- *Entering the* `while` *loop*: When you enter the `while` loop, `i` is 1 and `count` has the value corresponding to whatever you've typed in (let's say **3**). When the loop starts, you first check whether `i <= count` is `true`. In this case, it amounts to `1<=3`, which is `true`, so you execute the loop statement:

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

- *First time through the* `while` *loop:* First, the value of `i` (which is 1) is added to the variable `sum`. The variable `sum` was equal to 0, so it's now equal to 1. Because you've used the postfix increment operator, the variable `i` is incremented after the value to be stored in `sum` has been calculated. So `i` is now 2, and you return to the beginning of the loop. You check the `while` expression and see whether the value in `i` is still less than or equal to `count`. Because `i` is now 2, which is indeed less than 3, you execute the loop statement again.
- *Second time* through *the* `while` *loop*: In the second loop iteration, you add the new value of `i` (which is now 2) to the old value of `sum` (which is 1) and store the result in `sum`. The variable `sum` now equals 3. You add 1 to `i` so `i` now has the value 3, and you go back to the beginning of the loop to check whether the control expression is still `true`.
- *Third time through the* `while` *loop*: At this point `i` is equal to `count`, so you can still continue the loop. You add the new value of `i` (which is 3) to the old value of `sum` (which is also 3) and store the result in `sum`, which now has the value 6. You add 1 to `i`, so `i` now has the value 4, and you go back to check the loop expression once more.
- *Last time through the* `while` *loop*: Now `i`, which has the value 4, is greater than `count`, which has the value 3, so the expression `i <= count` is `false`, and you leave the loop.

This example used the increment operator as postfix. How could you change the preceding program to use the prefix form of the ++ operator? Have a try and see whether you can work it out. The answer is given in the next section.

Using ++ As a Prefix Operator

The obvious bit of code that will change will be the `while` loop:

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

Try just changing this statement in Program 4.8. If you run the program now you get the wrong answer:

```
Enter the number of integers you want to sum: 3
Total of the first 3 numbers is 9
```

This is because the ++ operator is adding 1 to the value of `i` before it stores the value in `sum`. The variable `i` starts at 1 and is increased to 2 on the first iteration, whereupon that value is added to `sum`.

To make the first loop iteration work correctly, you need to start `i` off as 0. This means that the first increment would set the value of `i` to 1, which is what you want. So you must change the declaration of `i` to the following:

```
int i = 0;
```

However, the program still doesn't work properly, because it continues doing the calculation until the value in `i` is greater than `count`, so you get one more iteration than you need. The alteration you need to fix this is to change the control expression so that the loop continues while `i` is less than but not equal to `count`:

```
while(i < count)
```

Now the program will produce the correct answer. This example should help you really understand postfixing and prefixing these operators.

Nested Loops

Sometimes you may want to place one loop inside another. You might want to count the number of occupants in each house on a street. You step from house to house, and for each house you count the number of occupants. Going through all the houses could be an outer loop, and for each iteration of the outer loop you would have an inner loop that counts the occupants.

The simplest way to understand how a nested loop works is to look at a simple example.

TRY IT OUT: USING NESTED LOOPS

To demonstrate a nested loop, you'll use a simple example based on the summing integers program. Originally, you produced the sums of all the integers from 1 up to the value entered. Now for every house, you'll produce the sum of all the numbers from the first house, 1, up to the current house. If you look at the program output, it will become clearer.

```
/* Program 4.9 Sums of integers step-by-step */
#include <stdio.h>

int main(void)
{
  long sum = 0L;                      /* Stores the sum of integers      */
  int count = 0;                      /* Number of sums to be calculated */

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

  for(int i = 1 ; i <= count ; i++)
  {
    sum = 0L;                         /* Initialize sum for the inner loop */

    /* Calculate sum of integers from 1 to i */
    for(int j = 1 ; j <= i ; j++)
      sum += j;

    printf("\n%d\t%ld", i, sum);      /* Output sum of 1 to i */
  }
  return 0;
}
```

You should see some output like this:

```
Enter the number of integers you want to sum: 5

1               1
2               3
3               6
4               10
5               15
```

As you can see, if you enter **5**, the program calculates the sums of the integers from 1 to 1, from 1 to 2, from 1 to 3, from 1 to 4, and from 1 to 5.

How It Works

The program calculates the sum from 1 to each integer value, for all values from 1 up to the value of `count` that you enter. The important thing to grasp about this nested loop is that the inner loop completes all its iterations *for each iteration* of the outer loop. Thus, the outer loop sets up the value of `i` that determines how many times the inner loop will repeat:

```
for(int i = 1 ; i <= count ; i++)
{
  sum = 0L;                          /* Initialize sum for the inner loop */

  /* Calculate sum of integers from 1 to i */
  for(int j = 1 ; j <= i ; j++)
    sum += j;

  printf("\n%d\t%ld", i, sum);       /* Output sum of 1 to i */
}
```

The outer loop starts off by initializing `i` to 1, and the loop is repeated for successive values of `i` up to `count`. For each iteration of the outer loop, and therefore for each value of `i`, `sum` is initialized to 0, the inner loop is executed, and the result displayed by the `printf()` statement. The inner loop accumulates the sum of all the integers from 1 to the current value of `i`:

```
/* Calculate sum of integers from 1 to i */
for(int j = 1 ; j <= i ; j++)
  sum += j;
```

Each time the inner loop finishes, the `printf()` to output the value of `sum` is executed. Control then goes back to the beginning of the outer loop for the next iteration.

Look at the output again to see the action of the nested loop. The first loop simply sets the variable `sum` to 0 each time around, and the inner loop adds up all the numbers from 1 to the current value of `i`. You could modify the nested loop to use a `while` loop for the inner loop and to produce output that would show what the program is doing a little more explicitly.

