Chapter 6

Control Statements

In most of the C programs we have encountered so far, the instructions were executed in the same order in which they appeared within the program. Each instruction was executed once and only once. Programs of this type are unrealistically simple, since they do not include any logical control structures. Thus, these programs did not include tests to determine if certain conditions are true or false, they did not require the repeated execution of groups of statements, and they did not involve the execution of individual groups of statements on a selective basis. Most C programs that are of practical interest make extensive use of features such as these.

For example, a realistic C program may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as *branching*. There is also a special kind of branching, called *selection*, in which one group of statements is selected from several available groups. In addition, the program may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as *looping*. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true.

All of these operations can be carried out using the various control statements included in C. We will see how this is accomplished in this chapter. The use of these statements will open the door to programming problems that are much broader and more interesting than those considered earlier.

6.1 PRELIMINARIES

Before considering the detailed control statements available in C, let us review some concepts presented in Chaps. 2 and 3 that must be used in conjunction with these statements. Understanding these concepts is essential in order to proceed further.

First, we will need to form logical expressions that are either true or false. To do so, we can use the four *relational operators*, `<`, `<=`, `>`, `>=`, and the two *equality operators*, `==` and `!=` (see Sec. 3.3).

EXAMPLE 6.1 Several logical expressions are shown below.

```
count <= 100

sqrt(a+b+c) > 0.005

answer == 0

balance >= cutoff

ch1 < 'T'

letter != 'x'
```

The first four expressions involve numerical operands. Their meaning should be readily apparent.

In the fifth expression, `ch1` is assumed to be a char-type variable. This expression will be true if the character represented by `ch1` comes before `T` in the character set, i.e., if the numerical value used to encode the character is less than the numerical value used to encode the letter `T`.

The last expression makes use of the char-type variable `letter`. This expression will be true if the character represented by `letter` is something other than `x`.

In addition to the relational and equality operators, C contains two *logical connectives* (also called *logical operators*), && (AND) and || (OR), and the *unary negation operator* ! (see Sec. 3.3). The logical connectives are used to combine logical expressions, thus forming more complex expressions. The negation operator is used to reverse the meaning of a logical expression (e.g., from true to false).

EXAMPLE 6.2 Here are some logical expressions that illustrate the use of the logical connectives and the negation operator.

```
(count <= 100) && (ch1 != '*')

(balance < 1000.0) || (status == 'R')

(answer < 0) || ((answer > 5.0) && (answer <= 10.0))

!((pay >= 1000.0) && (status == 's'))
```

Note that `ch1` and `status` are assumed to be char-type variables in these examples. The remaining variables are assumed to be numeric (either integer or floating-point).

Since the relational and equality operators have a higher precedence than the logical operators, some of the parentheses are not needed in the above expressions (see Table 3-1 in Sec. 3.5). Thus, we could have written these expressions as

```
count <= 100 && ch1 != '*'

balance < 1000.0 || status == 'R'

answer < 0 || answer > 5.0 && answer <= 10.0

!(pay >= 1000.0 && status == 's')
```

It is a good idea, however, to include pairs of parentheses if there is any doubt about the operator precedences. This is particularly true of expressions that are relatively complicated, such as the third expression above.

The *conditional operator* ?: also makes use of an expression that is either true or false (see Sec. 3.5). An appropriate value is selected, depending on the outcome of this logical expression. This operator is equivalent to a simple *if - else* structure (see Sec. 6.6).

EXAMPLE 6.3 Suppose `status` is a char-type variable and `balance` is a floating-point variable. We wish to assign the character `C` (current) to `status` if `balance` has a value of zero, and `O` (overdue) if `balance` has a value that is greater than zero. This can be accomplished by writing

```
status = (balance == 0) ? 'C' : 'O'
```

Finally, recall that there are three different kinds of statements in C: *expression statements*, *compound statements* and *control statements* (see Sec. 2.8). An expression statement consists of an expression, followed by a semicolon (see Sec. 2.7). A compound statement consists of a sequence of two or more consecutive statements enclosed in braces ({ and }). The enclosed statements can be expression statements, other compound statements or control statements. Most control statements contain expression statements or compound statements, including embedded compound statements.

