Chapter 13 Advanced Program Control

13-1. Ending Loops Early

On Day 6 you learned how the for loop, the while loop, and the do...while loop can control program execution. These loop constructions execute a block of C statements never, once, or more than once, depending on conditions in the program. In all three cases, termination or exit of the loop occurs only when a certain condition occurs.

At times, however, you might want to exert more control over loop execution. The break and continue statements provide this control.

13-1-1. The break Statement

The break statement can be placed only in the body of a for loop, while loop, or do...while loop. (It's valid in a switch statement too, but that topic isn't covered until later in this chapter.) When a break statement is encountered, execution exits the loop. The following is an example:

```
for ( count = 0; count < 10; count++ )
{
   if ( count == 5 )
      break;
}
```

Left to itself, the for loop would execute 10 times. On the sixth iteration, however, count is equal to 5, and the break statement executes, causing the for loop to terminate. Execution then passes to the statement immediately following the for loop's closing brace. When a break statement is encountered inside a nested loop, it causes the program to exit the innermost loop only.

Listing 13.1 demonstrates the use of break.

Listing 13.1. Using the break statement.

```
1:  /* Demonstrates the break statement. */
2:
3:  #include <stdio.h>
4:
5:  char s[] = "This is a test string. It contains two sentences.";
6:
7:  main()
8:  {
9:     int count;
10:
11:     printf("\nOriginal string: %s", s);
12:
13:     for (count = 0; s[count]!='\0'; count++)
14:     {
15:         if (s[count] == `.')
16:         {
17:             s[count+1] = `\0';
18:             break;
19:         }
```

```
20:     }
21:     printf("\nModified string: %s\n", s);
22:
23:     return 0;
24: }
Original string: This is a test string. It contains two sentences.
Modified string: This is a test string.
```

ANALYSIS: This program extracts the first sentence from a string. It searches the string, character by character, for the first period (which should mark the end of a sentence). This is done in the for loop in lines 13 through 20. Line 13 starts the for loop, incrementing count to go from character to character in the string, s. Line 15 checks to see whether the current character in the string is equal to a period. If it is, a null character is inserted immediately after the period (line 17). This, in effect, trims the string. Once you trim the string, you no longer need to continue the loop, so a break statement (line 18) quickly terminates the loop and sends control to the first line after the loop (line 21). If no period is found, the string isn't altered.

A loop can contain multiple break statements, but only the first break executed (if any) has any effect. If no break is executed, the loop terminates normally (according to its test condition). Figure 13.1 shows the operation of the break statement.

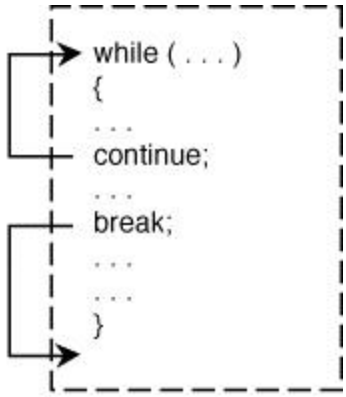

Figure 13-1. The operation of the break and continue statements.

The break Statement

```
break;
```

break is used inside a loop or switch statement. It causes the control of a program to skip past the end of the current loop (for, while, or do...while) or switch statement. No further iterations of the loop execute; the first command following the loop or switch statement executes.

Example

```
int x;
printf ( "Counting from 1 to 10\n" );
/* having no condition in the for loop will cause it to loop forever
*/
for( x = 1; ; x++ )
{
   if( x == 10 )   /* This checks for the value of 10 */
      break;      /* This ends the loop */
   printf( "\n%d", x );
}
```

13-1-2. The continue Statement

Like the break statement, the continue statement can be placed only in the body of a for loop, a while loop, or a do...while loop. When a continue statement executes, the next iteration of the enclosing loop begins immediately. The statements between the continue statement and the end of the loop aren't executed. The operation of continue is also shown in Figure 13.1. Notice how this differs from the operation of a break statement.

