**Chapter 12: Stream and Files**

Table of Contents

[Stream Class 3](#_Toc49595845)

[The Stream Class Hierarchy 3](#_Toc49595846)

[The ios Class 5](#_Toc49595847)

[Formatting Flags 5](#_Toc49595848)

[Manipulators 8](#_Toc49595849)

[ios Functions 9](#_Toc49595850)

[The istream Class 11](#_Toc49595851)

[The ostream Class 12](#_Toc49595852)

[Predefined Stream Objects 13](#_Toc49595853)

[Stream Errors 14](#_Toc49595854)

[Error-Status Bits 14](#_Toc49595855)

[Inputting Numbers 15](#_Toc49595856)

[Too Many Characters 16](#_Toc49595857)

[No-Input Input 16](#_Toc49595858)

[Inputting Strings and Characters 17](#_Toc49595859)

[Disk File I/O with Streams 18](#_Toc49595860)

[Formatted File I/O 18](#_Toc49595861)

[Writing Data 19](#_Toc49595862)

[Reading Data 20](#_Toc49595863)

[Strings with Embedded Blanks 20](#_Toc49595864)

[Detecting End-of-File 21](#_Toc49595865)

[Character I/O 22](#_Toc49595866)

[Binary I/O 23](#_Toc49595867)

[Closing Files 24](#_Toc49595868)

[Writing and Reading Objects 24](#_Toc49595869)

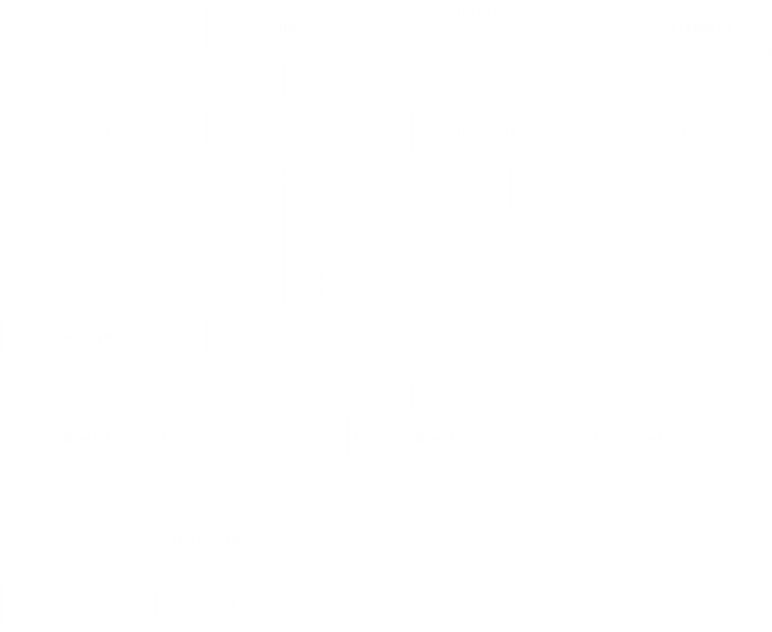
[The Mode Bits 26](#_Toc49595870)

## Stream Class

A stream is a general name given to a flow of data. Everything related to the flow of data, how the data will be taken as input, how it will be processed, how it will be outputted is all managed by this stream class. Note that there are multiple kinds of stream classes, each dealing with different kinds of data flow. We have already used the cin and cout objects of the stream class. They are part of the iostream header file and in the standard namespace (and are hidden from other namespaces).

The advantages of using stream class objects instead of C style file functions is that they are simpler to use since they do not require format specifiers, the ability to overload and use the insertion (<<) and extraction (>>) operators allow user created classes to work the same way as built-in types and they make the process of writing data to files or formatting data in memory very easy.

### The Stream Class Hierarchy



Above, we can see the hierarchy of the most important stream classes. We have already used the >> operator, which is a member of the istream class and the << operator, which is a member of the ostream class. Both of these classes in turn are derived from the ios class. The cout object is a predefined object of the ostream class, and the cin object is a predefined object of the istream class.