EXAMPLE 6.4 Here is an elementary compound statement which we have seen before, in Example 3.31.

```
{
    int lower, upper;

    lower = getchar();
    upper = toupper(lower);
    putchar(upper);
}
```

Here is a more complicated compound statement

```
{
    float sum = 0, sumsq = 0, sumsqrt = 0, x;

    scanf("%f", &x);
    while (x != 0)  {
          sum += x;
          sumsq += x*x;
          sumsqrt += sqrt(x);
          scanf("%f", &x);
    }
}
```

This last example contains one compound statement embedded within another.

The control statements presented within this chapter make extensive use of logical expressions and compound statements. *Assignment operators*, such as the one used in the above example (i.e., +=), will also be utilized.

6.2 BRANCHING: THE `if - else` STATEMENT

The `if - else` statement is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).

The `else` portion of the `if - else` statement is optional. Thus, in its simplest general form, the statement can be written as

```
if (expression) statement
```

The *expression* must be placed in parentheses, as shown. In this form, the *statement* will be executed only if the *expression* has a nonzero value (i.e., if *expression* is true). If the *expression* has a value of zero (i.e., if *expression* is false), then the *statement* will be ignored.

The *statement* can be either simple or compound. In practice, it is often a compound statement which may include other control statements.

EXAMPLE 6.5 Several representative `if` statements are shown below.

```
if (x < 0) printf("%f", x);

if (pastdue > 0)
   credit = 0;

if (x <= 3.0)   {
   y = 3 * pow(x, 2);
   printf("%f\n", y);
}

if ((balance < 1000.) || (status == 'R'))
   printf("%f", balance);

if ((a >= 0) && (b <= 5))   {
   xmid = (a + b) / 2;
   ymid = sqrt(xmid);
}
```

The first statement causes the value of the floating-point variable x to be printed (displayed) if its value is negative. In the second statement, a value of zero is assigned to `credit` if the value of `pastdue` exceeds zero. The third statement involves a compound statement, in which y is evaluated and then displayed if the value of x does not exceed 3. In the fourth statement we see a complex logical expression, which causes the value of `balance` to be displayed if its value is less than 1000 *or* if `status` has been assigned the character `'R'`.

The last statement involves both a complex logical expression and a compound statement. Thus, the variables `xmid` and `ymid` will both be assigned appropriate values if the current value of a is nonnegative *and* the current value of b does not exceed 5.

The general form of an `if` statement which includes the `else` clause is

`if (`*expression*`)` *statement 1* `else` *statement 2*

If the *expression* has a nonzero value (i.e., if *expression* is true), then *statement 1* will be executed. Otherwise (i.e., if *expression* is false), *statement 2* will be executed.

EXAMPLE 6.6 Here are several examples illustrating the full `if - else` statement.

```
if (status == 'S')
   tax = 0.20 * pay;
else
   tax = 0.14 * pay;

if (pastdue > 0)   {
   printf("account number %d is overdue", accountno);
   credit = 0;
}
else
   credit = 1000.0;

if (x <= 3)
   y = 3 * pow(x, 2);
else
   y = 2 * pow(x - 3), 2);
printf("%f\n", balance);

if (circle)  {
   scanf("%f", &radius);
   area = 3.14159 * radius * radius;
   printf("Area of circle = %f", area);
}
else  {
   scanf("%f %f", &length, &width);
   area = length * width;
   printf("Area of rectangle = %f", area);
}
```

In the first example the value of `tax` is determined in one of two possible ways, depending on the character that has been assigned to the variable `status`. Notice the semicolon at the end of each statement, particularly the first statement (`tax = 0.2 * pay;`). A more concise way to accomplish the same thing is to write

```
tax = (status == 'S') ? (0.20 * pay) : (0.14 * pay);
```

though this approach is not as clear.