Listing 13.2 uses the continue statement. This program accepts a line of input from the keyboard and then displays it with all lowercase vowels removed.

Listing 13.2. Using the continue statement.

```
1:  /* Demonstrates the continue statement. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:     /* Declare a buffer for input and a counter variable. */
8:
9:     char buffer[81];
10:     int ctr;
11:
12:     /* Input a line of text. */
13:
14:     puts("Enter a line of text:");
15:     gets(buffer);
16:
17:     /* Go through the string, displaying only those */
18:     /* characters that are not lowercase vowels. */
19:
20:     for (ctr = 0; buffer[ctr] !='\0'; ctr++)
21:     {
22:
23:         /* If the character is a lowercase vowel, loop back */
24:         /* without displaying it. */
25:
26:         if (buffer[ctr] == `a' || buffer[ctr] == `e'
27:            || buffer[ctr] == `i' || buffer[ctr] == `o'
28:            || buffer[ctr] == `u')
29:               continue;
30:
31:         /* If not a vowel, display it. */
32:
33:         putchar(buffer[ctr]);
34:     }
35:     return 0;
36: }
Enter a line of text:
This is a line of text
Ths s  ln f txt
```

ANALYSIS: Although this isn't the most practical program, it does use a continue statement effectively. Lines 9 and 10 declare the program's variables. buffer[] holds the string that the user enters in line 15. The other variable, ctr, increments through the elements of the array buffer[], while the for loop in lines 20 through 34 searches for vowels. For each letter in the loop, an if statement in lines 26 through 28 checks the letter against lowercase vowels. If there is a match, a continue statement executes, sending control back to line 20, the for statement. If the letter isn't a vowel, control passes to the if statement, and line 33 is executed. Line 33 contains a new library function, putchar(), which displays a single character on-screen.

The continue Statement

```
continue;
```

continue is used inside a loop. It causes the control of a program to skip the rest of the current iteration of a loop and start the next iteration.

Example

```
int x;
printf("Printing only the even numbers from 1 to 10\n");
for( x = 1; x <= 10; x++ )
{
   if( x % 2 != 0 )    /* See if the number is NOT even */
      continue;      /* Get next instance x */
   printf( "\n%d", x );
}
```

13-2. The goto Statement

The goto statement is one of C's *unconditional jump,* or *branching,* statements. When program execution reaches a goto statement, execution immediately jumps, or branches, to the location specified by the goto statement. This statement is unconditional because execution always branches when a goto statement is encountered; the branch doesn't depend on any program conditions (unlike if statements, for example).

A goto statement and its target must be in the same function, but they can be in different blocks. Take a look at Listing 13.3, a simple program that uses a goto statement.

Listing 13.3. Using the goto statement.

```
1: /* Demonstrates the goto statement */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int n;
8:
9: start: ;
10:
11:     puts("Enter a number between 0 and 10: ");
12:     scanf("%d", &n);
13:
14:     if (n < 0 ||n > 10 )
15:         goto start;
16:     else if (n == 0)
17:         goto location0;
18:     else if (n == 1)
19:         goto location1;
20:     else
21:         goto location2;
22:
23: location0: ;
24:     puts("You entered 0.\n");
```

```
25:     goto end;
26:
27: location1: ;
28:     puts("You entered 1.\n");
29:     goto end;
30:
31: location2: ;
32:     puts("You entered something between 2 and 10.\n");
33:
34: end: ;
35:     return 0;
36: }
Enter a number between 0 and 10:
1
You entered 1.
Enter a number between 0 and 10:
9
You entered something between 2 and 10.
```

ANALYSIS: This is a simple program that accepts a number between 0 and 10. If the number isn't between 0 and 10, the program uses a goto statement on line 15 to go to start, which is on line 9. Otherwise, the program checks on line 16 to see whether the number equals 0. If it does, a goto statement on line 17 sends control to location0 (line 23), which prints a statement on line 24 and executes another goto. The goto on line 25 sends control to end at the end of the program. The program executes the same logic for the value of 1 and all values between 2 and 10 as a whole.

