Harry's Style Guide

Helpful Hints for Programming in the C Language

Harry H. Porter III

Computer Science Department
Portland State University

October 22, 1998

Abstract

This paper contains several suggestions about how to write "C" programs that are more readable and more likely to be bug-free. This is a "C" programming style guide, documenting a number of coding practices, conventions, and standards, along with justifications of why these recommendations produce better programs.

Programming style leads to program correctness.

The most important aspect of a program is whether or not it works correctly. A program either works correctly or it does not, and when it does not work, nothing else really matters. A program that does not work correctly is useless.

The key to creating good programs is to create programs that are readable, easy to understand, and attractive to the eye. If a program is easy to understand, then it becomes easier to see how it works. If it is easier to read, then you can look it over, find its bugs, and repair it quickly and efficiently. If the program is clear and understandable, then you can read over it to see whether or not it will work the way it is supposed to.

Readability is a question of style. Adopting a well-thought-out and consistent style of programming goes a long way toward creating correct and readable programs. Anybody can create a working program; the great coders produce programs that are also clean and attractive to the eye.

Programs must work correctly.

Programs must work correctly. Always remember that program correctness is absolute: either the program does what it is supposed to or it doesn't. (Or you are so hopelessly confused that you don't even know what it is supposed to do.) Although some theoretical research has shown that it is often difficult to define correctness precisely or that it is often difficult to prove correctness beyond a shadow of a doubt, these results don't negate the fact that a program is either correct or flawed. Either a program meets its specifications, performing correctly in all cases, or the program contains bugs and is flat-out wrong. Our goal is to always produce correct programs. Anything we can do to make that task easier is worth trying.

Of course, gigantic programs are created by teams of people—leaving no single programmer with responsibility—and become so complex that bugs become inescapable. They get so big that no one can understand what is going on, and problems remain buried under vast heaps of confusion. Companies employ statistical measures to determine how many bugs are likely to remain undiscovered in the code and they make market decisions about whether it is more profitable to release programs with bugs than to delay the release in order to fix the bugs. Large software companies often intentionally release programs with known bugs because they have judged the cost of correcting those bugs to be greater than the damage those bugs do to the software companies' profit statements.

This dismal state of affairs makes it easy to forget the importance of producing correct code, but the ability to write correct code is more important now that ever before. Programs with bugs can cause planes to fall out of the sky and passengers to die. Programs with bugs can cause medical monitoring systems to malfunction and patients to die. Programs with bugs can cause incorrectly designed buildings to collapse and people to be crushed to death. Programs must always be correct.

Test until you discover every single bug.

The only effective way to discover program bugs is to test your program. As you design and code a program, you must also test it. In general, designing good test data is an art which can only be learned through practice. A good rule of thumb is to test until you are not finding any more bugs, and then test a little longer to make sure there are really no more bugs.

You must learn to judge your own feelings about whether a program has been tested adequately. If there is the slightest doubt in the back of your mind, do not stop testing until that doubt is eliminated. Do not stop until you are 100% confident that your code will pass whatever tests you might possibly think of in the future. Do not stop testing until you are sure there are no more bugs!

Testing should be done in small pieces as the program is written.

Testing should be done while you are coding the program, not after you are done coding. You are most familiar with the program while coding it, so that is the time to test it. You should write a small section of code, then you should test that little piece thoroughly. Only after convincing yourself it works, can you move on to writing the next section of code. I call this "fine-grained testing."

When coding, it is very tempting to continue coding until the entire program is complete, instead of stopping to perform thorough testing after completing small pieces of the program. This is the devil speaking to you and you must fight his perfidious temptations with all your will!

You may think it is faster to finish coding a large chunk of program and then go back and do the testing all at once. You may even rationalize it by thinking that your program cannot be easily tested before it is all done and that testing will go much faster when performed on the completed program. Well, this is true; testing will be faster, but only because it will not be as thorough.

For example, you might think it would be much easier to test the "computation section" of the program after you finish writing the piece of code that displays the results. Unfortunately, when you finally get around to testing, you will be much more likely to focus your testing on the "display code" and neglect minor aspects of the "computation code." You must test small pieces as you write them. You must practice "fine-grained testing."

Test the boundary conditions.

When you begin testing, your first tests should test "cases in the middle." In other words, your first tests should confirm that the code works correctly for easy examples and simple inputs, to make sure that you have all the main parts in order and can produce outputs. All programmers do this level of testing to make sure the program works for the simplest data, but many programmers skip the next step of testing, which is to test extreme cases.

You must test the boundary conditions, the unusual data sets, and the weird inputs much more thoroughly than the middle cases, because this is where the subtle bugs will lie hidden. What if the user hits "enter" immediately without entering any data at all? What if the array size is zero or the list has length zero? What if the value of "x" is the largest integer representable?

