Chapter 3. Style

There is no programming language, no matter how structured, that will prevent programmers from writing bad programs.

—L. Flon

This chapter discusses how to use good programming style to create a simple, easy-to-read program. Discussing style before we know how to program might seem backward, but style is the most important part of programming. Style is what separates the gems from the junk. It is what separates the programming artist from the butcher. You must learn good programming style first, before typing in your first line of code, so that everything you write will be of the highest quality.

Contrary to popular belief, programmers do not spend most of their time writing programs. Far more time is spent maintaining, upgrading, and debugging existing code than is ever spent on creating new works. According to Datamation, the amount of time spent on maintenance is skyrocketing. From 1980 to 1990, the average number of lines in a typical application went from 23,000 to 1,200,000. The average system age went from 4.75 to 9.4 years.

What's worse, 74% of the managers surveyed at the 1990 Annual Meeting and Conference of the Software Maintenance Association reported that they "have systems in their department, that have to be maintained by specific individuals because no one else understands them."

Most software is built on existing software. I recently completed the code for 12 new programs. Only one of these was created from scratch; the other 11 are adaptations of existing programs.

Some programmers believe that the purpose of a program is only to present the computer with a compact set of instructions. This concept is not true. Programs written only for the machine have two problems:

- They are difficult to correct because sometimes even the author does not understand them.
- Modifications and upgrades are difficult to make because the maintenance programmer must spend a considerable amount of time figuring out what the program does from its code. Ideally, a program serves two purposes:

first, it presents the computer with a set of instructions, and second, it provides the programmer with a clear, easy-to-read description of what the program does.

Example 2-1 contains a glaring error that many programmers still make, and that causes more trouble than any other problem. *The program contains no comments.*

A working but uncommented program is a time bomb waiting to explode. Sooner or later, someone will have to fix a bug in the program, modify it, or upgrade it, and the lack of comments will make the job much more difficult. A well-commented, simple program is a work of art. Learning how to comment is as important as learning how to code properly.

Comments in C start with a slash asterisk (/*) and end with an asterisk slash (*/). Example 3-1 is an improved version of Example 2-1.

Example 3-1. *hello2/hello2.c*

```
[File: hello2/hello2.c]
/********************************************************
 * hello -- program to print out "Hello World".        *
 *      Not an especially earth-shattering program.    *
 *                                                     *
 * Author:  Steve Oualline.                            *
 *                                                     *
 * Purpose:  Demonstration of a simple program.        *
 *                                                     *
 * Usage:                                              *
 *      Runs the program and the message appears.      *
 ********************************************************/
#include <stdio.h>

int main()
{
    /* Tell the world hello */
    printf("Hello World\n");
    return (0);
}
```

In this example, we put the beginning comments in a box of asterisks (*) called a *comment box*. This formatting is done to emphasize the more important comments, much as we use bold characters for the headings in this book. Less important comments are not boxed. For example:

```
/* Tell the world hello */
printf("Hello World\n");
```

In order to write a program, you must have a clear idea of what you are going to do. One of the best ways to organize your thoughts is to write them down in a language that is clear andeasy to understand. After the process has been clearly stated, it can be translated into a computer program.

Understanding what you are doing is the most important part of programming. I once wrote two pages of comments describing a complex graphics algorithm. The comments were revised twice before I even started coding. The actual instructions took only half a page. Because I had organized my thoughts well (and was lucky), the program worked the first time.

Your program should read like an essay. It should be as clear and easy to understand as possible. Good programming style comes from experience and practice. The style described in the following pages is the result of many years of programming experience. It can be used as a starting point for developing your own style. These are not rules, only suggestions. Only one rule exists: make your program as *clear*, *concise*, and *simple* as possible.

Poor Man's Typesetting

In typesetting, you can use letter size, **bold**, and *italic* to make different parts of your text stand out. In programming, you are limited to a single, mono-spaced font. However, people have come up with ingenious ways to get around the limitations of the typeface.