The target of a goto statement can come either before or after that statement in the code. The only restriction, as mentioned earlier, is that both the goto and the target must be in the same function. They can be in different blocks, however. You can use goto to transfer execution both into and out of loops, such as a for statement, but you should never do this. In fact, I strongly recommend that you never use the goto statement anywhere in your programs. There are two reasons:

- You don't need it. No programming task requires the goto statement. You can always write the needed code using C's other branching statements.
- It's dangerous. The goto statement might seem like an ideal solution for certain programming problems, but it's easy to abuse. When program execution branches with a goto statement, no record is kept of where the execution came from, so execution can weave through the program willy-nilly. This type of programming is known as *spaghetti code*.

Some careful programmers can write perfectly fine programs that use goto. There might be situations in which a judicious use of goto is the simplest solution to a programming problem. It's never the only solution, however. If you're going to ignore this warning, at least be careful!

DO avoid using the goto statement if possible.

DON'T confuse break and continue. break ends a loop, whereas continue starts the next iteration.

The goto Statement

```
goto location;
```

location is a label statement that identifies the program location where execution is to branch. A *label statement* consists of an identifier followed by a colon and a C statement:

```
location: a C statement;
```

If you want the label by itself on a line, you can follow it with the null statement (a semicolon by itself):

```
location: ;
```

13-3. Infinite Loops

What is an infinite loop, and why would you want one in your program? An infinite loop is one that, if left to its own devices, would run forever. It can be a for loop, a while loop, or a do...while loop. For example, if you write

```
while (1)
{
    /* additional code goes here */
}
```

you create an infinite loop. The condition that the while tests is the constant 1, which is always true and can't be changed by the program. Because 1 can never be changed on its own, the loop never terminates.

In the preceding section, you saw that the break statement can be used to exit a loop. Without the break statement, an infinite loop would be useless. With break, you can take advantage of infinite loops.

You can also create an infinite for loop or an infinite do...while loop, as follows:

```
for (;;)
{
    /* additional code goes here */
}
do
{
    /* additional code goes here */
} while (1);
```

The principle remains the same for all three loop types. This section's examples use the while loop.

An infinite loop can be used to test many conditions and determine whether the loop should terminate. It might be difficult to include all the test conditions in parentheses after the while statement. It might be easier to test the conditions individually in the body of the loop, and then exit by executing a break as needed.

An infinite loop can also create a menu system that directs your program's operation. You might remember from Day 5, "Functions: The Basics," that a program's main() function often serves as a sort of "traffic cop," directing execution among the various functions that do the real work of the program. This is often accomplished by a menu of some kind: The user is presented with a list of choices and makes an entry by selecting one of them. One of the available choices should be to terminate the program. Once a choice is made, one of C's decision statements is used to direct program execution accordingly.

Listing 13.4 demonstrates a menu system.

