Chapter 1: A Tutorial Introduction

Page 5: I completely agree with the authors that writing real programs, and soon, is the best way to learn programming. This way, concepts which would otherwise seem abstract make sense, and the positive feedback you get from getting even a small program to work gives you a great incentive to improve it or write the next one.

Diving in with “real” programs right away has another advantage, if only pragmatic: if you’re using a conventional compiler, you can’t run a fragment of a program and see what it does; nothing will run until you have a complete (if tiny or trivial) program. You can’t learn everything you’d need to write a complete program all at once, so you’ll have to take some things “on faith” and parrot them in your first programs before you begin to understand them. (You can’t learn to program just one expression or statement at a time any more than you can learn to speak a foreign language one word at a time. If all you know is a handful of words, you can’t actually say anything: you also need to know something about the language’s word order and grammar and sentence structure and declension of articles and verbs.)

The authors list a few drawbacks of this “dive in and program” approach, and I must add one more. It’s a small step from learning-by-doing to learning-by-trial-and-error, and when you learn programming by trial-and-error, you can very easily learn many errors. When you’re not sure whether something will work, or you’re not even sure what you could use that might work, and you try something, and it does work, you do not have any guarantee that what you tried worked for the right reason. You might just have “learned” something that works only by accident or only on your compiler, and it may be very hard to un-learn it later, when it stops working. (Also, if what you tried didn’t work, it may have been due to a bug in the compiler, such that it should have worked.)

Therefore, whenever you’re not sure of something, be very careful before you go off and try it “just to see if it will work.” Of course, you can never be absolutely sure that something is going to work before you try it, otherwise we’d never have to try things. But you should have an expectation that something is going to work before you try it, and if you can’t predict how to do something or whether something would work and find yourself having to determine it experimentally, make a note in your mind that whatever you’ve just learned (based on the outcome of the experiment) is suspect.

Section 1.1: Getting Started

Page 6: Deep sentence: With these mechanical details mastered, everything else is comparatively easy.

The claim that a program as simple as “hello, world” is a big hurdle may seem outrageous, but it’s really quite true. It is a hurdle: on an unfamiliar computer, it can be arbitrarily difficult to figure out how
to enter a text file containing program source, or how to compile and link it, or how to invoke it, or what happened after (if?) it ran. The most experienced C programmers immediately go back to this one, simple program whenever they’re trying out a new system or a new way of entering or building programs or a new way of printing output from within programs. As they say, everything else is comparatively easy.

One hurdle which the authors don’t mention but which many of you may find yourself facing is the choice of an appropriate compiler. On many Unix machines, the cc command which the authors describe is an older compiler which does not recognize modern, ANSI Standard C syntax. An old compiler will accept the simple program on page 6, but it will not accept many of the other programs in the book. If you find yourself getting baffling compilation errors on programs which you’ve typed in exactly as they’re shown in the book, it probably indicates that you’re using an older compiler. On many machines, another compiler called acc or gcc is available, and you’ll want to use it, instead.

Deep sentence: main will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you.

We heard about this already in the Introduction, but here it is again: as far as the compiler and the language definition are concerned, there’s no difference between a function that you write and a function someone else wrote for you, including a function like printf which seems to be part of the language. There’s nothing magic about printf; there’s nothing that it can do that one of your functions couldn’t. (Well, actually, there are a few magic, or at least surprising, things about printf, but they’re magic in ways that your functions can be, too.)

There is one slight problem with the simple “hello, world” program in the book. The problem will usually be ignored (that is, the program will usually work correctly), but if you receive any warning or error messages or have any problems having to do with the “value returned from main,” jump forward to page 26 to learn why main ought to end with the line

```
return 0;
```

**Section 1.2: Variables and Arithmetic Expressions**

Page 10: Deep sentence: Although C compilers do not care about how a program looks, proper indentation and spacing are critical in making programs easy for people to read. We recommend writing only one statement per line, and using blanks around operators to clarify grouping. The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently.

There are two things to note here. One is that (with one or two exceptions) the compiler really does not care how a program looks; it doesn’t matter how it’s broken into lines. The fragments

```
while(i < j)
    i = 2 * i;
```

and

```
while(i < j) i = 2 * i;
```
and

    while (i < j) i = 2 * i;

and

    while (i < j)
    i = 2 * i;

and

    while  (i < j)
    i = 2 * i;

are all treated exactly the same way by the compiler.

