CHAPTER 4

Loops

In the last chapter you learned how to compare items and base your decisions on the result. You were able to choose how the computer reacted based on the input to a program. In this chapter, you'll learn how you can repeat a block of statements until some condition is met. This is called a **loop**.

The number of times that a loop is repeated can be controlled simply by a count—repeating the statement block a given number of times—or it can be more complex—repeating a block until some condition is met, such as the user entering **quit**, for instance. The latter would enable you to program the calculator example in the previous chapter to repeat as many times as required without having to use a `goto` statement.

In this chapter, you'll learn the following:

- How you can repeat a statement, or a block of statements, as many times as you want
- How you can repeat a statement or a block of statements until a particular condition is fulfilled
- How you use the `for`, `while`, and `do-while` loops
- What the increment and decrement operators do, and how you can use them
- How you can write a program that plays a Simple Simon game

How Loops Work

As I said, the programming mechanism that executes a series of statements repeatedly a given number of times, or until a particular condition is fulfilled, is called a **loop**. The loop is a fundamental programming tool, along with the ability to compare items. Once you can compare data values and repeat a block of statements, you can combine these capabilities to control how many times the block of statements is executed. For example, you can keep performing a particular action until two items that you are comparing are the same. Once they *are* the same, you can go on to perform a different action.

In the lottery example in Chapter 3 in Program 3.8, you could give the user exactly three guesses—in other words, you could let him continue to guess until a variable called `number_of_guesses`, for instance, equals 3. This would involve a loop to repeat the code that reads a guess from the keyboard and checks the accuracy of the value entered. Figure 4-1 illustrates the way a typical loop would work in this case.

More often than not, you'll find that you want to apply the same calculation to different sets of data values. Without loops, you would need to write out the instructions to be performed as many times as there were sets of data values to be processed, which would not be very satisfactory. A loop allows you to use the same program code for any number of sets of data to be entered.

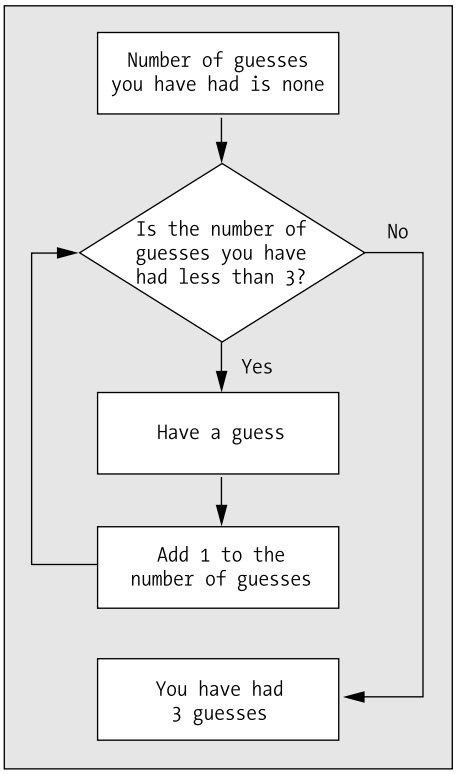

Figure 4-1. *Logic of a typical loop*

Before I discuss the various types of loops that you have available in C, I'll first introduce two new arithmetic operators that you'll encounter frequently in C programs: the **increment operator** and the **decrement operator**. These operators are often used with loops, which is why I'll discuss them here. I'll start with the briefest of introductions to the increment and decrement operators and then go straight into an example of how you can use them in the context of a loop. Once you're comfortable with how loops work, you'll return to the increment and decrement operators to investigate some of their idiosyncrasies.

Introducing the Increment and Decrement Operators

The increment operator (++) and the decrement operator (--) will increment or decrement the value stored in the integer variable that they apply to by 1. Suppose you have defined an integer variable, `number`, that currently has the value 6. You can increment it by 1 with the following statement:

```
++number;                              /* Increase the value by 1 */
```

After executing this statement, number will contain the value 7. Similarly, you could decrease the value of number by one with the following statement:

```
--number;                              /* Decrease the value by 1 */
```

These operators are different from the other arithmetic operators you have encountered. When you use any of the other arithmetic operators, you create an expression that will result in a value, which may be stored in a variable or used as part of a more complex expression. They do not directly modify the value stored in a variable. When you write the expression –number, for instance, the result of evaluating this expression is 6 if number has the value +6, but the value stored in number is unchanged. On the other hand, the expression --number *does* modify the value in number. This expression will decrement the value in number by 1, so number will end up as 5 if it was originally 6.

There's much more you'll need to know about the increment and decrement operators, but I'll defer that until later. Right now, let's get back to the main discussion and take a look at the simplest form of loop, the for loop. There are other types of loops as you'll see later, but I'll give the for loop a larger slice of time because once you understand it the others will be easy.

The for Loop

You can use the for loop in its basic form to execute a block of statements a given number of times. Let's suppose you want to display the numbers from 1 to 10. Instead of writing ten printf() statements, you could write this:

```
int count;
for(count = 1 ; count <= 10 ; ++count)
  printf("\n%d", count);
```

The for loop operation is controlled by the contents of the parentheses that follow the keyword for. This is illustrated in Figure 4-2. The action that you want to repeat each time the loop repeats is the statement immediately following the first line that contains the keyword for. Although you have just a single statement here, this could equally well be a block of statements between braces.

Figure 4-2 shows the three **control expressions** that are separated by semicolons and that control the operation of the loop.