Listing 13.4. Using an infinite loop to implement a menu system.

```
1:  /* Demonstrates using an infinite loop to implement */
```

```
2:  /* a menu system. */
3:  #include <stdio.h>
4:  #define DELAY  1500000       /* Used in delay loop. */
5:
6:  int menu(void);
7:  void delay(void);
8:
9:  main()
10: {
11:    int choice;
12:
13:    while (1)
14:    {
15:
16:     /* Get the user's selection. */
17;
18:     choice = menu();
19:
20:     /* Branch based on the input. */
21:
22;     if (choice == 1)
23:     {
24:         puts("\nExecuting choice 1.");
25:         delay();
26:     }
27:     else if (choice == 2)
28:          {
29:              puts("\nExecuting choice 2.");
30:              delay();
31:          }
32;     else if (choice == 3)
33:          {
34:              puts("\nExecuting choice 3.");
35:              delay();
36:          }
37:     else if (choice == 4)
38:          {
39:              puts("\nExecuting choice 4.");
40:              delay();
41:          }
42:     else if (choice == 5)       /* Exit program. */
43:          {
44:              puts("\nExiting program now...\n");
45:              delay();
46:              break;
47:          }
48:          else
49:          {
50:              puts("\nInvalid choice, try again.");
51:              delay();
52:          }
```

```
53:    }
54:    return 0;
55: }
56:
57: /* Displays a menu and inputs user's selection. */
58: int menu(void)
59: {
60:     int reply;
61:
62:     puts("\nEnter 1 for task A.");
63:     puts("Enter 2 for task B.");
64:     puts("Enter 3 for task C.");
65:     puts("Enter 4 for task D.");
66:     puts("Enter 5 to exit program.");
67:
68:     scanf("%d", &reply);
69:
70:     return reply;
71: }
72:
73: void delay( void )
74: {
75:     long x;
76:     for ( x = 0; x < DELAY; x++ )
77:         ;
78: }
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
1
Executing choice 1.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
5
Exiting program now...
```

ANALYSIS: In Listing 13.4, a function named menu() is called on line 18 and defined on lines 58 through 71. menu() displays a menu on-screen, accepts user input, and returns the input to the main program. In main(), a series of nested if statements tests the returned value and directs execution accordingly. The only thing this

program does is display messages on-screen. In a real program, the code would call various functions to perform the selected task.

This program also uses a second function named delay(). delay() is defined on lines 73 through 78 and really doesn't do much. Simply stated, the for statement on line 76 loops, doing nothing (line 77). The statement loops DELAY times. This is an effective method of pausing the program momentarily. If the delay is too short or too long, the defined value of DELAY can be adjusted accordingly.

Both Borland and Symantec offer a function similar to delay(), called sleep(). This function pauses program execution for the number of seconds that is passed as its argument. To use sleep(), a program must include the header file TIME.H if you're using the Symantec compiler. You must use DOS.H if you're using a Borland compiler. If you're using either of these compilers or a compiler that supports sleep(), you could use it instead of delay().

WARNING: There are better ways to pause the computer than what is shown in Listing 13.4. If you choose to use a function such as sleep(), as just mentioned, be cautious. The sleep() function is not ANSI-compatible. This means that it might not work with other compilers or on all platforms.

13-4. The switch Statement

C's most flexible program control statement is the switch statement, which lets your program execute different statements based on an expression that can have more than two values. Earlier control statements, such as if, were limited to evaluating an expression that could have only two values: true or false. To control program flow based on more than two values, you had to use multiple nested if statements, as shown in Listing 13.4. The switch statement makes such nesting unnecessary.

The general form of the switch statement is as follows:

```
switch (expression)
{
    case  template_1: statement(s);
    case  template_2: statement(s);
    ...
    case  template_n: statement(s);
    default: statement(s);
}
```

In this statement, *expression* is any expression that evaluates to an integer value: type long, int, or char. The switch statement evaluates *expression* and compares the value against the templates following each case label, and then one of the following happens:

- If a match is found between *expression* and one of the templates, execution is transferred to the statement that follows the case label.
- If no match is found, execution is transferred to the statement following the optional default label.
- If no match is found and there is no default label, execution passes to the first statement following the switch statement's closing brace.

The switch statement is demonstrated in Listing 13.5, which displays a message based on the user's input.

Listing 13.5. Using the switch statement.

```
1:  /* Demonstrates the switch statement. */
```

```
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      int reply;
8:
9:      puts("Enter a number between 1 and 5:");
10:     scanf("%d", &reply);
11:
12:     switch (reply)
13:     {
14:         case 1:
15:             puts("You entered 1.");
16:         case 2:
17:             puts("You entered 2.");
18:         case 3:
19:             puts("You entered 3.");
20:         case 4:
21:             puts("You entered 4.");
22:         case 5:
23:             puts("You entered 5.");
24:         default:
25:             puts("Out of range, try again.");
26:     }
27:
28:     return 0;
29: }
Enter a number between 1 and 5:
2
You entered 2.
You entered 3.
You entered 4.
You entered 5.
Out of range, try again.
```

ANALYSIS: Well, that's certainly not right, is it? It looks as though the switch statement finds the first matching template and then executes everything that follows (not just the statements associated with the template). That's exactly what does happen, though. That's how switch is supposed to work. In effect, it performs a goto to the matching template. To ensure that only the statements associated with the matching template are executed, include a break statement where needed. Listing 13.6 shows the program rewritten with break statements. Now it functions properly.