The second thing to note is that style issues (such as how a program is laid out) are important, but they’re not something to be too dogmatic about, and there are also other, deeper style issues besides mere layout and typography.

There is some value in having a reasonably standard style (or a few standard styles) for code layout. Please don’t take the authors’ advice to “pick a style that suits you” as an invitation to invent your own brand-new style. If (perhaps after you’ve been programming in C for a while) you have specific objections to specific facets of existing styles, you’re welcome to modify them, but if you don’t have any particular leanings, you’re probably best off copying an existing style at first. (If you want to place your own stamp of originality on the programs that you write, there are better avenues for your creativity than inventing a bizarre layout; you might instead try to make the logic easier to follow, or the user interface easier to use, or the code freer of bugs.)

Deep sentence: ...in C, as in many other languages, integer division truncates: any fractional part is discarded.

The authors say all there is to say here, but remember it: just when you’ve forgotten this sentence, you’ll wonder why something is coming out zero when you thought it was supposed the be the quotient of two nonzero numbers.

Page 12: Here is more discussion on the difference between integer and floating-point division. Nothing deep; just something to remember.

Page 13: Hidden here are descriptions of some more of printf’s “conversion specifiers.” %o and %x print integers, in octal (base 8) and hexadecimal (base 16), respectively. Since a percent sign normally tells printf to expect an additional argument and insert its value, you might wonder how to get printf to just print a %. The answer is to double it: %%. 

Also, note (as was mentioned on page 11) that you must match up the arguments to printf with the conversion specification; the compiler can’t (or won’t) generally check them for you or fix things up if you get them wrong. If fahr is a float, the code
printf("%d\n", fahr);

will not work. You might ask, “Can’t the compiler see that %d needs an integer and fahr is floating-point and do the conversion automatically, just like in the assignments and comparisons on page 12?” And the answer is, no. As far as the compiler knows, you’ve just passed a character string and some other arguments to printf; it doesn’t know that there’s a connection between the arguments and some special characters inside the string. This is one of the implications of the fact, stated earlier, that functions like printf are not special. (Actually, some compilers or other program checkers do know that a function named printf is special, and will do some extra checking for you, but you can’t count on it.)

Section 1.3: The For Statement

Pages 13-14: Deep sentence: ...in any context where it is permissible to use the value of a variable of some type, you can use a more complicated expression of that type.

You may have used other languages which placed restrictions on where you could use expressions or how complicated they could be. C has relatively few such restrictions. There’s nothing magical about the printf call above; this ability to perform a computation inside of an argument is not unique to printf. In any function call, the arguments in the argument list are expressions, and it doesn’t matter if they are simple expressions which just fetch the value of one variable, like fahr, or more complicated expressions, like \[ \frac{5.0}{9.0} \times (\text{fahr} - 32). \]

Section 1.4: Symbolic Constants

Pages 14-15: Deep sentence: Notice that there is no semicolon at the end of a #define line.

Actually, all lines that begin with # are special; we’ll learn more about them later.

Section 1.5: Character Input and Output

Page 15: Note that you do not need to worry about whether your computer uses a carriage return (CR) or linefeed (LF) or CRLF combination or something else to terminate lines in text files; in a C program, the line terminator will always appear to be the newline, \n.

Section 1.5.1: File Copying

Page 16: Pay particular attention to the discussion of why the variable to hold getchar’s return value is declared as an int rather than a char. The distinction may not seem terribly significant now, but it is important. If you use a char, it may seem to work, but it may break down mysteriously later. Always remember to use an int for anything you assign getchar’s return value to.

Page 17: The line

\[
\text{while } ((\text{c} = \text{getchar})()) \neq \text{EOF}\]

epitomizes the cryptic brevity which C is notorious for. You may find this terseness infuriating (and you’re not alone!), and it can certainly be carried too far, but bear with me for a moment while I defend it.

The simple example on pages 16 and 17 illustrates the tradeoffs well. We have four things to do:

call getchar,
assign its return value to a variable,
test the return value against EOF, and
process the character (in this case, print it again).

We can’t eliminate any of these steps. We have to assign getchar’s value to a variable (we can’t just use it directly) because we have to do two different things with it (test, and print). Therefore, compressing the assignment and test into the same line (as on page 17) is the only good way of avoiding two distinct calls to getchar (as on page 16). You may not agree that the compressed idiom is better for being more compact or easier to read, but the fact that there is now only one call to getchar is a real virtue.

