

18.7 — Random file I/O

BY ALEX ON APRIL 4TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```
1 inf.seekg(14, ios::cur); // move forward 14 bytes
2 inf.seekg(-18, ios::cur); // move backwards 18 bytes
3 inf.seekg(22, ios::beg); // move to 22nd byte in file
4 inf.seekg(24); // move to 24th byte in file
5 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:

```
1 inf.seekg(0, ios::beg); // move to beginning of file
2 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:

```
1  int main()
2  {
3      using namespace std;
4
5      ifstream inf("Sample.dat");
6
7      // If we couldn't open the input file stream for reading
8      if (!inf)
9      {
10         // Print an error and exit
11         cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;
12         exit(1);
13     }
14
15     string strData;
16
17     inf.seekg(5); // move to 5th character
18     // Get the rest of the line and print it
19     getline(inf, strData);
20     cout << strData << endl;
21
22     inf.seekg(8, ios::cur); // move 8 more bytes into file
23     // Get rest of the line and print it
24     getline(inf, strData);
25     cout << strData << endl;
26
27     inf.seekg(-15, ios::end); // move 15 bytes before end of file
28     // Get rest of the line and print it
29     getline(inf, strData);
30     cout << strData << endl;
31
32     return 0;
33 }
```

This produces the result:

is line 1
line 2
his is line 4

Note: Some compilers have buggy implementations of `seekg()` and `seekp()` 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:

```
1  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:

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

This prints:

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 (because it's already in the spot you want), you can always seek to the current position:

```
1 // assume iofile is an object of type fstream
2 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.

```
1 int main()
2 {
3     using namespace std;
4
5     // Note we have to specify both in and out because we're using fstream
6     fstream iofile("Sample.dat", ios::in | ios::out);
7
8     // If we couldn't open iofile, print an error
9     if (!iofile)
10    {
11        // Print an error and exit
12        cerr << "Uh oh, Sample.dat could not be opened!" << endl;
13        exit(1);
14    }
15
16    char chChar; // we're going to do this character by character
17
18    // While there's still data to process
19    while (iofile.get(chChar))
20    {
21        switch (chChar)
22        {
23            // If we find a vowel
24            case 'a':
25            case 'e':
26            case 'i':
27            case 'o':
28            case 'u':
29            case 'A':
30            case 'E':
31            case 'I':
32            case 'O':
33            case 'U':
34
35                // Back up one character
```

```

36         iosfile.seekg(-1, ios::cur);
37
38         // Because we did a seek, we can now safely do a write, so
39         // let's write a # over the vowel
40         iosfile << '#';
41
42         // Now we want to go back to read mode so the next call
43         // to get() will perform correctly. We'll seekg() to the current
44         // location because we don't want to move the file pointer.
45         iosfile.seekg(iosfile.tellg(), ios::beg);
46
47         break;
48     }
49 }
50
51 return 0;
52 }
```

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](#)