Lab-3

Data Structures

File I/O in C++

Khalid Mengal

Contents

Input / Output

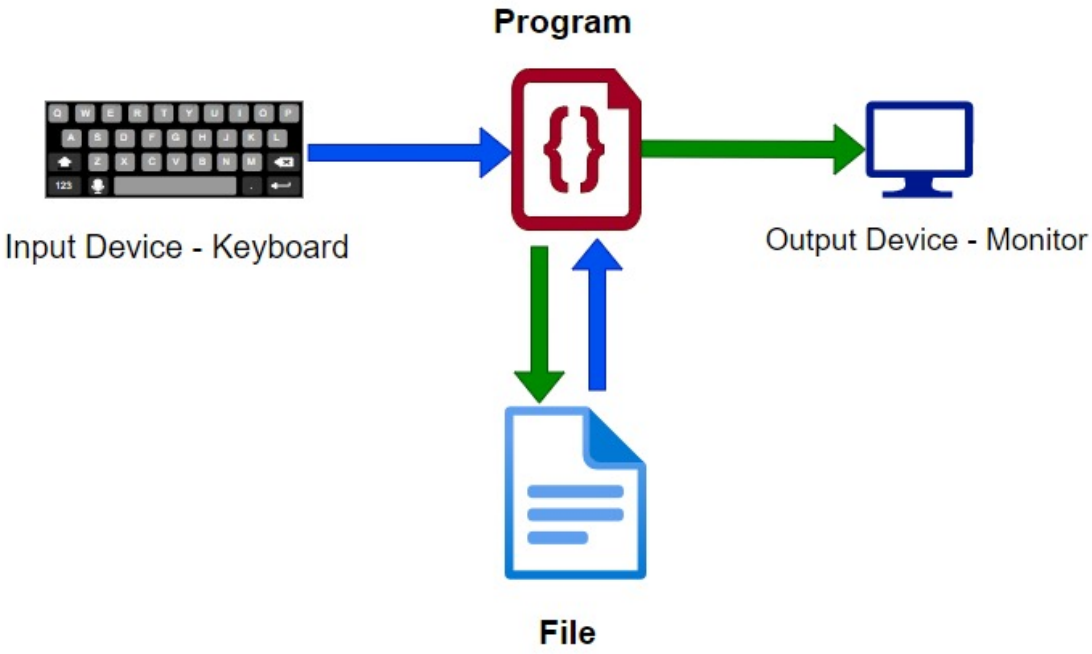
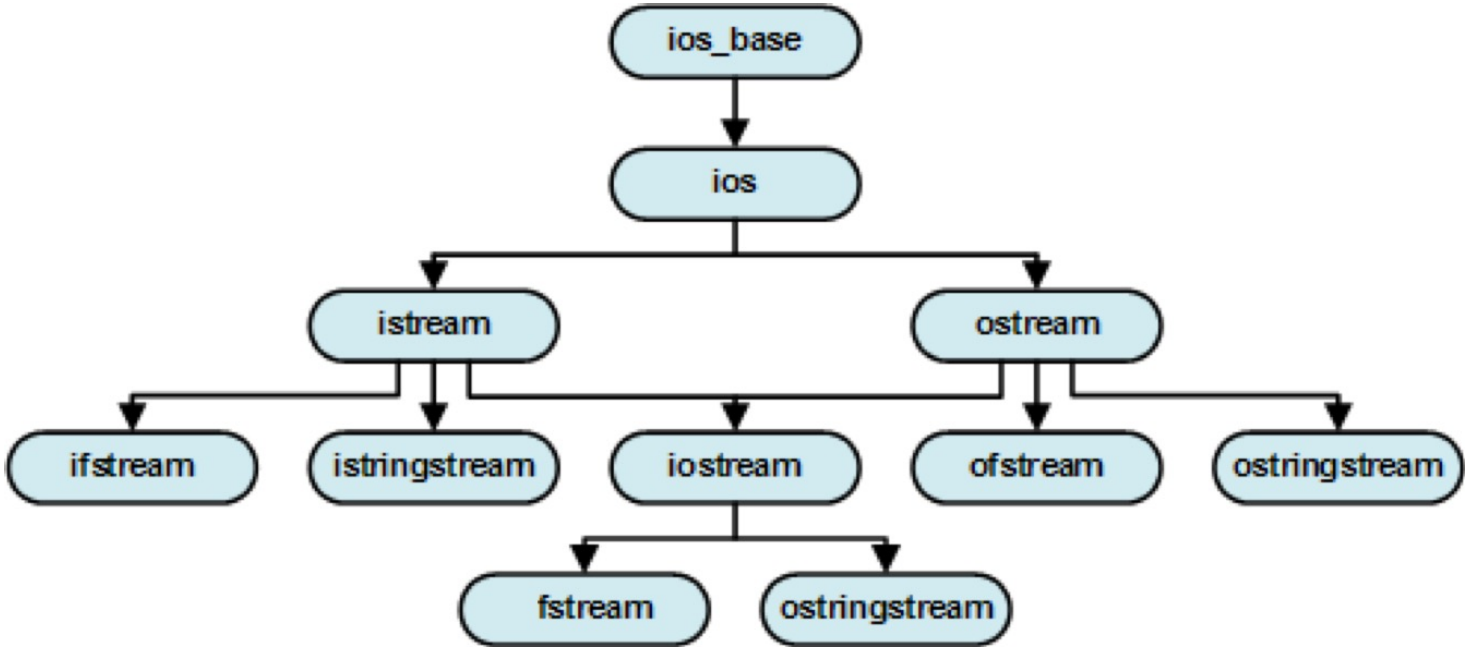


