# A Gentle Introduction to C++ IO Streams

By Manasij Mukherjee

One of the great strengths of C++ is its I/O system, IO Streams. As Bjarne Stroustrup says in his book "The C++ Programming Language", "Designing and implementing a general input/output facility for a programming language is notoriously difficult". He did an excellent job, and the C++ IOstreams library is part of the reason for C++'s success. IO streams provide an incredibly flexible yet simple way to design the input/output routines of any application.

IOstreams can be used for a wide variety of data manipulations thanks to the following features:

- A 'stream' is internally nothing but a series of characters. The characters may be either normal characters (char) or wide characters (wchar_t). Streams provide you with a universal character-based interface to any type of storage medium (for example, a file), without requiring you to know the details of how to write to the storage medium. Any object that can be written to one type of stream, can be written to all types of streams. In other words, as long as an object has a stream representation, any storage medium can accept objects with that stream representation.
- Streams work with built-in data types, and you can make user-defined types work with streams by overloading the insertion operator (<<) to put objects into streams, and the extraction operator (>>) to read objects from streams.
- The stream library.s unified approach makes it very friendly to use. Using a consistent interface for outputting to the screen and sending files over a network makes life easier. The programs below will show you what is possible.

The IO stream class hierarchy is quite complicated, so rather than introduce you to the full hierarchy at this point, I'll start with explaining the concepts of the design and show you examples of streams in action. Once you are familiar with elements of the design and how to apply those concepts to design a robust I/O system for your software, an understanding of what belongs where in the hierarchy will come naturally.

## What do input and output really mean?

To get a good idea about what input and output are, think of information as a stream of characters. This makes sense because whatever we enter through a keyboard can only be characters. Suppose the user enters the number 7479....WAIT...! How do you know the user entered a number? The problem is that you don't really know. All you have is a set of 4 characters: '7', '4', '7' and '9'. It is completely up to you, the programmer, whether you want the input to be a number, to be a string, or to be fodder for /dev/random; whether the characters can be valid for the desired type totally depends upon how whether that type can interpret the characters in the input stream as a description for an object of that type.

You have to get the input characters into a recognizable data type for them to be of any use other than as a character array.

IO streams not only defines the relation between a stream of characters and the standard data types but also allows you to define a relationship between a stream of characters and your own classes. It also allows you nearly limitless freedom to manipulate those streams both using object oriented interfaces and working directly on character buffers when necessary. (Of course some of the lower level manipulations may be undefined; for example, you can't probe forward into an input stream to see the future!)

## How do streams work?

Streams are serial interfaces to storage, buffers files, or any other storage medium. The difference between storage media is intentionally hidden by the interface; you may not even know what kind of storage you're working with but the interface is exactly the same.

The "serial" nature of streams is a very important element of their interface. You cannot directly make random access random reads or writes in a stream (unlike, say, using an array index to access any value you want) although you can seek to a position in a stream and perform a read at that point.

Using a serial representation gives a consistent interface for all devices. Many devices have the capability of both producing and consuming data at the same time; if data is being continually produced, the simplest way to think about reading that data is by doing a fetch of the next characters in a stream. If that data hasn't been produced yet (the user hasn't typed anything, or the network is still busy processing a packet), you wait for more data to become available, and the read will return that data. Even if you try to seek past the end (or beginning) of a stream, the stream pointer (i.e. get or put pointer) will remain at the boundary, making the situation safe. (Compare this with accessing data off the end of an array, where the behavior is undefined.)

The underlying low-level interface that corresponds to the actual medium very closely is a character buffer (the stream buffer, technically called the streambuf), which can be thought of as the backbone of the stream. Being a buffer, it does not hold the entire content of the stream, if the stream is large enough, so you can't use it for random access.

The most important of the basic stream operations are:

1. First, the stream is initialized with the appropriate type (like a std::string for a stringstream and the filename for an fstream) of values and suitable modes (like ios::in for input and ios::out for output and many more depending on the type of the stream).
2. After that, you can specify where the I/O should occur, through the get and put pointers. Depending on how you open the stream, the location may already be set appropriately (for example, if you open a file with ios::app, your get pointer set at the end of the stream, allowing appends).