TRY IT OUT: NESTING A WHILE LOOP WITHIN A FOR LOOP

In the previous example you nested a `for` loop inside a `for` loop. In this example you'll nest a `while` loop inside a `for` loop.

```
/* Program 4.10 Sums of integers with a while loop nested in a for loop */
#include <stdio.h>

int main(void)
{
  long sum = 1L;                 /* Stores the sum of integers      */
  int j = 1;                     /* Inner loop control variable     */
  int count = 0;                 /* Number of sums to be calculated */
```

```
  /* Prompt for, and read the input count */
  printf("\nEnter the number of integers you want to sum: ");
  scanf(" %d", &count);

  for(int i = 1 ; i <= count ; i++)
  {
    sum = 1L;                    /* Initialize sum for the inner loop */
    j=1;                         /* Initialize integer to be added    */
    printf("\n1");

    /* Calculate sum of integers from 1 to i */
    while(j < i)
    {
      sum += ++j;
      printf("+%d", j);          /* Output +j - on the same line */
    }
    printf(" = %ld\n", sum);     /* Output  = sum  */
  }
  return 0;
}
```

This program produces the following output:

```
Enter the number of integers you want to sum: 5

1 = 1

1+2 = 3

1+2+3 = 6

1+2+3+4 = 10

1+2+3+4+5 = 15
```

How It Works

The differences are inside the outer loop:

```
  for(int i = 1 ; i <= count ; i++)
  {
    sum = 1L;                    /* Initialize sum for the inner loop */
    j=1;                         /* Initialize integer to be added    */
    printf("\n1");

    /* Calculate sum of integers from 1 to i */
    while(j < i)
    {
      sum += ++j;
      printf("+%d", j);          /* Output +j - on the same line */
    }
    printf(" = %ld\n", sum);     /* Output  = sum */
  }
```

The outer loop control is exactly the same as before. The difference is what occurs during each iteration. The variable `sum` is initialized to 1 within the outer loop, because the `while` loop will add values to `sum` starting with 2. The integer to be added is stored in `j`, which is also initialized to 1. The first `printf()` in the outer loop just outputs a newline character followed by 1, the first integer in the set to be summed. The inner loop adds the integers from 2 up to the value of `i`. For each integer value in `j` that's added to `sum`, the `printf()` in the inner loop outputs `+j` on the same line as the 1 that was output first. Thus the inner loop will output +2, then +3, and so on for as long as `j` is less than `i`. Of course, for the first iteration of the outer loop, `i` is 1, so the inner loop will not execute at all, because `j<i` (1 < 1) is false from the beginning.

When the inner loop ends, the last `printf()` statement is executed. This outputs an equal sign followed by the value of `sum`. Control then returns to the beginning of the outer loop for the next iteration.

Nested Loops and the goto Statement

You've learned how you can nest one loop inside another, but it doesn't end there. You can nest as many loops one inside another as you want, for instance

```
for(int i = 0 ; i<10 ; ++i)
  for(int j = 0 ; j<20 ; ++k)          /* Loop executed 10 times            */
    for(int k = 0 ; k<30 ; ++k)        /* Loop executed 10x20 times         */
    {                                  /* Loop body executed 10x20x30 times */
       /* Do something useful */
    }
```

The inner loop controlled by `j` will execute once for each iteration of the outer loop that is controlled by `i`. The innermost loop controlled by `k` will execute once for each iteration of the loop controlled by `j`. Thus the body of the innermost loop will be executed 6,000 times.

Occasionally with deeply nested loops like this you'll want to break out of all the nested loops from the innermost loop and then continue with the statement following the outermost loop. A `break` statement in the innermost loop will only break out of that loop, and execution will continue with the loop controlled by `j`. To escape the nested loops completely using `break` statements therefore requires quite complicated logic to break out of each level until you escape the outermost loop. This is one situation in which the `goto` can be very useful because it provides a way to avoid all the complicated logic. For example

```
for(int i = 0 ; i<10 ; ++i)
  for(int j = 0 ; j<20 ; ++k)          /* Loop executed 10 times            */
    for(int k = 0 ; k<30 ; ++k)        /* Loop executed 10x20 times         */
    {                                  /* Loop body executed 10x20x30 times */
       /* Do something useful */
       if(must_escape)
         goto out;
    }
out: /*Statement following the nested loops */
```

This fragment presumes that must_escape can be altered within the innermost loop to signal that the whole nested loop should end. If the variable must_escape is true, you execute the goto statement to branch directly to the statement with the label out. So you have a direct exit from the complete nest of loops without any complicated decision-making in the outer loop levels.

The do-while Loop

The third type of loop is the do-while loop. Now you may be asking why you need this when you already have the for loop and the while loop. Well, there's actually a very subtle difference between the do-while loop and the other two. The test for whether the loop should continue is at the *end* of the loop so the loop statement or statement block always executes at least once.

The while loop tests at the beginning of the loop. So before any action takes place, you check the expression. Look at this fragment of code:

```
int number = 4;

while(number < 4)
{
  printf("\nNumber = %d", number);
  number++;
}
```

Here, you would never output anything. The control expression number < 4 is false from the start, so the loop block is never executed.

The do-while loop, however, works differently. You can see this if you replace the preceding while loop with a do-while loop and leave the rest of the statements the same:

```
int number = 4;

do
{
  printf("\nNumber = %d", number);
  number++;
}
while(number < 4);
```

Now when you execute this loop, you get number = 4 displayed. This is because the expression number < 4 is only checked at the end of the first iteration of the loop.

