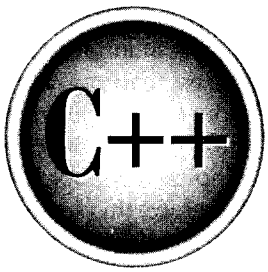
The Complete Reference



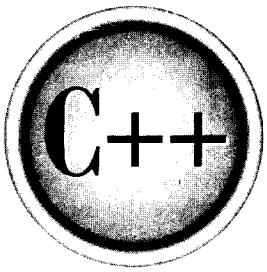
Part IV

The Standard C++ Class Library

Standard C++ defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. The class library is in addition to the function library described in Part Three. The class library forms a major portion of the C++ language and defines much of its character. Despite its size, the class library is easy to master because it is organized around object-oriented principles.

The Standard C++ library is quite large and an in-depth description of all of its classes, features, attributes, and implementation details is beyond the scope of this book. (A full description of the class library would easily fill a large book!) However, while most of the class library is for general use, some of it is intended mostly for compiler developers, or those programmers implementing extensions or enhancements. Therefore, this section describes only those parts of the class library that are typically used in an application. If you will be using the library for specialized work, you will need to acquire a copy of the C++ standard, which contains the technical description of the class library.

The Complete Reference



Chapter 32

The Standard C++ I/O Classes

This chapter describes the Standard C++ I/O class library. As explained in Part Two, there are currently two versions of C++'s I/O library in common use. The first is the old-style library, which is not defined by Standard C++. The second is the modern, templatized Standard C++ I/O system. Since the modern I/O library is essentially a superset of the old-style one, its is the only one described here. However, much of the information still applies to the older version.

Note *For an overview of C++ I/O, see Chapters 20 and 21.*

The I/O Classes

The Standard C++ I/O system is constructed from a rather complex system of template classes. These classes are shown here.

Class	Purpose
basic_ios	Provides general-purpose I/O operations
basic_streambuf	Low-level support for I/O
basic_istream	Support for input operations
basic_ostream	Support for output operations
basic_iostream	Support for input/output operations
basic_filebuf	Low-level support for file I/O
basic_ifstream	Support for file input
basic_ofstream	Support for file output
basic_fstream	Support for file input/output
basic_stringbuf	Low-level support for string-based I/O
basic_istringstream	Support for string-based input
basic_ostringstream	Support for string-based output
basic_stringstream	Support for string-based input/output

Also part of the I/O class hierarchy is the non-template class **ios_base**. It provides definitions for various elements of the I/O system.

The C++ I/O system is built upon two related but different template class hierarchies. The first is derived from the low-level I/O class called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. The classes **basic_filebuf** and **basic_stringbuf** are derived from **basic_streambuf**. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** or its subclasses directly.

The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error-checking, and status information related to stream I/O. **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively. Specifically, from **basic_istream** are derived the classes **basic_ifstream** and **basic_istringstream**, from **basic_ostream** are derived **basic_ofstream** and **basic_ostringstream**, and from **basic_iostream** are derived **basic_fstream** and **basic_stringstream**. A base class for **basic_ios** is **ios_base**. Thus, any class derived from **basic_ios** has access to the members of **ios_base**.

The I/O classes are parameterized for the type of characters that they act upon and for the traits associated with those characters. For example, here is the template specification for **basic_ios**:

```
template <class CharType, class Attr = char_traits<CharType> >
  class basic_ios: public ios_base
```

Here, **CharType** specifies the type of character (such as **char** or **wchar_t**) and **Attr** specifies a type that describes its attributes. The generic type **char_traits** is a utility class that defines the attributes associated with a character.

As explained in Chapter 20, the I/O library creates two specializations of the template class hierarchies just described: one for 8-bit characters and one for wide characters. Here is a complete list of the mapping of template class names to their character and wide-character versions.

