Friend Classes, I/O Streams

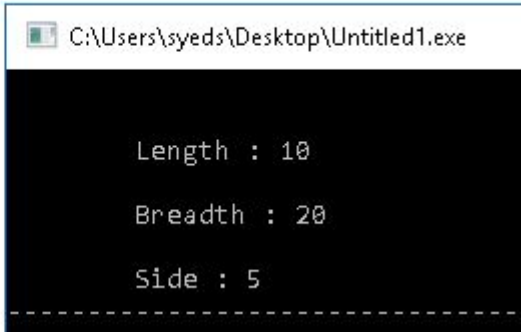
By, Mahrukh Khan

Friend Class

- like a friend function, a class can also be made a friend of another class using keyword.
- A friend class can access all the private and protected members of other class.
- In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.
- When a class is made a friend class, all the member functions of that class becomes friend functions.

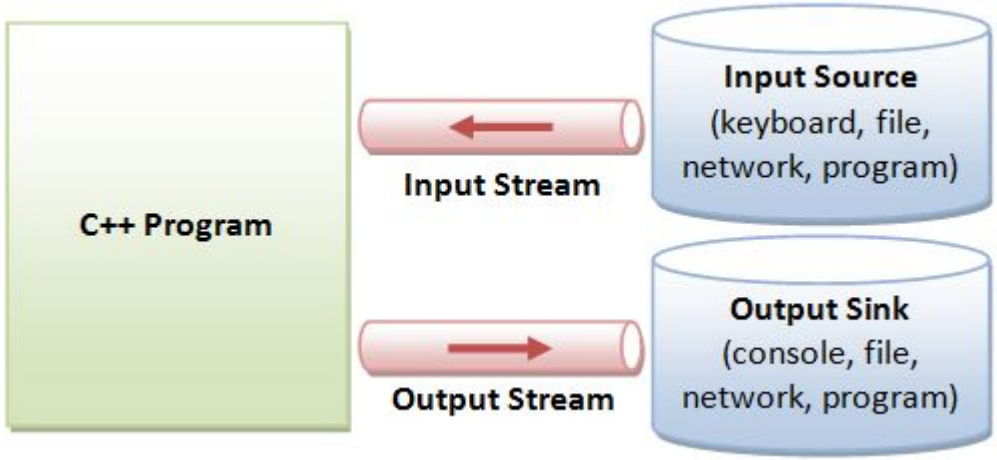
```
class Rectangle {
    int L,B;

    public:
    Rectangle()     {
        L=10;
        B=20;
    }

    friend class Square;
};
class Square    {
    int S;

    public:
    Square()        {
        S=5;
    }

    void Display(Rectangle Rect)        {
        cout<<"\n\n\tLength : "<<Rect.L;
        cout<<"\n\n\tBreadth : "<<Rect.B;
        cout<<"\n\n\tSide : "<<S;
    };
```

```
int main()
{

    Rectangle R;
    Square S;

    S.Display(R);

}
```



