File Access

Data stored in variables that you create in your program are stored in volatile memory. This means that when the program terminates all that data will be gone. In order to have persistent data between program runs, you need to save your data to non-volatile storage such as the disk or flash drive. There are two ways for storing your data to non-volatile storage.

1) You can write all your data to non-volatile storage right before the program exits, and when you run your program again, you load all the data back into memory from the non-volatile storage. This is best done using **sequential file access**. With this method you need to keep all your data in memory and changes to the data during the program run are only done in memory.

2) Every time you make changes to the data in memory, you will immediately write it out to non-volatile storage. This is best done using **random file access**. This way you don't need to keep all your data in memory, only those that you will be changing.

1. Sequential File Access (12.1)

For a sequential file, you can only write/read data to/from a file sequentially from the beginning to the end.

The following example shows how to write and read data to and from a sequential file on the disk.

```
/* Example of sequential file access
 * with writing and reading of a text file.
 *
 * Enoch Hwang 2010
 */
#include <iostream>
#include <fstream>
using namespace std;

int main(){
    ////////////////// prepare for writing ///////////////////////
    ofstream outFile("test.txt");   // open the text file for writing
    if(!outFile) {      // test to see if the file was opened successfully
      cout << "Cannot open file for writing.\n";
      return 0;
    }

    outFile << 10 << " " << 3.1415 << "\n";      // write to the file
    outFile << "This is a text line." << "\n";   // write again to the file

    int account = 12;
    char name[80] = "This is the second line";
    double balance = 123.45;
    // write three more things to the file
    outFile << account << " " << name << " " << balance << endl;
    outFile.close();    // close the file
    ////////////////// prepare for reading //////////////////////
    char buffer[256];   // buffer for reading each line from the file

    ifstream inFile("test.txt");     // open the text file for reading
    if (!inFile){
      cout << "Error opening file for reading.\n";
      return 0;
    }
    // read content of file
    while (!inFile.eof()){           // while not end-of-file
      inFile.getline (buffer,256);   // read one line  of 256 characters
      cout << buffer << endl;
    }
    inFile.close();      // close the file
    return 0;

}
```

Sample output

```
10 3.1415
This is a text line.
12 This is the second line 123.45
```

2. Access modes to open a file

In the above example we used the following to open a file for writing

```
    ofstream outFile("test.txt");    // open the text file for writing
```

and the following for reading

```
    ifstream inFile("test.txt");     // open the text file for reading
```

For **ofstream** the file is opened for output only. Data may be written to the file, but not read from the file. If the file does not exist, it is created. If the file already exists, its contents are deleted.

For **ifstream** the file is opened for input only. Data may be read from the file, but not written to it. The file's contents will be read from the beginning. If the file does not exist, the open function fails.

The following shows another way to open a file for writing using `fstream`. The access mode flag `ios::out` means to open file for writing.

```
   fstream dataFile;                          // create file variable
   dataFile.open("test.txt", ios::out);       // open file for output mode
```

Other access mode flags

`ios::out`   Output mode. The file's contents will be deleted if it already exists.

`ios::in`	Input mode. Data will be read from the file. If the file does not exist, it will not be created, and the open will fail.
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. If the file does not exist, it will create it.
`ios::ate`	If the file already exists, it will go directly to the end of it. Output may be written anywhere in the file.
`ios::binary`	Binary mode. Data are written to or read from using pure binary format. The default mode is text mode.

You can combine one or more flags using the or | operator.

```
   fstream dataFile;
   dataFile.open("AddressBook.dat", ios::in | ios::out | ios::binary);
```

or

```
   fstream dataFile("AddressBook.dat", ios::in | ios::out | ios::binary);
```

3. getline function

getline gets an entire line up to a delimiter character. The default delimiter character, if one is not given is \n which is the newline character. If you want to use another delimiter character you can specify it in the third argument. The following will read from dataFile everything up to the & and store the characters in myString.

```
#include <string>   // needed for getline
using namespace std;

string myString;
getline(dataFile, myString, '&');
```

4. get and put member functions

The **get** and **put** functions read and write one character at a time.

```
   dataFile.get(ch);      // read one character from dataFile into ch
   dataFile.put(ch);      // write the character in ch to dataFile
```

5. write and read member functions (12.7 and 12.8)

The **write** function writes binary data to a file, and the **read** function reads binary data from a file. The syntax is

```
   dataFile.write(address, size);

   dataFile.read(address, size);
```

where

- *dataFile* is the name of a file stream object.
- *address* is the starting address of a block of memory to be written to the file. It is expected to be a pointer to a char.
- *size* is the number of bytes of memory to write.

