C Programming Style

Paul Krzyzanowski

Introduction

The past several years of teaching computer science courses to classes consisting largely of juniors, seniors, and graduate students have forced me to look at hundreds of samples of student code. Since most of this code was written for programming assignments, I must assume that the submitted assignment was a student's best effort. Many of the assignments "worked" in that they performed the task requested by the assignment. A surprising number of submissions did not. The assignments that worked were generally unforgiving of improper input data and often core dumped when presented with such data.

Looking at the source code, I often saw a complete absence of anything that may resemble *style*. Less than one percent of larger programs were ever broken up into multiple files. Worse yet, most students were reluctant to ever split their algorithm into several functions; modular decomposition seems to be largely unknown.

The purpose of this essay is to provide opinions on programming style. You may agree with some things that I have to say and disagree with others, but my goal is to encourage you to think about the style of your program.

This essay is influenced by Rob Pike's *Notes on Programming in C* and Kernighan and Plauger's *The Elements of Programming Style*. It is also influenced by the hundreds or thousands of programs I've had to look at, work with, or write over the past seventeen or so years.

Strive for clarity

Above all else, try to keep your programs simple and clear. This is easier said than done, especially if you have not seen many programs or have not written many yourself. It is easy to get immersed in a programming project and maintain a large amount of mental state while programming. During this time, the meaning of every variable and purpose of every function is clear. This mental model erodes rather quickly once the project is completed. Will the functions, data structures, and variable names still make sense to you in the future? More importantly perhaps, will they make sense to someone else who has to figure out your program and possibly modify it?

Don't over-comment

There is a style of programming that believes that every line of source code should have a comment associated with it. This is simply nonsense. When programming, you should assume that your program will be read in the future by someone versed in the language and basic algorithms. There is no need to explain the language syntax or basic algorithms. If you use good, clear variable and function naming and structure your code aesthetically, a programmer should have no trouble reading your code. Comments often get in the way of readability, particularly if the occupy lines between code. Comments that merely paraphrase code are not of much use. The classic example is:

```
i = i + 1;  /* add one to i */
```

Even more complex code segments are not necessarily worthy of comments:

```
for (s=buf; *s && (*s != '@'); ++s) /* search for the first @ /
        ;
```

If your code is so obscure that it needs a comment to be understood, it may be worth thinking about rewriting it. Comments also suffer from obsolescence. Code may evolve, but the comments are often not updated.

Now that comments have been put down, it's time to defend them. Your program *should* have comments. It is useful to explain global variables, structures, and data types. Comments are also useful in introducing a function, particularly one whose purpose isn't immediately obvious from its name. Introductory comments should also serve to document any side-effects produced by the function. Comments on central data structures are generally more useful than comments on algorithms.

Write your program from the top down

Recall that you're assuming your program will be read by someone. It makes logical sense for the main functions to appear first and support functions to appear later. A programmer can easily get to the core algorithm and read on further if necessary to get the details. The opposite method (and one generally encouraged by the Pascal school) forces the programmer to have to wade through the details firstt.

Choose variable names sensibly

Daniel D. McCracken's 1961 *a guide to FORTRAN programming* has this to say about variable names:

> The name of a fixed point variable is one to six digits or letters, of which the first is I, J, K, L, M, or N. Examples of acceptable names of fixed point variables: I, KLM, MATRIX, L123, I6M2K.

Luckily, those days are behind us. Unfortunately, we see more and more code that looks like[1]:

```
DWORD __stdcall OnW32DeviceIoControl(PDIOCPARAMETERS p) {
switch (p->dwIoControlCode) {
       case DIOC_GETVERSION:
              return 0;
       ...
       case 1:
              if (!p->lpvOutBuffer || p->cbOutBuffer < 2)
                     return ERROR_INVALID_PARAMETER;
              (WORD)p->lpvOutBuffer) = 0x0100;
              if (p->lpcbBytesReturned)
                     *(PDWORD)(p->lpcbBytesReturned) = 2;
              return 0;
       ...
}
```

When choosing a variable name, length does not necessarily improve clarity (`buf` is no less clear than `buffer`, `maxaddr` is no less clear than `MaximumAddress`). Wisely chosen variable names lead to self-documenting code. Long names often do more harm than good by obscuring the algorithm in which they are used. Convention can play a large part in choosing sensible variable names. Long before computers, mathematicians used conventions such as *i*, *j*, and *k* for indices. In C programming, the variable names `s` and `t` are often used for character pointers in library functions that operate on generic strings. Obeying conventions such as these helps keep your code concise and clear. However, variables that actually do refer to something (rather than serving as indices or scanning pointers) should have sensible names that represent their function. Rob Pike advocates longer variable names in cases there is little context around them, such as global variables and shorter names when there is a lot of context to fill in the meaning.

