Module-5

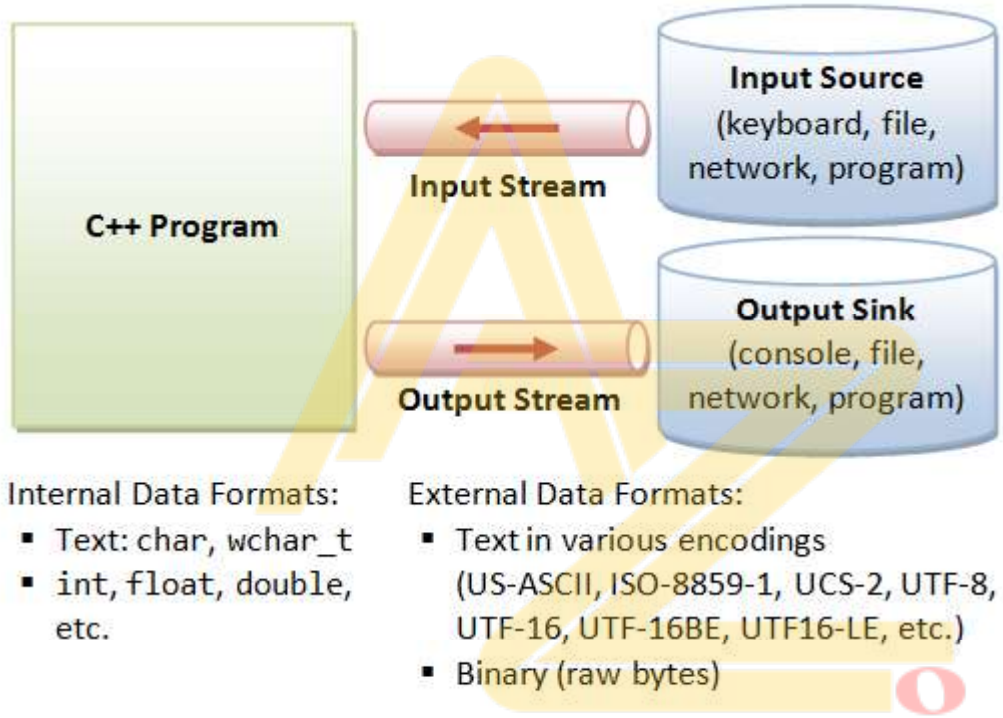
Streams and Working with Files

Table of Contents

C++ Stream

Streams

C/C++ IO are based on *streams*, which are sequence of bytes flowing in and out of the programs (just like water and oil flowing through a pipe). In input operations, data bytes flow from an *input source* (such as keyboard, file, network or another program) into the program. In output operations, data bytes flow from the program to an *output sink* (such as console, file, network or another program). Streams acts as an intermediaries between the programs and the actual IO devices, in such the way that frees the programmers from handling the actual devices, so as to archive device independent IO operations.



C++ provides both the *formatted* and *unformatted* IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.

To perform input and output, a C++ program:

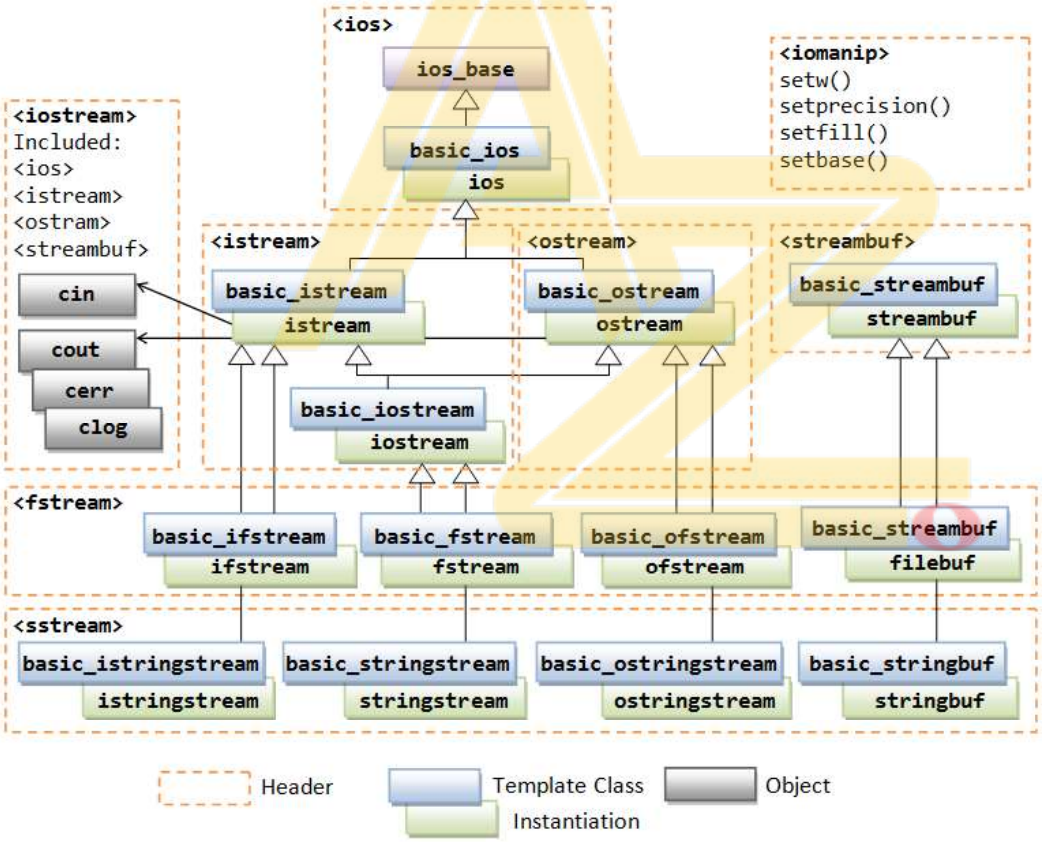
1. Construct a stream object.
2. Connect (Associate) the stream object to an actual IO device (e.g., keyboard, console, file, network, another program).