Listing 13.6. Correct use of switch, including break statements as needed.

```
1: /* Demonstrates the switch statement correctly. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     int reply;
```

```
8:
9:     puts("\nEnter a number between 1 and 5:");
10:    scanf("%d", &reply);
11:
12:    switch (reply)
13:    {
14:      case 0:
15:          break;
16       case 1:
17:         {
18:           puts("You entered 1.\n");
19:           break;
20:         }
21:      case 2:
22:         {
23:           puts("You entered 2.\n");
24:           break;
25:         }
26:      case 3:
27:         {
28:           puts("You entered 3.\n");
29:           break;
30:         }
31:      case 4:
32:         {
33:           puts("You entered 4.\n");
34:           break;
35:         }
36:      case 5:
37:         {
38:           puts("You entered 5.\n");
39:           break;
40:         }
41:      default:
42:         {
43:           puts("Out of range, try again.\n");
44:         }
45:    }              /* End of switch */
46:
47: }
Enter a number between 1 and 5:
1
You entered 1.
Enter a number between 1 and 5:
6
Out of range, try again.
```

Compile and run this version; it runs correctly.

One common use of the switch statement is to implement the sort of menu shown in Listing 13.4. Listing 13.7 uses switch instead of if to implement a menu. Using switch is much better than using nested if statements, which were used in the earlier version of the menu program, shown in Listing 13.4.

Listing 13.7. Using the switch statement to execute a menu system.

```
1:  /* Demonstrates using an infinite loop and the switch */
2:  /* statement to implement a menu system. */
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:
6:  #define DELAY 150000
7:
8:  int menu(void);
9:  void delay(void);
10:
11: main()
12: {
13:
14:    while (1)
15:    {
16:          /* Get user's selection and branch based on the input. */
17:
18:          switch(menu())
19:          {
20:             case 1:
21:                 {
22:                   puts("\nExecuting choice 1.");
23:                   delay();
24:                   break;
25:                 }
26:             case 2:
27:                 {
28:                   puts("\nExecuting choice 2.");
29:                   delay();
30:                   break;
31:                 }
32:             case 3:
33:                 {
34:                   puts("\nExecuting choice 3.");
35:                   delay();
36:                   break;
37:                 }
38:             case 4:
39:                 {
40:                   puts("\nExecuting choice 4.");
41:                   delay();
42:                   break;
43:                 }
44:             case 5:     /* Exit program. */
45:                 {
46:                   puts("\nExiting program now...\n");
47:                   delay();
48:                   exit(0);
49:                 }
```

```
50:             default:
51:                 {
52:                   puts("\nInvalid choice, try again.");
53:                   delay();
54:                 }
55:          }  /* End of switch */
56:     }       /* End of while  */
57:
58: }
59:
60: /* Displays a menu and inputs user's selection. */
61: int menu(void)
62: {
63:     int reply;
64:
65:     puts("\nEnter 1 for task A.");
66:     puts("Enter 2 for task B.");
67:     puts("Enter 3 for task C.");
68:     puts("Enter 4 for task D.");
69:     puts("Enter 5 to exit program.");
70:
71:     scanf("%d", &reply);
72:
73:     return reply;
74: }
75:
76: void delay( void )
77: {
78:     long x;
79:     for( x = 0; x < DELAY; x++ )
80:         ;
81: }
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
1
Executing choice 1.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
6
Invalid choice, try again.
Enter 1 for task A.
Enter 2 for task B.
Enter 3 for task C.
Enter 4 for task D.
Enter 5 to exit program.
```

```
5
Exiting program now...
```

One other new statement is in this program: the exit() library function in the statements associated with case 5: on line 48. You can't use break here, as you did in Listing 13.4. Executing a break would merely break out of the switch statement; it wouldn't break out of the infinite while loop. As you'll learn in the next section, the exit() function terminates the program.