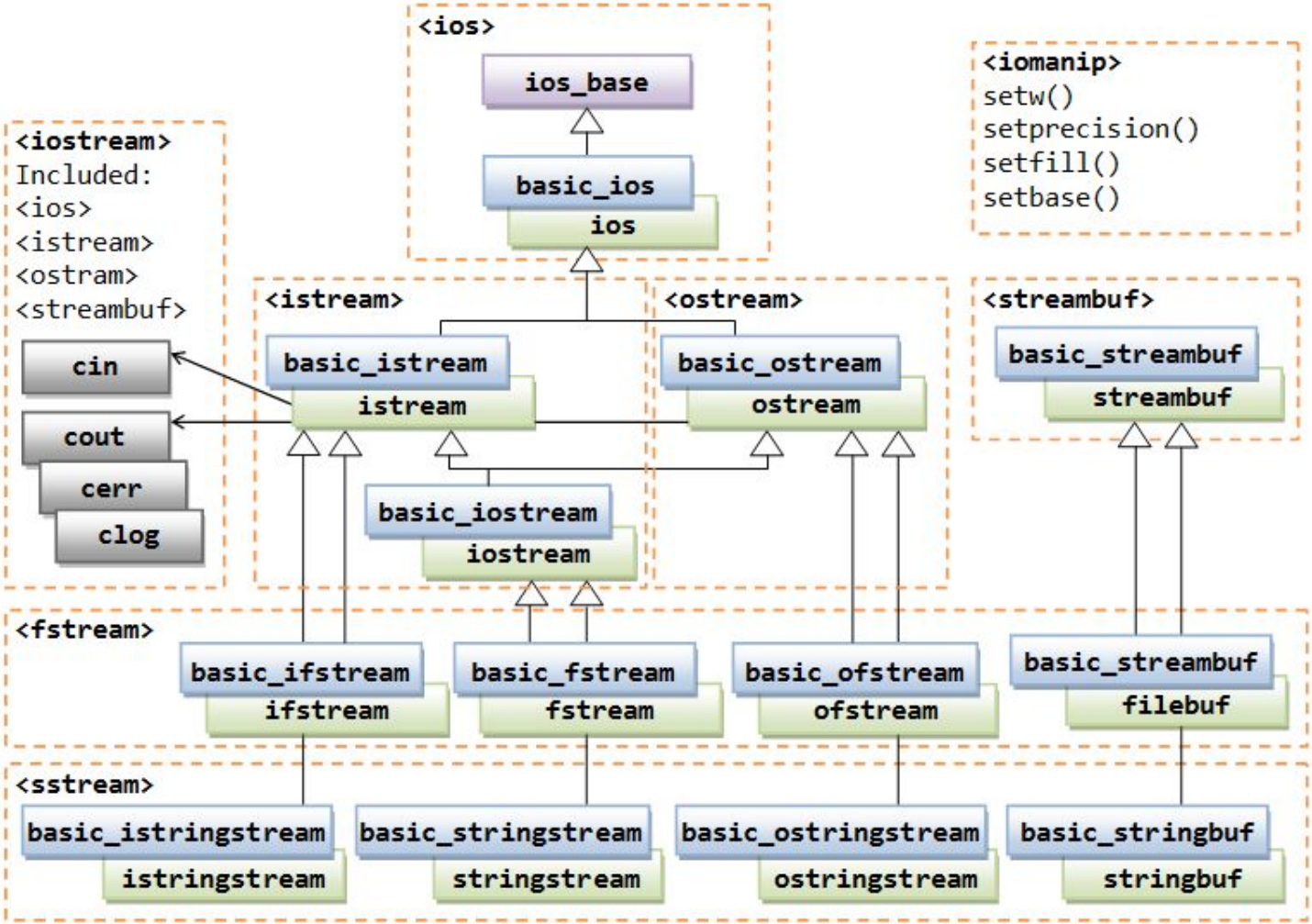
Streams

- C++ I/O occurs in streams, which are sequences of bytes. A 'stream' is internally nothing but a series of characters.
- If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called input stream and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called output stream.
- 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.
- C++ continues this approach and formalizes IO in libraries such as iostream and fstream.


C++ Program
Input Stream
Input Source
(keyboard, file,
network, program)
Output Stream
Output Sink
(console, file,
network, program)

C++ I/O Headers

iostream	iostream stands for standard input output stream. This header file contains definitions to objects like cin, cout, cerr etc.
iomanip	iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
fstream	This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.


<ios>
ios_base
basic_ios
ios
<iomanip>
setw()
setprecision()
setfill()
setbase()
<iostream>
Included:
<ios>
<istream>
<ostram>
<streambuf>
cin
cout
cerr
clog
<istream>
basic_istream
istream
<ostream>
basic_ostream
ostream
basic_iostream
iostream
<streambuf>
basic_streambuf
streambuf
<fstream>
basic_ifstream
ifstream
basic_fstream
fstream
basic_ofstream
ofstream
basic_streambuf
filebuf
<sstream>
basic_istringstream
istringstream
basic_stringstream
stringstream
basic_ostringstream
ostringstream
basic_stringbuf
stringbuf

iostream

- 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.
- cout (of ostream class), wcout (of wostream class): corresponding to the *standard output stream*, defaulted to the display console.
- cerr (of ostream class), wcerr (of wostream class): corresponding to the *standard error stream*, defaulted to the display console.
- clog (of ostream class), wclog (of wostream class): corresponding to the *standard log stream*, defaulted to the display console

Formatted I/O : cout and cin

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

  cout << *value*;

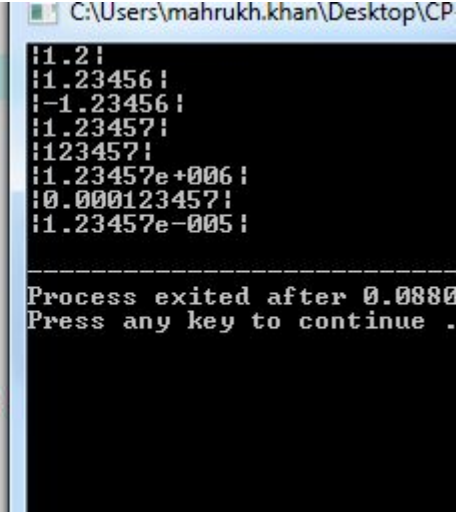
  cin >> *variable*;

- 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.