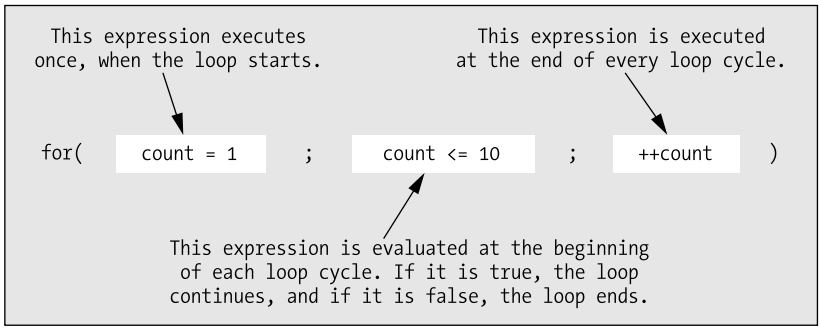

Figure 4-2. *Control expressions in a* for *loop*

The effect of each control expression is shown in Figure 4-2, but let's take a much closer look at exactly what's going on.

- The first control expression is executed only once, when the loop starts. In the example, the first expression sets a variable, `count`, to 1. This is the expression `count = 1`.
- The second control expression must be a logical expression that produces a result of `true` or `false`; in this case, it's the expression `count <= 10`. The second expression is evaluated before each loop iteration starts. If the expression evaluates to `true`, the loop continues, and if it's `false`, the loop ends and execution of the program continues with the first statement following the loop block or loop statement. Remember that `false` is a zero value, and any nonzero value is `true`. Thus, the example loop will execute the `printf()` statement as long as `count` is less than or equal to 10. The loop will end when `count` reaches 11.
- The third control expression, `++count` in this case, is executed at the end of each iteration. Here you use the increment operator to add 1 to the value of `count`. On the first iteration, `count` will be 1, so the `printf()` will output 1. On the second iteration, `count` will have been incremented to 2, so the `printf()` will output the value 2. This will continue until the value 10 has been displayed. At the start of the next iteration, `count` will be incremented to 11, and because the second control expression will then be `false`, the loop will end.

Notice the punctuation. The `for` loop control expressions are contained within parentheses, and each expression is separated from the next by a semicolon. You can omit any of the control expressions, but if you do you must still include the semicolon. For example, you could declare and initialize the variable `count` to 1 outside the loop:

```
int count = 1;
```

Now you don't need to specify the first control expression at all, and the `for` loop could look like this:

```
for( ; count <= 10 ; ++count)
  printf("\n%d", count);
```

As a trivial example, you could make this into a real program simply by adding a few lines of code:

```
/* Program 4.1 List ten integers */
#include <stdio.h>

int main(void)
{
  int count = 1;
  for( ; count <= 10 ; ++count)
    printf("\n%d", count);
  printf("\nWe have finished.\n");
  return 0;
}
```

This program will list the numbers from 1 to 10 on separate lines and then output this message:

```
We have finished.
```

The flow chart in Figure 4-3 illustrates the logic of this program.

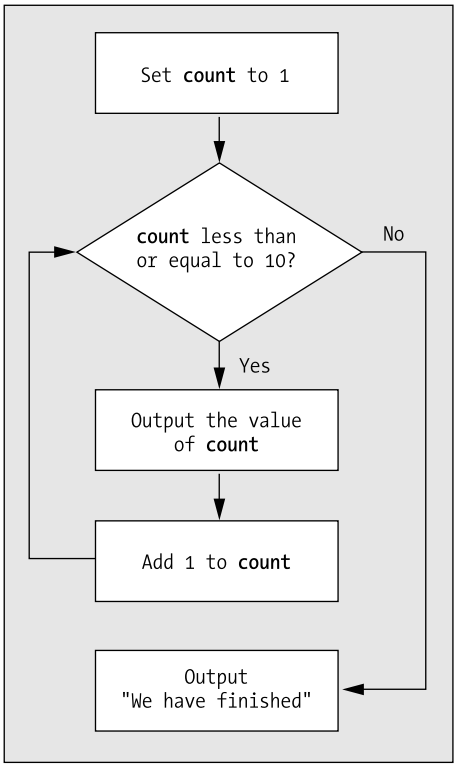

Figure 4-3. *The logic of Program 4.1*

In this example, it's easy to see what the variable `count` starts out as, so this code is quite OK. In general, though, unless the variable controlling the loop is initialized very close to the loop statement itself, it's better to initialize it in the first control expression. That way, there's less potential for error.

You can also declare the loop variable within the first loop control expression, in which case the variable is local to the loop and does not exist once the loop has finished. You could write the `main()` function like this:

```
int main(void)
{
  for(int count = 1 ; count <= 10 ; ++count)
    printf("\n%d", count);
  printf("\nWe have finished.\n");
  return 0;
}
```

Now `count` is declared within the first `for` loop expression. This means that `count` does not exist once the loop ends, so you could not output its value after the loop. When you really do need access to the loop control variable outside the loop, you just declare it in a separate statement preceding the loop, as in Program 4.1.

Let's try a slightly different example.