Tests of this nature are best designed immediately after writing the code, because it is then that everything is fresh in your mind. The moment immediately after finishing a small block of code is the moment you understand it best; if you wait before performing your testing, you will slowly begin forgetting some subtleties of your code. This is why you must test immediately after writing the code without waiting.

Execute every line of code.

A standard rule of thumb is to make sure that you exercise every single line of code at least once. After all, if your test data fails to ever execute some single line of source code, then any bug in that line will not have been discovered. It may occasionally be tempting to avoid testing one or two lines of code because they look so simple and straightforward that nothing could possibly go wrong. Some code might seem self-evidently correct. Only amateur programmers ever fall for this logic for the simple reason that if you could always see bugs by looking at the code, there would never be any need to test the code. You must execute every line of code.

For "if" statements, this means that you must run a test in which the "then" statements are executed and you must also run a test in which the "else" statements are executed. (If there are no "else" statements, this means you must run tests in which the "if" condition is true sometimes and tests where it is false sometimes.) You must make sure your "if" statement is correct.

If Boolean conditions can be satisfied several ways—as in the following example—then you must exercise each possibility individually. In the following example, you must run cases where each of the three sub-conditions is true.

```
if (char == EOF ||
    i >= MAX_ID_LENGTH ||
    symbolTableFull ()) {
  printf ("error");
} else {
  ...
}
```

For "while" loops, you should run a test case in which the loop body is executed zero times (i.e., the condition is false the very first time the loop is encountered) and you should also run tests in which the "while" loop is executed several times.

With "for" loops like the following example, you should run test cases where "initValue" is less than "MAX," where "initValue" is equal to "MAX," and where "initValue" is greater than "MAX." The first test will check the case when the loop body is executed one or more times; the second and third tests will make sure the code works when the loop body is not executed at all.

```
for (i = initValue; i <= MAX; i++) {
  ...
}
```

It goes without saying that you must examine the output from your program to make sure it is correct. It is never sufficient to just run a program to make sure it doesn't "crash." You must make sure that the result is correct. You must also make sure that there is a connection between the portion of code that you are testing and the output that you are getting. It is not sufficient to simply execute every line of code at least once; you must also make sure that the output you get indicates that every line of code is working correctly.

Begin with random testing, if necessary.

The idea of "random testing" is to sample the space of possible inputs by selecting a variety of test inputs more-or-less at random. The goal is to adequately cover the entire set of possible inputs by choosing a sufficiently large subset of inputs. If the program works on randomly selected test inputs, the theory is that it will work on all inputs.

In debugging a program, many tests must be run and you must organize all this testing some way. One approach is to begin by simply making up tests at random.

Such randomly selected test data is often useful when you just want to see if any of your code works at all. So you can begin with "test cases in the middle" by randomly selecting the first few test inputs. As long as you are finding bugs this way, it is perfectly okay to proceed with randomly selected testing. This is a good way to get a fair percentage of your code working. Sometimes it is necessary to "get something working" before you can begin serious testing.

Don't use random testing for "fine-grained" debugging.

Random testing is never adequate by itself and I don't recommend spending too much time with it. Unfortunately, the set of possible inputs is generally infinite, and furthermore, the space of inputs is rarely uniform and continuous. All this makes it impossible to select a subset of inputs that accurately represents the set of all possible inputs. Instead, you are taking stabs in the dark and it is very easy to miss gaping holes in your code.

(However, it should be noted that random testing will often turn up subtle bugs that would go undiscovered otherwise, especially due to misconceptions about the problem domain itself. Therefore, really thorough testing also requires some random testing. A discussion of serious random testing is beyond the scope of this document.)

To begin thorough fine-grained testing, I recommend that you look at your code, line by line, as you create your test data. Design each test to exercise one (and only one) section of code. Run the test and then verify that the section of code being tested really works. Then move on to the next section of code and design a new test.

For example, suppose you are testing a subroutine you just wrote. First, you begin with a randomly chosen input that causes your program to produce the correct output. Start your fine-grained testing at the top of the code for the subroutine. Design a test to make sure the first line of the subroutine works correctly. Suppose the subroutine begins with an "if" statement; design one test that satisfies the condition and a second test that fail the conditions. Proceed through the body of the subroutine line by line.

For additional thoroughness, come back and test later.