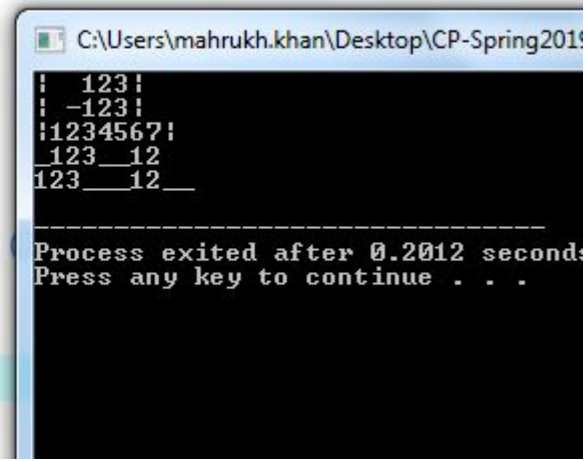
iomanip

- C++ provides a set of manipulators to perform input and output formatting:
  - <iomanip> header: setw(), setprecision(), setbas(), setfill().
  - <iostream> header: fixed|scientific, left|right|internal, boolalpha|noboolalpha, etc.
- 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 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

```cpp
int main(){
cout << "|" << 1.20000 << "|" << endl;
cout << "|" << 1.23456 << "|" << endl;
cout << "|" << -1.23456 << "|" << endl;
cout << "|" << 1.234567 << "|" << endl;
cout << "|" << 123456.7 << "|" << endl;
cout << "|" << 1234567.89 << "|" << endl;
cout << "|" << 0.0001234567 << "|" << endl;
cout << "|" << 0.00001234567 << "|" << endl;
}
```




```cpp
#include<iomanip>
using namespace std;
int main(){
cout << "|" << setw(5) << 123 << "|" <<endl;
cout << "|" << setw(5) << -123 << "|" << endl;
cout << "|" << setw(5) << 1234567 << "|" << endl;
cout << setfill('_');   // Set the fill character
cout << setw(4) << 123 << setw(4) << 12 << endl;
cout << left;           // left align (sticky)
cout << setw(6) << 123 << setw(4) << 12 << endl;
}
```



```cpp
int main(){
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|
```

C:\Users\mahrukh.khan\Desktop\CP-Spring2019\Lessons\Week 12\cout.exe

```
|123.457|
|1.23457e+006|
123.000,123.400
123
|1234567.890000|
|1.234568e+006|
|123.46|
|123.00|
|123|
```

Overloading Stream Extraction/Insertion Operators

- The stream extraction operator function operator>> takes the istream reference input and the PhoneNumber reference number as arguments and returns an istream reference.
- Function operator>> returns istream reference input (i.e., cin). This enables input operations on PhoneNumber objects to be cascaded with input operations on other PhoneNumber objects or other data types.
- The functions operator>> and operator<< are declared in PhoneNumber as non-member,friend functions . They're non-member functions because the object of class PhoneNumber must be the operator's right operand. If these were to be PhoneNumber member functions, the following awkward statements would have to be used to output and input an Array:

```
phone << cout;
phone >> cin;
```

Why these functions are friends and not member functions

- Overloaded input and output operators are declared as friends if they need to access non-public class members directly for performance reasons .
- The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type ostream &, as in cout << classObject. To use the operator in this manner where the right operand is an object of a user-defined class, it must be overloaded as a non-member function. To be a member function, operator << would have to be a member of the ostream class. This is not possible for user-defined classes, since we are not allowed to modify C++ Standard Library classes.
- Similarly, the overloaded stream extraction operator (>>) is used in an expression in which the left operand has the type istream &, as in the expression cin >> classObject, and the right operand is an object of a user-defined class, so it, too, must be a non-member function. Also, each of these overloaded operator functions may require access to the private data members of the class object being output or input, so these overloaded operator functions can be made friend functions of the class for performance reasons.

Files

- Ofstream: This data type represents the output file stream and is used to create files and to write information to files.
- Ifstream: This data type represents the input file stream and is used to read information from files.
- Fstream: This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files

Opening a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. In order to open a file with a stream object we use its member function open:
open (filename, mode);
Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags.All these flags can be combined using the bitwise operator OR (|)

`ios::in`	Open for input operations.
`ios::out`	Open for output operations.
`ios::binary`	Open in binary mode.
`ios::ate`	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
`ios::app`	All output operations are performed at the end of the file, appending the content to the current content of the file.
`ios::trunc`	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

Default File Modes

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument

class	default mode parameter
`ofstream`	ios::out
`ifstream`	ios::in
`fstream`	ios::in | ios::out

Closing a File

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file

```
myfile.close();
```

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.

