# Program control and data structures

Before we cover the for statement, let's take a short detour and learn the basics of arrays. The for statement and arrays are closely linked in C, so it's difficult to define one without explaining the other. To help you understand the arrays used in the for statement examples to come, a quick treatment of arrays follows.

An array is an indexed group of data storage locations that have the same name and are distinguished from each other by a subscript, or index--a number following the variable name, enclosed in brackets. (This will become clearer as you continue.) Like other C variables, arrays must be declared. An array declaration includes both the data type and the size of the array (the number of elements in the array). For example, the following statement declares an array named data that is type int and has 1,000 elements:

```
int data[1000];
```

The individual elements are referred to by subscript as data[0] through data[999]. The first element is data[0], not data[1]. In other languages, such as BASIC, the first element of an array is 1; this is not true in C.

Each element of this array is equivalent to a normal integer variable and can be used the same way. The subscript of an array can be another C variable, as in this example:

```
int data[1000];
int count;
count = 100;
data[count] = 12;     /* The same as data[100] = 12 */
```

This has been a quick introduction to arrays. However, you should now be able to understand how arrays are used in the program examples later in this chapter. If every detail of arrays isn't clear to you, don't worry. You will learn more about arrays on Day 8.

DON'T declare arrays with subscripts larger than you will need; it wastes memory.

DON'T forget that in C, arrays are referenced starting with subscript 0, not 1.
Controlling Program Execution

The default order of execution in a C program is top-down. Execution starts at the beginning of the main() function and progresses, statement by statement, until the end of main() is reached. However, this order is rarely encountered in real C programs. The C language includes a variety of program control statements that let you control the order of program execution. You have already learned how to use C's fundamental decision operator, the if statement, so let's explore three additional control statements you will find useful.

## The for Statement

The for statement is a C programming construct that executes a block of one or more statements a certain number of times. It is sometimes called the for loop because program execution typically loops through the statement more than once. You've seen a few for statements used in programming examples earlier in this book. Now you're ready to see how the for statement works.

A for statement has the following structure:

```
for ( initial; condition; increment )
   statement;
```

initial, condition, and increment are all C expressions, and statement is a single or compound C statement. When a for statement is encountered during program execution, the following events occur:

1. The expression initial is evaluated. initial is usually an assignment statement that sets a variable to a particular value.

2. The expression condition is evaluated. condition is typically a relational expression.

3. If condition evaluates to false (that is, as zero), the for statement terminates, and execution passes to the first statement following statement.

4. If condition evaluates to true (that is, as nonzero), the C statement(s) in statement are executed.

5. The expression increment is evaluated, and execution returns to step 2.

Figure 6.1 shows the operation of a for statement. Note that statement never executes if condition is false the first time it's evaluated.

Figure 6.1. A schematic representation of a for statement.

Here is a simple example. Listing 6.1 uses a for statement to print the numbers 1 through 20. You can see that the resulting code is much more compact than it would be if a separate printf() statement were used for each of the 20 values.

Listing 6.1. A simple for statement.

```
1:   /* Demonstrates a simple for statement */
2:
3:   #include <stdio.h>
4:
5:   int count;
6:
7:   main()
8:   {
9:       /* Print the numbers 1 through 20 */
10:
11:      for (count = 1; count <= 20; count++)
12:          printf("%d\n", count);
13:
14:      return 0;
15:  }
```

Figure 6.2. How the for loop in Listing 6.1 operates.

ANALYSIS: Line 3 includes the standard input/output header file. Line 5 declares a type int variable, named count, that will be used in the for loop. Lines 11 and 12 are the for loop. When the for statement is reached, the initial statement is executed first. In this listing, the initial statement is count = 1. This initializes count so that it can be used by the rest of the loop. The second step in executing this for statement is the evaluation of the condition count <= 20. Because count was just initialized to 1, you know that it is less than 20, so the statement in the for command, the printf(), is executed. After executing the printing function, the increment expression, count++, is evaluated. This adds 1 to count, making it 2. Now the program loops back and checks the condition again. If it is true, the printf() reexecutes, the increment adds to count (making it 3), and the condition is checked. This loop continues until the condition evaluates to false, at which point the program exits the loop and continues to the next line (line 14), which returns 0 before ending the program.

The for statement is frequently used, as in the previous example, to "count up," incrementing a counter from one value to another. You also can use it to "count down," decrementing (rather than incrementing) the counter variable.

```
for (count = 100; count > 0; count--)
```

You can also "count by" a value other than 1, as in this example:

```
for (count = 0; count < 1000; count += 5)
```

The for statement is quite flexible. For example, you can omit the initialization expression if the test variable has been initialized previously in your program. (You still must use the semicolon separator as shown, however.)

```
count = 1;
for ( ; count < 1000; count++)
```

The initialization expression doesn't need to be an actual initialization; it can be any valid C expression. Whatever it is, it is executed once when the for statement is first reached. For example, the following prints the statement Now sorting the array...:

```
count = 1;
for (printf("Now sorting the array...") ; count < 1000; count++)
   /* Sorting statements here */
```

You can also omit the increment expression, performing the updating in the body of the for statement. Again, the semicolon must be included. To print the numbers from 0 to 99, for example, you could write

```
for (count = 0; count < 100; )
   printf("%d", count++);
```

The test expression that terminates the loop can be any C expression. As long as it evaluates to true (nonzero), the for statement continues to execute. You can use C's logical operators to construct complex test expressions. For example, the following for statement prints the elements of an array named array[], stopping when all elements have been printed or an element with a value of 0 is encountered:

```
for (count = 0; count < 1000 && array[count] != 0; count++)
   printf("%d", array[count]);
```

You could simplify this for loop even further by writing it as follows. (If you don't understand the change made to the test expression, you need to review Day 4.)

```
for (count = 0; count < 1000 && array[count]; )
   printf("%d", array[count++]);
```

You can follow the for statement with a null statement, allowing all the work to be done in the for statement itself. Remember, the null statement is a semicolon alone on a line. For example, to initialize all elements of a 1,000-element array to the value 50, you could write

```
for (count = 0; count < 1000; array[count++] = 50)
   ;
```

In this for statement, 50 is assigned to each member of the array by the increment part of the statement.

Day 4 mentioned that C's comma operator is most often used in for statements. You can create an expression by separating two subexpressions with the comma operator. The two subexpressions are evaluated (in left-to-right order), and the entire expression evaluates to the value of the right subexpression. By using the comma operator, you can make each part of a for statement perform multiple duties.

Imagine that you have two 1,000-element arrays, a[] and b[]. You want to copy the contents of a[] to b[] in reverse order so that after the copy operation, b[0] = a[999], b[1] = a[998], and so on. The following for statement does the trick:

```
for (i = 0, j = 999; i < 1000; i++, j--)
   b[j] = a[i];
```

The comma operator is used to initialize two variables, i and j. It is also used to increment part of these two variables with each loop.

The for Statement

```
for (initial; condition; increment)
   statement(s)
```

initial is any valid C expression. It is usually an assignment statement that sets a variable to a particular value.

condition is any valid C expression. It is usually a relational expression. When condition evaluates to false (zero), the for statement terminates, and execution passes to the first statement following statement(s); otherwise, the C statement(s) in statement(s) are executed.

increment is any valid C expression. It is usually an expression that increments a variable initialized by the initial expression.

statement(s) are the C statements that are executed as long as the condition remains true.

A for statement is a looping statement. It can have an initialization, test condition, and increment as parts of its command. The for statement executes the initial expression first. It then checks the condition. If the condition is true, the statements execute. Once the statements are completed, the increment expression is evaluated. The for statement then rechecks the condition and continues to loop until the condition is false.