The general representation of the do-while loop is as follows:

```
do
  Statement;
while(expression);
```

Notice the semicolon after the while statement in a do-while loop. There isn't one in the while loop. As always, a block of statements between braces can be in place of Statement. In a do-while loop, if the value of expression is true (nonzero), the loop continues. The loop will exit only when the value of expression becomes false (zero). You can see how this works more clearly in Figure 4-6.

Here, you can see that you eat a sandwich *before* you check whether you're hungry. You'll always eat at least one sandwich so this loop is not to be used as part of a calorie-controlled diet.

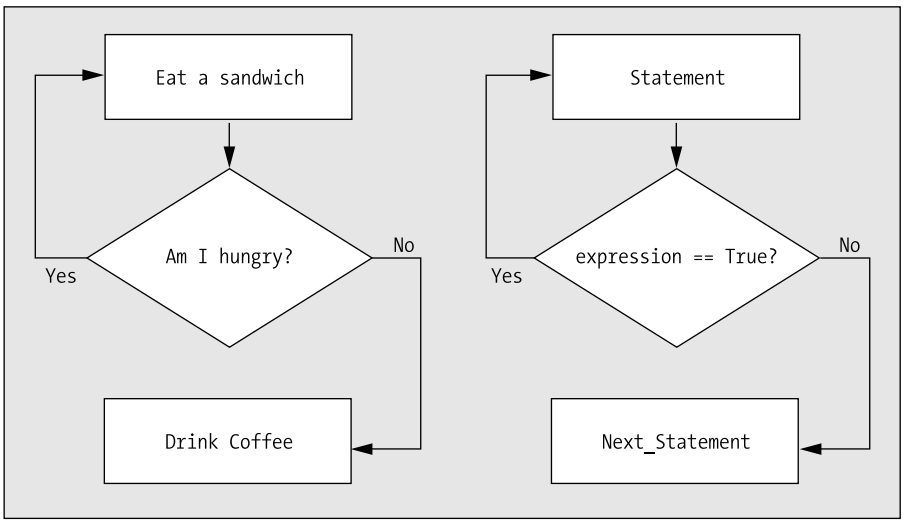

Figure 4-6. *Operation of the* `do-while` *loop*

TRY IT OUT: USING A DO-WHILE LOOP

You can try out the `do-while` loop with a little program that reverses the digits of a positive number:

```
/* Program 4.11 Reversing the digits */
#include <stdio.h>
int main(void)
{
  int number = 0;                    /* The number to be reversed */
  int rebmun = 0;                    /* The reversed number       */
  int temp = 0;                      /* Working storage           */

  /* Get the value to be reversed */
  printf("\nEnter a positive integer: ");
  scanf(" %d", &number);

  temp = number;                     /* Copy to working storage  */

  /* Reverse the number stored in temp */
  do
  {
    rebmun = 10*rebmun + temp % 10;  /* Add the rightmost digit    */
    temp = temp/10;                  /* Remove the rightmost digit */
  } while(temp);                     /* Continue while temp>0      */

  printf("\nThe number %d reversed is  %d rebmun ehT\n",
                                          number, rebmun );
  return 0;
}
```

The following is a sample of output from this program:

```
Enter a positive integer: 43

The number 43 reversed is 34 rebmun ehT
```

How It Works

The best way to explain what's going on here is to take you through a small example. Assume that the number 43 is entered by the user.

After reading the input integer and storing it in the variable number, the program copies the value in number to the variable temp:

```
temp = number;                          /* Copy to working storage  */
```

This is necessary, because the process of reversing the digits destroys the original value, and you want to output the original integer along with the reversed version.

The reversal of the digits is done in the `do-while` loop:

```
do
{
  rebmun = 10*rebmun + temp % 10;    /* Add the rightmost digit    */
  temp = temp/10;                    /* Remove the rightmost digit */
} while(temp);                       /* Continue while temp>0      */
```

The `do-while` loop is most appropriate here because any number will have at least one digit. You get the rightmost digit from the value stored in `temp` by using the modulus operator, `%`, to get the remainder after dividing by 10. Because `temp` originally contains 43, `temp%10` will be 3. You assign the value of `10*rebmun + temp%10` to `rebmun`. Initially, the value of the variable `rebmun` is 0, so on the first iteration 3 is stored in `rebmun`.

You've now stored the rightmost digit of your input in `rebmun`, and so you now remove it from `temp` by dividing `temp` by 10. Because `temp` contains 43, `temp/10` will be rounded down to 4.

At the end of the loop the `while(temp)` condition is checked, and because `temp` contains the value 4, it is true. Therefore, you go back to the top of the loop to begin another iteration.

Note Remember, any nonzero integer will convert to `true`. The Boolean value `false` corresponds to zero.

This time, the value stored in `rebmun` will be 10 times `rebmun`, which is 30, plus the remainder when `temp` is divided by 10, which is 4, so the result is that `rebmun` becomes 34. You again divide `temp` by 10, so it will contain 0. Now when you arrive at the end of the loop iteration, `temp` is 0, which is `false`, so the loop finishes and you've reversed the number. You can see how this would work with numbers with more digits. An example of the output from the program running with a longer number entered is as follows:

```
Enter a positive integer: 1234

The number 1234 reversed is 4321 rebmun ehT
```

This form of loop is used relatively rarely, compared with the other two forms. Keep it in the back of your mind, though; when you need a loop that always executes at least once, the `do-while` loop delivers the goods.

