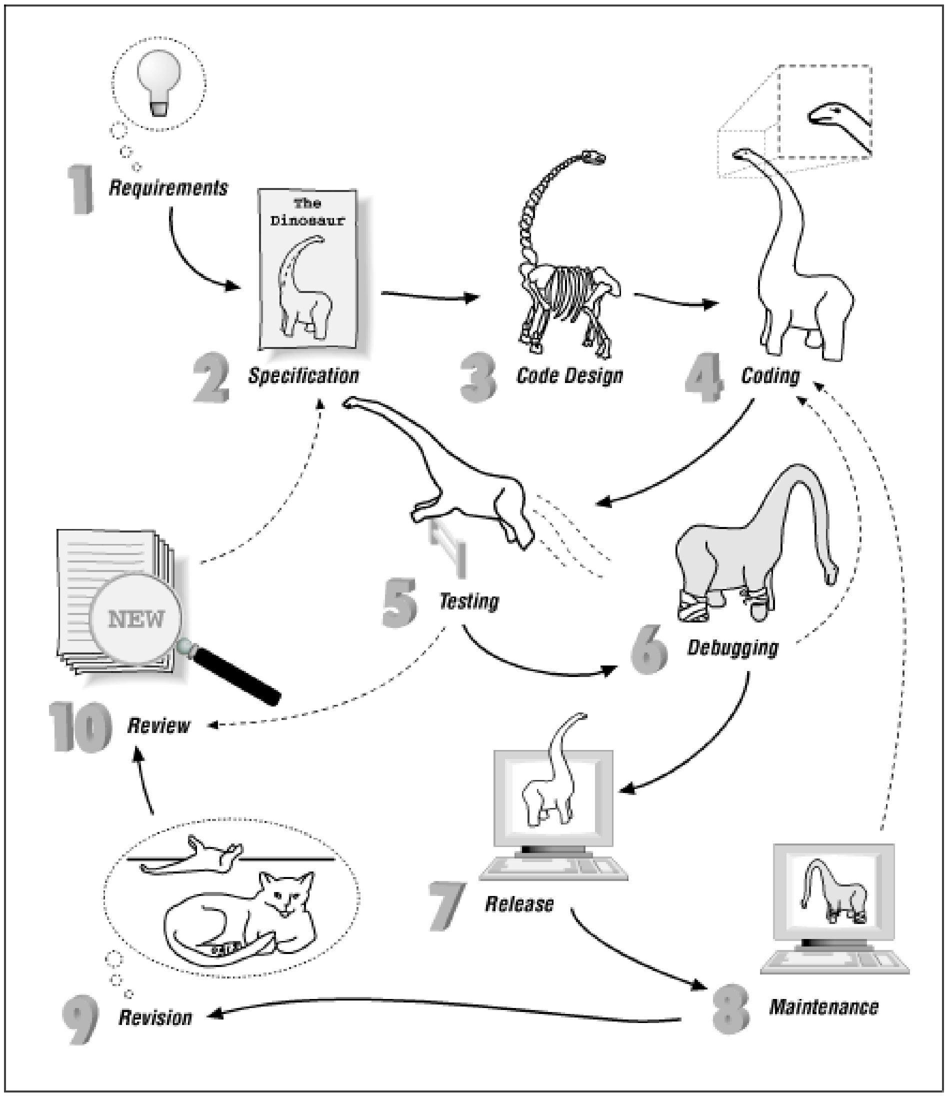
Chapter 7. Programming Process

Programming is more than just writing code. Software has a life cycle. It is born, grows up, becomes mature, and finally dies, only to be replaced by a newer, younger product. Figure 7-1 illustrates the life cycle of a program. Understanding this cycle is important because, as a programmer, you will spend only a small amount of time writing new code. Most programming time is spent modifying and debugging existing code. Software does not exist in a vacuum; it must be documented, maintained, enhanced, and sold. In this chapter, we will take a look at a small programming project using one programmer. Larger projects that involve many people will be discussed in Chapter 18. Although our final code is less than 100 lines, the principles used in its construction can be applied to programs with thousands of lines of code.