Example 1

```
/* Prints the value of x as it counts from 0 to 9 */
int x;
for (x = 0; x <10; x++)
   printf( "\nThe value of x is %d", x );
```

Example 2

```
/*Obtains values from the user until 99 is entered */
int nbr = 0;
for ( ; nbr != 99; )
  scanf( "%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                       */
int value[10];
int ctr,nbr=0;
for (ctr = 0; ctr < 10 && nbr != 99; ctr++)
{
   puts("Enter a number, 99 to quit ");
   scanf("%d", &nbr);
   value[ctr] = nbr;
```

}
Nesting for Statements

A for statement can be executed within another for statement. This is called nesting. (You saw this on Day 4 with the if statement.) By nesting for statements, you can do some complex programming. Listing 6.2 is not a complex program, but it illustrates the nesting of two for statements.

Listing 6.2. Nested for statements.

```
1:  /* Demonstrates nesting two for statements */
2:
3:  #include <stdio.h>
4:
5:  void draw_box( int, int);
6:
7:  main()
8:  {
9:     draw_box( 8, 35 );
10:
11:    return 0;
12: }
13:
14: void draw_box( int row, int column )
15: {
16:    int col;
17:    for ( ; row > 0; row--)
18:    {
19:       for (col = column; col > 0; col--)
20:          printf("X");
21:
22:       printf("\n");
23:    }
24: }
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

ANALYSIS: The main work of this program is accomplished on line 20. When you run this program, 280 Xs are printed on-screen, forming an 8*35 square. The program has only one command to print an X, but it is nested in two loops.

In this listing, a function prototype for draw_box() is declared on line 5. This function takes two type int variables, row and column, which contain the dimensions of the box of Xs to be drawn. In line 9, main() calls draw_box() and passes the value 8 as the row and the value 35 as the column.

Looking closely at the draw_box() function, you might see a couple things you don't readily understand. The first is why the local variable col was declared. The second is why the second printf() in line 22 was used. Both of these will become clearer after you look at the two for loops.

Line 17 starts the first for loop. The initialization is skipped because the initial value of row was passed to the function. Looking at the condition, you see that this for loop is executed until the row is 0. On first executing line 17, row is 8; therefore, the program continues to line 19.

Line 19 contains the second for statement. Here the passed parameter, column, is copied to a local variable, col, of type int. The value of col is 35 initially (the value passed via column), and column retains its original value. Because col is greater than 0, line 20 is executed, printing an X. col is then decremented, and the loop continues. When col is 0, the for loop ends, and control goes to line 22. Line 22 causes the on-screen printing to start on a new line. (Printing is covered in detail on Day 7, "Fundamentals of Input and Output.") After moving to a new line on the screen, control reaches the end of the first for loop's statements, thus executing the increment expression, which subtracts 1 from row, making it 7. This puts control back at line 19. Notice that the value of col was 0 when it was last used. If column had been used instead of col, it would fail the condition test, because it will never be greater than 0. Only the first line would be printed. Take the initializer out of line 19 and change the two col variables to column to see what actually happens.

DON'T put too much processing in the for statement. Although you can use the comma separator, it is often clearer to put some of the functionality into the body of the loop.

DO remember the semicolon if you use a for with a null statement. Put the semi-colon placeholder on a separate line, or place a space between it and the end of the for statement. It's clearer to put it on a separate line.

```
for (count = 0; count < 1000; array[count] = 50) ;
   /* note space! */
```

The while Statement

The while statement, also called the while loop, executes a block of statements as long as a specified condition is true. The while statement has the following form:

```
while (condition)
   statement
```

condition is any C expression, and statement is a single or compound C statement. When program execution reaches a while statement, the following events occur:

1. The expression condition is evaluated.

2. If condition evaluates to false (that is, zero), the while statement terminates, and execution passes to the first statement following statement.

3. If condition evaluates to true (that is, nonzero), the C statement(s) in statement are executed.

4. Execution returns to step 1.

The operation of a while statement is shown in Figure 6.3.

Figure 6.3. The operation of a while statement.

Listing 6.3 is a simple program that uses a while statement to print the numbers 1 through 20. (This is the same task that is performed by a for statement in Listing 6.1.)

Listing 6.3. A simple while statement.

```
1:   /* Demonstrates a simple while statement */
2:
3:   #include <stdio.h>
4:
5:   int count;
6:
```

```
7:   int main()
8:   {
9:       /* Print the numbers 1 through 20 */
10:
11:      count = 1;
12:
13:      while (count <= 20)
14:      {
15:          printf("%d\n", count);
16:          count++;
17:      }
18:      return 0;
19:  }
1
```

ANALYSIS: Examine Listing 6.3 and compare it with Listing 6.1, which uses a for statement to perform the same task. In line 11, count is initialized to 1. Because the while statement doesn't contain an initialization section, you must take care of initializing any variables before starting the while. Line 13 is the actual while statement, and it contains the same condition statement from Listing 6.1, count <= 20. In the while loop, line 16 takes care of incrementing count. What do you think would happen if you forgot to put line 16 in this program? Your program wouldn't know when to stop, because count would always be 1, which is always less than 20.

You might have noticed that a while statement is essentially a for statement without the initialization and increment components. Thus,

for ( ; condition ; )

is equivalent to

while (condition)

Because of this equality, anything that can be done with a for statement can also be done with a while statement. When you use a while statement, any necessary initialization must first be performed in a separate statement, and the updating must be performed by a statement that is part of the while loop.

When initialization and updating are required, most experienced C programmers prefer to use a for statement rather than a while statement. This preference is based primarily on source code readability. When you use a for statement, the initialization, test, and increment expressions are located together and are easy to find and modify. With a while statement, the initialization and update expressions are located separately and might be less obvious.

The while Statement

```
while (condition)
   statement(s)
```

condition is any valid C expression, usually a relational expression. When condition evaluates to false (zero), the while statement terminates, and execution passes to the first statement following statement(s); otherwise, the first C statement in statement(s) is executed.

statement(s) is the C statement(s) that is executed as long as condition remains true.

A while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). If the condition is not true when the while command is first executed, the statement(s) is never executed.

Example 1

```
int x = 0;
while (x < 10)
{
   printf("\nThe value of x is %d", x );
   x++;
}
```

Example 2

```
/* get numbers until you get one greater than 99 */
int nbr=0;
while (nbr <= 99)
   scanf("%d", &nbr );
```

Example 3

```
/* Lets user enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                       */
int value[10];
int ctr = 0;
int nbr;
while (ctr < 10 && nbr != 99)
{
   puts("Enter a number, 99 to quit ");
   scanf("%d", &nbr);
   value[ctr] = nbr;
   ctr++;
}
```

Nesting while Statements

Just like the for and if statements, while statements can also be nested. Listing 6.4 shows an example of nested while statements. Although this isn't the best use of a while statement, the example does present some new ideas.

Listing 6.4. Nested while statements.

```
1:   /* Demonstrates nested while statements */
2:
3:   #include <stdio.h>
4:
5:   int array[5];
6:
7:   main()
8:   {
9:      int ctr = 0,
10:         nbr = 0;
11:
12:      printf("This program prompts you to enter 5 numbers\n");
13:      printf("Each number should be from 1 to 10\n");
14:
15:      while ( ctr < 5 )
16:      {
17:         nbr = 0;
18:         while (nbr < 1 || nbr > 10)
19:         {
20:            printf("\nEnter number %d of 5: ", ctr + 1 );
21:            scanf("%d", &nbr );
```

```
22:         }
23:
24:         array[ctr] = nbr;
25:         ctr++;
26:      }
27:
28:      for (ctr = 0; ctr < 5; ctr++)
29:         printf("Value %d is %d\n", ctr + 1, array[ctr] );
30:
31:      return 0;
32:  }
```