3. Perform input/output operations on the stream, via the functions defined in the stream's pubic interface in a device independent manner. Some functions convert the data between the external format and internal format (formatted IO); while other does not (unformatted or binary IO).
4. Disconnect (Dissociate) the stream to the actual IO device (e.g., close the file).
5. Free the stream object.

C++ IO Headers, Templates and Classes

Headers

C++ IO is provided in headers <iostream> (which included <ios>, <istream>, <ostream> and <streambuf>), <fstream> (for file IO), and <sstream> (for string IO). Furthermore, the header <iomanip> provided manipulators such as setw(), setprecision()setfill() and setbase() for formatting.



Template Classes

In order to support various character sets (char and wchar_t in C++98/03; and char16_t, char32_t introduced in C++11), the stream classes are written as template classes, which could be instantiated with an actual character type. Most of the template classes take two type parameters. For example,

```
template <class charT, class traits = char_traits<charT> >
class basic_istream;

template <class charT, class traits = char_traits<charT> >
class basic_ostream;
```

where:

- charT is the character type, such as char or wchar_t;
- traits, of another template class char_traits<charT>, defined the properties of the character operations such as the collating sequence (sorting order) of character set.

Template Instantiations and typedef

As mention, the basic_xxx template classes can be instantiated with a character type, such as char and wchar_t. C++ further provides typedef statements to name these classes:

```
typedef basic_ios<char>          ios;
typedef basic_ios<wchar_t>       wios;
typedef basic_istream<char>      istream;
typedef basic_istream<wchar_t>   wistream;
typedef basic_ostream<char>      ostream;
typedef basic_ostream<wchar_t>   wostream;
typedef basic_iostream<char>     iostream;
typedef basic_iostream<wchar_t>  wiostream;
typedef basic_streambuf<char>    streambuf;
typedef basic_streambuf<wchar_t> wstreambuf;
```

Specialization Classes for char type

We shall focus on the specialization classes for char type:

- ios_base and ios: superclasses to maintain common stream properties such as format flag, field width, precision and locale. The superclass ios_base (which is not a template class) maintains data that is independent of the template parameters; whereas the subclass ios (instantiation of template basic_ios<char>) maintains data which is dependent of the template parameters.
- istream (basic_istream<char>), ostream (basic_ostream<char>): provide the input and output public interfaces.
- iostream (basic_iostream<char>): subclass of both istream and ostream, which supports bidirectional input and output operations. Take note that istream and ostream are unidirectional streams; whereas iostream is bidirectional. basic_iostream template and iostream class is declared in the <istream> header, not <iostream> header.

- ifstream, ofstream and fstream: for file input, output and bidirectional input/output.
- istringstream, ostringstream and stringstream: for string buffer input, output and bidirectional input/output.
- streambuf, filebuf and stringbuf: provide memory buffer for the stream, file-stream and string-stream, and the public interface for accessing and managing the buffer.

Buffered IO

The <iostream> Header and the Standard Stream Objects: cin, cout, cerr and clog

The <iostream> header also included the these headers: <ios>, <istream>, <ostream> and <streambuf>. Hence, your program needs to include only the <iostream> header for IO operations.

The <iostream> header declares these *standard stream objects*:

1. cin (of istream class, basic_istream<char> specialization), wcin (of wistream class, basic_istream<wchar_t> specialization): corresponding to the *standard input stream*, defaulted to keyword.
2. cout (of ostream class), wcout (of wostream class): corresponding to the *standard output stream*, defaulted to the display console.
3. cerr (of ostream class), wcerr (of wostream class): corresponding to the *standard error stream*, defaulted to the display console.
4. clog (of ostream class), wclog (of wostream class): corresponding to the *standard log stream*, defaulted to the display console.

The Stream Insertion << and Stream Extraction >> Operators

Formatted output is carried out on streams via the stream insertion << and stream extraction >> operators. For example,

```
cout << value;
cin >> variable;
```

Take note that cin/cout shall be the left operand and the data flow in the direction of the arrows.

The << and >> operators are overloaded to handle fundamental types (such as int and double), and classes (such as string). You can also overload these operators for your own user-defined types.

The cin << and cout >> return a reference to cin and cout, and thus, support cascading operations. For example,

```
cout << value1 << value2 << .... ;
cin >> variable1 << variable2 << .... ;
```

The ostream Class

The ostream class is a typedef to basic_ostream<char>. It contains two set of output functions: *formatted output* and *unformatted output*.

- The formatted output functions (via overloaded stream insertion operator <<) convert numeric values (such as int, double) from their internal representations (e.g., 16-/32-bit int, 64-bit double) to a stream of characters that representing the numeric values in text form.
- The unformatted output functions (e.g., put(), write()) outputs the bytes as they are, without format conversion.

Formatting Output via the Overloaded Stream Insertion << Operator

The ostream class overloads the stream insertion << operator for each of the C++ fundamental types (char, unsigned char, signed char, short, unsigned short, int, unsigned int, long, unsigned long, long long (C++11), unsigned long long (C++11), float, double and long double. It converts a numeric value from its internal representation to the text form.

```
ostream & operator<< (type)   // type of int, double etc
```

The << operator returns a reference to the invoking ostream object. Hence, you can concatenate << operations, e.g., cout << 123 << 1.13 << endl;.

The << operator is also overloaded for the following pointer types:

- const char *, const signed char *, const unsigned char *: for outputting C-strings and literals. It uses the terminating null character to decide the end of the char array.
- void *: can be used to print an address.

