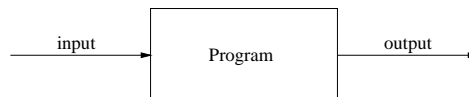


The C++ IOStreams Library

1 Introduction

The simplest¹ way to view a program is as a device for transforming input into output.



A program is useless unless we have a way of getting information into the program, and a way of getting it out again.

Input for a program can be from many sources eg:

- User Input.
Prompt the user for some information and have them enter it at the command line or in some text field of a GUI.
- The Command Line.
Read various options/arguments from the command line when the program is invoked.
- Environment Variables.
Most operating systems/shells allow the user to set "environment variables" eg

`MOZILLA_HOME=/usr/local/apps/netscape`

though the precise form varies from OS to OS and shell to shell. A program can gain access to these environment variables through platform dependent library functions.
- Files.
Read information from a configuration file, import a document for the user to edit, exhibit an image for a user to manipulate etc.
- Sensors.
Take readings directly from electronic sensors.
- Other Programs.
Take your input from the output of another program, request another program (still running) to send some data etc.

Any one program may use any or all of these sources. Once inside input can be broadly classified as either:

- Configuration information that affects the state of the program and how it behaves.
- Data to be processed, either automatically or interactively with a user.

¹All programs can be considered to be of this form if you sufficiently complicate your description of the input and output. However other models with, say, feedback loops would be a more comfortable fit to interactive GUI programs.

Output from a program can go to many places:

- The User.
Display some information on the command line or in a GUI message box etc. This sort of output is ephemeral unless the user takes some action to capture it.
- Files.
Store some information in a file. This may contain textual output (eg some program code) or binary output (eg a JPEG image).
- Actuators.
Send signals to some equipment that causes it to undertake some action (eg switching off a nuclear reactor).
- Other Programs.
Connect your output to the input of another program, send some data (whilst you are still active) to another program etc.

The C++ IOStreams Library, part of the C++ Standard Library, provides a uniform way of handling input from (and output to) any of the above sources (including hardware devices) except for command line arguments and environment variables. Such arguments are delivered to the program by the operating system via the `argv` parameter of the `main` function:

```
int main(int argc, char** argv)
{
    // argc contains the number of arguments
    // argv is an array of character arrays holding the actual arguments
    ...
}
```

The library consists of a set of class templates which can be specialised by the user to handle any sort of input data. They come with ready-made specialisations to handle simple character based input and output either to and from the console or to and from normal files.

These notes give an overview of the facilities of the IOStreams library. Another set of notes will discuss command line argument handling. These notes are moderately long for a set of notes but they are short in comparison to the facilities provided — the main reference I used whilst writing them, Josuttis [1], devotes a chapter of 101 pages to the topic!

2 Basic Usage

You will already have made much use of `cout` and `cin` streams for doing simple input and output to and from the console (command line). For example the following program (the code is in the file `examples/ex1.cc`) prompts the user to enter some values and then prints out the values entered.

```
#include <iostream>    // This might have to be iostream.h
#include <string>

int main(int argc, char** argv)
{
    int x;
    double g;
    char c;
    string str;
```

```

cout << "Enter an integer: ";
cin >> x;
cout << "Enter a real number: ";
cin >> g;
cout << "Enter a character: ";
cin >> c;
cout << "Enter a string: ";
cin >> str;

cout << "You entered: " << endl;
cout << "    the integer " << x << endl;
cout << "    the real number " << g << endl;
cout << "    the character '" << c << "'" << endl;
cout << "    the string \"" << str << "\"" << endl;
}

```

Note how the *insertion* operator << and the *extraction* operator >> are overloaded to handle many different types. The writer of a new class can also overload these operators so that objects of that class can be written and read in the same way as the builtin types.

Note however that these operators do not do *validation* — if the user enters something unexpected at the prompt you can end up with garbage values for the variable. Also the input stream is not flushed after a read operation. Thus if at the first prompt

Enter an integer:

the user in fact entered

0.34tykjvo hi there

the program would continue without waiting for user input:

```

Enter a real number: Enter a character: Enter a string: You entered:
    the integer 0
    the real number 0.34
    the character 't'
    the string "ykjvo"

```

Whilst if the user had entered

.90y78hi

you would get

```

Enter a real number: Enter a character: Enter a string: You entered:
    the integer -1073743840
    the real number 2.08408
    the character '
    the string ""

```

The behaviour of the extraction operator is to skip leading white space then read characters until a character not belonging to the string representation of the data type being input is encountered. Any remaining characters are left in the `cin` stream to be read by the next use of the extraction operator.