Figure 7-1. Software life cycle



The major steps in making a program are:

- **Requirements.** Programs start when someone gets an idea and starts to implement it. The requirement document describes, in very general terms, what is wanted.
- **Program specification.** The specification is a description of what the program does. In the beginning, a *preliminary specification* is used to describe what the program is going to do. Later, as the program becomes more refined, so does the specification. Finally, when the program is finished, the specification serves as a complete description of what the program does.
- **Code design.** The programmer does an overall design of the program. The design should include major algorithms, module definitions, file formats, and data structures.

- **Coding.** The next step is writing the program. This step involves first writing a prototype and then filling it in to create the full program.
- **Testing.** The programmer should design a test plan and then use it to test his program. When possible, the programmer should have someone else test the program.
- **Debugging.** Unfortunately, very few programs work the first time. They must be corrected and tested again.
- **Release.** The program is packaged, documented, and sent out into the world to be used.
- **Maintenance.** Programs are never perfect. Bugs will be found and will need correction. This step is the maintenance phase of programming.
- **Revision and updating.** After a program has been working for a while, the users will want changes, such as more features or more intelligent algorithms. At this point, a new specification is created and the process starts again.

7.1 Setting Up

The operating system allows you to group files in directories. Just as file folders serve as a way of keeping papers together in a filing cabinet, directories serve as a way of keeping files together. (Windows 95 goes so far as to call its directories "folders.") In this chapter, we create a simple calculator program. All the files for this program are stored in a directory named *calc*. In UNIX, we create a new directory under our home directory and then move to it, as shown in the following example:

```
% cd ~
% mkdir calc
% cd ~/calc
```

On MS-DOS type:

```
C:\> cd \
C:\> mkdir calc
C:\> cd \calc
C:\CALC>
```

This directory setup is extremely s imple. As you generate more and more programs, you will probably want a more elaborate directory structure. More information on how to organize directories or folders can be found in your operating system manual.

7.2 Specification

For this chapter, we assume that we have the requirement to "write a program that acts like a four-function calculator." Typically, the requirements that you are given is vague and incomplete. The programmer refines it into something that exactly defines the program that he is going to produce. So the first step is to write a preliminary users' specification document that describes what your program is going to do and how to use it. The document does not describe the internal structure of the program or the algorithm you plan on using. A sample specification for our four-function calculator appears below in Calc: A Four-Function Calculator.

The preliminary specification serves two purposes. First, you should give it to your boss (or customer) to make sure that you agree on what each of you said. Second, you can circulate it among your colleagues and see if they have any suggestions or corrections.

This preliminary specification was circulated and received the comments:

- How are you going to get out of the program?
- What happens when you try to divide by 0?

Calc: A Four-Function Calculator

Preliminary Specification

Dec. 10, 1989

Steve Oualline

Warning: This document is a preliminary specification. Any resemblance to any software living or dead is purely coincidental.

Calc is a program that allows the user to turn a $2,000 computer into a $1.98 four-function calculator. The program will add, subtract, multiply, and divide simple integers.

When the program is run, it will zero the result register and display the register's contents. The user can then type in an operator and number. The result will be updated and displayed. The following operators are valid:

Operator	Meaning

+	Addition
-	Subtraction
*	Multiplication
/	Division

For example (user input is in boldface):

```
calc
Result:  0
Enter operator and number:  + 123
Result:  123
Enter operator and number:  - 23
Result:  100
Enter operator and number:  / 25
Result:  4
Enter operator and number:  * 4
Result:  16
0
```

So, we add a new operator, q for quit, and we add the statement:

"Dividing by 0 results in an error message and the result register is left unchanged."

IV + IX = XIII?