For example,

```
char str1[] = "apple";
const char * str2 = "orange";


cout << str1 << endl;   // with char *, print C-string
cout << str2 << endl;   // with char *, print C-string
cout << (void *) str1 << endl;  // with void *, print address (regular cast)
cout << static_cast<void *>(str2) << endl;  // with void *, print address
```

Flushing the Output Buffer

You can flush the output buffer via:

1. flush member function or manipulator:

```
// Member function of ostream class - std::ostream::flush
ostream & flush ();
// Example
cout << "hello";
```

```
6.      cout.flush();
7.
8.      // Manipulator - std::flush
9.      ostream & flush (ostream & os);
10.     // Example
```

```
cout << "hello" << flush;
```

11. endl manipulator, which inserts a newline and flush the buffer. Outputting a newline character '\n' may not flush the output buffer; but endl does.

```
12.     // Manipulator - std::endl
```

```
ostream & endl (ostream & os)
```

13. cin: output buffer is flushed when input is pending, e.g.,

```
14.     cout << "Enter a number: ";
15.     int number;
```

```
cin << number;  // flush output buffer so as to show the prompting message
```

The istream class

Similar to the ostream class, the istream class is a typedef to basic_istream<char>. It also supports formatted input and unformatted input.

- In formatting input, via overloading the >> extraction operator, it converts the text form (a stream of character) into internal representation (such as 16-/32-bit int, 64-byte double).
- In unformatting input, such as get(), getlin(), read(), it reads the characters as they are, without conversion.

Formatting Input via the Overloaded Stream Extraction >> Operator

The istream class overloads the extraction >> operator for each of the C++ fundamental types (char, unsigned char, signed char, short, unsigned short, int, unsigned int, long, unsigned long, long long (C++11), unsigned long long (C++11), float, double and long double. It performs formatting by converting the input texts into the internal representation of the respective types.

```
istream & operator<< (type &)   // type of int, double etc.
```

The >> operator returns a reference to the invokind istream object. Hence, you can concatenate >> operations, e.g., cin >> number1 << number2 <<....

The >> operator is also overloaded for the following pointer types:

- const char *, const signed char *, const unsigned char *: for inputting C-strings. It uses whitespace as delimiter and adds a terminating null character to the C-string.

[TODO] Read "C-string input".

Flushing the Input Buffer - ignore()

You can use the ignore() to discard characters in the input buffer:

```
istream & ignore (int n = 1, int delim = EOF);
   // Read and discard up to n characters or delim, whichever comes first

// Examples
cin.ignore(numeric_limits<streamsize>::max());        // Ignore to the end-of-file
cin.ignore(numeric_limits<streamsize>::max(), '\n');  // Ignore to the end-of-line
```

1.8 Unformatted Input/Output Functions

put(), get() and getline()

The ostream's member function put() can be used to put out a char. put() returns the invoking ostream reference, and thus, can be cascaded. For example,

```
// ostream class
ostream & put (char c);  // put char c to ostream
// Examples
cout.put('A');
cout.put('A').put('p').put('p').put('\n');
cout.put(65);
// istream class
// Single character input
int get ();
    // Get a char and return as int. It returns EOF at end-of-file
istream & get (char & c);
    // Get a char, store in c and return the invoking istream reference

// C-string input
istream & get (char * cstr, streamsize n, char delim = '\n');
    // Get n-1 chars or until delimiter and store in C-string array cstr.
    // Append null char to terminate C-string
    // Keep the delim char in the input stream.
istream & getline (char * cstr, streamsize n, char delim = '\n');
```

```
    // Same as get(), but extract and discard delim char from the
    // input stream.
// Examples
int inChar;
while ((inChar = cin.get()) != EOF) {  // Read till End-of-file
  cout.put(inchar);
}
```

[TODO] Example

read(), write() and gcount()

```
// istream class
istream & read (char * buf, streamsize n);
    // Read n characters from istream and keep in char array buf.
    // Unlike get()/getline(), it does not append null char at the end of input.
    // It is used for binary input, instead of C-string.
streamsize gcount() const;
    // Return the number of character extracted by the last unformatted input operation
    // get(), getline(), ignore() or read().

// ostream class
ostream & write (const char * buf, streamsize n)
    // Write n character from char array.
// Example
[TODO]
```

Other istream functions - peek() and putback()

```
char peek ();
    //returns the next character in the input buffer without extracting it.

istream & putback (char c);
    // insert the character back to the input buffer.
```

States of stream

The steam superclass ios_base maintains a data member to describe the *states* of the stream, which is a bitmask of the type iostate. The flags are:

- eofbit: set when an input operation reaches end-of-file.

- failbit: The last input operation failed to read the expected characters or output operation failed to write the expected characters, e.g., getline() reads n characters without reaching delimiter character.
- badbit: serious error due to failure of an IO operation (e.g. file read/write error) or stream buffer.
- goodbit: Absence of above error with value of 0.

These flags are defined as public static members in ios_base. They can be accessed directly via ios_base::failbit or via subclasses such as cin::failbit, ios::failbit. However, it is more convenience to use these public member functions of ios class:

- good(): returns true if goodbit is set (i.e., no error).
- eof(): returns true if eofbit is set.
- fail(): returns true if failbit or badbit is set.
- bad(): returns true if badbit is set.
- clear(): clear eofbit, failbit and badbit.

Formatting Input/Output via Manipulators in <iomanip> and <iostream>

C++ provides a set of manipulators to perform input and output formatting:

1. <iomanip> header: setw(), setprecision(), setbas(), setfill().
2. <iostream> header: fixed|scientific, left|right|internal, boolalpha|noboolalpha, etc.

Default Output Formatting

The ostream's << stream insertion operator is overloaded to convert a numeric value from its internal representation (e.g., 16-/32-bit int, 64-bit double) to the text form.