You can force the program to skip reading the rest of a line after an extraction operation by using the `ignore` member function of the `cin` stream object. This function has a number of forms but the one to use here is (this is not quite the real signature but it shows the intent more clearly)

```
ignore(int count, char delim)
```

The behaviour is to read and discard up to `count` characters until the character `delim` is read (this character is also discarded). Thus

```
cin.ignore(1000, '\n');
```

would discard up to a 1000 characters or until an end-of-line character was found (whichever came first).

You can use functions like this and others if you want to do validated input (that is you don't trust the user to enter correct data). But a better way to proceed is probably to write a GUI with data-entry fields which can be more easily checked for validity. Use extraction from `cin` for quick-and-dirty programs or for formatted input from some previously constructed file attached to the `cin` stream (see later).

The other aspect of the above examples is that when reading a string with the extraction operator you cannot include white space in the string. That is if you wanted to read the string "Hello, World!" you would have to read it in two bits (Hello, and World!) and then put it back together!

If you want to read strings with embedded spaces you should use the (global) function `getline`

```
void getline(istream& in, string str, char delim='\n');
```

provided by the string library. This function ignores leading spaces and then reads all characters until the line delimiter or end-of-file is reached. The line delimiter is extracted from the stream but not appended to the string. The default line delimiter character is the newline character `'\n'` but you can pass your own as an optional argument (in which case the newline character is not special and may be appended to the string being read in). Thus a better way to read the string in the above program would be

```
cin.ignore(1000, '\n'); // To flush any new line characters etc --
                        // see what happens if you don't use this function
cout << "Enter a string: ";
getline(cin, str);
```

3 Behind the Scenes

The IOStreams Library is based on the concept of a "stream". A stream can represent a file, the console, blocks of memory or hardware devices. The library provides a common set of interfaces (functions) for handling streams. The general picture is as in Figure 1.

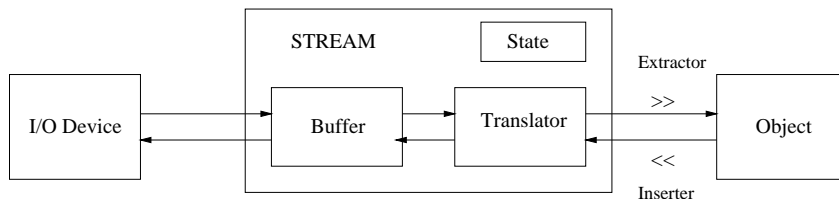


Figure 1: Stream Structure

The user of a stream object rarely has to be concerned with the buffer and translation components which handle conversion of individual bits from the I/O device into the structured data of an object (of either a built

in type such as an integer or a user defined type). However the user will need to keep a watch on the state component which notes whether or not an I/O operation has succeeded or failed.

The library is built using a fairly complicated set of class templates with the relationships depicted in Figure 2.