A single episode of testing cannot be considered thorough. After your small piece of code is integrated into a larger program, it may begin to interact in strange, unforeseen ways. To be sure that it works in its context, you must come back later and test it again. You must ensure that it can be used the way it is intended to be used. For example, after completing a subroutine and doing the initial round of testing on it, suppose you proceed to work on another piece of the program that uses this subroutine. Later, you must test to see that the larger program can call the subroutine. You must ensure, if nothing else, that the correct subroutine is invoked.

Also, when you come back to test a piece of code later, you may see things that you missed before. Your eyes may strangely be opened to new possibilities for the same reason that we often understand our problems more clearly after taking a coffee break or spending a little time diverted with other mental distractions.

However, of all the testing, the initial "fine-grained testing" is the most important. You can often achieve a correct, working program with only fine-grained testing, but you can rarely achieve a correct program without it.

Who will read your programs?

There are several styles of programming appropriate for different programming projects, and you must know who your audience is when you write a program. The biggest question is whether other people will read your code, or whether you can be sure that you alone will read it.

If you are programming for a company or other organization, they should have standards and recommendations about programming style. Follow their recommendations, even if they seem awkward, silly, or outright bad. The primary motivation for company-wide standards is to make the code written by different individuals look as similar as possible. Once you are familiar with the group style, it will be easier to read code written by other programmers. If their code looks like yours, it will be much easier to accept it and to focus on the issue at hand; namely its correctness. If everyone's code looks different, you can easily find yourself favoring your own code and despising their way of doing things.

When there are no stylistic conventions imposed on you, create a style of your own. If you are programming in an environment without restrictions, find a consistent style and begin using it. As time goes by, your style may evolve and improve, but seek to maintain a consistent style.

Code is most often read by its original author.

It is far easier for a program to be modified by the person who wrote it, than by another programmer. Thus, whenever a program must be corrected or altered, it is logical for the users to seek out the original author to fix it. Nevertheless, it always difficult to go back to an old program—even if you wrote it yourself—and modify it.

When you write comments, it is possible that you will be the only one to ever read them again. This is no excuse to alter you programming style or relax your standards! In fact, it is a good reason to keep them high. If your employer confronts you with code you wrote two years ago, asking you to correct or change it, and you are faced with poorly-written code that is difficult to understand, the loser will be you. If even you can't maintain your own code, then it is without doubt unmaintainable and consequently worthless. On the contrary, if your programming style leads to a clear program, it

will be all the more possible for you to fix it quickly or even to delegate its maintenance to someone lower on the corporate totem-pole, leaving yourself free to take a long lunches, play golf, or write new code.

Don't ignore things; don't procrastinate.

As you are coding or testing, you will notice things about your program that you will need to go back to take care of later. For example, as you are coding you might think "Don't forget to test the case where x=0." Or perhaps while you are testing you might think, "Whoops, I messed up this section. I did a similar thing in another subroutine; I wonder if I messed that up, too."

It is very important not to forget these thoughts. Keep a piece of paper next to your computer and make a list of things you need to go back to. Write down tests you really ought to try during debugging. Write down things about your code you intend to go back and check on. Often, when you make one change to your code, it will require several other related changes. Each change must be tested. Right at the beginning, make a list of all the things that will need to be changed. Then, as you go through your list, you can concentrate on each thing more fully, without worrying that you will forget one of the items on the list.

Occasionally you will catch only a hint of a problem with your code. *Never ignore a problem!* You must learn to love problems to be a good programmer. If you have the slightest scent of a bug, you must chase after it like a hound after the fox. Intuitions and suspicions can be forgotten so easily. Write it down on your list. If you have the slightest whiff of a bug, *check it out fully!* This may be your only chance to find that bug, and you don't want to miss it.

Never say to yourself, "I can ignore that because I'll take care of it later." Do not ever leave a bug in your program. Do not ever put off fixing your code. Perhaps you say, "This is in a section of code I am not working on right now." Perhaps you think that it is such an obvious bug that you will be forced to deal with it anyway when you are debugging later. No matter what kind of lies you may be tempted to tell yourself, never let a problem in your code go untended. At least write it down on your list so that you will be sure to attend to it later.

A particularly nasty form of procrastination occurs when you say, "I'll write the code this way since it would take too long to write it more carefully. I just want to get something working first and I'll come back and improve this code later." It is much wiser to do it right the first time, and in the long run, it will always save time. If you do it right the first time, you will sleep much easier at night. If you skimp and cheat and write code that you know could be better, you will loathe yourself and have nightmares all night! Don't write code that you know is not the best it could be!

What is a coding style?

Of course, a compiler will scan your program so it must conform to the formal definition of the language and the compiler will check this mechanically, but all programming languages allow a great deal of flexibility in exactly how the program is written. For example, how will you indent your "if" statements? How much will you comment and what will the comments look like? It is these issues that separate well-written correct programs from poorly-written but still correct programs.