```
This program prompts you to enter 5 numbers
Each number should be from 1 to 10
Enter number 1 of 5: 3
Enter number 2 of 5: 6
Enter number 3 of 5: 3
Enter number 4 of 5: 9
Enter number 5 of 5: 2
Value 1 is 3
Value 2 is 6
Value 3 is 3
Value 4 is 9
Value 5 is 2
```

ANALYSIS: As in previous listings, line 1 contains a comment with a description of the program, and line 3 contains an #include statement for the standard input/output header file. Line 5 contains a declaration for an array (named array) that can hold five integer values. The function main() contains two additional local variables, ctr and nbr (lines 9 and 10). Notice that these variables are initialized to zero at the same time they are declared. Also notice that the comma operator is used as a separator at the end of line 9, allowing nbr to be declared as an int without restating the int type command. Stating declarations in this manner is a common practice for many C programmers. Lines 12 and 13 print messages stating what the program does and what is expected of the user. Lines 15 through 26 contain the first while command and its statements. Lines 18 through 22 also contain a nested while loop with its own statements that are all part of the outer while.

This outer loop continues to execute while ctr is less than 5 (line 15). As long as ctr is less than 5, line 17 sets nbr to 0, lines 18 through 22 (the nested while statement) gather a number in variable nbr, line 24 places the number in array, and line 25 increments ctr. Then the loop starts again. Therefore, the outer loop gathers five numbers and places each into array, indexed by ctr.

The inner loop is a good use of a while statement. Only the numbers from 1 to 10 are valid, so until the user enters a valid number, there is no point continuing the program. Lines 18 through 22 prevent continuation. This while statement states that while the number is less than 1 or greater than 10, the program should print a message to enter a number, and then get the number.

Lines 28 and 29 print the values that are stored in array. Notice that because the while statements are done with the variable ctr, the for command can reuse it. Starting at zero and incrementing by one, the for loops five times, printing the value of ctr plus one (because the count started at zero) and printing the corresponding value in array.

For additional practice, there are two things you can change in this program. The first is the values that the program accepts. Instead of 1 to 10, try making it accept from 1 to 100. You can also change the number of values that it accepts. Currently, it allows for five numbers. Try making it accept 10.

DON'T use the following convention if it isn't necessary:

```
while (x)
```

Instead, use this convention:
while (x != 0)

Although both work, the second is clearer when you're debugging (trying to find problems in) the code. When compiled, these produce virtually the same code.

DO use the for statement instead of the while statement if you need to initialize and increment within your loop. The for statement keeps the initialization, condition, and increment statements all together. The while statement does not.
The do...while Loop

C's third loop construct is the do...while loop, which executes a block of statements as long as a specified condition is true. The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the for loop and the while loop.

The structure of the do...while loop is as follows:

```
do
   statement
while (condition);
```

condition is any C expression, and statement is a single or compound C statement. When program execution reaches a do...while statement, the following events occur:

1. The statements in statement are executed.

2. condition is evaluated. If it's true, execution returns to step 1. If it's false, the loop terminates.

The operation of a do...while loop is shown in Figure 6.4.

Figure 6.4. The operation of a do...while loop.

The statements associated with a do...while loop are always executed at least once. This is because the test condition is evaluated at the end, instead of the beginning, of the loop. In contrast, for loops and while loops evaluate the test condition at the start of the loop, so the associated statements are not executed at all if the test condition is initially false.

The do...while loop is used less frequently than while and for loops. It is most appropriate when the statement(s) associated with the loop must be executed at least once. You could, of course, accomplish the same thing with a while loop by making sure that the test condition is true when execution first reaches the loop. A do...while loop probably would be more straightforward, however.

Listing 6.5 shows an example of a do...while loop.

Listing 6.5. A simple do...while loop.

```
1:   /* Demonstrates a simple do...while statement */
2:
3:   #include <stdio.h>
4:
5:   int get_menu_choice( void );
6:
7:   main()
8:   {
9:       int choice;
10:
11:      choice = get_menu_choice();
```

```
12:
13:     printf("You chose Menu Option %d\n", choice );
14:
15:     return 0;
16:  }
17:
18:  int get_menu_choice( void )
19:  {
20:     int selection = 0;
21:
22:     do
23:     {
24:        printf("\n" );
25:        printf("\n1 - Add a Record" );
26:        printf("\n2 - Change a record");
27:        printf("\n3 - Delete a record");
28:        printf("\n4 - Quit");
29:        printf("\n" );
30:        printf("\nEnter a selection: " );
31:
32:        scanf("%d", &selection );
33:
34:     }while ( selection < 1 || selection > 4 );
35:
36:     return selection;
37:  }
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 8
1 - Add a Record
2 - Change a record
3 - Delete a record
4 - Quit
Enter a selection: 4
You chose Menu Option 4
```

ANALYSIS: This program provides a menu with four choices. The user selects one of the four choices, and then the program prints the number selected. Programs later in this book use and expand on this concept. For now, you should be able to follow most of the listing. The main() function (lines 7 through 16) adds nothing to what you already know.

NOTE: The body of main() could have been written into one line, like this:
printf( "You chose Menu Option %d", get_menu_option() );

If you were to expand this program and act on the selection, you would need the value returned by get_menu_choice(), so it is wise to assign the value to a variable (such as choice).

Lines 18 through 37 contain get_menu_choice(). This function displays a menu on-screen (lines 24 through 30) and then gets a selection. Because you have to display a menu at least once to get an answer, it is appropriate to use a do...while loop. In the case of this program, the menu is displayed until a valid choice is entered. Line 34 contains the while part of the do...while statement and validates the value of the selection, appropriately named selection. If the value entered is not between 1 and 4, the menu is redisplayed, and the user is prompted for a new value. When a valid selection is entered, the program continues to line 36, which returns the value in the variable selection.

The do...while Statement

```
do
{
   statement(s)
}while (condition);
```

condition is any valid C expression, usually a relational expression. When condition evaluates to false (zero), the while statement terminates, and execution passes to the first statement following the while statement; otherwise, the program loops back to the do, and the C statement(s) in statement(s) is executed.

statement(s) is either a single C statement or a block of statements that are executed the first time through the loop and then as long as condition remains true.

A do...while statement is a C looping statement. It allows repeated execution of a statement or block of statements as long as the condition remains true (nonzero). Unlike the while statement, a do...while loop executes its statements at least once.

Example 1

```
/* prints even though condition fails! */
int x = 10;
do
{
   printf("\nThe value of x is %d", x );
}while (x != 10);
```

Example 2

```
/* gets numbers until the number is greater than 99 */
int nbr;
do
{
   scanf("%d", &nbr );
}while (nbr <= 99);
```

Example 3

```
/* Enables user to enter up to 10 integer values       */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops                             */
int value[10];
int ctr = 0;
int nbr;
do
{
   puts("Enter a number, 99 to quit ");
   scanf( "%d", &nbr);
   value[ctr] = nbr;
   ctr++;
}while (ctr < 10 && nbr != 99);
```

Nested Loops

The term nested loop refers to a loop that is contained within another loop. You have seen examples of some nested statements. C places no limitations on the nesting of loops, except that each inner loop must be enclosed completely in the outer loop; you can't have overlapping loops. Thus, the following is not allowed:

```
for ( count = 1; count < 100; count++)
{
   do
   {
      /* the do...while loop */
} /* end of for loop */
   }while (x != 0);
```

If the do...while loop is placed entirely in the for loop, there is no problem:

```
for (count = 1; count < 100; count++)
{
   do
   {
      /* the do...while loop */
   }while (x != 0);
} /* end of for loop */
```

When you use nested loops, remember that changes made in the inner loop might affect the outer loop as well. Note, however, that the inner loop might be independent from any variables in the outer loop; in this example, they are not. In the previous example, if the inner do...while loop modifies the value of count, the number of times the outer for loop executes is affected.

Good indenting style makes code with nested loops easier to read. Each level of loop should be indented one step further than the last level. This clearly labels the code associated with each loop.

DON'T try to overlap loops. You can nest them, but they must be entirely within each other.

DO use the do...while loop when you know that a loop should be executed at least once.

Summary

Now you are almost ready to start writing real C programs on your own.

C has three loop statements that control program execution: for, while, and do...while. Each of these constructs lets your program execute a block of statements zero times, one time, or more than one time, based on the condition of certain program variables. Many programming tasks are well-served by the repetitive execution allowed by these loop statements.

Although all three can be used to accomplish the same task, each is different. The for statement lets you initialize, evaluate, and increment all in one command. The while statement operates as long as a condition is true. The do...while statement always executes its statements at least once and continues to execute them until a condition is false.

Nesting is the placing of one command within another. C allows for the nesting of any of its commands. Nesting the if statement was demonstrated on Day 4. In this chapter, the for, while, and do...while statements were nested.

Quiz

1. What is the index value of the first element in an array?

2. What is the difference between a for statement and a while statement?

3. What is the difference between a while statement and a do...while statement?

4. Is it true that a while statement can be used and still get the same results as coding a for statement?

5. What must you remember when nesting statements?

6. Can a while statement be nested in a do...while statement?

7. What are the four parts of a for statement?

8. What are the two parts of a while statement?

9. What are the two parts of a do...while statement?

Exercises
1. Write a declaration for an array that will hold 50 type long values.

2. Show a statement that assigns the value of 123.456 to the 50th element in the array from exercise 1.

3. What is the value of x when the following statement is complete?

