EEC-101
Programming with C++

Module-1.8 File IO

Ramanuja Panigrahi

Topics

- Concepts of algorithm & flow charts;
- Input/output,
- constants, variables
- operators;
- Naming conventions and styles;
- Conditions and selection statements;
- Looping and control structures (while, for, do-while, break and continue);
- File I/O, Header files, String processing;
- Pre-processor directives such as #include, #define, #ifdef, #ifndef;
- Compiling and linking.

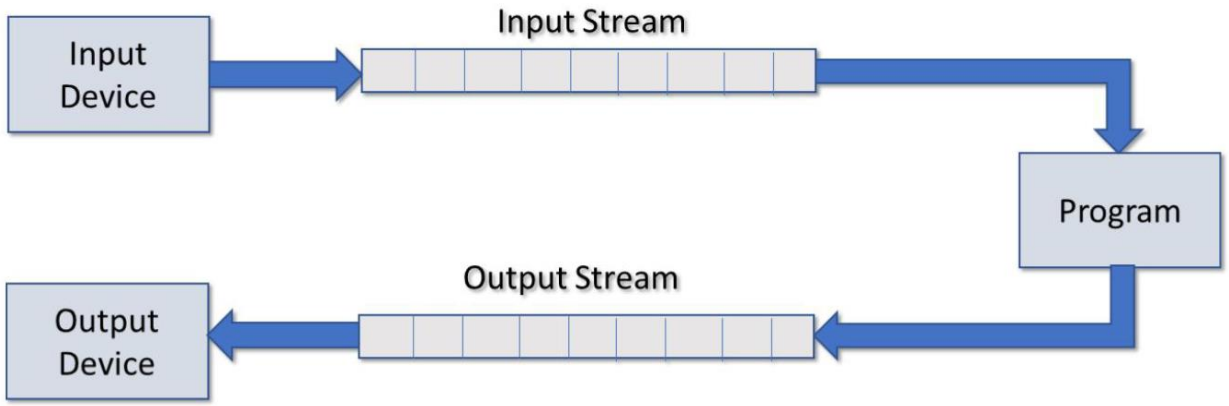
I/O

- Writing an IO library for any programming language is an extremely difficult task.
- There can be all sorts of devices providing data to your program, and your program can send data to many types of devices as well.
- These devices may be physical devices such as hard disks, the console, and the keyboard, or they can be virtual devices, such as some connection to a web server.

- C++ provides a stream abstraction to work with IO devices.

Streams in C++

- We give input to the executing program and the execution program gives back the output. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called streams. In other words, streams are nothing but the flow of data in a sequence.



The input and output operation between the executing program and the devices like the keyboard and monitor is known as "console I/O operation."

The input and output operation between the executing program and files are known as "disk I/O operation".

Header Files

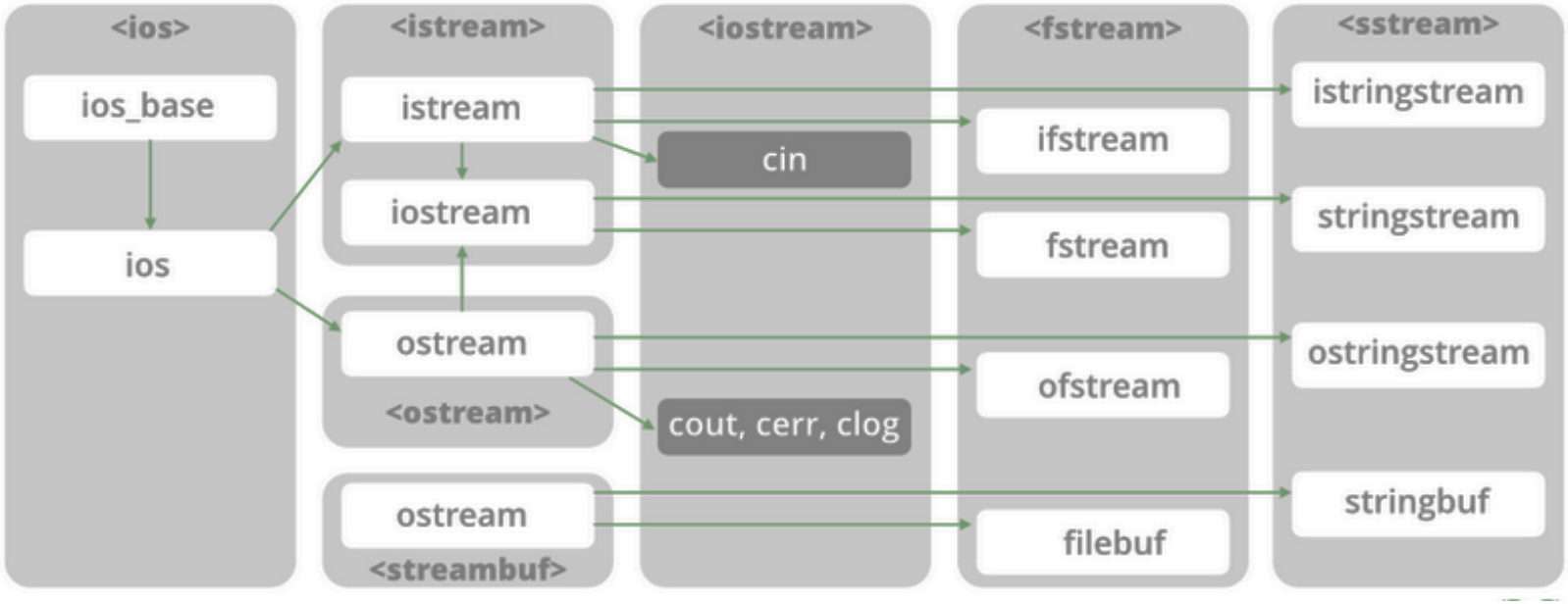
three of the most commonly included header files that allow us to work with stream IO.