The second example examines the past-due status of an account. If the value of `pastdue` exceeds zero, a message is displayed and the credit limit is set at zero; otherwise, the credit limit is set at 1000.0. In the third example, the value of y is computed differently, depending on whether or not the corresponding value of x exceeds 3.

The fourth example shows how an area can be calculated for either of two different geometric figures. If `circle` is assigned a nonzero value, the radius of a circle is read into the computer, the area is calculated and then displayed. If the value of `circle` is zero, however, then the length and width of a rectangle are read into the computer, the area is calculated and then displayed. In each case, the type of geometric figure is included in the label that accompanies the value of the area.

It is possible to *nest* (i.e., embed) `if - else` statements, one within another. There are several different forms that nested `if - else` statements can take. The most general form of two-layer nesting is

```
if e1 if e2 s1
      else s2
else  if e3 s3
      else s4
```

where *e1*, *e2* and *e3* represent logical expressions and *s1*, *s2*, *s3* and *s4* represent statements. Now, one complete `if - else` statement will be executed if *e1* is nonzero (true), and another complete `if - else` statement will be executed if *e1* is zero (false). It is, of course, possible that *s1*, *s2*, *s3* and *s4* will contain other `if - else` statements. We would then have multilayer nesting.

Some other forms of two-layer nesting are

```
if e1 s1
else if e2 s2

if e1 s1
else if e2 s2
     else s3

if e1 if e2 s1
      else s2
else s3

if e1 if e2 s1
      else s2
```

In the first three cases the association between the `else` clauses and their corresponding expressions is straightforward. In the last case, however, it is not clear which expression (*e1* or *e2*) is associated with the `else` clause. The answer is *e2*. The rule is that the `else` clause is always associated with the closest preceding unmatched (i.e., `else`-less) `if`. This is suggested by the indentation, though the indentation itself is not the deciding factor. Thus, the last example is equivalent to

```
if e1  {
   if e2 s1 else s2
}
```

If we wanted to associate the `else` clause with *e1* rather than *e2*, we could do so by writing

```
if e1  {
   if e2 s1
}
else s2
```

This type of nesting must be carried out carefully in order to avoid possible ambiguities.

In some situations it may be desirable to nest multiple `if - else` statements, in order to create a situation in which one of several different courses of action will be selected. For example, the general form of four nested `if - else` statements could be written as

```
if e1 s1
else if e2 s2
     else if e3 s3
          else if e4 s4
               else s5
```

When a logical expression is encountered whose value is nonzero (true), the corresponding statement will be executed and the remainder of the nested `if - else` statements will be bypassed. Thus, control will be transferred out of the entire nest once a true condition is encountered.

The final `else` clause will apply if none of the expressions is true. It can be used to provide a default condition or an error message.

EXAMPLE 6.7 Here is an illustration of three nested `if - else` statements.

```
if ((time >= 0.) && (time < 12.)) printf("Good Morning");
else if ((time >= 12.) && (time < 18.)) printf("Good Afternoon");
     else if ((time >= 18.) && (time < 24.)) printf("Good Evening");
          else printf("Time is out of range");
```

This example causes a different message to be displayed at various times of the day. Specifically, the message `Good Morning` will be displayed if `time` has a value between 0 and 12; `Good Afternoon` will be displayed if `time` has a value between 12 and 18; and `Good Evening` will be displayed if `time` has a value between 18 and 24. An error message (`Time is out of range`) will be displayed if the value of `time` is less than zero, or greater than or equal to 24.

6.3 LOOPING: THE `while` STATEMENT

The `while` statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.

The general form of the `while` statement is

```
while (expression) statement
```

The *`statement`* will be executed repeatedly, as long as the *`expression`* is true (i.e., as long *`expression`* has a nonzero value). This *`statement`* can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the *`expression`*, thus providing a stopping condition for the loop.

