C and C++ Programming:

“eof” is never correct

I/O is dealing with the *unknown*.

Anything can happen.

You cannot know in advance.

Do not bother checking whether something *may* succeed.

This information is useless.

```
if (feof(fp)) { /* ... */ } // useless (FILE * fp)

if (in.eof()) { /* ... */ } // useless (std::istream & in)
```

You *cannot* predict whether the next input operation will succeed.

Don’t even try!

Instead: *Just do it.*

And *after you have tried, check if you succeeded.*

Assume we have: `FILE * fp = fopen("data.txt", "rb");`

First attempt:

```
unsigned char buf[50];  // want to read 50 bytes
if (fread(buf, sizeof buf, 1, fp) == sizeof buf) { /* success */ }
```

Correct, but may fail: `fread` may not read the requested number of bytes at once. Better in a *loop*:

```
for (size_t n, total = sizeof buf;
     (n = fread(buf, total, 1, fp)) > 0;
      total -= n) { }

if (total > 0) { /* error, did not read the desired amount */ }
```

Note the pattern: Read first, then see what you got.

The same works for formatted extraction:

```
if (fscanf(fp, "%d %d %d", &a, &b, &c) != 3) {
    /* error */
} else {
    /* OK to use a, b, c */
}
```

Again the pattern: Attempt the extraction first, then check.

Assume that it is *undefined behaviour* to access the destination variables unless the extraction succeeded!

(This may not actually be the case, but it’s a good way to think.)

Output is similar – it may not succeed. Make sure to check the result:

```
unsigned char buf[50];  // to be written, populated

for (size_t n, total = sizeof buf;
     (n = fwrite(buf, total, 1, fp)) > 0;
      total -= n) { }

if (total > 0) { /* error, did not write all the data */ }
```

In C++ with iostreams, stream objects tell you if they are “good”:

```
std::ifstream infile("data.txt");

if (infile) {
    /* infile is good */
} else {
    /* not good, e.g. could not open the file */
}
```

No operations succeed on a stream that is bad (i.e. not good).

Most iostream operations return the stream itself:

```
infile >> n;                   // returns infile
std::getline(infile, str);     // returns infile
infile.read(buf, sizeof buf);  // returns infile
```

Use this to your advantage!

```
if (infile >> n)               { /* OK to use n */ }
if (std::getline(infile, str)) { /* OK to use str */ }

if (infile.read(buf, sizeof buf) &&
    infile.gcount() == sizeof buf) { /* OK */ }
```

The use of `gcount()` is as before: reading may result in partial data, and we must also check if we got the desired amount. Again, best in a loop.

I thought you would never ask.

There are *some* useful operations that return an *integer* of value `EOF`:

- `fgetc(fp)`
- `infile.get()`

These attempt to read a *single character* from the stream, and return `EOF` when there is nothing more to read.

Warning: `EOF` is never a valid character! You must not convert the result of the extraction to `char` (or whatever suitable character type) until *after* you have compared it against the integer `EOF`.

```
int n = fgetc(fp);                // must be "int"!
if (n != EOF) { char c = n; }  // convert only when good
```

Let’s be clear: We were just talking about an integral constant, and *not* a function.

It is the *functions* with “eof” in their name that are never correct.

The integral constant has its place and its uses.

Grand finale: Loop over the lines in a file, and parse those lines which contain precisely three integers and nothing else.

```
for (std::string line; std::getline(infile, line); ) {
    std::istringstream iss(line);
    int a, b, c;
    if (iss >> a >> b >> c >> std::ws && iss.get() == EOF) {
        consume(a, b, c);
    } else {
        // failed to parse line
        // must NOT attempt to access a, b, c
    }
}
```

Notes:

- `iss >> a >> b >> c >> std::ws` parses three integers and gobbles up final whitespace; it is "good" if the extraction succeeded.
- Since we expect to have consumed the entire line, `iss.get()` should now result in `EOF`.

End of message.