Defining Aliases

- The keyword **typedef** can be used to declare synonyms (aliases) for previously defined data types

- Creating a name using *typedef* does not create a new data type; typedef creates only an alternate name for the existing data type

Example

```
int main()
{
typedef int i;
i var1 = 5;
cout << var1;  // outputs 5
typedef float f;
f var2 = 2.8;
cout << var2;  // outputs 2.8
}
```

Get and put stream pointers

- **ifstream**, like **istream**, has a pointer known as *get pointer* that points to the next element to be read.
- **ofstream**, like **ostream**, has a pointer *put pointer* that points to the location where the next element has to be written.
- Finally **fstream**, like **iostream**, inherits both: *get* and *put*

Stream Functions

Thes stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

- **tellp() and tellg()**
  - These two member functions admit no parameters and return a value of type **pos_type** (according ANSI-C++ standard) that is an integer data type representing the current position of *get* stream pointer (in case of tellg) or *put* stream pointer (in case of tellp).
- **seekg()** and **seekp()**
  - This pair of functions serve respectively to change the position of stream pointers *get* and *put*. Both functions are overloaded with two different prototypes:

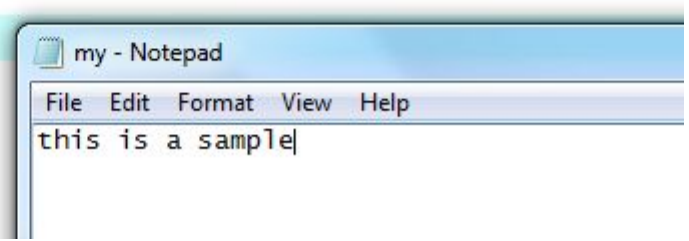
Stream Functions

- **seekg (pos_type** *position* **);**
- **seekp (pos_type** *position* **);**
- Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. The type required is the same as that returned by functions **tellg** and **tellp**.
- **seekg (off_type** *offset***, seekdir** *direction* **);**
- **seekp (off_type** *offset***, seekdir** *direction* **);**
- Using this prototype, an offset from a concrete point determined by parameter *direction* can be specified. It can be:

`ios::beg`	offset specified from the beginning of the stream
`ios::cur`	offset specified from the current position of the stream pointer
`ios::end`	offset specified from the end of the stream

Example

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

int main()
{
    long position;
    fstream file;
    file.open("my.txt");
    file.write("this is an apple", 16);
    position = file.tellp();
    file.seekp(position - 7);
    file.write(" sam", 4);
    file.close();
}
```



State Flags

- **bad()**
  - Returns **true** if a failure occurs in a reading or writing operation. For example in case we try to write to a file that is not open for writing or if the device where we try to write has no space left.
- **fail()**
  - Returns **true** in the same cases as **bad()** plus in case that a format error happens, as trying to read an integer number and an alphabetical character is received.
- **eof()**
  - Returns **true** if a file opened for reading has reached the end.
- **good()**
  - It is the most generic: returns **false** in the same cases in which calling any of the previous functions would return **true**.

In order to reset the state flags checked by the previous member functions you can use member function **clear()**, with no parameters.

Buffer and Synchronization

- When we operate with file streams, these are associated to an internal buffer object of type streambuf. This buffer object may represent a memory block that acts as an intermediary between the stream and the physical file.
- When the buffer is flushed, all data that it contains is written to the physic media (if it is an out stream) or simply erased (if it is an in stream). This process is called synchronization and it takes place under any of the following circumstances:
  - **When the file is closed:** before closing a file all buffers that have not yet been completely written or read are synchronized.
  - **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
  - **Explicitly with manipulators:** When certain manipulators are used on streams a synchronization takes place. These manipulators are: **flush** and **endl**.
  - **Explicitly with function sync():** Calling member function **sync()** (no parameters) causes an immediate syncronization. This function returns an **int** value equal to **-1** if the stream has no associated buffer or in case of failure.

Write()

- The write() function is used to write object or record (sequence of bytes) to the file. A record may be an array, structure or class.
- The write() function takes two arguments.
  - **&obj :** Initial byte of an object stored in memory.
  - **sizeof(obj) :** size of object represents the total number of bytes to be written from initial byte.

Syntax of write() function

```
fstream fout;
fout.write( (char *) &obj, sizeof(obj) );
```

Read()

- The read() function is used to read object (sequence of bytes) to the file.
- The read() function returns NULL if no data read.
- The read() function takes two arguments.
  - **&obj :** Initial byte of an object stored in file.
  - **sizeof(obj) :** size of object represents the total number of bytes to be read from initial byte.

Syntax of read() function

```
fstream fin;
fin.read( (char *) &obj, sizeof(obj) );
```