Template Class	Character-Based Class	Wide-Character-Based Class
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream

Template Class	Character-Based Class	Wide-Character-Based Class
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream
basic_fstream	fstream	wfstream
basic_istringstream	istringstream	wistringstream
basic_ostringstream	ostringstream	wostringstream
basic_stringstream	stringstream	wstringstream
basic_streambuf	streambuf	wstreambuf
basic_filebuf	filebuf	wfilebuf
basic_stringbuf	stringbuf	wstringbuf

Since the vast majority of programmers will be using character-based I/O, those are the names used by this chapter. Thus, when referring to the I/O classes, we will simply use their character-based names rather than their internal, template names. For instance, this chapter will use the name **ios** rather than **basic_ios**, **istream** rather than **basic_istream**, and **fstream** rather than **basic_fstream**. Remember, parallel classes exist for wide-character streams and they work in the same way as those described here.

The I/O Headers

The Standard C++ I/O system relies upon several headers. They are shown here.

Header	For
<fstream>	File I/O
<iomanip>	Parameterized I/O manipulators
<ios>	Basic I/O support
<iosfwd>	Forward declarations used by the I/O system
<iostream>	General I/O
<istream>	Basic input support
<ostream>	Basic output support

Header	For
<sstream>	String-based streams
<streambuf>	Low-level I/O support

Several of these headers are used internally by the I/O system. In general, your program will only include **<iostream>**, **<fstream>**, **<sstream>**, or **<iomanip>**.

The Format Flags and I/O Manipulators

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios_base** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

These values are used to set or clear the format flags, using functions such as **setf( )** and **unsetf( )**. A detailed description of these flags is found in Chapter 20.

In addition to setting or clearing the format flags directly, you may alter the format parameters of a stream through the use of special functions called manipulators, which can be included in an I/O expression. The standard manipulators are shown in the following table:

Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolapha** flag.	Input/Output
dec	Turns on **dec** flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on **fixed** flag.	Output
flush	Flush a stream.	Output
hex	Turns on **hex** flag.	Input/Output

Manipulator	Purpose	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
noboolalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output
noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

To use a manipulator that takes a parameter, you must include **<iomanip>**.

Several Data Types

In addition to the **fmtflags** type just described, the Standard C++ I/O system defines several other types.

The streamsize and streamoff Types

An object of type **streamsize** is capable of holding the largest number of bytes that will be transferred in any one I/O operation. It is typically some form of integer. An object of type **streamoff** is capable of holding a value that indicates an offset position within a stream. It is typically some form of integer. These types are defined in the header **<ios>**, which is automatically included by the I/O system.

The streampos and wstreampos Types

An object of type **streampos** is capable of holding a value that represents a position within a **char** stream. The **wstreampos** type is capable of holding a value that represents a position with a **wchar_t** stream. These are defined in **<iosfwd>**, which is automatically included by the I/O system.

The pos_type and off_type Types

The types **pos_type** and **off_type** create objects (typically integers) that are capable of holding a value that represents the position and an offset, respectively, within a stream. These types are defined by **ios** (and other classes) and are essentially the same as **streamoff** and **streampos** (or their wide-character equivalents).

The openmode Type

The type **openmode** is defined by **ios_base** and describes how a file will be opened. It will be one or more of these values.

app	Append to end of file.
ate	Seek to end of file on creation.
binary	Open file for binary operations.
in	Open file for input.
out	Open file for output.
trunc	Erase previously existing file.

You can combine two or more of these values by ORing them together.

The iostate Type

The current status of an I/O stream is described by an object of type **iostate**, which is an enumeration defined by **ios_base** that includes these members.

Name	Meaning
goodbit	No errors occurred.
eofbit	End-of-file is encountered.
failbit	A nonfatal I/O error has occurred.
badbit	A fatal I/O error has occurred.

