A: `feof` will only return a nonzero value when a previous read operation has failed: you can't use `feof` to check for end-of-file *before* attempting to read. Instead, you should first attempt to read, then check the return value from the input function. If the return value indicates that the operation was unsuccessful, you can then use `feof` to determine whether the failure was due to end-of-file. In other words, it's best not to think of calling `feof` as a way to *detect* end-of-file. Instead, think of it as a way to *confirm* that end-of-file was the reason for the failure of a read operation.

Q: I still don't understand why the I/O library provides macros named `putc` and `getc` in addition to functions named `fputc` and `fgetc`. According to Section 21.1, there are already two versions of `putc` and `getc` (a macro and a function). If we need a genuine function instead of a macro, we can expose the `putc` or `getc` function by undefining the macro. So why do `fputc` and `fgetc` exist? [p. 566]

A: Historical reasons. Prior to standardization, C had no rule that there be a true function to back up each parameterized macro in the library. `putc` and `getc` were traditionally implemented only as macros; `fputc` and `fgetc` were implemented only as functions.

***Q: What's wrong with storing the return value of `fgetc`, `getc`, or `getchar` in a `char` variable? I don't see how testing a `char` variable against `EOF` could give the wrong answer. [p. 568]**

A: There are two cases in which this test can give the wrong result. To make the following discussion concrete, I'll assume two's-complement arithmetic.

First, suppose that `char` is an unsigned type. (Recall that some compilers treat `char` as a signed type but others treat it as an unsigned type.) Now suppose that `getc` returns `EOF`, which we store in a `char` variable named `ch`. If `EOF` represents –1 (its typical value), `ch` will end up with the value 255. Comparing `ch` (an unsigned character) with `EOF` (a signed integer) requires converting `ch` to a signed integer (255, in this case). The comparison against `EOF` fails, since 255 is not equal to –1.

Now assume that `char` is a signed type instead. Consider what happens if `getc` reads a byte containing the value 255 from a binary stream. Storing 255 in the `ch` variable gives it the value –1, since `ch` is a signed character. Testing whether `ch` is equal to `EOF` will (erroneously) give a true result.

Q: The character input functions described in Section 22.4 require that the Enter key be pressed before they can read what the user has typed. How can I write a program that responds to individual keystrokes?

A: As you've noticed, the `getc`, `fgetc`, and `getchar` functions are buffered; they don't start to read input until the user has pressed the Enter key. In order to read characters as they're entered—which is important for some kinds of programs—you'll need to use a nonstandard library that's tailored to your operating system. In UNIX, for example, the `curses` library often provides this capability.