Why is “while (!feof (file))” always wrong?

Asked 9 years, 2 months ago Active yesterday Viewed 222k times

573

282

I've seen people trying to read files like this in a lot of posts lately:

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
    char *path = "stdin";
    FILE *fp = argc > 1 ? fopen(path=argv[1], "r") : stdin;

    if( fp == NULL ) {
        perror(path);
        return EXIT_FAILURE;
    }

    while( !feof(fp) ) {  /* THIS IS WRONG */
        /* Read and process data from file… */
    }
    if( fclose(fp) != 0 ) {
        perror(path);
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

What is wrong with this loop?

c file while-loop eof feof

share improve this question follow

edited Mar 17 at 10:26 RobertS supports Monica Cellio 8,666 ●2 ●7 ●34

asked Mar 25 '11 at 11:42 William Pursell 157k ●43 ●228 ●260

19 Why it's bad to use `feof()` to control a loop – Grijesh Chauhan Oct 17 '13 at 18:25

1 Why is iostream::eof inside a loop condition considered wrong? – Jonathan Wakely Dec 4 '14 at 9:30

add a comment

5 Answers

Active Oldest Votes

453

I'd like to provide an abstract, high-level perspective.

Concurrency and simultaneity

I/O operations interact with the environment. The environment is not part of your program, and not under your control. The environment truly exists "concurrently" with your program. As with all things concurrent, questions about the "current state" don't make sense: There is no concept of "simultaneity" across concurrent events. Many properties of state simply don't *exist* concurrently.

Let me make this more precise: Suppose you want to ask, "do you have more data". You could ask this of a concurrent container, or of your I/O system. But the answer is generally unactionable, and thus meaningless. So what if the container says "yes" – by the time you try reading, it may no longer have data. Similarly, if the answer is "no", by the time you try reading, data may have arrived. The conclusion is that there simply *is* no property like "I have data", since you cannot act meaningfully in response to any possible answer. (The situation is slightly better with buffered input, where you might conceivably get a "yes, I have data" that constitutes some kind of guarantee, but you would still have to be able to deal with the opposite case. And with output the situation is certainly just as bad as I described: you never know if that disk or that network buffer is full.)

So we conclude that it is impossible, and in fact un*reasonable*, to ask an I/O system whether it *will be* able to perform an I/O operation. The only possible way we can interact with it (just as with a concurrent container) is to *attempt* the operation and check whether it succeeded or failed. At that moment where you interact with the environment, then and only then can you know whether the interaction was actually possible, and at that point you must commit to performing the interaction. (This is a "synchronisation point", if you will.)

EOF

Now we get to EOF. EOF is the *response* you get from an *attempted* I/O operation. It means that you were trying to read or write something, but when doing so you failed to read or write any data, and instead the end of the input or output was encountered. This is true for essentially all the I/O APIs, whether it be the C standard library, C++ iostreams, or other libraries. As long as the I/O operations succeed, you simply *cannot know* whether further, future operations will succeed. You *must* always first try the operation and then respond to success or failure.

Examples

In each of the examples, note carefully that we *first* attempt the I/O operation and *then* consume the result if it is valid. Note further that we *always* must use the result of the I/O operation, though the result takes different shapes and forms in each example.

- C stdio, read from a file:

```
for (;;) {
    size_t n = fread(buf, 1, bufsize, infile);
    consume(buf, n);
    if (n < bufsize) { break; }
}
```

  The result we must use is `n`, the number of elements that were read (which may be as little as zero).

- C stdio, `scanf`:

```
for (int a, b, c; scanf("%d %d %d", &a, &b, &c) == 3; ) {
    consume(a, b, c);
}
```

  The result we must use is the return value of `scanf`, the number of elements converted.

- C++, iostreams formatted extraction:

```
for (int n; std::cin >> n; ) {
    consume(n);
}
```

  The result we must use is `std::cin` itself, which can be evaluated in a boolean context and tells us whether the stream is still in the `good()` state.

- C++, iostreams getline:

```
for (std::string line; std::getline(std::cin, line); ) {
    consume(line);
}
```

  The result we must use is again `std::cin`, just as before.

- POSIX, `write(2)` to flush a buffer:

```
char const * p = buf;
ssize_t n = bufsize;
for (ssize_t k = bufsize; (k = write(fd, p, n)) > 0; p += k, n -= k) {}
if (n != 0) { /* error, failed to write complete buffer */ }
```

  The result we use here is `k`, the number of bytes written. The point here is that we can only know how many bytes were written *after* the write operation.