- By default, the values are displayed with a field-width just enough to hold the text, without additional leading or trailing spaces. You need to provide spaces between the values, if desired.
- For integers, all digits will be displayed, by default. For example,

```
cout << "|" <<  1 << "|"  << endl;        // |1|
cout << "|" << -1 << "|"  << endl;        // |-1|
cout << "|" <<  123456789 << "|" << endl;  // |123456789|
```

```
cout << "|" << -123456789 << "|" << endl;  // |-123456789|
```

- For floating-point numbers, the default *precison* is 6 digits, except that the trailing zeros will not be shown. This default precision (of 6 digits) include all digits before and after the decimal point, but exclude the leading zeros. Scientific notation (E-notation) will be used if the exponent is 6 or more or -5 or less. In scientific notation, the default precision is also 6 digits; the exponent is displayed in 3 digits with plus/minus sign (e.g., +006, -005). For example,

```
cout << "|" << 1.20000 << "|" << endl;       // |1.2| (trailing zeros not displayed)
```

```
• cout << "|" << 1.23456 << "|" << endl;       // |1.23456| (default precision is 6 digits)
• cout << "|" << -1.23456 << "|" << endl;      // |-1.23456|
• cout << "|" << 1.234567 << "|" << endl;      // |1.23457|
• cout << "|" << 123456.7 << "|" << endl;      // |123457|
• cout << "|" << 1234567.89 << "|" << endl;    // |1.23457e+006| (scientific-notation for e>=6)
• cout << "|" << 0.0001234567 << "|" << endl;  // |0.000123457|  (leading zeros not counted towards precision)
```

```
cout << "|" << 0.00001234567 << "|" << endl; // |1.23457e-005| (scientific-notation for e<=-5)
```

- bool values are displayed as 0 or 1 by default, instead of true or false.

Field Width (setw), Fill Character (setfill) and Alignment (left|right|internal)

The ios_base superclass (included in <iostream> header) maintains data members for field-width (width) and formatting flags (fmtflags); and provides member functions (such as width(), setf()) for manipulating them.

However, it is more convenience to use the so-called *IO manipulators*, which returns a reference to the invoking stream object and thus can be concatenated in << operator (e.g., cout << setfill(':') << left << setw(5) <<...). They are:

- **setw()** manipulator (in <iomanip> header) to set the field width.
- **setfill()** manipulator (in <iomanip> header) to set the fill character
- **left|right|internal** manipulator (in <iostream> header) to set the text alignment.

The default field-width is 0, i.e., just enough space to display the value. C++ never truncates data, and will expand the field to display the entire value if the field-width is too small. The setw() operation is *non-sticky*. That is, it is applicable only to the next IO operation, and reset back to 0 after the operation. The field-width property is applicable to both output and input operations.

Except setw(), all the other IO manipulators are *sticky*, i.e., they take effect until a new value is set.

```
// Test setw() - need <iomanip>
cout << "|" << setw(5) << 123 << "|" << 123 << endl;  // |  123|123
      // setw() is non-sticky. "|" and 123 displayed with default width
cout << "|" << setw(5) << -123 << "|" << endl;       // | -123|123
      // minus sign is included in field width
cout << "|" << setw(5) << 1234567 << "|" << endl;    // |1234567|
      // no truncation of data

// Test setfill() and alignment (left|right|internal)
cout << setfill('_');  // Set the fill character (sticky)
cout << setw(6) << 123 << setw(4) << 12 << endl;     // ___123__12
cout << left;          // left align (sticky)
cout << setw(6) << 123 << setw(4) << 12 << endl;     // 123___12__
```

Example: Alignment

```
cout << showpos;  // show positive sign
cout           << '|' << setw(6) << 123 << '|' << endl;  // |  +123| (default alignment)
cout << left   << '|' << setw(6) << 123 << '|' << endl;  // |+123  |
cout << right  << '|' << setw(6) << 123 << '|' << endl;  // |  +123|
cout << internal << '|' << setw(6) << 123 << '|' << endl;  // |+  123|
```

The internal alignment left-align the sign, but right-align the number, as illustrated.

[TODO] Example of field-width for input operations

You can also use ostream's member function width() (e.g. cout.width(n)) to set the field width, but width() cannot be used with cout << operator.

Floating-point Format (fixed|scientific) and Precision (setprecision)

The IO stream superclass ios_base also maintains data member for the floating-point precision and display format; and provides member functions (such as precision()) for manipulating them.

Again, it is more convenience to use IO manipulators, which can be concatenated in <<. They are:

- **setprecision()** manipulator (in <iomanip> header) to set the precision of floating-point number.
- **fixed|scientific** manipulators (in <iostream> header) to set the floating-point display format.

Floating point number can be display in 3 formatting modes: *default*|fixed|scientific. The precision is interpreted differently in *default* and *non-default* modes (due to legacy).

- In *default* mode (neither fixed nor scientific used), a floating-point number is displayed in fixed-point notation (e.g., 12.34) for exponent in the range of [-4, 5]; and scientific notation (e.g., 1.2e+006) otherwise. The precision in default mode includes digits before and after the decimal point but exclude the leading zeros. Fewer digits might be shown as the trailing zeros are not displayed. The default precision is 6. See the earlier examples for default mode with default precision of 6. As mentioned, the trailing zeros are not displayed in default mode, you can use manipulator showpoint|noshowpoint to show or hide the trailing zeros.
- In both fixed (e.g., 12.34) and scientific (e.g., 1.2e+006), the precision sets the number of digits after decimal point. The default precision is also 6.