The continue Statement

Sometimes a situation will arise in which you don't want to end a loop, but you want to skip the current iteration and continue with the next. The `continue` statement in the body of a loop does this and is written simply as follows:

```
continue;
```

Of course, `continue` is a keyword, so you must not use it for other purposes. Here's an example of how the `continue` statement works:

```
for(int day = 1; day<=7 ; ++day)
{
  if(day == 3)
    continue;
  /* Do something useful with day */
}
```

This loop will execute with values of `day` from 1 to 7. When `day` has the value 3, however, the `continue` statement will execute, and the rest of the current iteration is skipped and the loop continues with the next iteration when `day` will be 4.

You'll see more examples of using `continue` later in the book.

Designing a Program

It's time to try your skills on a bigger programming problem and to apply some of what you've learned in this chapter and the previous chapters. You'll also see a few new standard library functions that you're sure to find useful.

The Problem

The problem that you're going to solve is to write a game of Simple Simon. Simple Simon is a memory-test game. The computer displays a sequence of digits on the screen for a short period of time. You then have to memorize them, and when the digits disappear from the screen, you must enter exactly the same sequence of digits. Each time you succeed, you can repeat the process to get a longer list of digits for you to try. The objective is to continue the process for as long as possible.

The Analysis

The program must generate a sequence of integers between 0 and 9 and display the sequence on the screen for one second before erasing it. The player then has to try to enter the identical sequence of digits. The sequences gradually get longer until the player gets a sequence wrong. A score is then calculated based on the number of successful tries and the time taken, and the player is asked if he would like to play again.

The logic of the program is quite straightforward. You could express it in general terms in the flow chart shown in Figure 4-7.

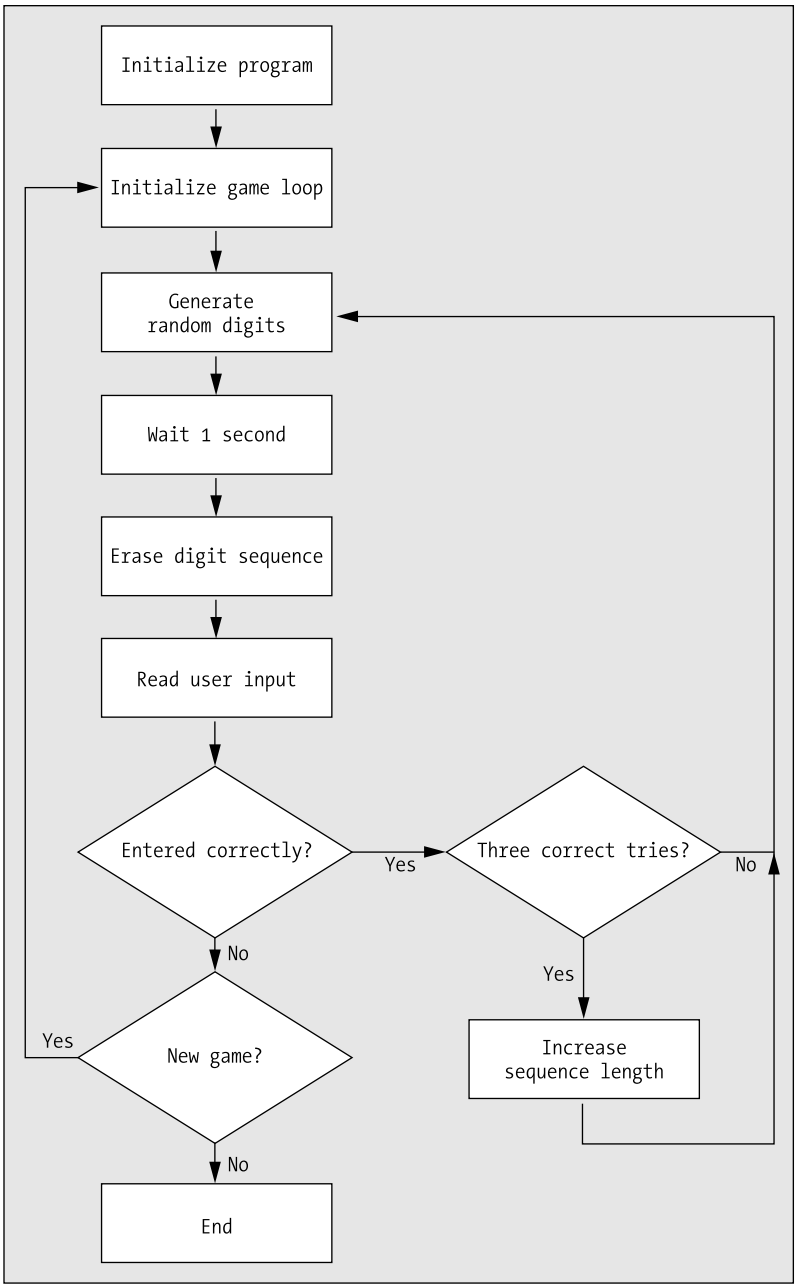

Figure 4-7. *The basic logic of the Simple Simon program*

Each box describes an action in the program, and the diamond shapes represent decisions. Let's use the flow chart as the basis for coding the program.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

You can start by putting in the main loop for a game. The player will always want to have at least one game, so the loop check should go at the end of the loop. The `do-while` loop fits the bill very nicely. The initial program code will be this:

```
/* Program 4.12 Simple Simon */
#include <stdio.h>                      /* For input and output   */
#include <ctype.h>                      /* For toupper() function */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* Rest of the declarations for the program */

  /* Describe how the game is played */
  printf("\nTo play Simple Simon, ");
  printf("watch the screen for a sequence of digits.");
  printf("\nWatch carefully, as the digits are only displayed"
                                            " for a second! ");
  printf("\nThe computer will remove them, and then prompt you ");
  printf("to enter the same sequence.");
  printf("\nWhen you do, you must put spaces between the digits. \n");
  printf("\nGood Luck!\nPress Enter to play\n");
  scanf("%c", &another_game);

  /* One outer loop iteration is one game */
  do
  {
    /* Code to play the game */

    /* Output the score when the game is finished */

    /* Check if a new game is required */
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

As long as the player enters `y` or `Y` at the end of a game, she will be able to play again. Note how you can automatically concatenate two strings in the `printf()` statement:

```
  printf("\nWatch carefully, as the digits are only displayed"
                                              " for a second! ");
```

This is a convenient way of splitting a long string over two or more lines. You just put each piece of the string between its own pair of double-quote characters, and the compiler will take care of assembling them into a single string.

Step 2

Next, you can add a declaration for another variable, called `correct`, that you'll need in the program to record whether the entry from the player is correct or not. You'll use this variable to control the loop that plays a single game:

```
/* Program 4.12 Simple Simon */
#include <stdio.h>                    /* For input and output    */
#include <ctype.h>                    /* For toupper() function  */
#include <stdbool.h>                 /* For bool, true, false  */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  bool correct = true;

  /* Rest of the declarations for the program */

  /* Describe how the game is played */
  printf("\nTo play Simple Simon, ");
  printf("watch the screen for a sequence of digits.");
  printf("\nWatch carefully, as the digits are only displayed"
                                              " for a second! ");
  printf("\nThe computer will remove them, and then prompt you ");
  printf("to enter the same sequence.");
  printf("\nWhen you do, you must put spaces between the digits. \n");
  printf("\nGood Luck!\nPress Enter to play\n");
  scanf("%c", &another_game);

  /* One outer loop iteration is one game */
  do
  {
    correct = true;          /* By default indicates correct sequence entered */

    /* Other code to initialize the game */

    /* Inner loop continues as long as sequences are entered correctly */
    while(correct)
    {
      /* Play the game */
    }

    /* Output the score when the game is finished */

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

You are using the _Bool variable, correct, here, but because you have added an #include directive for the <stdbool.h> header, you can use bool as the type name. The <stdbool.h> header also defines the symbols true and false to correspond to 1 and 0 respectively.

■ **Caution** The code will compile as it is, and you should compile it to check it out, but you should not run it yet. As you develop your own programs, you'll want to make sure that the code will at least compile along each step of the way. If you wrote all the program code in one attempt, you could end up with hundreds of errors to correct, and as you correct one problem, more may appear. This can be very frustrating. By checking out the program incrementally, you can minimize this issue, and the problems will be easier to manage. This brings us back to our current program. If you run this, your computer will be completely taken over by the program, because it contains an infinite loop. The reason for this is the inner while loop. The condition for this loop is always true because the loop doesn't do anything to change the value of correct. However, you'll be adding that bit of the program shortly.

Step 3

Now you have a slightly more difficult task to do: generating the sequence of random digits. There are two problems to be tackled here. The first is to generate the sequence of random digits. The second is to check the player's input against the computer-generated sequence.

The main difficulty with generating the sequence of digits is that the numbers have to be random. You've already seen that you have a standard function, rand(), available that returns a random integer each time you call it. You can get a random digit by just getting the remainder after dividing by 10, by using the % operator.

To ensure that you get a different sequence each time the program is executed, you'll also need to call srand() to initialize the sequence with the value returned by the time() function in the way you've already seen. Both the rand() and srand() functions require that you include the <tdlib.h> header file into the program, and the time() function requires an #include directive for <time.h>.

Now let's think about this a bit more. You'll need the sequence of random digits twice: once to display it initially before you erase it, and the second time to check against the player's input. You could consider saving the sequence of digits as an integer value of type unsigned long long. The problem with this is that the sequence could get very long if the player is good, and it could exceed the upper limit for integer values of type unsigned long long. There is another possible approach. The rand() function can produce the same sequence of numbers twice. All you need to do is to start the sequence each time with the same seed by calling srand(). This means that you won't need to store the sequence of numbers. You can just generate the same sequence twice.