```
for (x = 0; x < 100, x++) ;
```

4. What is the value of ctr when the following statement is complete?

```
for (ctr = 2; ctr < 10; ctr += 3) ;
```

5. How many Xs does the following print?

```
for (x = 0; x < 10; x++)
   for (y = 5; y > 0; y--)
      puts("X");
```

6. Write a for statement to count from 1 to 100 by 3s.

7. Write a while statement to count from 1 to 100 by 3s.

8. Write a do...while statement to count from 1 to 100 by 3s.

9. BUG BUSTER: What is wrong with the following code fragment?

```
record = 0;
while (record < 100)
{
   printf( "\nRecord %d ", record );
   printf( "\nGetting next number..." );
}
```

10. BUG BUSTER: What is wrong with the following code fragment? (MAXVALUES is not the problem!)

```
for (counter = 1; counter < MAXVALUES; counter++);
   printf("\nCounter = %d", counter );
```

**Fundamentals of Input and Output**

In most programs you create, you will need to display information on the screen or read information from the keyboard. Many of the programs presented in earlier chapters performed these tasks, but you might not have understood exactly how. Today you will learn
The basics of C's input and output statements

How to display information on-screen with the printf() and puts() library functions

How to format the information that is displayed on-screen

How to read data from the keyboard with the scanf() library function

This chapter isn't intended to be a complete treatment of these topics, but it provides enough information so that you can start writing real programs. These topics are covered in greater detail later in this book.

Displaying Information On-Screen

You will want most of your programs to display information on-screen. The two most frequently used ways to do this are with C's library functions printf() and puts().

The printf() Function

The printf() function, part of the standard C library, is perhaps the most versatile way for a program to display data on-screen. You've already seen printf() used in many of the examples in this book. Now you will to see how printf() works.

Printing a text message on-screen is simple. Call the printf() function, passing the desired message enclosed in double quotation marks. For example, to display An error has occurred! on-screen, you write

```
printf("An error has occurred!");
```

In addition to text messages, however, you frequently need to display the value of program variables. This is a little more complicated than displaying only a message. For example, suppose you want to display the value of the numeric variable x on-screen, along with some identifying text. Furthermore, you want the information to start at the beginning of a new line. You could use the printf() function as follows:

```
printf("\nThe value of x is %d", x);
```

The resulting screen display, assuming that the value of x is 12, would be

```
The value of x is 12
```

In this example, two arguments are passed to printf(). The first argument is enclosed in double quotation marks and is called the format string. The second argument is the name of the variable (x) containing the value to be printed.

The printf() Format Strings

A printf() format string specifies how the output is formatted. Here are the three possible components of a format string:
Literal text is displayed exactly as entered in the format string. In the preceding example, the characters starting with the T (in The) and up to, but not including, the % comprise a literal string.

An escape sequence provides special formatting control. An escape sequence consists of a backslash (\) followed by a single character. In the preceding example, \n is an escape sequence. It is called the newline character, and it means "move to the start of the next line." Escape sequences are also used to print certain characters. Escape sequences are listed in Table 7.1.

A conversion specifier consists of the percent sign (%) followed by a single character. In the example, the conversion specifier is %d. A conversion specifier tells printf() how to interpret the variable(s) being printed. The %d tells printf() to interpret the variable x as a signed decimal integer.
Table 7.1. The most frequently used escape sequences.

| Sequence | Meaning |
|---|---|
| \a | Bell (alert) |
| \b | Backspace |
| \n | Newline |

\t Horizontal tab
\\ Backslash
\? Question mark
\' Single quotation

The printf() Escape Sequences

Now let's look at the format string components in more detail. Escape sequences are used to control the location of output by moving the screen cursor. They are also used to print characters that would otherwise have a special meaning to printf(). For example, to print a single backslash character, include a double backslash (\\) in the format string. The first backslash tells printf() that the second backslash is to be interpreted as a literal character, not as the start of an escape sequence. In general, the backslash tells printf() to interpret the next character in a special manner. Here are some examples:

| Sequence | Meaning |
|---|---|
| n | The character n |
| \n | Newline |
| \" | The double quotation character |
| " | The start or end of a string |

Table 7.1 lists C's most commonly used escape sequences. A full list can be found on Day 15, "Pointers: Beyond the Basics."

Listing 7.1 demonstrates some of the frequently used escape sequences.

Listing 7.1. Using printf() escape sequences.

```
1:  /* Demonstration of frequently used escape sequences */
2:
3:  #include <stdio.h>
4:
5:  #define QUIT  3
6:
7:  int  get_menu_choice( void );
8:  void print_report( void );
9:
10: main()
11: {
12:    int choice = 0;
13:
14:    while (choice != QUIT)
15:    {
16:       choice = get_menu_choice();
17:
18:       if (choice == 1)
19:          printf("\nBeeping the computer\a\a\a" );
20:       else
21:       {
22:          if (choice == 2)
23:             print_report();
24:       }
25:    }
26:    printf("You chose to quit!\n");
27:
28:    return 0;
29: }
30:
```

```
31:  int get_menu_choice( void )
32:  {
33:     int selection = 0;
34:
35:     do
36:     {
37:        printf( "\n" );
38:        printf( "\n1 - Beep Computer" );
39:        printf( "\n2 - Display Report");
40:        printf( "\n3 - Quit");
41:        printf( "\n" );
42:        printf( "\nEnter a selection:" );
43:
44:        scanf( "%d", &selection );
45:
46:     }while ( selection < 1 || selection > 3 );
47:
48:     return selection;
49:  }
50:
51:  void print_report( void )
52:  {
53:     printf( "\nSAMPLE REPORT" );
54:     printf( "\n\nSequence\tMeaning" );
55:     printf( "\n==========\t=======" );
56:     printf( "\n\\a\t\tbell (alert)" );
57:     printf( "\n\\b\t\tbackspace" );
58:     printf( "\n...\t\t...");
59:  }
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence        Meaning
==========      =======
\a              bell (alert)
\b              backspace
...             ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

ANALYSIS: Listing 7.1 seems long compared with previous examples, but it offers some additions that are worth noting. The STDIO.H header was included in line 3 because printf() is used in this listing. In line 5, a constant named QUIT is defined. From Day 3, "Storing Data: Variables and Constants," you know that #define makes using the constant QUIT equivalent to using the value 3. Lines 7 and 8 are function prototypes. This program has two functions: get_menu_choice() and print_report(). get_menu_choice() is defined in lines 31 through 49. This is similar to the menu function in Listing 6.5. Lines 37 and 41 contain calls to printf() that print the newline escape

sequence. Lines 38, 39, 40, and 42 also use the newline escape character, and they print text. Line 37 could have been eliminated by changing line 38 to the following:

```
printf( "\n\n1 - Beep Computer" );
```

However, leaving line 37 makes the program easier to read.

