

```
1 2 3 4 5.6 7 8.
```

will have ***read_ints***() read in the five integers

```
1 2 3 4 5
```

and leave the dot as the next character to be read from input. Whitespace is defined as the standard C whitespace (blank, tab, newline, formfeed, and carriage return) by a call to ***isspace***() as defined in <*cctype*> (§20.4.2).

The most common mistake when using ***istream***s is to fail to notice that input didn't happen as expected because the input wasn't of the expected format. One should either check the state of an input stream (§21.3.3) before relying on values supposedly read in or use exceptions (§21.3.6).

The format expected for input is specified by the current locale (§21.7). By default, the ***bool*** values ***true*** and ***false*** are represented by ***1*** and ***0***, respectively. Integers must be decimal and floating-point numbers of the form used to write them in a C++ program. By setting ***base_field*** (§21.4.2), it is possible to read ***0123*** as an octal number with the decimal value ***83*** and ***0xff*** as a hexadecimal number with the decimal value ***255***. The format used to read pointers is completely implementation-dependent (have a look to see what your implementation does).

Surprisingly, there is no member >> for reading a character. The reason is simply that >> for characters can be implemented using the ***get***() character input operations (§21.3.4), so it doesn't need to be a member. From a stream, we can read a character into the stream's character type. If that character type is ***char***, we can also read into a ***signed char*** and ***unsigned char***:

```
template<class Ch, class Tr>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, Ch&);

template<class Tr>
basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, unsigned char&);

template<class Tr>
basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, signed char&);
```

From a user's point of view, it does not matter whether a >> is a member.

Like the other >> operators, these functions first skip whitespace. For example:

```
void f()
{
    char c;
    cin >> c;
    // ...
}
```

This places the first non-whitespace character from ***cin*** into ***c***.

In addition, we can read into an array of characters:

```
template<class Ch, class Tr>
    basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, Ch*);
template<class Tr>
    basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, unsigned char*);
template<class Tr>
    basic_istream<char,Tr>& operator>>(basic_istream<char,Tr>&, signed char*);
```

These operations first skip whitespace. Then they read into their array operand until they encounter a whitespace character or end-of-file. Finally, they terminate the string with a `0`. Clearly, this offers ample opportunity for overflow, so reading into a `string` (§20.3.15) is usually better. However, you can specify a maximum for the number of characters to be read by `>>`: `is.width(n)` specifies that the next `>>` on `is` will read at most `n-1` characters into an array. For example:

```
void g()
{
    char v[4];
    cin.width(4);
    cin >> v;
    cout << "v = " << v << endl;
}
```

This will read at most three characters into `v` and add a terminating `0`.

Setting `width()` for an `istream` affects only the immediately following `>>` into an array and does not affect reading into other types of variables.

21.3.3 Stream State [io.state]

Every stream (`istream` or `ostream`) has a *state* associated with it. Errors and nonstandard conditions are handled by setting and testing this state appropriately.

The stream state is found in `basic_istream`'s base `basic_ios` from `<ios>`:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    bool good() const;    // next operation might succeed
    bool eof() const;     // end of input seen
    bool fail() const;    // next operation will fail
    bool bad() const;     // stream is corrupted

    iostate rdstate() const;                        // get io state flags
    void clear(iostate f = goodbit);                // set io state flags
    void setstate(iostate f) { clear(rdstate()|f); } // add f to io state flags

    operator void*() const;                         // nonzero if !fail()
    bool operator!() const { return fail(); }

    // ...
};
```

If the state is `good()` the previous input operation succeeded. If the state is `good()`, the next input

operation might succeed; otherwise, it will fail. Applying an input operation to a stream that is not in the *good*() state is a null operation. If we try to read into a variable *v* and the operation fails, the value of *v* should be unchanged (it is unchanged if *v* is a variable of one of the types handled by *istream* or *ostream* member functions). The difference between the states *fail*() and *bad*() is subtle. When the state is *fail*() but not also *bad*(), it is assumed that the stream is uncorrupted and that no characters have been lost. When the state is *bad*(), all bets are off.

The state of a stream is represented as a set of flags. Like most constants used to express the behavior of streams, these flags are defined in *basic_ios*' base *ios_base*:

```
class ios_base {
public:
    // ...

    typedef implementation_defined2 iostate;
    static const iostate badbit,     // stream is corrupted
                         eofbit,     // end-of-file seen
                         failbit,    // next operation will fail
                         goodbit;    // goodbit==0

    // ...
};
```

The I/O state flags can be directly manipulated. For example:

```
void f()
{
    ios_base::iostate s = cin.rdstate(); // returns a set of iostate bits

    if (s & ios_base::badbit) {
        // cin characters possibly lost
    }
    // ...
    cin.setstate(ios_base::failbit);
    // ...
}
```

When a stream is used as a condition, the state of the stream is tested by *operator void**() or *operator*!(). The test succeeds only if the state is *good*(). For example, a general copy function can be written like this:

```
template<class T> void iocopy(istream& is, ostream& os)
{
    T buf;
    while (is>>buf) os << buf << '\n';
}
```

The *is>>buf* returns a reference to *is*, which is tested by a call of *is*::*operator void**(). For example:

```
void f(istream& i1, istream& i2, istream& i3, istream& i4)
{
    iocopy<complex>(i1,cout);      // copy complex numbers
    iocopy<double>(i2,cout);       // copy doubles
    iocopy<char>(i3,cout);         // copy chars
    iocopy<string>(i4,cout);       // copy whitespace-separated words
}
```

21.3.4 Input of Characters [io.in.unformatted]

The >> operator is intended for formatted input; that is, reading objects of an expected type and format. Where this is not desirable and we want to read characters as characters and then examine them, we use the *get*() functions:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
    // ...
    // unformatted input:

    streamsize gcount() const;      // number of char read by last get()

    int_type get();                 // read one Ch (or Tr::eof())

    basic_istream& get(Ch& c);      // read one Ch into c

    basic_istream& get(Ch* p, streamsize n);        // newline is terminator
    basic_istream& get(Ch* p, streamsize n, Ch term);

    basic_istream& getline(Ch* p, streamsize n);    // newline is terminator
    basic_istream& getline(Ch* p, streamsize n, Ch term);

    basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof());
    basic_istream& read(Ch* p, streamsize n);       // read at most n char

    // ...
};
```

The *get*() and *getline*() functions treat whitespace characters exactly like other characters. They are intended for input operations, where one doesn't make assumptions about the meanings of the characters read.

The function *istream*::*get*(*char*&) reads a single character into its argument. For example, a character-by-character copy program can be written like this:

```
int main()
{
    char c;
    while(cin.get(c)) cout.put(c);
}
```

The three-argument *s*.*get*(*p*,*n*,*term*) reads at at most *n*-*1* characters into *p*[*0*]..*p*[*n*-*2*]. A call of *get*() will always place a *0* at the end of the characters (if any) it placed in the buffer, so *p*

must point to an array of at least *n* characters. The third argument, *term*, specifies a terminator. A typical use of the three-argument *get*() is to read a ''line'' into a fixed-sized buffer for further analysis. For example:

```
void f()
{
    char buf[100];
    cin >> buf;                 // suspect: will overflow some day
    cin.get(buf,100,'\n');      // safe
    // ...
}
```

If the terminator is found, it is left as the first unread character on the stream. Never call *get*() twice without removing the terminator. For example:

```
void subtle_infinite_loop()
{
    char buf[256];

    while (cin) {
        cin.get(buf,256);       // read a line
        cout << buf;            // print a line. Oops: forgot to remove '\n' from cin
    }
}
```

This example is a good reason to prefer *getline*() over *get*(). A *getline*() behaves like its corresponding *get*(), except that it removes its terminator from the *istream*. For example:

```
void f()
{
    char word[100][MAX];
    int i = 0;
    while(cin.getline(word[i++],100,'\n') && i<MAX);
    // ...
}
```

When efficiency isn't paramount, it is better to read into a *string* (§3.6, §20.3.15). In that way, the most common allocation and overflow problems cannot occur. However, the *get*(), *getline*(), and *read*() functions are needed to implement such higher-level facilities. The relatively messy interface is the price we pay for speed, for not having to re-scan the input to figure out what terminated the input operation, for being able to reliably limit the number of characters read, etc.

A call *read*(*p*,*n*) reads at most *n* characters into *p*[*0*]..*p*[*n-1*]. The read function does not rely on a terminator, and it doesn't put a terminating *0* into its target. Consequently, it really can read *n* characters (rather than just *n-1*). In other words, it simply reads characters and doesn't try to make its target into a C-style string.

The *ignore*() function reads characters like *read*(), but it doesn't store them anywhere. Like *read*(), it really can read *n* characters (rather than *n-1*). The default number of characters read by *ignore*() is *1*, so a call of *ignore*() without an argument means ''throw the next character away.'' Like *getline*(), it optionally takes a terminator and removes that terminator from the input stream if it gets to it. Note that *ignore*()'s default terminator is end-of-file.