The seekdir Type

The **seekdir** type describes how a random-access file operation will take place. It is defined within **ios_base**. Its valid values are shown here.

beg	Beginning-of-file
cur	Current location
end	End-of-file

The failure Class

In **ios_base** is defined the exception type **failure**. It serves as a base class for the types of exceptions that can be thrown by the I/O system. It inherits **exception** (the standard exception class). The **failure** class has the following constructor:

```
explicit failure(const string &str);
```

Here, *str* is a message that describes the error. This message can be obtained from a **failure** object by calling its **what( )** function, shown here:

```
virtual const char *what( ) const throw( );
```

Overload << and >> Operators

The following classes overload the << and/or >> operators relative to all of the built-in data types.

basic_istream
basic_ostream
basic_iostream

Any classes derived from these classes inherit these operators.

The General-Purpose I/O Functions

The remainder of this chapter describes the general-purpose I/O functions supplied by Standard C++. As explained, the Standard C++ I/O system is built upon an intricate hierarchy of template classes. Many of the members of the low-level classes are not used for application programming. Thus, they are not described here.

bad

```
#include <icstream>
bool bad() const;
```

The **bad( )** function is a member of **ios.**

The **bad( )** function returns **true** if a fatal I/O error has occurred in the associated stream; otherwise, **false** is returned.

A related function is **good( )**.

clear

```
#include <iostream>
void clear(iostate flags = goodbit);
```

The **clear( )** function is a member of **ios.**

The **clear( )** function clears the status flags associated with a stream. If *flags* is **goodbit** (as it is by default), then all error flags are cleared (reset to zero). Otherwise, the status flags will be set to whatever value is specified in *flags*.

A related function is **rdstate( )**.

eof

```
#include <iostream>
bool eof() const;
```

The **eof( )** function is a member of **ios.**

The **eof( )** function returns **true** when the end of the associated input file has been encountered; otherwise it returns **false**.

Related functions are **bad( )**, **fail( )**, **good( )**, **rdstate( )**, and **clear( )**.

exceptions

```
#include <iostream>
iostate exceptions() const;
void exceptions(iostate flags);
```

The **exceptions( )** function is a member of **ios.**

The first form returns an **iostate** object that indicates which flags cause an exception. The second form sets these values.

A related function is **rdstate( )**.

fail

```
#include <iostream>
bool fail() const;
```

The **fail( )** function is a member of **ios.**

The **fail( )** function returns **true** if an I/O error has occurred in the associated stream. Otherwise, it returns **false**

Related functions are **good( )**, **eof( )**, **bad( )**, **clear( )**, and **rdstate( )**.

fill

```
#include <iostream>
char fill() const;
char fill(char ch);
```

The **fill( )** function is a member of **ios.**

By default, when a field needs to be filled, it is filled with spaces. However, you can specify the fill character using the **fill( )** function and specifying the new fill character in *ch*. The old fill character is returned.

To obtain the current fill character, use the first form of **fill( )**, which returns the current fill character.

Related functions are **precision( )** and **width( )**.

flags

```
#include <iostream>
fmtflags flags() const;
fmtflags flags(fmtflags f);
```

The **flags( )** function is a member of **ios** (inherited from **ios_base**).

The first form of **flags( )** simply returns the current format flags settings of the associated stream.

The second form of **flags( )** sets all format flags associated with a stream as specified by *f*. When you use this version, the bit pattern found in *f* is copied into the format flags associated with the stream. This version also returns the previous settings.

Related functions are **unsetf( )** and **setf( )**.

flush

```
#include <iostream>
ostream &flush();
```

The **flush( )** function is a member of **ostream**.

The **flush( )** function causes the buffer connected to the associated output stream to be physically written to the device. The function returns a reference to its associated stream.

Related functions are **put( )** and **write( )**.

fstream, ifstream, and ofstream

```
#include <fstream>
fstream();
explicit fstream(const char *filename,
                 ios::openmode mode = ios::in | ios::out);
ifstream();
explicit ifstream(const char *filename, ios::openmode mode=ios::in);

ofstream();
explicit ofstream(const char *filename,
                  ios::openmode mode=ios::out);
```

The **fstream( )**, **ifstream( )**, and **ofstream( )** functions are the constructors of the **fstream**, **ifstream**, and **ofstream** classes, respectively.