In the remainder of this document, I list several rules which, if followed, will make you a better coder.

Rule 1: Use a "block comment" for every function or procedure.

Each subroutine should be preceded by a "block comment." (By "subroutine" I mean procedures and functions.) A block comment describes the nature of the subroutine: its name, its arguments and its returned value. Block comments occur only in front of each subroutine and not within subroutines.

```
/* match (t)
**
** This routine ensures that the next token is a t.  If so, it scans
** it and advances to the next token; if not it prints an error message
** and halts execution.
*/
void match (token t) {
  if (nextToken == t) {
    advance ();
  } else {
    syntaxError (t, nextToken);
  }
}
```

As shown in the example, a row of double stars along the left edge serves to visually set the block comment apart from everything else. When approaching a large program for the first time, the reader will look only at the block comments, in an attempt to get an overview of the program. These comments may often be used when skimming a program in an attempt to find the section of relevant code to examine more carefully. The row of double stars helps when scanning thousands of lines of code.

Insert exactly three blank lines before the block comment to set it off from the preceding routine. Again, this is just a visual aid to the eye in finding routine boundaries.

Some people suggest using a long line full of asterisks to set off the block comment. Some people go to the trouble of surrounding the block comment in boxes made out of asterisks. I feel that level of art is overkill; it is too difficult to maintain and can be visually distracting. Notice that there is a single blank between the asterisks and the text of the comment and that the last line of the comment serves as a sort of blank line between the text of the comment and the first line of the subroutine.

The first line of the block comment gives the name of the subroutine and its parameters. The parameters are given by name. Parameter types are not given here since they are listed in the first line of the subroutine and repeating them in the block comment becomes a point of possible inconsistency. Within the text of the comment, the parameters may be referred to by the names used in the first line of the block comment.

The text of the comment should be written in complete, correct English sentences. Complete sentences are much easier to read and understand than fragments. Incomplete sentences often leave room for misinterpretation. The comment should be of the form "This routine does such-and-such." In other words, the text takes the voice of telling you (the reader) what the code below does. The comment should not be of the form "Do such-and-such."

The block comment should mention each argument (i.e., each parameter) and explain its relevance to the task being performed. If the routine returns a value, this should be described in the comment. If the subroutine returns a Boolean, the comment should be clear about what TRUE means and what FALSE means by saying something like, "The routine returns true if such-and-such holds, and false otherwise."

A block comment should describe what the subroutine does, not how it does it. It is written mostly for the benefit of whoever is writing the code that will call the routine—not the person coding the subroutine itself—so the first goal is to clearly specify what the routine does so that it may be used by others. The details about the algorithm used within the subroutine are generally not important to the caller so they are not included.

It is assumed that if the reader wants to know how the subroutine works, he will read the code itself. There is no value in repeating what the code does in the block comment, since the code is right there anyway. At worst, the comment can be incorrect or misleading and can cause the reader to misinterpret the code. However, if the algorithm is sufficiently complex, it may be a good idea to comment on it. This should be done in a separate paragraph in the block comment.

Rule 2: Do not put opening braces on a line by themselves.

Note that the opening brace for the subroutine occurs on the same line as the subroutine name and parameter list. It follows the closing parenthesis and is separated by exactly one blank. The opening brace for the "if" statement also follows a closing parenthesis and—again—a single space is used to set it off. The rule eliminates a lot of blank lines; these blank lines waste space on the screen, they are visually distracting, and they lower the impact of truly blank lines inserted for clarity.

Rule 3: Always indent in increments of 2 spaces.

Indent consistently. If you are indenting by increments of two spaces, don't indent some lines by other amounts.

```
if (......) {         /* right */
  if (......) {
    x = ...;

if (......) {         /* wrong */
  if (......) {
        x = ...;
```

Why two spaces and not another amount? First, one space is not enough, since our eyes are so used to seeing single spaces. The visual effect is not strong enough. With one space, you are apt to miss the indentation altogether.

The tab key usually gives you eight spaces, but this is way too much indentation. Complex programs quickly run off the right end of the page and wrap-around is far more damaging to readability.

```
if (......) {
        while (......) {
                if (......) {
                        for (......) {
                                if (.......) {   /* not enough space i
s left here */
```

Four spaces is acceptable, but two spaces are adequate and easier to type. Using two spaces also has the effect of keeping the code closer together. This reduces eye movement and makes the code easier to read.

```
if (......) {
```

```
while (......) {
  if (......) {
    for (......) {
      if (.......) {    /* This is easier to read. */
```

Rule 4: Limit your lines to 80 characters in length.