For examples,

```
// default floating-point format
cout << "|" << 123.456789 << "|" << endl;    // |123.457| (fixed-point format)
      // default precision is 6, i.e., 6 digits before and after the decimal point
cout << "|" << 1234567.89 << "|" << endl;    // |1.23457e+006| (scientific-notation for e>=6)
      // default precision is 6, i.e., 6 digits before and after the decimal point
```

```
// showpoint - show trailing zeros in default mode
cout << showpoint << 123. << "," << 123.4 << endl; // 123.000,123.400
cout << noshowpoint << 123. << endl;              // 123

// fixed-point formatting
cout << fixed;
cout << "|" << 1234567.89 << "|" << endl;    // |1234567.890000|
      // default precision is 6, i.e., 6 digits after the decimal point

// scientific formatting
cout << scientific;
cout << "|" << 1234567.89 << "|" << endl;    // |1.234568e+006|
      // default precision is 6, i.e., 6 digits after the decimal point

// Test precision
cout << fixed << setprecision(2);   // sticky
cout << "|" << 123.456789 << "|" << endl;  // |123.46|
cout << "|" << 123. << "|" << endl;        // |123.00|

cout << setprecision(0);
cout << "|" << 123.456789 << "|" << endl;  // |123|
```

You can also use ostream's member function precision(*n*) (e.g. cout.precision(n)) to set the floating-point precision, but precision() cannot be used with cout << operator.

Integral Number Base (dec|oct|hex, setbase)

C++ support number bases (radixes) of decimal, hexadecimal and octal. You can use the following manipulators (defined in ios_base class, included in <iostream> header) to manipulate the integral number base:

- **hex|dec|oct**: Set the integral number base. Negative hex and oct are displayed in 2's complement format. Alternatively, you can use setbase(8|10|16) (in header <iomanip>).
- **showbase|noshowbase**: write hex values with 0x prefix; and oct values with 0 prefix.
- **showpos|noshowpos**: write positive dec value with + sign.
- **uppercase|nouppercase**: write uppercase in certain insertion operations, e.g., hex digits. It does not convert characters or strings to uppercase!

These manipulators are *sticky*.

For examples,

```
cout << 1234 << endl;       // 1234 (default is dec)
cout << hex << 1234 << endl; // 4d2
cout << 1234 << "," << -1234 << endl; // 4d2,fffffb2e
                              // (hex is sticky, negative number in 2's complement)
cout << oct << 1234 << endl;         // 2322
cout << 1234 << "," << -1234 << endl; // 2322,37777775456
cout << setbase(10) << 1234 << endl;  // 1234 (setbase requires <iomanip> header)

// showbase - show hex with 0x prefix; oct with 0 prefix
cout << showbase << 123 << "," << hex << 123 << "," << oct << 123 << endl;  // 123,0x7b,0173
cout << noshowbase << dec;

// showpos - show dec's plus (+) sign
cout << showpos << 123 << endl;  // +123

// uppercase - display in uppercase (e.g., hex digits)
cout << uppercase << hex << 123 << endl;  // 7B
```

bool values (boolalpha|noboolalpha)

- **boolalpha|noboolalpha**: read/write bool value as alphabetic string true or false.

```
// boolalpha - display bool as true/false
cout << boolalpha << false << "," << true << endl;   // false,true
```

```
cout << noboolalpha << false << "," << true << endl; // 0,1
```

Other manipulators

- **skipws|noskipws**: skip leading white spaces for certain input operations.
- **unitbuf|nounibuf**: flush output after each insertion operation.

Notes

- You need to include the <iomanip> header for setw(), setprecision(), setfill(), and setbase().
- You can use ios_base's (in <iostream> header) member functions setf() and unsetf() to set the individual formatting flags. However, they are not as user-friendly as using manipulators as discussed above. Furthermore, they cannot be used with cout << operator.

The C++ string class Input/Output

File Input/Output (Header <fstream>)

C++ handles file IO similar to standard IO. In header <fstream>, the class ofstream is a subclass of ostream; ifstream is a subclass of istream; and fstream is a subclass of iostream for bi-directional IO. You need to include both <iostream> and <fstream> headers in your program for file IO.

To write to a file, you construct a ofsteam object connecting to the output file, and use the ostream functions such as stream insertion <<, put() and write(). Similarly, to read from an input file, construct an ifstream object connecting to the input file, and use the istream functions such as stream extraction >>, get(), getline() and read().

File IO requires an additional step to connect the file to the stream (i.e., file open) and disconnect from the stream (i.e., file close).

File Output

The steps are:

1. Construct an ostream object.
2. Connect it to a file (i.e., file open) and set the mode of file operation (e.g, truncate, append).
3. Perform output operation via insertion >> operator or write(), put() functions.
4. Disconnect (close the file which flushes the output buffer) and free the ostream object.

```
#include <fstream>

.......

ofstream fout;

fout.open(filename, mode);

......

fout.close();


// OR combine declaration and open()
ofstream fout(filename, mode);
```

By default, opening an output file creates a new file if the filename does not exist; or truncates it (clear its content) and starts writing as an empty file.

open(), close() and is_open()

```
void open (const char* filename,
        ios::openmode mode = ios::in | ios::out);
  // open() accepts only C-string. For string object, need to use c_str() to get the C-string

void close ();    // Closes the file, flush the buffer and disconnect from stream object

bool is_open ();  // Returns true if the file is successfully opened
```

File Modes

File modes are defined as static public member in ios_base superclass. They can be referenced from ios_base or its subclasses - we typically use subclass ios. The available file mode flags are:

1. ios::**in** - open file for input operation
2. ios::**out** - open file for output operation
3. ios::**app** - output appends at the end of the file.
4. ios::**trunc** - truncate the file and discard old contents.
5. ios::**binary** - for binary (raw byte) IO operation, instead of character-based.
6. ios::**ate** - position the file pointer "at the end" for input/output.

You can set multiple flags via bit-or (|) operator, e.g., ios::out | ios::app to append output at the end of the file.

For output, the default is ios::out | ios::trunc. For input, the default is ios::in.

File Input

The steps are:

1. Construct an istream object.
2. Connect it to a file (i.e., file open) and set the mode of file operation.
3. Perform output operation via extraction << operator or read(), get(), getline() functions.
4. Disconnect (close the file) and free the istream object.

```
#include <fstream>

.......

ifstream fin;

fin.open(filename, mode);

......

fin.close();
```

```
// OR combine declaration and open()
ifstream fin(filename, mode);
```

By default, opening an input file ....