Looking at the main() function, you see the start of a while loop on line 14. The while loop's statements will keep looping as long as choice is not equal to QUIT. Because QUIT is a constant, you could have replaced it with 3; however, the program wouldn't be as clear. Line 16 gets the variable choice, which is then analyzed in lines 18 through 25 in an if statement. If the user chooses 1, line 19 prints the newline character, a message, and then three beeps. If the user selects 2, line 23 calls the function print_report().

print_report() is defined on lines 51 through 59. This simple function shows the ease of using printf() and the escape sequences to print formatted information to the screen. You've already seen the newline character. Lines 54 through 58 also use the tab escape character, \t. It aligns the columns of the report vertically. Lines 56 and 57 might seem confusing at first, but if you start at the left and work to the right, they make sense. Line 56 prints a newline (\n), then a backslash (\), then the letter a, and then two tabs (\t\t). The line ends with some descriptive text, (bell (alert)). Line 57 follows the same format.

This program prints the first two lines of Table 7.1, along with a report title and column headings. In exercise 9 at the end of this chapter, you will complete this program by making it print the rest of the table.

The printf() Conversion Specifiers

The format string must contain one conversion specifier for each printed variable. printf() then displays each variable as directed by its corresponding conversion specifier. You'll learn more about this process on Day 15. For now, be sure to use the conversion specifier that corresponds to the type of variable being printed.

Exactly what does this mean? If you're printing a variable that is a signed decimal integer (types int and long), use the %d conversion specifier. For an unsigned decimal integer (types unsigned int and unsigned long), use %u. For a floating-point variable (types float and double), use the %f specifier. The conversion specifiers you need most often are listed in Table 7.2.

Table 7.2. The most commonly needed conversion specifiers.

| Specifier | Meaning | Types Converted |
|---|---|---|
| %c | Single character | char |
| %d | Signed decimal integer | int, short |
| %ld | Signed long decimal integer | long |
| %f | Decimal floating-point number | float, double |
| %s | Character string | char arrays |
| %u | Unsigned decimal integer | unsigned int, unsigned short |
| %lu | Unsigned long decimal integer | unsigned long |

The literal text of a format specifier is anything that doesn't qualify as either an escape sequence or a conversion specifier. Literal text is simply printed as is, including all spaces.

What about printing the values of more than one variable? A single printf() statement can print an unlimited number of variables, but the format string must contain one conversion specifier for each variable. The conversion specifiers are paired with variables in left-to-right order. If you write

```
printf("Rate = %f, amount = %d", rate, amount);
```

the variable rate is paired with the %f specifier, and the variable amount is paired with the %d specifier. The positions of the conversion specifiers in the format string determine the position of the output. If there are more

variables passed to printf() than there are conversion specifiers, the unmatched variables aren't printed. If there are more specifiers than variables, the unmatched specifiers print "garbage."

You aren't limited to printing the value of variables with printf(). The arguments can be any valid C expression. For example, to print the sum of x and y, you could write

```
z = x + y;
printf("%d", z);
```

You also could write

```
printf("%d", x + y);
```

Any program that uses printf() should include the header file STDIO.H. Listing 7.2 demonstrates the use of printf(). Day 15 gives more details on printf().

Listing 7.2. Using printf() to display numerical values.

```
1:  /* Demonstration using printf() to display numerical values. */
2:
3:  #include <stdio.h>
4:
5:  int a = 2, b = 10, c = 50;
6:  float f = 1.05, g = 25.5, h = -0.1;
7:
8:  main()
9:  {
10:     printf("\nDecimal values without tabs: %d %d %d", a, b, c);
11:     printf("\nDecimal values with tabs: \t%d \t%d \t%d", a, b, c);
12:
13:     printf("\nThree floats on 1 line: \t%f\t%f\t%f", f, g, h);
14:     printf("\nThree floats on 3 lines: \n\t%f\n\t%f\n\t%f", f, g, h);
15:
16:     printf("\nThe rate is %f%%", f);
17:     printf("\nThe result of %f/%f = %f\n", g, f, g / f);
18:
19:     return 0;
20: }
```

```
Decimal values without tabs: 2 10 50
Decimal values with tabs:       2       10      50
Three floats on 1 line:         1.050000        25.500000       -0.100000
Three floats on 3 lines:
        1.050000
        25.500000
        -0.100000
The rate is 1.050000%
The result of 25.500000/1.050000 = 24.285715
```

ANALYSIS: Listing 7.2 prints six lines of information. Lines 10 and 11 each print three decimals: a, b, and c. Line 10 prints them without tabs, and line 11 prints them with tabs. Lines 13 and 14 each print three float variables: f, g, and h. Line 13 prints them on one line, and line 14 prints them on three lines. Line 16 prints a float variable, f, followed by a percent sign. Because a percent sign is normally a message to print a variable, you must place two in a row to print a single percent sign. This is exactly like the backslash escape character. Line 17 shows one final concept. When printing values in conversion specifiers, you don't have to use variables. You can also use expressions such as g / f, or even constants.

DON'T try to put multiple lines of text into one printf() statement. In most instances, it's clearer to print multiple lines with multiple print statements than to use just one with several newline (\n) escape characters.

DON'T forget to use the newline escape character when printing multiple lines of information in separate printf() statements.

DON'T misspell stdio.h. Many C programmers accidentally type studio.h; however, there is no u.
The printf() Function
#include <stdio.h>
printf( format-string[,arguments,...]);

printf() is a function that accepts a series of arguments, each applying to a conversion specifier in the given format string. printf() prints the formatted information to the standard output device, usually the display screen. When using printf(), you need to include the standard input/output header file, STDIO.H.

The format-string is required; however, arguments are optional. For each argument, there must be a conversion specifier. Table 7.2 lists the most commonly used conversion specifiers.

The format-string can also contain escape sequences. Table 7.1 lists the most frequently used escape sequences.

The following are examples of calls to printf() and their output:

Example 1 Input

```
#include <stdio.h>
main()
{
   printf("This is an example of something printed!");
   return 0;
}
```

Example 1 Output

This is an example of something printed!

Example 2 Input

```
printf("This prints a character, %c\na number, %d\na floating \
point, %f", `z', 123, 456.789 );
```

Example 2 Output

```
This prints a character, z
a number, 123
a floating point, 456.789
```

Displaying Messages with puts()

The puts() function can also be used to display text messages on-screen, but it can't display numeric variables. puts() takes a single string as its argument and displays it, automatically adding a newline at the end. For example, the statement

```
puts("Hello, world.");
```

performs the same action as

printf("Hello, world.\n");

You can include escape sequences (including \n) in a string passed to puts(). They have the same effect as when they are used with printf() (see Table 7.1).

Any program that uses puts() should include the header file STDIO.H. Note that STDIO.H should be included only once in a program.

DO use the puts() function instead of the printf() function whenever you want to print text but don't need to print any variables.

DON'T try to use conversion specifiers with the puts() statement.
The puts() Function
#include <stdio.h>
puts( string );

puts() is a function that copies a string to the standard output device, usually the display screen. When you use puts(), include the standard input/output header file (STDIO.H). puts() also appends a newline character to the end of the string that is printed. The for- mat string can contain escape sequences. Table 7.1 lists the most frequently used escape sequences.

The following are examples of calls to puts() and their output:

Example 1 Input

puts("This is printed with the puts() function!");

Example 1 Output

This is printed with the puts() function!

Example 2 Input

puts("This prints on the first line. \nThis prints on the second line.");
puts("This prints on the third line.");
puts("If these were printf()s, all four lines would be on two lines!");

Example 2 Output

This prints on the first line.
This prints on the second line.
This prints on the third line.
If these were printf()s, all four lines would be on two lines!
Inputting Numeric Data with scanf()

Just as most programs need to output data to the screen, they also need to input data from the keyboard. The most flexible way your program can read numeric data from the keyboard is by using the scanf() library function.

The scanf() function reads data from the keyboard according to a specified format and assigns the input data to one or more program variables. Like printf(), scanf() uses a format string to describe the format of the input. The format string utilizes the same conversion specifiers as the printf() function. For example, the statement

scanf("%d", &x);

reads a decimal integer from the keyboard and assigns it to the integer variable x. Likewise, the following statement reads a floating-point value from the keyboard and assigns it to the variable rate:

scanf("%f", &rate);

What is that ampersand (&) before the variable's name? The & symbol is C's address-of operator, which is fully explained on Day 9, "Understanding Pointers." For now, all you need to remember is that scanf() requires the & symbol before each numeric variable name in its argument list (unless the variable is a pointer, which is also explained on Day 9).