TRY IT OUT: DRAWING A BOX

Suppose that you want to draw a box on the screen using * characters. You could just use the `printf()` statement a lot of times, but the typing would be exhausting. You can use a `for` loop to draw a box much more easily. Let's try it:

```
/* Program 4.2 Drawing a box */
#include <stdio.h>

int main(void)
{
  printf("\n**************");         /* Draw the top of the box    */

  for(int count = 1 ; count <= 8 ; ++count)
    printf("\n*            *");       /* Draw the sides of the box  */

  printf("\n**************\n");       /* Draw the bottom of the box */
  return 0;
}
```

No prizes for guessing, but the output for this program looks like this:

```
**************
*            *
*            *
*            *
*            *
*            *
*            *
*            *
*            *
**************
```

How It Works

The program itself is really very simple. The first `printf()` statement outputs the top of the box to the screen:

```
printf("\n**************");     /* Draw the top of the box    */
```

The next statement is the `for` loop:

```
for(int count = 1 ; count <= 8 ; ++count)
  printf("\n*            *");   /* Draw the sides of the box  */
```

This repeats the `printf()` statement eight times to output the sides of the box. You probably understand this, but let's look again at how it works and pick up a bit more jargon. The loop control is the following:

```
for(int count = 1 ; count <= 8 ; ++count)
```

The operation of the loop is controlled by the three expressions that appear between the parentheses following the keyword `for`. The first expression is the following:

```
int count = 1
```

This creates and initializes the **loop control variable**, or **loop counter**, which in this case is an integer variable, `count`. You could have used other types of variables for this, but integers are convenient for the job. The next loop control expression is the following:

```
count <= 8
```

This is the **continuation condition** for the loop. This is checked *before* each loop iteration to see whether the loop should continue. If the expression is `true`, the loop continues. If it's `false`, the loop ends and execution continues with the statement following the loop. In this example, the loop continues as long as the variable `count` is less than or equal to 8. The last expression is the following:

```
++count
```

This statement increments the loop counter at the end of each loop iteration. The loop statement that outputs the sides of the box will therefore be executed eight times. After the eighth iteration, `count` will be incremented to 9 and the continuation condition will be `false`, so the loop will end.

Program execution will then continue by executing the statement that follows the loop:

```
printf("\n**************\n");   /* Draw the bottom of the box */
```

This outputs the bottom of the box on the screen.

■ **Tip** Whenever you find yourself repeating something more than a couple of times, it's worth considering a loop. They'll usually save you time and memory.

General Syntax of the for Loop

The general pattern of the `for` loop is as follows:

```
for(starting_condition; continuation_condition ; action_per_iteration)
  Statement;

Next_statement;
```

The statement to be repeated is represented by `Statement`. In general, this could equally well be a block of statements (a group of statements) enclosed between a pair of braces.

The `starting_condition` usually (but not always) sets an initial value to a loop control variable. The loop control variable is typically, but not necessarily, a counter of some kind that tracks how often the loop has been repeated. You can also declare and initialize several variables of the same type here with the declarations separated by commas; in this case all the variables will be local to the loop and will not exist once the loop ends.

The `continuation_condition` is a logical expression evaluating to `true` or `false`. This determines whether the loop should continue to be executed. As long as this condition has the value `true`, the loop continues. It typically checks the value of the loop control variable, but any logical expression can be placed here, as long as you know what you're doing.

As you've already seen, the `continuation_condition` is tested at the beginning of the loop rather than at the end. This obviously makes it possible to have a `for` loop whose statements aren't executed at all if the `continuation_condition` starts out as `false`.

The action_per_iteration is executed at the end of each loop iteration and is usually (but again not necessarily) an increment or decrement of one or more loop control variables. Where several variables are modified, you separate the expression by commas. At each iteration of the loop, the statement or block of statements immediately following the for statement is executed. The loop is terminated, and execution continues with Next_statement as soon as the continuation_condition is false.

Here's an example of a loop with two variables declared in the first loop control condition:

```
for(int i = 1, j = 2 ; i<=5 ; i++, j = j+2)
  printf("\n %5d", i*j);
```

The output produced by this fragment will be the values 2, 8, 18, 32, and 50 on separate lines.

More on the Increment and Decrement Operators

Now that you've seen an increment operator in action, let's delve a little deeper and find out what else these increment and decrement operators can do. They're both **unary operators**, which means that they're used with only one operand. You know they're used to increment (increase) or decrement (decrease) a value stored in a variable of one of the integer types by 1.

The Increment Operator

Let's start with the increment operator. It takes the form ++ and adds 1 to the variable it acts on. For example, assuming your variables are of type int, the following three statements all have exactly the same effect:

```
count = count + 1;
count += 1;
++count;
```

Each of these statement increments the variable count by 1. The last form is clearly the most concise.

Thus, if you declare a variable count and initialize it to 1

```
int count = 1;
```

and then you repeat the following statement six times in a loop

```
++count;
```

by the end of the loop, count will have a value of 7.

You can also use the increment operator in an expression. The action of this operator in an expression is to increment the value of the variable and then use the incremented value in the expression. For example, suppose count has the value 5 and you execute the statement

```
total = ++count + 6;
```

The variable count will be incremented to 6 and the variable total will be assigned the value 12, so the one statement modifies two variables. The variable count, with the value 5, has 1 added to it, making it 6, and then 6 is added to this value to produce 12 for the expression on the right side of the assignment operator. This value is stored in total.

The Prefix and Postfix Forms of the Increment Operator

Up to now you've written the operator ++ in front of the variable to which it applies. This is called the **prefix form**. The operator can also be written *after* the variable to which it applies, and this is referred to as the **postfix form**. In this case, the effect is significantly different from the prefix form when it's used in an expression. If you write `count++` in an expression, the incrementing of the variable `count` occurs *after* its value has been used. This sounds more complicated than it is.

Let's look at a variation on the earlier example:

```
total = 6 + count++;
```

With the same initial value of 5 for `count`, `total` is assigned the value 11. This is because the initial value of `count` is used to evaluate the expression on the right of the assignment (6 + 5). The variable `count` is incremented by 1 after its value has been used in the expression. The preceding statement is therefore equivalent to these two statements:

```
total = 6 + count;
++count;
```

Note, however, that when you use the increment operator in a statement by itself (as in the preceding second statement, which increments `count`), it doesn't matter whether you write the prefix or the postfix version of the operator. They both have the same effect.