However, having execution "fall through" parts of a switch construction can be useful at times. Say, for example, that you want the same block of statements executed if one of several values is encountered. Simply omit the break statements and list all the case templates before the statements. If the test expression matches any of the case conditions, execution will "fall through" the following case statements until it reaches the block of code you want executed. This is illustrated by Listing 13.8.

Listing 13.8. Another way to use the switch statement.

```
1:  /* Another use of the switch statement. */
2:
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:
6:  main()
7:  {
8:     int reply;
9:
10:     while (1)
11:     {
12:         puts("\nEnter a value between 1 and 10, 0 to exit: ");
13:         scanf("%d", &reply);
14:
15:         switch (reply)
16:         {
17:             case 0:
18:                 exit(0);
19:             case 1:
20:             case 2:
21:             case 3:
22:             case 4:
23:             case 5:
24:                 {
25:                   puts("You entered 5 or below.\n");
26:                   break;
27:                 }
28:             case 6:
29:             case 7:
30:             case 8:
31:             case 9:
32:             case 10:
33:                 {
34:                   puts("You entered 6 or higher.\n");
35:                   break;
36:                 }
37:             default:
```

```
38:                   puts("Between 1 and 10, please!\n");
39:          } /* end of switch */
40:      }     /*end of while */
41:
43: }
Enter a value between 1 and 10, 0 to exit:
11
Between 1 and 10, please!
Enter a value between 1 and 10, 0 to exit:
1
You entered 5 or below.
Enter a value between 1 and 10, 0 to exit:
6
You entered 6 or higher.
Enter a value between 1 and 10, 0 to exit:
0
```

ANALYSIS: [This program accepts a value from the keyboard and then states whether the value is 5 or below, 6 or higher, or not between 1 and 10. If the value is 0, line 18 executes a call to the exit() function, thus ending the program.

The switch Statement

```
switch  (expression)
{
    case  template_1: statement(s);
    case  template_2: statement(s);
    ...
    case  template_n: statement(s);
    default: statement(s);
}
```

The switch statement allows for multiple branches from a single expression. It's more efficient and easier to follow than a multileveled if statement. A switch statement evaluates an expression and then branches to the case statement that contains the template matching the expression's result. If no template matches the expression's result, control goes to the default statement. If there is no default statement, control goes to the end of the switch statement.

Program flow continues from the case statement down unless a break statement is encountered. In that case, control goes to the end of the switch statement.

Example 1

```
switch( letter )
{
   case `A':
   case `a':
       printf( "You entered A" );
       break;
   case `B':
   case `b':
       printf( "You entered B");
       break;
```

```
   ...
   ...
   default:
       printf( "I don't have a case for %c", letter );
}
```

Example 2

```
switch( number )
{
   case 0:    puts( "Your number is 0 or less.");
   case 1:    puts( "Your number is 1 or less.");
   case 2:    puts( "Your number is 2 or less.");
   case 3:    puts( "Your number is 3 or less.");
   ...
   ...
   case 99:   puts( "Your number is 99 or less.");
              break;
   default:   puts( "Your number is greater than 99.");
}
```

Because there are no break statements for the first case statements, this example finds the case that matches the number and prints every case from that point down to the break in case 99. If the number was 3, you would be told that your number is equal to 3 or less, 4 or less, 5 or less, up to 99 or less. The program continues printing until it reaches the break statement in case 99.

DON'T forget to use break statements if your switch statements need them.

DO use a default case in a switch statement, even if you think you've covered all possible cases.