The original IBM punch cards could each store a single line of exactly 80 characters. Today, different window sizes can accommodate various line lengths. If your code is displayed in a window that is too narrow, long lines will be wrapped onto a new line. This makes code difficult to read.

To prevent unwanted wrap-around, I recommend limiting lines to 80 characters in length, since that is often the default size for windows displaying text. I don't recommend counting characters on each line—that is too bothersome—but I suggest breaking apart lines that are obviously longer than 80 characters.

When breaking a long line into two, indent the second line a whole bunch to make it obvious that it is a continuation line.

```
/* wrong */
x = a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q +
  r + s;

/* right */
x = a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q +
                                                                r + s;
```

Rule 5: Indent continued lines to reveal their relationships.

There are a couple of cases where a list of several items on a single line would be clearer if the items in the list were lined up and placed of consecutive lines. Don't hesitate to line things up when it helps to show their structure. Here are a couple of examples:

```
foo (x,
     y + 23,
     anotherRoutine (a + 1,
                     b + 2,
                     c + 3),
     sqrt (z * x),
     w - (u * v));

if ((x == ...) ||
    (y == ...) ||
    (z == ...)) {
  printf (...);
}
```

Rule 6: Never use the tab character.

Some software (still!) has problems with the tab character. For example, the amount of space implied by a tab is not fully agreed upon and—what is worse!—can be changed by different users. Always use spaces. Your program will then look exactly the way you want. (You may assume that

your program will be displayed with a fixed-sized character font, but it will look okay even if it is not.)

Typing all those spaces may seem like extra work but I find that I can hit two spaces about as fast as I can hit the tab key. If you are concerned about the additional disk space used by all those spaces instead of the more-efficient tab characters, take a deep breath and try to have perspective: with today's disk capacities this is not a reasonable argument. Spend your energy learning to use file compression software and save far more bytes.

Rule 7: Always uses braces in "if," "for" and "while" loops.

Use braces, even if there is only one statement in the body and the braces are not strictly necessary. The primary motivation is consistency. With only a single statement, there are two options: you can either include braces or not. Since you will be using braces in other cases (i.e., when the body has two or more statements), you might as well use braces always.

```
while (...)      /* wrong */
  x = ...;

while (...) {    /* right */
  x = ...;
}
```

One general principle of finding a good coding style is: when there are two ways to code the same thing, determine which is the more general way and then always code in that style. Here, the goal is to have all "while" loops look the same, regardless of how many statements the body contains.

The closing brace serves to mark the end-point of the loop or "if" statement. This point is a very important place and needs some symbol to mark it. Without the braces, as in the following example, the location of the end of the loop is not as clear.

```
while (... &&    /* wrong */
       ...)
  if (... &&
      ...)
    if (... &&
        ... &&
        ...)
      z = ...;

while (... &&    /* right */
       ...) {
  if (... &&
      ...) {
    if (... &&
        ... &&
        ...) {
      z = ...;
    }
  }
}
```

Another reason for using braces is that it makes it easier to insert an additional statement into a loop or "if" statement without making a mistake. If braces are always used, you can simply insert a statement; you don't need to also remember to add a set of braces. If braces are used sometimes but

not others, then you may occasionally forget to add braces. Then you might end up with code like the following, which is misleading and probably incorrect. In any case, you find yourself thinking about braces instead of thinking about the code itself.

```
if (x * y == z)
  a = (b - c) / d;
  e = (f + g) * (h - i);
if (...)
```

Rule 8: Line up all closing braces.

Each closing brace should be on a line by itself (with the exception of the brace in front of the "else" keyword, as discussed below). The closing brace should be in the same column as the first character of the line containing the matching opening brace. Every statement within the loop or "if" statement should be indented by exactly two spaces.

```
while (x <= MAX_LIMIT) {
  x = ...;
  if (y == 0) {
    x = ...;
    y = ...;
  }
  z = ...;
}
```

The closing brace for a subroutine will be the only closing brace appearing in the first column, since only the subroutine header line will start in column one.

Rule 9: Put "else" on a line by itself.

Format all "if" statements as suggested by this example:

```
if (nextToken == t) {
  advance ();
} else {
  syntaxError (t, nextToken);
}
```

All "then" and "else" clauses should include braces, even if there is only one statement. Every "else" should appear on a line by itself and will be preceded by a closing brace and a space and followed by a space and an opening brace, as shown above. Line up the parts of the "if-else" as shown above.

Put the statements in the "then" and "else" clauses on separate lines; do not put the statements on the same line as the "if" or "else."

```
if (nextToken == t) advance ();     /* wrong */
  else syntaxError (t, nextToken);  /* wrong */
```

Rule 10: Format "if-elseif" clauses specially.