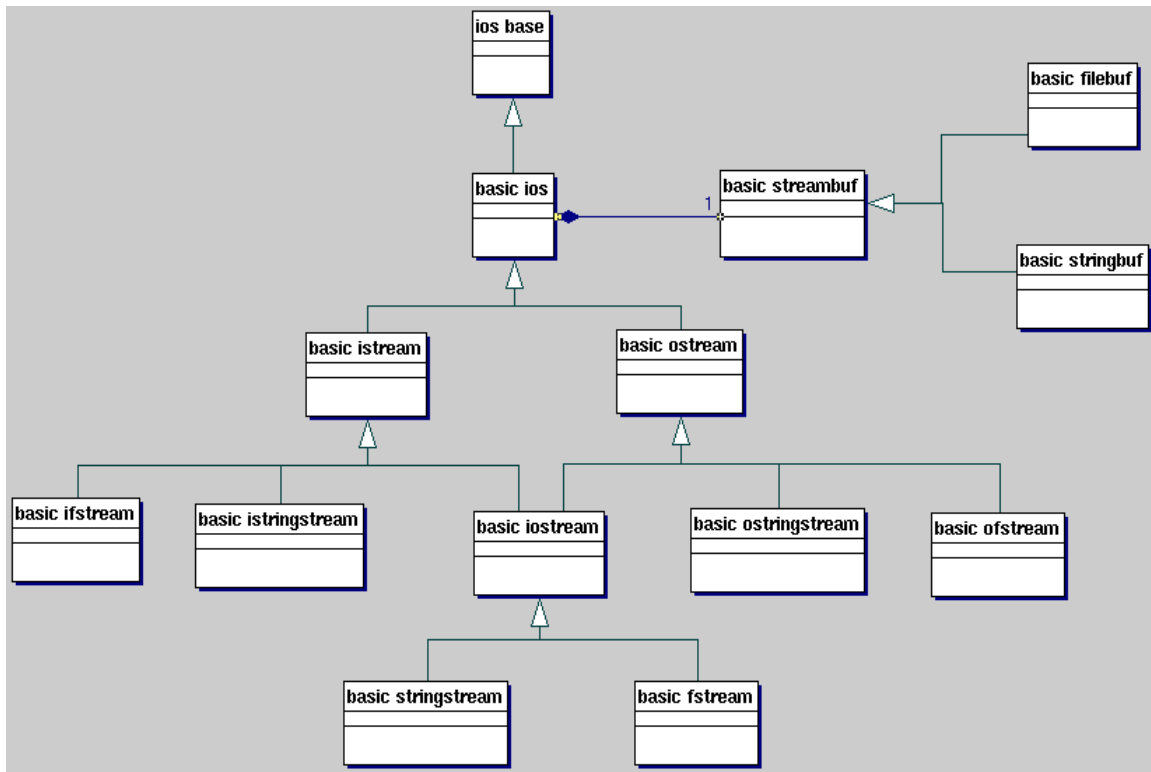


Figure 2: The Class Structure of the IOStream Library

Each of the `basic_xxxx` classes in the diagram is a class template, eg

```
template <class charT> class basic_ios { .... };
```

and the standard stream classes are obtained by a series of typedef's:

```
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;
typedef basic_istringstream<char> istringstream;
typedef basic_ostreamringstream<char> ostreamringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
```

The `iostream` header file provides the standard streams:

```
istream cin;
ostream cout;
ostream cerr;
ostream clog;
```

Note. The above description is based on the current standard. Not all compilers/libraries have caught up with the standard and some are still using an old-style `iostream` library which was not built according to the above scheme. However the new library (above) was designed so that the instantiated classes have the same interfaces as the old-style and work in the same way. The main exception is that old-style string streams are `istrstream`, `ostrstream` and `strstream` rather than `istringstream` etc. Currently the GNU C++ Standard library is in a state of transition. It still uses `strstream` etc and does not base the stream classes on `basic_XXXX` template classes but does provide most of the new style `iostream` functions and manipulators (as of 2.95, but 2.7 is further behind).

4 The State of Streams

Each stream maintains a state that identifies whether an I/O operation was successful or not, and, if not, the reason for the failure. The state of a stream is determined by the settings of a number of flags. These flags are constants of type `iosstate` (a member of the base class `ios_base`). The flags are:

- **goodbit:** If this is set all the other flags are cleared.
- **eofbit:** If this is set then end-of-file was encountered.
- **failbit:** If this is set then the operation was not processed correctly but the stream is generally OK. For example this flag would be set if an integer was to be read but the next character is a letter.
- **badbit:** This is set if the stream has somehow become corrupted or if data is lost.

The current state of the flags can be determined by calling various boolean returning member functions that all stream classes possess.

- `good()` – Returns true if the stream is OK (`goodbit` is set).
- `eof()` – Returns true if end-of-file was encountered (`eofbit` is set).
- `fail()` – Returns true if an error has occurred (`failbit` or `badbit` is set).
- `bad()` – Returns true if a fatal error has occurred (`badbit` is set).

Streams also possess the following more general flag manipulation functions.