A college instructor once gave his students an assignment to "write a four-function calculator." One of his students noticed that this assignment was a very loose specification and decided to have a little fun. The professor didn't say what sort of numbers had to be used, so the student created a program that worked only with Roman numerals (V+ III = VIII). The program came with a complete user manual—written in Latin.

7.3 Code Design

After the preliminary specification has been approved, we can start designing code. In the code design phase, the programmer plans his work. In large programming projects involving many people, the code would be broken up into modules, to be assigned to the programmers. At this stage, file formats are planned, data structures are designed, and major algorithms are decided upon.

Our simple calculator uses no files and requires no fancy data structures. What's left for this phase is to design the major algorithm. Outlined in pseudo code, a shorthand halfway between English and real code, the major algorithm is:

```
Loop
  Read an operator and number
  Do the calculation
  Display the result
End-Loop
```

7.4 Prototype

After the code design is completed, we can begin writing the program. But rather than try to write the entire program at once and then debug it, we will use a method called *fast prototyping*. We implement the smallest portion of the specification that will still do something. In our case, we will cut our four functions down to a one-function calculator. After we get this small part working, we can build the rest of the functions onto this stable foundation. Also, the prototype gives the boss something to look at and play with, giving him a good idea of the project's direction. Good communication is the key to good programming, and the more you can show someone the better. The code for the first version of our four-function calculator is found in Example 7-1.

Example 7-1. *calc1/calc1.c*

```
#include <stdio.h>
char  line[100];/* line of data from the input */
int   result;   /* the result of the calculations */
char  operator; /* operator the user specified */
int   value;    /* value specified after the operator */

int main()
{
    result = 0; /* initialize the result */

    /* Loop forever (or till we hit the break statement) */
    while (1) {
        printf("Result: %d\n", result);

        printf("Enter operator and number: ");
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c %d", &operator, &value);
        if (operator = '+') {
            result += value;
        } else {
            printf("Unknown operator %c\n", operator);
        }
    }
}
```

The program begins by initializing the variable `result` to 0. The main body of the program is a loop starting with:

```
while (1) {
```

This loop will repeat until a **break** statement is reached. The code:

```
printf("Enter operator and number: ");
    fgets(line, sizeof(line), stdin);
    sscanf(line,"%c %d", &operator, &value);
```

asks the user for an operator and number. These are scanned and stored in the variables `operator` and `value`. Next, we start checking the operators. If the operator is a plus sign (+), we perform an addition using the line:

```
if (operator = '+') {
    result += value;
```

So far, we only recognize the plus (+) operator. As soon as this operator works correctly, we will add more operators by adding more **if** statements.

Finally, if an illegal operator is entered, the line:

```
} else {
     printf("Unknown operator %c\n", operator);
}
```

writes an error message telling the user that he made a mistake.

7.5 Makefile

After the source has been entered, it needs to be compiled and linked. Up until now we have been running the compiler manually. This process is somewhat tedious and prone to error. Also, larger programs consist of many modules and are extremely difficult to compile by hand. Fortunately, both UNIX and MS-DOS/Windows have a utility called *make*[1] that will handle the details of compilation. For now, use this example as a template and substitute the name of your program in place of "calc."

make will be discussed in detail in Chapter 18. The program looks at the file called *Makefile* for a description of how to compile your program and runs the compiler for you.

[1] Microsoft's Visual C++ calls this utility nmake.

Because the Makefile contains the rules for compilation, it is customized for the compiler. The following is a set of *Makefile*s for all of the compilers described in this book.

7.5.1 Generic UNIX

```
File: calc1/makefile.unx
#-----------------------------------------------#
#       Makefile for Unix systems              #
#    using a GNU C compiler                   #
#-----------------------------------------------#
CC=gcc
CFLAGS=-g
#
# Compiler flags:
#       -g      -- Enable debugging

calc1: calc1.c
       $(CC) $(CFLAGS) -o calc1 calc1.c

clean:
       rm -f calc1
```

The *make* utility is responsible for one of the nastiest surprises for unsuspecting users. The line:

```
$(CC) $(CFLAGS) -o calc1 calc1.c
```

must begin with a tab. Eight spaces won't work. A space and a tab won't work. The line must start with a tab. Check your editor and make sure that you can tell the difference between a tab and bunch of spaces.