A single scanf() can input more than one value if you include multiple conversion specifiers in the format string and variable names (again, each preceded by & in the argument list). The following statement inputs an integer value and a floating-point value and assigns them to the variables x and rate, respectively:

scanf("%d %f", &x, &rate);

When multiple variables are entered, scanf() uses white space to separate input into fields. White space can be spaces, tabs, or new lines. Each conversion specifier in the scanf() format string is matched with an input field; the end of each input field is identified by white space.

This gives you considerable flexibility. In response to the preceding scanf(), you could enter

10 12.45

You also could enter this:

10 12.45

or this:

10
12.45

As long as there's some white space between values, scanf() can assign each value to its variable.

As with the other functions discussed in this chapter, programs that use scanf() must include the STDIO.H header file. Although Listing 7.3 gives an example of using scanf(), a more complete description is presented on Day 15.

Listing 7.3. Using scanf() to obtain numerical values.

```
1:  /* Demonstration of using scanf() */
2:
3:  #include <stdio.h>
4:
5:  #define QUIT 4
6:
7:  int get_menu_choice( void );
8:
9:  main()
10: {
11:     int   choice    = 0;
12:     int   int_var   = 0;
13:     float float_var = 0.0;
14:     unsigned unsigned_var = 0;
15:
16:     while (choice != QUIT)
17:     {
```

```
18:         choice = get_menu_choice();
19:
20:         if (choice == 1)
21:         {
22:            puts("\nEnter a signed decimal integer (i.e. -123)");
23:            scanf("%d", &int_var);
24:         }
25:         if (choice == 2)
26:         {
27:            puts("\nEnter a decimal floating-point number\
28                 (i.e. 1.23)");
29:            scanf("%f", &float_var);
30:         }
31:         if (choice == 3)
32:         {
33:             puts("\nEnter an unsigned decimal integer \
34                 (i.e. 123)" );
35:             scanf( "%u", &unsigned_var );
36:         }
37:     }
38:     printf("\nYour values are: int: %d  float: %f  unsigned: %u \n",
39:                               int_var, float_var, unsigned_var );
40:
41:    return 0;
42:  }
43:
44:  int get_menu_choice( void )
45:  {
46:     int selection = 0;
47:
48:     do
49:     {
50:        puts( "\n1 - Get a signed decimal integer" );
51:        puts( "2 - Get a decimal floating-point number" );
52:        puts( "3 - Get an unsigned decimal integer" );
53:        puts( "4 - Quit" );
54:        puts( "\nEnter a selection:" );
55:
56:        scanf( "%d", &selection );
57:
58:     }while ( selection < 1 || selection > 4 );
59:
60:     return selection;
61:  }
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
1
Enter a signed decimal integer (i.e. -123)
-123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
```

```
Enter a selection:
3
Enter an unsigned decimal integer (i.e. 123)
321
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
2
Enter a decimal floating point number (i.e. 1.23)
1231.123
1 - Get a signed decimal integer
2 - Get a decimal floating-point number
3 - Get an unsigned decimal integer
4 - Quit
Enter a selection:
4
Your values are: int: -123  float: 1231.123047 unsigned: 321
```

ANALYSIS: Listing 7.3 uses the same menu concepts that were used in Listing 7.1. The differences in get_menu_choice() (lines 44 through 61) are minor but should be noted. First, puts() is used instead of printf(). Because no variables are printed, there is no need to use printf(). Because puts() is being used, the newline escape characters have been removed from lines 51 through 53. Line 58 was also changed to allow values from 1 to 4 because there are now four menu options. Notice that line 56 has not changed; however, now it should make a little more sense. scanf() gets a decimal value and places it in the variable selection. The function returns selection to the calling program in line 60.

Listings 7.1 and 7.3 use the same main() structure. An if statement evaluates choice, the return value of get_menu_choice(). Based on choice's value, the program prints a message, asks for a number to be entered, and reads the value using scanf(). Notice the difference between lines 23, 29, and 35. Each is set up to get a different type of variable. Lines 12 through 14 declare variables of the appropriate types.

When the user selects Quit, the program prints the last-entered number for all three types. If the user didn't enter a value, 0 is printed, because lines 12, 13, and 14 initialized all three types. One final note on lines 20 through 36: The if statements used here are not structured well. If you're thinking that an if...else structure would have been better, you're correct. Day 14, "Working with the Screen, Printer, and Keyboard," introduces a new control statement, switch. This statement offers an even better option.

DON'T forget to include the address-of operator (&) when using scanf() variables.

DO use printf() or puts() in conjunction with scanf(). Use the printing functions to display a prompting message for the data you want scanf() to get.

The scanf() Function

```
#include <stdio.h>
scanf( format-string[,arguments,...]);
```

scanf() is a function that uses a conversion specifier in a given format-string to place values into variable arguments. The arguments should be the addresses of the variables rather than the actual variables themselves. For numeric variables, you can pass the address by putting the address-of operator (&) at the beginning of the variable name. When using scanf(), you should include the STDIO.H header file.

scanf() reads input fields from the standard input stream, usually the keyboard. It places each of these read fields into an argument. When it places the information, it converts it to the format of the corresponding specifier in the

format string. For each argument, there must be a conversion specifier. Table 7.2 lists the most commonly needed conversion specifiers.

Example 1

```
int x, y, z;
scanf( "%d %d %d", &x, &y, &z);
```

Example 2

```
#include <stdio.h>
main()
{
   float y;
   int x;
   puts( "Enter a float, then an int" );
   scanf( "%f %d", &y, &x);
   printf( "\nYou entered %f and %d ", y, x );
   return 0;
}
```

Summary

With the completion of this chapter, you are ready to write your own C programs. By combining the printf(), puts(), and scanf() functions and the programming control statements you learned about in earlier chapters, you have the tools needed to write simple programs.

Screen display is performed with the printf() and puts() functions. The puts() function can display text messages only, whereas printf() can display text messages and variables. Both functions use escape sequences for special characters and printing controls.

The scanf() function reads one or more numeric values from the keyboard and interprets each one according to a conversion specifier. Each value is assigned to a program variable.

Q&A

Q Why should I use puts() if printf() does everything puts() does and more?

A Because printf() does more, it has additional overhead. When you're trying to write a small, efficient program, or when your programs get big and resources are valuable, you will want to take advantage of the smaller overhead of puts(). In general, you should use the simplest available resource.

Q Why do I need to include STDIO.H when I use printf(), puts(), or scanf()?

A STDIO.H contains the prototypes for the standard input/output functions. printf(), puts(), and scanf() are three of these standard functions. Try running a program without the STDIO.H header and see the errors and warnings you get.

Q What happens if I leave the address-of operator (&) off a scanf() variable?

A This is an easy mistake to make. Unpredictable results can occur if you forget the address-of operator. When you read about pointers on Days 9 and 13, you will understand this better. For now, know that if you omit the address-of operator, scanf() doesn't place the entered information in your variable, but in some other place in memory. This could do anything from apparently having no effect to locking up your computer so that you must reboot.

Workshop

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

Quiz
1. What is the difference between puts() and printf()?

2. What header file should you include when you use printf()?

3. What do the following escape sequences do?

a. \\

b. \b

c. \n

d. \t

e. \a

4. What conversion specifiers should be used to print the following?

a. A character string

b. A signed decimal integer

c. A decimal floating-point number

5. What is the difference between using each of the following in the literal text of puts()?

a. b

b. \b

c. \

d. \\

**Using Numeric Arrays**
Arrays are a type of data storage that you often use in C programs. You had a brief introduction to arrays on Day 6, "Basic Program Control." Today you will learn
What an array is

The definition of single- and multidimensional numeric arrays

How to declare and initialize arrays
What Is an Array?

