C++ Binary File I/O

C++ file input and output are typically achieved by using an object of one of the following classes:

- ifstream for reading input only.
- ofstream for writing output only.
- fstream for reading and writing from/to one file.

To define any of the above classes, you need to include <fstream> package, which includes those classes.

Caution:

For binary file I/O you **do not** use the conventional text-oriented << and >> operators!

It can be done, but that is an advanced topic. We will discuss the easy way here.

Basic Model for File I/O

- The **file stream classes** are simply be viewed as a stream or array of uninterpreted bytes. That is, it is like an “array” of bytes stored and indexed from *zero* to *len-1*, where *len* is the total number of bytes in the entire file.

Each open file has two “positions” associated with it:

1. The **current reading position**, which is the index of the next byte that will be read from the file. It simply points to the next character that the basic get method will return.
2. The **current writing position**, which is the index of the byte location where the next output byte will be placed. It simply points to the location where the basic put method will place byte(s).

.

Repositioning with the **get** and **put** stream pointers

- All i/o streams objects have, at least, one internal stream pointer:
- For input streams like, ifstream and istream, you use **get** to move the pointer that points to the element to be read in the next input operation.
- For output streams, like ofstream and ostream, you use the **put** pointer that points to the location where the next element has to be written.
- Recall that 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
- They 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()

- You may change the position of the **get** and **put** stream pointers using seekg() and seekp() functions.
- Both functions are overloaded with two different prototypes. The first prototype is:

  ```
  seekg ( position );
  seekp ( position );
  ```

  - With 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;
}
```

Opening a File

A file stream object can be opened in one of two ways.

ifstream myFile (“data.bin”, **ios::in** | **ios::binary**);

Or, after a file stream object has been declared, you can call its open method:

ifstream myFile;
...
myFile.open ("data.bin", **ios::in** | **ios::binary**);

Either approach will work with an ifstream, an ofstream, or an fstream object.

For example, you may declare an output file as following:

```
  ofstream myFile;
...
```

```
myFile.open ("data2.bin", ios::out | ios::binary);
```

When manipulating text files, one omits the second parameter (the i/o mode parameter). However, in order to manipulate binary files, you should always specify the i/o mode, including ios::binary as one of the mode flags. For read/write access to a file, use an fstream:

```
fstream myFile;
myFile.open ("data3.bin", ios::in | ios::out | ios::binary);
```

Reading and Writing to a Binary File

- Read and Write are special functions for binary files.
    - o write is a member function of ostream inherited by ofstream.
    - o read is a member function of istream that is inherited by ifstream.

- Objects of class fstream have both members. Their prototypes are:
    - o write (memory_block, size)
    - o read (memory_block, size);

    - o Where memory_block is of type “pointer to char” (char*) (for the time consider this as an array), 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.
    - o The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

ifstream::pos_type size;
char * memblock;

int main () {
  ifstream file ("example.txt", 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 complete file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

In this example the entire file is read and stored in a memory block. Let's examine how this is done:
First, the file is open 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 to member tellg(), we will directly obtain the size of the file. Notice the type we have used to declare variable size:

```
ifstream::pos_type size;
```

ifstream::pos_type is a specific type used for buffer and file positioning and is the type returned by file.tellg(). This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB we could use int:

```
int size;
size = (int) file.tellg();
```

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

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

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

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

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

Closing a File

For all file stream objects, use:

```
myFile.close();
```

Example 1: This program reads the whole file as binary and dumps it to screen. Hence, make sure that your input file is a text file. Otherwise, you will not see meaning full characters on the screen.

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

int block =100;

int main () {

int size, iteration, reminder;
char memblock[101];  // character array
  ifstream file ("example.txt", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
      file.seekg (0, ios::end);  //seek to the end
    size = file.tellg();       //tell the location of get pointer, ie.
Find the size

    file.seekg (0, ios::beg); // put the get pointer to the beging of
the file.
      iteration=  size /block;
      reminder = size % block;

      while ( iteration > 0){
            file.read (memblock, block);  // read the whole file.
        memblock[block]='\0';  // append a null character to the end.
                            // This makes the char array a C string.
                            // If you do not, some garbage characters
may be printed.
            cout << memblock ;
            //memblock[0]='\0';
            //cout << endl<<endl;
            iteration--;
       } //end while
      if (reminder > 0) {
            //memblock[0]='\0';
            file.read (memblock, reminder);  // read the whole file.
        memblock[reminder]='\0';  // append a null character to the
end.
                            // This makes the char array a C string.
                            // If you do not, some garbage characters
may be printed.
            cout << memblock ;
      }//end if
      cout <<endl;
      cout << "Done writing the file to screen. Good Bye \n";


   }
  else cout << "Unable to open file" << endl;
  return 0;
}
```

Example 2; Reading the whole file and displaying on terminal at once. Again keep in mind that since you are displaying on terminal, you should input a text file.

```
      // reading a complete binary file
      #include <iostream>
      #include <fstream>
      using namespace std;

      int size;
      char *memblock;  // pointer to character

      int main () {
       ifstream file ("example.txt", ios::in|ios::binary|ios::ate);
       if (file.is_open())
       {
      file.seekg (0, ios::end);  //seek to the end
        size = file.tellg();       //tell the location of get pointer,
    ie. size
        memblock = new char [size+1];  // get size +1 charter space. +1
    space is for '\0'
        file.seekg (0, ios::beg); // put the get pointer to the
    begging.
        file.read (memblock, size);  // read the whole file.
        file.close();
        memblock[size]='\0';  // append a null character to the end.
                            // This makes the char array a C string.
                            // If you do not, some garbage characters
    may be printed.
        cout << "the complete file content is in memory. Here is what I
    read\n";
      cout << memblock << endl;

        delete[] memblock;
       }
       else cout << "Unable to open file";
       return 0;
    }
```

Exercise:

1) Write a program that will take an executable file, and duplicate it. In addition your program should display the size of the file on the terminal.
2) Modify the program given in example 1 such that it only prints approximately 50% of the file. It should skip 25% of the file at the beginning and 25% at the end.