Late Developer
Random thoughts of an old C++ guy

All About EOF

December 4, 2012

Introduction

Of all of the problems posted by beginner C++ and C programmers on sites like Reddit and Stack Overflow, surely the commonest is the confusion over the treatment of the end-of-file condition when writing programs that either interact with the user or read input from files. My own estimate is that over 95% of questions exhibit a complete misunderstanding of the end-of-file concept. This article is an attempt to explain all of the issues regarding this confused and confusing subject, with particular reference to C++ and C programmers using the Windows and Unix-like (such as Linux, which I'll use from now on as the exemplar for this OS type) operating systems.

The myth of the EOF character

The first problem that many beginners face when confronted with the end-of-file issue is that of the EOF character – basically, there isn't one, but people think there is. Neither the Windows nor the Linux operating systems have any concept of a marker character for indicating the end-of-file. If you create a text file using Notepad on Windows, or Vim on Linux, or any other editor/OS combination you fancy, that file will nowhere in it contain a special character marking the end-of-file. Both Windows and Linux have file systems that always know the exact length in bytes of the contents of a file, and have absolutely no need of any special character marking the file's end.

So if neither Windows nor Linux use an EOF character, where does the idea come from? Well, long ago and far away, there was an operating system (to use the term loosely) called CP/M, which ran on 8-bit processors like the Zilog Z80 and Intel 8080. The CP/M file system did *not* know exactly how long a file was in bytes – it only knew how many disk blocks it took up. This meant that if you edited a small file containing the text **hello world**, CP/M would not know that the file was 11 bytes long – it would only know it took up a single disk block, which would be a minimum of 128 bytes. As people generally like to know how big their files *appear* to be, rather than the number of blocks they take up, an end-of-file character was needed. CP/M re-used the Control-Z character (decimal code 26, hex 1A, original intended use lost in the mists of time) from the ASCII character set for this purpose – when a CP/M application read a Control-Z character it would typically treat that read as though an end-of-file had occurred.. There was nothing forcing applications to do this; apps that processed binary data would need some other means of knowing if they were at the end-of-file, and the OS itself did not treat Control-Z specially.

When MS-DOS came along, compatibility with with CP/M was very important, as a lot of the first MS-DOS applications were simply CP/M apps that had been fed through mechanical translators that turned Z80/8080 machine code into 8086 machine code. As the applications were not re-written, they still treated Control-Z as the end-of-file marker, and some do to this very day. In fact, this treatment of Control-Z is built in to the Microsoft C Runtime Library, *if* a file is opened in text mode. It's important to restate that the Widows OS itself knows and cares nothing about Control-Z – this behaviour is purely down to the MS library, which unfortunately just about every Windows program uses. It's also important to realise that this is purely a Windows issue – Linux (and the other Unixes) have never used Control-Z (or anything else) as an end-of-file marker in any shape or form.

Some demo code

You can demonstrate this unfortunate feature of the MS libraries with this code. First, write a program that puts a Control-Z into a text file

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

int main() {
    ofstream ofs( "myfile.txt" );
    ofs << "line 1\n";
    ofs << char(26);
    ofs << "line 2\n";
}
```

If you compile and run this on either Windows or Linux, it will create a text file with an embedded Control-Z (ASCII code 26) between the two lines of text. On neither platform does Control-Z have any special meaning on output. You can now try and read the file using command line facilities. On Windows:

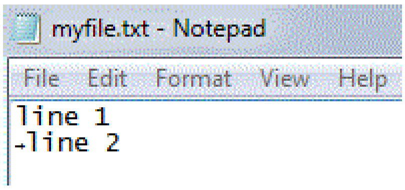
```
c:\users\neilb\home\temp>type myfile.txt
line 1
```

Notice only the first line is displayed. On Linux:

```
[neilb@ophelia temp]$ cat myfile.txt
line 1
?line 2
```

Both lines are displayed, and a strange character (represented here by the question mark) is also displayed between them, as the cat command has just read Control-Z like any other character and printed it out – exactly what gets displayed depends on your terminal software.

This might seem to indicate that the Windows OS *does* know about the Control-Z character, but that’s not the case – only certain application code knows about it. If you open the file using the Windows Notepad utility, you will see this:



```
line 1
→line 2
```

(https://latedev.files.wordpress.com/2012/12/screenshot_myfile-txt-notepad_2012-12-04_12-43-56.gif)

Both lines of text are displayed, with the Control-Z character between them – Notepad does not know about Control-Z as an end-of-file marker at all.

Text versus binary mode

So what is the difference between the type command used above and the Notepad application? It’s actually hard to say. Possibly the type command has some special code that checks for the Control-Z character in its input. However, Windows programmers using the C++ iostream library and the C stream library have the option of opening a file in either text mode or binary mode, and this will make a difference to what gets read.

Here’s the canonical way to read a text file in C++:

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

int main() {
    ifstream ifs( "myfile.txt" );
    string line;
    while( getline( ifs, line ) ) {
        cout << line << '\n';
    }
}
```

If you compile and run this file on Windows, you will see that the Control-Z is treated as an end-of-file marker; the output is:

```
line 1
```

However, if you open the file in binary mode by making this change:

```
ifstream ifs( "myfile.txt", ios::binary );
```

the output is:

```
line 1
?line 2
```

The Control-Z character is only treated as being special in text mode (the default) – in binary mode, it is just treated as any other character. Note that this is only true for Windows; on Linux both programs behave in exactly the same manner.