7.5.2 UNIX with the Free Software Foundation's gcc Compiler

```
File: calc1/makefile.gcc
#-----------------------------------------------#
#      Makefile for UNIX systems              #
#    using a GNU C compiler                   #
#-----------------------------------------------#
CC=gcc
CFLAGS=-g -D__USE_FIXED_PROTOTYPES__ -ansi
#
# Compiler flags:
#       -g     -- Enable debugging
#       -Wall   -- Turn on all warnings (not used since it gives away
#                      the bug in this program)
#       -D__USE_FIXED_PROTOTYPES__
#             -- Force the compiler to use the correct headers
#       -ansi   -- Don't use GNU extensions.  Stick to ANSI C.

calc1: calc1.c
       $(CC) $(CFLAGS) -o calc1 calc1.c

clean:
       rm -f calc1
```

7.5.3 Borland C++

```
[File: calc1/makefile.bcc]
#
# Makefile for Borland's Borland-C++ compiler
#
CC=bcc
#
# Flags
#       -N  -- Check for stack overflow
#       -v  -- Enable debugging
#       -w  -- Turn on all warnings
#       -ml -- Large model
#
CFLAGS=-N -v -w -ml

calc1.exe: calc1.c
```

```
	$(CC) $(CFLAGS) -ecalc1 calc1.c

clean:
	erase calc1.exe
```

7.5.4 Turbo C++

```
File: calc1/makefile.tcc
#-----------------------------------------------#
#      Makefile for DOS systems              #
#    using a Turbo C compiler.                #
#-----------------------------------------------#
CC=tcc
CFLAGS=-v -w -ml

calc1.exe: calc1.c
	$(CC) $(CFLAGS) -ecalc1.exe calc1.c

clean:
	del calc1.exe
```

7.5.5 Visual C++

```
[File: calc1/makefile.msc]
#-----------------------------------------------#
#      Makefile for DOS systems              #
#    using a Microsoft Visual C++ compiler.    #
#-----------------------------------------------#
CC=cl
#
# Flags
#      AL -- Compile for large model
#      Zi -- Enable debugging
#      W1 -- Turn on warnings
#
CFLAGS=/AL /Zi /W1

calc1.exe: calc1.c
	$(CC) $(CFLAGS) calc1.c

clean:
	erase calc1.exe
```

To compile the program, just execute the make command. make will determine which compilation commands are needed and then execute them.

make uses the modification dates of the files to determine whether or not a compile is necessary. Compilation creates an object file. The modification date of the object file is later than the modification date of its source. If the source is edited, the source's modification date is updated, and the object file is then out of date. make checks these dates, and if the source was modified after the object, make recompiles the object.

7.6 Testing

After the program is compiled without errors, we can move on to the testing phase. Now is the time to start writing a test plan. This document is simply a list of the steps we perform to make sure the program works. It is written for two reasons:

- If a bug is found, we want to be able to reproduce it.
- If we change the program, we will want to retest it to make sure new code did not break any of the sections of the program that were previously working.

Our test plan starts out as:

```
Try the following operations:
+ 123  Result should be 123
+ 52   Result should be 175
x 37   Error message should be output
```

After we run the program, we get:

```
Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
Result: 175
Enter operator and number: x 37
Result: 212
```

Something is clearly wrong. The entry x 37 should have generated an error message, but it didn't. A bug is in the progra m. So we begin the debugging phase. One of the advantages of making a small working prototype is that we can isolate errors early.

7.7 Debugging

First we inspect the program to see if we can detect the error. In such a small program we can easily spot the mistake. However, let's assume that instead of a 21-line program, we have a much larger program containing 5,000 lines. Such a program would make inspection more difficult, so we need to proceed to the next step.