For one programming assignment, a number of students had variables of the form:

```
char buffer1[442];
char buffer2[442];
char buffer3[442];
char buffer4[442];
```

If you find yourself programming like this, alarms should go off and you should reconsider your algorithm and data structures. Do you really need exactly four buffers? Could there be a case when you may need more or less? Why is the size set to 442? Perhaps it will make more sense to have an array of pointers to strings? If you decide that you *really* do need those four buffers and they *really* need to be 442 bytes long, then you can improve the clarity of your code by defining 442 as a symbolic constant (with a

[1] Walter Oney, *Systems Programming for Windows 95*, ©1996 Microsoft Press, p. 270-271

comment explaining why) and picking more descriptive names for the four buffers. Similarly, if you find yourself declaring:

```
int a, b, c, d, ff, gg, ii;
char a_buf[], b_buf[];
```

then it's time to reconsider the clarity of you variable names and possibly comment some of the data definitions.

A large number of programmers favor mixed capitalization in variable names. For example, names like `keyboardInputBuffer`. I personally favor something more concise, such as `kbdbuf` for the clarity reasons cited above. If I'm forced to be verbose, I find underscores less offensive than embedded capitals. Consider the readability of English prose:

> fourscore_and_seven_years_ago_our_fathers_brought_forth_on_this_continent_a_new_
> nation_conceived_in_liberty_and_dedicated_to_the_proposition_that_all_men_are_created
> _equal.

versus:

> fourscoreAndSevenYearsAgoOurFathersBroughtForthOnThisContinentANewNationConce
> ivedInLibertyAndDedicatedToThePropositionThatAllMenAreCreatedEqual

Others disagree. Consistency is nice.

Modularize

Most student assignments that I've seen demonstrate a reluctance to decompose code. This turns even simpler assignments into difficult chores. Functional decomposition is the essence of algorithmic programming and a key to preserving clarity. Long gone are the days when a subroutine would be written only because the common code needs to be used in several places. Code should be structured so the underlying algorithms are clearly visible. Try to keep each function short and let it have a single purpose. If details need to be handled, let a support function take care of them.

Files are a useful vehicle for further modularization. If you have a number of related functions or your main program decomposes into several logical entities, don't hesitate to create files. Don't worry if the files are small. Perhaps the most useful file is a header file that is included by each of the source files. The header file will define all the data structures and types used by the program. It will also contain function prototypes for all of the functions. This file will serve as the most useful reference for the data structures and interfaces provided in your program.

Avoid cute typography

The visual layout of a program is crucial to its readability. Blocks (code between `{` and `}` should be indented. Two styles are acceptable:

```
if (whatever) {
      statement;
      statement;
}
else {
      statement;
}
```

and:

```
if (whatever) {
      statement;
      statement;
} else {
      statement;
}
```

Quite a few students, no doubt influenced by Pascal's `begin` – `end` blocks, write:

```
    if (whatever)
       {
          statement;
          statement;
       }
    else
       {
          statement;
       }
```

This is painful to the eyes of any C programmer and simply not acceptable. Even worse is an attempt to hide the syntax of the language with constructs like:

```
    #define IF if (
    #define THEN ){
    #define ELSE ;} else {
    #define ENDIF ;}

    IF whatever THEN
          statement;
          statement
    ELSE
          statement
    ENDIF
```

When indenting your code, I don't care whether your tab stop is three characters, four characters, eight, or twelve. The important point is I don't have to care and you should not assume that I will be looking at your code with the same editor and the same tab settings that you used in writing it. This is a problem with some editors (e.g. emacs) that insist on using a mix of spaces and tabs (for example, four spaces for the first indentation level and a tab character for the second indentation level). If your editor does this, figure out how to fix it, don't use this feature, or switch editors. If you use the tab character to indent your code, don't switch to using a space character to do so. The resultant code will not look right unless the reader has the same tab settings and font as the writer.

Some output filtering programs "pretty-print" your source to look like an Algol-68 report, rendering all the reserved keywords in boldface and comments in italics. Rob Pike points out that this accentuates irrelevant detail and is **as** sensible **as** putting all prepositions **in** English text **in** bold type.

Clean your code

If you decide to remove an `if` statement, make sure that you remove the extra indentation from the following statement. There is also no need to have code that is commented-out. It only serves to confuse the reader. While programming, and after programming, look at the aesthetics of your program source.

Be robust in accepting input; produce useful output

I cringe when I see an assignment that declares:

```
    char line[80];
```

The line is obviously written by someone who accepts input in the form of IBM punched cards whose physical limitation is eighty characters per card.

When accepting input in your program, be robust. A core dump due to bad input is *not* acceptable. Ever. Check your input for validity at all times. If it's not valid, recover gracefully. Produce a concise and descriptive error message pinpointing the problem. Print the error message to the standard error stream (*stderr*).

If your program accepts command-line arguments, make sure the input is foolproof and free-format. Don't require the user to know a particular order in entering arguments. Learn to use the *getopt* function. If the user makes a mistake, print an error message explaining the syntax of the program. Don't accept to many options: users will never learn them. Look at Unix's *ls* command as a bad example.

