C++ Files and Streams

So far, we have been using the iostream standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This lab will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types.

S/N	Data Type	Description
1	ofstream	This data type represents the output file stream and is used to create files and to write information to files.
2	ifstream	This data type represents the input file stream and is used to read information from files.
3	fstream	This data type represents the file stream generally, and has the capabilities of both **ofstream** and **ifstream** which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Creating a file

To create a file, use either the **ofstream** or **fstream** type, and specify the name of the file.

Implement the following in a new project, within a file titled **FileCreation.cpp**, and then build and execute your program

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

int main() {
  // Create and open a text file
  ofstream MyFile("exampleFile.txt");

  // Close the file
  MyFile.close();

  return 0;
}
```

In the code above, the argument “exampleFile.txt” specifies the name and location of the file to be created. In this instance, a text file with the name **exampleFile** will be created within the same directory as your C++ project, check your project’s directory for this file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. An **ifstream** object is used to open a file for reading purpose only. In this regard, an open file is represented within a program by a stream and any input or output operation performed on this stream object will be applied to the physical file associated to it.

The following is the standard syntax for the **open()** function, which is a function of **fstream**, **ifstream**, and **ofstream** objects.

```
void open(const char *filename, ios::openmode mode);
```

In the code above, the first parameter specifies the name and location of the file to be opened and the second parameter of the **open()** function defines the mode in which the file should be opened.

S/N	Mode Flag	Description
1	ios::app	Append mode. All output operations are performed at the end of the file, appending the content to the current content of the file.
2	ios::ate	Open a file for output and move the read/write control to the end of the file. In other words, 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.
3	ios::in	Open for input(reading) operations.
4	ios::out	Open for output(writing) operations.
5	ios::trunc	If the file already exists, its contents will be truncated before opening the file. In other words, if the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.
6	ios::binary	Open in binary mode.

You can combine two or more of these values by bitwise operator OR (|) operator. For example if you want to open a file in write mode and want to truncate it in case that already exists, the following will be the syntax.

```
ofstream outfile;

outfile.open("file.txt", ios::out | ios::trunc);
```

Similarly, you can open a file for reading and writing purpose as follows

```
fstream  afile;

afile.open("file.text", ios::out | ios::in);
```

Each of the open functions of classes **ofstream**, **ifstream** and **fstream** has a default mode that is used if the file is opened without a second argument:

S/N	Data Type	Default mode parameter
1	ofstream	ios::out
2	ifstream	ios::in
3	fstream	ios::in | ios::out

For **ifstream** and **ofstream** types, **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 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).

To check if a file stream was successful opening a file, you can do it by calling the **is_open** function. This 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 a C++ program terminates it automatically flushes all the streams, releases all the allocated memory and closes all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination so that the operating system is notified and its resources become available again.

The following is the standard syntax for the **close()** function, which is a function of **fstream**, **ifstream**, and **ofstream** objects.

```
void close();
```

```
myfile.close();
```

Once this 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.

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Text Files

Text file streams are those where **the ios::binary** flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Implement the following in a new project, within a file titled **WriteToTextFile.cpp**, and then build and execute your program.

```
// 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;
}
```

Implement the following in a new project, within a file titled **ReadATextFile.cpp**, and then build and execute your program.

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

int main () {

  string line;

  ofstream myfile("example.txt");

  if(myfile.is_open())
  {
    while(getline(myfile,line))
    {
      cout << line << '\n';
    }
    myfile.close();
  }

  else
  {
     cout << "Unable to open file";
  }

  return 0;
}
```

In the code above, 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.

Implement the following in a new project, within a file titled **FileReadAndWrite**.cpp, and then build and execute your program.

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

int main () {
   char data[100];

   // open a text file in write mode.
   ofstream outfile;
   outfile.open("afile.txt");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the text file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the text file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a text file in read mode.
   ifstream infile;
   infile.open("afile.txt");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

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 functions:

tellg() and **tellp()**

These two functions with no parameters return a value of the 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);

seeks(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:

S/N	Parameter type	Description
1	ios::beg	offset counted from the beginning of the stream
2	ios::cur	offset counted from the current position
3	ios::end	offset counted from the end of the stream

Implement the following in a new project, within a file titled **SizeOfAFile.cpp**, and then build and execute your program.

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

int main () {
  streampos begin;
  streampos 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;
}
```

In the code above, the type used for **begin** and **end** is **streampos**. **streampos** is a specific type used for buffer and file positioning and is the type returned by **tellg()**. Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.

Binary Files

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

File streams include two functions specifically designed to read and write binary data sequentially: **write** and **read**. The first one (**write**) is a function of **ostream** (inherited by ofstream). And **read** is a function of **istream** (inherited by ifstream). Objects of type **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.

Implement the following in a new project, within a file titled **BinaryFile.cpp**, and then build and execute your program.

```
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

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

  ifstream file("example.bin", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    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;
  }
  else
  {
    cout << "Unable to open file";
  }

  return 0;
}
```

In the code above, the entire file is read and stored in a memory block as follows:

First, the file is opened with the **ios::ate** flag, which means that the get pointer will be positioned at the end of the file. This way, when we call **tellg()**, we will directly obtain the size of the file.

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file, as follows:

```
memblock = new char[size];
```

Right after that, we proceed to set the get position at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close as follows:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

At this point we could operate with the data obtained from the file. But our program simply announces that the content of the file is in memory and then finishes.

References

- https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm
- https://cplusplus.com/doc/tutorial/files/
- https://www.w3schools.com/cpp/cpp_files.asp