DO use a switch statement instead of an if statement if more than two conditions are being evaluated for the same variable.

DO line up your case statements so that they're easy to read.

13-5. Exiting the Program

A C program normally terminates when execution reaches the closing brace of the main() function. However, you can terminate a program at any time by calling the library function exit(). You can also specify one or more functions to be automatically executed at termination.

13-5-1. The exit() Function

The exit() function terminates program execution and returns control to the operating system. This function takes a single type int argument that is passed back to the operating system to indicate the program's success or failure. The syntax of the exit() function is

```
exit(status);
```

If *status* has a value of 0, it indicates that the program terminated normally. A value of 1 indicates that the program terminated with some sort of error. The return value is usually ignored. In a DOS system, you can test the return value with a DOS batch file and the if errorlevel statement. This isn't a book about DOS, so you need to refer to your DOS documentation if you want to use a program's return value. If you're using an operating system other than DOS, you should check its documentation to determine how to use a return value from a program.

To use the exit() function, a program must include the header file STDLIB.H. This header file also defines two symbolic constants for use as arguments to the exit() function:

```
#define EXIT_SUCCESS   0
#define EXIT_FAILURE   1
```

Thus, to exit with a return value of 0, call exit(EXIT_SUCCESS); for a return value of 1, call exit(EXIT_FAILURE).

DO use the exit() command to get out of the program if there's a problem.

DO pass meaningful values to the exit() function.

13-6. Executing Operating System Commands in a Program

The C standard library includes a function, system(), that lets you execute operating system commands in a running C program. This can be useful, allowing you to read a disk's directory listing or format a disk without exiting the program. To use the system() function, a program must include the header file STDLIB.H. The format of system() is

```
system(command);
```

The argument *command* can be either a string constant or a pointer to a string. For example, to obtain a directory listing in DOS, you could write either

```
system("dir");
```

or

```
char *command = "dir";
system(command);
```

After the operating system command is executed, execution returns to the program at the location immediately following the call to system(). If the command you pass to system() isn't a valid operating system command, you get a Bad command or file name error message before returning to the program. The use of system() is illustrated in Listing 13.9.

Listing 13.9. Using the system() function to execute system commands.

```
1:  /* Demonstrates the system() function. */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:
5:  main()
```

```
6:  {
7:      /* Declare a buffer to hold input. */
8:
9:      char input[40];
10:
11:     while (1)
12:     {
13:         /* Get the user's command. */
14:
15:         puts("\nInput the desired system command, blank to exit");
16:         gets(input);
17:
18:         /* Exit if a blank line was entered. */
19:
20:         if (input[0] == `\0')
21:             exit(0);
22:
23:         /* Execute the command. */
24:
25:         system(input);
26:     }
27:
28: }
Input the desired system command, blank to exit
dir *.bak
Volume in drive E is BRAD_VOL_B
Directory of E:\BOOK\LISTINGS
LIST1414 BAK      1416 05-22-97   5:18p
1 file(s)       1416 bytes
240068096 bytes free
Input the desired DOS command, blank to exit
```

NOTE: dir *.bak is a DOS command that tells the system to list all the files in the current directory that have a .BAK extension. This command also works under Microsoft Windows. For UNIX machines, you could enter ls *.bak and get similar results. If you're using System 7 or some other operating system, you'll need to enter the appropriate operating system command.

ANALYSIS: Listing 13.9 illustrates the use of system(). Using a while loop in lines 11 through 26, this program enables operating system commands. Lines 15 and 16 prompt the user to enter the operating system command. If the user presses Enter without entering a command, lines 20 and 21 call exit() to end the program. Line 25 calls system() with the command entered by the user. If you run this program on your system, you'll get different output, of course.

The commands that you pass to system() aren't limited to simple operating commands, such as listing directories or formatting disks. You can also pass the name of any executable file or batch file--and that program is executed normally. For example, if you passed the argument LIST1308, you would execute the program called LIST1308. When you exit the program, execution passes back to where the system() call was made.