For all of these functions, it is not immediately obvious what terminated the read – and it can be hard even to remember which function has what termination criterion. However, we can always inquire whether we reached end-of-file (§21.3.3). Also, *gcount*( ) gives the number of characters read from the stream by the most recent, unformatted input function call. For example:

```
void read_a_line(int max)
{
    // ...
    if (cin.fail()) {       // Oops: bad input format
        cin.clear();                // clear the input flags (§21.3.3)
        cin.ignore(max,';');        // skip to semicolon

        if (!cin) {
            // oops: we reached the end of the stream
        }
        else if (cin.gcount()==max) {
            // oops: read max characters
        }
        else {
            // found and discarded the semicolon
        }
    }
}
```

Unfortunately, if the maximum number of characters are read there is no way of knowing whether the terminator was found (as the last character).

The *get*( ) that doesn't take an argument is the <*iostream*> version of the <*cstdio*> *getchar*( ) (§21.8). It simply reads a character and returns the character's numeric value. In that way, it avoids making assumptions about the character type used. If there is no input character to return, *get*( ) returns a suitable ''end-of-file'' marker (that is, the stream's *traits_type*::*eof*( )) and sets the *istream* into *eof*-state (§21.3.3). For example:

```
void f(unsigned char* p)
{
    int i;
    while((i=cin.get()) && i!=EOF) {
        *p++ = i;
        // ...
    }
}
```

EOF is the value of *eof*( ) from the usual *char_traits* for *char*. *EOF* is presented in <*iostream*>. Thus, this loop could have been written *read*(*p*,*MAX_INT*), but presumably we wrote an explicit loop because we wanted to look at each character as it came in. It has been said that C's greatest strength is its ability to read a character and decide to do nothing with it – and to do this fast. It is indeed an important and underrated strength, and one that C++ aims to preserve.

The standard header <*cctype*> defines several functions that can be useful when processing input (§20.4.2). For example, an *eatwhite*( ) function that reads whitespace characters from a stream could be defined like this:

```
istream& eatwhite(istream& is)
{
    char c;
    while (is.get(c)) {
        if (!isspace(c)) {    // is c a whitespace character?
            is.putback(c); // put c back into the input buffer
            break;
        }
    }
    return is;
}
```

The call `is.putback(c)` makes `c` be the next character read from the stream `is` (§21.6.4).

21.3.5 Input of User-Defined Types [io.in.udt]

An input operation can be defined for a user-defined type exactly as an output operation was. However, for an input operation, it is essential that the second argument be of a non-*const* reference type. For example:

```
istream& operator>>(istream& s, complex& a)
/*
    input formats for a complex ("f" indicates a floating-point number):
        f
        (f)
        (f,f)
*/
{
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(ios_base::badbit);  // set state
    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}
```

Despite the scarcity of error-handling code, this will actually handle most kinds of errors. The local variable `c` is initialized to avoid having its value accidentally be `'('` after a failed first `>>` operation. The final check of the stream state ensures that the value of the argument `a` is changed only if everything went well.

The operation for setting a stream state is called *clear*() because its most common use is to reset the state of a stream to *good*(); *ios_base*::*goodbit* is the default argument value for *ios_base*::*clear*() (§21.3.3).

21.3.6 Exceptions [io.except]

It is not convenient to test for errors after each I/O operation, so a common cause of error is failing to do so where it matters. In particular, output operations are typically unchecked, but they do occasionally fail.

The only function that directly changes the state of a stream is *clear*(). Thus, an obvious way of getting notified by a state change is to ask *clear*() to throw an exception. The *ios_base* member *exceptions*() does just that:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    class failure;    // exception class (see §14.10)

    iostate exceptions() const;          // get exception state
    void exceptions(iostate except);     // set exception state

    // ...
};
```

For example,

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit);
```

requests that *clear*() should throw an *ios_base*::*failure* exception if *cout* goes into states *bad*, *fail*, or *eof* – in other words, if any output operation on *cout* doesn't perform flawlessly. Similarly,

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

allows us to catch the not-too-uncommon case in which the input is not in the format we expected, so an input operation didn't return a value from the stream.

A call of *exceptions*() with no arguments returns the set of I/O state flags that triggers an exception. For example:

```
void print_exceptions(ios_base& ios)
{
    ios_base::iostate s = ios.exceptions();
    if (s&ios_base::badbit) cout << "throws for bad";
    if (s&ios_base::failbit) cout << "throws for fail";
    if (s&ios_base::eofbit) cout << "throws for eof";
    if (s == 0) cout << "doesn´t throw";
}
```

The primary use of I/O exceptions is to catch unlikely – and therefore often forgotten – errors. Another is to control I/O. For example:

```
void readints(vector<int>& s)          // not my favorite style!
{
    ios_base::iostate old_state = cin.exceptions();  // save exception state
    cin.exceptions(ios_base::eofbit);                // throw for eof

    for (;;)
        try {
            int i;
            cin>>i;
            s.push_back(i);
        }
        catch(ios_base::eof) {
            // ok: end of file reached
        }

    cin.exceptions(old_state);        // reset exception state
}
```

The question to ask about this use of exceptions is, ''Is that an error?'' or ''Is that really exceptional?'' (§14.5). Usually, I find that the answer to either question is no. Consequently, I prefer to deal with the stream state directly. What can be handled with local control structures within a function is rarely improved by the use of exceptions.

21.3.7 Tying of Streams [io.tie]

The *basic_ios* function *tie*() is used to set up and break connections between an *istream* and an *ostream*:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
    // ...

    basic_ostream<Ch,Tr>* tie() const;                          // get pointer to tied stream
    basic_ostream<Ch,Tr>* tie(basic_ostream<Ch,Tr>* s);         // tie *this to s

    // ...
};
```

Consider:

```
string get_passwd()
{
    string s;
    cout << "Password: ";
    cin >> s;
    // ...
}
```

How can we be sure that *Password:* appears on the screen before the read operation is executed? The output on *cout* is buffered, so if *cin* and *cout* had been independent *Password:* would not have appeared on the screen until the output buffer was full. The answer is that *cout* is tied to *cin* by the operation *cin.tie*(&*cout*).

When an *ostream* is tied to an *istream*, the *ostream* is flushed whenever an input operation on the *istream* causes underflow; that is, whenever new characters are needed from the ultimate input source to complete the input operation. Thus,

```
cout << "Password: ";
cin >> s;
```

is equivalent to:

```
cout << "Password: ";
cout.flush();
cin >> s;
```

A stream can have at most one *ostream* at a time tied to it. A call `s.tie(0)` unties the stream `s` from the stream it was tied to, if any. Like most other stream functions that set a value, `tie(s)` returns the previous value; that is, it returns the previously tied stream or `0`. A call without an argument, `tie()`, returns the current value without changing it.

Of the standard streams, *cout* is tied to *cin* and *wcout* is tied to *wcin*. The *cerr* streams need not be tied because they are unbuffered, while the *clog* streams are not meant for user interaction.

21.3.8 Sentries [io.sentry]

When I wrote operators << and >> for *complex*, I did not worry about tied streams (§21.3.7) or whether changing stream state would cause exceptions (§21.3.6). I assumed – correctly – that the library-provided functions would take care of that for me. But how? There are a couple of dozen such functions. If we had to write intricate code to handle tied streams, *locale*s (§21.7), exceptions, etc., in each, then the code could get rather messy.

The approach taken is to provide the common code through a *sentry* class. Code that needs to be executed first (the ''prefix code'') – such as flushing a tied stream – is provided as the *sentry*'s constructor. Code that needs to be executed last (the ''suffix code'') – such as throwing exceptions caused by state changes – is provided as the *sentry*'s destructor:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
    // ...
    class sentry;
    // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream<Ch, Tr>::sentry {
public:
    explicit sentry(basic_ostream<Ch, Tr>& s);
    ~sentry();
    operator bool();

    // ...
};
```

Thus, common code is factored out and an individual function can be written like this:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& basic_ostream<Ch,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;
}
```

This technique of using constructors and destructors to provide common prefix and suffix code through a class is useful in many contexts.

Naturally, ***basic_istream*** has a similar ***sentry*** member class.

21.4 Formatting [io.format]

The examples in §21.2 were all of what is commonly called *unformatted output*. That is, an object was turned into a sequence of characters according to default rules. Often, the programmer needs more detailed control. For example, we need to be able to control the amount of space used for an output operation and the format used for output of numbers. Similarly, some aspects of input can be explicitly controlled.

Control of I/O formatting resides in class ***basic_ios*** and its base ***ios_base***. For example, class ***basic_ios*** holds the information about the base (octal, decimal, or hexadecimal) to be used when integers are written or read, the precision of floating-point numbers written or read, etc. It also holds the functions to set and examine these per-stream control variables.

Class ***basic_ios*** is a base of ***basic_istream*** and ***basic_ostream***, so format control is on a per-stream basis.

21.4.1 Format State [io.format.state]

Formatting of I/O is controlled by a set of flags and integer values in the stream's ***ios_base***:

```
class ios_base {
public:
    // ...
    // names of format flags:

    typedef implementation_defined1 fmtflags;
    static const fmtflags
        skipws,             // skip whitespace on input

        left,               // field adjustment: pad after value
        right,              // pad before value
        internal,           // pad between sign and value
```