- You can access all the flags using

```
ios::iosstate rdstate();
```

as in the following example (to be found in `examples/rdstateex.cc`).

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    int x;
```

```
    cout << "Enter an integer: ";
```

```
    cin >> x;
```

```
    // The state of the stream can be gotten with rdstate.
```

```
    ios::iosstate flags = cin.rdstate();
```

```
    // We can test for which bits are set as follows.
```

```
    // Note the use of the bitwise & operator.
```

```

if (flags & ios::failbit)
    cout << "failbit set." << endl;
else
    cout << "failbit not set." << endl;

if (flags & ios::badbit)
    cout << "badbit set." << endl;
else
    cout << "badbit not set." << endl;

if (flags & ios::eofbit)
    cout << "eofbit set." << endl;
else
    cout << "eofbit not set." << endl;

// It's usually easier to test the bits directly:
if (cin.good())
    cout << "Stream state is good." << endl;
else
    cout << "Stream state is not good." << endl;

if (cin.fail())
    cout << "Are you sure you entered an integer?" << endl;
else
    cout << "You entered: " << x << endl;
}

```

- You can clear all, or selected, flags by using

```
void clear(ios::iostate flags = ios::goodbit);
```

Thus `cin.clear()` will clear all flags whilst `cin.clear(ios::failbit)` will clear just the `failbit` flag.

5 Manipulating Your Stream

At the end of many output statements a manipulator is written:

```
cout << "You entered: " << x << endl;
```

The most important manipulators provided by the `IOStream` library are as follows.

- `endl` – used with a `ostream`, output a newline and flush the buffer.
- `ends` – used with a `ostream`, output a null character (`\0`).
- `flush` – used with a `ostream`, flushes the output buffer.
- `ws` – used with a `istream`, reads and discards whitespace.

6 Formatting Your Stream

When outputting values to the console or a file (in other words on a `istream`) you often want to format them in some way, eg print a floating point number in a field of width 5 with three decimal places etc.

The easiest way to do this is to use one of the many formatting manipulators and flags. For example the following program generates 10 random numbers between 0 and 1 (not including 1) and writes them out to a file one to a line with 3 decimal places, a leading zero and a decimal point. (The code can be found in `examples/rndnums.cc`.)

```
#include <iostream> // May need to be <iostream.h>
#include <iomanip>   // May need to be <iomanip.h>
#include <cstdlib>  // Required for rand() -- may need to be <stdlib.h>
#include <ctime>    // For access to time functions (may need to be <time.h>)

double generateRandNum()
{ int intrN = rand(); // Each time rand() is called it generates a randomly chosen
                      // integer between 0 and RAND_MAX

  // This is not the way to generate "good" random numbers between 0 and 1
  // but 'twill do for a quick and dirty hack.
  return double(intrN)/RAND_MAX;
}

int main()
{ double rn = 0.0;

  // Use system time to seed random number generator with different number each
  // time program is run.
  srand(time(NULL));

  // Standard says that fixed should be a manipulator so that we could write
  // cout << fixed;
  // But g++ does not currently support this.
  cout.setf(ios::fixed);
  cout << setprecision(3);

  for (int i=0; i<10; i++)
    { rn = generateRandNum();
      cout << rn << endl;
    }
}
```

The two lines that are doing the formatting in the above are:

```
cout.setf(ios::fixed);
cout << setprecision(3);
```

Here `ios::fixed` is a flag that says the fixed point notation should be used and `precision(val)` is a manipulator that sets `val` as the precision of floating point values. As it is a manipulator with argument you have to include `<iomanip>`.

Note how the function `setf` (a member function of all streams) can be used to set a flag. The function for clearing a flag is `unsetf`. Thus to turn off fixed point notation you would do:

```
cout.unsetf(ios::fixed);
```

There are manipulators and flags which control many different aspects of input and output of character values. The following sections and tables summarise the available manipulators, the flags they set, whether they are to be used for input or output and whether they are currently available with `g++`. These lists and explanations are not exhaustive (even if they seem exhausting).

6.1 Boolean Values

Manipulator	Affected Flag	Effect	Input/Output	g++ (2.95)
<code>boolalpha</code>	<code>ios::boolalpha</code>	This controls how <code>bool</code> variables are printed. If the flag is set then boolean values are printed as <code>true</code> or <code>false</code> . If the flag is not set then they print as 1 or 0. This manipulator sets the flag.	Input/Output	Neither
<code>noboolalpha</code>	<code>ios::boolalpha</code>	The <code>noboolalpha</code> manipulator unsets the <code>boolalpha</code> flag.	Input/Output	Neither

As you can see the GNU g++ IOSTream Library does not yet provide this feature. If you want to print `true` or `false` for the values of a boolean variable you can do something like:

```
cout << (b? "true" : "false") << endl;
```

6.2 Field Widths, Fill Characters and Output Adjustment

Manipulator	Affected Flag	Effect	Input/Output	g++ (2.95)
<code>setw(val)</code>	None	Sets the field width to <code>val</code> . This is a minimum value (if the output value is wider than the set width then the width setting is ignored). The setting only affects the next formatted output.	Input/Output	Manipulator
<code>setfill(c)</code>	None	Sets the character <code>c</code> to be the fill character.	Output	Manipulator
<code>left</code>	<code>ios::left</code>	The manipulator sets the flag. If the flag is set then the value is output left adjusted in the field.	Output	Flag
<code>right</code>	<code>ios::right</code>	The manipulator sets the flag. If the flag is set then the value is right-adjusted in the field.	Output	Flag
<code>internal</code>	<code>ios::internal</code>	The manipulator sets the flag. If the flag is set then the value is printed with the sign left-adjusted and the value right-adjusted.	Output	Flag

Note that after any formatted output operation the default field width is restored. The fill character and the adjustment setting remain unchanged until reset.

6.3 Positive Sign and Uppercase Letters

Manipulator	Affected Flag	Effect	Input/Output	g++ (2.95)
<code>showpos</code>	<code>ios::showpos</code>	The manipulator sets the <code>showpos</code> flag. If this flag is set then a positive sign (+) will be printed in front of any positive number (when using decimal notation).	Output	Flag
<code>noshowpos</code>	<code>ios::showpos</code>	This manipulator clears the <code>showpos</code> flag.	Output	Flag
<code>uppercase</code>	<code>ios::uppercase</code>	The manipulator sets the flag. If the flag is set then integers using hexadecimal notation, or floating point numbers using scientific notation, use uppercase for any letters appearing in them.	Output	Flag
<code>nouppercase</code>	<code>ios::uppercase</code>	The manipulator clears the flag.	Output	Flag

6.4 Number Base

Manipulator	Affected Flag	Effect	Input/Output	g++ (2.95)
<code>dec</code>	<code>ios::dec</code>	This flag sets the number base for integer number printing and reading to decimal.	Input/Output	Both
<code>hex</code>	<code>ios::hex</code>	Manipulator sets the <code>hex</code> flag. Which causes input and output of integers to use hexadecimal notation.	Input/Output	Both
<code>oct</code>	<code>ios::oct</code>	Manipulator sets the <code>oct</code> flag. Which causes input and output of integers to use octal notation.	Input/Output	Both

If none of these flags is set then a decimal base is used, if more than one flag is set then a decimal base is used. During input if no flag is set then the base is determined by the leading characters. A number starting with `0x` or `0X` is read as a hexadecimal number, whilst if it starts with a leading zero it is read as an octal number. Using the manipulators is easier than using the flags.

To read/write binary numbers use the `bitset` class from the Standard Template Library.

6.5 Floating Point Representation

Manipulator	Affected Flag	Effect	Input/Output	g++ (2.95)
<code>fixed</code>	<code>ios::fixed</code>	The manipulator sets the flag. If flag is set use fixed point decimal notation for floating point numbers.	Output	Flag
<code>scientific</code>	<code>ios::scientific</code>	The manipulator sets the flag. If the flag is set then use scientific notation for floating point numbers.	Output	Flag
<code>showpoint</code>	<code>ios::showpoint</code>	The manipulator sets the flag. If the flag is set always write a decimal.	Output	Flag
<code>noshowpoint</code>	<code>ios::showpoint</code>	The manipulator clears the flag.	Output	Flag
<code>setprecision(val)</code>	None	Sets the precision of floating point printing to <code>val</code> .	Output	Manipulator

As you can see there are no manipulators for clearing the `fixed` or `scientific` flags. Use `unsetf` for this task. Output streams also have a `precision()` member function which returns the current precision.

6.6 Miscellaneous

Manipulator	Affected Flag	Effect	Input/Output	g++ (2.95)
<code>endl</code>	None	Output a newline character and flush the stream.	Output	Manipulator
<code>ends</code>	None	Output a null character.	Output	Manipulator
<code>flush</code>	None	Flush a stream.	Output	Manipulator

7 Stream Member Functions for Input and Output

In all the examples above we have used the extraction `>>` and insertion `<<` operators to read and write to our streams. However there are other member functions that can be used to read or write data. The main difference of these functions from the `<<` and `>>` operators is that they do not skip whitespace. If you need to read in whitespace, or C-style character strings then these are the functions to use. (Note that to read in C++-style strings with embedded blanks you should use the non-member function `getline` supplied with the `string` class.)

7.1 Input

The following functions are available. The class `istream` used to denote the class they are a member of can be any input stream. Most return the stream as the result of calling them I will ignore the use of this return value – that is we will just use them in the form