Some of the various commenting tricks are:

```
/********************************************************
 ********************************************************
 ******** WARNING:  This is an example of a      *******
 ********   warning message that grabs the      *******
 ********   attention of the programmer.        *******
 ********************************************************
 ********************************************************/

/*------------> Another, less important warning <--------*/

/*>>>>>>>>>>>>  Major section header  <<<<<<<<<<<<<<<< */

/********************************************************
 * We use boxed comments in this book to denote the     *
 * beginning of a section or program.                  *
 *****************************************************/

/*---------------------------------------------------*\
 * This is another way of drawing boxes.             *
\*---------------------------------------------------*/

/*
 * This is the beginning of a section.
 * ^^^^ ^^ ^^^ ^^^^^^^^^ ^^ ^ ^^^^^^^
 *
 * In the paragraph that follows, we explain what
 * the section does and how it works.
 */

/*
 * A medium-level comment explaining the next
 * dozen (or so) lines of code.  Even though we don't have
 * the bold typeface, we can **emphasize** words.
 */

/* A simple comment explaining the next line */
```

At the beginning of the program is a comment block that contains information about the program. Boxing the comments makes them stand out. The list that follows contains some of the sections that should be included at the beginning of your program. Not all programs will need all sections, so use only those that apply:

- **Heading**. The first comment should contain the name of the program. Also include a short description of what the program does. You may have the most amazing program, one that slices, dices, and solves all the world's problems, but the program is useless if no one knows what it is.
- **Author**. You've gone to a lot of trouble to create this program. Take credit for it. Also, anyone who has to modify the program can come to you for information and help.
- **Purpose**. Why did you write this program? What does it do?
- **Usage**. In this section, give a short explanation of how to run the program. In an ideal world, every program would come with a set of documents describing how to use it. The world is not ideal. Oualline's law of documentation states: 90% of the time the documentation is lost. Of the remaining 10%, 9% of the time the revision of the documentation is different from the revision of the program and therefore completely useless. The 1% of the time you actually have documentation and the correct revision of the documentation, the information will be written in Chinese[1] .

[1] My wife comes from Hong Kong and has a talking electronic translator. It is a very expensive and complex device, and comes with a 150-page manual, written entirely in Chinese.

To avoid Oualline's law of documentation, put the documentation in the program.

- **References**. Creative copying is a legitimate form of programming (if you don't break the copyright laws in the process). In the real world, you needn't worry about how you get a working program, as long as you get it, but give credit where credit is due. In this section, you should reference the original author of any work you copied.
- **File formats**. List the files that your program reads or writes and a short description of their formats.
- **Restrictions**. List any limits or restrictions that apply to the program, such as "The data file must be correctly formatted" or "The program does not check for input errors."
- **Revision history**. This section contains a list indicating who modified the program, and when and what changes were made. Many computers have a source control system (RCS and SCCS on UNIX; PCVS and MKS-RCS on MS-DOS/Windows) that will keep track of this information for you.
- **Error handling**. If the program detects an error, describe what the program does with it.
- **Notes**. Include special comments or other information that has not already been covered.

The format of your beginning comments will depend on what is needed for the environment in which you are programming. For example, if you are a student, the instructor may ask you to include in the program heading the assignment number, your name and student identification number, and other information. In industry, a project number or part number might be included.

Comments should explain everything the programmer needs to know about the program, but no more. You can overcomment a program. (This case is rare, but it does occur.) When deciding on the format for your heading comments, make sure there is a reason for everything you include.

Inserting Comments—The Easy Way

If you are using the UNIX editor `vi`, put the following in your *.exrc* file to make constructing boxes easier:

```
:abbr #b
/************************************************
```

```
:abbr #e
************************************************/
```

These two lines define `vi` abbreviations #b and #e, so that typing:

```
#b
```

at the beginning of a block will cause the string:

```
/***********************************************
```

to appear (for beginning a comment box). Typing:

```
#e
```

will end a box. The number of stars was carefully selected so that the end of the box is aligned on a tab stop.