Most systems have C debugging programs; however, each system is different. Some systems have no debugger. In such a case, we must resort to a diagnostic print statement. The technique is simple: put a `printf` at the points at which you know the data is good (just to make sure the data is *really* good). Then put a `printf` at points at which the data is bad. Run the program and keep putting in `printf` statements until you isolate the area in the program that contains the mistake. Our program, with diagnostic `printf` statements added, looks like:

```
printf("Enter operator and number: ");
fgets(line, sizeof(line), stdin);
sscanf("%d %c", &value, &operator);
printf("## after scanf %c\n", operator);
if (operator = '+') {
    printf("## after if %c\n", operator);
    result += value;
```

> The ## at the beginning of each `printf` is used to indicate a temporary debugging `printf`. When the debugging is complete, the ## makes the associated statements easy to identify and remove.

Running our program again results in:

```
Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
## after scanf +
## after if +
Result: 175
Enter operator and number: x 37
## after scanf x
## after if +
Result: 212
```

From this example we see that something is going wrong with the **if** statement. Somehow, the variable operator is an "x" going in and a "+" coming out. Closer inspection reveals that we have made the old mistake of using = instead of ==. After we fix this bug, the program runs correctly. Building on this working foundation, we add code for the other operators: dash (-), asterisk (*), and slash (/). The result is shown in Example 7-2.

Example 7-2. *calc2/calc2.c*

```
#include <stdio.h>
char  line[100];/* line of text from input */

int   result;   /* the result of the calculations */
char  operator; /* operator the user specified */
int   value;    /* value specified after the operator */

int main()
{
    result = 0; /* initialize the result */

    /* loop forever (or until break reached) */
    while (1) {
        printf("Result: %d\n", result);
        printf("Enter operator and number: ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%c %d", &operator, &value);

        if ((operator == 'q') || (operator == 'Q'))
            break;

        if (operator == '+') {
            result += value;
        } else if (operator == '-') {
            result -= value;
        } else if (operator == '*') {
            result *= value;
        } else if (operator == '/') {
            if (value == 0) {
                printf("Error:Divide by zero\n");
                printf("   operation ignored\n");
            } else
```

```
                result /= value;
        } else {
            printf("Unknown operator %c\n", operator);
        }
    }
    return (0);
}
```

We expand our test plan to include the new operators and try it again:

```
+ 123    Result should be 123
+ 52     Result should be 175
x 37     Error message should be output
- 175    Result should be zero
+ 10     Result should be 10
/ 5      Result should be 2
/ 0      Divide by zero error
* 8      Result should be 16
q        Program should exit
```

While testing the prog ram, we find that, much to our surprise, the program works. The word "Preliminary" is removed from the specification, and the program, test plan, and specification are released.

7.7 Debugging

First we inspect the program to see if we can detect the error. In such a small program we can easily spot the mistake. However, let's assume that instead of a 21-line program, we have a much larger program containing 5,000 lines. Such a program would make inspection more difficult, so we need to proceed to the next step.

Most systems have C debugging programs; however, each system is different. Some systems have no debugger. In such a case, we must resort to a diagnostic print statement. The technique is simple: put a `printf` at the points at which you know the data is good (just to make sure the data is *really* good). Then put a `printf` at points at which the data is bad. Run the program and keep putting in `printf` statements until you isolate the area in the program that contains the mistake. Our program, with diagnostic `printf` statements added, looks like:

```
printf("Enter operator and number: ");
fgets(line, sizeof(line), stdin);
sscanf("%d %c", &value, &operator);
printf("## after scanf %c\n", operator);
```

```
if (operator = '+') {
    printf("## after if %c\n", operator);
    result += value;
```

The ## at the beginning of each `printf` is used to indicate a temporary debugging `printf`. When the debugging is complete, the ## makes the associated statements easy to identify and remove.