The versions of **fstream( )**, **ifstream( )**, and **ofstream( )** that take no parameters create a stream that is not associated with any file. This stream can then be linked to a file using **open( )**.

The versions of **fstream( )**, **ifstream( )**, and **ofstream( )** that take a filename for their first parameters are the most commonly used in application programs. Although it is entirely proper to open a file using the **open( )** function, most of the time you will not do so because these **ifstream**, **ofstream**, and **fstream** constructors automatically open the file when the stream is created. The constructors have the same parameters and defaults as the **open( )** function. (See **open** for details.) For instance, this is the most common way you will see a file opened:

```
ifstream mystream("myfile");
```

If for some reason the file cannot be opened, the value of the associated stream variable will be **false**. Therefore, whether you use a constructor to open the file or an explicit call to **open( )**, you will want to confirm that the file has actually been opened by testing the value of the stream.

Related functions are **close( )** and **open( )**.

gcount

```
#include <iostream>
streamsize gcount( ) const;
```

The **gcount( )** function is a member of **istream**.

The **gcount( )** function returns the number of characters read by the last input operation.

Related functions are **get( )**, **getline( )**, and **read( )**.

get

```
#include <iostream>
int get( );
istream &get(char &ch);
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
istream &get(streambuf &buf);
istream &get(streambuf &buf, char delim);
```

The **get( )** function is a member of **istream**.

In general, **get( )** reads characters from an input stream. The parameterless form of **get( )** reads a single character from the associated stream and returns that value.

get(char &*ch*) reads a character from the associated stream and puts that value in *ch*. It returns a reference to the stream.

get(char **buf*, streamsize *num*) reads characters into the array pointed to by *buf* until either *num*–1 characters have been read, a newline is found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **get( )**. If the newline character is encountered in the input stream, it is *not* extracted. Instead, it remains in the stream until the next input operation. This function returns a reference to the stream.

get(char **buf*, streamsize *num*, char *delim*) reads characters into the array pointed to by *buf* until either *num*–1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **get( )**. If the delimiter character is encountered in the input stream, it is *not* extracted. Instead, it remains in the stream until the next input operation. This function returns a reference to the stream.

get(streambuf &*buf*) reads characters from the input stream into the **streambuf** object. Characters are read until a newline is found or the end of the file is encountered. It returns a reference to the stream. If the newline character is encountered in the input stream, it is not extracted.

get(streambuf &*buf*, char *delim*) reads characters from the input stream into the **streambuf** object. Characters are read until the character specified by *delim* is found or the end of the file is encountered. It returns a reference to the stream. If the delimiter character is encountered in the input stream, it is not extracted.

Related functions are **put( )**, **read( )**, and **getline( )**.

getline

```
#include <iostream>
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

The **getline( )** function is a member of **istream**.

getline(char **buf*, streamsize *num*) reads characters into the array pointed to by *buf* until either *num*–1 characters have been read, a newline character has been found, or the end of the file has been encountered. The array pointed to by *buf* will be null terminated by **getline( )**. If the newline character is encountered in the input stream, it is extracted but is not put into *buf*. This function returns a reference to the stream.

getline(char **buf*, streamsize *num*, char *delim*) reads characters into the array pointed to by *buf* until either *num*–1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered. The array pointed

to by *buf* will be null terminated by **getline( )**. If the delimiter character is encountered in the input stream, it is extracted but is not put into *buf*. This function returns a reference to the stream.

Related functions are **get( )** and **read( )**.

good

```
#include <iostream>
bool good() const;
```

The **good( )** function is a member of **ios**.

The **good( )** function returns **true** if no I/O errors have occurred in the associated stream; otherwise, it returns **false**.

Related functions are **bad( )**, **fail( )**, **eof( )**, **clear( )**, and **rdstate( )**.

ignore