Similar macros or related tools are available in most other editors. For instance, GNU Emacs lets you achieve the a similar effect by putting the following LISP code in a file named *.emacs* in your home directory:

```
(defun c-begin-comment-box ()

  "Insert the beginning of a comment, followed by a string
of asterisks."

  (interactive)

  (insert
"/***********************************************\n")

  )

(defun c-end-comment-box ()

  "Insert a string of asterisks, followed by the end of
a comment."

  (interactive)

  (insert
"***********************************************/\n")
```

```
  )

(add-hook 'c-mode-hook

  '(lambda ()

     (define-key c-mode-map "\C-cb"
'c-begin-comment-box)

     (define-key c-mode-map "\C-ce" 'c-end-comment-box)

   )

)
```

The actual code for your program consists of two parts: variables and executable instructions. Variables are used to hold the data used by your program. Executable instructions tell the computer what to do with the data.

3.1 Common Coding Practices

A *variable* is a place in the computer's memory for storing a value. C identifies that place by the variable name. Names can be of any length and should be chosen so their meanings are clear. (Actually, a length limit exists, but it is so large that you probably will never encounter it.) Every variable in C must be declared. Variable declarations will be discussed in Chapter 9. The following declaration tells C that we're going to use three integer (**int**) variables: p, q, and r:

```
int p,q,r;
```

But what are these variables for? The reader has no idea. They could represent the number of angels on the head of a pin or the location and acceleration of a plasma bolt in a game of Space Invaders. Avoid abbreviations. Exs. abb. are diff. to rd. and hd. to ustnd. (Excess abbreviations are difficult to read and hard to understand.)

Now consider another declaration:

```
int account_number;
int balance_owed;
```

Now we know that we're dealing with an accounting program, but we could still use some more information. For example, is the balance_owed in dollars or cents? We should have added a comment after each declaration to explain what we were doing. For example:

```
int account_number;      /* Index for account table */
int balance_owed;        /* Total owed us (in pennies)*/
```

By putting a comment after each declaration, we, in effect, create a mini-dictionary where we define the meaning of each variable name. Because the definition of each variable is in a known place, it's easy to look up the meaning of a name. (Programming tools like editors, cross-referencers, and searching tools such as grep can also help you quickly find a variable's definition.)

Units are very important. I was once asked to modify a program that converted plot data files from one format to another. Many different units of length were used throughout the program, and none of the variable declarations were commented. I tried very hard to figure out what was going on, but I could not determine what units were being used in the program. Finally, I gave up and put the following comment in the program:

```
/********************************************************
 * Note:  I have no idea what the input units are, nor  *
 *      do I have any idea what the output units are,   *
 *      but I have discovered that if I divide by 3     *
 *      the plot sizes look about right.               *
 ********************************************************/
```

You should take every opportunity to make sure that your program is clear and easy to understand. Do not be clever. Clever kills. Clever makes for unreadable and unmaintainable programs. Programs, by their nature, are extremely complex. Anything that you can to do to cut down on this complexity will make your programs better. Consider the following code, written by a very clever programmer:[2]

[2] Note that the first version of this code:

```
while ('\n' != (*p++ = *q++));
```

It is almost impossible for the reader to tell at a glance what this mess does. Properly written this should be:

```
while (1) {
    *destination_ptr = *source_ptr;
    if (*destination_ptr == '\n')
       break;    /* Exit the loop if at end of line */
    destination_ptr++;
    source_ptr++;
}
```

Although the second version is longer, it is much clearer and easier to understand.[3] Even a novice programmer who does not know C well can tell that this program has something to do with moving data from a source to a destination.

[3] Expert C programmers can spot a slight difference between the two versions, but both do the required job.

The computer doesn't care which version is used. A good compiler will generate the same machine code for both versions. The programmer is the beneficiary of the verbose code.

3.2 Coding Religion

Computer scientists have devised many programming styles. These include structured programming, top-down programming, goto-less programming, and object-oriented design (OOD). Each of these styles has its own following or cult. I use the term "religion" because people are taught to follow the rules blindly without knowing the reasons behind them. For example, followers of the goto-less cult will never use a **goto** statement, even when it is natural to do so.

The rules presented in this book result from years of programming experience. I have discovered that by following these rules, I can create better programs. You do not have to follow them blindly. If you find a better system, by all means use it. (If your solution really works, drop me a line. I'd like to use it too.)