In a tiny program like this, the repeated call to getchar isn’t much of a problem. But in a real program, if the thing being read is at all complicated (not just a single character read with getchar), and if the processing is at all complicated (such that the input call before the loop and the input call at the end of the loop become widely separated), and if the way that input is done is ever changed some day, it’s just too likely that one of the input calls will get changed but not the other.

(Also, note that when an assignment like c = getchar() appears within a larger expression, the surrounding expression receives the same value that is assigned. Using an assignment as a subexpression in this way is perfectly legal and quite common in C.)

When you run the character copying program, and it begins copying its input (your typing) to its output (your screen), you may find yourself wondering how to stop it. It stops when it receives end-of-file (EOF), but how do you send EOF? The answer depends on what kind of computer you’re using. On Unix and Unix-related systems, it’s almost always control-D. On MS-DOS machines, it’s control-Z followed by the RETURN key. Under Think C on the Macintosh, it’s control-D, just like Unix. On other systems, you may have to do some research to learn how to send EOF.

(Note, too, that the character you type to generate an end-of-file condition from the keyboard has nothing to do with the EOF value returned by getchar. The EOF value returned by getchar is a code indicating that the input system has detected an end-of-file condition, whether it’s reading the keyboard or a file or a magnetic tape or a network connection or anything else.)

Another excellent thing to know when doing any kind of programming is how to terminate a runaway program. If a program is running forever waiting for input, you can usually stop it by sending it an end-of-file, as above, but if it’s running forever not waiting for something (i.e. if it’s in an infinite loop) you’ll have to take more drastic measures. Under Unix, control-C will terminate the current program, almost no matter what. Under MS-DOS, control-C or control-BREAK will sometimes terminate the current program, but by default MS-DOS only checks for control-C when it’s looking for input, so an infinite loop can be unkillable. There’s a DOS command, I think it’s
break on

which tells DOS to look for control-C more often, and I recommend using this command if you’re doing any programming. (If a program is in a really tight infinite loop under MS-DOS, there can be no way of killing it short of rebooting.) On the Mac, try command-period or command-option-ESCAPE.

Finally, don’t be disappointed (as I was) the first time you run the character copying program. You’ll type a character, and see it on the screen right away, and assume it’s your program working, but it’s only your computer echoing every key you type, as it always does. When you hit RETURN, a full line of characters is made available to your program, which it reads all at once, and then copies to the screen (again). In other words, when you run this program, it will probably seem to echo the input a line at a time, rather than a character at a time. You may wonder how a program can read a character right away, without waiting for the user to hit RETURN. That’s an excellent question, but unfortunately the answer is rather complicated, and beyond the scope of this introduction. (Among other things, how to read a character right away is one of the things that’s not defined by the C language, and it’s not defined by any of the standard library functions, either. How to do it depends on which operating system you’re using.)

Section 1.5.2: Character Counting

Page 18: Ignore the mention of efficiency with respect to nc = nc+1 vs. ++nc. Once you’ve gotten used to ++ meaning “increment by 1,” you’ll probably find yourself preferring ++nc simply because it is more concise, and incrementing things by 1 is so common. (Personally, once I got used to it, I found ++ more natural, too, because after all, expressions like nc = nc+1, though they’re common enough in programming, are very unnatural from an algebraic perspective.)

Pages 18-19: You may find it odd to have a loop with no body, but such loops do crop up. Just make sure that the explicit null statement (or, if you prefer, empty {}) marking the empty loop body is plainly visible.

The whole first paragraph of page 19 counts as “deep.” A clean, well-designed loop will work properly for all of its “boundary conditions”: zero trips through the loop, one trip, many trips, maximum trips (if there is any maximum, and if so, also maximum minus one). If a loop for some reason doesn’t work at a particular boundary condition, it’s tempting to claim that that condition is rare or impossible and that the loop is therefore okay. But if the loop can’t handle the boundary condition, why can’t it? It’s probably awkwardly constructed, and straightening it out so that it naturally handles all boundary conditions will usually make it clearer and easier to understand (and may also remove other lurking bugs).

Section 1.5.3: Line Counting

Page 19: Note the word of caution about = vs. == carefully. Typing one when you mean the other is, unfortunately, a very easy mistake to make.

Note that the character constants discussed on page 19 are very different from the string constants introduced on page 7.