```
        boolalpha,          // use symbolic representation of true and false

        dec,                // integer base: base 10 output (decimal)
        hex,                // base 16 output (hexadecimal)
        oct,                // base 8 output (octal)

        scientific,         // floating-point notation: d.dddddddEdd
        fixed,              // dddd.dd

        showbase,           // on output prefix oct by 0 and hex by 0x
        showpoint,          // print trailing zeros
        showpos,            // explicit '+' for positive ints
        uppercase,          // 'E', 'X' rather than 'e', 'x'

        adjustfield,        // flags related to field adjustment (§21.4.5)
        basefield,          // flags related to integer base (§21.4.2)
        floatfield;         // flags related to floating-point output (§21.4.3)

        fmtflags unitbuf;   // flush output after each output operation

    fmtflags flags() const;         // read flags
    fmtflags flags(fmtflags f);     // set flags

    fmtflags setf(fmtflags f) { return flags(flags()|f); }              // add flag
    fmtflags setf(fmtflags f, fmtflags mask) { return flags(flags()|(f&mask)); }// add flag
    void unsetf(fmtflags mask) { flags(flags()&~mask); }                // clear flags

    // ...
};
```

The values of the flags are implementation-defined. Use the symbolic names exclusively, rather than specific numeric values, even if those values happen to be correct on your implementation today.

Defining an interface as a set of flags, and providing operations for setting and clearing those flags is a time-honored if somewhat old-fashioned technique. Its main virtue is that a user can compose a set of options. For example:

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

This allows us to pass options around and install them where needed. For example:

```
void your_function(ios_base::fmtflags opt)
{
    ios_base::fmtflags old_options = cout.flags(opt);      // save old_options and set new ones
    // ...
    cout.flags(old_options);   // reset options
}

void my_function()
{
    your_function(my_opt);
    // ...
}
```

The *flags*() function returns the old option set.

Being able to read and set all options allows us to set an individual flag. For example:

```
myostream.flags(myostream.flags()|ios_base::showpos);
```

This makes *mystream* display an explicit + in front of positive numbers without affecting other options. The old options are read, and ***showpos*** is set by or-ing it into the set. The function ***setf***() does exactly that, so the example could equivalently have been written:

```
myostream.setf(ios_base::showpos);
```

Once set, a flag retains its value until it is unset.

Controlling I/O options by explicitly setting and clearing flags is crude and error-prone. For simple cases, manipulators (§21.4.6) provide a cleaner interface. Using flags to control stream state is a better study in implementation technique than in interface design.

21.4.1.1 Copying Format State [io.copyfmt]

The complete format state of a stream can be copied by ***copyfmt***():

```
template<class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    basic_ios& copyfmt(const basic_ios& f);
    // ...
};
```

The stream's buffer (§21.6) and the state of that buffer isn't copied by ***copyfmt***(). However, all of the rest of the state is, including the requested exceptions (§21.3.6) and any user-supplied additions to that state (§21.7.1).

21.4.2 Integer Output [io.out.int]

The technique of or-ing in a new option with ***flags***() or ***setf***() works only when a single bit controls a feature. This is not the case for options such as the base used for printing integers and the style of floating-point output. For such options, the value that specifies a style is not necessarily represented by a single bit or as a set of independent single bits.

The solution adopted in ***<iostream>*** is to provide a version of ***setf***() that takes a second ''pseudo argument'' that indicates which kind of option we want to set in addition to the new value. For example,

```
cout.setf(ios_base::oct, ios_base::basefield);   // octal
cout.setf(ios_base::dec, ios_base::basefield);   // decimal
cout.setf(ios_base::hex, ios_base::basefield);   // hexadecimal
```

sets the base of integers without side effects on other parts of the stream state. Once set, a base is used until reset. For example,

```
cout << 1234 << ' ' << 1234 << ' ';          //default: decimal
```

```
cout.setf(ios_base::oct,ios_base::basefield);   // octal
cout << 1234 << ' ' << 1234 << ' ';

cout.setf(ios_base::hex,ios_base::basefield);   // hexadecimal
cout << 1234 << ' ' << 1234 << ' ';
```

produces *1234 1234 2322 2322 4d2 4d2*.

If we need to be able to tell which base was used for each number, we can set ***showbase***. Thus, adding

```
cout.setf(ios_base::showbase);
```

before the previous operations, we get *1234 1234 02322 02322 0x4d2 0x4d2*. The standard manipulators (§21.4.6.2) provide a more elegant way of specifying the base of integer output.

21.4.3 Floating-Point Output [io.out.float]

Floating-point output is controlled by a *format* and a *precision*:

- The *general* format 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. It corresponds to ***printf***( )'s ***%g*** (§21.8).
- The *scientific* format 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. It corresponds to ***printf***( )'s ***%e***.
- The *fixed* format 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. It corresponds to ***printf***( )'s ***%f***.

We control the floating-point output format through the state manipulation functions. In particular, we can set the notation used for printing floating-point values without side effects on other parts of the stream state. For example,

```
cout << "default:\t" << 1234.56789 << '\n';

cout.setf(ios_base::scientific,ios_base::floatfield);   // use scientific format
cout << "scientific:\t" << 1234.56789 << '\n';

cout.setf(ios_base::fixed,ios_base::floatfield);        // use fixed-point format
cout << "fixed:\t" << 1234.56789 << '\n';

cout.setf(0,ios_base::floatfield);                      // reset to default (that is, general format)
cout << "default:\t" << 1234.56789 << '\n';
```

produces

```
default:    1234.57
scientific: 1.234568e+03
fixed:      1234.567890
default:    1234.57
```

The default precision (for all formats) is ***6***. The precision is controlled by an ***ios_base*** member function:

```
class ios_base {
public:
    // ...
    streamsize precision() const;          // get precision
    streamsize precision(streamsize n);    // set precision (and get old precision)
    // ...
};
```

A call of `precision()` affects all floating-point I/O operations for a stream up until the next call of `precision()`. Thus,

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

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

produces

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

Note that floating-point values are rounded rather than just truncated and that `precision()` doesn't affect integer output.

The *uppercase* flag (§21.4.1) determines whether *e* or *E* is used to indicate the exponents in the scientific format.

Manipulators provide a more elegant way of specifying output format for floating-point output (§21.4.6.2).

21.4.4 Output Fields [io.fields]

Often, we want to fill a specific space on an output line with text. We want to use exactly *n* characters and not fewer (and more only if the text does not fit). To do this, we specify a field width and a character to be used if padding is needed:

```
class ios_base {
public:
    // ...
    streamsize width() const;              // get field width
    streamsize width(streamsize wide);     // set field width
    // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    Ch fill() const;                       // get filler character
    Ch fill(Ch ch);                        // set filler character
    // ...
};
```

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 (§21.2.1), *string* (§20.3.15), and *bitfield* (§17.5.3.3). For example,

```
cout.width(4);
cout << 12;
```

will 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";
```

gives the output *##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*) function 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";
```

produces *abcdef* rather than just *abcd*. It is usually better to get the right output looking ugly than to get the wrong output looking just fine (see also §21.10[21]).

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

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

This produces *##12:13*, rather than *##12###:##13*, as would have been the case had *width*(*4*) applied to subsequent operations. Had all subsequent output operations been affected by *width*(), we would have had to explicitly specify *width*() for essentially all values.

The standard manipulators (§21.4.6.2) provide a more elegant way of specifying the width of an output field.

21.4.5 Field Adjustment [io.field.adjust]

The adjustment of characters within a field can be controlled by *setf*() calls:

```
cout.setf(ios_base::left, ios_base::adjustfield);      // left
cout.setf(ios_base::right, ios_base::adjustfield);     // right
cout.setf(ios_base::internal, ios_base::adjustfield);  // internal
```

This sets the adjustment of output within an output field defined by *ios_base*::*width*() without side effects on other parts of the stream state.

Adjustment can be specified like this:

```
cout.fill('#');

cout << '(';
cout.width(4);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::left,ios_base::adjustfield);
cout << -12 << "),(";

cout.width(4);
cout.setf(ios_base::internal,ios_base::adjustfield);
cout << -12 << ")";
```

This produces: `(#-12),(-12#),(-#12)`. Internal adjustment places fill characters between the sign and the value. As shown, right adjustment is the default.

21.4.6 Manipulators [io.manipulators]

To save the programmer from having to deal with the state of a stream in terms of flags, the standard library provides a set of functions for manipulating that state. The key idea is to insert an operation that modifies the state in between the objects being read or written. For example, we can explicitly request that an output buffer be flushed:

```
cout << x << flush << y << flush;
```

Here, `cout.flush()` is called at the appropriate times. This is done by a version of `<<` that takes a pointer to function argument and invokes it:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch,Tr> {
public:
    // ...

    basic_ostream& operator<<(basic_ostream& (*f)(basic_ostream&)) { return f(*this); }
    basic_ostream& operator<<(ios_base& (*f)(ios_base&));
    basic_ostream& operator<<(basic_ios<Ch,Tr>& (*f)(basic_ios<Ch,Tr>&));

    // ...
};
```

For this to work, a function must be a nonmember or static-member function with the right type. In particular, `flush()` is defined like this:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>& s)
{
    return s.flush();    // call ostream's member flush()
}
```

These declarations ensure that

