XIX
CHAPTER

Programming Style and Standards

This chapter focuses primarily on the layout of your code. Usage of comments, white space, variable and function naming standards, and bracing techniques are covered. In this chapter, you will learn that the use of comments and white space do not affect your program's speed, size, or efficiency. You will also learn three standards of putting braces in your code. When it comes to naming your variables and functions, you will learn two notation styles ("camel" and "Hungarian") and will learn that putting underscores in your variable and function names makes them more readable. You will also learn how to name your functions and how long your function and variable names should be.

In addition to naming conventions and standards, several general programming topics are covered, such as recursion (what it is and how to use it); null loops; infinite loops; iterative processing via the `while`, `do...while`, and `for` loops; the difference between the `continue` and `break` statements; and the best way to represent true and false in your programs.

This chapter has many topics to cover, so hold on tight—and be sure to pay attention to the naming styles and conventions. They could help make your programs much more readable and understandable.

XIX.1: Should the underscore be used in variable names?

Answer:

Using the underscore in variable names is a matter of style. There is nothing wrong with using underscores or avoiding them altogether. The important thing to remember is to be consistent—use the same naming conventions throughout your application. This means that if you are programming in a team environment, you and your team members should decide on a naming convention and stick with it. If not everyone uses the same convention, your program integration will be horrible and hard to read. Additionally, you should adhere to the style used by the third-party libraries (if any) that are used in your program. If at all possible, use the same naming convention as the third-party library. Doing so will make your programs more readable and consistent.

Many C programmers find the underscore method of variable naming to be convenient. Perhaps this is because the underscore method tends to be very readable. For instance, the following two function names are similar, but one could argue that the underscored function name is more readable:

```
check_disk_space_available(selected_disk_drive);
```

```
CheckDiskSpaceAvailable(SelectedDiskDrive);
```

The second notation used here is called camel notation—see FAQ XIX.5 for an explanation of camel notation.

Cross Reference:

XIX.2: Can a variable’s name be used to indicate its data type?

XIX.5: What is camel notation?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.9: How many letters long should variable names be? What is the ANSI standard for significance?

XIX.10: What is Hungarian notation, and should I use it?

XIX.2: Can a variable’s name be used to indicate its data type?

Answer:

Yes, indicating the data type in a variable’s name has become a very popular convention in today’s world of large, complex systems. Usually, the variable’s type is represented by one or two characters, and the variable name is prefixed with these characters. A well-known naming convention that uses this technique is called *Hungarian* notation, named after Microsoft programmer Charles Simonyi. Table XIX.2 contains some common prefixes.

Table XIX.2. Some common Hungarian notation prefixes.

Data Type	*Prefix*	*Example*
`char`	`c`	`cInChar`
`int`	`i`	`iReturnValue`
`long`	`l`	`lNumRecs`
`string`	`sz`	`szInputString` (terminated by zero byte)
`int array`	`ai`	`aiErrorNumbers`
`char*`	`psz`	`pszInputString`

Environments such as Microsoft Windows make heavy use of Hungarian notation or some derivative. Other fourth-generation environments, such as Visual Basic and Access, have also adopted a variation of the Hungarian notation.

You don't have to stick exactly to a particular notation when writing your programs—it is perfectly OK to create your own customized derivative. This is especially true when you are creating notations for your own `typedefs`. For instance, if you have a `typedef` named `SOURCEFILE` that keeps information such as the source filename, handle, number of lines, last compile date and time, number of errors, and so on, you might want to create a prefix notation such as "sf" (source file). This way, when you see a variable named `sfBuffer`, you know that it refers to a variable that holds the contents of your `SOURCEFILE` structure.

Whatever the case may be, it is a good idea to adopt some form of naming convention for your variables and functions. This is especially true when you are working on large projects with many different programmers or when you are working in environments such as Microsoft Windows. Adopting a well-thought-out naming convention might help you make your programs more readable, especially if your code is extremely complex.

Cross Reference:

XIX.1: Should the underscore be used in variable names?

XIX.5: What is camel notation?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.9: How many letters long should variable names be? What is the ANSI standard for significance?

XIX.10: What is Hungarian notation, and should I use it?

XIX.3: Does the use of comments affect program speed, executable size, or efficiency?

Answer:

No. When your program is compiled, all comments are ignored by the compiler, and only executable statements are parsed and eventually put into the final compiled version of the program.