Example on Simple File IO

```
1   /* Testing Simple File IO (TestSimpleFileIO.cpp) */
2   #include <iostream>
3   #include <fstream>
4   #include <cstdlib>
5   #include <string>
6   using namespace std;
7
8   int main() {
9     string filename = "test.txt";
10
11    // Write to File
12    ofstream fout(filename.c_str());  // default mode is ios::out | ios::trunc
13    if (!fout) {
14      cerr << "error: open file for output failed!" << endl;
15      abort();  // in <cstdlib> header
16    }
17    fout << "apple" << endl;
18    fout << "orange" << endl;
19    fout << "banana" << endl;
20    fout.close();
21
22    // Read from file
23    ifstream fin(filename.c_str());  // default mode ios::in
24    if (!fin) {
25      cerr << "error: open file for input failed!" << endl;
26      abort();
27    }
28    char ch;
29    while (fin.get(ch)) {  // till end-of-file
30      cout << ch;
```

```
31        }
32        fin.close();
33        return 0;
34     }
```

Program Notes:

- Most of the <fstream> functions (such as constructors, open()) supports filename in C-string only. You may need to extract the C-string from string object via the c_str() member function.
- You could use is_open() to check if the file is opened successfully.
- The get(char &) function returns a null pointer (converted to false) when it reaches end-of-file.

Binary file, read() and write()

We need to use read() and write() member functions for binary file (file mode of ios::binary), which read/write raw bytes without interpreting the bytes.

```
1         /* Testing Binary File IO (TestBinaryFileIO.cpp) */
2         #include <iostream>
3         #include <fstream>
4         #include <cstdlib>
5         #include <string>
6         using namespace std;
7
8         int main() {
9           string filename = "test.bin";
10
11          // Write to File
12          ofstream fout(filename.c_str(), ios::out | ios::binary);
13          if (!fout.is_open()) {
14            cerr << "error: open file for output failed!" << endl;
15            abort();
16          }
17          int i = 1234;
18          double d = 12.34;
19          fout.write((char *)&i, sizeof(int));
20          fout.write((char *)&d, sizeof(double));
21          fout.close();
22
```

```
23          // Read from file
24          ifstream fin(filename.c_str(), ios::in | ios::binary);
25          if (!fin.is_open()) {
26            cerr << "error: open file for input failed!" << endl;
27            abort();
28          }
29          int i_in;
30          double d_in;
31          fin.read((char *)&i_in, sizeof(int));
32          cout << i_in << endl;
33          fin.read((char *)&d_in, sizeof(double));
34          cout << d_in << endl;
35          fin.close();
36          return 0;
37        }
```

Random Access File

Random access file is associated with a file pointer, which can be moved directly to any location in the file. Random access is crucial in certain applications such as databases and indexes.

You can position the input pointer via seekg() and output pointer via seekp(). Each of them has two versions: absolute and relative positioning.

```
// Input file pointer (g for get)
istream & seekg (streampos pos);  // absolute position relative to beginning
istream & seekg (streamoff offset, ios::seekdir way);
    // with offset (positive or negative) relative to seekdir:
    // ios::beg (beginning), ios::cur (current), ios::end (end)
streampos tellg ();  // Returns the position of input pointer

// Output file pointer (p for put)
ostream & seekp (streampos pos);  // absolute
ostream & seekp (streamoff offset, ios::seekdir way);  // relative
streampos tellp ();   // Returns the position of output pointer
```

Random access file is typically process as binary file, in both input and output modes.

[TODO] Example

String Streams

C++ provides a <sstream> header, which uses the same public interface to support IO between a program and string object (buffer).

The string streams is based on ostringstream (subclass of ostream), istringstream (subclass of istream) and bi-directional stringstream (subclass of iostream).

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
```

Stream input can be used to validate input data; stream output can be used to format the output.

ostringstream

```
explicit ostringstream (ios::openmode mode = ios::out);  // default with empty string
explicit ostringstream (const string & buf,
                        ios::openmode mode = ios::out);  // with initial str

string str () const;          // Get contents
void str (const string & str); // Set contents
```

For example,

```
// construct output string stream (buffer) - need <sstream> header
ostringstream sout;

// Write into string buffer
sout << "apple" << endl;
sout << "orange" << endl;
sout << "banana" << endl;

// Get contents
cout << sout.str() << endl;
```

The ostringstream is responsible for dynamic memory allocation and management.

istringstream

```
explicit istringstream (ios::openmode mode = ios::in);  // default with empty string
explicit istringstream (const string & buf,
                        ios::openmode mode = ios::in);  // with initial string
```

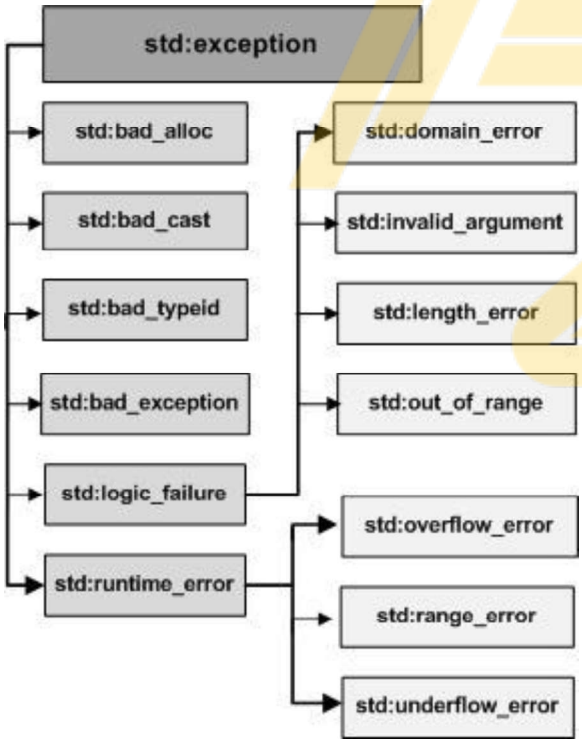
For example,