The classes used for input and output to the video display and keyboard respectively are part of the iostream header file. The classes used specifically for disk file I/O are part of the fstream header file.

The ios class is the base class for the hierarchy. It contains many constants and member functions common to input and output operations of all kinds, including formatting flags. It also contains a pointer to the streambuff class which contains the actual memory buffer into which data is read or written, and low-level routines for handling this data. We do not normally need to worry about the streambuff class, since it is referenced automatically by other classes.

The istream and ostream classes are derived from the ios class and are dedicated to input and output respectively, using functions such as get(), getline(), read(), put(), write() and the overloaded << and >> operators.

The iostream class is derived from the istream and ostream classes. It allows any classes that derived from it to use devices like disk files, which can be opened for both input and output at the same time. The istream\_withassign, ostream\_withassign and iostream\_withassign classes are inherited from the istream, ostream and iostream classes respectively, and add assignment operators to the classes. They have been removed from later versions of C++.

### The ios Class

The ios class is the base of the stream class hierarchy and contains the major features needed to operate C++ streams. The three most important features are formatting flags, error-status flags and the file operation mode, which will be discussed a little later, under the topic of disk files.

### Formatting Flags

Formatting flags are a set of enum definitions in ios. They act like switches that specify choices for various aspects of input and output format and operation.

|  |  |
| --- | --- |
| **Flag** | **Meaning** |
| skipws | Skip whitespaces on input. |
| left | Left-align |
| right | Right-align |
| internal | Use padding between sign or base indicator and number |
| dec | Convert to decimal |
| oct | Convert to octal |
| hex | Convert to hexadecimal |
| boolalpha | Convert bool to ‘true’ or ‘false’ strings |
| showbase | Use base indicator on output |
| showpoint | Show decimal point on output |
| uppercase | Use uppercase X, E and hex output letters (default is lowercase) |
| showpos | Display + before positive integers |
| scientific | Use exponential format on floating point output |
| fixed | Use fixed format on floating-point output |
| unitbuf | Flush all streams after insertion |
| stdio | Flush stdout, stderror after insertion |

There are many ways to set formatting flags, and different ones are set in different ways. Since they are part of the ios class, they are usually preceded by the name ios along with the scope resolution operator. All the flags can be set using the setf() and unsetf() member functions of the ios class.

cout.setf(ios::left); *// left align output text*cout>>"This text is left-justified.";  
cout.unsetf(ios::left); *// return to default (right align)*

C++

Many formatting flags can also be set using manipulators, which we will discuss next.

We can check whether a certain flag is on or off using a bitwise & operation. Each flag has a separate binary value of 2n. For example, say the value for left is 0010. This would be its value if it were switched on. Thus, we can perform a bitwise & operation between the flag and this value to check its status. If it is on, we will get a non-zero result. If it is off, we will get a zero result. This is essentially what is done in the program below.

#include <bitset>  
#include <iostream>  
using namespace std;  
  
void showflags(); *// function to print state of flags*int main()  
{  
 showflags(); *// print default state* cout.setf(ios::oct | ios::showbase | ios::fixed); *// switch on flags* showflags(); *// print new state*}  
  
*// checks status of individual flag against default and prints*void printFlagStatus(ios::fmtflags check, ios::fmtflags flag, string flagName)  
{  
 cout<<bitset<16>(flag)<<" "; *// prints flags value  
 // checks if flag on or off* if(check & flag) cout<<flagName<<" on\n";  
 else cout<<flagName<<" off\n";  
}  
  