Where you have an expression such as `a++ + b`—or worse, `a+++b`—it's less than obvious what is meant to happen or what the compiler will achieve. The expressions are actually the same, but in the second case you might really have meant `a + ++b`, which is different because it evaluates to one more than the other two expressions.

For example, if `a = 10` and `b = 5`, then in the statement

```
x = a++ + b;
```

`x` will have the value 15 (from 10 + 5) because `a` is incremented after the expression is evaluated. The next time you use the variable `a`, however, it will have the value 11.

On the other hand, if you execute the following statement, with the same initial values for `a` and `b`

```
y = a + (++b);
```

`y` will have the value 16 (from 10 + 6) because `b` is incremented before the statement is evaluated.

It's a good idea to use parentheses in all these cases to make sure there's no confusion. So you should write these statements as follows:

```
x = (a++) + b;
y = a + (++b);
```

The Decrement Operator

The decrement operator works in much the same way as the increment operator. It takes the form `--` and subtracts 1 from the variable it acts on. It's used in exactly the same way as ++. For example, assuming the variables are of type `int`, the following three statements all have exactly the same effect:

```
count = count - 1;
count -= 1;
--count;
```

They each decrement the variable count by 1. For example, if count has the value 10, then the statement

```
total = --count + 6;
```

results in the variable total being assigned the value 15 (from 9 + 6). The variable count, with the initial value of 10, has 1 subtracted from it so that its value is 9. Then 6 is added to the new value, making the value of the expression on the right of the assignment operator 15.

Exactly the same rules that I discussed in relation to the prefix and postfix forms of the increment operator apply to the decrement operator. For example, if count has the initial value 5, then the statement

```
total = --count + 6;
```

results in total having the value 10 (from 4 + 6) assigned, whereas

```
total = 6 + count-- ;
```

sets the value of total to 11 (from 6 + 5). Both operators are usually applied to integers, but you'll also see, in later chapters, how they can be applied to certain other data types in C.

The for Loop Revisited

Now that you understand a bit more about ++ and --, let's move on with another example that uses a loop.

TRY IT OUT: SUMMING NUMBERS

This is a more useful and interesting program than drawing a box with asterisks (unless what you really need is a box drawn with asterisks). Have you ever wanted to know what all the house numbers on your street totaled? Here you're going to read in an integer value and then use a for loop to sum all the integers from 1 to the value that was entered:

```
/*  Program 4.3   Sum the integers from 1 to a user-specified number */
#include <stdio.h>

int main(void)
{
  long sum = 0L;                  /* Stores the sum of the integers      */
  int count = 0;                  /* The number of integers to be summed */

  /* Read the number of integers to be summed */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum integers from 1 to count */
  for(int i = 1 ; i <= count ; i++)
    sum += i;

  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

The typical output you should get from this program is the following:

```
Enter the number of integers you want to sum: 10

Total of the first 10 integers is 55
```

How It Works

You start by declaring and initializing two variables that you'll need during the calculation:

```
long sum = 0L;                  /* Stores the sum of the integers      */
int count = 0;                  /* The number of integers to be summed */
```

You use `sum` to hold the final value of your calculations. You declare it as type `long` to allow the maximum total you can deal with to be as large an integer as possible. The variable `count` will store the integer that's entered as the number of integers to be summed, and you'll use this value to control the number of iterations in the `for` loop.

You deal with the input by means of the following statements:

```
printf("\nEnter the number of integers you want to sum: ");
scanf(" %d", &count);
```

After the prompt, you read in the integer that will define the sum required. If the user enters **4**, for instance, the program will compute the sum of 1, 2, 3, and 4.

The sum is calculated in the following loop:

```
for(int i = 1 ; i <= count ; i++)
  sum += i;
```

The loop variable `i` is declared and initialized to 1 by the starting condition in the `for` loop. On each iteration the value of `i` is added to `sum`, and then `i` is incremented so the values 1, 2, 3, and so on, up to the value stored in `count`, will be added to `sum`. The loop ends when the value of `i` exceeds the value of `count`.

As I've hinted by saying "not necessarily" in my descriptions of how the `for` loop is controlled, there is a lot of flexibility about what you can use as control expressions. The next program demonstrates how this flexibility might be applied to shortening the previous example slightly.

TRY IT OUT: THE FLEXIBLE FOR LOOP

This example demonstrates how you can carry out a calculation within the third control expression in a `for` loop.

```
/* Program 4.4 Summing integers - compact version */
#include <stdio.h>

int main(void)
{
  long sum = 0L;                  /* Stores the sum of the integers      */
  int count = 0;                  /* The number of integers to be summed */
```

```
  /* Read the number of integers to be summed */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum integers from 1 to count */
  for(int i = 1 ; i<= count ; sum += i++ );

  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

Typical output would be the following:

```
Enter the number of integers you want to sum: 6

Total of the first 6 numbers is 21
```

How It Works

This program will execute exactly the same as the previous program. The only difference is that you've placed the operation that accumulates the sum in the third control expression for the loop:

```
for(int i = 1 ; i<= count ; sum += i++ );
```

The loop statement is empty: it's just the semicolon after the closing parenthesis. This expression adds the value of `i` to `sum` and then increments `i` ready for the next iteration. It works this way because you've used the postfix form of the increment operator. If you use the prefix form here, you'll get the wrong answer, because the total in `sum` will include the number `count+1` from the first iteration of the loop, instead of just `count`.

