Chapter 8. More Control Statements

8.1 for Statement

The **for** statement allows the programmer to execute a block of code for a specified number of times. The general form of the **for** statement is:

```
for (initial-statement; condition; iteration-statement)
    body-statement;
```

This statement is equivalent to:

```
initial-statement;
while (condition) {
    body-statement;
    iteration-statement;
}
```

For example, Example 8 -1 uses a **while** loop to add five numbers.

Example 8-1. *total5w/total1w.c*

```
#include <stdio.h>

int total;      /* total of all the numbers */
int current;    /* current value from the user */
int counter;    /* while loop counter */

char line[80];  /* Line from keyboard */

int main() {
    total = 0;

    counter = 0;
    while (counter < 5) {
        printf("Number? ");

        fgets(line, sizeof(line), stdin);
```

```
        sscanf(line, "%d", &current);
        total += current;

        ++counter;
    }
    printf("The grand total is %d\n", total);
    return (0);
}
```

The same program can be rewritten using a **for** statement as shown in Example 8-2.

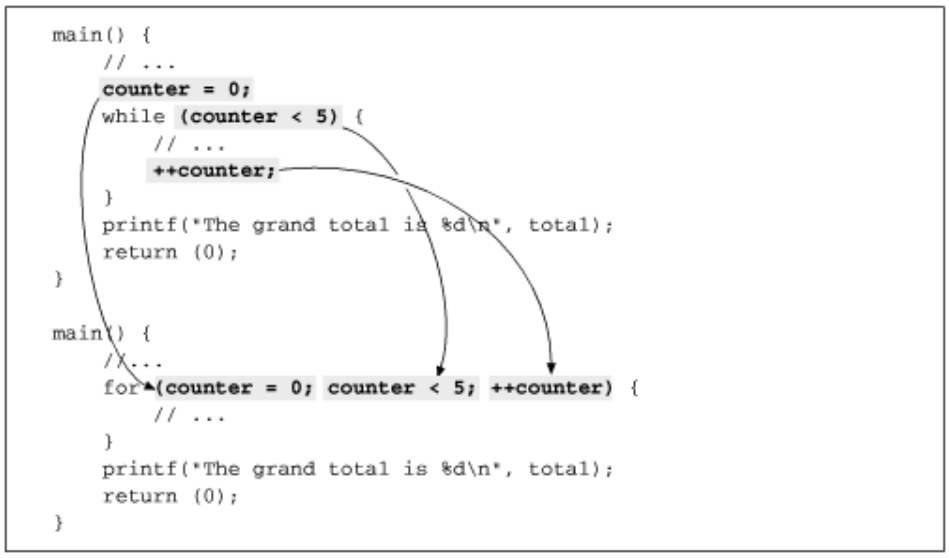
Example 8-2. *total5f/total5f.c*

```
#include <stdio.h>

int total;      /* total of all the numbers */
int current;    /* current value from the user */
int counter;    /* for loop counter */

char line[80];  /* Input from keyboard */

int main() {
    total = 0;
    for (counter = 0; counter < 5; ++counter) {
        printf("Number? ");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &current);
        total += current;
    }
    printf("The grand total is %d\n", total);
    return (0);
}
```

Note that counter goes from to 4. Ordinarily, you count five items as 1, 2, 3, 4, 5; but you will perform much better in C if you change your thinking to zero-based counting and then count five items as 0, 1, 2, 3, 4. (One-based counting is one of the main causes of array overflow errors. See Chapter 5.)

Careful examination of the two flavors of our program reveals the similarities between the two versions as seen in Figure 8-1.

Figure 8-1. Similarities between "while" and "for"



Many other programming languages do not allow you to change the control variable (in this case, counter) inside the loop. C is not so picky. You can change the control variable at any time —you can jump into and out of the loop and generally do things that would make a PASCAL or FORTRAN programmer cringe. (Although C gives you the freedom to do such insane things, that doesn't mean you should do them.)