```
cout << flush;
```

is resolved as

```
cout.operator<<(flush);
```

which calls

```
flush(cout);
```

which then invokes

```
cout.flush();
```

The whole rigmarole is done (at compile time) to allow **basic_ostream::flush()** to be called using the **cout<<flush** notation.

There is a wide variety of operations we might like to perform just before or just after an input or output operation. For example:

```
cout << x;
cout.flush();
cout << y;


cin.noskipws();      // don't skip whitespace
cin >> x;
```

When the operations are written as separate statements, the logical connections between the operations are not obvious. Once the logical connection is lost, the code gets harder to understand. The notion of manipulators allows operations such as **flush()** and **noskipws()** to be inserted directly in the list of input or output operations. For example:

```
cout << x << flush << y << flush;
cin >> noskipws >> x;
```

Naturally, class **basic_istream** provides >> operators for invoking manipulators in a way similar to class **basic_ostream**:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    // ...

    basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
    basic_istream& operator>>(basic_ios<Ch, Tr>& (*pf)(basic_ios<Ch, Tr>&));
    basic_istream& operator>>(ios_base& (*pf)(ios_base&));

    // ...
};
```

21.4.6.1 Manipulators Taking Arguments [io.manip.arg]

Manipulators that take arguments can also be useful. For example, we might want to write

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

to print the value of the floating-point variable *angle* with four digits.

To do this, *setprecision* must return an object that is initialized by *4* and that calls *cout*::*setprecision*(*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;

    smanip(ios_base& (*ff)(ios_base&, int), int ii) : f(ff), i(ii) { }
};

template<cladd Ch, class Tr>
ostream<Ch,Tr>& operator<<(ostream<Ch,Tr>& os, smanip& m)
{
    return m.f(os,m.i);
}
```

The *smanip* constructor stores its arguments in *f* and *i*, and *operator*<< calls *f*(*i*). We can now define *setprecision*() like this:

```
ios_base& set_precision(ios_base& s, int n)      // helper
{
    return s.setprecision(n); // call the member function
}

inline smanip setprecision(int n)
{
    return smanip(set_precision,n);      // make the function object
}
```

We can now write:

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

A programmer can define new manipulators in the style of *smanip* as needed (§21.10[22]). Doing this does not require modification of the definitions of standard library templates and classes such as *basic_istream*, *basic_ostream*, *basic_ios*, and *ios_base*.

21.4.6.2 Standard I/O Manipulators [io.std.manipulators]

The standard library provides manipulators corresponding to the various format states and state changes. The standard manipulators are defined in namespace *std*. Manipulators taking *io_base*, *istream*, and *ostream*arguments are presented in *<ios>*, *<ostream>*, and *<iostream>*, respectively. The rest of the standard manipulators are presented in *<iomanip>*.

```
ios_base& boolalpha(ios_base&);      // symbolic representation of true and false (input and output)
ios_base& noboolalpha(ios_base& s); // s.unsetf(ios_base::boolalpha)

ios_base& showbase(ios_base&);       // on output prefix oct by 0 and hex by 0x
ios_base& noshowbase(ios_base& s); // s.unsetf(ios_base::showbase)

ios_base& showpoint(ios_base&);
ios_base& noshowpoint(ios_base& s); // s.unsetf(ios_base::showpoint)

ios_base& showpos(ios_base&);
ios_base& noshowpos(ios_base& s);   // s.unsetf(ios_base::showpos)

ios_base& skipws(ios_base&);         // skip whitespace
ios_base& noskipws(ios_base& s);     // s.unsetf(ios_base::skipws)

ios_base& uppercase(ios_base&);      // X and E rather than x and e
ios_base& nouppercase(ios_base&);    // x and e rather than X and E

ios_base& internal(ios_base&);       // adjust §21.4.5
ios_base& left(ios_base&);           // pad after value
ios_base& right(ios_base&);          // pad before value

ios_base& dec(ios_base&);            // integer base is 10 (§21.4.2)
ios_base& hex(ios_base&);            // integer base is 16
ios_base& oct(ios_base&);            // integer base is 8

ios_base& fixed(ios_base&);          // floating-point format dddd.dd (§21.4.3)
ios_base& scientific(ios_base&);     // scientific format d.ddddEdd

template <class Ch, class Tr>
    basic_ostream<Ch,Tr>& endl(basic_ostream<Ch,Tr>&);    // put '\n' and flush
template <class Ch, class Tr>
    basic_ostream<Ch,Tr>& ends(basic_ostream<Ch,Tr>&);    // put '\0' and flush
template <class Ch, class Tr>
    basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>&);   // flush stream

template <class Ch, class Tr>
    basic_istream<Ch,Tr>& ws(basic_istream<Ch,Tr>&);      // eat whitespace

smanip resetiosflags(ios_base::fmtflags f);     // clear flags (§21.4)
smanip setiosflags(ios_base::fmtflags f);       // set flags (§21.4)
smanip setbase(int b);                          // output integers in base b
smanip setfill(int c);                          // make c the fill character
smanip setprecision(int n);                     // n digits after decimal point
smanip setw(int n);                             // next field is n char
```

For example,

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

produces `1234,4d2,2322` and

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

produces `(##12) (12)`.

When using manipulators that do not take arguments, *do not* add parentheses. When using standard manipulators that take arguments, remember to **#include <iomanip>**. For example:

```
#include <iostream>

int main()
{
    std::cout << setprecision(4)    // error: setprecision undefined (forgot <iomanip>)
        << scientific()             // error: ostream<<ostream& (spurious parentheses)
        << d << endl;
}
```

21.4.6.3 User-Defined Manipulators [io.ud.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.

The **precision** used persists for all output operations, but a **width()** operation 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 is 4

void f(double d)
{
    Form sci8 = gen4;
    sci8.scientific().precision(8); // scientific format, precision 8

    cout << d << ' ' << gen4(d) << ' ' << sci8(d) << ' ' << d << '\n';
}
```

A call `f(1234.56789)` writes

```
1234.57 1235 1.23456789e+03 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 Bound_form;    // Form plus value

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 (§21.4.3)
    // ...
```

```
public:
    explicit Form(int p = 6) : prc(p)    // default precision is 6
    {
        fmt = 0;    // general format (§21.4.3)
        wdt = 0;    // as wide as necessary
    }

    Bound_form operator()(double d) const; // make a Bound_form for *this and 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*** can then be output to a given stream by a suitable `<<` function:

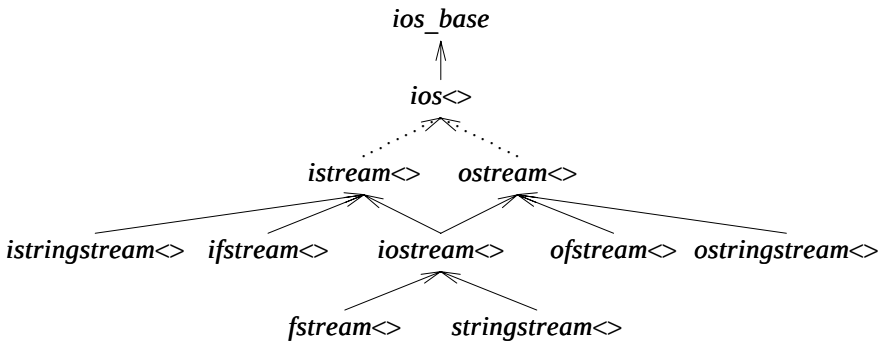
```
struct Bound_form {
    const Form& f;
    double val;

    Bound_form(const Form& ff, double v) : f(ff), val(v) { }
};

Bound_form Form::operator()(double d) { return Bound_form(*this,d); }

ostream& operator<<(ostream& os, const Bound_form& bf)
{
    ostringstream s;            // string streams are described in §21.5.3
    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 (§21.10[21]). The ***Form*** and ***Bound_form*** classes are easily extended for formatting integers, strings, etc. (see §21.10[20]).

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.

21.5 File Streams and String Streams [io.files]

When a C++ program starts, ***cout***, ***cerr***, ***clog***, ***cin***, and their wide-character equivalents (§21.2.1) are available for use. These streams are set up by default and their correspondence with I/O devices or files is determined by ''the system.'' In addition, you can create your own streams. In this case, you must specify to what the streams are attached. Attaching a stream to a file or to a ***string*** is common enough so as to be supported directly by the standard library. Here is the hierarchy of standard stream classes:



The classes suffixed by **<>** are templates parameterized on the character type, and their names have a ***basic_*** prefix. A dotted line indicates a virtual base class (§15.2.4).

Files and strings are examples of containers that you can both read from and write to. Consequently, you can have a stream that supports both **<<** and **>>**. Such a stream is called an ***iostream***, which is defined in namespace ***std*** and presented in **<*iostream*>**:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_iostream : public basic_istream<Ch,Tr>, public basic_ostream<Ch,Tr> {
public:
    explicit basic_iostream(basic_streambuf<Ch,Tr>* sb);
    virtual ~basic_iostream();
};

typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

Reading and writing from an ***iostream*** is controlled through the put-buffer and get-buffer operations on the ***iostream***'s ***streambuf*** (§21.6.4).