Modifying the for Loop Variable

Of course, you aren't limited to incrementing the loop control variable by 1. You can change it by any value, positive or negative. You could sum the first *n* integers backward if you wish, as in the following example:

```
/* Program 4.5 Summing integers backward */
#include <stdio.h>
int main(void)
{
  long sum = 0L;                  /* Stores the sum of the integers      */
  int count = 0;                  /* The number of integers to be summed */

  /* Read the number of integers to be summed */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum integers from count to 1 */
  for(int i = count ; i >= 1 ; sum += i--);

  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

This produces the same output as the previous example. The only change is in the loop control expressions. The loop counter is initialized to `count`, rather than to 1, and it's *decremented* on each iteration. The effect is to add the values `count`, `count-1`, `count-2`, and so on, down to 1. Again, if you used the prefix form, the answer would be wrong, because you would start with adding `count-1` instead of just `count`.

Just to keep any mathematically inclined readers happy, I should mention that it's quite unnecessary to use a loop to sum the first *n* integers. The following tidy little formula for the sum of the integers from 1 to *n* will do the trick much more efficiently:

```
n*(n+1)/2
```

However, it wouldn't teach you much about loops, would it?

A for Loop with No Parameters

As I've already mentioned, you have no obligation to put any parameters in the `for` loop statement at all. The minimal `for` loop looks like this:

```
for( ;; )
  statement;
```

Here, as previously, `statement` could also be a block of statements enclosed between braces, and in this case it usually will be. Because the condition for continuing the loop is absent, as is the initial condition and the loop increment, the loop will continue indefinitely. As a result, unless you want your computer to be indefinitely doing nothing, `statement` must contain the means of exiting from the loop. To stop repeating the loop, the loop body must contain two things: a test of some kind to determine whether the condition for ending the loop has been reached, and a statement that will end the current loop iteration and continue execution with the statement following the loop.

The break Statement in a Loop

You encountered the `break` statement in the context of the `switch` statement in Chapter 3. Its effect was to stop executing the code within the `switch` block and continue with the first statement following the `switch`. The `break` statement works in essentially the same way within the body of a loop—any kind of loop. For instance

```
char answer = 0;
for( ;; )
{
  /* Code to read and process some data */
  printf("Do you want  to enter some more(y/n): ");
  scanf("%c", &answer);
  if(tolower(answer) == 'n')
    break;                              /* Go to statement after the loop */
}
/* Statement after the loop */
```

Here you have a loop that will execute indefinitely. The `scanf()` statement reads a character into `answer`, and if the character entered is **n** or **N**, the `break` statement will be executed. The effect is to stop executing the loop and to continue with the first statement following the loop. Let's see this in action in another example.

TRY IT OUT: A MINIMAL FOR LOOP

This example computes the average of an arbitrary number of values:

```
/* Program 4.6 The almost indefinite loop - computing an average */
#include <stdio.h>
#include <ctype.h>           /* For tolower() function */

int main(void)
{
  char answer = 'N';        /* Records yes or no to continue the loop */
  double total = 0.0;       /* Total of values entered                */
  double value = 0.0;       /* Value entered                          */
  int count = 0;            /* Number of values entered               */

  printf("\nThis program calculates the average of"
                                    " any number of values.");

  for( ;; )                         /* Indefinite loop */
  {
    printf("\nEnter a value: ");      /* Prompt for the next value */
    scanf(" %lf", &value);            /* Read the next value       */
    total += value;                   /* Add value to total        */
    ++count;                          /* Increment count of values */

    /* check for more input */
    printf("Do you want to enter another value? (Y or N): ");
    scanf(" %c", &answer);            /* Read response Y or N   */

    if(tolower(answer) == 'n')        /* look for any sign of no */
      break;                          /* Exit from the loop      */
  }
  /* output the average to 2 decimal places */
  printf("\nThe average is %.2lf\n", total/count );
  return 0;
}
```

Typical output from this program is the following:

```
This program calculates the average of any number of values.
Enter a value: 2.5
Do you want to enter another value? (Y or N): y

Enter a value: 3.5
Do you want to enter another value? (Y or N): y

Enter a value: 6
Do you want to enter another value? (Y or N): n

The average is 4.00
```

How It Works

The general logic of the program is illustrated in Figure 4-4.

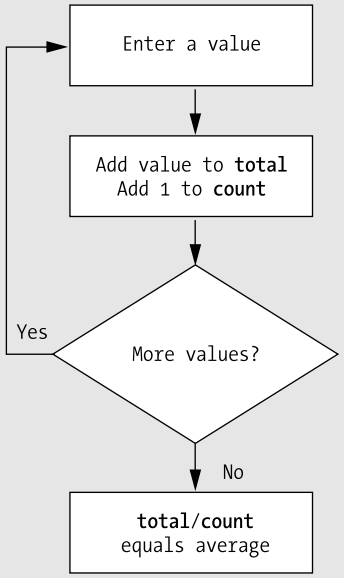

Figure 4-4. *Basic logic of the program*

You've set up the loop to continue indefinitely because the `for` loop has no end condition specified—or indeed any loop control expressions:

```
for( ;; )                          /* Indefinite loop */
```

Therefore, so far as the loop control is concerned, the block of statements enclosed between the braces will be repeated indefinitely.

You display a prompt and read an input value in the loop with these statements:

```
printf("\nEnter a value: ");      /* Prompt for the next value */
scanf(" %lf", &value);            /* Read the next value       */
```

Next, you add the value entered to your variable `total`:

```
total += value;                    /* Add value to total        */
```

You then increment the count of the number of values:

```
++count;                           /* Increment count of values */
```

Having read a value and added it to the total, you check with the user to see if more input is to be entered:

```
/* check for more input */
printf("Do you want to enter another value? (Y or N): ");
scanf(" %c", &answer);             /* Read response Y or N   */
```

This prompts for either **Y** or **N** to be entered. The character entered is checked in the `if` statement:

```
if(tolower(answer) == 'n')        /* look for any sign of no */
  break;                          /* Exit from the loop      */
```

The character stored in answer is converted to lowercase by the `tolower()` function that's declared in the `<ctype.h>` header file, so you only need to test for `n`. If you enter a character **N**, or **n**, to indicate that you've finished entering data, the `break` statement will be executed. Executing `break` within a loop has the effect of immediately ending the loop so that execution continues with the statement following the closing brace for the loop block. This is the statement:

```
printf("\nThe average is %.2lf\n", total/count);
```

This statement calculates the average of the values entered by dividing the value in `total` by the count of the number of values. The result is then displayed.

Limiting Input Using a for Loop

You can use a `for` loop to limit the amount of input from the user. Each iteration of the loop will permit some input to be entered. When the loop has completed a given number of iterations, the loop ends so no more data can be entered. You can write a simple program to demonstrate how this can work. The program will implement a guessing game.

TRY IT OUT: A GUESSING GAME

This program is going to get the user to guess the number that the program has picked as the lucky number. It uses one `for` loop and plenty of `if` statements. I've also thrown in a conditional operator, just to make sure you haven't forgotten how to use it!

```
/*  Program 4.7  A Guessing Game */
#include <stdio.h>

int main(void)
{
  int chosen = 15;                    /* The lucky number            */
  int guess = 0;                      /* Stores a guess              */
  int count = 3;                      /* The maximum number of tries */

  printf("\nThis is a guessing game.");
  printf("\nI have chosen a number between 1 and 20"
                                    " which you must guess.\n");

  for( ; count>0 ; --count)
  {
    printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");
    printf("\nEnter a guess: ");      /* Prompt for a guess  */
    scanf("%d", &guess);              /* Read in a guess     */

    /* Check for a correct guess */
    if(guess == chosen)
    {
```

```
printf("\nYou guessed it!\n");
      return 0;                       /* End the program     */
    }

    /* Check for an invalid guess */
    if(guess<1 || guess > 20)
      printf("I said between 1 and 20.\n ");
    else
      printf("Sorry. %d is wrong.\n", guess);
  }
  printf("\nYou have had three tries and failed. The number was %d\n",
                                                                  chosen);
  return 0;
}
```

Some sample output would be the following:

```
This is a guessing game.
I have chosen a number between 1 and 20 which you must guess.

You have 3 tries left.
Enter a guess: 5
Sorry. 5 is wrong.

You have 2 tries left.
Enter a guess: 18
Sorry. 18 is wrong.

You have 1 try left.
Enter a guess: 7
Sorry. 7 is wrong.

You have had three tries and failed. The number was 15
```

How It Works

You first declare and initialize three variables of type `int`, `chosen`, `guess`, and `count`:

```
int chosen = 15;                  /* The lucky number           */
int guess = 0;                    /* Stores a guess             */
int count = 3;                    /* The maximum number of tries */
```

These are to store, respectively, the number that's to be guessed, the number that's the user's guess, and the number of guesses the user is permitted. Notice that you've created a variable to store the chosen number. You could just have used the number 15 in the program, but doing it this way makes it much easier to alter the value of the number that the user must guess. It also makes it obvious what is happening in the code when you use the variable `chosen`.

You provide the user with an initial explanation of the program:

```
printf("\nThis is a guessing game.");
printf("\nI have chosen a number between 1 and 20"
                                    " which you must guess.\n");
```

The number of guesses that can be entered is controlled by this loop:

```
for( ; count>0 ; --count)
  {
    ...
  }
```

All the operational details of the game are within this loop, which will continue as long as `count` is positive, so the loop will repeat `count` times.

There's a prompt for a guess to be entered, and the guess itself is read by these statements:

```
printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");
printf("\nEnter a guess: ");      /* Prompt for a guess  */
scanf("%d", &guess);              /* Read in a guess     */
```

The first `printf()` looks a little complicated, but all it does is insert `"y"` after `"tr"` in the output when `count` is 1, and `"ies"` after `"tr"` in the output in all other cases. You must, after all, get your plurals right.

After reading a `guess` value using `scanf()`, you check whether it's correct with these statements:

```
/* Check for a correct guess */
if(guess == chosen)
{
  printf("\nYou guessed it!");
  return 0;                       /* End the program     */
}
```

If the guess is correct, you display a suitable message and execute the `return` statement. The `return` statement ends the function `main()`, and so the program ends. You'll learn more about the `return` statement when I discuss functions in greater detail in Chapter 8.

The program will reach the last check in the loop only if the guess is incorrect:

```
/* Check for an invalid guess */
if(guess<1 || guess > 20)
  printf("I said between 1 and 20.\n ");
else
  printf("Sorry. %d is wrong.\n", guess);
```

This group of statements tests whether the value entered is within the prescribed limits. If it isn't, a message is displayed reiterating the limits. If it's a valid guess, a message is displayed to the effect that it's incorrect.

The loop ends after three iterations and thus three guesses. The statement after the loop is the following:

```
printf("\nYou have had three tries and failed. The number was %d\n",
                                                              chosen);
```

This will be executed only if all three guesses were wrong. It displays an appropriate message, revealing the number to be guessed, and then the program ends.

This program is designed so that you can easily change the value of the variable `chosen` and have endless fun. Well, endless fun for a short while, anyway.

Generating Pseudo-Random Integers

The previous example would have been much more entertaining if the number to be guessed could have been generated within the program so that it was different each time the program executed. Well, you can do that using the `rand()` function that's declared in the `<stdlib.h>` header file:

```
int chosen = 0;
chosen = rand();  /* Set to a random integer */
```

Each time you call the `rand()` function, it will return a random integer. The value will be from 0 to a maximum of `RAND_MAX`, the value of which is defined in `<stdlib.h>`. The integers generated by the `rand()` function are described as **pseudo-random** numbers because truly random numbers can arise only in natural processes and can't be generated algorithmically.

The sequence of numbers that's generated by the `rand()` function uses a starting seed number, and for a given seed the sequence will always be the same. If you use the function with the default seed value, as in the previous snippet, you'll always get exactly the same sequence, which won't make the game very challenging but is useful when you are testing a program. However, C provides another standard function, `srand()`, which you can call to initialize the sequence with a particular seed that you pass as an argument to the function. This function is also declared in the `<stdlib.h>` header.

At first sight, this doesn't seem to get you much further with the guessing game, as you now need to generate a different seed each time the program executes. Yet another library function can help with this: the `time()` function that's declared in the `<time.h>` header file. The `time()` function returns the number of seconds that have elapsed since January 1, 1970, as an integer, and because time always marches on, you can get a different value returned by the `time()` function each time the program executes. The `time()` function requires an argument to be specified that you'll specify as `NULL`. `NULL` is a symbol that's defined in `<stdlib.h>`, but I'll defer further discussion of it until Chapter 7.

Thus to get a different sequence of pseudo-random numbers each time a program is run, you can use the following statements:

```
srand(time(NULL));                    /* Use clock value as starting seed */
int chosen = 0;
chosen = rand();                  /* Set to a random integer 0 to RAND_MAX */
```

The value of the upper limit, `RAND_MAX`, is likely to be quite large—often the maximum value that can be stored as type `int`. When you need a more limited range of values, you can scale the value returned by `rand()` to provide values within the range that you want. Suppose you want to obtain values in a range from 0 up to, but not including, `limit`. The simplest approach to obtaining values in this range is like this:

```
srand(time(NULL));                    /* Use clock value as starting seed     */
int limit = 20.0;                     /* Upper limit for pseudo-random values */
int chosen = 0;
chosen = rand()%limit;                /* 0 to limit-1 inclusive               */
```

Of course, if you want numbers from 1 to `limit`, you can write this:

```
chosen = 1+rand()%limit;              /* 1 to limit   inclusive               */
```

This works reasonably well with the implementation of `rand()` in my compiler and library. However, this isn't a good way in general of limiting the range of numbers produced by a pseudo-random number generator. This is because you're essentially chopping off the high-order bits in the value that's returned and implicitly assuming that the bits that are left will also represent random values. This isn't necessarily the case.

You could try using `rand()` in a variation of the previous example:

```
/*  Program 4.7A  A More Interesting Guessing Game */
#include <stdio.h>
#include <stdlib.h>               /* For rand() and srand() */
#include <time.h>                 /* For time() function    */
```

```
int main(void)
{
  int chosen = 0;                /* The lucky number                    */
  int guess = 0;                 /* Stores a guess                        */
  int count = 3;                 /* The maximum number of tries           */
  int limit = 20;                /* Upper limit for pseudo-random values */

  srand(time(NULL));             /* Use clock value as starting seed     */
  chosen = 1 + rand()%limit;     /* Random int 1 to limit                */

  printf("\nThis is a guessing game.");
  printf("\nI have chosen a number between 1 and 20"
                                    " which you must guess.\n");

  for( ; count>0 ; --count)
  {
    printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");
    printf("\nEnter a guess: ");      /* Prompt for a guess  */
    scanf("%d", &guess);              /* Read in a guess     */

    /* Check for a correct guess */
    if(guess == chosen)
    {
      printf("\nYou guessed it!\n");
      return 0;                       /* End the program */
    }

    /* Check for an invalid guess */
    if(guess<1 || guess > 20)
      printf("I said between 1 and 20.\n ");
    else
      printf("Sorry. %d is wrong.\n", guess);
  }
   printf("\nYou have had three tries and failed. The number was %ld\n",
                                                                  chosen);
  return 0;
}
```

This program should give you a different number to guess most of the time.

More for Loop Control Options

You've seen how you can increment or decrement the loop counter by 1 using the ++ and -- operators. You can increment or decrement the loop counter by any amount that you wish. Here's an example of how you can do this:

```
long sum = 0L;
for(int n = 1 ; n<20 ; n += 2)
  sum += n;
printf("Sum is %ld", sum);
```

The loop in the preceding code fragment sums all the odd integers from 1 to 20. The third control expression increments the loop variable `n` by 2 on each iteration. You can write any expression here, including any assignment. For instance, to sum every seventh integer from 1 to 1000, you could write the following loop:

```
for(int n = 1 ; n<1000 ; n = n+7)
  sum += n;
```

Now the third loop control expression increments n by 7 at the end of each iteration, so you'll get the sum 1 + 8 + 15 + 22 + and so on up to 1000.

You aren't limited to a single loop control expression. You could rewrite the loop in the first code fragment, summing the odd numbers from 1 to 20, like this:

```
for(int n = 1 ; n<20 ; sum += n, n += 2)
  ;
```

Now the third control expression consists of two expressions separated by a comma. These will execute in sequence at the end of each loop iteration. So first the expression

```
sum +=n
```

will add the current value of n to sum. Next, the second expression