Image Source: https://linuxhint.com/cplusplus_read_write/

Introduction to File I/O in C++

- C++ provide set of classes and methods to perform file I/O
  - Streams defined in <**fstream**>
  - Operations e.g. **<<**, **>>**
  - Methods
    - close()
    - open()
    - read()
    - write()
    - seekg()
    - getg()
    - . .

C++ Stream hierarchy



Introduction to File I/O in C++

- File I/O interacts with the OS differently than standard I/O, we will use streams defined in **<fstream>** (rather than **<iostream>**)

```
#include <fstream>

using namespace std;

        …
```

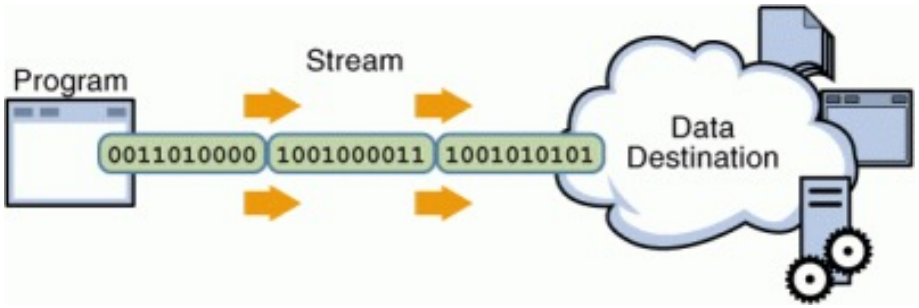


Image Source: https://letrungthang.wordpress.com/2011/01/17/cc-input-output-stream/

File I/O Streams

- Before you can begin performing file I/O, you must:
  1. **Create** a file stream object
  2. **Associate** that file stream object with a file (by "opening" the file)

```
ifstream fin("myFile.txt");        // Create a file input stream and associate
                                   //it with "myFile.txt"

ofstream fout;                     // Create a file output stream
...
fout.open("myFile.txt");           // Associate the stream with the file "myFile.txt"
```

File Open Modes

- Both the call to **open()** and the **constructor** method for initializing a file stream take *two* arguments:
  1. **const char* filename** Name of the file
  2. **openmode mode** Flags that determine the behavior of the file stream

- However, for each type of file stream (**ifstream**/ **ofstream**/**fstream**) there are default arguments provided for **openmode**
  - This is why the previous examples only gave the name of the file as arguments

File Open Modes

- The following flags are used to set the behavior of the file stream:

in	Open for reading operations (default for ***ifstream***)
out	Open for writing operations (default for ***ofstream***)
app	Start writing at end-of-file (***APP****end*) Seek to the end of the stream before each output operation.
ate	Start reading or writing at end-of-file (***AT***end)
binary	Open file in binary (not text) mode
trunc	Truncates the old file to zero if it already exists or creates a new file if it does not (default for ***ofstream***)

File Open Modes

- For example, if we want to open a file for input in binary mode, we could use any of the following notations:

- **ifstream fin;**

  fin.open("myFile.txt", ios::in | ios::binary);

  or

- **ifstream fin("myFile.txt", ios::in | ios::binary);**

  or

- **fstream fin("myFile.txt", ios::in | ios::binary);**

File Open Modes

- Verify the state of a file stream each time we attempt to associate it with a file
  - General structure:

```
ofstream fout("myFile.txt");
if (fout.is_open())
{
        // Perform file I/O
}
else
{
        // Error-related code
}
```

Performing File I/O

- I/O is performed in the exact same manner as it was for general I/O, using the **insertion** and **extraction** operators
- Output example:

```
ofstream fout("myFile.txt");
if (fout.is_open())
{
        fout << "Hello" << endl;
```

Overloaded insertion operator

Because file streams are based on general I/O streams, all of the same manipulators and functions can be used (such as **endl**)

Performing File I/O

- Input Example:

```
ifstream fin("myFile.txt");
if (fin.is_open())
{
    int x;
    fin >> x;                    // read an int from file and save it to variable x
    .
    sting str;
    fin>>str;                    // read a string from file and save it to str

    .
    char buffer[256];
    fin.getline(buffer, 256);    //read a line from file and save it char array
    ...
```

File input can be read in line-by line just as standard input can

Closing File Streams

- not necessary to explicitly close a file
- automatically closed when they go out of scope.
- It is a good practice to close the file explicitly using close() function.

```
int main()
{
   ofstream fout("myFile.txt");
   …
}
```