So what to do? There are two things to remember:

- If you want your files to open portably in text mode, don’t embed Control-Z characters in them.
- If you must embed Control-Z characters, and you want the files to be read portably, open the files in binary mode.

Note that here "portably" means "using any application, even if you only program on Windows!"

But what about Control-D?

Some Linux users may at this point be thinking, "But what about the Control-D character I use to end shell input? Isn't that an end-of-file character?" Well, no, it isn't. If you embed a Control-D character in a text file, Linux will pay absolutely no attention to it. In fact, the Control-D you type in at the shell to end input is simply a signal to the shell to close the standard input stream. No character is inserted into the stream at all. In fact, using the **stty** utility, you can change what character causes standard input to be closed from Control-D to whatever you like, but in no case will a special character be inserted in the input stream, and even if it were, Linux would not treat it as an end-of-file marker.

The EOF value in C++ and C

Just to confuse things even more, both C++ and C define a special value with the name EOF. In C, it is defined in **<stdio.h>** as:

```
#define EOF     (-1)
```

and similarly in **<cstdio>** for C++.

Notice that EOF in this context has nothing to do with Control-Z. It doesn't have the value 26 and in fact in use it is not a character at all but an integer. It is used as the return value of functions like this:

```
int getchar(void);
```

The getchar() function is used to read individual characters from standard input and returns the special value EOF when the end-of-file is reached. The end of file may or may be indicated by the Control-Z character (see discussion above), but in no case will the EOF value be the same as the ASCII code for Control-Z. In fact, getchar() returns an int, not a char, and it's important that its return value is stored in an int, as a comparison between a char and a signed integer is not guaranteed to work correctly. The canonical way to use this function to read standard input is:

```
#include <stdio.h>

int main() {
    int c;
    while( (c = getchar()) != EOF ) {
        putchar( c );
    }
}
```

The eof() and feof() functions

Another layer of confusion is added by both C++ and C providing functions to check the state of an input stream. Almost all beginner programmers get confused by these functions, so it may be a good idea to state up-front what they do and how they should *not* be used:

*Both eof() and feof() check the state of an input stream to see if an end-of-file condition has occurred. Such a condition can only occur following an attempted read operation. If you call either function without previously performing a read, **your code is wrong**! Never loop on an eof function.*

To illustrate this, let's write a program that reads a file, and adds line numbers to the file contents on output. To simplify things, we'll use a fixed file name and skip any error checking. Most beginners will write something like this:

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

int main() {
    ifstream ifs( "afile.txt" );
    int n = 0;
    while( ! ifs.eof() ) {
        string line;
        getline( ifs, line );
        cout << ++n << " " << line << '\n';
    }
}
```

This seems sensible enough, but remember the advice – *"If you call either function without previously performing a read, **your code is wrong**!"* and in this case we are indeed calling eof() before a read operation. To see why this is wrong, consider what happens if **afile.txt** is an empty file. The first time through the loop the check for eof() will fail, as no read operation has occurred. We then read something, which will set the end-of-file condition, but too late. And we then output a line, with line number 1, that does not exist in the input file. By similar logic, the program always outputs one spurious extra line.

To write the program properly, you need to call the eof() function after the read operation, or not at all. If you are not expecting to encounter problems other than end-of-file, you would write the code like this:

```
int main() {
    ifstream ifs( "afile.txt" );
    int n = 0;
    string line;
    while( getline( ifs, line ) ) {
        cout << ++n << " " << line << '\n';
    }
}
```

This uses a conversion operator which turns the return value of getline() , which is the stream passed as the first parameter, into something that can be tested in a while-loop – the loop continues as long as the stream is not in an end-of-file (or other error) condition.

Similarly in C. You should not write code like this:

```
#include <stdio.h>

int main() {
    FILE * f = fopen( "afile.txt", "r" );
    char line[100];
    int n = 0;
    while( ! feof( f ) ) {
        fgets( line, 100, f );
        printf( "%d %s", ++n, line );
    }
    fclose( f );
}
```

which will almost certainly print garbage if handed an empty file (and exhibit undefined behaviour too). You want:

```
#include <stdio.h>

int main() {
    FILE * f = fopen( "afile.txt", "r" );
    char line[100];
    int n = 0;
    while( fgets( line, 100, f ) != NULL ) {
        printf( "%d %s", ++n, line );
    }
    fclose( f );
}
```

So if eof() and feof() are so apparently useless, why do the C++ and C Standard Libraries supply them? Well, they are useful in the case where a read error could be caused by something other than end-of-file, and you want to distinguish if that’s the case:

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

int main() {
    ifstream ifs( "afile.txt" );
    int n = 0;
    string line;
    while( getline( ifs, line ) ) {
        cout << ++n << " " << line << '\n';
    }
    if ( ifs.eof() ) {
        // OK - EOF is an expected condition
    }
    else {
        // ERROR - we hit something other than EOF
    }
}
```

Summary

All the above may make it seem that the EOF issue is extremely complicated, but it really only comes down to three basic rules:

There is no EOF character, unless you open files in text mode on Windows, or implement one yourself.
The EOF symbol in C++ and C is not an end-of-file character, it is special return value of certain library functions.
Don’t loop on the eof() or feof() functions.

If you keep these rules in mind, you should avoid being bitten by most of the bugs associated with misunderstanding the nature of the end-of-file condition in C++ and C.

From → c++, linux, windows

Leave a Comment

Blog at WordPress.com.

L