Because comments have no bearing on your program's speed, size, or efficiency, you should use comments as often as possible. Each of your program modules should have a header that explains the purpose of the module and any special considerations. Similarly, each function you write should have information such as author name, date written, modification dates and reasons, parameter usage guidelines, description of the function, and so forth. This information will help other programmers understand your programs better, or it might help you remember some key ideas of implementation later.

You also should use comments in your source code (in-between programming statements). For instance, if you have a particular portion of code that is complex or if you feel that something needs a bit more clarity, do not hesitate to put a comment in the code. Doing so might take a little more time up front, but you or someone else might be able to save several hours of valuable time by glancing at the comment and immediately knowing what the programmer had in mind.

See FAQ XIX.4 for an example program that shows how using comments, white space, and the underscore naming convention can make your code much cleaner and much more understandable by others.

Cross Reference:

XIX.4: Does the use of white space affect program speed, executable size, or efficiency?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.4: Does the use of white space affect program speed, executable size, or efficiency?

Answer:

No. As with comments, all white space is ignored by the compiler. When your program is compiled, all white space and comments are ignored, and only the executable statements are parsed and eventually put into the final compiled version of the program.

The use of white space in your C programs can help make your programs more readable and improve clarity by separating out your executable statements, functions, comments, and so forth. Many times, you improve your program's readability by simply adding blank lines between statements. For instance, consider the following portion of code:

```
/* clcpy by GBlansten */

void clcpy(EMP* e, int rh, int ot)
{ e->grspy=(e->rt*rh)+(e->rt*ot*1.5);
e->txamt=e->grspy*e->txrt;
e->ntpy=e->grspy-e->txamt;
updacctdata(e);
if (e->dd==false) cutpyck(e);
else prtstb(e); }
```

As you can see, this function is quite a mess. Sure, it works, but no programmer in the world would like to maintain this type of code. Consider what the function would look like if you were to apply some of the naming conventions used in this chapter (such as using underscores and eliminating short cryptic names), use some bracing techniques (such as Allman's technique), and add some white space and comments:

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

Function Name: calc_pay
Parameters:    emp       - EMPLOYEE pointer that points to employee data
               reg_hours - The number of regular hours (<= 40) employee
                           has worked
               ot_hours  - The number of overtime hours (> 40) employee
                           has worked
Author:        Gern Blansten
Date Written:  13 dec 1993
Modifications: 04 sep 1994 by Lloyd E. Work
               - Rewrote function to make it readable by human beings.

Description:   This function calculates an employee's gross pay, tax
               amount, and net pay, and either prints a paycheck for the
               employee or (in the case of those who have direct deposit)
               prints a paycheck stub.

**********************************************************************/

void calc_pay(EMPLOYEE* emp, int reg_hours, int ot_hours)
{

     /* gross pay = (employee rate * regular hours) +
                    (employee rate * overtime hours * 1.5) */

     emp->gross_pay = (emp->rate * reg_hours) +
                      (emp->rate * ot_hours * 1.5);

     /* tax amount = gross pay * employee's tax rate */

     emp->tax_amount = emp->gross_pay * emp->tax_rate;

     /* net pay = gross pay - tax amount */

     emp->net_pay = emp->gross_pay - emp->tax_amount;

     /* update the accounting data */

     update_accounting_data(emp);

     /* check for direct deposit */

     if (emp->direct_deposit == false)

          cut_paycheck(emp);          /* print a paycheck */

     else

          print_paystub(emp);          /* print a paycheck stub */

}
```

As you can see, Lloyd's version (the one with liberal use of comments, white space, descriptive variable names, and so on) is much more readable than Gern's ill-fated version. Chances are that good 'ol Gern has been (or soon will be) replaced....

You should use white space (and comments, for that matter) as much as you see fit. Doing so will help your programs to be much more readable—and possibly lengthen your job expectancy.

Cross Reference:

XIX.3: Does the use of comments affect program speed, executable size, or efficiency?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.5: What is camel notation?

Answer:

Camel notation, as it has come to be known, involves using mixed upper- and lowercase letters to form variable and function names. For instance, here is the same function named using the camel notation method and the underscore method:

```
PrintEmployeePaychecks();

print_employee_paychecks();
```

The first version of this function uses the camel notation—each logical break in the function name is accentuated by the use of a capital letter. The second version of the function uses the underscore method—each logical break in the function name is accentuated by the use of an underscore.

Camel notation has gained in popularity over the years, and it is used quite a bit in many newer libraries and environments such as Microsoft Windows. The underscore method, on the other hand, has been around since C's first years and is very popular in older programs and environments such as UNIX.

Cross Reference:

XIX.1: Should the underscore be used in variable names?

XIX.2: Can a variable's name be used to indicate its data type?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.9: How many letters long should variable names be? What is the ANSI standard for significance?

XIX.10: What is Hungarian notation, and should I use it?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

Answer:

No. When you compile your program, each variable and function name is converted to a "symbol"—that is, a smaller, symbolic representation of the original function. So, whether you have a function named

```
PrintOutAllOfTheClientsMonthEndReports();
```

or

```
prt_rpts();
```

the results are the same. Generally, you should use descriptive function and variable names so that your programs will be more readable. Check your compiler's documentation to see how many characters of significance are allowed—most ANSI compilers allow at least 31 characters of significance. In other words, only the first 31 characters of a variable or function name are checked for their uniqueness—the rest of the characters are ignored.

A good rule of thumb is to make your function and variable names read just like the English language, as if you were reading a book. You should be able to read the function or variable name and easily recognize it and know generally what its function is.

Cross Reference:

XIX.1: Should the underscore be used in variable names?

XIX.2: Can a variable's name be used to indicate its data type?

XIX.3: Does the use of comments affect program speed, executable size, or efficiency?

XIX.4: Does the use of white space affect program speed, executable size, or efficiency?

XIX.5: What is camel notation?

XIX.9: How many letters long should variable names be? What is the ANSI standard for significance?

XIX.10: What is Hungarian notation, and should I use it?

XIX.7: What is the correct way to name a function?

Answer:

Functions should generally begin with a verb and end with a noun. This practice follows the general convention used by the English language. Here are some examples of properly named functions:

```
PrintReports();
SpawnUtilityProgram();
ExitSystem();
InitializeDisk();
```

Notice that in all of these examples, a verb is used to begin the function name, and a noun is used to complete the function name. If you were to read these in English, you might recognize these functions as

print the reports

spawn the utility program

exit the system

initialize the disk

Using the verb-noun convention (especially in English-language countries) makes your programs immediately more readable and familiar to the programmer who is reading your code.

Cross Reference:

XIX.5: What is camel notation?

XIX.8: What is the correct way to use braces?

XIX.10: What is Hungarian notation, and should I use it?

XIX.8: What is the correct way to use braces?

Answer:

In C, there is no right and wrong way to use braces—as long as you have a closing brace for every opening brace, you will not have brace problems in your programs. However, three prominent bracing styles are commonly used: Kernighan and Ritchie, Allman, and Whitesmiths. These three styles will be discussed next.

In the book *The C Programming Language*, Brian Kernighan and Dennis Ritchie introduced their style of implementing braces. The style looks like this:

```
if (argc < 3) {
     printf("Error! Not enough arguments. Correct usage is:\n");
     printf("C:>copyfile <source_file> <destination_file>\n");
     exit(1);
}
else {
     open_files();
     while (!feof(infile)) {
          read_data();
          write_data();
     }
     close_files();
}
```

Notice that with the K&R style, the opening brace is placed on the same line as the statement it is used with, and the closing brace is aligned below the statement it closes. For instance, in the preceding example, the `if` statement has its opening brace on the same line, and its closing brace is aligned below it. The same is true of the `if` statement's corresponding `else` condition and of the `while` statement that occurs later in the program.

Here is the same example, except this time the Allman brace style is used:

```
if (argc < 3)
{
     printf("Error! Not enough arguments. Correct usage is:\n");
     printf("C:>copyfile <source_file> <destination_file>\n");
     exit(1);
}
else
{
     open_files();
     while (!feof(infile))
     {
          read_data();
          write_data();
     }
     close_files();
}
```

Notice that with the Allman style, each brace is placed on its own line. Both the opening and the closing braces are aligned with the statement that is used.

Here is the same example with the Whitesmiths style of bracing:

```
if (argc < 3)
     {
     printf("Error! Not enough arguments. Correct usage is:\n");
     printf("C:>copyfile <source_file> <destination_file>\n");
     exit(1);
     }
else
     {
     open_files();
     while (!feof(infile))
          {
          read_data();
          write_data();
          }
     close_files();
     }
```

As with the Allman style, the Whitesmiths style calls for putting braces on their own lines. However, the braces are indented to be aligned with the statements the braces contain. For instance, in the preceding example, the opening brace of the `if` statement is aligned with the first `printf()` function call.

Whatever method you choose to use, *be consistent*—and you will help yourself and others read your programs more easily.

Cross Reference:

XIX.5: What is camel notation?

XIX.7: What is the correct way to name a function?

XIX.10: What is Hungarian notation, and should I use it?

XIX.9: How many letters long should variable names be? What is the ANSI standard for significance?

Answer:

Generally, your variable names should be long enough to effectively describe the variable or function you are naming. Short, cryptic names should be avoided, because they often cause problems when other programmers try to interpret your code. Instead of using a short, cryptic function name such as

```
opndatfls();
```

you should use a longer name such as

```
open_data_files();
```

or

```
OpenDataFiles();
```

The same is true of variable names. Instead of using a cryptic variable name such as

```
fmem
```

why not expand it to its full definition:

```
free_memory_available
```

Using expanded names will help make your code much easier to read and understand. Most ANSI compilers allow at least 31 characters of significance—that is, only the first 31 characters are checked for uniqueness.

A good rule of thumb is to make your function and variable names read just like the English language, as if you were reading a book. You should be able to read the function or variable name and easily recognize it and know generally what its function is.

Cross Reference:

XIX.1: Should the underscore be used in variable names?

XIX.2: Can a variable's name be used to indicate its data type?

XIX.5: What is camel notation?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.10: What is Hungarian notation, and should I use it?

XIX.10: What is Hungarian notation, and should I use it?

Answer:

Hungarian notation was originally created by Microsoft programmer Charles Simonyi (no doubt of Hungarian descent). With this notation, the names of your variables or functions are prefixed with one or two characters that represent the data type of the variable or function.

This kind of notation has many advantages. It is used extensively in environments such as Microsoft Windows. See FAQ XIX.2 for a full explanation of Hungarian notation and some example notation standards you might want to adopt.

Cross Reference:

XIX.1: Should the underscore be used in variable names?

XIX.2: Can a variable's name be used to indicate its data type?

XIX.5: What is camel notation?

XIX.6: Do longer variable names affect the speed, executable size, or efficiency of a program?

XIX.9: How many letters long should variable names be? What is the ANSI standard for significance?

XIX.11: What is iterative processing?

Answer:

Iterative processing involves executing the same programming statements repetitively, possibly breaking at a point when a condition occurs. The C language provides some built-in constructs for iterative processing, such as `while` loops, `do...while` loops, and `for` loops. With each of these, a predefined number of statements is executed repetitively while a certain condition remains true. Here is an example of iterative processing:

```
while (x < 100)
{

     y = 0;

     do {

          for(z=0; z<100; z++)
               y++;

     } while (y < 1000);

     x++;

}
```

In this example, the statements included in the `while` loop are executed 100 times. Within the `while` loop is a `do...while` loop. In the `do...while` loop is a `for` loop that is executed 10 times. Within the `for` loop, the variable `y` is incremented 100 times. Therefore, the statement

```
y++;
```

is executed 100,000 times (100 `while`s × 10 `do...while`s × 100 `for`s). `y` will not be 100,000 when the `while` loop is complete, however, because `y` is reset to 0 each 1000 iterations.

Iterative processing is used tremendously throughout C programs. Often, you will use iterative processing to read from and write to arrays and files. For example, here is a program that uses iterative processing to read in your AUTOEXEC.BAT file and print its contents on-screen:

```
#include <stdio.h>
#include <stdlib.h>

int main(void);

int main(void)
{

     FILE* autoexec_file;
     char  buffer[250];
```

```
          if ((autoexec_file = fopen("C:\\AUTOEXEC.BAT", "rt")) == NULL)
          {
               fprintf(stderr, "Cannot open AUTOEXEC.BAT file.\n");
               exit(1);
          }

          printf("Contents of AUTOEXEC.BAT file:\n\n");

          while (!feof(autoexec_file))
          {
               fgets(buffer, 200, autoexec_file);
               printf("%s", buffer);
          }

          fclose(autoexec_file);

          return(0);