void showflags()  
{  
 ios::fmtflags f;  
 f = cout.flags(); *// get flag settings* cout<<bitset<16>(f)<<" fmtflag value\n"; *// 1s for flags on by default* printFlagStatus(f, ios::boolalpha, "boolalpha");  
 printFlagStatus(f, ios::dec, "dec");  
 printFlagStatus(f, ios::fixed, "fixed");  
 printFlagStatus(f, ios::hex, "hex");  
 printFlagStatus(f, ios::internal, "internal");  
 printFlagStatus(f, ios::left, "left");  
 printFlagStatus(f, ios::oct, "oct");  
 printFlagStatus(f, ios::right, "right");  
 printFlagStatus(f, ios::scientific, "scientific");  
 printFlagStatus(f, ios::showbase, "showbase");  
 printFlagStatus(f, ios::showpoint, "showpoint");  
 printFlagStatus(f, ios::showpos, "showpos");  
 printFlagStatus(f, ios::skipws, "skipws");  
 printFlagStatus(f, ios::unitbuf, "unitbuf");  
 printFlagStatus(f, ios::uppercase, "uppercase");  
 cout<<"\n";  
}

C++

We can use similar thinking to turn on a flag, by using the bitwise | (or) operator. This is done in the main function above.

### Manipulators

Manipulators are formatting instructions inserted directly into a stream. We have extensively used one such manipulator, endl, which sends a newline to the stream and flushes it. There are also manipulators that take arguments, such as setiosflags(), which is used to set flags from the ios class. These require the iomanip header file.

|  |  |  |
| --- | --- | --- |
| **Manipulator** | **Purpose** | **Argument** |
| ws | Turn on whitespace skipping on input | - |
| dec | Convert to decimal | - |
| oct | Convert to octal | - |
| hex | Convert to hexadecimal | - |
| endl | Insert newline and flush the output stream | - |
| ends | Insert null character and terminate an output string | - |
| flush | Flush the output stream | - |
| lock | Lock file handle | - |
| unlock | Unlock file handle | - |
| setw() | Set field width for output | field width (int) |
| setfill() | Set fill character for output (default is space) | fill characters (int) |
| setprecision() | Set precision (number of digits displayed) | precision (int) |
| setiosflags() | Set specified flags | formatting flags (long) |
| resetiosflags() | Clear specified flags | formatting flags (long) |

The manipulators are inserted directly into the stream. Note that they only affect the data that follows them in the stream, not the data that precedes them.

cout<<hex<<someVariable;

C++

### ios Functions

The ios class contains some functions that are used to set formatting flags and perform other tasks.

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| ch = fill(); | Returns the fill character |
| fill(ch); | Sets the fill character |
| p = precision(); | Get the precision |
| precision(p); | Set the precision |
| w = width(); | Get the current field width (in characters) |
| width(w); | Set the current field width |
| setf(flags); | Set specified formatting flags |
| unsetf(flags); | Unset specified formatting flags |
| setf(flags, field); | First clear field, then set flags |

These functions are used with specific stream objects using the normal dot operator.

cout.width(14); *// sets field width to 14 characters*cout.fill(\*); *// make the fill character \**

C++

The two-argument version of setf() uses the second argument to reset all the flags of a particular type, or field, and then set the flag specified in the first argument.

cout.setf(ios::left, ios::adjustfield);  
*// clears any flags dealing with text justification  
// and then sets text to left-justified*

C++

|  |  |
| --- | --- |
| **Flag** | **Field** |
| dec, oct, hex | basefield |
| left, right, internal | adjustfield |
| scientific, fixed | floatfield |

### The istream Class

The istream class performs input-specific activities. Some functions commonly used from the istream class are:

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| >> | Formatted extraction for all basic (and overloaded) types |
| get(ch); | Extract one character into ch |
| get(str) | Extract characters into array str until ‘\n’ is detected |
| get(str, **MAX**) | Extract up to **MAX** characters into array |
| get(str, **DELIM**) | Extract characters into array str until specified delimiter (‘\n’ by default). Leave delimiting character in stream. |
| get(str, **MAX**, **DELIM**) | Extract characters into array str until **MAX** characters or the **DELIM** character. Leave delimiting character in stream. |
| getline(str, **MAX**, **DELIM**) | Extract characters into array str until **MAX** characters or the **DELIM** character. Extract delimiting character. |
| putback(ch) | Insert last character read back into input stream |
| ignore(**MAX**, **DELIM**) | Extract and discard up to **MAX** characters until and including the specified delimiter (‘\n’ by default) |
| peek(ch) | Read one character. Leave it in stream. |
| count = gcount() | Return number of characters read by a (immediately preceding) call to get(), getline() or read(). |
| read(str, **MAX**) | For files, extract up to **MAX** characters into str, until EOF. |
| seekg() | Set distance (in bytes) of file pointer from start of file. |
| seekg(pos, seek\_dir) | Set distance (in bytes) of file pointer from specified place in file. seek\_dir can be ios::beg, ios::cur, ios::end. |
| pos = tellg(pos) | Return position (in bytes) of file pointer from start of file. |

Most of these functions deal with the cin object, but the last four deal with disk files.

### The ostream Class

The ostream class handles output activities. Some of the most commonly used functions of this class are:

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| << | Formatted insertion for all basic (and overloaded) types. |
| put(ch) | Insert character ch into stream. |
| flush() | Flush buffer contents and insert newline. |
| write(str, **SIZE**) | Insert **SIZE** characters from array str into file. |
| seekp(position) | Set distance in bytes of file pointer from start of file. |
| seekp(position, seek\_dir) | Set distance in bytes of file pointer, from specified place in file. |
| pos = tellp() | Return position of file pointer, in bytes. |

The last four functions in the table deal with disk files.

### Predefined Stream Objects

Standard streams are pre-connected input and output channels between a computer program and its environment, generally the text terminal, when it begins execution. In C, there are three I/O connections, standard input (stdin), standard output (stdout) and standard error (stderr). In C++, there are four I/O connections, standard input (cin), standard output (cout), standard error (cerr) and the buffered version of standard error (clog). The standard input uses the keyboard as the default device, while the other three use the screen as the default device.

## Stream Errors

The process for input and output we have used so far was very straightforward in that we assumed nothing would go wrong during the I/O process. Stream errors are not usually a problem with smaller programs, but they can be with more sophisticated ones, where we need input to be of a specific format. Stream errors may also occur due to hardware failure.

### Error-Status Bits

The stream error-status flags have an ios enum member that reports errors that occur during input and output operations.

Unused

eofbit 0x01

failbit 0x02

badbit 0x04

hardfail 0x08

|  |  |
| --- | --- |
| **Name** | **Meaning** |
| goodbit | No errors |
| eofbit | End of file reached |
| failbit | Operation failed (due to user error or premature EOF |
| badbit | Invalid operation |
| hardfail | Unrecoverable error |

These flags can be read (and even set) with various ios functions.

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| int = eof(); | Returns true if EOF flag set |
| int = fail(); | Returns true if failbit or badbit or hardfail flag set |
| int = bad(); | Returns true if badbit or hardbit flag set |
| int = good(); | Returns true if everything is alright and no flags are set |
| clear(int = 0); | With no argument, clears all error bits; otherwise sets specific flags (e.g. clear(ios::failbit)) |

### Inputting Numbers

while(true) *// cycle until input OK*{  
 cout << "\nEnter an integer: ";  
 cin >> i;  
 if(cin.good() ) *// if no errors* {  
 cin.ignore(10, '\n'); *// remove newline* break; *// exit loop* }  
 cin.clear(); *// clear the error bits* cout << "Incorrect input";  
 cin.ignore(10, '\n'); *// remove newline*}  
cout << "integer is " << i; *// error-free integer*

C++

This code commonly detects errors such as non-integer inputs but it can also detect system-related failures more commonly seen with disk files. Other data types can be analysed in the same way.