Sometimes a sequence of nested "if" statements would be more properly programmed using an "if-elseif" construct, but unfortunately C does not have such a construct. This happens when there are a series of tests to be performed sequentially, and exactly one will be satisfied.

Here is an example:

```
if (c == EOF) {                 /* wrong */
  return (LX_EOF);
} else {
  if (isalpha (c)) {
    return (LX_ALPHA);
  } else {
    if (isdigit (c)) {
      return (LX_DIGIT);
    } else {
      printError ("...");
    }
  }
}
```

When code like this occurs, it is better to format it as follows. This is how to code an "if-elseif" statement in C.

```
if (c == EOF) {                 /* right */
  return (LX_EOF);
} else if (isalpha (c)) {
  return (LX_ALPHA);
} else if (isdigit (c)) {
  return (LX_DIGIT);
} else {
  printError ("...");
}
```

Rule 11: Surround all operators by a space.

Operators are very important and using a single space on either side sets them off visually. This rule applies to the assignment operator (=) as well.

```
x = ((a + b) * c) < 100) && (d > 1000);  /* right */
x=((a+b)*c)<100)&&(d>1000);              /* wrong */
```

Rule 12: Do not use the assignment operator in expressions.

In C, the programmer may use the assignment operator (=) in embedded expressions. The justifications for allowing this are (1) it makes the language simpler and (2) the resulting code may be more efficient.

Unfortunately, mis-typing an equality operator (==) is a common mistake and, once made, is often difficult to catch. Even when coded correctly, the = operator can be easily mistaken for == when reading the code, making it easy to overlook an assignment and therefore difficult to understand the code correctly.

Therefore, I avoid using the assignment operator everywhere except in an assignment statement directly.

```
if (done = (atEOF || problems)) {          /* wrong */
  ...
}

done = atEOF || problems;                  /* right */
if (done) {
  ...
}
```

After getting used to this convention, every = operator in an expression begins to stand out like a red flag, which helps to spot errors more easily. As to the question of target code efficiency, an optimizing compiler will probably generate the same code regardless of how the source code was written, so this is not a reasonable concern.

Rule 13: Use a space between a function name and the following parenthesis.

In a statement or expression that invokes a subroutine, arguments will be surrounded by parentheses. Insert a space before the opening parenthesis. Apply this rule even when the argument list is empty.

```
x = myRoutine (a, b, c);
otherRoutine (d, e, f);
compute ();
```

Rule 14: Insert a single space after every comma.

This is the convention in printed English text, and it is for a reason. It makes lists easier to read if the same rule is also followed in programs.

```
routineOne (a, b, c);  /* right */
routineTwo (a,b,c);    /* tempting, but wrong */
```

Rule 15: Avoid inserting blank lines.

Blank lines add to the program length without adding much clarity. However, there are several places where blank lines should be used:

(1) Insert 3 blank lines before the block comment for each subroutine.
(2) Insert a blank line between the variable declarations and the code within a subroutine.
(3) If the code of a long procedure is logically divided into several sections, add a blank line between them.

Rule 16: Declare variables at the beginning of the subroutine.

Declare all variables at the beginning of a subroutine. Do not include declarations in the middle of the subroutine, even for temporary variables. A “temporary variable” is one that is used only within the span of a couple of lines; it does not hold a value over long sequences of execution.

The reason for putting all declarations at the beginning of a subroutine is that when reading code a person will often need to refer back to the declaration to see what the type of the variable is. Placing

declarations in the body of the code tends to hide them. This is true even of temporary variables. Put all declarations at the beginning of the subroutine.

For temporary variables, use short, one character names so they can be easily recognized as being less important than other variables. Names like "t," "i," and "x" are recommended.

Insert a single blank line after the variable declarations and the actual code.

```
foo (...) {
  int t, totalCount;

  if (...) {
    t = ...;                        /* Don't use "int t = ...;" here. */
    totalCount = prevCount + t;
  }
  ...
}
```

Rule 17: Write block comments before coding.

When you begin to write a subroutine, start by typing in the block comment first. If you are unable to write the block comment, you are not yet ready to begin coding! You must have a very clear idea of what a subroutine will do before you code it. If you can not articulate it in words, then something is very wrong. Finish writing the block comment before you code one line of the subroutine.

This will free you to think only about the code while you are writing it. While coding, if you discover a problem in the block comment, change the comment before going on. If you focus on the comments first, the code will be easy to write and more likely to be correct. If you focus on the code first, it will be confused and the comments will probably not get completed.

Rule 18: Keep the comments up-to-date.

Never change a program without reviewing the comments at the same time. Read over the comments and make sure that they reflect the current state of the program. A program without accurate comments is a sure sign of disaster.