```
n += 2
```

will increment n by 2. Because these expressions execute in sequence from left to right, you must write them in the sequence shown. If you reverse the sequence, the result will be incorrect.

You aren't limited to just two expressions either. You can have as many expressions here as you like, as long as they're separated by commas. Of course, you should make use of this only when there is a distinct advantage in doing so. Too much of this can make your code hard to understand.

The first and second control expressions can also consist of several expressions separated by commas, but the need for this is quite rare.

Floating-Point Loop Control Variables

The loop control variable can also be a floating-point variable. Here's a loop to sum the fractions from 1/1 to 1/10:

```
double sum = 0.0;
for(double x = 1.0 ; x<11 ; x += 1.0)
  sum += 1.0/x;
```

You'll find this sort of thing isn't required very often. It's important to remember that fractional values often don't have an exact representation in floating-point form, so it's unwise to rely on equality as the condition for ending a loop, for example

```
for(double x = 0.0 ; x != 2.0 ; x+= 0.2)      /* Indefinite loop!!! */
  printf("\nx = %.2lf",x);
```

This loop is supposed to output the values of x from 0.0 to 2.0 in steps of 0.2, so there should be 11 lines of output. Because 0.2 doesn't have an exact representation as a binary floating-point value, x never has the value 2.0, so this loop will take over your computer and run indefinitely (until you stop it; use Ctrl+C under Microsoft Windows).

The while Loop

That's enough of the for loop. Now that you've seen several examples of for loops, let's look at a different kind of loop: the while loop. With a while loop, the mechanism for repeating a set of statements allows execution to continue for as long as a specified logical expression evaluates to true. In English, I could represent this as follows:

```
While this condition is true
   Keep on doing this
```

Alternatively, here's a particular example:

```
While you are hungry
   Eat sandwiches
```

This means that you ask yourself "Am I hungry?" before eating the next sandwich. If the answer is yes, then you eat a sandwich and then ask yourself "Am I still hungry?" You keep eating sandwiches until the answer is no, at which point you go on to do something else—drink some coffee, maybe. One word of caution: enacting a loop in this way yourself is probably best done in private.

The general syntax for the `while` loop is as follows:

```
while( expression )
  Statement1;

  Statement2;
```

As always, `Statement1` could be a block of statements.

The logic of the `while` loop is shown in Figure 4-5.

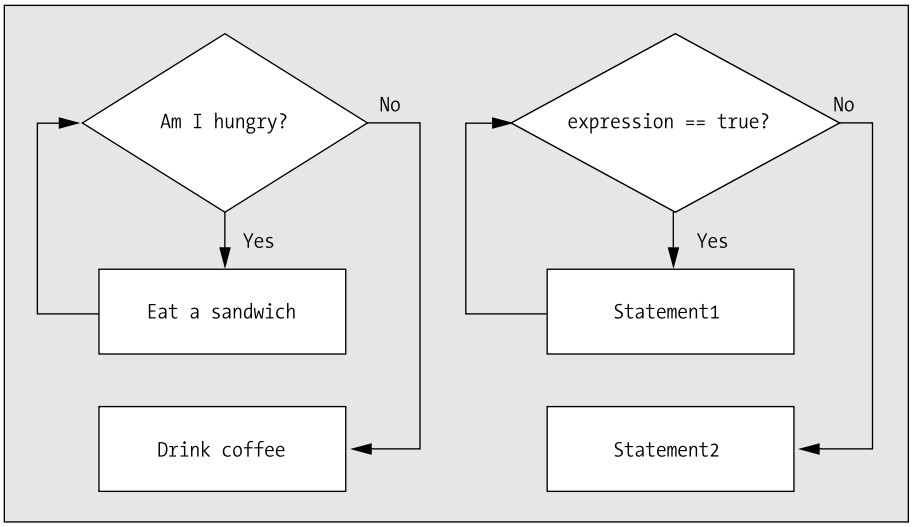

Figure 4-5. *The logic of the* `while` *loop*

Just like the `for` loop, the condition for continuation of the `while` loop is tested at the start, so if `expression` starts out false, none of the loop statements will be executed. If you answer the first question "'No, I'm not hungry," then you don't get to eat any sandwiches at all, and you move straight on to the coffee.

TRY IT OUT: USING THE WHILE LOOP

The `while` loop looks fairly straightforward, so let's go right into applying it in that old favorite, humming and summing house numbers:

```
/*  Program 4.8 While programming and summing integers */
#include <stdio.h>

int main(void)
{
  long sum = 0L;                  /* The sum of the integers            */
  int i = 1;                      /* Indexes through the integers       */
  int count = 0;                  /* The count of integers to be summed */

  /* Get the count of the number of integers to sum */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  /* Sum the integers from 1 to count */
  while(i <= count)
    sum += i++;

  printf("Total of the first %d numbers is %ld\n", count, sum);
  return 0;
}
```

Typical output from this program is the following:

```
Enter the number of integers you want to sum: 7
Total of the first 7 numbers is 28
```

How It Works

Well, really this works pretty much the same as when you used the `for` loop. The only aspect of this example worth discussing is the `while` loop:

```
while(i <= count)
  sum += i++;
```

The loop contains a single statement action that accumulates the total in `sum`. This continues to be executed with `i` values up to and including the value stored in `count`. Because you have the postfix increment operator here (the ++ comes after the variable), `i` is incremented *after* its value is used to compute `sum` on each iteration. What the statement really means is this:

```
sum += i;
i++;
```

So the value of `sum` isn't affected by the increment of `i` until the next loop iteration.
I'll try to explain this in relatively plain English, so that you understand what's really happening.