For the *address*, you'll need to use the `reinterpret_cast<char *>` to convert the pointer for your data type to a pointer to a char as shown next

```
   dataFile.write(reinterpret_cast<char*>(&person), sizeof(person));
```

The `&person` gives you a pointer to a person structure and the `reinterpret_cast<char *>` converts the person structure pointer to a pointer of char.

For example

```
struct Info {
   char name[20];
   int age;
   char email[30];
};

   Info person;
   fstream dataFile("AddressBook.dat", ios::in | ios::out | ios::binary);

   dataFile.write(reinterpret_cast<char*>(&person), sizeof(person));

   dataFile.read(reinterpret_cast<char*>(&person), sizeof(person));
```

Note that in the record structure you should not use any dynamic data types such as the string data type because the size can change, and if the size changes then you wouldn't be able to calculate the starting address of the next record. That's why we need to use a char array instead which has a fixed size.

6. *Random File Access* *(12.9)*

For a random-access file, you can write/read data to/from anywhere in the file by jumping back and forth in the file.

The **seekp** and **seekg** functions move the file access pointer to a certain byte position in the file for the subsequent file access function call (**get**, **put**, **read** or **write**). You use **seekp** before a **put** or **write**, and **seekg** before a **get** or **read**.

```
   dataFile.seekp(20, ios::beg);       // move to 20 bytes from the beginning
   dataFile.write(reinterpret_cast<char*>(&person), sizeof(person));

   dataFile.seekg(-34, ios::end);     // move 34 bytes back from the end
   dataFile.read(reinterpret_cast<char*>(&person), sizeof(person));

   dataFile.seekp(15, ios::cur);     // move 15 bytes from the current position
```

The **tellp** and **tellg** functions return the current byte position that the file access pointer is at. The following example shows how to find out the number of bytes that a file contains.

```
   dataFile.seekg(0, ios::end);     // move the file pointer to the end of the file
   int numBytes = dataFile.tellg(); // get the byte position of the end
   cout << "The file has " << numBytes << " bytes";
```

Note that if you move the file pointer past the end of file then the file pointer will always stay at –1 for subsequent seeks, i.e., subsequent seeks will not move the file pointer anymore. You'll need to close the file and open it again to make the seek work again.

Example

```
// Random file access
#include <iostream>
#include <fstream>
#include <string>   // needed for getline
#include <cstring>  // needed for strcpy
using namespace std;

struct Info {
   char name[20];
   int age;
   char email[30];
};

int main() {
   Info person;
   fstream dataFile;

   dataFile.open("AddressBook.dat", ios::in | ios::out | ios::binary);
   if (!dataFile) { // file doesn't exist so create new file
      dataFile.open("AddressBook.dat", ios::in | ios::out | ios::binary |
         ios::trunc);
   }

   // input person info
   string input;
   cout << "Enter name? ";
   getline(cin, input);
   // convert string to c-string
   strcpy(person.name, input.c_str());       // use this in replit
   //strcpy_s(person.name, input.c_str());   // use this in Visual Studio
   cout << "Enter age? ";
   getline(cin, input);
   person.age = stoi(input);
   cout << "Enter email? ";
   getline(cin, input);
   // convert string to c-string
   strcpy(person.email, input.c_str());      // use this in replit
   //strcpy_s(person.email, input.c_str());  // use this in Visual Studio

   // write record 3
   int record_number = 3;
   dataFile.seekp(record_number * sizeof(person), ios::beg);
   dataFile.write(reinterpret_cast<char*>(&person), sizeof(person));

   // read record 3
   record_number = 3;
   dataFile.seekg(record_number * sizeof(person), ios::beg);
   dataFile.read(reinterpret_cast<char*>(&person), sizeof(person));

   cout << "Name:  " << person.name << endl;
   cout << "Age:   " << person.age<< endl;
   cout << "Email: " << person.email << endl;

}
```

7. *Exercises* (Problems with an asterisk are more difficult)

1. Write a program to open a sequential file that will append a line of text to the end of the file each time that the program is run. The program first asks the user to enter a line of text. This line is then added (appended) to the end of the existing file, so the file gets longer and longer. Run the program five or more times and see that the file has five or more lines.

2. * Write a program to make a copy of the file created in question 1. You will open the file created in question 1 for reading and open another file for writing. Read the lines from the first file and write it into the second file.

3. Implement the example random access file program in section 6.

4. * Continuing from the random access file program, add a menu for the user to select from one of five options: 1) to add a new record to a given record number; 2) to print out the information for a record for a given record number; 3) to change the information for a given record number; 4) to delete a record for a given record number; and 5) to exit the program. For options 1, 2, 3 and 4, the program needs to ask the user to enter a record number. For options 1 and 3, the program will further ask the user for the new information for the record.