Section 1.5.4: Word Counting
Page 21: Deep sentence: In a program as tiny as this, it makes little difference, but in larger programs, the increase in clarity is well worth the modest extra effort to write it this way from the beginning.

I agree with this. Some people complain that symbolic constants make a program harder to read, because you always have to look them up to see what they mean. As long as you choose appropriate names for symbolic constants and use them consistently (i.e. even if APPLE and ORANGE happen to have the same value, don’t use one when you mean the other), no one will have this complaint about your programs.

Note that there’s no direct way to simplify the condition

```c
if (c == ' ' || c == '
' || c == '	')
```

In particular, something like

```c
if (c == (' ' || '
' || '	'))
```

would not work. (What would it do?)

Section 1.6: Arrays

Page 22: Note carefully that arrays in C are 0-based, not 1-based as they are in some languages. (As we’ll see, 0-based arrays turn out to be more convenient than 1-based arrays more of the time, but they may take a bit of getting used to at first.)

When they say “as reflected in the for loops that initialize and print the array,” they’re referring to the fact that the vast majority of for loops in C look like this:

```c
for(i = 0; i < 10; ++i)
```

and count from 0 to 10. The loop

```c
for(i = 1; i <= 10; ++i)
```

would count from 1 to 10, but loops like this are comparatively rare. (In fact, whenever you see either “= 1” or “<=” in a for loop, it’s an indication that something unusual is going on which you’ll want to be aware of, and it may even be a bug.)

Page 23: They’ve started going a little fast here, so read up if they’re losing you. What’s this magic expression c-'0' that they’re using as an array subscript? Remember, as we saw first on page 19, that characters in C are represented by small integers corresponding to their values in the machine’s character set. In ASCII, which most machines use, ‘A’ is character code 65, ‘0’ (zero) is code 48, ‘9’ is code 57, and all the other characters have their own values which I won’t bother to list. If we’ve just read the character ‘9’ from the file, it has value 57, so c-'0' is 57 - 48 which is 9, and we’ll increment cell number 9 in the array, just like we want to. Furthermore, even if we’re not using a machine which uses ASCII, by subtracting ‘0’, we’ll always subtract whatever the right value is to map the characters from ‘0’ to ‘9’ down to the array cell range 0 to 9.
Section 1.7: Functions

Page 24: Deep sentence: ...you will often see a short function defined and called only once, just because it clarifies some piece of code.

Ideally, this is true in any language. Breaking a program up into functions (or subroutines or procedures or whatever a language calls them) is one of the first and one of the most important ways to keep control of the proliferating complexity in a software project.

Page 25: Note that the for loop at the top of the page runs from 1 to n rather than 0 to n-1, and may therefore seem suspect by the above note for page 22. In this case, since all that matters is that the loop is traversed n times, it doesn’t matter which values i takes on.

Not only the names of the parameters and local variables, but also their values (as we’ll see in section 1.8), are all local to a function. Rather than remembering a list of things that are local, it’s easier to remember that everything is local: the whole point of a function as an abstraction mechanism is that it’s a black box; you don’t have to know or care about any of its implementation details, such as what it chooses to name its parameters and local variables. You pass it some arguments, and it returns you a value according to its specification.

The distinction between the terms argument and parameter may seem overly picky, but it’s a good way of reinforcing the notion that the parameters and other details of a function’s implementation are almost completely separated from (that is, of no concern to) the caller.

Page 26: Note the discussion about return values from main. The first few sample programs in this chapter, including the very first “hello, world” example on page 6, have omitted a return value, which is, strictly speaking, incorrect. Do get in the habit of returning a value from main, both to be correct, and because “programs should return status to their environment.”

By “Parameter names need not agree” they mean that it’s not a problem that the prototype declaration of power says that the first parameter is named m, while the actual function definition that it’s named base.

Pages 26-7: It’s probably a good idea if you’re aware of this “old style” function syntax, so that you won’t be taken aback when you come across it, perhaps in code written by reactionary old fogies (such as the author of these notes) who still tend to use it out of habit when they’re not paying attention.

Section 1.8: Arguments -- Call by Value

Page 27: If, on the other hand, you are not used to other languages such as Fortran, these call-by-value semantics may not be surprising (any more than anything else in C which is new to you).

Even though you can modify a parameter in a function (i.e. treat it as a “conveniently initialized local variable”), you certainly don’t have to, especially if (as is often the case) you’ll need an unmodified copy of the parameter later in the function.