   The member functions associated with setting the get and put pointers are:

   - seekg()and seekp() for .dragging. the get and put pointer, respectively, to the position specified. Both seek methods take an argument (of type streampos) providing a position in the file relative to the beginning of the file (using ios::beg), the end of the file (using ios::end), or the current position (using ios::cur). You may also provide just a specific location, such as io::beg, for the beginning of the file.
   - tellg() and tellp() provide the current location of the get and put pointers, respectively

   The following one-liners should clear up most questions:

   ```
   seekg(0); seekg(0,ios::beg);         //sets the get pointer to the beginning.
   seekg(5,ios::beg);      //sets the get pointer to 5 chars forward of the beginning.
   tellp(); tellg()              //returns the current value of the put/get pointer
   seekp(-10,ios::end);   //sets the put pointer to 10 chars before the end
   seekp(1,ios::cur);      //proceeds to next char
   ```

   N.B: Be careful when seeking the put pointer into the middle of in the stream. If you put anything in the stream, it will directly into the stream at the put location, overwriting the previous contents. In other words, if you need to insert data in the middle of a stream, you have to manually move the data that would be overwritten. As a side note, if you're finding yourself doing that too often, then you may want to use a string representation of your data, which can simplify this kind of random access operation.

3. Once you're at the right location in the stream, input and output is done through the << and >> operators. If you want to input an object to the stream, use the << operator; for output, use >>. The class for your object must, of course, have provided overloads for these methods. Here's a short example:

   ```
   //Inserts var into string (like objects are displayed by
   // putting them to cout)
   output_stream<<var;
   //Gets the value from the stream's characters positioned
   // after the get pointer and puts it into var.
   input_stream>>var;
   ```

   If var is an object (either a built in class or a user defined type), the exact process of the input or output is dependent on the overloaded >> or << operator respectively.

## Error handling with IO streams

Handling errors gracefully is important for building a robust system. The 'errors' or 'signals' (e.g., reaching the end of the file) in this case generally occur when a stream encounters something it didn't expect.

Whether a stream is currently valid can be checked by simply by using the stream as a Boolean:

```
ifstream file( "test.txt" );
if ( ! file )
{
        cout << "An error occurred opening the file" << endl;
}
```

More detailed status of the stream can be obtained using the good(), bad(), fail() and eof() functions. The clear() function will reset the stream and clear the error, which is necessary to perform any further IO on the stream.

- good() returns true when everything is okay.
- bad() returns true when a fatal error has occurred.
- fail() returns true after an unsuccessful stream operation like an unexpected type of input being encountered.
- eof() returns true when the end of file is reached.

You can detect that a particular read or write operation failed by testing the result of the read. For example, to check that a valid integer is read from the user, you can do this:

```
int x;
if ( cin >> x )
{
        cout << "Please enter a valid number" << endl;
}
```

This works because the read operation returns a reference to the stream.

## An example of creating a stream-enabled object

Here is a simple example of an utility designed for writing out logfile entries from command line arguments that takes advantage of some important stream facilities. If you don't understand something, refer to the tutorial on that particular topic.

```
#include <iostream>
#include <ctime>
#include <sstream>
#include <fstream>

using namespace std;

// timestamp returns the current time as a string
std::string timestamp();

class LogStatement;
ostream& operator<<(ostream& ost, const LogStatement& ls);

class LogStatement
{
public:
        LogStatement(std::string s): data(s), time_string( timestamp() )
        { };

        //This method handles all the outputs.
        friend ostream& operator<<(ostream&, const LogStatement&);
private:
        std::string data;
        std::string time_string;

};

ostream& operator<<(ostream& ost, const LogStatement& ls)
{
        ost<<"~|"<<ls.time_string<<'|'<<ls.data<<"|~";
        return ost;

}

std::string timestamp()
{
        //Notice the use of a stringstream, yet another useful stream medium!
        ostringstream stream;
        time_t rawtime;
        tm * timeinfo;

        time(&rawtime);
        timeinfo = localtime( &rawtime );

        stream << (timeinfo->tm_year)+1900<<" "<<timeinfo->tm_mon
        <<" "<<timeinfo->tm_mday<<" "<<timeinfo->tm_hour
        <<" "<<timeinfo->tm_min<<" "<<timeinfo->tm_sec;
        // The str() function of output stringstreams return a std::string.
        return stream.str();
}

int main(int argc, char** argv)
{
        if(argc<2)
        {
              // A return of -1 denotes an error condition.
              return -1;
        }
        ostringstream log_data;
        // This takes all the char arrays in the argv
        // (except the filename) and produces a stream.
        for(int i=1;i<argc;i++)
        {
               log_data<<argv[i]<<' ';
        }

        LogStatement log_entry(log_data.str());

        clog<<log_entry<<endl;

        ofstream logfile("logfile",ios::app);

        // check for errors opening the file
        if ( ! logfile )
        {
                return -1;
        }

        logfile<<log_entry<<endl;
        logfile.close();

        return 0;
}
```