```
#include <iostream>
istream &ignore(streamsize num = 1, int delim = EOF);
```

The **ignore( )** function is a member of **istream**.

You can use the **ignore( )** member function to read and discard characters from the input stream. It reads and discards characters until either *num* characters have been ignored (1 by default) or until the character specified by *delim* is encountered (**EOF** by default). If the delimiting character is encountered, it is removed from the input stream. The function returns a reference to the stream.

Related functions are **get( )** and **getline( )**.

open

```
#include <fstream>
void fstream::open(const char *filename,
                   ios::openmode mode = ios::in | ios:: out);
void ifstream::open(const char *filename,
                   ios::openmode mode = ios::in);
void ofstream::open(const char *filename,
                   ios::openmode mode = ios:: out | ios::trunc);
```

The **open( )** function is a member of **fstream**, **ifstream**, and **ofstream**.

A file is associated with a stream by using the **open( )** function. Here, *filename* is the name of the file, which may include a path specifier. The value of *mode* determines how the file is opened. It must be one (or more) of these values:

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

You can combine two or more of these values by ORing them together.

Including **ios::app** causes all output to that file to be appended to the end. This value can only be used with files capable of output. Including **ios::ate** causes a seek to the end of the file to occur when the file is opened. Although **ios::ate** causes a seek to the end-of-file, I/O operations can still occur anywhere within the file.

The **ios::binary** value causes the file to be opened for binary I/O operations. By default, files are opened in text mode.

The **ios::in** value specifies that the file is capable of input. The **ios::out** value specifies that the file is capable of output. However, creating an **ifstream** stream implies input, and creating an **ofstream** stream implies output, and opening a file using **fstream** implies both input and output.

The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length.

In all cases, if **open( )** fails, the stream will be **false**. Therefore, before using a file, you should test to make sure that the open operation succeeded.

Related functions are **close( )**, **fstream( )**, **ifstream( )**, and **ofstream( )**.

peek

```
#include <iostream>
int peek();
```

The **peek( )** function is a member of **istream**.

The **peek( )** function returns the next character in the stream or **EOF** if the end of the file is encountered. It does not, under any circumstances, remove the character from the stream.

A related function is **get( )**.

precision

```
#include <iostream>
streamsize precision() const;
streamsize precision(streamsize p);
```

The **precision( )** function is a member of **ios** (inherited from **ios_base**).

By default, six digits of precision are displayed when floating-point values are output. However, using the second form of **precision( )**, you can set this number to the value specified in *p*. The original value is returned.

The first version of **precision( )** returns the current value.

Related functions are **width( )** and **fill( )**.

put

```
#include <iostream>
ostream &put(char ch);
```

The **put( )** function is a member of **ostream**.

The **put( )** function writes *ch* to the associated output stream. It returns a reference to the stream.

Related functions are **write( )** and **get( )**.

putback

```
#include <iostream>
istream &putback(char ch);
```

The **putback( )** function is a member of **istream**.

The **putback( )** function returns *ch* to the associated input stream.

A related function is **peek( )**.

rdstate

```
#include <iostream>
iostate rdstate() const;
```

The **rdstate( )** function is a member of **ios**.

The **rdstate( )** function returns the status of the associated stream. The C++ I/O system maintains status information about the outcome of each I/O operation relative to each active stream. The current state of a stream is held in an object of type **iostate**, in which the following flags are defined:

Name	Meaning
goodbit	No errors occurred.
eofbit	End-of-file is encountered.
failbit	A nonfatal I/O error has occurred.
badbit	A fatal I/O error has occurred.

These flags are enumerated inside **ios** (via **ios_base**).

rdstate( ) returns **goodbit** when no error has occurred; otherwise, an error bit has been set.

Related functions are **eof( )**, **good( )**, **bad( )**, **clear( )**, **setstate( )**, and **fail( )**.

read

```
#include <iostream>
istream &read(char *buf, streamsize num);
```

The **read( )** function is a member of **istream**.

The **read( )** function reads *num* bytes from the associated input stream and puts them in the buffer pointed to by *buf*. If the end of the file is reached before *num* characters have been read, **read( )** simply stops, sets **failbit**, and the buffer contains as many characters as were available. (See **gcount( )**.) **read( )** returns a reference to the stream.

Related functions are **gcount( )**, **readsome( )**, **get( )**, **getline( )**, and **write( )**.

readsome

