23.6 — Basic file I/O

ALEX SEPTEMBER 5, 2021

File I/O in C++ works very similarly to normal I/O (with a few minor added complexities). There are 3 basic file I/O classes in C++: ifstream (derived from istream), ofstream (derived from ostream), and fstream (derived from iostream). These classes do file input, output, and input/output respectively. To use the file I/O classes, you will need to include the fstream header.

Unlike the cout, cin, cerr, and clog streams, which are already ready for use, file streams have to be explicitly set up by the programmer. However, this is extremely simple: to open a file for reading and/or writing, simply instantiate an object of the appropriate file I/O class, with the name of the file as a parameter. Then use the insertion (>) operator to write to or read data from the file. Once you are done, there are several ways to close a file: explicitly call the close() function, or just let the file I/O variable go out of scope (the file I/O class destructor will close the file for you).

File output

To do file output in the following example, we're going to use the ofstream class. This is extremely straighforward:

```cpp
#include <fstream>
#include <iostream>

int main()
{
    // ofstream is used for writing files
    // We'll make a file called Sample.dat
    std::ofstream outf{ "Sample.dat" };

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

    // We'll write two lines into this file
    outf << "This is line 1" << '\n';
    outf << "This is line 2" << '\n';

    return 0;

    // When outf goes out of scope, the ofstream
    // destructor will close the file
}
```

If you look in your project directory, you should see a file called Sample.dat. If you open it with a text editor, you will see that it indeed contains two lines we wrote to the file.

Note that it is also possible to use the put() function to write a single character to the file.

File input

Now, we’ll take the file we wrote in the last example and read it back in from disk. Note that ifstream returns a 0 if we’ve reached the end of the file (EOF). We’ll use this fact to determine how much to read.

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    // ifstream is used for reading files
    // We'll read from a file called Sample.dat
    std::ifstream inf{ "Sample.dat" };

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

    // While there's still stuff left to read
    while (inf)
    {
        // read stuff from the file into a string and print it
        std::string strInput;
        inf >> strInput;
        std::cout << strInput << '\n';
    }

    return 0;

    // When inf goes out of scope, the ifstream
    // destructor will close the file
}
```

This produces the result:

```
This
is
line
1
This
is
line
2
```

Hmmm, that wasn’t quite what we wanted. Remember that the extraction operator breaks on whitespace. In order to read in entire lines, we’ll have to use the getline() function.

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    // ifstream is used for reading files
    // We'll read from a file called Sample.dat
    std::ifstream inf{ "Sample.dat" };

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

    // While there's still stuff left to read
    while (inf)
    {
        // read stuff from the file into a string and print it
        std::string strInput;
        std::getline(inf, strInput);
        std::cout << strInput << '\n';
    }

    return 0;

    // When inf goes out of scope, the ifstream
    // destructor will close the file
}
```

This produces the result:

```
This is line 1
This is line 2
```

Buffered output

Output in C++ may be buffered. This means that anything that is output to a file stream may not be written to disk immediately. Instead, several output operations may be batched and handled together. This is done primarily for performance reasons. When a buffer is written to disk, this is called flushing the buffer. One way to cause the buffer to be flushed is to close the file -- the contents of the buffer will be flushed to disk, and then the file will be closed.

Buffering is usually not a problem, but in certain circumstance it can cause complications for the unwary. The main culprit in this case is when there is data in the buffer, and then program terminates immediately (either by crashing, or by calling exit()). In these cases, the destructors for the file stream classes are not executed, which means the files are never closed, which means the buffers are never flushed. In this case, the data in the buffer is not written to disk, and is lost forever. This is why it is always a good idea to explicitly close any open files before calling exit().

It is possible to flush the buffer manually using the ostream::flush() function or sending std::flush to the output stream. Either of these methods can be useful to ensure the contents of the buffer are written to disk immediately, just in case the program crashes.

One interesting note is that std::endl; also flushes the output stream. Consequently, overuse of std::endl (causing unnecessary buffer flushes) can have performance impacts when doing buffered I/O where flushes are expensive (such as writing to a file). For this reason, performance conscious programmers will often use '\n' instead of std::endl to insert a newline into the output stream, to avoid unnecessary flushing of the buffer.

File modes

What happens if we try to write to a file that already exists? Running the output example again shows that the original file is completely overwritten each time the program is run. What if, instead, we wanted to append some more data to the end of the file? It turns out that the file stream constructors take an optional second parameter that allows you to specify information about how the file should be opened. This parameter is called mode, and the valid flags that it accepts live in the Ios class.

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks to the end of the file before reading/writing
binary	Opens the file in binary mode (instead of text mode)
in	Opens the file in read mode (default for ifstream)
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it already exists

It is possible to specify multiple flags by bitwise ORing them together (using the | operator). Ifstream defaults to std::ios::in file mode. Ofstream defaults to std::ios::out file mode. And fstream defaults to std::ios::in | std::ios::out file mode, meaning you can both read and write by default.

Tip

Due to the way fstream was designed, it may fail if std::ios::in is used and the file being opened does not exist. If you need to create a new file using fstream, use std::ios::out mode only.

Let's write a program that appends two more lines to the Sample.dat file we previously created:

```cpp
#include <iostream>
#include <fstream>

int main()
{
    // We'll pass the ios:app flag to tell the ofstream to append
    // rather than rewrite the file.  We do not need to pass in
std::ios::out
    // because ofstream defaults to std::ios::out
    std::ofstream outf{ "Sample.dat", std::ios::app };

    // If we couldn't open the output file stream for writing
    if (!outf)
    {
        // Print an error and exit
        std::cerr << "Uh oh, Sample.dat could not be opened for
writing!\n";
        return 1;
    }

    outf << "This is line 3" << '\n';
    outf << "This is line 4" << '\n';

    return 0;

    // When outf goes out of scope, the ofstream
    // destructor will close the file
}
```

Now if we take a look at Sample.dat (using one of the above sample programs that prints its contents, or loading it in a text editor), we will see the following:

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

Explicitly opening files using open()

Just like it is possible to explicitly close a file stream using close(), it’s also possible to explicitly open a file stream using open(). open() works just like the file stream constructors -- it takes a file name and an optional file mode.

For example:

```cpp
std::ofstream outf{ "Sample.dat" };
outf << "This is line 1" << '\n';
outf << "This is line 2" << '\n';
outf.close(); // explicitly close the file

// Oops, we forgot something
outf.open("Sample.dat", std::ios::app);
outf << "This is line 3\n";
outf.close();
```

Next lesson
23.7 Random file I/O

Back to table of contents

Previous lesson
23.5 Stream states and input validation

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

166 COMMENTS

Newest

X

X