This example should be pretty straightforward. The only two 'new' things here are:

1. The type ostream, which is the base class for any stream that accepts writes, such as ofstream and ostringstream. The standard stream objects: std::cout, std::cerr, std::clog and their 'wide' versions (e.g. std::wcout), are objects of this ostream class.
2. The use of stringstreams to simplify string handling. See the section on stringstreams below for more details on how stringstreams work.

## A few other things to note

The << operator takes in an ostream object, modifies it and returns it, even though it would have been enough to take the ostream object and modify it, with no return value. The value of returning the object is that you can chain operations, as in the following statement:

```
cout<<"Hey, I'm "<< n <<" years old.";
```

As the operator accepts an ostream, you can do anything with the data written to the stream simply by choosing a different subclass of ostream. The example shows writing to a file and to the system log console/file. You could also compress it and archive it, send it over a network, parse it later with another program to determine the number of log entries in a specified time interval, etc. Basically, you can use the class for any sort of output operation for which a subclass of ostream exists.

## Parts of the IO stream library

Now that you've seen the basic problems solved by IO streams and how they work, let's look at the different elements of the IO streams library, with a few examples of each in action.

### Standard Stream Objects for Console I/O: (cout, cin, cerr, clog, etc.)

These are declared in the <iostream> header and provide a consistent interface for console I/O. They allow you a lot of control about the exact way you want to read a value from the stream into the given variable (or write the variable to the stream).

### File I/O

File I/O is done by manually declaring objects of the ifstream, ofstream or fstream classes (from the <fstream> header) and then associating a file with the stream by using the stream's open method with the file's name as an argument. File I/O is particularly important because files are often used to represent a large variety of media, such as the console, devices, disk files, virtual memory, list of running processes and even the black hole '/dev/null'. This is especially true on *nix systems where it is commonly said that "Everything is a file".

Here is a very very simple example of writing to a file:

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

using namespace std;

int main()
{
        ofstream ofs("a.txt",ios::app);
        if(ofs.good())
        {
                ofs<<"Hello a.txt, I'm appending this on you.";
        }
        return 0;
}
```

### String Streams

Strings are streams and streams are strings. Both are character arrays, but each has a totally different interface (random access strings vs serial stringstreams). By providing both std::string and stringstreams, the C++ standard library ensures that you have the flexibility to choose either interface for your design.

By including the <sstream> header, you can make objects of the istringstream, ostringstream and stringstream types. These objects make certain kinds of string manipulations much easier.

You can, for example, open a string in a stringstream, extract a floating point number from it to do some operations, and put it back in the stream.

```
#include <iostream>
#include <sstream>

using namespace std;

int main()
{
        stringstream my_stream(ios::in|ios::out);
        std::string dat("Hey, I have a double : 74.79 .");

        my_stream.str(dat);
        my_stream.seekg(-7,ios::end);

        double val;
        my_stream>>val;

        val= val*val;

        my_stream.seekp(-7,ios::end);
        my_stream<<val;

        std::string new_val = my_stream.str();
        cout<<new_val;

        return 0;
}
```

The output is

```
Hey, I have a double : 5593.54
```

## The lower level, where streams meet buffers

This is the most interesting and confusing part of this library, letting you manipulate streams at their lowest levels, bytes and bits. This is accomplished by the abstract class streambuf from which stringbuf and filebuf are derived. Every stream object has one of those as their backbones. Their function rdbuf() lets you access a pointer to the underlying stream buffer.

Here is a very simple example of copying a file efficiently with those buffers (thankfully, no ultra-complicated manipulation is involved here!).

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
        ifstream ifs("a.txt");
        //ios::trunc means that the output file will be overwritten if exists
        ofstream ofs("a.txt.copy", ios::trunc);
        if (ifs &&  ofs )
        {
                ofs<<ifs.rdbuf();
        }

        return 0;
}
```