- POSIX `getline()`

```
char *buffer = NULL;
size_t bufsiz = 0;
ssize_t nbytes;
while ((nbytes = getline(&buffer, &bufsiz, fp)) != -1)
{
    /* Use nbytes of data in buffer */
}
free(buffer);
```

  The result we must use is `nbytes`, the number of bytes up to and including the newline (or EOF if the file did not end with a newline).

  Note that the function explicitly returns `-1` (and not EOF!) when an error occurs or it reaches EOF.

You may notice that we very rarely spell out the actual word "EOF". We usually detect the error condition in some other way that is more immediately interesting to us (e.g. failure to perform as much I/O as we had desired). In every example there is some API feature that could tell us explicitly that the EOF state has been encountered, but this is in fact not a terribly useful piece of information. It is much more of a detail than we often care about. What matters is whether the I/O succeeded, more-so than how it failed.

- A final example that actually queries the EOF state: Suppose you have a string and want to test that it represents an integer in its entirety, with no extra bits at the end except whitespace. Using C++ iostreams, it goes like this:

```
std::string input = "   123   ";   // example

std::istringstream iss(input);
int value;
if (iss >> value >> std::ws && iss.get() == EOF) {
    consume(value);
} else {
    // error, "input" is not parsable as an integer
}
```

  We use two results here. The first is `iss`, the stream object itself, to check that the formatted extraction to `value` succeeded. But then, after also consuming whitespace, we perform another I/O/ operation, `iss.get()`, and expect it to fail as EOF, which is the case if the entire string has already been consumed by the formatted extraction.

  In the C standard library you can achieve something similar with the `strto*l` functions by checking that the end pointer has reached the end of the input string.

The answer

`while(!feof)` is wrong because it tests for something that is irrelevant and fails to test for something that you need to know. The result is that you are erroneously executing code that assumes that it is accessing data that was read successfully, when in fact this never happened.

share improve this answer follow

edited Mar 17 at 10:33 RobertS supports Monica Cellio 8,666 ●2 ●7 ●34

answered Oct 24 '14 at 22:28 Kerrek SB 408k ●73 ●772 ●992

34 @CiaPan: I don't think that's true. Both C99 and C11 allow this. – Kerrek SB Jan 29 '15 at 12:10

11 But ANSI C does not. – CiaPan Jan 29 '15 at 12:25

3 @JonathanMee: It's bad for all the reasons I mention: you cannot look into the future. You cannot tell what will happen in the future. – Kerrek SB Feb 3 '15 at 20:52

3 @JonathanMee: Yes, that would be appropriate, though usually you can combine this check into the operation (since most iostreams operations return the stream object, which itself has a boolean conversion), and that way you make it obvious that you're not ignoring the return value. – Kerrek SB Feb 3 '15 at 21:02

4 Third paragraph is remarkably misleading/inaccurate for an accepted and highly upvoted answer. `feof()` does not "ask the I/O system whether it has more data". `feof()`, according to the (Linux) *manpage*: "tests the end-of-file indicator for the stream pointed to by stream, returning nonzero if it is set." (also, an explicit call to `clearerr()` is the only way to reset this indicator); In this respect, William Pursell's answer is much better. – Arne Vogel Sep 10 '18 at 11:53

show **19** more comments

234

It's wrong because (in the absence of a read error) it enters the loop one more time than the author expects. If there is a read error, the loop never terminates.

Consider the following code:

```
/* WARNING: demonstration of bad coding technique!! */

#include <stdio.h>
#include <stdlib.h>

FILE *Fopen(const char *path, const char *mode);

int main(int argc, char **argv)
{
    FILE *in;
    unsigned count;

    in = argc > 1 ? Fopen(argv[1], "r") : stdin;
    count = 0;

    /* WARNING: this is a bug */
    while( !feof(in) ) {  /* This is WRONG! */
        fgetc(in);
        count++;
    }
    printf("Number of characters read: %u\n", count);
    return EXIT_SUCCESS;
}

FILE * Fopen(const char *path, const char *mode)
{
    FILE *f = fopen(path, mode);
    if( f == NULL ) {
        perror(path);
        exit(EXIT_FAILURE);
    }
    return f;
}
```

This program will consistently print one greater than the number of characters in the input stream (assuming no read errors). Consider the case where the input stream is empty:

```
$ ./a.out < /dev/null
Number of characters read: 1
```

In this case, `feof()` is called before any data has been read, so it returns false. The loop is entered, `fgetc()` is called (and returns `EOF`), and count is incremented. Then `feof()` is called and returns true, causing the loop to abort.