EXAMPLE 6.8 Consecutive Integer Quantities Suppose we want to display the consecutive digits 0, 1, 2, . . . , 9, with one digit on each line. This can be accomplished with the following program.

```
#include <stdio.h>

main()     /* display the integers 0 through 9 */

{
    int digit = 0;

    while (digit <= 9)  {
          printf("%d\n", digit);
          ++digit;
    }
}
```

Initially, `digit` is assigned a value of 0. The `while` loop then displays the current value of `digit`, increases its value by 1 and then repeats the cycle, until the value of `digit` exceeds 9. The net effect is that the body of the loop will be repeated 10 times, resulting in 10 consecutive lines of output. Each line will contain a successive integer value, beginning with 0 and ending with 9. Thus, when the program is executed, the following output will be generated.

```
0
1
2
3
4
5
6
7
8
9
```

This program can be written more concisely as

```
#include <stdio.h>

main()      /* display the integers 0 through 9 */
{
    int digit = 0;

    while (digit <= 9)
          printf("%d\n", digit++);
}
```

When executed, this program will generate the same output as the first program.

In some looping situations, the number of passes through the loop is known in advance. The previous example illustrates this type of loop. Sometimes, however, the number of passes through the loop is not known in advance. Rather, the looping action continues indefinitely, until the specified logical condition has been satisfied. The `while` statement is particularly well suited for this second type of loop.

EXAMPLE 6.9 Lowercase to Uppercase Text Conversion In this example we will read a line of lowercase text character-by-character and store the characters in a char-type array called `letter`. The program will continue reading input characters until an end-of-line (EOF) character has been read. The characters will then be converted to uppercase, using the library function `toupper`, and displayed.

Two separate `while` loops will be used. The first will read the text from the keyboard. Note that the number of passes through this loop is not known in advance. The second `while` loop will perform the conversion and write out the converted text. It will make a known number of passes, since the number of characters to be displayed will be determined by counting the number of passes through the first loop.

The complete program is shown below.

```
/* convert a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define EOL  '\n'

main()
{
    char letter[80];
    int tag, count = 0;
```

```
    /* read in the lowercase text */
    while ((letter[count] = getchar()) != EOL)  ++count;
    tag = count;

    /* display the uppercase text */
    count = 0;
    while (count < tag)  {
          putchar(toupper(letter[count]));
          ++count;
    }
}
```

Notice that `count` is initially assigned a value of zero. Its value increases by 1 during each pass through the first loop. The final value of `count`, at the conclusion of the first loop, is then assigned to `tag`. The value of tag determines the number of passes through the second loop.

The first `while` loop, i.e.,

```
while ((letter[count] = getchar()) != EOL)  ++count;
```

is written very concisely. This single-statement loop is equivalent to the following:

```
letter[count] = getchar();
while (letter[count] != EOL)  {
      count = count + 1;
      letter[count] = getchar();
}
```

This latter form will be more familiar to those readers experienced with other high-level programming languages, such as Pascal or BASIC. Either form is correct, though the original form is more representative of typical C programming style.

When the program is executed, any line of text entered into the computer will be displayed in uppercase. Suppose, for example, that the following line of text had been entered:

```
Fourscore and seven years ago our fathers brought forth . . .
```

The computer would respond by printing

```
FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH . . .
```

EXAMPLE 6.10 Averaging a List of Numbers Let us now use a `while` statement to calculate the average of a list of `n` numbers. Our strategy will be based on the use of a partial sum that is initially set equal to zero, then updated as each new number is read into the computer. Thus, the problem very naturally lends itself to the use of a `while` loop.

The calculations will be carried out in the following manner.