21.5.1 File Streams [io.filestream]

Here is a complete program that copies one file to another. The file names are taken as command-line arguments:

```
#include <fstream>
#include <cstdlib>
```

```
void error(const char* p, const char* p2 = "")
{
    cerr << p << ' ' << p2 << '\n';
    std::exit(1);
}

int main(int argc, char* argv[])
{
    if (argc != 3) error("wrong number of arguments");

    std::ifstream from(argv[1]);                    // open input file stream
    if (!from) error("cannot open input file",argv[1]);

    std::ofstream to(argv[2]);                      // open output file stream
    if (!to) error("cannot open output file",argv[2]);

    char ch;
    while (from.get(ch)) to.put(ch);

    if (!from.eof() || !to) error("something strange happened");
}
```

A file is opened for input by creating an object of class ***ifstream*** (input file stream) with the file name as the argument. Similarly, a file is opened for output by creating an object of class ***ofstream*** (output file stream) with the file name as the argument. In both cases, we test the state of the created object to see if the file was successfully opened.

A ***basic_ofstream*** is declared like this in ***<fstream>***:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ofstream : public basic_ostream<Ch,Tr> {
public:
    basic_ofstream();
    explicit basic_ofstream(const char* p, openmode m = out);

    basic_filebuf<Ch,Tr>* rdbuf() const;

    bool is_open() const;
    void open(const char* p, openmode m = out);
    void close();
};
```

As usual, ***typedef***s are available for the most common types:

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;

typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<wchar_t> wfstream;
```

An ***ifstream*** is like an ***ofstream***, except that it is derived from ***istream*** and is by default opened for reading. In addition, the standard library offers an ***fstream***, which is like an ***ofstream***, except that it is derived from ***iostream*** and by default can be both read from and written to.

File stream constructors take a second argument specifying alternative modes of opening:

```
class ios_base {
public:
    // ...

    typedef implementation_defined3 openmode;
    static openmode app,      // append
                    ate,      // open and seek to end of file (pronounced ''at end'')
                    binary,   // I/O to be done in binary mode (rather than text mode)
                    in,       // open for reading
                    out,      // open for writing
                    trunc;    // truncate file to 0-length

    // ...
};
```

The actual values of ***openmode***s and their meanings are implementation-defined. Please consult your systems and library manual for details – and do experiment. The comments should give some idea of the intended meaning of the modes. For example, we can open a file so that anything written to it is appended to the end:

```
ofstream mystream(name.c_str(),ios_base::app);
```

It is also possible to open a file for both input and output. For example:

```
fstream dictionary("concordance",ios_base::in|ios_base::out);
```

21.5.2 Closing of Streams [io.close]

A file can be explicitly closed by calling ***close***() on its stream:

```
void f(ostream& mystream)
{
    // ...

    mystream.close();
}
```

However, this is implicitly done by the stream's destructor. So an explicit call of ***close***() is needed only if the file must be closed before reaching the end of the scope in which its stream was declared.

This raises the question of how an implementation can ensure that the predefined streams ***cout***, ***cin***, ***cerr***, and ***clog*** are created before their first use and closed (only) after their last use. Naturally, different implementations of the ***<iostream>*** stream library can use different techniques to achieve this. After all, exactly how it is done is an implementation detail that should not be visible to the user. Here, I present just one technique that is general enough to be used to ensure proper order of construction and destruction of global objects of a variety of types. An implementation may be able to do better by taking advantage of special features of a compiler or linker.

The fundamental idea is to define a helper class that is a counter that keeps track of how many times ***<iostream>*** has been included in a separately compiled source file:

```
class ios_base::Init {
      static int count;
public:
      Init();
      ~Init();
};

namespace {                          // in <iostream>, one copy in each file #including <iostream>
      ios_base::Init __ioinit;
}

int ios_base::Init::count = 0;       // in some .c file
```

Each translation unit (§9.1) declares its own object called ***__ioinit***. The constructor for the ***__ioinit*** objects uses ***ios_base::Init::count*** as a first-time switch to ensure that actual initialization of the global objects of the stream I/O library is done exactly once:

```
ios_base::Init::Init()
{
      if (count++ == 0) { /* initialize cout, cerr, cin, etc. */ }
}
```

Conversely, the destructor for the ***__ioinit*** objects uses ***ios_base::Init::count*** as a last-time switch to ensure that the streams are closed:

```
ios_base::Init::~Init()
{
      if (--count == 0) { /* clean up cout (flush, etc.), cerr, cin, etc. */ }
}
```

This is a general technique for dealing with libraries that require initialization and cleanup of global objects. In a system in which all code resides in main memory during execution, the technique is almost free. When that is not the case, the overhead of bringing each object file into main memory to execute its initialization function can be noticeable. When possible, it is better to avoid global objects. For a class in which each operation performs significant work, it can be reasonable to test a first-time switch (like ***ios_base::Init::count***) in each operation to ensure initialization. However, that approach would have been prohibitively expensive for streams. The overhead of a first-time switch in the functions that read and write single characters would have been quite noticeable.

21.5.3 String Streams [io.stringstream]

A stream can be attached to a ***string***. That is, we can read from a ***string*** and write to a ***string*** using the formatting facilities provided by streams. Such streams are called a ***stringstream***s. They are defined in ***<sstream>***:

```
template <class Ch, class Tr=char_traits<Ch> >
class basic_stringstream : public basic_iostream<Ch,Tr> {
public:
      explicit basic_stringstream(ios_base::openmode m = out|in);
      explicit basic_stringstream(const basic_string<Ch>& s, openmode m = out|in);
```

```
        basic_string<Ch> str() const;              // get copy of string
        void str(const basic_string<Ch>& s);       // set value to copy of s

        basic_stringbuf<Ch,Tr>* rdbuf() const;
};

typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;

typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

For example, an ***ostringstream*** can be used to format message ***strings***:

```
extern const char* std_message[];

string compose(int n, const string& cs)
{
        ostringstream ost;
        ost << "error(" << n << ") " << std_message[n] << " (user comment: " << cs << ´)´;
        return ost.str();
}
```

There is no need to check for overflow because ***ost*** is expanded as needed. This technique can be most useful for coping with cases in which the formatting required is more complicated than what is common for a line-oriented output device.

An initial value can be provided for an ***ostringstream***, so we could equivalently have written:

```
string compose2(int n, const string& cs)
{
        ostringstream ost("error(");
        ost << n << ") " << std_message[n] << " (user comment: " << cs << ´)´;
        return ost.str();
}
```

An ***istringstream*** is an input stream reading from a ***string***:

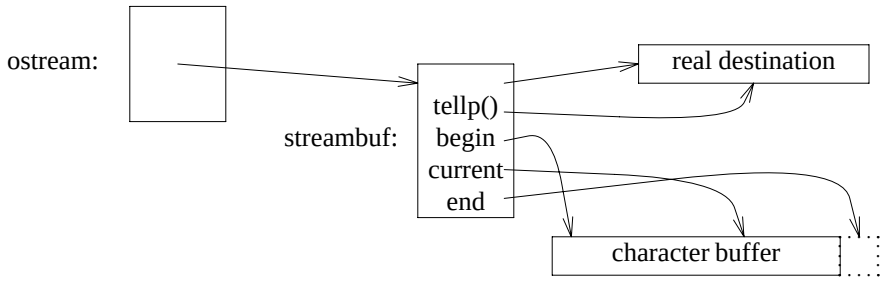
```
#include <sstream>

void word_per_line(const string& s)  // prints one word per line
{
        istringstream ist(s);
        string w;
        while (ist>>w) cout << w << ´\n´;
}

int main()
{
        word_per_line("If you think C++ is difficult, try English");
}
```

The initializer ***string*** is copied into the ***istringstream***. The end of the string terminates input.

It is possible to define streams that directly read from and write to arrays of characters (§21.10[26]). This is often useful when dealing with older code, especially since the ***ostrstream*** and ***istrstream*** classes doing that were part of the original streams library.

21.6 Buffering [io.buf]

Conceptually, an output stream puts characters into a buffer. Some time later, the characters are then written to wherever they are supposed to go. Such a buffer is called a ***streambuf*** (§21.6.4). 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 determines 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.

21.6.1 Output Streams and Buffers [io.ostreambuf]

An ***ostream*** provides operations for converting values of various types into character sequences according to conventions (§21.2.1) and explicit formatting directives (§21.4). In addition, an ***ostream*** provides operations that deal directly with its ***streambuf***:

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

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*** (§21.5.3) or an ***ofstream*** (§21.5.1) are created by initializing an ***ostream*** with a suitable ***streambuf*** (§21.6.4).

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***:

```
class ios_base {
        // ...

        typedef implementation_defined4 seekdir;
        static const seekdir beg,  // seek from beginning of current file
                             cur,  // seek from current position
                             end;  // seek backwards from end of current file

        // ...
};
```

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.seekp(10);
        fout << '#';
        fout.seekp(-2,ios_base::cur);
        fout << '*';
}
```

This places a # into ***file[10]*** and a * in ***file[8]***. There is no similar way to do random access on elements of a plain ***istream*** or ***ostream*** (see §21.10[13]).