```
#include <iostream>
streamsize readsome(char *buf, streamsize num);
```

The **readsome( )** function is a member of **istream**.

The **readsome( )** function reads *num* bytes from the associated input stream and puts them in the buffer pointed to by *buf*. If the stream contains less than *num*

characters, that number of characters are read. **readsome( )** returns the number of characters read. The difference between **read( )** and **readsome( )** is that **readsome( )** does not set the **failbit** if there are less than *num* characters available.

Related functions are **gcount( )**, **read( )**, and **write( )**.

seekg and seekp

```
#include <iostream>
istream &seekg(off_type offset, ios::seekdir origin)
istream &seekg(pos_type position);

ostream &seekp(off_type offset, ios::seekdir origin);
ostream &seekp(pos_type position);
```

The **seekg( )** function is a member of **istream**, and the **seekp( )** function is a member of **ostream**.

In C++'s I/O system, you perform random access using the **seekg( )** and **seekp( )** functions. To this end, the C++ I/O system manages two pointers associated with a file. One is the *get pointer*, which specifies where in the file the next input operation will occur. The other is the *put pointer*, which specifies where in the file the next output operation will occur. Each time an input or an output operation takes place, the appropriate pointer is automatically sequentially advanced. However, using the **seekg( )** and **seekp( )** functions, it is possible to access the file in a nonsequential fashion.

The two-parameter version of **seekg( )** moves the get pointer *offset* number of bytes from the location specified by *origin*. The two-parameter version of **seekp( )** moves the put pointer *offset* number of bytes from the location specified by *origin*. The *offset* parameter is of type **off_type**, which is capable of containing the largest valid value that *offset* can have.

The *origin* parameter is of type **seekdir** and is an enumeration that has these values:

ios::beg	Seek from beginning
ios::cur	Seek from current position
ios::end	Seek from end

The single-parameter versions of **seekg( )** and **seekp( )** move the file pointers to the location specified by *position*. This value must have been previously obtained using a call to either **tellg( )** or **tellp( )**, respectively. **pos_type** is a type that is capable of containing the largest valid value that *position* can have. These functions return a reference to the associated stream.

Related functions are **tellg( )** and **tellp( )**.

setf

```
#include <iostream>
fmtflags setf(fmtflags flags);
fmtflags setf(fmtflags flags1, fmtflags flags2);
```

The **setf( )** function is a member of **ios** (inherited from **ios_base**).

The **setf( )** function sets the format flags associated with a stream. See the discussion of format flags earlier in this section.

The first version of **setf( )** turns on the format flags specified by *flags*. (All other flags are unaffected.) For example, to turn on the **showpos** flag for **cout**, you can use this statement:

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

When you want to set more than one flag, you can OR together the values of the flags you want set.

It is important to understand that a call to **setf( )** is done relative to a specific stream. There is no concept of calling **setf( )** by itself. Put differently, there is no concept in C++ of global format status. Each stream maintains its own format status information individually.

The second version of **setf( )** affects only the flags that are set in *flags2*. The corresponding flags are first reset and then set according to the flags specified by *flags1*. Even if *flags1* contains other set flags, only those specified by *flags2* will be affected.

Both versions of **setf( )** return the previous settings of the format flags associated with the stream.

Related functions are **unsetf( )** and **flags( )**.

setstate

```
#include <iostream>
void setstate(iostate flags) const;
```

The **setstate( )** function is a member of **ios**.

The **setstate( )** function sets the status of the associated stream as described by *flags*. See **rdstate( )** for further details.

Related functions are **clear( )** and **rdstate( )**.

str