Header File	Description
`iostream`	Provides definitions for formatted input and output from/to streams
`fstream`	Provides definitions for formatted input and output from/to file streams
`iomanip`	Provides definitions for manipulators used to format stream I/O

When we include these header files, we'll have access to many c++ classes that we can use for file IO.

Classes for Stream I/O in C++

- The I/O system of C++ contains a set of classes that define the file handling methods.
- These include ifstream, ofstream and fstream classes.
- These classes are derived from fstream and from the corresponding iostream class.



1. ios:-

- This class is the base class for other classes in this class hierarchy.
- This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

2. istream:-

- istream stands for input stream.
- This class is derived from the class ‘ios’.
- This class handles the input stream.
- The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions such as get(), getline() and read().

3. ostream:-

- ostream stands for output stream.
- This class is derived from the class ‘ios’.
- This class handles the output stream.
- The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as put() and write().

4. ifstream:-

- This class provides input operations.
- It contains open() function with default input mode.
- Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

5. ofstream:-

- This class provides output operations.
- It contains open() function with default output mode.
- Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.

6. fstream:-

- This class provides support for simultaneous input and output operations.
- Inherits all the functions from istream and ostream classes through iostream

File Handling

- File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk).

- To achieve file handling we need to follow the following steps:-
  - STEP 1-Naming a file
  - STEP 2-Opening a file
  - STEP 3-Writing data into the file
  - STEP 4-Reading data from the file
  - STEP 5-Closing a file.

C++ File Input/ Output

ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream, and ostream.

- The ifstream class provides high-level input operations from files. So if you want to read from a file, you can declare your object to be an ifstream object.
- The ofstream class provides high-level output operations from files. If you want to create a new file or write to a file, you can declare it as an object of ofstream.
- The fstream class provides high-level IO on file-based streams. Fstream is derived from ifstream and of osstream using multiple inheritance. If we need to do both input and output to a file at the same time, we can declare an fstream object to do it.

How to read a file?

- steps needed to read input from text files using c++ streams.

1. `#include <fstream>`
2. Declare an fstream or `ifstream` object
3. Connect it to a file on your file system (opens it for reading)
4. Read data from the file via the stream
5. Close the stream

Creating the Stream Object.

- we're creating an object named in_file, and we're using the fstream class as its type.
- Notice that I'm initializing the object as well.

fstream in_file {“file1.txt”}; //// similar to int my_int {5}

fstream in_file {“file1.txt”, ios::in};
ifstream in_file {“file1.txt”};

- In this example, we're using std::ios::in. This means to open the file in **input mode.**
- File will be opened if found.

Another way

- Many times we don't know the file name, and we have to get it from the user or from some other source at runtime. Once we have the file name, we can use it to open the corresponding file.

```
std::ifstream in_file;
std::string filename;
std::cin >> filename;  // get the file name

in_file.open(filename);
// or
in_file.open(filename, std::ios::binary);
```

open (filename, mode); // will discuss about mode later

is_open()

- Now that we attempted to create the stream and connect it to a file, we have to be sure that this was successful before we start reading from the file.