```
// construct input string stream (buffer) - need <sstream> header
istringstream sin("123 12.34 hello");

// Read from buffer
int i;
double d;
string s;
sin >> i >> d >> s;
cout << i << "," << d << "," << s << endl;
```

C++ Standard Exceptions:

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in an a parent-child class hierarchy shown below:



Here is the small description of each exception mentioned in the above hierarchy:

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

Define New Exceptions:

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example which shows how you can use std::exception class to implement your own exception in standard way:

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

```
struct MyException : public exception
{
  const char * what () const throw ()
  {
    return "C++ Exception";
  }
};

int main()
{
  try
  {
    throw MyException();
  }
  catch(MyException& e)
  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //Other errors
  }
}
```

This would produce following result:

Here **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

Create an Input Stream

To create an input stream, you must declare the stream to be of class ifstream. Here is the syntax:

ifstream fin;

Create an Output Stream

To create an output stream, you must declare it as class ofstream. Here is an example:

ofstream fout;

Create both Input/Output Streams

Streams that will be performing both input and output operations must be declared as class fstream. Here is an example:

Opening a File in C++

Once a stream has been created, next step is to associate a file with it. And thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways :

1. Using the constructor function of the stream class.
2. Using the function open().

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let's discuss each of these methods one by one.

Opening File Using Constructors

We know that a constructor of class initializes an object of its class when it (the object) is being created. Same way, the constructors of stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them. This is carried out as explained here: To open a file named myfile as an input file (i.e., data will be need from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type. Here is an example:

```
ifstream fin("myfile", ios::in) ;
```

The above given statement creates an object, fin, of input file stream. The object name is a user-defined name (i.e., any valid identifier name can be given). After creating the ifstream object fin, the file myfile is opened and attached to the input stream, fin. Now, both the data being read from myfile has been channelised through the input stream object.

Now to read from this file, this stream object will be used using the getfrom operator (">>"). Here is an example:

```
char ch;
fin >> ch ;       // read a character from the file
float amt ;
fin >> amt ;      // read a floating-point number form the file
```

Similarly, when you want a program to write a file i.e., to open an output file (on which no operation can take place except writing only). This will be accomplish by

1. creating ofstream object to manage the output stream
2. associating that object with a particular file

Here is an example,

```
ofstream fout("secret" ios::out) ;     // create ofstream object named as fout
```

This would create an output stream, object named as fout and attach the file secret with it.

Now, to write something to it, you can use << (put to operator) in familiar way. Here is an example,

```
int code = 2193 ;
fout << code << "xyz" ;    /* will write value of code
                                                and "xyz" to fout's associated
                                                file namely "secret" here. */
```

The connections with a file are closed automatically when the input and the output stream objects expires i.e., when they go out of scope. (For example, a global object expires when the program terminates). Also, you can close a connection with a file explicitly by using the close() method :

```
fin.close() ;     // close input connection to file
fout.close() ;    // close output connection to file
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. The stream still remains there. For example, after the above statements, the streams fin and fout still exist along with the buffers they manage. You can reconnect the stream to the same file or to another file, if required. Closing a file flushes the buffer which means the data remaining in the buffer (input or output stream) is moved out of it in the direction it is ought to be. For example, when an input file's connection is closed, the data is moved from the input buffer to the program and when an output file's connection is closed, the data is moved from the output buffer to the disk file.

Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e., processing them one by one), then you can open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file. For example,

```
ifstream fin;                    // create an input stream
fin.open("Master.dat", ios::in);    // associate fin stream with file Master.dat
:                                // process Master.dat
fin.close();                     // terminate association with Master.dat

fin.open("Tran.dat", ios::in);      // associate fin stream with file Tran.dat
:                                // process Tran.dat
fin.close();                     // terminate association
```

The above code lets you handle reading two files in succession. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time.

The Concept of File Modes

The filemode describes how a file is to be used : to read from it, to write to it, to append it, and so on.

When you associate a stream with a file, either by initializing a file stream object with a file name or by using the open() method, you can provide a second argument specifying the file mode, as mentioned below :

```
stream_object.open("filename", (filemode) ) ;
```

The second method argument of open(), the filemode, is of type int, and you can choose one from several constants defined in the ios class.

List of File Modes in C++

Following table lists the filemodes available in C++ with their meaning :

Constant	Meaning	Stream Type
ios :: in	It opens file for reading, i.e., in input mode.	ifstream
ios :: out	It opens file for writing, i.e., in output mode. This also opens the file in ios :: trunc mode, by default. This means an existing file is truncated when opened, i.e., its previous contents are discarded.	ofstream
ios :: ate	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	ofstream ifstream
ios :: app	This causes all output to that file to be appended to the end. This value can be used only with files capable of output.	ofstream
ios :: trunc	This value causes the contents of a pre-existing file by the same name to be destroyed and truncates the file to zero length.	ofstream
ios :: nocreate	This cause the open() function to fail if the file does not already exist. It will not create a new file with that name.	ofstream
ios :: noreplace	This causes the open() function to fail if the file already exists. This is used when you want to create a new file and at the same time.	ofstream
ios :: binary	This causes a file to be opened in binary mode. By default, files are opened in text mode. When a file is opened in text mode, various character translations may take place, such as the conversion of carriage-return into newlines. However, no such character translations occur in file opened in binary mode.	ofstream ifstream

If the ifstream and ofstream constructors and the open() methods take two arguments each, how have we got by using just one in the previous examples ? As you probably have guessed, the prototypes for these class member functions provide default values for the second argument (the filemode argument). For example, the ifstream open() method and constructor use ios :: in (open for reading) as the default value for the mode argument, while the ofstream open() method and constructor use ios :: out (open for writing) as the default.