Page 28: Don’t worry too much about the exception mentioned for arrays--there are a number of exceptions for arrays, and we’ll have much more to say about them later. But be aware that we are
deliberately glossing over a few details here, and they are details which will be come important later on. (In particular, the statement on page 27 that “the called function cannot directly alter a variable in the calling function” may not seem to be true for arrays, and this is what the authors mean when they say that “The story is different”. We’ll be seeing several functions which return things—usually strings—to their callers by writing into caller-supplied arrays. In chapter 5 we’ll learn how this is possible. If this discrepancy wouldn’t have bothered you now, pretend I didn’t mention it.)

Section 1.9: Character Arrays

Pay attention to the way this program is developed first in “pseudocode,” and then refined into real C code. A clear pseudocode statement not only makes it easier to think about the structure of the eventual real code, but if you make the eventual real code mimic the pseudocode, the real code will be equally straightforward and easy to read.

The function getline, introduced here, is extremely useful, and we’ll have as much use for it in our own programs as the authors do in theirs. (In other words, they have succeeded in their goal of making it “useful in other contexts.” In fact, I’ve been using a getline function much like this one ever since I learned C from K&R, and I generally find it preferable to the standard library’s line-reading function.)

Pages 28 through 30 introduce quite a lot of material all at once; you’ll probably want to read it several times, especially if arrays or character strings are new to you.

Earlier we said that C provided no particular built-in support for composite objects such as character strings, and here we begin to see the significance of that omission. A string is just an array of characters, and you can access the characters within a string exactly as easily (because you use exactly the same syntax) as you access the elements within any other array.

If you’ve used BASIC, you will probably wonder where C’s SUBSTR function is. C doesn’t have one, for two reasons. First of all, there’s less of a need for one, because it’s so easy to get at the individual characters within a string in C. More importantly, a SUBSTR function implies that you take a string and extract a substring as a new string. However, creating a new string (i.e. the extracted substring) involves allocating arbitrary amounts of memory to hold the string, and C rarely if ever allocates memory implicitly for you.

If anything, it’s too easy to access the individual characters within strings in C. String handling illustrates one of the potentially frustrating aspects of C we mentioned earlier: the language doesn’t define any high-level string handling features for you, so you’re free to do whatever low-level string processing you wish. The down side is that constantly manipulating strings down at the character level, and always having to remember to allocate memory for new strings, can get tedious after a while.

The preceding paragraph is not meant to discourage you, but just to point out a reality: any C program which manipulates strings (and this includes most C programs) will find itself doing a certain amount of character-level fiddling and a certain amount of memory allocation. It will also find that it can do just about anything it wants to do (and that its programmer has the patience to do) with the strings it manipulates.

Since string processing, and at this relatively low level, is so common in C, you’ll want to pay careful attention to the discussion on page 30 of how strings are stored in character arrays, and particularly to
the fact that a `\0` character is always present to mark the end of a string. (It’s easy to forget to count
the `\0` character when allocating space for a string, for instance.) Notice the nice picture on page 30;
this is a good way of thinking about data structures (and not just simple character arrays, either).

Page 29: Note that the program explicitly allocates space for the two strings it manipulates: the current
line line, and the longest line longest. (It only needs these two strings at any one time, even though the
input consists of arbitrarily many lines.) Note that it cannot simply assign one string to another (because
C provides no built-in support for composite objects such as character strings); the program calls the
copy function to do so. (The authors write their own copy function for explanatory purposes; the
standard library contains a string-copying function which would normally be used.) The only strings that
aren’t explicitly allocated are the arrays in the getline and copy functions; as the discussion briefly
mentions, these do not need to be allocated because they’re already allocated in the caller. (There are
a number of subtleties about array parameters to functions; we’ll have more to say about them later.)

The code on page 29 contains a number of examples of compressed assignments and tests;
evidently the authors expect you to get used to this style in a hurry. The line:

```c
while ((len = getline(line, MAXLINE)) > 0)
```

is similar to the getchar loops earlier in this chapter; it calls getline, saves its return value in the
variable len, and tests it against 0.