- To check if a file stream was successful in opening a file, you can do it by calling member is_open() with no arguments.

- This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (in_file.is_open()) { /* ok, proceed with output */ }
```

Example

```
if (in_file.is_open()) {
    // read from it
} else {
    // file could not be opened
    // does it exist?
    // should the program terminate?
}
```

Closing a file

- After input /output operations are finished on a file we shall close it so that its resources become available again.
- In order to do that we have to call the stream's member function .close().
- This member function takes no parameters, and what it does is flush the associated buffers and close the file:

```
in_file.close();
```

- Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.
- In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close().

Reading from a file

- There are many ways to read from a file.
- Using stream extraction operator(>>)
  - We can use the extraction operator for formatted read
  - Same way we used it with cin

```
int num {};
double total {};
std::string name {};

in_file >> num;
in_file >> total >> name;
```

File1.txt

```
100
255.67
Larry
```

- Using getline()
  - We can use `getline` to read the file one line at a time

```
std::string line{};

std::getline(in_file, line);
```

```
This is a line
```

How to read multiple lines?

```
fstream in_file{"file2.txt"};
string line {};

if (!in_file) // check if file is open
{
cout<<"file open error";
return 1;
}
while (!in_file.eof())  // while not at the end
{
    getline (in_file,line); // read a line
     cout<<line<<endl; // write a line
}
```

Method-1

```
//while (getline (in_file,line))
//{
//      cout<<line<<endl; // write a line
//}
in_file.close();
```

Method-2

- This program will read the contains from“file2.txt”.
- To do that you need to create a text file named file2 and store some info in it.

- Reading one character at a time

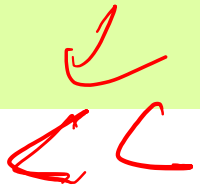
Method-3

```
char c;
while (in_file.get(c))
{
      cout<<c; // write a line
}
in_file.close();
system("pause");
return 0; }
```

Writing Text to a file

`fstream` and `ofstream` are commonly used for output files

1. `#include <fstream>`
2. Declare an `fstream` or `ofstream` object
3. Connect it to a file on your file system (opens it for writing)
4. Write data to the file via the stream
5. Close the stream



- By default, c++ will create the file if it does not exist.
- If it does exist, the contents of the file will be truncated or removed unless we state otherwise.

- Notice that I'm initializing the object as well.

```
fstream out_file {“file3.txt”, ios::out};  //// similar to int my_int {5}

ofstream in_file {“file1.txt”};
```

```
std::ofstream out_file {"../myfile.txt",
                        std::ios::out};

std::ofstream out_file {"../myfile.txt"};
```

- Open for writing in binary mode

```
std::ofstream out_file {"../myfile.txt",
                         std::ios::binary};
```

File is open?

Check if file opened successfully (is_open)

```
if (out_file.is_open()) {
    // read from it
} else {
    // file could not be created or opened
    // does it exist?
    // should the program terminate?
}
```

Closing a file

-

```
out_file.close();
```

Writing to file using <<

- We can use the insertion operator for formatted write
- Same way we used it with cout

```
int num {100};
double total {255.67};
std::string name {"Larry"};

out_file << num   << "\n"
         << total << "\n"
         << name  << std::endl;
```

```
100
255.67
Larry
```

Open a file

To open a file with a stream object we use the function open():

open (filename, mode);

in_file.open("File1.txt", ~~~)
mode

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 to any value, the initial position is the beginning of the file.

modes

ios::app

All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.

ios::trunc

If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (|).

For example:

Open the file example.bin in binary mode to add data we could do it by the following call to member function open():

```
ofstream myfile;

myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Default Mode

- Each one of the open() member functions of the 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

Example

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    fstream out_file;
    out_file.open("ramanuja.txt", ios::out );


  if (out_file.is_open())
  {
    out_file << "HELLO! EEC-101.\n";
    out_file << "WE ARE AT GB-202.\n";
    out_file.close();
  }
  else cout << "Unable to open file";

out_file.open("ramanuja.txt", ios::out|ios::app);
   if (out file.is open())
   if (out_file.is_open())
  {
    out_file << "Added this line without deleting previous";
    ;
    out_file.close();
  }
out_file.open("ramanuja.txt", ios::out|ios::trunc);

   if (out_file.is_open())
  {
    out_file << "Deleted all previous line and added this";
    ;
    out_file.close();
  }