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.

21.6.2 Input Streams and Buffers [io.istreambuf]

An ***istream*** provides operations for reading characters and converting them into values of various types (§21.3.1). In addition, an ***istream*** provides operations that deal directly with its ***streambuf***:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch,Tr> {
public:
        // ...

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

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

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

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

The positioning functions work like their *ostream* counterparts (§21.6.1). 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 an unwanted character back to be read some other time, as shown in §21.3.5. 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()* reads the next character but leaves it in the *streambuf* so that it can be read again. Thus, *c=peek()* is equivalent to *(c=get(),unget(),c)* and to *(putback(c=get()),c)*. Note that setting *failbit* might trigger an exception (§21.3.6).

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. Consequently, *sync()* returns *0* if it succeeded. If it failed, it sets *ios_base::badbit* (§21.3.3) and returns *-1*. Again, setting *badbit* might trigger an exception (§21.3.6).

The >> and *get()* operations that target a *streambuf* are primarily useful for implementers of I/O facilities. Only such implementers should manipulate *streambuf*s directly.

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()* (§21.6.4).

21.6.3 Streams and Buffers [io.rdbuf]

The connection between a stream and its buffer is maintained in the stream's *basic_ios*:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    basic_streambuf<charT,traits>* rdbuf() const;                 // get buffer
    basic_streambuf<charT,traits>* rdbuf(basic_streambuf<Ch,Tr>* b); // set buffer
```

```
    locale imbue(const locale& loc);          // set locale (and get old locale)

    char narrow(char_type c, char d) const;   // make char value from char_type c
    char_type widen(char c) const;            // make char_type value from char c

    // ...

protected:
    basic_ios();
    void init(basic_streambuf<Ch,Tr>* b);     // set initial buffer
};
```

In addition to reading and setting the stream's ***streambuf*** (§21.6.4), ***basic_ios*** provides ***imbue()*** to read and re-set the stream's locale (§21.7) by calling ***imbue()*** on its ***ios_base*** (§21.7.1) and ***pubimbue()*** on its buffer (§21.6.4).

The ***narrow()*** and ***widen()*** functions are used to convert ***char***s to and from a buffer's ***char_type***. The second argument of ***narrow(c,d)*** is the ***char*** returned if there isn't a ***char*** corresponding to the ***char_type*** value ***c***.

21.6.4 Stream Buffers [io.streambuf]

The I/O operations are specified without any mention of file types, but not all devices can be treated identically with respect to buffering strategies. For example, an ***ostream*** bound to a ***string*** (§21.5.3) needs a different kind of buffer than does an ***ostream*** bound to a file (§21.5.1). These problems are handled by providing different buffer types for different streams at the time of initialization. There is only one set of operations on these buffer types, so the ***ostream*** functions do not contain code distinguishing them. The different types of buffers are derived from class ***streambuf***. Class ***streambuf*** provides virtual functions for operations where buffering strategies differ, such as the functions that handle overflow and underflow.

The ***basic_streambuf*** class provides two interfaces. The public interface is aimed primarily at implementers of stream classes such as ***istream***, ***ostream***, ***fstream***, ***stringstream***, etc. In addition, a protected interface is provided for implementers of new buffering strategies and of ***streambuf***s for new input sources and output destinations.

To understand a ***streambuf***, it is useful first to consider the underlying model of a buffer area provided by the protected interface. Assume that the ***streambuf*** has a *put area* into which << writes, and a *get area* from which >> reads. Each area is described by a beginning pointer, current pointer, and one-past-the-end pointer. These pointers are made available through functions:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:
    Ch* eback() const;          // start of get-buffer
    Ch* gptr() const;           // next filled character (next char read comes from here)
    Ch* egptr() const;          // one-past-end of get-buffer

    void gbump(int n);          // add n to gptr()
    void setg(Ch* begin, Ch* next, Ch* end); // set eback(), gptr(), and egptr()
```

```
        Ch* pbase() const;          // start of put-buffer
        Ch* pptr() const;           // next free char (next char written goes here)
        Ch* epptr() const;          // one-past-end of put-buffer
        void pbump(int n);          // add n to pptr()
        void setp(Ch* begin, Ch* end);      // set pbase() and pptr() to begin, and epptr() to end
        // ...
};
```

Given an array of characters, *setg()* and *setp()* can set up the pointers appropriately. An implementation might access its get area like this:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_streambuf<Ch,Tr>::int_type basic_streambuf<Ch,Tr>::snextc() // read next character
{
        if (gptr()==0) return uflow();              // no input buffering
        gbump(1);                                   // move to next character
        if (gptr()>=egptr()) return underflow(); // re-fill buffer
        return *gptr();                             // return the now current character
}
```

The public interface of a *streambuf* looks like this:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
public:
        // usual typedefs (§21.2.1)

        basic_streambuf();
        virtual ~basic_streambuf();

        locale pubimbue(const locale &loc);             // set locale (and get old locale)
        locale getloc() const;                          // get locale

        basic_streambuf* pubsetbuf(Ch* p, streamsize n);    // set buffer space

        // position (§21.6.1):
        pos_type pubseekoff(off_type off, ios_base::seekdir way,
                            ios_base::openmode m = ios_base::in|ios_base::out);
        pos_type pubseekpos(pos_type p, ios_base::openmode m = ios_base::in|ios_base::out);

        int pubsync();                                  // sync() input (§21.6.2)


        int_type    snextc();                       // get next character
        int_type    sbumpc();                       // advance gptr() by 1
        int_type    sgetc();                        // get current char
        streamsize sgetn(Ch* p, streamsize n);      // get into p[0]..p[n-1]

        int_type    sputbackc(Ch c);    // put c back into buffer (§21.6.2)
        int_type    sungetc();          // unget last char

        int_type    sputc(Ch c);                        // put c
        streamsize sputn(const Ch* p, streamsize n);    // put p[0]..p[n-1]

        streamsize in_avail();              // is input ready?
```

```
    // ...
};
```

The public interface contains functions for inserting characters into the buffer and extracting characters from the buffer. These functions are simple and easily inlined. This is crucial for efficiency.

Functions that implement parts of a specific buffering strategy invoke corresponding functions in the protected interface. For example, ***pubsetbuf***`()` calls ***setbuf***`()`, which is overridden by a derived class to implement that class' notion of getting memory for the buffered characters. Using two functions to implement an operation such as ***setbuf***`()` allows an ***iostream*** implementer to do some ''housekeeping'' before and after the user's code. For example, an implementer might wrap a *try-block* around the call of the virtual function and catch exceptions thrown by the user code. This use of a pair of public and protected functions is yet another general technique that just happens to be useful in the context of I/O.

By default, ***setbuf***`(0,0)` means ''unbuffered'' and ***setbuf***`(p,n)` means use `p[0]..p[n-1]` to hold buffered characters.

A call to ***in_avail***`()` is used to see how many characters are available in the buffer. This can be used to avoid waiting for input. When reading from a stream connected to a keyboard, ***cin***`.`***get***`(c)` might wait until the user comes back from lunch. On some systems and for some applications, it can be worthwhile taking that into account when reading. For example:

```
if (cin.rdbuf().in_avail()) {   // get() will not block
    cin.get(c);
    // do something
}
else {                          // get() might block
    // do something else
}
```

In addition to the public interface used by ***basic_istream*** and ***basic_ostream***, ***basic_streambuf*** offers a protected interface to implementers of ***streambuf***s. This is where the virtual functions that determine policy are declared:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:
    // ...

    virtual void imbue(const locale &loc);                  // set locale

    virtual basic_streambuf* setbuf(Ch* p, streamsize n);

    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                    ios_base::openmode m = ios_base::in|ios_base::out);
    virtual pos_type seekpos(pos_type p,
                    ios_base::openmode m = ios_base::in|ios_base::out);

    virtual int sync();                                     // sync() input (§21.6.2)
```

```
        virtual int showmanyc();
        virtual streamsize xsgetn(Ch* p, streamsize n);          // get n chars
        virtual int_type underflow();                            // get area empty
        virtual int_type uflow();

        virtual int_type pbackfail(int_type c = Tr::eof());      // putback failed

        virtual streamsize xsputn(const Ch* p, streamsize n);    // put n chars
        virtual int_type overflow(int_type c = Tr::eof());       // put area full
};
```

The ***underflow***() and ***uflow***() functions are called to get the next character from the real input source when the buffer is empty. If no more input is available from that source, the stream is set into ***eof*** state (§21.3.3). If doing that doesn't cause an exception, ***traits_type***::***eof***() is returned. Unbuffered input uses ***uflow***(); buffered input uses ***underflow***(). Remember that there typically are more buffers in your system than the ones introduced by the ***iostream*** library, so you can suffer buffering delays even when using unbuffered stream I/O.

The ***overflow***() function is called to transfer characters to the real output destination when the buffer is full. A call ***overflow***(*c*) outputs the contents of the buffer plus the character *c*. If no more output can be written to that target, the stream is put into ***eof*** state (§21.3.3). If doing that doesn't cause an exception, ***traits_type***::***eof***() is returned.

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 disc 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 a few experiments.

By default, every stream gets the global locale (§21.7). A ***pubimbue***(***loc***) or ***imbue***(***loc***) call makes a stream use ***loc*** as its locale.

A ***streambuf*** for a particular kind of stream is derived from ***basic_streambuf***. It provides the constructors and initialization functions that connect the ***streambuf*** to a real source of (target for) characters and overrides the virtual functions that determine the buffering strategy. For example:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_filebuf : public basic_streambuf<Ch,Tr> {
public:
        basic_filebuf();
        virtual ~basic_filebuf();

        bool is_open() const;
        basic_filebuf* open(const char* p, ios_base::openmode mode);
        basic_filebuf* close();

protected:
        virtual int showmanyc();
        virtual int_type underflow();
        virtual int_type uflow();
```

```
    virtual int_type pbackfail(int_type c = Tr::eof());
    virtual int_type overflow(int_type c = Tr::eof());

    virtual basic_streambuf<Ch,Tr>* setbuf(Ch* p, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                    ios_base::openmode m = ios_base::in|ios_base::out);
    virtual pos_type seekpos(pos_type p,
                    ios_base::openmode m = ios_base::in|ios_base::out);
    virtual int sync();
    virtual void imbue(const locale& loc);
};
```

The functions for manipulating buffers, etc., are inherited unchanged from ***basic_streambuf***. Only functions that affect initialization and buffering policy need to be separately provided.

