

## 28.7 — Random file I/O

👤 [ALEX](#)<sup>1</sup> ⌚ **APRIL 16, 2024**

### 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
<code>beg</code>	The offset is relative to the beginning of the file (default)
<code>cur</code>	The offset is relative to the current location of the file pointer
<code>end</code>	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, std::ios::cur); // move forward 14 bytes
2 | inf.seekg(-18, std::ios::cur); // move backwards 18 bytes
3 | inf.seekg(22, std::ios::beg); // move to 22nd byte in file
4 | inf.seekg(24); // move to 24th byte in file
5 | inf.seekg(-28, std::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, std::ios::beg); // move to beginning of file
2 | inf.seekg(0, std::ios::end); // move to end of file
```

## Warning

In a text file, seeking to a position other than the beginning of the file may result in unexpected behavior.

In programming, a newline ('\n') is actually an abstraction.

- On Windows, a newline is represented as sequential CR (carriage return) and LF (line feed) characters (thus taking 2 bytes of storage).
- On Unix, a newline is represented as a LF (line feed) character (thus taking 1 byte of storage).

Seeking past a newline in either direction takes a variable number of bytes depending on how the file was encoded, which means results will vary depending on which encoding is used.

Also on some operating systems, files may be padded with trailing zero bytes (bytes that have value 0). Seeking to the end of the file (or an offset from the end of the file) will produce different results on such files.

Just to give you an idea of how they work, 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  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      std::ifstream inf{ "Sample.txt" };
8
9      // If we couldn't open the input file stream for reading
10     if (!inf)
11     {
12         // Print an error and exit
13         std::cerr << "Uh oh, Sample.txt could not be opened for reading!\n";
14         return 1;
15     }
16
17     std::string strData;
18
19     inf.seekg(5); // move to 5th character
20     // Get the rest of the line and print it, moving to line 2
21     std::getline(inf, strData);
22     std::cout << strData << '\n';
23
24     inf.seekg(8, std::ios::cur); // move 8 more bytes into file
25     // Get rest of the line and print it
26     std::getline(inf, strData);
27     std::cout << strData << '\n';
28
29     inf.seekg(-14, std::ios::end); // move 14 bytes before end of file
30     // Get rest of the line and print it
31     std::getline(inf, strData); // undefined behavior
32     std::cout << strData << '\n';
33
34     return 0;
35 }

```

This produces the result:

```

is line 1
line 2
This is line 4

```

You may get a different result for the third line depending on how your file is encoded.

`seekg()` and `seekp()` are better used on binary files. You can open the above file in binary mode via:

```

1 | std::ifstream inf {"Sample.txt", std::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 | std::ifstream inf {"Sample.txt"};
2 | inf.seekg(0, std::ios::end); // move to end of file
3 | std::cout << inf.tellg();

```

On the author's machine, this prints:

which is how long sample.txt is in bytes (assuming a newline after the last line).

### Author's note

The result of `64` in the prior example occurred on Windows. If you run the example on Unix, you'll get `60` instead, due to the smaller newline representation. You may get something else if your file is padded with trailing zero bytes.

### 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(), std::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, std::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  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main()
6  {
7      // Note we have to specify both in and out because we're using fstream
8      std::fstream iofile{ "Sample.txt", std::ios::in | std::ios::out };
9
10     // If we couldn't open iofile, print an error
11     if (!iofile)
12     {
13         // Print an error and exit
14         std::cerr << "Uh oh, Sample.txt could not be opened!\n";
15         return 1;
16     }
17
18     char chChar{}; // we're going to do this character by character
19
20     // While there's still data to process
21     while (iofile.get(chChar))
22     {
23         switch (chChar)
24         {
25             // If we find a vowel
26             case 'a':
27             case 'e':
28             case 'i':
29             case 'o':
30             case 'u':
31             case 'A':
32             case 'E':
33             case 'I':
34             case 'O':
35             case 'U':
36
37                 // Back up one character
38                 iofile.seekg(-1, std::ios::cur);
39
40                 // Because we did a seek, we can now safely do a write, so
41                 // let's write a # over the vowel
42                 iofile << '#';
43
44                 // Now we want to go back to read mode so the next call
45                 // to get() will perform correctly. We'll seekg() to the current
46                 // location because we don't want to move the file pointer.
47                 iofile.seekg(iofile.tellg(), std::ios::beg);
48
49                 break;
50         }
51     }
52
53     return 0;
54 }

```

After running the above program, our Sample.txt file will look like this:

```

Th#s #s l#n# 1
Th#s #s l#n# 2
Th#s #s l#n# 3
Th#s #s l#n# 4

```

## 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.

#### Warning

Do not write memory 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.



#### [Next lesson](#)

A.1 [Static and dynamic libraries](#)

2



#### [Back to table of contents](#)

3



#### [Previous lesson](#)

28.6 [Basic file I/O](#)

4

5



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...

 Name\*


 Email\* 

Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!?

 Avatars from <https://gravatar.com/><sup>7</sup> are connected to your provided email address.

182 COMMENTS

Newest ▼



**whoever**

 June 26, 2025 6:15 am PDT

no quiz? :(



2

 Reply



**Leni**

 February 23, 2025 6:16 pm PST

The explanation of `seekg()` and `seekp()` is thorough, highlighting their role in random file access. However, it should emphasize when to use them with text versus binary files to avoid issues like newline encoding differences.



1

 Reply



**Tobito**

 February 8, 2025 9:10 am PST

I have problem on displaying the data from `fstream` after i have writing the data to the binary file, the file have data but it can not displaying anything. Any idea what's the cause for it



0

 Reply



**ssny**

 January 7, 2025 12:09 pm PST

On the last example, the results I get on `sample.txt` are

1	This #s a l#ne 1#
2	This#is#a #in# 2
3	Thi# i# a#li#e#3
4	Th#s #s # l#n# 4

It works fine when the `std::ios::binary` flag is set. I assume it's related to newlines. Running it on Windows and using g++

 Last edited 6 months ago by ssny

 1  Reply



sandersan

 Reply to [ssny](#)<sup>8</sup>  May 17, 2025 8:53 am PDT

+1. I encountered the same issue.

 1  Reply

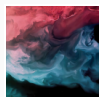


EmtyC



 January 4, 2025 3:44 pm PST

Oh...no, it's over... :D nice journey it was :-> (meeting you in the end? lesson tomorrow, it's late and I still haven't covered the C++ standards introductions)

 7  Reply



EmtyC

 Reply to [EmtyC](#)<sup>9</sup>  January 4, 2025 3:47 pm PST

Nah, I am still having some plans about some `contributions` to be made so, not yet over from this site, yipeeee :->

 1  Reply

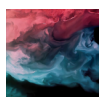


EmtyC

 January 4, 2025 3:42 pm PST

Other note, this chapter seems missing some things considering the importance of I/O in the STL, but like you said, this is a C++ tutorial not a STL tutorial :->

 2  Reply



EmtyC

 January 4, 2025 3:39 pm PST

Also, you may note that since `seeks` and `tells` are defined in `std::ostream` and `std::istream` (I know it's actually in the template `std::basic_i/ostream<param1, param2>` but for simplicity you know :->) so you can use them in the `sstringstream` classes (`i/ostringstream`, and the `stringstream`)



One big note is that they have no, or undefined(not sure about this thought), behavior if used with std::cin and std::cout (or really any std::basic\_ios linked to a non-random access capable stream)

 0     Reply




**khayfawahbe**  
🕒 December 10, 2024 2:17 pm PST

Ana etfasha5t, still proud though

 0     Reply



**Abdelrahman**  
 Reply to [khayfawahbe](#)<sup>10</sup>    🕒 December 22, 2024 3:58 pm PST

Haha, I'm yet to reach this place (lisa btfashi5)  
proud of you too ^\_^

 0     Reply



**Asicx**  
🕒 November 12, 2024 1:50 pm PST

This chapter really deserves a quiz section.

 2     Reply



**emme**  
🕒 September 13, 2024 5:47 am PDT

Oh man, it feels like it ended on the most interesting part.

 3     Reply



**mzulali0**  
 Reply to [emme](#)<sup>11</sup>    🕒 October 1, 2024 8:49 pm PDT

Had me gripping my seat.

 1     Reply



**emme**  
 Reply to [mzulali0](#)<sup>12</sup>    🕒 October 2, 2024 1:26 am PDT

Top 10 anime cliffhangers

 1     Reply



Shubham

Reply to [emme](#)<sup>13</sup> ⌚ November 5, 2024 5:58 am PST

Is it worth learning cpp from this website. I want to be a game develeoper and after i did some reasearvh cpp is most widely used should i learn from here or watch yt vid tutorials

👍 0

➡ Reply



EmtyC

Reply to [Shubham](#)<sup>14</sup> ⌚ January 4, 2025 3:33 pm PST

Ya, undeniably yes. Unless you have an issue with text based learning(as Alex, the author, said) it's an absolute gem in terms of ease of understanding. Also, you get the privilege that Sir Alex may respond to you (although might depend on your question, and if someone responded to you already, but he might miss your comment though, although rare), which you might not get in every place or tutorial.

👍 1

➡ Reply



Alex Author

Reply to [Shubham](#)<sup>14</sup> ⌚ November 7, 2024 6:45 pm PST

Try both and do whatever you find more natural. Some people like learning from text, others from videos.

👍 1

➡ Reply

## Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/a1-static-and-dynamic-libraries/>
3. <https://www.learncpp.com/>
4. <https://www.learncpp.com/cpp-tutorial/basic-file-io/>
5. <https://www.learncpp.com/random-file-io/>
6. <https://www.learncpp.com/cpp-tutorial/function-templates/>
7. <https://gravatar.com/>
8. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-606362>
9. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-606240>
10. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-605096>
11. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-601933>
12. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-602613>
13. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-602623>
14. <https://www.learncpp.com/cpp-tutorial/random-file-io/#comment-603860>
15. <https://g.ezoic.net/privacy/learncpp.com>