Now let's add some more code to the program, which will generate the sequence of random digits and check them against what the player enters:

```
/* Program 4.12 Simple Simon */
#include <stdio.h>                    /* For input and output    */
#include <ctype.h>                    /* For toupper() function  */
#include <stdbool.h>                  /* For bool, true, false   */
#include <stdlib.h>                   /* For rand() and srand() */
#include <time.h>                     /* For time() function    */

int main(void)
{
   /* Records if another game is to be played */
   char another_game = 'Y';
```

```
/* true if correct sequence entered, false otherwise */
bool correct = true;

/* Number of sequences entered successfully */
int counter = 0;

int sequence_length = 0;       /* Number of digits in a sequence        */
time_t seed = 0;               /* Seed value for random number sequence */
int number = 0;                /* Stores an input digit                 */

/* Rest of the declarations for the program */

/* Describe how the game is played */
printf("\nTo play Simple Simon, ");
printf("watch the screen for a sequence of digits.");
printf("\nWatch carefully, as the digits are only displayed"
                                          " for a second! ");
printf("\nThe computer will remove them, and then prompt you ");
printf("to enter the same sequence.");
printf("\nWhen you do, you must put spaces between the digits. \n");
printf("\nGood Luck!\nPress Enter to play\n");
scanf("%c", &another_game);

/* One outer loop iteration is one game */
do
{
  correct = true;           /* By default indicates correct sequence entered  */
  counter = 0;              /* Initialize count of number of successful tries */
  sequence_length = 2;      /* Initial length of a digit sequence             */

  /* Other code to initialize the game */

  /* Inner loop continues as long as sequences are entered correctly */
  while(correct)
  {
    /* On every third successful try, increase the sequence length  */
    sequence_length += counter++%3 == 0;

    /* Set seed to be the number of seconds since Jan 1,1970        */
    seed = time(NULL);

    /* Generate a sequence of numbers and display the number        */
    srand((unsigned int)seed);    /* Initialize the random sequence */
    for(int i = 1; i <= sequence_length; i++)
      printf("%d ", rand() % 10); /* Output a random digit */

    /* Wait one second */

    /* Now overwrite the digit sequence */

    /* Prompt for the input sequence */
```

```
      /* Check the input sequence of digits against the original  */
      srand((unsigned int)seed);      /* Restart the random sequence  */
      for(int i = 1; i <= sequence_length; i++)
      {
        scanf("%d", &number);         /* Read an input number         */
        if(number != rand() % 10)     /* Compare against random digit */
        {
          correct = false;            /* Incorrect entry              */
          break;                      /* No need to check further...  */
        }
      }
      printf("%s\n", correct ? "Correct!" : "Wrong!");
    }

    /* Output the score when the game is finished */

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

You've declared five new variables that you need to implement the `while` loop that will continue to execute as long as the player is successful. Each iteration of this loop displays a sequence that the player must repeat. The `counter` variable records the number of times that the player is successful, and `sequence_length` records the current length of a sequence of digits. Although you initialize these variables when you declare them, you must also initialize their values in the `do-while` loop to ensure that the initial conditions are set for every game. You declare a variable, `seed`, of type `long`, that you'll pass as an argument to `srand()` to initialize the random number sequence returned by the function `rand()`. The value for `seed` is obtained in the `while` loop by calling the standard library function `time()`.

At the beginning of the `while` loop, you can see that you increase the value stored in `sequence_length` by adding the value of the expression `counter++%3 == 0` to it. This expression will be 1 when the value of counter is a multiple of 3, and 0 otherwise. This will increment the sequence length by 1 after every third successful try. The expression also increments the value in `counter` by 1 after the expression has been evaluated.

There are some other things to notice about the code. The first is the conversion of `seed` from type `time_t` to type `unsigned int` when you pass it to the `srand()` function. This is because `srand()` requires that you use type `unsigned int`, but the `time()` function returns a value of type time_t. Because the number of seconds since January 1, 1970, is in excess of 800,000,000, you need a 4-byte variable to store it. Second, you obtain a digit between 0 and 9 by taking the remainder when you divide the random integer returned by `rand()` by 10. This isn't the best way of obtaining random digits in the range 0 to 9, but it's a very easy way and is adequate for our purposes. Although numbers generated by `srand()` are randomly distributed, the low order decimal digit for the numbers in the sequence are not necessarily random. To get random digits we should be dividing the whole range of values produced by `srand()` into ten segments and associating one of the digits 0 to 9 with each segment. The digit corresponding to a given pseudo-random number can then be selected based on the segment in which the number lies.

The sequence of digits is displayed by the `for` loop. The loop just outputs the low-order decimal digit of the value returned by `rand()`. You then have some comments indicating the other code that you still have to add that will delay the program for one second and then erase the sequence from the screen. This is followed by the code to check the sequence that was entered by the player. This reinitializes the random number–generating process by calling `srand()` with the same seed value that was used at the outset. Each digit entered is compared with the low-order digit of the value returned by the function `rand()`. If there's a discrepancy, correct is set to `false` so the `while` loop will end.

Of course, if you try to run this code as it is, the sequence won't be erased, so it isn't usable yet. The next step is to add the code to complete the `while` loop.

Step 4

You must now erase the sequence, after a delay of one second. How can you get the program to wait? One way is to use another standard library function. The library function `clock()` returns the time since the program started, in units of clock ticks. The `<time.h>` header file defines a symbol `CLOCKS_PER_SEC` that's the number of clock ticks in one second. All you have to do is wait until the value returned by the function `clock()` has increased by `CLOCKS_PER_SEC`, whereupon one second will have passed. You can do this by storing the value returned by the function `clock()` and then checking, in a loop, when the value returned by `clock()` is `CLOCKS_PER_SEC` more than the value that you saved. With a variable `now` to store the current time, the code for the loop would be as follows:

```
for( ;clock() - now < CLOCKS_PER_SEC; );         /* Wait one second */
```

You also need to decide how you can erase the sequence of computer-generated digits. This is actually quite easy. You can move to the beginning of the line by outputting the escape character `'\r'`, which is a carriage return. All you then need to do is output a sufficient number of spaces to overwrite the sequence of digits. Let's fill out the code you need in the `while` loop:

```
/* Program 4.12 Simple Simon */
#include <stdio.h>                      /* For input and output   */
#include <ctype.h>                      /* For toupper() function */
#include <stdbool.h>                    /* For bool, true, false  */
#include <stdlib.h>                     /* For rand() and srand() */
#include <time.h>                       /* For time() and clock() */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  int correct = true;

  /* Number of sequences entered successfully       */
  int counter = 0;

  int sequence_length = 0;       /* Number of digits in a sequence        */
  time_t seed = 0;               /* Seed value for random number sequence */
  int number = 0;                /* Stores an input digit                 */
```

```
/* Stores current time - seed for random values */
time_t now = 0;
/* Rest of the declarations for the program */

/* Describe how the game is played */
printf("\nTo play Simple Simon, ");
printf("watch the screen for a sequence of digits.");
printf("\nWatch carefully, as the digits are only displayed"
                                          " for a second! ");
printf("\nThe computer will remove them, and then prompt you ");
printf("to enter the same sequence.");
printf("\nWhen you do, you must put spaces between the digits. \n");
printf("\nGood Luck!\nPress Enter to play\n");
scanf("%c", &another_game);