Running our program again results in:

```
Result: 0
Enter operator and number: + 123
Result: 123
Enter operator and number: + 52
## after scanf +
## after if +
Result: 175
Enter operator and number: x 37
## after scanf x
## after if +
Result: 212
```

From this example we see that something is going wrong with the **if** statement. Somehow, the variable operator is an "x" going in and a "+" coming out. Closer inspection reveals that we have made the old mistake of using = instead of ==. After we fix this bug, the program runs correctly. Building on this working foundation, we add code for the other operators: dash (-), asterisk (*), and slash (/). The result is shown in Example 7-2.

Example 7-2. *calc2/calc2.c*

```
#include <stdio.h>
char  line[100];/* line of text from input */

int   result;   /* the result of the calculations */
char  operator; /* operator the user specified */
int   value;    /* value specified after the operator */

int main()
{
    result = 0; /* initialize the result */

    /* loop forever (or until break reached) */
    while (1) {
       printf("Result: %d\n", result);
       printf("Enter operator and number: ");

       fgets(line, sizeof(line), stdin);
       sscanf(line, "%c %d", &operator, &value);

       if ((operator == 'q') || (operator == 'Q'))
          break;

       if (operator == '+') {
          result += value;
       } else if (operator == '-') {
          result -= value;
       } else if (operator == '*') {
          result *= value;
       } else if (operator == '/') {
          if (value == 0) {
              printf("Error:Divide by zero\n");
              printf("   operation ignored\n");
          } else
              result /= value;
       } else {
          printf("Unknown operator %c\n", operator);
       }
    }
    return (0);
}
```

We expand our test plan to include the new operators and try it again:

```
+ 123    Result should be 123
+ 52     Result should be 175
x 37     Error message should be output
- 175    Result should be zero
+ 10     Result should be 10
/ 5      Result should be 2
/ 0      Divide by zero error
* 8      Result should be 16
q        Program should exit
```

While testing the program, we find that, much to our surprise, the program works. The word "Preliminary" is removed from the specification, and the program, test plan, and specification are released.

7.8 Maintenance

Good programme rs put each program through a long and rigorous testing process before releasing it to the outside world. Then the first user tries the program and almost immediately finds a bug. This step is the maintenance phase. Bugs are fixed, the program is tested (to make sure that the fixes didn't break anything), and the program is released again.

7.9 Revisions

Although the program is officially finished, we are not done with it. After the program is in use for a few months, someone will come to us and ask, "Can you add a modulus operator?" So we revise the specifications, add the change to the program, update the test plan, test the program, and then release the program again.

As time passes, more people will come to us with additional requests for changes. Soon our program has trig functions, linear regressions, statistics, binary arithmetic, and financial calculations. Our design is based on the concept of one-character operators. Soon we find ourselves running out of characters to use. At this point, our program is doing work far in excess of what it was initially designed to do. Sooner or later we reach the point where the program needs to be scrapped and a new one written from scratch. At that point, we write a preliminary specification and start the process again.

7.10 Electronic Archaeology

Electronic archeology is the art of digging through old code to discover amazing things (like how and why the code works).

Unfortunately, most programmers don't start a project at the design step. Instead, they are immediately thrust into the maintenance or revision stage and must face the worst possible job: understanding and modifying someone else's code.

Your computer can aid greatly in your search to discover the true meaning of someone else's code. Many tools are available for examining and formatting code. Some of these tools include:

- **Cross references.** These programs have names like `xref`, `cxref`, and `cross`. System V Unix has the utility `cscope`. A cross reference prints out a list of variables and indicates where each variable is used.
- **Program indenters.** Programs like `cb` and `indent` will take a program and indent it *correctly* (correct indentation is something defined by the tool maker).
- **Pretty printers.** A pretty printer such as `vgrind` or `cprint` will take the source and typeset it for printing on a laser printer.
- **Call graphs.** On System V Unix the program `cflow` can be used to analyze the program. On other systems there is a public-domain utility, `calls`, which produces call graphs. The call graphs show who calls whom and who is called by whom.