As usual, the obvious ***typedef***s and their wide stream counterparts are provided:

```
typedef basic_streambuf<char> streambuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_filebuf<char> filebuf;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<wchar_t> wfilebuf;
```

21.7 Locale [io.locale]

A ***locale*** is an object that controls the classification of characters into letters, digits, etc.; the collation order of strings; and the appearance of numeric values on input and output. Most commonly a ***locale*** is used implicitly by the ***iostream***s library to ensure that the usual conventions for some natural language or culture is adhered to. In such cases, a programmer never sees a ***locale*** object. However, by changing a ***stream***'s ***locale***, a programmer can change the way the stream behaves to suit a different set of conventions

A locale is an object of class ***locale*** defined in namespace ***std*** presented in <***locale***>:

```
class locale {
public:
    // ...

    locale() throw();                           // copy of current global locale
    explicit locale(const char* name);          // construct locale using C locale name
    basic_string<char> name() const;            // give name of this locale

    locale(const locale&) throw();                      // copy locale
    const locale& operator=(const locale& ) throw();    // copy locale

    static locale global(const locale&);        // set the global locale (get the previous locale)
    static const locale& classic();             // get the locale that C defines
};
```

Here, I omitted all of the interesting pieces and left only what is needed to switch from one existing locale to another. For example:

```
void f()
{
    std::locale loc("POSIX");           // standard locale for POSIX

    cin.imbue(loc);                     // let cin use loc
    // ...
    cin.imbue(std::locale::global());   // reset cin to use the default locale
}
```

The *imbue()* function is a member of *basic_ios* (§21.7.1).

As shown, some fairly standard locales have character string names. These tend to be shared with C.

It is possible to set the *locale* that is used by all newly constructed streams:

```
void g(const locale& loc = locale())        // use current global locale by default
{
    locale old_global = locale::global(loc);    // make loc the default locale
    // ...
}
```

Setting the global *locale* does not change the behavior of existing streams that are using the previous value of the global *locale*. In particular, *cin*, *cout*, etc., are not affected. If they should be changed, they must be explicitly *imbue()*d.

Imbuing a stream with a *locale* changes facets of its behavior. It is possible to use members of a *locale* directly, to define new *locale*s, and to extend *locale*s with new facets. For example, a *locale* can also be used explicitly to control the appearance of monetary units, dates, etc., on input and output (§21.10[25]) and conversion between codesets. However, discussion of that is beyond the scope of this book. Please consult your implementation's documentation.

The C-style locale is presented in <*clocale*> and <*locale.h*>.

21.7.1 Stream Callbacks [io.callbacks]

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()*:

```
class ios_base {
public:
    // ...

    ~ios_base();

    locale imbue(const locale& loc);    // get and set locale
    locale getloc() const;              // get locale

    static int xalloc();        // get an integer and a pointer (both initialized to 0)
    long& iword(int i);         // access the integer iword(i)
    void*& pword(int i);        // access the pointer pword(i)
```

```
        // callbacks:
        enum event { erase_event, imbue_event, copyfmt_event };  // event type
        typedef void (*event_callback)(event, ios_base&, int i);
        void register_callback(event_callback f, int i);          // attach f to word(i)
};
```

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 the state changes, registered functions are called with the argument ***i*** supplied by their ***register_callback***( ).

This storage and callback mechanism is fairly obscure. Use it only when you absolutely need to extend the low-level formatting facilities.

21.8 C Input/Output [io.c]

Because C++ and C code are often intermixed, C++ stream I/O is sometimes mixed with the C ***printf***( ) family of I/O functions. The C-style I/O functions are presented by ***<cstdio>*** and ***<stdio.h>***. Also, because C functions can be called from C++ some programmers may prefer to use the more familiar C I/O functions. Even if you prefer stream I/O, you will undoubtedly encounter C-style I/O at some time.

C and C++ I/O can be mixed on a per-character basis. A call of ***sync_with_stdio***( ) before the first stream I/O operation in the execution of a program guarantees that the C-style and C++-style 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 on some implementations.

```
class ios_base {
        // ...
        static bool sync_with_stdio(bool sync = true);  // get and set
};
```

The general advantage of the stream output functions over the C standard library function ***printf***( ) is that the stream functions are type safe and have a common style for specifying output of objects of built-in and user-defined types.

The general C output functions

```
int printf(const char* format ...);              // write to stdout
int fprintf(FILE*, const char* format ...);      // write to ''file'' (stdout, stderr)
int sprintf(char* p, const char* format ...);    // write to p[0]..
```

produce formatted output of an arbitrary sequence of arguments under control of the format string ***format***. The format string contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next argument. Each conversion specification is introduced by the character **%**. For example:

```
printf("there were %d members present.",no_of_members);
```

Here *%d* specifies that *no_of_members* is to be treated as an *int* and printed as the appropriate sequence of decimal digits. With *no_of_members==127*, the output is

there were 127 members present.

The set of conversion specifications is quite large and provides a great degree of flexibility. Following the %, there may be:

- `-` an optional minus sign that specifies left-adjustment of the converted value in the field;
- `+` an optional plus sign that specifies that a value of a signed type will always begin with a `+` or `-` sign;
- `#` an optional # that specifies that floating-point values will be printed with a decimal point even if no nonzero digits follow, that trailing zeroes will be printed, that octal values will be printed with an initial *0*, and that hexadecimal values will be printed with an initial *0x* or *0X*;
- *d* an optional digit string specifying a field width; if the converted value has fewer characters than the field width, it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- `.` an optional period that serves to separate the field width from the next digit string;
- *d* an optional digit string specifying a precision that specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- `*` a field width or precision may be `*` instead of a digit string. In this case an integer argument supplies the field width or precision;
- `h` an optional character *h*, specifying that a following *d*, *o*, *x*, or *u* corresponds to a short integer argument;
- `l` an optional character *l*, specifying that a following *d*, *o*, *x*, or *u* corresponds to a long integer argument;
- `%` indicating that the character % is to be printed; no argument is used;
- *c* a character that indicates the type of conversion to be applied. The conversion characters and their meanings are:
  - `d` The integer argument is converted to decimal notation;
  - `o` The integer argument is converted to octal notation;
  - `x` The integer argument is converted to hexadecimal notation with an initial *0x*;
  - `X` The integer argument is converted to hexadecimal notation with an initial *0X*;
  - `f` The *float* or *double* argument is converted to decimal notation in the style *[-]ddd.ddd*. The number of *d*'s after the decimal point is equal to the precision for the argument. If necessary, the number is rounded. If the precision is missing, six digits are given; if the precision is explicitly *0* and # isn't specified, no decimal point is printed;
  - `e` The *float* or *double* argument is converted to decimal notation in the scientific style *[-]d.ddde+dd* or *[-]d.ddde-dd*, where there is one digit before the decimal point and the number of digits after the decimal point is equal to the precision specification for the argument. If necessary, the number is rounded. If the precision is missing, six digits are given; if the precision is explicitly *0* and # isn't specified, no digits and no decimal point are printed;

- E As ***e***, but with an uppercase ***E*** used to identify the exponent;
- g The ***float*** or ***double*** argument is printed in style d, in style f, or in style e, whichever gives the greatest precision in minimum space;
- G As ***g***, but with an uppercase ***E*** used to identify the exponent.
- c The character argument is printed. Null characters are ignored;
- s The argument is taken to be a string (character pointer), and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is 0 or missing, all characters up to a null are printed.
- p The argument is taken to be a pointer. The representation printed is implementation-dependent.
- u The unsigned integer argument is converted to decimal notation;

In no case does a nonexistent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.