### Too Many Characters

Some errors may cause extra characters to be left in the input stream after the input is supposedly complete. This causes them to be passed along to the next input operation.

cin.ignore(10, '\n');

C++

The above line will throw away up to 10 characters including the specified character.

### No-Input Input

There may be situations where the user gives no input, i.e. presses the Enter key or Space key where an input is required, causing the program to continue waiting for an input. This can cause the user to become confused if they do not know what to expect and may even cause problems with graphical interfaces. In these situations, it is important to tell the input stream not to ignore whitespace characters.

cout << "\nEnter an integer: ";  
cin.unsetf(ios::skipws); *// don’t ignore whitespace*cin >> i;  
if( cin.good() ) *// no error*{}  
*// error*

C++

### Inputting Strings and Characters

It is not really possible to cause errors when taking input from the user as strings since everything is taken in. However, with disk files, an EOF or something worse may occur. These need to be checked.

## Disk File I/O with Streams

Working with disk files requires the ifstream, fstream and ofstream classes. Objects of these classes can be associated with disk files and be used to read and write to files.

ifstream is derived from istream, fstream is derived from iostream and oftream is derived from ostream. The file-oriented classes are also derived from the fstreambase class. This contains an object of class filebuf, which is the file-oriented buffer. The three main classes are part of the fstream header file.

Functions like fread() and fwrite() from the C language still work in C++, but they are less well suited. The C++ approach is cleaner and easier. C and C++ functions should not be mixed since they may cause problems.

### Formatted File I/O

In formatted I/O, numbers are stored as a series of characters. Thus, 6.02 will literally be stored as ‘6’, ‘.’, ‘0’ and ‘2’. Characters and strings are stored normally. This can be inefficient for numbers with many digits but there are situations where it is appropriate and easy to implement.

### Writing Data

ofstream outfile("fdata.txt"); *//create ofstream object*outfile << str1 << ' ' << str2;

C++

The outfile object of the ofstream class was initialized to a file. If the file did not exist, it was created and opened. If it did exist, it is truncated and the new data replaced the old data. Alternatively, we could have created the object and opened the file separately.

ofstream outfile;  
outfile.open("fdata.txt");

C++

This is useful when opening the file has potential to fail.

Notice that the insertion operator is appropriately overloaded for us to use. When the program terminates, the outfile object’s destructor is automatically called which closes the file for us. Note that numbers will be stored as formatted text, i.e. characters.

### Reading Data

ifstream infile("fdata.txt"); *//create ifstream object  
//extract (read) data from it*infile >> str1 >> str2;

C++

Provided we formatted the data properly while writing the file, the infile object of the ifstream class should have no problem handling it. Numbers are converted back to their appropriate data type. They are not read as characters.

### Strings with Embedded Blanks

We may run into some problems if we try to store strings that have blanks in them, since the blanks are used to identify the separation of two different strings. In these cases, we will need to use a special delimiter characters after each string. The newline character is commonly used. Then, we can use the getline() function rather than the normal extraction operator to read the data. The function can take two arguments, the first being the variable in which to store the line and the second being the maximum length to read.

### Detecting End-of-File

EOF is a signal sent to the program by the OS when there is no more data to read. We can detect an EOF flag.

while (!inFile.eof()) *// until EOF encountered*

C++

However, doing this will mean we do not detect other error flags, which may occur, albeit rarely. It is better to check for all errors instead.

while (inFile.good())

C++

The stream can also be tested directly. Any stream objects have a value that can test for usual error conditions, which returns 0 if an error occurs.

while (inFile)

C++

### Character I/O

We can output and input single characters using the put() and get() functions of the ostream and istream classes respectively.

outfile.put(ch); *//write it to file*

infile.get(ch); *//read character*

C++

Another method is to use the rdbuf() function of the ios class, which returns a pointer to the streambuf object associated with the stream object. Since this object holds a buffer with the characters read from the stream, we can use the pointer as a data object in its own right.