Which tools should you use? Whichever work for you. Different programmers work in different ways. Some of the techniques for examining code are listed in the sections below. Choose the ones that work for you and use them.

7.11 Marking Up the Program

Take a printout of the program and make notes all over it. Use red or blue ink so that you can tell the difference between the printout and the notes. Use a highlighter to emphasize important sections. These notes are useful; put them in the program as comments, then make a new printout and start the process again.

7.12 Using the Debugger

The debugger is a great tool for understanding how something works. Most debuggers allow the user to step through the program one line at a time, examining variables and discovering how things really work. After you find out what the code does, make notes and put them in the program as comments.

7.13 Text Editor as a Browser

One of the best tools for going through someone else's code is your text editor. Suppose you want to find out what the variable `sc` is used for. Use the search command to find the first place `sc` is used. Search again and find the second time it is used. Continue searching until you know what the variable does.

Suppose you find out that `sc` is used as a sequence counter. Because you're already in the editor, you can easily do a global search and replace to change `sc` to `sequence_counter`. (Disaster warning: *Before* you make the change, make sure that `sequence_counter` is not already defined as a variable. Also, watch out for unwanted replacements, such as changing the *sc* in "escape.") Comment the declaration and you're on your way to creating an understandable program.

7.14 Add Comments

Don't be afraid of putting any information you have, no matter how little, into the comments. Some of the comments I've used include:

```
int state;  /* Controls some sort of state machine */
int rmxy;   /* Something to do with color correction ? */
```

Finally, there is a catch-all comment:

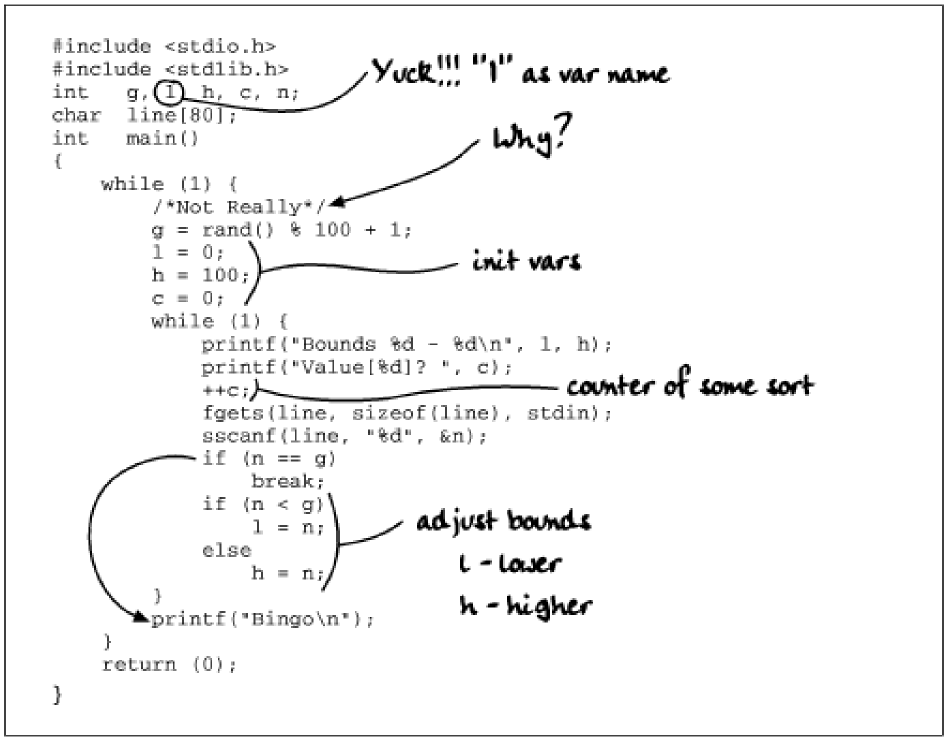
```
int idn;    /* ??? */
```

which means "I have no idea what this variable does." Even though the variable's purpose is unknown, it is now marked as something that needs more work.

