18.7 — Random file I/O

BY ALEX ON APRIL 4TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 4TH, 2018

The file pointer

Each file stream class contains a file pointer that is used to keep track of the current read/write position within the file. When something is read from or written to a file, the reading/writing happens at the file pointer's current location. By default, when opening a file for reading or writing, the file pointer is set to the beginning of the file. However, if a file is opened in append mode, the file pointer is moved to the end of the file, so that writing does not overwrite any of the current contents of the file.

Random file access with seekg() and seekp()

So far, all of the file access we've done has been sequential -- that is, we've read or written the file contents in order. However, it is also possible to do random file access -- that is, skip around to various points in the file to read its contents. This can be useful when your file is full of records, and you wish to retrieve a specific record. Rather than reading all of the records until you get to the one you want, you can skip directly to the record you wish to retrieve.

Random file access is done by manipulating the file pointer using either seekg() function (for input) and seekp() function (for output). In case you are wondering, the g stands for "get" and the p for "put". For some types of streams, seekg() (changing the read position) and seekp() (changing the write position) operate independently -- however, with file streams, the read and write position are always identical, so seekg and seekp can be used interchangeably.

The seekg() and seekp() functions take two parameters. The first parameter is an offset that determines how many bytes to move the file pointer. The second parameter is an Ios flag that specifies what the offset parameter should be offset from.

Ios seek flag	Meaning
beg	The offset is relative to the beginning of the file (default)
cur	The offset is relative to the current location of the file pointer
end	The offset is relative to the end of the file

A positive offset means move the file pointer towards the end of the file, whereas a negative offset means move the file pointer towards the beginning of the file.

Here are some examples:

```
inf.seekg(14, ios::cur); // move forward 14 bytes
inf.seekg(-18, ios::cur); // move backwards 18 bytes
inf.seekg(22, ios::beg); // move to 22nd byte in file
inf.seekg(24); // move to 24th byte in file
inf.seekg(-28, ios::end); // move to the 28th byte before end of the file
```

Moving to the beginning or end of the file is easy:

```
inf.seekg(0, ios::beg); // move to beginning of file
inf.seekg(0, ios::end); // move to end of file
```

Let's do an example using seekg() and the input file we created in the last lesson. That input file looks like this:

```
This is line 1
This is line 2
This is line 3
This is line 4
```

Here is the example:

```
int main()
{
    using namespace std;
```

```cpp

    ifstream inf("Sample.dat");

    // If we couldn't open the input file stream for reading
    if (!inf)
    {
        // Print an error and exit
        cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;
        exit(1);
    }

    string strData;

    inf.seekg(5); // move to 5th character
    // Get the rest of the line and print it
    getline(inf, strData);
    cout << strData << endl;

    inf.seekg(8, ios::cur); // move 8 more bytes into file
    // Get rest of the line and print it
    getline(inf, strData);
    cout << strData << endl;

    inf.seekg(-15, ios::end); // move 15 bytes before end of file
    // Get rest of the line and print it
    getline(inf, strData);
    cout << strData << endl;

    return 0;
}
```

This produces the result:

```
is line 1
line 2
his is line 4
```

Note: Some compilers have buggy implementations of seekg() and tellg() when used in conjunction with text files (due to buffering). If your compiler is one of them (and you'll know because your output will differ from the above), you can try opening the file in binary mode instead:

```cpp
    ifstream inf("Sample.dat", ifstream::binary);
```

Two other useful functions are tellg() and tellp(), which return the absolute position of the file pointer. This can be used to determine the size of a file:

```cpp
ifstream inf("Sample.dat");
inf.seekg(0, ios::end); // move to end of file
cout << inf.tellg();
```

This prints:

```
64
```

which is how long sample.dat is in bytes (assuming a carriage return after the last line).

Reading and writing a file at the same time using fstream

The fstream class is capable of both reading and writing a file at the same time -- almost! The big caveat here is that it is not possible to switch between reading and writing arbitrarily. Once a read or write has taken place, the only way to switch between the two is to perform an operation that modifies the file position (e.g. a seek). If you don't actually want to move the file pointer, you can always seek to the current position:

```cpp
// assume iofile is an object of type fstream
```

```
iofile.seekg(iofile.tellg(), ios::beg); // seek to current file position
```

If you do not do this, any number of strange and bizarre things may occur.

(Note: Although it may seem that `iofile.seekg(0, ios::cur)` would also work, it appears some compilers may optimize this away.)

One other bit of trickiness: Unlike ifstream, where we could say `while (inf)` to determine if there was more to read, this will not work with fstream.

Let's do a file I/O example using fstream. We're going to write a program that opens a file, reads its contents, and changes any vowels it finds to a '#' symbol.

```
int main()
{
    using namespace std;

    // Note we have to specify both in and out because we're using fstream
    fstream iofile("Sample.dat", ios::in | ios::out);

    // If we couldn't open iofile, print an error
    if (!iofile)
    {
        // Print an error and exit
        cerr << "Uh oh, Sample.dat could not be opened!" << endl;
        exit(1);
    }

    char chChar; // we're going to do this character by character

    // While there's still data to process
    while (iofile.get(chChar))
    {
        switch (chChar)
        {
            // If we find a vowel
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U':

                // Back up one character
                iofile.seekg(-1, ios::cur);

                // Because we did a seek, we can now safely do a write, so
                // let's write a # over the vowel
                iofile << '#';

                // Now we want to go back to read mode so the next call
                // to get() will perform correctly.  We'll seekg() to the current
                // location because we don't want to move the file pointer.
                iofile.seekg(iofile.tellg(), ios::beg);

                break;
        }
    }

    return 0;
}
```

Other useful file functions

To delete a file, simply use the remove() function.

Also, the is_open() function will return true if the stream is currently open, and false otherwise.

A warning about writing pointers to disk

While streaming variables to a file is quite easy, things become more complicated when you’re dealing with pointers. Remember that a pointer simply holds the address of the variable it is pointing to. Although it is possible to read and write addresses to disk, it is extremely dangerous to do so. This is because a variable’s address may differ from execution to execution. Consequently, although a variable may have lived at address 0x0012FF7C when you wrote that address to disk, it may not live there any more when you read that address back in!

For example, let’s say you had an integer named nValue that lived at address 0x0012FF7C. You assigned nValue the value 5. You also declared a pointer named *pnValue that points to nValue. pnValue holds nValue’s address of 0x0012FF7C. You want to save these for later, so you write the value 5 and the address 0x0012FF7C to disk.

A few weeks later, you run the program again and read these values back from disk. You read the value 5 into another variable named nValue, which lives at 0x0012FF78. You read the address 0x0012FF7C into a new pointer named *pnValue. Because pnValue now points to 0x0012FF7C when the nValue lives at 0x0012FF78, pnValue is no longer pointing to nValue, and trying to access pnValue will lead you into trouble.

Rule: Do not write addresses to files. The variables that were originally at those addresses may be at different addresses when you read their values back in from disk, and the addresses will be invalid.

A.1 -- Static and dynamic libraries

Index

18.6 -- Basic file I/O

66 comments to 18.7 — Random file I/O

« Older Comments 1 2

Michael
January 2, 2018 at 8:06 pm · Reply

Hi Alex,
why did you do a seekg() when you want to write something in? Should we use seekp() then?

Alex
January 4, 2018 at 3:58 pm · Reply

For file streams, there's only one file pointer, so seekg() and seekp() operate identically. It's probably slightly more correct to use seekp() when intending to write, but functionally it doesn't make a difference in this case.