An array is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an array element. Why do you need arrays in your programs? This question can be answered with an example. If you're keeping track of your business expenses for 1998 and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with 12 compartments.

Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This

approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments. Figure 8.1 illustrates the difference between using individual variables and an array.

Figure 8.1. Variables are like individual folders, whereas an array is like a single folder with many compartments.

Single-Dimensional Arrays

A single-dimensional array has only a single subscript. A subscript is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array. An example should make this clear. For the business expenses program, you could use the following line to declare an array of type float:

```
float expenses[12];
```

The array is named expenses, and it contains 12 elements. Each of the 12 elements is the exact equivalent of a single float variable. All of C's data types can be used for arrays. C array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11. In the preceding example, January's expense total would be stored in expenses[0], February's in expenses[1], and so on.

When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array. Individual array elements are stored in sequential memory locations, as shown in Figure 8.2.

Figure 8.2. Array elements are stored in sequential memory locations.

The location of array declarations in your source code is important. As with nonarray variables, the declaration's location affects how your program can use the array. The effect of a declaration's location is covered in more detail on Day 12, "Understanding Variable Scope." For now, place your array declarations with other variable declarations, just before the start of main().

An array element can be used in your program anywhere a nonarray variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets. For example, the following statement stores the value 89.95 in the second array element (remember, the first array element is expenses[0], not expenses[1]):

```
expenses[1] = 89.95;
```

Likewise, the statement

```
expenses[10] = expenses[11];
```

assigns a copy of the value that is stored in array element expenses[11] into array element expenses[10]. When you refer to an array element, the array subscript can be a literal constant, as in these examples. However, your programs might often use a subscript that is a C integer variable or expression, or even another array element. Here are some examples:

```
float expenses[100];
int a[10];
/* additional statements go here */
expenses[i] = 100;        /* i is an integer variable */
expenses[2 + 3] = 100;    /* equivalent to expenses[5] */
expenses[a[2]] = 100;     /* a[] is an integer array */
```

That last example might need an explanation. If, for instance, you have an integer array named a[] and the value 8 is stored in element a[2], writing

expenses[a[2]]

has the same effect as writing

expenses[8];

When you use arrays, keep the element numbering scheme in mind: In an array of n ele-ments, the allowable subscripts range from 0 to n-1. If you use the subscript value n, you might get program errors. The C compiler doesn't recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results.

WARNING: Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9.

Sometimes you might want to treat an array of n elements as if its elements were numbered 1 through n. For instance, in the previous example, a more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array as follows. You could also store some related data in element 0 (the yearly expense total, perhaps).

float expenses[13];

The program EXPENSES.C in Listing 8.1 demonstrates the use of an array. This is a simple program with no real practical use; it's for demonstration purposes only.

Listing 8.1. EXPENSES.C demonstrates the use of an array.

```
1:   /* EXPENSES.C - Demonstrates use of an array */
2:
3:   #include <stdio.h>
4:
5:   /* Declare an array to hold expenses, and a counter variable */
6:
7:   float expenses[13];
8:   int count;
9:
10:  main()
11:  {
12:     /* Input data from keyboard into array */
13:
14:     for (count = 1; count < 13; count++)
15:     {
16:        printf("Enter expenses for month %d: ", count);
17:        scanf("%f", &expenses[count]);
18:     }
19:
20:     /* Print array contents */
21:
22:     for (count = 1; count < 13; count++)
23:     {
24:        printf("Month %d = $%.2f\n", count, expenses[count]);
25:     }
26:     return 0;
27:  }
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
```

```
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00
Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
Month 8 = $195.00
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

ANAALYSIS: When you run EXPENSES.C, the program prompts you to enter expenses for months 1 through 12. The values you enter are stored in an array. You must enter a value for each month. After the 12th value is entered, the array contents are displayed on-screen.

The flow of the program is similar to listings you've seen before. Line 1 starts with a comment that describes what the program does. Notice that the name of the program, EXPENSES.C, is included. When the name of the program is included in a comment, you know which program you're viewing. This is helpful when you're reviewing printouts of a listing.

Line 5 contains an additional comment explaining the variables that are being declared. In line 7, an array of 13 elements is declared. In this program, only 12 elements are needed, one for each month, but 13 have been declared. The for loop in lines 14 through 18 ignores element 0. This lets the program use elements 1 through 12, which relate directly to the 12 months. Going back to line 8, a variable, count, is declared and is used throughout the program as a counter and an array index.

The program's main() function begins on line 10. As stated earlier, this program uses a for loop to print a message and accept a value for each of the 12 months. Notice that in line 17, the scanf() function uses an array element. In line 7, the expenses array was declared as float, so %f is used. The address-of operator (&) also is placed before the array element, just as if it were a regular type float variable and not an array element.

Lines 22 through 25 contain a second for loop that prints the values just entered. An additional formatting command has been added to the printf() function so that the expenses values print in a more orderly fashion. For now, know that %.2f prints a floating number with two digits to the right of the decimal. Additional formatting commands are covered in more detail on Day 14, "Working with the Screen, Printer, and Keyboard."

DON'T forget that array subscripts start at element 0.

DO use arrays instead of creating several variables that store the same thing. For example, if you want to store total sales for each month of the year, create an array with 12 elements to hold sales rather than creating a sales variable for each month.

Multidimensional Arrays

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts, a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C array can have. (There is a limit on total array size, as discussed later in this chapter.)

For example, you might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

```
int checker[8][8];
```

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7]. The structure of this two-dimensional array is illustrated in Figure 8.3.

Figure 8.3. A two-dimensional array has a row-and-column structure.

Similarly, a three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory. More detail on array storage is presented on Day 15, "Pointers: Beyond the Basics."

Naming and Declaring Arrays

The rules for assigning names to arrays are the same as for variable names, covered on Day 3, "Storing Data: Variables and Constants." An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). As you have probably realized, array declarations follow the same form as declarations of nonarray variables, except that the number of elements in the array must be enclosed in square brackets immediately following the array name.

When you declare an array, you can specify the number of elements with a literal constant (as was done in the earlier examples) or with a symbolic constant created with the #define directive. Thus, the following:

```
#define MONTHS 12
int array[MONTHS];
```

is equivalent to this statement:

```
int array[12];
```

With most compilers, however, you can't declare an array's elements with a symbolic con-stant created with the const keyword:

```
const int MONTHS = 12;
int array[MONTHS];          /* Wrong! */
```

Listing 8.2, GRADES.C, is another program demonstrating the use of a single-dimensional array. GRADES.C uses an array to store 10 grades.

Listing 8.2. GRADES.C stores 10 grades in an array.

```
1:   /*GRADES.C - Sample program with array */
2:   /* Get 10 grades and then average them */
3:
4:   #include <stdio.h>
5:
6:   #define MAX_GRADE 100
7:   #define STUDENTS  10
8:
9:   int grades[STUDENTS];
10:
```

```
11:  int idx;
12:  int total = 0;          /* used for average */
13:
14:  main()
15:  {
16:     for( idx=0;idx< STUDENTS;idx++)
17:     {
18:        printf( "Enter Person %d's grade: ", idx +1);
19:        scanf( "%d", &grades[idx] );
20:
21:        while ( grades[idx] > MAX_GRADE )
22:        {
23:           printf( "\nThe highest grade possible is %d",
24                 MAX_GRADE );
25:           printf( "\nEnter correct grade: " );
26:           scanf( "%d", &grades[idx] );
27:        }
28:
29:        total += grades[idx];
30:     }
31:
32:     printf( "\n\nThe average score is %d\n", ( total / STUDENTS) );
33:
34:     return (0);
35:  }
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

ANALYSIS: Like EXPENSES.C, this listing prompts the user for input. It prompts for 10 people's grades. Instead of printing each grade, it prints the average score.

As you learned earlier, arrays are named like regular variables. On line 9, the array for this program is named grades. It should be safe to assume that this array holds grades. On lines 6 and 7, two constants, MAX_GRADE and STUDENTS, are defined. These constants can be changed easily. Knowing that STUDENTS is defined as 10, you then know that the grades array has 10 elements. Two other variables are declared, idx and total. An abbreviation of index, idx is used as a counter and array subscript. A running total of all grades is kept in total.