  system("pause");
  return 0;
}
```

Copying files

```
// File copy using get/put
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream in_file {"file2.txt"};
    std::ofstream out_file{"poem.txt"};
    if (!in_file) {
        std::cerr << "Error opening input file" << std::endl;
        return 1;
    }
     if (!out_file) {
        std::cerr << "Error opening output file" << std::endl;
        return 1;
    }
    char c;
    while (in_file.get(c))
        out_file.put(c);

    std::cout << "File copied" << std::endl;
    in_file.close();
    out_file.close();
    return 0;
}
```

Buffers and Synchronization

- When we operate with file streams, these are associated with an internal buffer of type streambuf. **This buffer is a memory block that acts as an intermediary between the stream and the physical file.**

For example, with an ofstream, each time the member function put (which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.
When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called *synchronization* and takes place under any of the following circumstances:

1. **When the file is closed:** before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
2. **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
3. **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
4. **Explicitly, with member function sync():** Calling stream's member function sync(), which takes no parameters, causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.

state flags

To check the state of a stream (all of them return a bool value):

bad()	Returns true if a reading or writing operation fails. In the case that 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(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
eof()	Returns true if a file open for reading has reached the end.
good()	It is the most generic state flag: it 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 any of these functions use the function clear(), which takes no parameters.

Text files

- In opening mode of Text file streams we do not include the ios::binary flag
- Text files store text, and thus, all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.
- Data output operations on text files are performed in the same way we operated with cout:
- Data input from a file can also be performed in the same way that we did with cin:

File Pointers(get and put stream pointers)

- All input/output stream objects have, at least, one internal stream pointer

  - ✓ ifstream has a pointer known as the *get pointer* that points to the element to be read in the next input operation.
  - ✓ ofstream has a pointer known as the *put pointer* that points to the location where the next element has to be written.
  - ✓ fstream, inherits both, the get and the put pointers.

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

seekg() Moves **get pointer** (input) to a specified location

seekp() Moves **put pointer** (output) to a specified location

tellg() Gives the current position of the get pointer

tellp() Gives the current position of the put pointer

Example 1:

```
file_in.seekg(15);
```

(moves the file pointer to the byte number 15, i.e 16th byte as bytes in file are numbered from zero)

Example 2:

```
ofstream file_out;
file_out.open("hellobat.txt",ios::app)
int p = file_out.tellp()
```

(the output pointer is moved to the end of file hellobat.txt and the value **p** will represent the number of bytes in the file)

Specifying the offset

seekg*(offset, refposition);*

seekp*(offset, refposition*);

Offset Represents number of bytes the file pointer is moved from the location specified by the parameter ***refposition***

refposition

ios::beg	start of the file
ios::cur	current position of the pointer
ios::end	end of file

Example:

out_file.seekg(m, ios::cur); go forward m bytes from the current position

File Size

```
// obtaining file size
 #include <iostream>
 #include <fstream>
using namespace std;
int main () {
 long start,last;
ifstream myfile ("file2.txt");
start = myfile.tellg(); //Gives the current position of the get pointer
myfile.seekg (0, ios::end); //Moves get pointer (input) to a specified location
last = myfile.tellg();//Gives the current position of the get pointer
myfile.close();
cout << "size is: " << (last-start) << " bytes.\n";
return 0; }
```

Examples

```
/*******
Write a program to read a sequence of float numbers from an input file "input_nos.dat".
These float numbers can be either positive or negative.
The program should count total positive and negative numbers and find their averages separately.
Create an output file "output_nos.dat" which will display
count of positive numbers, the numbers and their average;
count of negative numbers, negative numbers and their average.
*********/

#include<iostream>
#include<fstream>

using namespace std;
int main()
{
    ifstream infile("input_nos.dat");
    ofstream outfile("output_nos.dat");
    float num,s1=0,s2=0;
    int p=0 , n=0;
    if(infile.fail())
    {
        cout<<"Input file opening failed!";
        exit(1); //terminate program
    }
if(outfile.fail())
{
    cout<<"Output file opening failed!";
    exit(1);
}

while(infile>>num)
{
    outfile<<num<<endl;
    if(num>=0)
    {
        s1+=num;
        p++;
    }
    if (num<0)
    {
        s2+=num;
        n++;
    }
}
outfile<<"Number of positive numbers: "<<p<<endl;
outfile<<"Average of all positive numbers: "<<s1/p<<endl;
outfile<<"Number of negative numbers: "<<n<<endl;
outfile<<"Average of all negative numbers: "<<s2/n<<endl;
infile.close();
outfile.close();    return 0;    }
```