1. Assign a value of 1 to the integer variable `count`. This variable will be used as a loop counter.
2. Assign a value of 0 to the floating-point variable `sum`.
3. Read in the value for the integer variable `n`.
4. Carry out the following steps repeatedly, as long as `count` does not exceed `n`.
   (*a*) Read in one of the numbers in the list. Each number will be represented by the floating-point variable `x`.
   (*b*) Add the value of `x` to the current value of `sum`.
   (*c*) Increase the value of `count` by 1.
5. Divide the value of `sum` by `n` to obtain the desired average.
6. Write out the calculated value for the average.

Here is the actual C program. Notice that the input operations are all accompanied by prompts that ask the user for the required information.

```
/* calculate the average of n numbers */

#include <stdio.h>

main()

{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);

    /* read in the numbers */
    while (count <= n)  {
          printf("x = ");
          scanf("%f", &x);
          sum += x;
          ++count;
    }

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

Notice that the `while` loop contains a compound statement which, among other things, causes the value of `count` to increase. Eventually, this will cause the logical expression

```
count <= n
```

to become false, thus terminating the loop. Also, note that the loop will not be executed at all if `n` is assigned a value that is less than 1 (which, of course, would make no sense).

Now suppose that the program will be used to process the following six values: 1, 2, 3, 4, 5, 6. Execution of the program will produce the following interactive dialog. (Note that the user's responses have been underlined.)

```
How many numbers? 6
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6

The average is 3.500000
```

6.4 MORE LOOPING: THE `do - while` STATEMENT

When a loop is constructed using the `while` statement described in Sec. 6.3, the test for continuation of the loop is carried out at the *beginning* of each pass. Sometimes, however, it is desirable to have a loop with the test for continuation at the *end* of each pass. This can be accomplished by means of the `do - while` statement.

The general form of the `do - while` statement is

```
do statement while (expression);
```

The *statement* will be executed repeatedly, as long as the value of *expression* is true (i.e., is nonzero). Notice that *statement* will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The *statement* can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature that eventually alters the value of *expression* so the looping action can terminate.

For many applications it is more natural to test for continuation of a loop at the beginning rather than at the end of the loop. For this reason, the do - while statement is used less frequently than the while statement described in Sec. 6.3. For illustrative purposes, however, the programming examples shown in Sec. 6.3 are repeated below using the do - while statement for the conditional loops.

EXAMPLE 6.11 Consecutive Integer Quantities In Example 6.8 we saw two complete C programs that use the while statement to display the consecutive digits 0, 1, 2, . . . , 9. Here is another program to do the same thing, using the do - while statement in place of the while statement.

```
#include <stdio.h>

main()    /* display the integers 0 through 9 */
{
    int digit = 0;

    do
    printf("%d\n", digit++);
    while (digit <= 9);
}
```

As in the earlier example, digit is initially assigned a value of 0. The do - while loop displays the current value of digit, increases its value by 1, and then tests to see if the current value of digit exceeds 9. If so, the loop terminates; otherwise, the loop continues, using the new value of digit. Note that the test is carried out at the end of each pass through the loop. The net effect is that the loop will be repeated 10 times, resulting in 10 successive lines of output. Each line will appear exactly as shown in Example 6.8.

Comparing this program with the second program presented in Example 6.8, we see about the same level of complexity in both programs. Neither of the conditional looping structures (i.e., while or do - while) appears more desirable than the other.

EXAMPLE 6.12 Lowercase to Uppercase Text Conversion Now let us rewrite the program shown in Example 6.9, which converts lowercase text to uppercase, so that the two while loops are replaced by do - while loops. As in the earlier program, our overall strategy will be to read in a line of lowercase text on a character-by-character basis, store the characters in a char-type array called letter, and then write them out in uppercase using the library function toupper. We will make use of a do - while statement to read in the text on a character-by-character basis, and another do - while statement to convert the characters to uppercase and then write them out.

Here is the complete C program.