```
#include <sstream>
string str() const;
void str(string &s);
```

The **str( )** function is a member of **stringstream**, **istringstream**, and **ostringstream**.

The first form of the **str( )** function returns a **string** object that contains the current contents of the string-based stream.

The second form frees the string currently contained in the string stream and substitutes the string referred to by *s*.

Related functions are **get( )** and **put( )**.

stringstream, istringstream, ostringstream

```
#include <sstream>
explicit stringstream(ios::openmode mode = ios::in | ios::out);
explicit stringstream(const string &str,
                      ios::openmode mode = ios::in | ios::out);
explicit istringstream(ios::openmode mode=ios::in);
explicit istringstream(const string str, ios::openmode mode=ios::in);
explicit ostringstream(ios::openmode mode=ios::out);
explicit ostringstream(const string str, ios::openmode
                      mode=ios::out);
```

The **stringstream( )**, **istringstream( )**, and **ostringstream( )** functions are the constructors of the **stringstream**, **istringstream**, and **ostringstream** classes, respectively. These construct streams that are tied to strings.

The versions of **stringstream( )**, **istringstream( )**, and **ostringstream( )** that specify only the **openmode** parameter create empty streams. The versions that take a **string** parameter initialize the string stream.

Here is an example that demonstrates the use of a string stream.

```
// Demonstrate string streams.
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main()
{
  stringstream s("This is initial string.");

  // get string
  string str = s.str();
  cout << str << endl;

  // output to string stream
  s << "Numbers: " << 10 << " " << 123.2;

  int i;
  double d;
  s >> str >> i >> d;
  cout << str << " " << i << " " << d;

  return 0;
}
```

The output produced by this program is shown here:

```
This is initial string.
Numbers: 10 123.2
```

A related function is **str( )**.

sync_with_stdio

```
#include <iostream>
bool sync_with_stdio(bool sync = true );
```

The **sync_with_stdio( )** function is a member of **ios** (inherited from **ios_base**).

Calling **sync_with_stdio( )** allows the standard C-like I/O system to be safely used concurrently with the C++ class-based I/O system. To turn off **stdio** synchronization, pass **false** to **sync_with_stdio( )**. The previous setting is returned: **true** for synchronized; **false** for no synchronization. By default, the standard streams are synchronized. This function is reliable only if called prior to any other I/O operations.

tellg and tellp

```
#include <iostream>
pos_type tellg();
pos_type tellp();
```

The **tellg( )** function is a member of **istream**, and **tellp( )** is a member of **ostream**.

The C++ I/O system manages two pointers associated with a file. One is the *get pointer*, which specifies where in the file the next input operation will occur. The other is the *put pointer*, which specifies where in the file the next output operation will occur. Each time an input or an output operation takes place, the appropriate pointer is automatically sequentially advanced. You can determine the current position of the get pointer using **tellg( )** and of the put pointer using **tellp( )**.

pos_type is a type that is capable of holding the largest value that either function can return.

The values returned by **tellg( )** and **tellp( )** can be used as parameters to **seekg( )** and **seekp( )**, respectively.

Related functions are **seekg( )** and **seekp( )**.

unsetf

```
#include <iostream>
void unsetf(fmtflags flags);
```

The **unsetf( )** function is a member of **ios** (inherited from **ios_base**).

The **unsetf( )** function is used to clear one or more format flags.

The flags specified by *flags* are cleared. (All other flags are unaffected.)

Related functions are **setf( )** and **flags( )**.

width

```
#include <iostream>
streamsize width() const;
streamsize width(streamsize w);
```

The **width( )** function is a member of **ios** (inherited from **ios_base**).

To obtain the current field width, use the first form of **width( )**. It returns the current field width. To set the field width, use the second form. Here, *w* becomes the field width, and the previous field width is returned.

Related functions are **precision( )** and **fill( )**.

write

```
#include <iostream>
ostream &write(const char *buf, streamsize num);
```

The **write( )** function is a member of **ostream**.

The **write( )** function writes *num* bytes to the associated output stream from the buffer pointed to by *buf*. It returns a reference to the stream.

Related functions are **read( )** and **put( )**.