Here is a more elaborate example:

```
char* line_format = "\n#line %d \"%s\"\n";

int main()
{
        int line = 13;
        char* file_name = "C++/main.c";

        printf("int a;\n");
        printf(line_format,line,file_name);
        printf("int b;\n");
}
```

which produces:

```
int a;

#line 13 "C++/main.c"
int b;
```

Using ***printf()*** is unsafe in the sense that type checking is not done. For example, here is a well-known way of getting unpredictable output, a core dump, or worse:

```
char x;
// ...
printf("bad input char: %s",x);          // %s should have been %c
```

The ***printf()*** does, however, provide great flexibility in a form that is familiar to C programmers.

Similarly, ***getchar()*** provides a familiar way of reading characters from input:

```
int i;
while ((i=getchar())!=EOF) {// C character input
        // use i
}
```

Note that to be able to test for end-of-file against the ***int*** value ***EOF***, the value of ***getchar()*** must be put into an ***int*** rather than into a ***char***.

For further details of C I/O, see your C reference manual or Kernighan and Ritchie: *The C Programming Language [Kernighan,1988]*.

21.9 Advice [io.advice]

[1] Define << and >> for user-defined types with values that have meaningful textural representations; §21.2.3, §21.3.5.
[2] Use parentheses when printing expressions containing operators of low precedence; §21.2.
[3] You don't need to modify ***istream*** or ***ostream*** to add new << and >> operators; §21.2.3.
[4] You can define a function so that it behaves as a ***virtual*** function based on its *second* (or subsequent) argument; §21.2.3.1.
[5] Remember that by default >> skips whitespace; §21.3.2.
[6] Use lower-level input functions such as ***get***() and ***read***() primarily in the implementation of higher-lever input functions; §21.3.4.
[7] Be careful with the termination criteria when using ***get***(), ***getline***(), and ***read***(); §21.3.4.
[8] Prefer manipulators to state flags for controlling I/O; §21.3.3, §21.4, §21.4.6.
[9] Use exceptions to catch rare I/O errors (only); §21.3.6.
[10] Tie streams used for interactive I/O; §21.3.7.
[11] Use sentries to concentrate entry and exit code for many functions in one place; §21.3.8.
[12] Don't use parentheses after a no-argument manipulator; §21.4.6.2.
[13] Remember to #***include*** <***iomanip***> when using standard manipulators; §21.4.6.2.
[14] You can achieve the effect (and efficiency) of a ternary operator by defining a simple function object; §21.4.6.3.
[15] Remember that ***width*** specifications apply to the following I/O operation only; §21.4.4.
[16] Remember that ***precision*** specifications apply to all following floating-point output operations; §21.4.3.
[17] Use string streams for in-memory formatting; §21.5.3.
[18] You can specify a mode for a file stream ; §21.5.1.
[19] Distinguish sharply between formatting (***iostream***s) and buffering (***streambuf***s) when extending the I/O system; §21.1, §21.6.
[20] Implement nonstandard ways of transmitting values as stream buffers; §21.6.4.
[21] Implement nonstandard ways of formatting values as stream operations; §21.2.3, §21.3.5.
[22] You can isolate and encapsulate calls of user-defined code by using a pair of functions; §21.6.4.
[23] You can use ***in_avail***() to determine whether an input operation will block before reading; §21.6.4.
[24] Distinguish between simple operations that need to be efficient and operations that implement policy (make the former ***inline*** and the latter ***virtual***); §21.6.4.
[25] Use ***locale*** to localize ''cultural differences;'' §21.7.
[26] Use ***sync_with_stdio***(*x*) to mix C-style and C++-style I/O and to disassociate C-style and C++-style I/O; §21.8.
[27] Beware of type errors in C-style I/O; §21.8.

21.10 Exercises [io.exercises]

1. (∗1.5) Read a file of floating-point numbers, make complex numbers out of pairs of numbers read, and write out the complex numbers.
2. (∗1.5) Define a type ***Name_and_address***. Define `<<` and `>>` for it. Copy a stream of ***Name_and_address*** objects.
3. (∗2.5) Copy a stream of ***Name_and_address*** objects in which you have inserted as many errors as you can think of (e.g., format errors and premature end of string). Handle these errors in a way that ensures that the copy function reads most of the correctly formatted ***Name_and_address***es, even when the input is completely messed up.
4. (∗2.5) Redefine the I/O format ***Name_and_address*** to make it more robust in the presence of format errors.
5. (∗2.5) Design some functions for requesting and reading information of various types. Ideas: integer, floating-point number, file name, mail address, date, personal information, etc. Try to make them foolproof.
6. (∗1.5) Write a program that prints (a) all lowercase letters, (b) all letters, (c) all letters and digits, (d) all characters that may appear in a C++ identifier on your system, (e) all punctuation characters, (f) the integer value of all control characters, (g) all whitespace characters, (h) the integer value of all whitespace characters, and finally (i) all printing characters.
7. (∗2) Read a sequence of lines of text into a fixed-sized character buffer. Remove all whitespace characters and replace each alphanumeric character with the next character in the alphabet (replace ***z*** by ***a*** and ***9*** by ***0***). Write out the resulting line.
8. (∗3) Write a ''miniature'' stream I/O system that provides classes ***istream***, ***ostream***, ***ifstream***, ***ofstream*** providing functions such as ***operator*`<<()`** and ***operator*`>>()`** for integers and operations such as ***open*`()`** and ***close*`()`** for files.
9. (∗4) Implement the C standard I/O library (`<`***stdio.h***`>`) using the C++ standard I/O library (`<`***iostream***`>`).
10. (∗4) Implement the C++ standard I/O library (`<`***iostream***`>`) using the C standard I/O library (`<`***stdio.h***`>`).
11. (∗4) Implement the C and C++ libraries so that they can be used simultaneously.
12. (∗2) Implement a class for which `[]` is overloaded to implement random reading of characters from a file.
13. (∗3) Repeat §21.10[12] but make `[]` useful for both reading and writing. Hint: Make `[]` return an object of a ''descriptor type'' for which assignment means ''assign through descriptor to file'' and implicit conversion to ***char*** ''means read from file through descriptor.''
14. (∗2) Repeat §21.10[13] but let `[]` index objects of arbitrary types, not just characters.
15. (∗3.5) Implement versions of ***istream*** and ***ostream*** that read and write numbers in their binary form rather than converting them into a character representation. Discuss the advantages and disadvantages of this approach compared to the character-based approach.
16. (∗3.5) Design and implement a pattern-matching input operation. Use ***printf***-style format strings to specify a pattern. It should be possible to try out several patterns against some input to find the actual format. One might derive a pattern-matching input class from ***istream***.
17. (∗4) Invent (and implement) a much better kind of pattern for pattern matching. Be specific about what is better about it.

18. (∗2) Define an output manipulator ***based*** that takes two arguments – a base and an ***int*** value – and outputs the integer in the representation specified by the base. For example, ***based***(*2*,*9*) should print ***1001***.
19. (∗2) Write manipulators that turn character echoing on and off.
20. (∗2) Implement ***Bound_form*** from §21.4.6.3 for the usual set of built-in types.
21. (∗2) Re-implement ***Bound_form*** from §21.4.6.3 so that an output operation never overflows its ***width***(). It should be possible for a programmer to ensure that output is never quietly truncated beyond its specified precision.
22. (∗3) Implement an ***encrypt***(*k*) manipulator that ensures that output on its ***ostream*** is encrypted using the key *k*. Provide a similar ***decrypt***(*k*) manipulator for an ***istream***. Provide the means for turning the encryption off for a stream so that further I/O is cleartext.
23. (∗2) Trace a character's route through your system from the keyboard to the screen for a simple:

```
char c;
cin >> c;
cout << c << endl;
```

24. (∗2) Modify ***readints***() (§21.3.6) to handle all exceptions. Hint: Resource acquisition is initialization.
25. (∗2.5) There is a standard way of reading, writing, and representing dates under control of a ***locale***. Find it in the documentation of your implementation and write a small program that reads and writes dates using this mechanism. Hint: ***struct tm***.
26. (∗2.5) Define an ***ostream*** called ***ostrstream*** that can be attached to an array of characters (a C-style string) in a way similar to the way ***ostringstream*** is attached to a ***string***. However, do not copy the array into or out of the ***ostrstream***. The ***ostrstream*** should simply provide a way of writing to its array argument. It might be used for in-memory formatting like this:

```
char buf[message_size];
ostrstream ost(buf,message_size);
do_something(arguments,ost);        // output to buf through ost
cout << buf;                        // ost adds terminating 0
```

    An operation such as ***do_something***() can write to the stream ***ost***, pass ***ost*** on to its suboperations, etc., using the standard output operations. There is no need to check for overflow because ***ost*** knows its size and will go into ***fail***() state when it is full. Finally, a ***display***() operation can write the message to a ''real'' output stream. This technique can be most useful for coping with cases in which the final display operation involves writing to something more complicated than a traditional line-oriented output device. For example, the text from ***ost*** could be placed in a fixed-sized area somewhere on a screen. Similarly, define class ***istrstream*** as an input string stream reading from a zero-terminated string of characters. Interpret the terminating zero character as end-of-file. These ***strstream***s were part of the original streams library and can often be found in ***<strstream.h>***.
27. (∗2.5) Implement a manipulator ***general***() that resets a stream to its original (general) format in the same way a ***scientific***() (§21.4.6.2) sets a stream to use scientific format.