```
/* convert a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define EOL .'\n'

main()
{
    char letter[80];
    int tag, count = -1;

    /* read in the lowercase text */
    do ++count; while ((letter[count] = getchar()) != EOL);
    tag = count;
```

```
    /* display the uppercase text */
    count = 0;
    do {
          putchar(toupper(letter[count]));
          ++count;
    }  while (count < tag);
}
```

We again see two different types of loops, even though they are both written as do - while loops. In particular, the number of passes through the first loop will not be known in advance, but the second loop will execute a known number of passes, as determined by the value assigned to tag.

Notice that the first loop, i.e.,

```
do ++count; while ((letter[count] = getchar()) != EOL);
```

is simple and concise, but the second loop,

```
do {
      putchar(toupper(letter[count]));
      ++count;
}  while (count < tag);
```

is somewhat more complex. Both loops resemble the corresponding while loops presented in Example 6.9. Note, however, that the first loop in the present program begins with a value of −1 assigned to count, whereas the initial value of count was 0 in Example 6.9.

When the program is executed, it behaves in exactly the same way as the program shown in Example 6.9.

Before leaving this example, we mention that the last loop could have been written more concisely as

```
do
    putchar(toupper(letter[count++]));
while (count < tag);
```

This may appear a bit strange to beginners, though it is characteristic of the programming style that is commonly used by experienced C programmers.

EXAMPLE 6.13 Averaging a List of Numbers The program shown in Example 6.10 can easily be rewritten to illustrate the use of the do - while statement. The logic will be the same, except that the test to determine if all n numbers have been entered into the computer will not be made until the end of the loop rather than the beginning. Thus the program will always make at least one pass through the loop, even if n is assigned a value of 0 (which would make no sense).

Here is the modified version of the program.

```
/* calculate the average of n numbers */

#include <stdio.h>

main()
{
    int n, count = 1;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);
```

```
    /* read in the numbers */
    do {
          printf("x = ");
          scanf("%f", &x);
          sum += x;
          ++count;
    }  while (count <= n);

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

When the program is executed it will behave exactly the same way as the earlier version shown in Example 6.10.

6.5 STILL MORE LOOPING: THE `for` STATEMENT

The `for` statement is the third and perhaps the most commonly used looping statement in C. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of the `for` statement is

```
for (expression 1; expression 2; expression 3) statement
```

where *expression 1* is used to initialize some parameter (called an *index*) that controls the looping action, *expression 2* represents a condition that must be true for the loop to continue execution, and *expression 3* is used to alter the value of the parameter initially assigned by *expression 1*. Typically, *expression 1* is an assignment expression, *expression 2* is a logical expression and *expression 3* is a unary expression or an assignment expression.

When the `for` statement is executed, *expression 2* is evaluated and tested at the *beginning* of each pass through the loop, and *expression 3* is evaluated at the *end* of each pass. Thus, the `for` statement is equivalent to

```
expression 1;
while (expression 2)  {
   statement
   expression 3;
}
```

The looping action will continue as long as the value of *expression 2* is not zero, that is, as long as the logical condition represented by *expression 2* is true.

The `for` statement, like the `while` and the `do - while` statements, can be used to carry out looping actions where the number of passes through the loop is not known in advance. Because of the features that are built into the `for` statement, however, it is particularly well suited for loops in which the number of passes *is* known in advance. As a rough rule of thumb, `while` loops are generally used when the number of passes is *not* known in advance, and `for` loops are generally used when the number of passes *is* known in advance.

EXAMPLE 6.14 Consecutive Integer Quantities We have already seen several different versions of a C program that will display the consecutive digits 0, 1, 2, . . . , 9, with one digit on each line (see Examples 6.8 and 6.11). Here is another program which does the same thing. Now, however, we will make use of the `for` statement rather than the `while` statement or the `do - while` statement, as in the earlier examples.

```
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit;

    for (digit = 0; digit <= 9; ++digit)
        printf("%d\n", digit);
}
```

The first line of the `for` statement contains three expressions, enclosed in parentheses. The first expression assigns an initial value 0 to the integer variable `digit`; the second expression continues the looping action as long as the current value of `digit` does not exceed 9 at the *beginning* of each pass; and the third expression increases the value of `digit` by 1 at the *end* of each pass through the loop. The `printf` function, which is included in the `for` loop, produces the desired output, as shown in Example 6.8.