This happens in all such cases. `feof()` does not return true until **after** a read on the stream encounters the end of file. The purpose of `feof()` is NOT to check if the next read will reach the end of file. The purpose of `feof()` is to distinguish between a read error and having reached the end of the file. If `fread()` returns 0, you must use `feof` / `ferror` to decide whether an error was encountered or if all of the data was consumed. Similarly if `fgetc` returns `EOF`. `feof()` is only useful **after** fread has returned zero or `fgetc` has returned `EOF`. Before that happens, `feof()` will always return 0.

It is always necessary to check the return value of a read (either an `fread()`, or an `fscanf()`, or an `fgetc()`) before calling `feof()`.

Even worse, consider the case where a read error occurs. In that case, `fgetc()` returns `EOF`, `feof()` returns false, and the loop never terminates. In all cases where `while(!feof(p))` is used, there must be at least a check inside the loop for `ferror()`, or at the very least the while condition should be replaced with `while(!feof(p) && !ferror(p))` or there is a very real possibility of an infinite loop, probably spewing all sorts of garbage as invalid data is being processed.

So, in summary, although I cannot state with certainty that there is never a situation in which it may be semantically correct to write "`while(!feof(f))`" (although there **must** be another check inside the loop with a break to avoid a infinite loop on a read error), it is the case that it is almost certainly always wrong. And even if a case ever arose where it would be correct, it is so idiomatically wrong that it would not be the right way to write the code. Anyone seeing that code should immediately hesitate and say, "that's a bug". And possibly slap the author (unless the author is your boss in which case discretion is advised.)

share improve this answer follow

edited Feb 5 at 18:51

answered Mar 25 '11 at 12:39 William Pursell 157k ●43 ●228 ●260

7 Sure it's wrong -- but aside from that it isn't "gratingly ugly". – nobar Jul 7 '13 at 0:53

89 You should add an example of correct code, as I imagine lots of people will come here looking for a quick fix. – jleahy Jul 12 '13 at 16:27

6 @Thomas: I'm not a C++ expert, but I believe file.eof() returns effectively the same result as `feof(file) || ferror(file)`, so it is very different. But this question is not intended to be applicable to C++. – William Pursell Aug 26 '14 at 23:36

6 @m-ric that's not right either, because you'll still try to process a read that failed. – Mark Ransom Apr 9 '15 at 18:25

4 this is the actual correct answer. feof() is used to know the outcome of previous read attempt. Thus probably you don't want to use it as your loop break condition. +1 – Jack Jan 28 '17 at 16:06

show **6** more comments

63

No it's not always wrong. If your loop condition is "while we haven't tried to read past end of file" then you use `while (!feof(f))`. This is however not a common loop condition - usually you want to test for something else (such as "can I read more"). `while (!feof(f))` isn't wrong, it's just *used* wrong.

share improve this answer follow

answered Mar 25 '11 at 11:49 Erik 75.3k ●10 ●179 ●180

1 I wonder ... `f = fopen("A:\\bigfile"); while (!feof(f)) { /* remove diskette */ }` or (going to test this) `f = fopen(NETWORK_FILE); while (!feof(f)) { /* unplug network cable */ }` – pmg Mar 25 '11 at 11:53

1 @pmg: As said, "not a common loop condition" hehe. I can't really think of any case I've needed it, usually I'm interested in "could I read what I wanted" with all that implies of error handling – Erik Mar 25 '11 at 11:56

@pmg: As said, you rarely want `while(!eof(f))` – Erik Mar 25 '11 at 12:41

9 More accurately, the condition is "while we haven't tried to read past the end of the file and there was no read error" `feof` is not about detecting end of file; it is about determining whether a read was short because of an error or because the input is exhausted. – William Pursell Jul 3 '13 at 15:18

add a comment

35

`feof()` indicates if one has tried to read past the end of file. That means it has little predictive effect: if it is true, you are sure that the next input operation will fail (you aren't sure the previous one failed BTW), but if it is false, you aren't sure the next input operation will succeed. More over, input operations may fail for other reasons than the end of file (a format error for formatted input, a pure IO failure -- disk failure, network timeout -- for all input kinds), so even if you could be predictive about the end of file (and anybody who has tried to implement Ada one, which is predictive, will tell you it can complex if you need to skip spaces, and that it has undesirable effects on interactive devices -- sometimes forcing the input of the next line before starting the handling of the previous one), you would have to be able to handle a failure.

