National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS212: Object Oriented Programming

Class: BSCS-5AB

Lab 11 : File Streams

Date: 3-06-2015

Time: 09:00 am – 12:00 pm / 2pm to 5pm

Instructor: Ms. Hirra Anwar

CLO-3: Develop programs using important C++ techniques, such as composition of objects, operator overloads, dynamic memory allocation, inheritance and polymorphism, file I/O, exception handling, templates, preprocessor directives, and basic data structure

Lab 11: File Streams

Introduction

This lab is related to reading and writing data to a file using file streams.

Objectives

- Learning about Sets
- Learning about Maps

Tools/Software Requirement

You will need Visual Studio.

Description

C++ provides the following classes to perform output and input of characters to/from files.

- 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. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.
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. An open file is represented within a program by a stream (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

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:

`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.

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

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

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).

For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

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

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

Is_Open():

To check if a file stream was successful opening a file, you can do it by calling to member is_open. 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.

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.

Random File Access

get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:

ifstream, like istream, keeps an internal *get position* with the location of the element to be read in the next input operation.

ofstream, like ostream, keeps an internal *put position* with the location where the next element has to be written.

Finally, fstream, keeps both, the *get* and the *put position*, like iostream.

These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the

following member functions:

tellg() and tellp()

These two member functions with no parameters return a value of the member type streampos, which is a type representing the current *get position* (in the case of tellg) or the *put position* (in the case of tellp).

seekg() and seekp()

These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:

seekg (position);
seekp (position);

Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.

The other form for these functions is:

seekg (offset, direction);
seekp (offset, direction);

Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter direction. offset is of type streamoff. And direction is of type seekdir, which is an *enumerated type* that determines the point from where offset is counted from, and that can take any of the following values:

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

The following example uses the member functions we have just seen to obtain the size of a file:

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin",
ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << "
bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

Binary files

For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions likegetline is not efficient, since we do not need to format any data and data is likely not formatted in lines.

File streams include two member functions specifically designed to read and write binary data sequentially: write andread. The first one (write) is a member function of ostream (inherited by ofstream). And read is a member function ofistream (inherited by ifstream). Objects of class fstream have both. Their prototypes are:

write (memory_block, size);
read (memory_block, size);

Where memory_block is of type char* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

```
// reading an entire binary file
```

```
the entire file content
```

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

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin",
ios::in|ios::binary|ios::ate);

    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in
memory";

    delete[] memblock;
    return 0;
}

```

Lab Task 1

Make sure you apply exception handling for all the questions below. If any program is unable to open a file or for any other exception, your code must inform the user accordingly.

1) Write a program which opens a file in reading and writing mode, takes input from a user and stores it in an array and writes that array to the file. It then reads information from the file and outputs it on the screen.
2) Make a class student that has functions for storing and reading information related to students into and from a file.
3) Write code for copying a binary file into another binary file. Open the two files in binary mode, one for reading and the other for writing.

Deliverables

Source code for Task 1 uploaded on LMS.