cout << infile.rdbuf(); *//send its buffer to cout*

C++

### Binary I/O

It is more efficient than formatted I/O to use binary I/O when writing a large amount of numerical data to disk. In this way, numbers are stored as they are in the computers RAM rather than as characters. This means less space is wasted. The write() and read() functions from the ofstream and ifstream classes are used to deal with data in binary format, since they can be set to work with binary data using the ios::binary argument, which causes them to simply transfer a buffer full of bytes.

int buff[MAX]; *//buffer for integers*int main()  
{  
 for(int j=0; j<MAX; j++) buff[j] = j; *//(0, 1, 2, ...)  
 //create output stream* ofstream os("edata.dat", ios::binary);  
 *//write to it* os.write( reinterpret\_cast<char\*>(buff), MAX\*sizeof(int) );  
 os.close(); *//must close it* for(j=0; j<MAX; j++) buff[j] = 0;  
 *//create input stream* ifstream is("edata.dat", ios::binary);  
 *//read from it* is.read( reinterpret\_cast<char\*>(buff), MAX\*sizeof(int) );  
 return 0;  
}

C++

Notice that the write() and read() functions took two arguments, the latter of which was the length in bytes of the data. The first argument is the address of the data buffer. This is cast using reinterpret\_cast to type char\*. This casting caused the functions to look at the data as though they were characters. It changes the type of a section of memory without caring whether it makes sense. This can be dangerous, but is sometimes necessary.

### Closing Files

We do not need to explicitly close a file when dealing with just one stream, since the destructor function automatically closes it for us. But when dealing with multiple streams, we may need to close files explicitly using the close() function. This is also a good practice to have in general.

### Writing and Reading Objects

Writing objects is generally done in the binary mode. This ensures that numerical data contained in objects is handled properly.

#include <fstream>  
#include <iostream>  
using namespace std;  
  
class person *//class of persons*{  
protected:  
 char name[80]; *//person’s name* short age; *//person’s age*public:  
 void getData() *//get person’s data* {  
 cout << "Enter name: ";  
 cin >> name;  
 cout << "Enter age: ";  
 cin >> age;  
 }  
 void showData() *//display person’s data* {  
 cout << "Name: " << name << endl;  
 cout << "Age: " << age << endl;  
 }  
};

int main()  
{  
 person pers; *//create a person* pers.getData(); *//get data for person  
 //create ofstream object* ofstream outfile("PERSON.DAT", ios::binary);  
 *//write to it* outfile.write(reinterpret\_cast<char\*>(&pers), sizeof(pers));  
 outfile.close();  
   
 ifstream infile("PERSON.DAT", ios::binary); *//create stream  
 //read stream* infile.read( reinterpret\_cast<char\*>(&pers), sizeof(pers) );  
 pers.showData(); *//display person* return 0;  
}

C++

To work correctly, programs that read and write objects to files must use the same class of objects. If two programs thought that certain members of the object being written/read had different lengths, then neither could accurately work with the file. The functions in the object may be different since they are not written to disk, but the data needs to be the same.

However, this is not true for derived classes. Derived classes have a number placed in front of the object’s data to help identify the object’s class when using virtual functions. If the functions change, this number changes too. Problems may also occur with objects that have pointer data members. Thus, it is best to make sure that objects are identical when reading or writing objects.

### The Mode Bits

We have seen the ios::binary argument already. This is called a mode bit since it defines how the file should be opened. There are several more mode bits that can be used with the open() function.

|  |  |
| --- | --- |
| **Mode Bit** | **Result** |
| in | Open to read (default for ifstream) |
| out | Open to write (default for ofstream) |
| ate | Start reading/writing at end of file |
| app | Start writing at end of file |
| trunc | Truncate file to zero length if it exists |
| nocreate | Error if file does not exist |
| noreplace | Error if file already exists, unless ate or app is set. |
| binary | Open in binary mode |