At this point, fout is no longer in-scope, so the file stream's association with "**myFile.txt**" will be terminated (the stream will be "closed") after which the stream object itself will be deallocated

Closing File Streams

- Reusing a File Stream:

```
ofstream fout ("myFile.txt");
if (fout.is_open())
{
    fout << "Hello" << endl;
    fout.close();                    // Close the file stream
}
...
fout.clear();                        // Reset the file stream's state
fout.open("myFile2.txt", ios_base::out | ios_base::app);
                                     // Reopen the file stream
if (fout.is_open())
{
        fout << "How are you today?" << endl;
        ...........
```

Binary File I/O

- Sometimes, there is no need for a human to directly read the contents of a file
- Binary file I/O, store and retrieve the **original binary representation** for data objects

Binary File I/O

- The advantages to binary file I/O include:
  - No need to convert between the internal representation and a character-based representation
    - Reduces the associated time to store/retrieve data
    - Possible conversion and round-off errors are avoided
  - Storing data objects takes less space

Binary File I/O

- Create a binary file output stream:

  ofstream fout ("myFile.txt", ios::binary);

- Create a binary file input stream:

  ifstream fin ("myFile2.txt", ios::binary);

Binary File I/O

- In order to perform binary I/O, we use the functions **write()** and **read()**, instead of the **insertion** and **extraction** operators
  - **Insertion operator(<<)** convert data objects into characters
  - **write()** function does a straight byte-by-byte copy
  - Same is true for **extraction operator(>>)** and **read()**)

Writing an Object to Disk

- We can also write C++ objects to a file

```
myObj obj;

ofstream fout ("myFile.txt", ios::binary);
 if(fout.is_open())
 {
        fout.write((char*) (&obj), sizeof(obj));
    ...............
```

Reading an Object from Disk

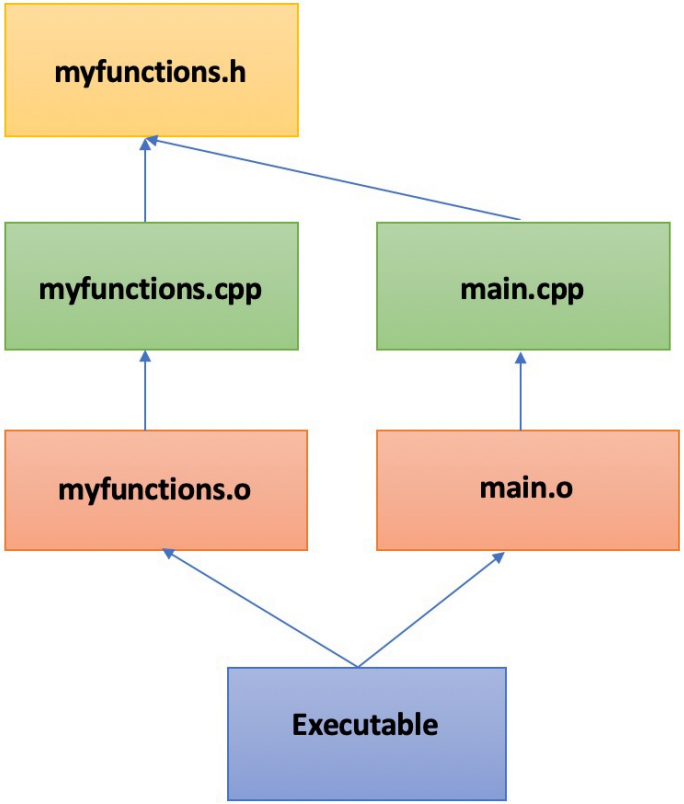
- Binary file input is performed in a similar manner:

```
myObj obj;
ifstream fin ("myFile.txt", ios::binary);
if(fin.is_open())
{
  fin.read((char*)(&obj), sizeof(obj));
  .......
```

Header Files in C++

- Contain declaration for Functions, structures, union, classes, constants etc.
- Can be imported/included in any other program
- Source file which include header file can access all declarations/definitions present in the header file
- Two types:
  - System Header Files: Comes with Compiler (e.g. iostream, cstdlib etc.)
  - User Header Files: Written by the programmer
- Usage:
  - #include<string> //System header files (from compilers include directory)
  - #include "filename" //User Header file (from current directory)

Header and cpp files



makefile

- makefile tells make utility how to compile and link a program
- A Simple makefile contains Rules:

```
target … : prerequisites …
        recipe
        …
        …
```

- Target: Usually the name of the file (generated by Program)
- Prerequisites: File(s) used as input to create the Target
- Recipe: Action(s) which are carried out when perquisites change

A Simple makefile

Makefile

```
output: main.o myfunctions.o
	g++ main.o myfunctions.o -o output
main.o: main.cpp
	g++ -c main.cpp
myfunctions.o:  myfunctions.cpp myfunctions.h
	g++ -c myfunctions.cpp
clean:
	rm *.o output
```

g++ -c Only run preprocess, compile, and assemble steps

The End

Exercise

- Write a C++ program that creates a file named test.txt and writes 100 random numbers on it.
- Write another program which reads contents from the file test.txt generated in previous exercise and prints it on screen.