```
cin.function();
```

discarding the return value.

After the use of any of these functions you can query the stream state to determine if the operation was successful.

- `int istream::get()`
Read and return the next character (which can be any character including *EOF*). Note that the return type is an `int` rather than a `char` to allow for return of *EOF* etc.
- `istream& istream::get(char& ch)`
Reads next character and assigns it to `ch`.
- `istream& istream::get(char* str, int count, char delim='\n')`
Read up to `count-1` characters into the character array pointed at by `str`. Reading is terminated either by reaching the count or finding the delimiter. The delimiter is not read — it remains in the stream. The delimit character defaults to the newline character. The caller must ensure that the array `str` is large enough for all the characters. `str` will have a string termination character appended.
- `istream& istream::getline(char* str, int count, char delim='\n')`
Like the previous function but the delimiter is removed from the stream if it is encountered. It is discarded (not stored in `str`).
- `istream& istream::read(char* str, int count)`
Read `count` characters from the stream and place in array `str`. The array does not have a string termination character appended. The array must be large enough to hold `count` characters. Encountering end-of-file during the read is considered an error.

- `istream& istream::ignore(int count, char delim)`

Introduced above.

- `int istream::peek()`

Returns the next character to be read from the stream without extracting it. The next read will read this character. *EOF* is returned if no more characters can be read.

7.2 Output

- `ostream& ostream::put(char ch)`

Writes the character `ch` to the stream.

- `ostream& ostream::write(const char* str, int count)`

Writes `count` characters of the string `str` to the stream. The string termination character does not terminate the write operation and is written to the stream. Behaviour is undefined if the array `str` does not contain at least `count` characters (including any terminator).

8 File Streams

So far we have used `cout` and `cin` as our streams. They are predefined streams of type `ostream` and `istream` respectively. If you use them in a program, then under a system that allows I/O redirection you can pipe output to a file other than the console, and read in from a file other than the console. Eg the random number generator program could be used in the following fashion (under UNIX) to generate a file of random numbers.

```
rndnums > rands.txt
```

We very often have the requirement of reading from or writing to some named file from within the program. In this situation we should use stream objects of type `ifstream` or `ofstream`. These are subclasses of the `istream` and `ostream` classes and hence inherit all their public member functions etc. In addition they provide some extra functionality of their own.

Note that we should still wherever possible write our classes and functions to use the `iostream` interface since then they can be used with the widest range of stream objects.

8.1 Constructing and Destroying File Objects

The constructors of `ofstream` and `ifstream` objects can take a string which is interpreted as a file name and attempt to open that file in the correct mode (for writing or reading respectively) and then bind the stream object to that file. Whether or not the opening of the file was successful is reflected in the stream object's state.

When the stream objects go out of scope the destructor will automatically close any associated open file.

Thus a typical file processing program would contain code like the following (file names would usually be passed in as command line arguments).

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{ string fname;
  cout << "Enter a file name: ";
```

```

getline(cin, fname);

ifstream ifstrm(fname.c_str());

if (!ifstrm)
    cout << "Couldn't open file: " << fname << endl;
else
    cout << "Found and opened file: " << fname << endl;
}

```

Note that the constructor requires a C-style character string so we have to use the `c_str()` conversion function of the `string` class.