I consider any program with incorrect or out-of-date comments as being incorrect itself, even if it produces the correct output.

Never put off changing the comments by thinking that you will go back to revise them after the coding is done! Instead, you will be busy testing as soon as the coding is done. Every program modification should start with a modification to the comments. If you are unwilling to modify the comments, you should not modify the code itself. The comments are a part of the program; they must be changed together.

Rule 19: Document each program file.

Here is an example of how the beginning of a file should look:

```
/* lexer.c -- PCAT Lexical Analyzer
**
** Harry Porter, 2/1/95.
```

```
** Modified: John Doe, 3/15/97.
** Modified: Jane Smith, 10/11/98.
**
** A lexical analyzer the PCAT programming language.  See the
** document titled "The PCAT Reference Manual" for details about
** the lexical nature of PCAT.
**
** The primary routine provided here is getToken(), which returns
** a token each time it is called, however other routines may be
** invoked from external code.
*/

#include <stdio.h>
#include "lexer.h"

char tokenBuffer [MAX_ID_LENGTH]; /* Buffer for storing current token */
int bufferIndex;                  /* Next unused tokenBuffer position */

/* Routines in this file: */
token getToken ();
token checkReserved ();
void lexicalError (char * message);
void initBuffer();
```

File Name Each file should start with a block comment describing what is in the file. The first line should be the "normal" file name. Sometimes files get renamed, so this may not be the actual file name, but it serves to identify the file when it is viewed. If the program has a formal name or title, this would also go on line one.

Author Information The second line (after a blank line) should include the original author's name and the date the file was created. Subsequent lines should list the names of all people who have modified the file and the dates. Do not ever modify someone else's program without putting your name and the date into the file. Once you modify the program, the responsibility for its correctness is yours. Leaving someone else's name as the last person who modified the file is considered fraudulent and dastardly.

Program Description The block comment should then describe the program in general terms. Often this will contain a reference to an external document, as in this example, but it is also okay to include the "user's manual" directly in the file for small or medium-sized systems. After all, this is probably the safest way to make sure that the user's manual doesn't get separated or lost and to make sure that it stays up to date. Putting the user's manual in the program's block comment also nicely handles the situation where there are several versions of the program floating around. Each version performs differently and needs its own user's manual tied to it.

In many programs, it is reasonable to include the user's manual in "print" statements within the program itself. This allows the user to have the user's manual displayed while he is using the program. (Perhaps the program has a "help" command that causes the information to be displayed.) In this case, the block comment at the beginning of the program can refer the reader to the section of the file containing those print statements.

If the file contains pieces of a larger program (as in this example), the block comment should also include some information on the interface between the code in this file and code in other files.

Program Preamble and Blank Lines Following the block comment will be often be (1) "#include" directives, (2) global data definitions, and (3) procedure headers. This example has all

three sections; each should be separated by a single blank line. It is okay to insert additional blank lines to separate large data declarations and the like for clarity, but always use a single line at a time. Since the first routine will be preceded by three blank lines, it will be clear where the "preamble" ends and the subroutine code begins.

Local Routines It is a great idea to include subroutine headers (i.e., declarations which list parameters but not code) for all routines occurring in the file (as was done above) at the beginning of a file. There should be exactly one line for each subroutine and they should occur in the same order as the subroutines appear in the body of the file. This section then becomes a handy index to the rest of the file. There should be no attempt here to document the subroutines themselves, since that will be done in their block comments.

Standard C practice is to put routine headers in ".h" files, which are then "#included." This is also acceptable and both approaches should be used together. Generally, the ".h" file should list only the routines that form the external interface. In the file itself, you should list all routines, including those that are only called from within the file.

Global Data Global variable data should be collected and placed at the beginning of the file. The understanding of the global data structures is usually critical to understanding the entire program; the global data structures should be documented as necessary. Often the author of the program will understand the global data structures so thoroughly that he will fail to document them adequately. However, a lack of understanding of the global data structures makes it almost impossible for anybody else to understand the rest of the program. Always be careful to provide accurate and complete descriptions of the global data.

Rule 20: Spell variable names using English words.

I recommend creating identifiers that consist of full English words. For example:

```
tokenBuffer
parseReadStatement
```

The first character of each word is capitalized and all other letters are lowercase. Begin the variable name with a lowercase and avoid the underscore character.

I recommend using only valid English words and spelling them correctly since it makes it much easier to remember how an identifier was spelled. I recommend avoiding the underscore character in identifiers because it is too easily confused with a minus sign and because it is too visually distracting.