/* One outer loop iteration is one game */
do
{
  correct = true;           /* By default indicates correct sequence entered  */
  counter = 0;              /* Initialize count of number of successful tries */
  sequence_length = 2;      /* Initial length of a digit sequence             */

  /* Other code to initialize the game */

  /* Inner loop continues as long as sequences are entered correctly */
  while(correct)
  {
    /* On every third successful try, increase the sequence length */
    sequence_length += counter++%3 == 0;

    /* Set seed to be the number of seconds since Jan 1,1970       */
    seed = time(NULL);

    now = clock();         /* record start time for sequence */

    /* Generate a sequence of numbers and display the number */
    srand((unsigned int)seed);      /* Initialize the random sequence */
    for(int i = 1; i <= sequence_length; i++)
      printf("%d ", rand() % 10);   /* Output a random digit          */

    /* Wait one second */
    for( ;clock() - now < CLOCKS_PER_SEC; );

    /* Now overwrite the digit sequence */
    printf("\r");          /* go to beginning of the line */
    for(int i = 1; i <= sequence_length; i++)
      printf("  ");        /* Output two spaces           */

    if(counter == 1)       /* Only output message for the first try */
      printf("\nNow you enter the sequence  - don't forget"
                                           " the spaces\n");
    else
      printf("\r");        /* Back to the beginning of the line     */
```

```
      /* Check the input sequence of digits against the original  */
      srand((unsigned int)seed);      /* Restart the random sequence  */
      for(int i = 1; i <= sequence_length; i++)
      {
        scanf("%d", &number);         /* Read an input number         */
        if(number != rand() % 10)     /* Compare against random digit */
        {
          correct = false;            /* Incorrect entry              */
          break;                      /* No need to check further...  */
        }
      }
      printf("%s\n", correct ? "Correct!" : "Wrong!");
    }

    /* Output the score when the game is finished */

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);
  } while(toupper(another_game) == 'Y');
  return 0;
}
```

You record the time returned by `clock()` before you output the sequence. The `for` loop that's executed when the sequence has been displayed continues until the value returned by `clock()` exceeds the time recorded in `now` by `CLOCKS_PER_SEC`, which of course will be one second.

Because you haven't written a newline character to the screen at any point when you displayed the sequence, you're still on the same line when you complete the output of the sequence. You can move the cursor back to the start of the line by executing a carriage return without a linefeed, and outputting `"\r"` does just that. You then output two spaces for each digit that was displayed, thus overwriting each of them with blanks. Immediately following that, you have a prompt for the player to enter the sequence that was displayed. You output this message only the first time around; otherwise it gets rather tedious. On the second and subsequent tries, you just back up to the beginning of the now blank line, ready for the user's input.

Step 5

All that remains is to generate a score to display, once the player has gotten a sequence wrong. You'll use the number of sequences completed and the number of seconds it took to complete them to calculate this score. You can arbitrarily assign 100 points to each digit correctly entered and divide this by the number of seconds the game took. This means the faster the player is, the higher the score, and the more sequences the player enters correctly, the higher the score.

Actually, there's also one more problem with this program that you need to address. If one of the numbers typed by the player is wrong, the loop exits and the player is asked if he wants to play again. However, if the digit in error isn't the last digit, you could end up with the next digit entered as the answer to the question "Do you want to play again (y/n)?" because these digits will still be in the keyboard buffer. What you need to do is remove any information that's still in the keyboard buffer. So there are two problems: first, how to address the keyboard buffer, and second, how to clean out the buffer.

■ **Note** The **keyboard buffer** is memory that's used to store input from the keyboard. The `scanf()` function looks in the keyboard buffer for input rather than getting it directly from the keyboard itself.

With standard input and output—that is, from the keyboard and to the screen—there are actually two buffers: one for input and one for output. The standard input and output streams are called `stdin` and `stdout` respectively. So to identify the input buffer for the keyboard you just use the name `stdin`. Now that you know how to identify the buffer, how do you remove the information in it? Well, there's a standard library function, `fflush()`, for clearing out buffers. Although this function tends to be used for files, which I'll cover later in the book, it will actually work for any buffer at all. You simply tell the function which stream buffer you want cleared out by passing the name of the stream as the argument. So to clean out the contents of the input buffer, you simply use this statement:

```
fflush(stdin);                          /* Flush the stdin buffer */
```

Here's the complete program, which includes calculating the scores and flushing the input buffer:

```
/* Program 4.12 Simple Simon */
#include <stdio.h>                      /* For input and output   */
#include <ctype.h>                      /* For toupper() function */
#include <stdbool.h>                    /* For bool, true, false  */
#include <stdlib.h>                     /* For rand() and srand() */
#include <time.h>                       /* For time() and clock() */

int main(void)
{
  /* Records if another game is to be played */
  char another_game = 'Y';

  /* true if correct sequence entered, false otherwise */
  int correct = false;

  /* Number of sequences entered successfully          */
  int counter = 0;

  int sequence_length = 0;     /* Number of digits in a sequence        */
  time_t seed = 0;             /* Seed value for random number sequence */
  int number = 0;              /* Stores an input digit                 */

  time_t now = 0;            /* Stores current time - seed for random values  */
  int time_taken = 0;        /* Time taken for game in seconds                */

  /* Describe how the game is played */
  printf("\nTo play Simple Simon, ");
  printf("watch the screen for a sequence of digits.");
  printf("\nWatch carefully, as the digits are only displayed"
                                          " for a second! ");
  printf("\nThe computer will remove them, and then prompt you ");
  printf("to enter the same sequence.");
  printf("\nWhen you do, you must put spaces between the digits. \n");
  printf("\nGood Luck!\nPress Enter to play\n");
  scanf("%c", &another_game);
```