Once the `fstream` object has been constructed you can write to it or read from it (depending on type) using any of the facilities described earlier.

8.2 Manual Opening and Closing

If you want to bind a file stream to more than one file during its lifetime, which file may be used for reading or writing, you will probably want to use the `open` and `close` member functions.

Thus just declaring a file stream, as in

```

ofstream ofstrm;
ifstream ifstrm;
fstream fstrm;    // An fstream object could be used for either reading or writing

```

uses the default constructor which does not bind the stream object to any file.

The following member functions are available to associate a stream with a file (and disassociate it).

- `open(char* name)`
Opens a file named `name` using the default mode.
- `open(char* name, ios::iostate flags)`
Opens a file named `name` using the mode given by `flags`.
- `close()`
Closes the file associated with the stream.
- `boolean is_open()`
Tests whether the stream has an associated open file.

The `flags` field in the second version of `open` can be any of the following.

Flag	Meaning
<code>ios::in</code>	Open file for reading (file must exist)
<code>ios::out</code>	Open file for writing (empties it if it exists, creates otherwise)
<code>ios::out ios::trunc</code>	Same as above
<code>ios::out ios::app</code>	Opens for writing – appends new material (creates if necessary)
<code>ios::in ios::out</code>	Open file for reading and writing – file must exist
<code>ios::in ios::out ios::trunc</code>	Open file for reading and writing (empties if it exists, creates otherwise)
<code>ios::binary</code>	Use when file is to be treated as containing binary data.

9 String Streams

Streams can also be used to write to and read from in-memory strings. This allows the program to format text for output in a string which can then be sent to an output channel at a later time.

Another use is to format some text before displaying it in a text field of a GUI component.

The C++ Standard Library provides classes `ostringstream`, `istringstream` and `stringstream` for this purpose. These are easier to use than the old-style `stringstream` classes. Unfortunately the GNU implementation of the Standard Library does not yet provide the new style string stream classes and we must use the old-style ones.

Since the output string stream is the most useful I only consider it here. The following short program illustrates the basics of its use.

```
#include <iostream>
#include <stringstream>

int main()
{ double x = 3.14;
  ostringstream buffer;

  buffer << "Double x: " << x << ends;

  cout << buffer.str();
}
```

Note that

- you must append `ends` to the constructed string as the null character is not otherwise added to the string.
- the member function `str()` is used to convert the `ostringstream` object to a C-style character string.

Once the `str()` function has been called the stream is no longer allowed to modify the character sequence. This is achieved by it calling the member function `freeze`. If you then want to reuse the stream (add you must unfreeze the stream by calling `freeze(false)` upon it and move to the start of the character array by calling `seekp(0, ios::beg)`). Eg:

```
#include <iostream>
#include <stringstream>

int main()
{ double x = 3.14;
  ostringstream buffer;

  buffer << "Double x: " << x << ends;

  cout << buffer.str() << endl;

  buffer.freeze(false);
  buffer.seekp(0, ios::beg);
  buffer << "x times 2 is " << 2*x << ends;

  cout << buffer.str() << endl;
}
```

Note that `seekp` can be used to position the new write to a different position than just the beginning – its first argument gives the offset from the position given by the second argument.

The available position indicators are

Constant	Meaning
<code>ios::beg</code>	Position is relative to beginning.
<code>ios::cur</code>	Position is relative to current position
<code>ios::end</code>	Position is relative to the end.

The random access function `seekp` is defined for all `ostream` objects, the corresponding function for `istream` objects is `seekg`. There are also `tellp()` and `tellg()` functions which return the current position in the stream.

Note however that although defined for all streams the random access functions have undefined behaviour when used with streams `cout`, `cin` etc. They should only be used for `fstream`'s and `strstream`'s.

References

- [1] N. M. Josuttis
The C++ Standard Library: A Tutorial and Reference
Addison-Wesley 1999, ISBN: 0-201-37926-0