Lab-8(File Input/output in C++) Data Structures

Khalid Mengal

Contents

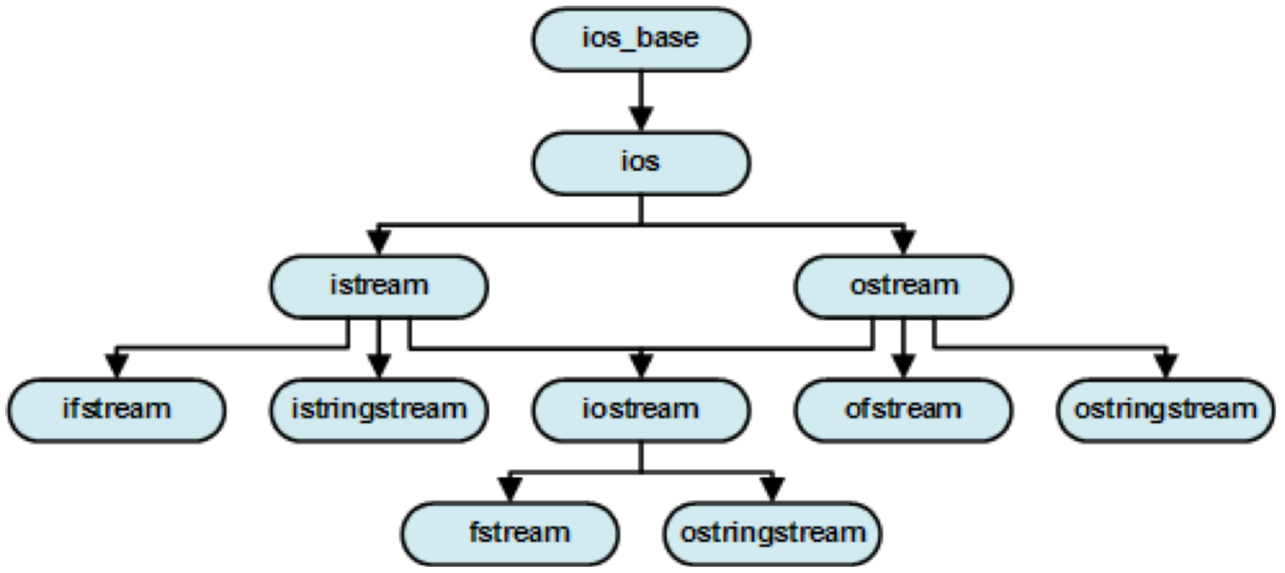
Introduction to File I/O in C++

- Main Memory (RAM) is **volatile**
- Information in RAM is lost if the power is interrupted
- We need a mechanism to store data more **permanently**
  - Store data on secondary storage e.g. disk

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;
        …
```

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	(truncate) Truncates the old file to zero if it already exists (and creates a new file if it does not)

File Open Modes

- Thus, if we wanted to open a file for input in binary mode, we could do it using any of the following methods:

- **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

- Default arguments provided:

Stream Type	**Default Arguments**
ifstream	ios_base::in
ofstream	ios_base::out | ios_base::trunc
fstream	ios_base::in | ios_base::out

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 an string from file and save 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("myFile.txt", ios_base::out | ios_base::app);
                                           // Reopen the file stream
if (fout.is_open())
{
         fout << "How are you today?" << endl;
         ...........
```

put() and **get()**

- The **put()** and **get()** member functions of **ostream** and **istream**, respectively can be used to output and input **single characters.**

```
string str = “NYUAD Abu Dhabi”;

ofstream outfile(“test.txt”);
 for(int i=0; i<str.size()); i++)
     outfile.put(str[i]);


 cout<<“File Written..”<<endl;
```

get() method

- Read the contents of file using get() function.

```
ifstream infile("text.txt");
char ch;
while(not infile.eof())
{
   infile.get(ch);
   cout<<ch;
}
```

Binary File I/O

- Often times, there is no need for a human to directly access 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 cam also write C++ objects to a file

```
myObj obj;

ofstream fout ("myFile.txt", ios::out |
                         ios::trunc | ios::binary);
 if(fout.is_open())
 {
        fout.write(reinterpret_cast<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::in | ios::binary);
if(fin.is_open())
{
  fin.read(reinterpret_cast<char *>(&obj); sizeof(obj));
  …….
```

Lab Exercise

- **Task 1:**
  - Write a C++ program that generates 100 random integer numbers (in the range of 0-1000) and write those numbers in a file called ***"file1.txt".*** File should contain single number per line.
- **Task 2:**
  - Write a C++ Program that opens and reads 100 integer numbers from a file called "***file1.txt***" and stores those numbers in an int array.
  - Add a function called "int **square(int)** to your program which calculates the square of a number provided to it as an argument. Your program should then write the square of 100 numbers into another file called ***"file2.txt"***.
- **Task 3:**
  - Create a function called "**sort(int array[], int size)**" that takes an int Array as an argument and sort it using any **sorting** algorithm. (e.g. selection sort, bubble sort, insertion sort). Write the contents of the sorted array into another file called ***"file3.txt".***

End