     }
```

Notice that this example uses a `while` statement to repeatedly call the `fgets()` and `printf()` functions to read in lines from the AUTOEXEC.BAT file and print them to the screen. This is just one example of how iterative processing can be used.

Cross Reference:

XIX.12: What is recursion, and how do you use it?

XIX.12: What is recursion, and how do you use it?

Answer:

In C, a function that calls itself (either directly or indirectly) is said to be *recursive*. You might be wondering why on earth a function would want to call itself. Perhaps this situation is best explained by an example. One classic case of recursion coming in handy is when a number's *factorial* is being calculated. To calculate a number's factorial value, you multiply that number (`x`) by its predecessor (`x`–1) and keep going until you've reached 1. For instance, the factorial of 5 can be calculated as shown here:

```
5 * 4 * 3 * 2 * 1
```

If `x` were 5, you could transform this calculation into an equation:

```
x! = x * (x-1) * (x-2) * (x-3) * (x-4) * 1
```

To perform this calculation using C, you could write a function called `calc_factorial()` that would repeatedly call itself, each time decrementing the number being calculated, until you have reached 1. Here is an example of how you might write the `calc_factorial()` function:

```
#include <stdio.h>

void main(void);
```

```
unsigned long calc_factorial(unsigned long x);

void main(void)
{

     int x = 5;

     printf("The factorial of %d is %ld.\n", x, calc_factorial(x));

}

unsigned long calc_factorial(unsigned long x)
{

     if (!x)
          return 1L;

     return(x * calc_factorial(x-1L));

}
```

In the preceding example, the `calc_factorial()` calls itself after decrementing the value of `x`. If `x` is equal to 0, the `if` statement will evaluate to true, and `calc_factorial()` is not recursively called. Hence, when 0 is reached, the function exits for one last time, returning the value 1. It returns 1 because you can safely multiply any value by 1 and still retain its original value. If your program contained the statement

```
x = calc_factorial(5);
```

it would expand out to this:

```
x = 5 * (5-1) * (4-1) * (3-1) * (2-1) * 1;
```

Hence, `x` would evaluate to the factorial of 5, which is 120.

Recursion is a neat concept and can be a great source for experimentation, but it does not come without cost. Recursive functions tend to take longer than straightforward programming statements (that is, `while` loops), and they also consume valuable stack space. Each time a recursive function calls itself, its state needs to be saved on the stack so that the program can return to it when it is done calling itself. Invariably, recursive functions can be trouble if they are not carefully thought out.

If possible, you should avoid writing recursive functions. For instance, the previous factorial function could have been written this way:

```
#include <stdio.h>

void main(void);
unsigned long calc_factorial(unsigned long x);

void main(void)
{

     int x = 5;

     printf("The factorial of %d is %ld.\n", x, calc_factorial(x));

}
```

```
unsigned long calc_factorial(unsigned long x)
{

     unsigned long factorial;

     factorial = x;

     while (x > 1L)
     {
          factorial *= --x;
     }

     return(factorial);

}
```

This version of the `calc_factorial()` function uses a `while` loop to calculate a value's factorial. Not only is it much faster than the recursive version, but it also consumes a minimal amount of stack space.

Cross Reference:

XIX.11: What is iterative processing?

XIX.13: What is the best way to represent true and false in C?

Answer:

In C, anything that evaluates to 0 is evaluated to be false, and anything that evaluates to a nonzero value is true. Therefore, the most common definition for false is 0, and the most common definition for true is 1. Many programs include header files that define this:

```
#define FALSE  0
#define TRUE   1
```

If you are writing a Windows program, you should note that this exact definition of `TRUE` and `FALSE` appears in the windows.h header file. This form of defining true and false is very common and perfectly acceptable. There are, however, a few other ways of defining true and false. For instance, consider this definition:

```
#define FALSE  0
#define TRUE   !FALSE
```

This simply says that `FALSE` is 0 and `TRUE` is anything but 0. Note that even negative numbers, such as –1, are nonzero and therefore evaluate to true.

Another popular way to define true and false is to create your own *enumerated* type, such as Boolean (or `BOOL`), like this:

```
enum BOOL {
     FALSE,
     TRUE
};
```

As you might already know, the first element of an enumerated type is assigned the value 0 by default.

Therefore, with the preceding `enum` definition, `FALSE` is assigned 0 and `TRUE` is assigned 1. Using an enumerated type has some benefits over using the more common symbolic constant (`#define`). See FAQ V.6 and FAQ V.7 for an explanation of the benefit of using `enum`.

Which method is best? There is no single answer to this question. If you are writing a Windows program, `TRUE` and `FALSE` are already defined for you, so there is no need to create your own definition of `TRUE` and `FALSE`. Otherwise, you can choose your own way from the methods described previously.

Cross Reference:

V.6: What is the benefit of using `enum` to declare a constant?

V.7: What is the benefit of using an `enum` rather than a `#define` constant?

XIX.14: What is the difference between a null loop and an infinite loop?

Answer:

A null loop does not continue indefinitely—it has a predefined number of iterations before exiting the loop. An infinite loop, on the other hand, continues without end and never exits the loop. This is best illustrated by comparing a null loop to an infinite loop.

Here is an example of a null loop:

```
for (x=0; x<500000; x++);
```

Notice that in this example, a semicolon is placed directly after the closing parenthesis of the `for` loop. As you might already know, C does not require semicolons to follow `for` loops. Usually, only the statements within the `for` loop are appended with semicolons. Putting the semicolon directly after the `for` loop (and using *no* braces) creates a null loop—literally, a loop that contains no programming statements. In the preceding example, when the `for` loop executes, the variable `x` will be incremented 500,000 times with no processing occurring between increments.

You might be wondering what null loops are used for. Most often, they are used for putting a pause in your program. The preceding example will make your program “pause” for however long it takes your computer to count to 500,000. However, there are many more uses for null loops. Consider the next example:

```
while (!kbhit());
```

This example uses a null loop to wait for a key to be pressed on the keyboard. This can be useful when your program needs to display a message such as `Press Any Key To Continue` or something similar (let’s hope your users are smart enough to avoid an endless search for the “Any Key”!).

An infinite loop, unlike a null loop, can *never* be terminated. Here is an example of an infinite loop:

```
while (1);
```

In this example, the `while` statement contains a constant that is nonzero. Therefore, the `while` condition will always evaluate to true and will never terminate. Notice that a semicolon is appended directly to the end of the closing parenthesis, and thus the while statement contains no other programming statements. Therefore, there is no way that this loop can terminate (unless, of course, the program is terminated).

Cross Reference:

XIX.15: What is the difference between continue and break?

XIX.15: What is the difference between *continue* and *break*?

Answer:

A continue statement is used to return to the beginning of a loop. The break statement is used to exit from a loop. For example, here is a typical continue statement:

```
while (!feof(infile)
{
     fread(inbuffer, 80, 1, infile); /* read in a line from input file */
     if (!strncmpi(inbuffer, "REM", 3))  /* check if it is
                                            a comment line */
          continue;     /* it's a comment, so jump back to the while() */
     else
          parse_line();             /* not a comment--parse this line */
}
```

In this example, a file is being read and parsed. The letters "REM" (short for "remark") are used to denote a comment line in the file that is being processed. Because a comment line means nothing to the program, it is skipped. As each line is read in from the input file, the first three letters of the line are compared with the letters "REM." If there is a match, the input line contains a comment, and the continue statement is used to jump back to the while statement to continue reading in lines from the input file. Otherwise, the line must contain a valid statement, so the parse_line() function is called.

A break statement, on the other hand, is used to exit a loop. Here is an example of a break statement:

```
while (!feof(infile)
{
     fread(inbuffer, 80, 1, infile); /* read in a line from input file */
     if (!strncmpi(inbuffer, "REM", 3))  /* check if it is
                                            a comment line */
          continue;     /* it's a comment, so jump back to the while() */
     else
     {
          if (parse_line() == FATAL_ERROR)  /* attempt to parse
                                               this line */
               break;        /* fatal error occurred, so exit the loop */
     }
}
```

This example builds on the example presented for the continue statement. Notice that in this example, the return value of the parse_line() function is checked. If the parse_line() function returns the value FATAL_ERROR, the while loop is immediately exited by use of the break statement. The break statement causes the loop to be exited, and control is passed to the first statement immediately following the loop.

Cross Reference:

XIX.14: What is the difference between a null loop and an infinite loop?