```
/* One outer loop iteration is one game */
do
{
  correct = true;         /* By default indicates correct sequence entered */
  counter = 0;            /* Initialize count of number of successful tries*/
  sequence_length = 2;    /* Initial length of a digit sequence            */
  time_taken = clock();  /* Record current time at start of game       */

  /* Inner loop continues as long as sequences are entered correctly */
  while(correct)
  {
    /* On every third successful try, increase the sequence length */
    sequence_length += counter++%3 == 0;

    /* Set seed to be the number of seconds since Jan 1,1970  */
    seed = time(NULL);

    now = clock();                  /* record start time for sequence  */

    /* Generate a sequence of numbers and display the number */
    srand((unsigned int)seed);       /* Initialize the random sequence */
    for(int i = 1; i <= sequence_length; i++)
      printf("%d ", rand() % 10);    /* Output a random digit          */

    /* Wait one second */
    for( ;clock() - now < CLOCKS_PER_SEC; );

    /* Now overwrite the digit sequence */
    printf("\r");                   /* go to beginning of the line */
    for(int i = 1; i <= sequence_length; i++)
      printf("  ");                 /* Output two spaces */

    if(counter == 1)           /* Only output message for the first try */
      printf("\nNow you enter the sequence  - don't forget"
                                          " the spaces\n");
    else
      printf("\r");                /* Back to the beginning of the line */

    /* Check the input sequence of digits against the original */
    srand((unsigned int)seed);     /* Restart the random sequence    */
    for(int i = 1; i <= sequence_length; i++)
    {
      scanf("%d", &number);         /* Read an input number         */
      if(number != rand() % 10)     /* Compare against random digit */
      {
        correct = false;            /* Incorrect entry             */
        break;                      /* No need to check further... */
      }
    }
    printf("%s\n", correct? "Correct!" : "Wrong!");
  }
```

```
    /* Calculate total time to play the game in seconds)*/
    time_taken = (clock() - time_taken) / CLOCKS_PER_SEC;

    /* Output the game score */
    printf("\n\n Your score is %d", --counter * 100 / time_taken);

    fflush(stdin);

    /* Check if new game required*/
    printf("\nDo you want to play again (y/n)? ");
    scanf("%c", &another_game);

  } while(toupper(another_game) == 'Y');
  return 0;
}
```

The declaration required for the function `fflush()` is in the `<stdio.h>` header file for which you already have an `#include` directive. Now you just need to see what happens when you actually play:

```
To play Simple Simon, watch the screen for a sequence of digits.
Watch carefully, as the digits are only displayed for a second!
The computer will remove them, and then prompt you to enter the same sequence.
When you do, you must put spaces between the digits.

Good Luck!
Press Enter to play


Now you enter the sequence  - don't forget the spaces
2 1 4
Correct!
8 7 1
Correct!
4 1 6
Correct!
7 9 6 6
Correct!
7 5 4 6
Wrong!


 Your score is 16
Do you want to play again (y/n)? n
```

Summary

In this chapter, I covered all you need to know about repeating actions using loops. With the powerful set of programming tools you've learned up to now, you should be able to create quite complex programs of your own. You have three different loops you can use to repeatedly execute a block of statements:

- The `for` loop, which you typically use for counting loops where the value of a control variable is incremented or decremented by a given amount on each iteration until some final value is reached.
- The `while` loop, which you use when the loop continues as long as a given condition is true. If the loop condition is false at the outset, the loop block will not be executed at all.
- The `do-while` loop, which works like the `while` loop except that the loop condition is checked at the end of the loop block. Consequently the loop block is always executed at least once.

In keeping with this chapter topic, I'll now reiterate some of the rules and recommendations I've presented in the book so far:

- Before you start programming, work out the logic of the process and computations you want to perform, and write it down—preferably in the form of a flow chart. Try to think of lateral approaches to a problem; there may be a better way than the obvious approach.
- Understand operator precedence in order to get complex expressions right. Whenever you are not sure about operator precedence, use parentheses to ensure expressions do what you want. Use parentheses to make complex expressions more readily understood.
- Comment your programs to explain all aspects of their operation and use. Assume the comments are for the benefit of someone else reading your program with a view to extend or modify it. Explain the purpose of each variable as you declare it.
- Program with readability foremost in your mind.
- In complicated logical expressions, avoid using the operator `!` as much as you can.
- Use indentation to visually indicate the structure of your program.

Prepared with this advice, you can now move on to the next chapter—after you've completed all the exercises, of course!

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Downloads section of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 4-1. Write a program that will generate a multiplication table of a size entered by the user. A table of size 4, for instance, would have four rows and four columns. The rows and columns would be labeled from 1 to 4. Each cell in the table will contain the product of the corresponding row and column numbers, so the value in the position corresponding to the third row and the fourth column would contain 12.

Exercise 4-2. Write a program that will output the printable characters for character code values from 0 to 127. Output each character code along with its symbol with two characters to a line. Make sure the columns are aligned. (Hint: You can use the `isgraph()` function that's declared in `ctype.h` to determine when a character is printable.)

Exercise 4-3. Extend the previous program to output the appropriate name, such as "newline," "space," "tab," and so on, for each whitespace character.

Exercise 4-4. Use nested loops to output a box bounded by asterisks as in Program 4.2, but with a width and height that's entered by the user. For example, a box ten characters wide and seven characters high would display as follows:

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

Exercise 4-5. Modify the guessing game implemented in Program 4.7 so that the program will continue with an option to play another game when the player fails to guess the number correctly and will allow as many games as the player requires.