CS2024 – C++ PROGRAMMING

Lecture #16: Files

C++: How to Program – Chapter 14

FILES

- A file is a *sequence of bytes* that is stored on a secondary storage device:
  - Hard disk
  - Flash memory
  - CD, DVD, etc.
- Reading and writing to files can be accomplished using the same operators and member functions that we use when using `cin` and `cout`
- Additional methods are available that are specific to files

WRITING TO A FILE

- We use the data type ofstream to open a file for writing:

```
// include the necessary header files
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc,char *argv[])
{
  ofstream out("myFile",ios::out);   // open file for writing
  if (out.is_open()) {               // Is file open?
    out << "Hello world!" << endl;
  }
  return 0;
}
```

OFSTREAM

```
ofstream out("myFile",ios::out);   // open file for writing
if (out.is_open()) {               // Is file open?
  out << "Hello world!" << endl;
}
```

- The file is opened when the constructor for `ofstream` is called.
  - First argument is the name of the file to open
  - Second argument specifies the "mode":
    - `ios::out` – open file for writing, overwrite existing file
    - `ios::app` – open file for writing, append to existing file
- If the file cannot be opened, `is_open()` returns `false`
  - Make sure file is open before you try to write to it!

DEMONSTRATION #1

Ofstream – writing to a file

OFSTREAM

```
ofstream out("myFile");   // ios::out is default
// Evaluate stream as boolean
if (out) {                // If file is open, out is "true"
  out << "Hello world!" << endl;
}
```

- The file is opened when the constructor for `ofstream` is called.
  - If no "mode" is specified, `ios::out` is assumed
- Instead of using `is_open()`…
  - You can just evaluate the stream variable as a boolean.
  - If the file opened successfully, the stream will evaluate to `true`

DEMONSTRATION #2

Ofstream – default value and shorthand!

READING DATA FROM A FILE

- We use the data type `ifstream` to open a file for reading:

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

int main(int argc,char *argv[])
{
  ifstream in("myFile",ios::in);      // open file for reading
  if (in.is_open()) {                 // Is file open?
    string str;
    in >> str;                        // read string into str
  }
  return 0;
}
```

IFSTREAM

```
ifstream in("myFile",ios::in);      // open file for reading
if (in.is_open()) {                 // Is file open?
   string str;
   in >> str;                        // read string into str
}
```

- The file is opened when the constructor for `ifstream` is called.
  - First argument is the name of the file to open
  - Second argument specifies the "mode":
    - `ios::in`– open file for reading
    - Output operations/modes are supported (advanced topic)
- If the file cannot be opened, `is_open()` returns `false`
  - Make sure file is open before you try to read from it!

IFSTREAM

```
ifstream in("myFile");      // ios::in is default mode
if (in) {                   // Is file open?
   string str;
   in >> str;                          // read string into str
}
```

- The file is opened when the constructor for `ifstream` is called.
  - If no mode is specified, `ios::in` is the default
- Instead of using `is_open()` …
  - You can evaluate stream varaible as a boolean
  - If the file opened successfully, the stream will evaluate to `true`

DEMONSTRATION #3

Reading from files (ifstream)

SEQUENTIAL FILES

- So far the demos we have been doing involve what is called a *sequential file*
- Consider the following code that creates and writes to an “ages” file
- Each “record” has two fields – first name and age

```
int main(int argc,char *argv[])
{
  ofstream out(“ages.txt”,ios::out); // open file for writing
  if (out.is_open()) {               // Is file open?
    out << “Christopher,20” << endl;
    out << “Alexander,17” << endl;
    out << ”Nicholas,14” << endl;
  }
  return 0;
}
```

SEQUENTIAL FILES

```
out << “Christopher,20” << endl;
out << “Alexander,17” << endl;
out << ”Nicholas,14” << endl;
```

- Suppose I wanted to open this file and then read the third record
- Each “record” in this file is a different size
- When reading data from the file we’d need to look for a newline as the record separator
- We can’t open the file and jump to a given spot because we don’t know how long each record is
- We’d have to read through the file *sequentially* to get to the third record

SEQUENTIAL FILES

- What makes a file *sequential* is not anything syntactical; rather it is how you choose to format what you write to a file.
- In fact, there is no structure imposed on data written to a file
  - It's just a sequence of bytes, remember?
- You can have any number of formats that you employ when saving data to a file…

SEQUENTIAL FILES

- Consider a file where every “field” is finished with a newline and every “record” has three fields
- File contents:

```
Christopher
20
6075557586
Alexander
17
2075553995
Nicholas
14
6075551433
EOF
```

SEQUENTIAL FILES

- Or a file where each record is preceded by its length in bytes
- File contents:

```
25
Christopher,20,6075557586
23
Alexander,17,2075553995
22
Nicholas,14,6075551433
EOF
```

SEQUENTIAL FILES

- In both of these cases it would be difficult to update the file.
- We'd have to copy the file to a new file and add/remove full lines as needed
- This being said, there are still methods available for both `ifstream` and `ofstream` that let us "move around" the file
- Some files can even be open for read and write at the same time.
- In these cases the stream object has two member variables that note where the next write operation would insert data in the file and where the next read operation would return data from.

SEQUENTIAL FILES

- `tellg()`
  - Stands for “seek get”
  - Returns the offset from the beginning of the file where the next read operation will read from.
- `tellp()`
  - Stands for “seek put”
  - Returns the offset from the beginning of the file where the next write operation will put data.
- `seekg(int)` – sets the “get” offset
- `seekp(int)` – sets the “put” offset

DEMONSTRATION #4

Seek/tell in sequential files

RANDOM ACCESS FILES

- A *random access file* is designed to allow you to open and access ”a random record” comparatively easily
- All “records” in this case are set to be the same size
- This lets us navigate the file more easily
  - If we want to access the nth record, we use tellg() to go to the n*record_size offset in the file
- We can also update the file in place
  - Our records have “fields” that are defined to have a maximum length and are ”padded out” if the data stored is smaller than the maximum length
  - So we can write into the file and not corrupt anything

RANDOM ACCESS FILES

- Let's go back to our age "database"
- We had three fields:
  - Name, age, phone
- Let's assign maximum lengths to these fields:
  - Name(20)
  - Age(3)
  - Phone(10)
- Since each field appear in the data file with a newline after it, our record will have 20+3+10+3 = 36 bytes.

RANDOM ACCESS FILES

- This means that we should be able to read a given record much more quickly if we know the ”record number”
- Just set the seek get offset to:
  - recordNumber * 36
- Again, must make sure data file has fields padded out so that all records are the same size:

DEMONSTRATION #5

Random Access File

UPDATING RANDOM ACCESS FILE

- Suppose we want to programmatically change a value in the file associated with a given record.
- We would like to ask the user what record to modify (for a given field) and then ask them if they want to change the data to a new value
- This entails being able to read and write to the same file
- We can accomplish this with an `fstream` variable

```
// Open file for reading and writing with the help of some
// new syntax!
fstream file("ages.dat", ios::in | ios::out);
```

UPDATING RANDOM ACCESS FILE

```
// Open file for reading and writing with the help of some
// new syntax!
fstream file("ages.dat", ios::in | ios::out);
```

- The expression ios::in | ios::out is a *bitmask*
- We will cover these towards the end of the semester but for now just understand that it tells the `fstream` object that you want to open the file for both reading and writing at the same time.

UPDATING RANDOM ACCESS FILE

- The general idea is that we will prompt the user for the “record number” to update and ask for a new age to enter.
  - We’ll advance to that spot in the file
  - We’ll make sure that we haven’t gone past the end of the file (invalid record number)
  - We’ll prompt the user for a new age
  - Then we’ll read the existing age and ask the user to confirm that they want to update from that age to the new one

UPDATING RANDOM ACCESS FILE

- We need to make sure that we don't advance past the end of the file…
- We can accomplish this with the following code:

```
cout << "What record number should we change? "
int recordNum;
cin >> recordNum;

// Calculate offset in file.  Each record is 36 bytes, plus the age
// field is 21 bytes from the start of the record
int offset = (36 * recordNum) + 21;
file.seekp(offset);

// Check what the next character is in the buffer.  If it is -1
// we've gone too far!
if (file.peek() == -1) {
    cout << "invalid index" << endl;
    return -1;
}
```

DEMONSTRATION #6

Updating a random access file

FINAL THOUGHTS

- Assignment #8 due tomorrow at 11:59PM
- I will provide final project details soon!