The DOS world introduced us to programs that look like:

```
C> fact
Bernice’s excellent factorial program
(c)1998 Bernice F. Lamingo

Please enter the number: 8

The answer to 8 factorial is 40320

C>
```

Pretty, eh? Compare this with:

```
$ fact 8
40320
$
```

The verbosity is gone. One line of input was produced instead of seven but no information was lost. The user presumably knew what the program was before running it, so there is no point in announcing the program. The user knew the input to the program, so there is no point in regurgitating that, Finally, the user really doesn’t care about the author. Far more importantly, the second form of output turns the program into a useful *tool* that can be used by other programs (shell scripts). The verbosity of the output as well as the inconvenient input of the former program disallows such use.

Keep your output short. If it makes sense, make the output such that it can be easily processed by other programs. This will increase the usefulness of your program.

Test your program … often

I’ve seen many attempts by students to sit down and write a multi-hundred line program and then try to debug it. The debugging efforts often lead to frustration and failure as there are numerous faults in the program and it is difficult to locate them. Debugging and testing should be done in conjunction with incremental development. It is useful to write your program so that it compiles at all times during creation. This often requires the writing of stub functions – placeholder functions that don’t really do the work of the function but pretend to do so. The point is to get your main algorithm compiled and working as quickly as possible, before the bells and whistles are added. After it works, you can start embellishing it by adding the features you need – one at a time and testing each time.

Test the program for robustness on accepting input. If you process a text file, feed it a huge object file. If the program expects to read in a number, give it text. Test your input buffer management by feeding your program huge lines of data. Perhaps the most common error is failing to detect an end-of-file at unexpected points. Finally, don’t use the *gets* function. It does not check for buffer overflow.

Optimize your code … later

No doubt you’ve studied or read about some nifty algorithms. You probably know of algorithms that can do your task in $n\log_2 n$ time instead of n^2 time. You may be eager to use them. It probably doesn’t make sense to do so 99% of the time. The fancier algorithms are generally more complicated and hence more bug-prone than the simpler ones. You’ll save yourself some grief by using the simpler algorithm for your first version of the program. Most everyone can program Shell’s sort without errors the first time but the a heap sort or quick sort will probably require a few iterations to get write. Using a brute-force search for a string is trivial, but implementing a Boyer-Moore search take a bit of effort. The second argument against the “better” algorithms is that there is often a constant setup time involved that is ignored in the order-complexity calculation. Complexity numbers have an implicit “*for large values of n*” assumption and your program will often not have the large volume of data for the efficiency of the better algorithm to be realized.

The bottom line is that you probably don’t know whether your algorithm is good enough or whether you should try using a better one. There’s one way to find out. Use the simple algorithm first. It will get your program working in less time. After that, profile your code. Learn to compile your program to include timing measurements and use a tool such as *prof* or *gprof* (or a line profiler if you have one available). The profiler will tell you where the program *really* is spending most of its time. If the routine that you were concerned about is taking 15% of the total execution time, then even if you make it *infinitely* faster, your program will run only 1.18 times faster. Alternatively, the profiler may confirm your suspicions and you may choose to improve the algorithm. Profile first. Optimize if necessary.

Most compilers are relatively smart. They reuse common subexpressions and keep frequently-accessed variables in machine registers. Writing dense code or otherwise trying to encourage a compiler to produce more efficient code is generally futile.

Pointers are not bad

Yet another edict of the Pascal school of programming was that pointers, especially pointers to functions, are to be avoided. While they can lead to problems if used incorrectly, they are both a useful tool and a notational device.

In the simplest case, they offer the programmer one less thing to worry about. A programmer examining code containing a pointer has only that value to be concerned with rather than dealing with a variable and an index into it. To a program familiar with the language

```
for (; *s; ++s) {
      if ( *s == '%') ...
```

is every bit as clear as

```
for (i=0; s[i]; ++i) {
      if ( s[i] == '%') ...
```

with one less variable to worry about. Pointers to functions are particularly useful in cases when the data determines which function gets called. Far too often, I see code that looks like:

```
if (strcmp(cmd, "run") == 0)
      run_cmd(args);
else if (strcmp(cmd, "step") == 0)
      step_cmd(args);
else if (strcmp(cmd, "exit") == 0)
      exit_cmd(args);
```

The list goes on. As more commands get added, the number of `if` statements grows. It would be far cleaner to leave the code alone and put the command to function mapping into a data structure. For instance, define:

```
struct ltab {
      char *cmd;
      void (*func)();
} map = {
      { "run", run_cmd },
      { "step", step_cmd },
      { "exit", exit_cmd },
      { 0, 0 }
};
```

and later in the code:

```
struct ltab *tab;

for (tab=map; tab->cmd; ++tab)      /* search through the table */
      if (strcmp(cmd, tab->cmd) == 0)
            break;
if (tab->cmd)
      tab->func(args);
else
      fprintf("illegal command: %s\n", cmd);
```

The code is somewhat shorter in cases when there are more than two possible commands. Far more importantly, a programmer can add new commands simply by adding an entry to a table and writing the appropriate processing function. All the related functions are now encouraged to have the same interface. This makes documentation easier. Finally, since the program and data are dissociated, the programmer may choose to replace the linear search with a better algorithm as the list gets longer – perhaps reading the list at start-up into a hash table.