The comparison

```c
i<lim-1 && (c=getchar())!=EOF && c!='\n'
```

in the for loop in the getline function does several things: it makes sure there is room for another
character in the array; it calls, assigns, and tests getchar’s return value against EOF, as before; and it also
tests the returned character against `\n`, to detect end of line. The surrounding code is mildly clumsy in
that it has to check for `\n` a second time; later, when we learn more about loops, we may find a way of
writing it more cleanly. You may also notice that the code deals correctly with the possibility that EOF is
seen without a `\n`.

The line

```c
while (((to[i] = from[i]) != '\0')
```

in the copy function does two things at once: it copies characters from the from array to the to
array, and at the same time it compares the copied character against `\0`, so that it stops at the end of
the string. (If you think this is cryptic, wait ‘til we get to page 106 in chapter 5!)

We’ve also just learned another printf conversion specifier: %s prints a string.

Page 30: Deep sentence: There is no way for a user of getline to know in advance how long an input line
might be, so getline checks for overflow.

Because dynamically allocating memory for arbitrary-length strings is mildly tedious in C, it’s
tempting to use fixed-size arrays. (It’s so tempting, in fact, that that’s what most programs do, and since
fixed-size arrays are also considerably easier to discuss, all of our early example programs will use them.)

Using fixed-size arrays is fine, as long as some assurance is made that they don’t overflow. Unfortunately, it’s also tempting (and easy) to forget to guard against array overflow, perhaps by deluding yourself into thinking that too-long inputs “can’t happen.” Murphy’s Law says that they do happen, and the various corollaries to Murphy’s Law say that they happen in the most unpleasant way and at the least convenient time. Don’t be cavalier about arrays; do make sure that they’re big enough and that you guard against overflowing them. (In another mark of C’s general insensitivity to beginning programmers, most compilers do not check for array overflow; if you write more data to an array than it is declared to hold, you quietly scribble on other parts of memory, usually with disastrous results.)

Section 1.10: External Variables and Scope

Page 31: There’s a bit of jargon in this section. An external variable is what is sometimes called a global variable. The authors introduce the term automatic to refer to the local variables we’ve seen so far; this is a good word to remember, even if you never use it, because people will spring it on you when they’re being precise, and if you don’t know this usage you’ll think they’re talking about transmissions or something. (To be precise, “local” is a broader category than “automatic”; there are both automatic and static local variables.)

Deep sentence: If [automatic variables] are not set, they will contain garbage.

Actually, if automatic variables always contained garbage, the situation wouldn’t be quite so bad. In practice, they often (though not always) do contain zero or some other predictable value, and this happens just often enough to lull you into the occasional false sense of security, by making a program with an inadvertently uninitialized variable seem to work.

Deep sentence: An external variable must be defined, exactly once, outside of any function; this sets aside storage for it. The variable must also be declared in each function that wants to access it; this states the type of the variable.

The basic rule is “define once; declare many times.” As we’ll see just below, it is not necessary for a declaration of an external variable to appear in every single function; it is possible for one external declaration to apply to many functions. (In the clause “the variable must also be declared in each function”, the word “declared” is an adjective, not a verb.)

Page 33: In fact, the “common practice” of placing “definitions of all external variables at the beginning of the source file” is so common that it’s rare to see external declarations within functions, as in the functions on page 32. The authors are using the in-function extern declarations partly because it is an alternative style, and partly because we haven’t talked about separate compilation (that is, building a single program from several separate source files) yet. Rather than jumping the gun and discussing those two topics now, I’ll just mention that the discussion in section 1.10 might be a bit misleading, and that you should probably wait until we get to the complete description of the issue in section 4.4 before you commit any of this to memory.

Deep sentence: You should note that we are using the words definition and declaration carefully when we refer to external variables in this section. “Definition” refers to the place where the variable is created or assigned storage; “declaration” refers to places where the nature of the variable is stated but no storage is allocated.
Do note the careful distinction; it’s an important one and one which I’ll be using, too.

Page 34: The authors’ criticism of the second (page 32) version of the longest-line program is accurate. The revision of the longest-line program to use external variables was done only to demonstrate the use of external variables, not to improve the program in any way (nor does it improve the program in any way).

As a general rule, external variables are acceptable for storing certain kinds of global state information which never changes, which is needed in many functions, and which would be a nuisance to pass around. I don’t think of external variables as “communicating between functions” but rather as “setting common state for the entire program.” When you start thinking of an external variables as being one of the ways you communicate with a particular function, and in particular when you find yourself changing the value of some external variable just before calling some function, to affect its operation in some way, you start getting into the troublesome uses of external variables, which you should avoid.