3.3 Indentation and Code Format

In order to make programs easier to understand, most programmers indent their programs. The general rule for a C program is to indent one level for each new block or conditional. In our previous example, there are three levels of logic, each with its own indentation level. The **while** statement is outermost. The statements inside the **while** are at the next level. Finally, the statement inside the **if** (**break**) is at the innermost level.

There are two styles of indentation, and a vast religious war is being raged in the programming community as to which style is better. The first is the short form:

```
while (! done) {
    printf("Processing\n");
    next_entry();
}
if (total <= 0) {
    printf("You owe nothing\n");
    total = 0;
} else {
    printf("You owe %d dollars\n", total);
    all_totals = all_totals + total;
}
```

In this case, curly braces ({}) are put on the same line as the statements. The other style puts the {} on lines by themselves:

```
while (! done)
{
    printf("Processing\n");
    next_entry();
}
if (total <= 0)
{
   printf("You owe nothing\n");
   total = 0;
}
else
{
   printf("You owe %d dollars\n", total);
   all_totals = all_totals + total;
}
```

Both formats are frequently used. You should use the format you feel most comfortable with. This book uses the short format because it's more compact and therefore saves book space.

The amount of indentation is left to the programmer. Two, four, and eight spaces are common. Studies have shown that a four-space indent makes the code most readable. However, being consistent in your indentation is far more important than the indention size you use.

Some editors, like the UNIX Emacs editor, the Turbo C++, Borland C++, and Microsoft Visual C++ internal editors, contain code that automatically indents your programs as you create them. Although these editor-based indentation systems are not perfect, they do go a long way to helping you create properly formatted code.

3.4 Clarity

A program should read like a technical paper. It should be organized into sections and paragraphs. Procedures form a natural section boundary. (We'll learn about function in Chapter 9.) You must organize your code into paragraphs. You should begin a paragraph with a topic-sentence comment and separate the comment from other paragraphs with a blank line. For example:

```
/* poor programming practice */
temp = box_x1;
box_x1 = box_x2;
box_x2 = temp;
temp = box_y1;
box_y1 = box_y2;
box_y2 = temp;
```

A better version would be:

```
/*
 * Swap the two corners
 */

/* Swap X coordinate */
temp = box_x1;
box_x1 = box_x2;
box_x2 = temp;

/* Swap Y coordinate */
temp = box_y1;
box_y1 = box_y2;
box_y2 = temp;
```

3.5 Simplicity

Your program should be simple. Some general rules of thumb are:

- A single function should not be longer than two or three pages. (See Chapter 9.) If the function gets longer, it can probably be split into two simpler functions. This rule comes about because the human mind can only hold so much in short-term memory. Three pages are about the most that the human mind can wrap itself around in one sitting.

  Also if your function goes beyond the three-page limit, it probably doesn't define a single operation, or probably contains too much detail.

- Avoid complex logic like multiply nested **if**s. The more complex your code, the more indentation levels you will need. When you start running into the right margin, you should consider splitting your code into multiple procedures, to decrease the level of complexity.

- Did you ever read a sentence, like this one, in which the author went on and on, stringing together sentence after sentence with the word "and," and didn't seem to understand the fact that several shorter sentences would do the job much better, and didn't it bother you? C statements should not go on forever. Long statements should be avoided. If it looks like an equation or formula is going to be longer than one or two lines, you should split it into two shorter equations.
- Finally, the most important rule: make your program as simple and easy to understand as possible, even if you must break some of the rules. The goal is clarity, and the rules given in this chapter are designed to help you accomplish that goal. If they get in the way, get rid of them. I have seen one program with a single statement that spanned over 20 pages; however, because of the specialized nature of the program, this statement was simple and easy to understand.

3.6 Summary

A program should be concise and easy to read. It must serve as a set of computer instructions, but also as a reference work describing the algorithms and data used inside it. Everything should be documented with comments. Comments serve two purpo ses. First, they tell the programmer to follow the code, and second, they help the programmer remember what he did.

Class discussion: Create a style sheet for class assignments. Discuss what comments should go into the programs and why.