The only restrictions on using system() have to do with memory. When system() is executed, the original program remains loaded in your computer's RAM, and a new copy of the operating system command processor and any program you run are loaded as well. This works only if the computer has sufficient memory. If not, you get an error message.

13-7. Summary

This chapter covered a variety of topics related to program control. You learned about the goto statement and why you should avoid using it in your programs. You saw that the break and continue statements give additional control over the execution of loops and that these statements can be used in conjunction with infinite loops to perform useful programming tasks. This chapter also explained how to use the exit() function to control program termination. Finally, you saw how to use the system() function to execute system commands from within your program.

Q&A

Q Is it better to use a switch statement or a nested loop?

A If you're checking a variable that can take on more than two values, the switch statement is almost always better. The resulting code is easier to read, too. If you're checking a true/false condition, go with an if statement.

Q Why should I avoid a goto statement?

A When you first see a goto statement, it's easy to believe that it could be useful. However, goto can cause you more problems than it fixes. A goto statement is an unstructured command that takes you to another point in a program. Many debuggers (software that helps you trace program problems) can't interrogate a goto properly. goto statements also lead to spaghetti code--code that goes all over the place.

Q Why don't all compilers have the same functions?

A In this chapter, you saw that certain C functions aren't available with all compilers or all computer systems. For example, sleep() is available with the Borland C compilers but not with the Microsoft compilers.

Although there are standards that all ANSI compilers follow, these standards don't prohibit compiler manufacturers from adding additional functionality. They do this by creating and including new functions. Each compiler manufacturer usually adds a number of functions that they believe will be helpful to their users.

Q Isn't C supposed to be a standardized language?

A C is, in fact, highly standardized. The American National Standards Institute (ANSI) has developed the ANSI C Standard, which specifies almost all details of the C language, including the functions that are provided. Some compiler vendors have added more functions--ones that aren't part of the ANSI standard--to their C compilers in an effort to one-up the competition. In addition, you sometimes come across a compiler that doesn't claim to meet the ANSI standard. If you limit yourself to ANSI-standard compilers, however, you'll find that 99 percent of program syntax and functions are common among them.

Q Is it good to use the system() function to execute system functions?

A The system() function might appear to be an easy way to do such things as list the files in a directory, but you should be cautious. Most operating system commands are specific to a particular operating system. If you use a system() call, your code probably won't be portable. If you want to run another program (not an operating system command), you shouldn't have portability problems.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. When is it advisable to use the goto statement in your programs?

2. What's the difference between the break statement and the continue statement?

3. What is an infinite loop, and how do you create one?

4. What two events cause program execution to terminate?

5. What variable types can a switch evaluate to?

6. What does the default statement do?

7. What does the exit() function do?

8. What does the system() function do?

Exercises

1. Write a statement that causes control of the program to go to the next iteration in a loop.

2. Write the statement(s) that send control of a program to the end of a loop.

3. Write a line of code that displays a listing of all the files in the current directory (for a DOS system).

4. BUG BUSTER: Is anything wrong with the following code?

```
switch( answer )
{
    case `Y': printf("You answered yes");
              break;
    case `N': printf( "You answered no");
}
```

5. BUG BUSTER: Is anything wrong with the following code?

```
switch( choice )
{
    default:
        printf("You did not choose 1 or 2");
    case 1:
        printf("You answered 1");
        break;
    case 2:
        printf( "You answered 2");
        break;
}
```

6. Rewrite exercise 5 using if statements.

7. Write an infinite do...while loop.

Because of the multitude of possible answers for the following exercises, answers are not provided. These are exercises for you to try "on your own."

8. ON YOUR OWN: Write a program that works like a calculator. The program should allow for addition, subtraction, multiplication, and division.

9. ON YOUR OWN: Write a program that provides a menu with five different options. The fifth option should quit the program. Each of the other options should execute a system command using the system() function.