These rules apply to variable names and subroutine names. Variable names should be given names that are nouns, like "tokenBuffer" and "whiteSpaceCharacterSet." Subroutines should be given names that are actions or commands, like "getToken" and "printError." Routines that return true or false should be given names that ask a yes-no question, such that a yes answer corresponds to a true result. For example, the routine "isLetter" might used to test a character and return true if it is a letter character. Boolean-valued variables can be given names that form a statement of fact, like "gotLetter" and "atEnd." Variables that are only used over the span of three or four statements can be given single letter names to indicate their temporary nature.

Here is an example, using these conventions:

```
if (atEnd) {                    /* right */
  printError ("...");
```

```
} else {
  t = getToken ();
  c = tokenBuffer[1];
  if (isLetter (c)) {
    gotLetter = TRUE;
  }
}
```

Below is the same code; the only difference is the variable names. Once you get used to it, the first version seems more readable than the second.

```
if (end_pos) {                    /* wrong */
  error_displ ("...");
} else {
  t = next_tok ();
  char = tkn_buf[1];
  if (letter (char)) {
    let = TRUE;
  }
}
```

Admittedly, using full words for variable names makes the names longer. However, the increase in readability (not reduction of typing) is what programming style is all about. In larger programs, longer variable names can make the code significantly easier to read.

If you wish to reduce typing even more, consider the following version of the same code, which is how some programmers actually code C. Code written like this is ciphertext. It is difficult to read, even by people well-versed in C. There is nothing to be gained but obscurity and programming bugs with code that looks like this.

```
if (ep)
        erdisp("...");
else {
        t=nxt();
        c=tbuf[1];
        if (let(c)) l=TRUE;}
```

Rule 21: Capitalize all constant names

Constant names (that is, identifiers introduced in "#define" statements) should be fully capitalized. Use complete English words and use the underscore character to separate different words. For example:

```
#define TRUE 1
#define FALSE 0
#define MAX_BUFFER_SIZE 100
#define MAX_NUMBER_OF_SYMBOLS 200
```

Rule 22: Don't use all the features you can.

Many people like to use newly acquired knowledge. In general, this is good since it helps you become familiar with the new material. But if your goal is to write programs that are easy to read and understand, then you make things more difficult whenever you incorporate new features. If you have just learned some new feature of a programming language, using that new feature will make it impossible for a person to understand your program unless he has at least as much knowledge as

you. And if he has more knowledge than you, it is more likely you will be reading and maintaining his programs!)

It is preferable to adopt a reasonable subset of C and just use features in that subset. For one thing, you become more familiar with the pieces you are using. When you use a new feature that you have just discovered, it is more likely you will make an error. For another thing, when another person is reading your code, he will need to adjust to your style of coding. If your code is unnecessarily complex, it will take longer before he becomes comfortable with your style.

Rule 23: Avoid using other software.

These comments also apply to using software packages when writing programs. Every time you use some strange piece of software, you cut the pool of people who can understand and maintain your code. Suddenly your code is dependent on lots of other stuff working correctly. If you use some strange preprocessor that you picked up off the net, for example, it may be impossible for other people to understand your code at all. They will need to train on this new piece of software (which they don't particularly want to do) before they are competent to maintain your software.

Even worse though, is what happens when new computers come out or new versions are released. Maybe the company that wrote that preprocessor is out of business... now your software is dead, too! Or maybe they came out with a new version of the preprocessor but decided to clean up some of the features (or bugs) in their old version. Unfortunately, their change to their software might require you to change your software. Don't you have better things to do?

The moral is to try to avoid relying on other people's software or other software packages as much as reasonably possible. The more software your code relies on, the sooner it will fail from changes external to itself. The more software your code relies on, the more time you will have to spend maintaining your old programs just to keep them working.

Obviously this rule must be applied with wisdom and foresight since using other people's software may dramatically reduce your programming effort. Although it may make your program more difficult to maintain, it may be worth it since it may reduce your coding timing dramatically.

Summary

The goal of good coding is to produce software that works correctly and that can be repaired or modified easily. With larger programs, the fundamental key to this is to create programs that are easy to read, both by you and others. Following a set of stylistic conventions, like the rules described here, when you code, goes a long way toward achieving the goal of creating good, readable software.

When coding, there are many decisions to be made. Some decisions are large and complex, but many decisions are trivial and mundane, like when to insert an extra space character. The countless unimportant decisions of formatting your code each distract you a tiny bit from the main task. Slowly they tire your mind, opening the way to mental fatigue, which in turn will detract from the quality of your code and impinge on the logical correctness of your programming. By adopting these stylistic conventions, you are freeing your mind from the drudge work of coding and allowing it to concentrate fully on the bigger picture, namely the much harder tasks of creating high quality, complex software systems.