The fstream class does not provide a mode by default and, therefore, one must specify the mode explicitly when using an object of fstream class.

Both ios::ate and ios::app place you at the end of the file just opened. The difference between the two is that the ios::app mode allows you to add data to the end of the file only, when the ios::ate mode lets you write data anywhere in the file, even over old data.

You can combine two or more filemode constants using the C++ bitwise OR operator (symbol |). For example, the following statement :

```
ofstream fout;
fout.open("Master", ios :: app | ios :: nocreate);
```

will open a file in the append mode if the file exists and will abandon the file opening operation if the file does not exist.

To open a binary file, you need to specify ios :: binary along with the file mode, e.g.,

```
fout.open("Master", ios :: app | ios :: binary);
```

or,

```
fout.open("Main", ios :: out | ios :: nocreate | ios :: binary);
```

Closing a File in C++

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. The close() function accomplishes this task and it takes the following general form :

```
stream_object.close();
```

For example, if a file Master is connected with an ofstream object fout, its connections with the stream fout can be terminated by the following statement :

fout.close() ;

C++ Opening and Closing a File Example

Here is an example given, for the complete understanding on:

- how to open a file in C++ ?
- how to close a file in C++ ?

Let's look at this program.

```
/* C++ Opening and Closing a File
 * This program demonstrates, how
 * to open a file to store or retrieve
 * information to/from it. And then how
 * to close that file after storing
 * or retrieving the information to/from it. */


#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
        ofstream fout;
        ifstream fin;
        char fname[20];
        char rec[80], ch;
        clrscr();

        cout<<"Enter file name: ";
        cin.get(fname, 20);

        fout.open(fname, ios::out);

        if(!fout)
        {
                cout<<"Error in opening the file "<<fname;
                getch();
                exit(1);
        }
        cin.get(ch);

        cout<<"\nEnter a line to store in the file:\n";
        cin.get(rec, 80);
```

```
        fout<<rec<<"\n";
        cout<<"\nThe entered line stored in the file successfully..!!";
        cout<<"\nPress any key to see...\n";
        getch();
        fout.close();

        fin.open(fname, ios::in);
        if(!fin)
        {
                cout<<"Error in opening the file "<<fname;
                cout<<"\nPress any key to exit...";
                getch();
                exit(2);
        }

        cin.get(ch);
        fin.get(rec, 80);
        cout<<"\nThe file contains:\n";
        cout<<rec;
        cout<<"\n\nPress any key to exit...\n";
        fin.close();

        getch();
}
```

EOD of File

so, just how much data is in that file? The exact contents of a file may not be precisely known. Usually the general format style of the file and the type of data contained within the file are known.

The amount of data stored in the file, however, is often unknown. So, do we spend our time counting data in a text file by hand, or do we let the computer deal with the amount of data? Of course, we let the computer do the counting.

C++ provides a special function, **eof( )**, that returns nonzero (meaning TRUE) when there are no more data to be read from an input file stream, and zero (meaning FALSE) otherwise.

Rules for using end-of-file (eof( )):

1. Always test for the end-of-file condition before processing data read from an input file stream.
    a. use a priming input statement before starting the loop

b. repeat the input statement at the bottom of the loop body

2. Use a while loop for getting data from an input file stream. A for loop is desirable only when you know the exact number of data items in the file, which we do not know.

```
#include <fstream.h>
#include <assert.h>

int main(void)
{
    int data;          // file contains an undermined number of integer values
    ifstream fin;     // declare stream variable name

    fin.open("myfile",ios::in);    // open file
    assert (!fin.fail( ));
    fin >> data;        // get first number from the file (priming the input statement)
                        // You must attempt to read info prior to an eof( ) test.
    while (!fin.eof( ))      //if not at end of file, continue reading numbers
    {
        cout<<data<<endl;    //print numbers to screen
        fin >> data;              //get next number from file
    }
    fin.close( );       //close file
    assert(!fin.fail( ));
    return 0;
}
```

The eof( ) function has been known to be persnickety under certain conditions. If you experience may want to consider this alternate approach to check for end of file:

```
//This example creates a file of apstrings
//then opens the new file and prints the info to the screen
#include<iostream.h>
#include<fstream.h>
```

```
#include<stdlib.h>              //for exit()
#include "apstring.cpp"

int main(void)
{
    ofstream fout;
    ifstream fin;
    apstring sentences, sent;

    fout.open("sentences.dat");  //creating the file
    if (!fout)
    {
        cerr<<"Unable to open file"<<endl;
        exit(1);
    }
    for(int i = 0; i < 5; i++)              //file will contain 5 apstring variables
    fout<<"This is sentence #"<<i+1<<endl;
    fout.close( );
   //open file and read from file
    fin.open("sentences.dat");      //open file to access information
    while (getline(fin,sent))   //The test condition is TRUE
                                // only while there is something to read.
    {                           //Works nicely as an end of file check.
        cout<<sent<<endl;
    }
    fin.close( );
    return 0;
}
```

Reference and Bibliography

1. Object Oriented Programming with C++ E.Balaguruswamy TMH 6th Edition, 2013
2. ObjectOriented Programming with C++ Robert Lafore Galgotia publication 2010
3. ObjectOriented Programming with C++ Sourav Sahay Oxford University 2006
4. Preece, J. Rogers, Y. and Sharp,H., 2007. Interaction Design: Beyond Human Computer Interaction. 2nd ed. New York: John Wiley & Sons, Inc.
5. Preece, J. Rogers, Y. and Sharp,H., 2001. Interaction Design: Beyond Human Computer Interaction. New York: John Wiley & Sons, Inc.
6. Andrew, G and Drew, P, 2009, Use Case Diagrams in Support of Use Case Modeling: Deriving Understanding from the Picture, Journal of Database Management, 20(1), 1- 24, January-March 2009.