As you go through someone else's code adding comments and improving style, the structure will become clearer to you. By inserting notes (comments), you make the code better and easier to understand for future programmers.

For example, suppose we are confronted with the following program written by someone from "The-Terser-the-Better" school of programming. Our assignment is to figure out what this code does. First, we pencil in some comments, as shown in Figure 7-2.

Figure 7-2. A terse program



```
#include <stdio.h>
#include <stdlib.h>
int   g, l, h, c, n;
char  line[80];
int   main()
{
    while (1) {
        /*Not Really*/
        g = rand() % 100 + 1;
        l = 0;
        h = 100;
        c = 0;
        while (1) {
            printf("Bounds %d - %d\n", l, h);
            printf("Value[%d]? ", c);
            ++c;
            fgets(line, sizeof(line), stdin);
            sscanf(line, "%d", &n);
            if (n == g)
                break;
            if (n < g)
                l = n;
            else
                h = n;
        }
        printf("Bingo\n");
    }
    return (0);
}
```

Our mystery program requires some work. After going through it and applying the principles described in this section, we get a well-commented, easy-to-understand program, such as Example 7-3.

Example 7-3. *good/good.c*

```
/********************************************************
 * guess -- A simple guessing game.                     *
 *                                                      *
 * Usage:                                               *
 *     guess                                            *
 *                                                      *
 *      A random number is chosen between 1 and 100.    *
 *      The player is given a set of bounds and         *
 *      must choose a number between them.              *
 *      If the player chooses the correct number, he wins. *
 *      Otherwise, the bounds are adjusted to reflect   *
 *      the player's guess and the game continues.      *
 *                                                      *
 *                                                      *
 * Restrictions:                                        *
 *      The random number is generated by the statement    *
 *      rand() % 100.  Because rand() returns a number     *
 *      0 <= rand() <= maxint  this slightly favors       *
 *      the lower numbers.                               *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>
int   number_to_guess;  /* random number to be guessed */
int   low_limit;       /* current lower limit of player's range */
int   high_limit;      /* current upper limit of player's range */
int   guess_count;      /* number of times player guessed */
int   player_number;    /* number gotten from the player */
char  line[80];        /* input buffer for a single line */
int main()
{
    while (1) {
        /*
         * Not a pure random number, see restrictions
         */
        number_to_guess = rand() % 100 + 1;

        /* Initialize variables for loop */
        low_limit = 0;
        high_limit = 100;
        guess_count = 0;

        while (1) {
            /* tell user what the bounds are and get his guess */
            printf("Bounds %d - %d\n", low_limit, high_limit);
            printf("Value[%d]? ", guess_count);

            ++guess_count;

            fgets(line, sizeof(line), stdin);
            sscanf(line, "%d", &player_number);

            /* did he guess right? */
            if (player_number == number_to_guess)
                break;

            /* adjust bounds for next guess */
            if (player_number < number_to_guess)
                low_limit = player_number;
            else
```

```
                high_limit = player_number;

        }
        printf("Bingo\n");
    }
}
```

7.15 Programming Exercises

For each of these assignments, follow the software life cycle from specification through release.

Exercise 7-1: Write a program to convert English units to metric (i.e., miles to kilometers, gallons to liters, etc.). Include a specification and a code design.

Exercise 7-2: Write a program to perform date arithmetic such as how many days there are between 6/6/90 and 4/3/92. Include a specification and a code design.

Exercise 7-3: A serial transmission line can transmit 960 characters each second. Write a program that will calculate the time required to send a file, given the file's size. Try the program on a 400MB (419,430,400-byte) file. Use appropriate units. (A 400MB file takes days.)

Exercise 7-4: Write a program to add an 8% sales tax to a given amount and round the result to the nearest penny.

Exercise 7-5: Write a program to tell if a number is prime.

Exercise 7-6: Write a program that takes a series of numbers and counts the number of positive and negative values.