So the correct idiom in C is to loop with the IO operation success as loop condition, and then test the cause of the failure. For instance:

```
while (fgets(line, sizeof(line), file)) {
    /* note that fgets don't strip the terminating \n, checking its
       presence allow to handle lines longer that sizeof(line), not showed here */
    ...
}
if (ferror(file)) {
   /* IO failure */
} else if (feof(file)) {
   /* format error (not possible with fgets, but would be with fscanf) or end of fi
} else {
   /* format error (not possible with fgets, but would be with fscanf) */
}
```

share improve this answer follow

edited Jul 10 '19 at 8:15 Antti Haapala 104k ●21 ●209 ●244

answered Feb 10 '12 at 10:22 AProgrammer 46.7k ●7 ●76 ●136

2 Getting to the end of a file is not an error, so I question the phrasing "input operations may fail for other reasons than the end of file". – William Pursell Sep 29 '12 at 12:59

@WilliamPursell, reaching the eof isn't necessarily an error, but being unable to do an input operation because of eof is one. And it is impossible in C to detect reliably the eof without having made an input operation fails. – AProgrammer Sep 29 '12 at 15:12

Agree last `else` not possible with `sizeof(line) >= 2` and `fgets(line, sizeof(line), file)` but possible with pathological `size <= 0` and `fgets(line, size, file)`. Maybe even possible with `sizeof(line) == 1`. – chux - Reinstate Monica Mar 26 '15 at 21:21

1 All that "predictive value" talk... I never thought about it that way. In my world, `feof(f)` does not PREDICT anything. It states that a PREVIOUS operation has hit the end of the file. Nothing more, nothing less. And if there was no previous operation (just opened it), it does not report end of file even if the file was empty to start with. So, apart of the concurrency explanation in another answer above, I do not think there is any reason not to loop on `feof(f)`. – BitTickler Sep 24 '17 at 20:36

@AProgrammer: A "read up to N bytes" request that yields zero, whether because of a "permanent" EOF or because no more data is available *yet*, is not an error. While feof() may not reliably predict that future requests will yield data, it may reliably indicate that future requests *won't*. Perhaps there should be a status function that would indicate "It is plausible that future read requests will succeed", with semantics that after reading to the end of an ordinary file, a quality implementation should say future reads are unlikely to succeed *absent some reason to believe they might*. – supercat Feb 3 '19 at 15:49

show **1** more comment

0

`feof()` is not very intuitive. In my very humble opinion, the `FILE`'s end-of-file state should be set to `true` if any read operation results in the end of file being reached. Instead, you have to manually check if the end of file has been reached after each read operation. For example, something like this will work if reading from a text file using `fgetc()`:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *in = fopen("testfile.txt", "r");

  while(1) {
    char c = fgetc(in);
    if (feof(in)) break;
    printf("%c", c);
  }

  fclose(in);
  return 0;
}
```

It would be great if something like this would work instead:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *in = fopen("testfile.txt", "r");

  while(!feof(in)) {
    printf("%c", fgetc(in));
  }

  fclose(in);
  return 0;
}
```

share improve this answer follow

answered yesterday Scott Deagan 71 ●7

1 `printf("%c", fgetc(in));`? That's undefined behavior. `fgetc()` returns `int`, not `char`. – Andrew Henle yesterday

It seems to me that the standard idiom `while( (c = getchar()) != EOF)` is very much "something like this". – William Pursell yesterday

`while( (c = getchar()) != EOF)` works on one of my desktop running GNU C 10.1.0, but fails on my Raspberry Pi 4 running GNU C 9.3.0. On my RPi4, it doesn't detect the end of file, and just keeps going. – Scott Deagan yesterday

@AndrewHenle You're right! Changing `char c` to `int c` works! Thanks!! – Scott Deagan 5 hours ago

add a comment

Highly active question. Earn 10 reputation in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.

Not the answer you're looking for? Browse other questions tagged c file while-loop eof feof or ask your own question.