LAB 14: File Handling

Objective(s):

To Understand about:

1. Able to read from a file
2. Able to write in a file

CLOs: CLO1, CLO4

Introduction

The programs you have written so far require you to re-enter data each time the program runs. This is because the data stored in RAM disappears once the program stops running or the computer is shut down. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. The data can then be retrieved and used at a later time.

There are always three steps that must be taken when a file is used by a program:

1. The file must be *opened*. If the file does not yet exist, opening it means creating it.
2. Data is then saved to the file, read from the file, or both.
3. When the program is finished using the file, the file must be *closed*.

Setting up a Program for File Input/Output

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.

File Stream Data Type	Description
`ofstream`	Output file stream. This data type can be used to create files and write data to them. With the `ofstream` data type, data may only be copied from variables to the file, but not vice versa.
`ifstream`	Input file stream. This data type can be used to open existing files and read data from them into memory. With the `ifstream` data type, data may only be copied from the file into variables, not but vice versa.
`fstream`	File stream. This data type can be used to create files, write data to them, and read data from them. With the `fstream` data type, data may be copied from variables into a file, or from a file into variables.

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. Let's see an example:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

```
[file example.txt]
Writing this to a file.
```

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

But let's go step by step:

Open 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. An open file is represented within a program by a stream object (an instantiation 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 null-terminated character sequence of type const char * (the same type that string literals have) representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

os::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.
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, if we want to 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);
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

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 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 object 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 conducted 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.
To check if a file stream was successful opening a file, you can do it by calling to 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 (myfile.is_open()) { /* ok, proceed with output */ }

Closing a file

When we are finished with our input and output operations 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 to flush the associated buffers and close the file:
myfile.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.

Text files

Text file streams are those where we do not include the ios::binary flag in their opening mode. These files are designed to 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

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

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Data input from a file can also be performed in the same way that we did with cin:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using getline. The value returned by getline is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

Checking state flags
In addition to good(), which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a bool value):

bad()
Returns true if a reading or writing operation fails. For example 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 member functions we have just seen we can use the member function clear(), which takes no parameters.

get and put stream pointers
All i/o streams objects have, at least, one internal stream pointer:

ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.

ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written**.**

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

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

tellg() and tellp()
These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()
These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype 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 the same as the one returned by functions tellg and tellp: the member type pos_type, which is an integer value.

The other prototype for these functions is:

seekg (offset, direction);

seekp (offset, direction);

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction. offset is of the member type off_type, which is also an integer type. And direction is of type seekdir, which is an enumerated type (enum) 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 of the stream pointer
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 () {
  long begin,end;
  ifstream myfile ("example.txt");
  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.
```

C++ write() function

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.

Syntax of write() function

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

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.

C++ read() function

The read() function is used to read object (sequence of bytes) to the file.

Syntax of read() function

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

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.
The read() function returns NULL if no data read.

Lab Tasks:

Task 1:

Write a program, which asks the user to enter name of 10 employees, store thes names in file. Display total number of words in the file.

Task 2:

Write a program that calculates the balance of a savings account at the end of a period of time. It should ask the user for the annual interest rate, the starting balance, and the number of months that have passed since the account was established. A loop should then iterate once for every month, performing the following:

- Ask the user for the amount deposited into the account during the month. (Do not accept negative numbers.) This amount should be added to the balance.
- Ask the user for the amount withdrawn from the account during the month. (Do not accept negative numbers.) This amount should be subtracted from the balance.
- Calculate the monthly interest. The monthly interest rate is the annual interest rate divided by twelve. Multiply the monthly interest rate by the balance, and add the result to the balance.

After the last iteration, the program should display the ending balance, the total amount of deposits, the total amount of withdrawals, and the total interest earned.

NOTE: If a negative balance is calculated at any point, a message should be displayed indicating the account has been closed and the loop should terminate.