Question 8-1: *When Example 8-3 runs, it prints*:

```
Celsius:101 Fahrenheit:213
```

and nothing more. Why? (Click here for the answer Section 8.4)

Example 8-3. *cent/cent.c*

```
#include <stdio.h>
/*
 * This program produces a Celsius to Fahrenheit conversion
 *    chart for the numbers 0 to 100.
 */

/* The current Celsius temperature we are working with */
int celsius;
int main() {
    for (celsius = 0; celsius <= 100; ++celsius);
       printf("Celsius:%d Fahrenheit:%d\n",
```

```
            celsius, (celsius * 9) / 5 + 32);
    return (0);
}
```

Question 8-2: *Example 8-4 reads a list of five numbers and counts the number of 3s and 7s in the data. Why does it give us the wrong answers?* (Click here for the answer Section 8.4)

Example 8-4. *seven/seven.c*

```
#include <stdio.h>
char line[100];     /* line of input */
int seven_count;    /* number of 7s in the data */
int data[5];        /* the data to count 3 and 7 in */
int three_count;    /* the number of 3s in the data */
int index;          /* index into the data */

int main() {

    seven_count = 0;
    three_count = 0;
    printf("Enter 5 numbers\n");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d %d %d %d",
        &data[1], &data[2], &data[3],
        &data[4], &data[5]);

    for (index = 1; index <= 5; ++index) {

        if (data[index] == 3)
            ++three_count;

        if (data[index] == 7)
            ++seven_count;
    }
    printf("Threes %d Sevens %d\n",
            three_count, seven_count);
    return (0);
}
```

When we run this program with the data 3 7 3 0 2, the results are:

```
Threes 4 Sevens 1
```

(Your results may vary.)

8.2 switch Statement

The **switch** statement is similar to a chain of **if**/**else** statements. The general form of a **switch** statement is:

```
switch  ( expression  ) {
    case  constant1 :
        statement
        . . . .
        break ;

    case  constant2 :
        statement
        . . . .
        /* Fall through */

    default:
        statement
        . . . .
        break ;

    case  constant3 :
        statement
        . . . .
        break ;
}
```

The **switch** statement evaluates the value of an expression and branches to one of the case labels. Duplicate labels are not allowed, so only one case will be selected. The expression must evaluate an integer, character, or enumeration.

The **case** labels can be in any order and must be constants. The **default** label can be put anywhere in the **switch**. No two **case** labels can have the same value.

When C sees a **switch** statement, it evaluates the expression and then looks for a matching **case** label. If none is found, the **default** label is used. If no **default** is found, the statement does nothing.

The **switch** statement is very similar to the PASCAL **case** statement. The main difference is that while PASCAL allows only one statement after the label, C

allows many. C will keep executing until it hits a **break** statement. In PASCAL, you can't fall through from one **case** to another, but in C you can.

Another difference between the C Switch and PASCAL case statements is that PASCAL requires that the **default** statement (otherwise statement) appear at the end. C allows the **default** statement to appear anywhere.

Example 8-5 contains a series of **if and else** statements:

Example 8-5. *Syntax for if and else*

```
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);
    }
```

This section of code can easily be rewritten as a **switch** statement. In this **switch**, we use a different **case** for each operation. The **default** clause takes care of all the illegal operators.

Rewriting our program using a **switch** statement makes it not only simpler, but easier to read. Our revised `calc` program is shown as Example 8-6.

Example 8-6. *calc3/calc3.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;
       switch (operator) {
       case '+':
          result += value;
          break;
       case '-':
          result -= value;
          break;
       case '*':
          result *= value;
          break;
       case '/':
          if (value == 0) {
             printf("Error:Divide by zero\n");
             printf("   operation ignored\n");
          } else
             result /= value;
          break;
       default:
          printf("Unknown operator %c\n", operator);
          break;
       }
    }
    return (0);
}
```

A **break** statement inside a **switch** tells the computer to continue execution after the **switch**. If a **break** statement is not there, execution will continue with the next statement.

For example:

```
control = 0;
/* a not so good example of programming */
switch (control) {
	case 0:
		printf("Reset\n");
	case 1:
		printf("Initializing\n");
		break;
	case 2:
		printf("Working\n");
}
```

In this case, when `control == 0`, the program will print:

```
Reset
Initializing
```

case `0` does not end with a **break** statement. After printing `Reset`, the program falls through to the next statement (`case 1`) and prints `Initializing`.

A problem exists with this syntax. You cannot determine if the program is supposed to fall through from case `0` to `case 1`, or if the programmer forgot to put in a **break** statement. In order to clear up this confusion, a **case** section should always end with a **break** statement or the comment `/* Fall through */`, as shown in the following example:

```
/* a better example of programming */
switch (control) {
	case 0:
		printf("Reset\n");
		/* Fall through */
	case 1:
		printf("Initializing\n");
		break;
	case 2:
		printf("Working\n");
}
```

Because `case 2` is last, it doesn't need a **`break`** statement. A **`break`** would cause the program to skip to the end of the **`switch`**, and we're already there.

Suppose we modify the program slightly and add another **`case`** to the **`switch`**:

```
/* We have a little problem */
switch (control) {
	case 0:
		printf("Reset\n");
		/* Fall through */
	case 1:
		printf("Initializing\n");
		break;
	case 2:
		printf("Working\n");
	case 3:
		printf("Closing down\n");
}
```

Now when `control == 2`, the program prints:

```
Working
Closing down
```

This result is an unpleasant surprise. The problem is caused by the fact that `case 2` is no longer the last **`case`**. We fall through. (Unintentionally—otherwise, we would have included a `/* Fall through */` comment.) A **`break`** is now necessary. If we always put in a **`break`** statement, we don't have to worry about whether or not it is really needed.

```
/* Almost there */
switch (control) {
	case 0:
		printf("Reset\n");
		/* Fall through */
	case 1:
		printf("Initializing\n");
		break;
	case 2:
		printf("Working\n");
		break;
}
```

Finally, we ask the question: what happens when `control == 5`? In this case, because no matching **case** or default clause exists, the entire **switch** statement is skipped.

In this example, the programmer did not include a **default** statement because `control` will never be anything but 0, 1, or 2. However, variables can get assigned strange values, so we need a little more defensive programming, as shown in the following example:

```
/* The final version */
switch (control) {
    case 0:
       printf("Reset\n");
        /* Fall through */
    case 1:
       printf("Initializing\n");
       break;
    case 2:
       printf("Working\n");
       break;
   default:
       printf(
          "Internal error, control value (%d) impossible\n",
              control);
       break;
}
```

Although a **default** is not required, it should be put in every **switch**. Even though the **default** may be:

```
default:
       /* Do nothing */
       break;
```

it should be included. This method indicates, at the very least, that you want to ignore out-of-range data.

8.3 switch, break, and continue

The **break** statement has two uses. Used inside a **switch**, **break** causes the program to go to the end of the **switch**. Inside a **for** or **while** loop, **break** causes a loop exit. The **continue** statement is valid only inside a loop. **Continue** will cause the program to go to the top of the loop. Figure 8-2 illustrates both **continue** and **break** inside a **switch** statement.

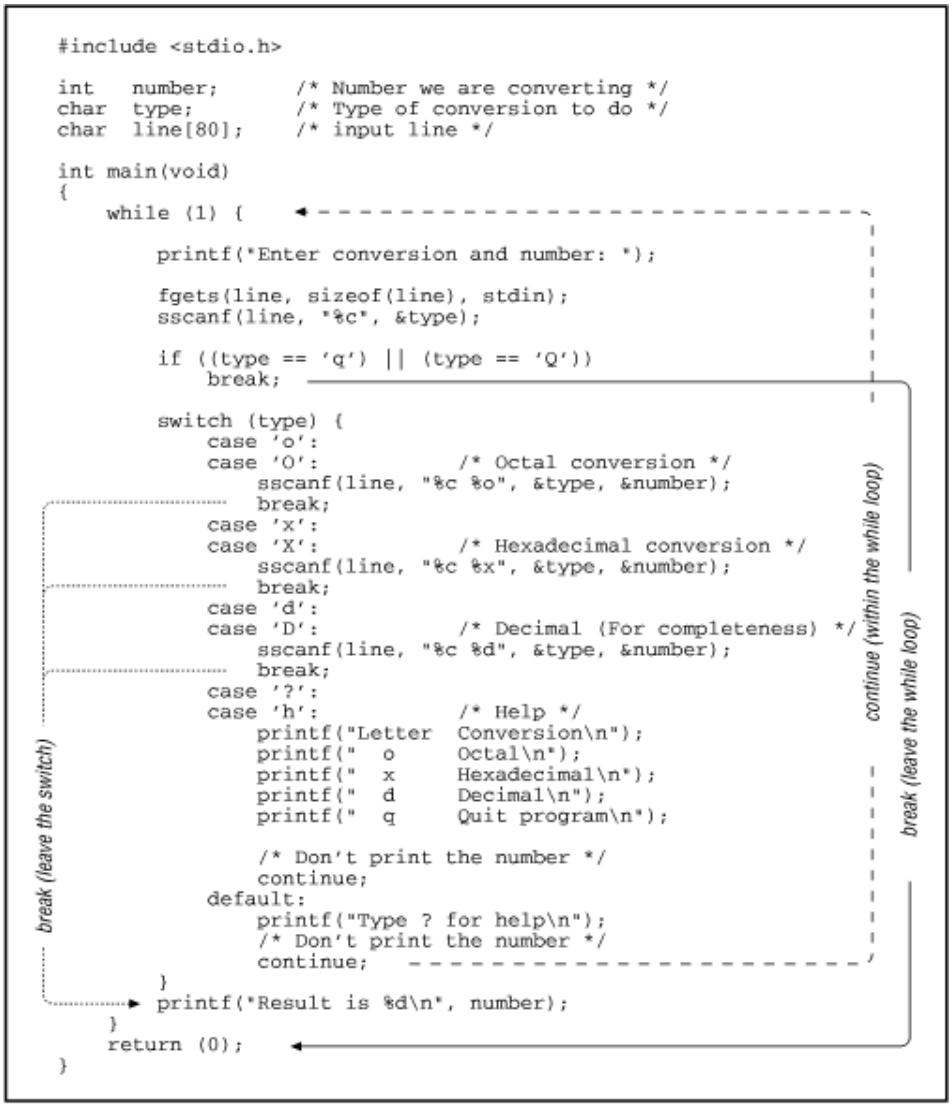
The program in Figure 8 -2 is designed to convert an integer with a number of different formats into different bases. If you want to know the value of an octal number, you would enter o (for octal) and the number. The command q is used to quit the program. For example:

```
Enter conversion and number: o 55
Result is 45
Enter conversion and number: q
```

The help command is special because we don't want to print a number after the command. After all, the result of help is a few lines of text, not a number. So a **continue** is used inside the **switch** to start the loop at the beginning. Inside the **switch**, the **continue** statement works on the loop, while the **break** statement works on the **switch**.

There is one **break** outside the **switch** that isdesigned to let the user exit the program. The control flow for this program can be seen in Figure 8 -2.

Figure 8-2. switch/continue



8.4 Answers

Answer 8-1: The problem lies with the semicolon (;) at the end of the **for** statement. The body of the **for** statement is between the closing parentheses and the semicolon. In this case, the body does not exist. Even though the `printf` statement is indented, it is not part of the **for** statement. The indentation is misleading. The C compiler does not look at indentation. The program does nothing until the expression:

```
celsius <= 100
```

becomes false (`celsius == 101`). Then the `printf` is executed.

Answer 8-2: The problem is that we read the number into `data[1]` through `data[5]`. In C, the range of legal array indices is to *array-size*-1, or in this case, to 4. `data[5]` is illegal. When we use it, strange things happen; in this case, the variable `three_count` is changed. The solution is to only use `data[0]` to `data[4]`.

So, we need to change the `sscanf` line to read:

```
sscanf(line, "%d %d %d %d %d",
    &data[0], &data[1], &data[2], &data[3], &data[4]);
```

Also, the **for** loop must be changed from:

```
for (index = 1; index <= 5; ++index)
```

to:

```
for (index = 0; index < 5; ++index)
```

Experienced C programmers could look at our broken **for** loop and immediately sense that something was wrong. Two clues that something strange is going on are 1) the **for** loop starts at 1, and 2) there is a <= operator in the loop. Most C **for** loops start at and use < for termination.

8.5 Programming Exercises

Exercise 8-1: Print a checker board (8-by-8 grid). Each square should be 5-by-3 characters wide. A 2-by-2 example follows:

```
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
|     |     |
|     |     |
|     |     |
+-----+-----+
```

Exercise 8-2: The total resistance of n resistors in parallel is:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \ldots + \frac{1}{R_n}$$

Suppose we have a network of two resistors with the values 400Ω and 200Ω.

Then our equation would be:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2}$$

Substituting in the value of the resistors we get:

$$\frac{1}{R} = \frac{1}{400} + \frac{1}{200}$$
$$\frac{1}{R} = \frac{3}{400}$$
$$R = \frac{400}{3}$$

So the total resistance of our two-resistor network is 133.3Ω.

Write a program to compute the total resistance for any number of parallel resistors.

Exercise 8-3: Write a program to average *n* numbers.

Exercise 8-4: Write a program to print out the multiplication table.

Exercise 8-5: Write a program that reads a character and prints out whether or not it is a vowel or a consonant.

Exercise 8-6: Write a program that converts numbers to words. For example, 895 results in "eight nine five."

Exercise 8-7: The number 85 is pronounced "eighty-five," not "eight five." Modify the previous program to handle the numbers through 100 so that all numbers come out as we really say them. For example, 13 would be "thirteen" and 100 would be "one hundred."