The heart of this program is the for loop in lines 16 through 30. The for statement initializes idx to 0, the first subscript for an array. It then loops as long as idx is less than the number of students. Each time it loops, it increments idx by 1. For each loop, the program prompts for a person's grade (lines 18 and 19). Notice that in line 18, 1 is added to idx in order to count the people from 1 to 10 instead of from 0 to 9. Because arrays start with subscript 0, the first grade is put in grade[0]. Instead of confusing users by asking for Person 0's grade, they are asked for Person 1's grade.

Lines 21 through 27 contain a while loop nested within the for loop. This is an edit check that ensures that the grade isn't higher than the maximum grade, MAX_GRADE. Users are prompted to enter a correct grade if they enter a grade that is too high. You should check program data whenever you can.

Line 29 adds the entered grade to a total counter. In line 32, this total is used to print the average score (total/STUDENTS).

DO use #define statements to create constants that can be used when declaring arrays. Then you can easily change the number of elements in the array. In GRADES.C, for example, you could change the number of students in the #define, and you wouldn't have to make any other changes in the program.

DO avoid multidimensional arrays with more than three dimensions. Remember, multidimensional arrays can get very big very quickly.

Initializing Arrays

You can initialize all or part of an array when you first declare it. Follow the array declaration with an equal sign and a list of values enclosed in braces and separated by commas. The listed values are assigned in order to array elements starting at number 0. For example, the following code assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3]:

```
int array[4] = { 100, 200, 300, 400 };
```

If you omit the array size, the compiler creates an array just large enough to hold the initialization values. Thus, the following statement would have exactly the same effect as the previous array declaration statement:

```
int array[] = { 100, 200, 300, 400 };
```

You can, however, include too few initialization values, as in this example:

```
int array[10] = { 1, 2, 3 };
```

If you don't explicitly initialize an array element, you can't be sure what value it holds when the program runs. If you include too many initializers (more initializers than array elements), the compiler detects an error.

Initializing Multidimensional Arrays

Multidimensional arrays can also be initialized. The list of initialization values is assigned to array elements in order, with the last array subscript changing first. For example:

```
int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

results in the following assignments:

```
array[0][0] is equal to 1
array[0][1] is equal to 2
array[0][2] is equal to 3
array[1][0] is equal to 4
array[1][1] is equal to 5
array[1][2] is equal to 6
...
array[3][1] is equal to 11
array[3][2] is equal to 12
```

When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one just given:

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,
{ 7, 8, 9 } , { 10, 11, 12 } };
```

Remember, initialization values must be separated by a comma--even when there is a brace between them. Also, be sure to use braces in pairs--a closing brace for every opening brace--or the compiler becomes confused.

Now look at an example that demonstrates the advantages of arrays. Listing 8.3, RANDOM.C, creates a 1,000-element, three-dimensional array and fills it with random numbers. The program then displays the array elements on-screen. Imagine how many lines of source code you would need to perform the same task with nonarray variables.

You see a new library function, getch(), in this program. The getch() function reads a single character from the keyboard. In Listing 8.3, getch() pauses the program until the user presses a key. The getch() function is covered in detail on Day 14.

Listing 8.3. RANDOM.C creates a multidimensional array.

```
1:  /* RANDOM.C - Demonstrates using a multidimensional array */
2:
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:  /* Declare a three-dimensional array with 1000 elements */
6:
7:  int random_array[10][10][10];
8:  int a, b, c;
9:
10: main()
11: {
12:    /* Fill the array with random numbers. The C library */
13:    /* function rand() returns a random number. Use one */
14:    /* for loop for each array subscript. */
15:
16:    for (a = 0; a < 10; a++)
17:    {
18:       for (b = 0; b < 10; b++)
19:       {
20:          for (c = 0; c < 10; c++)
21:          {
22:             random_array[a][b][c] = rand();
23:          }
24:       }
25:    }
26:
27:    /* Now display the array elements 10 at a time */
28:
29:    for (a = 0; a < 10; a++)
30:    {
31:       for (b = 0; b < 10; b++)
32:       {
33:          for (c = 0; c < 10; c++)
34:          {
35:             printf("\nrandom_array[%d][%d][%d] = ", a, b, c);
36:             printf("%d", random_array[a][b][c]);
```

```
37:             }
38:             printf("\nPress Enter to continue, CTRL-C to quit.");
39:
40:             getchar();
41:         }
42:     }
43:     return 0;
44: }       /* end of main() */
random_array[0][0][0] = 346
random_array[0][0][1] = 130
random_array[0][0][2] = 10982
random_array[0][0][3] = 1090
random_array[0][0][4] = 11656
random_array[0][0][5] = 7117
random_array[0][0][6] = 17595
random_array[0][0][7] = 6415
random_array[0][0][8] = 22948
random_array[0][0][9] = 31126
Press Enter to continue, CTRL-C to quit.
random_array[0][1][0] = 9004
random_array[0][1][1] = 14558
random_array[0][1][2] = 3571
random_array[0][1][3] = 22879
random_array[0][1][4] = 18492
random_array[0][1][5] = 1360
random_array[0][1][6] = 5412
random_array[0][1][7] = 26721
random_array[0][1][8] = 22463
random_array[0][1][9] = 25047
Press Enter to continue, CTRL-C to quit
...         ...
random_array[9][8][0] = 6287
random_array[9][8][1] = 26957
random_array[9][8][2] = 1530
random_array[9][8][3] = 14171
random_array[9][8][4] = 6951
random_array[9][8][5] = 213
random_array[9][8][6] = 14003
random_array[9][8][7] = 29736
random_array[9][8][8] = 15028
random_array[9][8][9] = 18968
Press Enter to continue, CTRL-C to quit.
random_array[9][9][0] = 28559
random_array[9][9][1] = 5268
random_array[9][9][2] = 20182
random_array[9][9][3] = 3633
random_array[9][9][4] = 24779
random_array[9][9][5] = 3024
random_array[9][9][6] = 10853
random_array[9][9][7] = 28205
random_array[9][9][8] = 8930
random_array[9][9][9] = 2873
Press Enter to continue, CTRL-C to quit.
```

ANALYSIS: On Day 6 you saw a program that used a nested for statement; this program has two nested for loops. Before you look at the for statements in detail, note that lines 7 and 8 declare four variables. The first is an array

named random_array, used to hold random numbers. random_array is a three-dimensional type int array that is 10-by-10-by-10, giving a total of 1,000 type int elements (10 * 10 * 10). Imagine coming up with 1,000 unique variable names if you couldn't use arrays! Line 8 then declares three variables, a, b, and c, used to control the for loops.

This program also includes the header file STDLIB.H (for standard library) on line 4. It is included to provide the prototype for the rand() function used on line 22.

The bulk of the program is contained in two nests of for statements. The first is in lines 16 through 25, and the second is in lines 29 through 42. Both for nests have the same structure. They work just like the loops in Listing 6.2, but they go one level deeper. In the first set of for statements, line 22 is executed repeatedly. Line 22 assigns the return value of a function, rand(), to an element of the random_array array, where rand() is a library function that returns a random number.

Going backward through the listing, you can see that line 20 changes variable c from 0 to 9. This loops through the farthest right subscript of the random_array array. Line 18 loops through b, the middle subscript of the random array. Each time b changes, it loops through all the c elements. Line 16 increments variable a, which loops through the farthest left subscript. Each time this subscript changes, it loops through all 10 values of subscript b, which in turn loop through all 10 values of c. This loop initializes every value in the random array to a random number.

Lines 29 through 42 contain the second nest of for statements. These work like the previous for statements, but this loop prints each of the values assigned previously. After 10 are displayed, line 38 prints a message and waits for Enter to be pressed. Line 40 takes care of the keypress using getchar(). If Enter hasn't been pressed, getchar() waits until it is. Run this program and watch the displayed values.