From a syntactic standpoint all three expressions need not be included in the `for` statement, though the semicolons must be present. However, the consequences of an omission should be clearly understood. The first and third expressions may be omitted if other means are provided for initializing the index and/or altering the index. If the second expression is omitted, however, it will be assumed to have a permanent value of 1 (true); thus, the loop will continue indefinitely unless it is terminated by some other means, such as a `break` or a `return` statement (see Secs. 6.8 and 7.2). As a practical matter, most `for` loops include all three expressions.

EXAMPLE 6.15 Consecutive Integer Quantities Revisited Here is still another example of a C program that generates the consecutive integers 0, 1, 2, . . . , 9, with one digit on each line. We now use a `for` statement in which two of the three expressions are omitted.

```
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit = 0;

    for (; digit <= 9; )
        printf("%d\n", digit++);
}
```

This version of the program is more obscure than that shown in Example 6.14, and hence less desirable.

Note the similarity between this program and the second program in Example 6.8, which makes use of a `while` loop.

EXAMPLE 6.16 Lowercase to Uppercase Text Conversion Here once again is a C program that converts lowercase text to uppercase. We have already seen other programs that do this, in Examples 6.9 and 6.12. Now, however, we make use of a `for` loop rather than a `while` loop or a `do - while` loop.

As before, our overall strategy will be to read in a line of lowercase text on a character-by-character basis, store the characters in a char-type array called `letter`, and then write them out in uppercase using the library function `toupper`. Two separate loops will be required: one to read and store the lowercase characters, the other to display the characters in uppercase. Note that we will now use a `for` statement to build a loop in which the number of passes is not known in advance.

Here is the complete C program.

```
/* convert a line of lowercase text to uppercase */

#include <stdio.h>
#include <ctype.h>

#define EOL  '\n'

main()

{
    char letter[80];
    int tag, count;

    /* read in the lowercase text */
    for (count = 0; (letter[count] = getchar()) != EOL; ++count)
    ;
    tag = count;

    /* display the uppercase text */
    for (count = 0; count < tag; ++count)
        putchar(toupper(letter[count]));
}
```

Comparing this program with the corresponding programs given in Examples 6.9 and 6.12, we see that the loops can be written more concisely using the `for` statement than with `while` or `do - while` statements.

EXAMPLE 6.17 Averaging a List of Numbers Now let us modify the program given in Example 6.10, which calculates the average of a list of n numbers, so that the looping action is accomplished by means of a `for` statement. The logic will be essentially the same, though some of the steps will be carried out in a slightly different order. In particular:

1. Assign a value of 0 to the floating-point variable `sum`.
2. Read in a value for the integer variable `n`.
3. Assign a value of 1 to the integer variable `count`, where `count` is an index that counts the number of passes through the loop.
4. Carry out the following steps repeatedly, as long as the value of `count` does not exceed `n`.
   (*a*) Read in one of the numbers in the list. Each number will be represented by the floating-point variable `x`.
   (*b*) Add the value of `x` to the current value of `sum`.
   (*c*) Increase the value of `count` by 1.
5. Divide the value of `sum` by `n` to obtain the desired average.
6. Write out the calculated value for the average.

Here is the complete C program. Notice that steps 3 and 4 are combined in the `for` statement, and that steps 3 and 4(*c*) are both carried out in the first line (first and third expressions, respectively). Also, notice that the input operations are all accompanied by prompts that ask the user for the desired information.

```
/* calculate the average of n numbers */

#include <stdio.h>

main()

{
    int n, count;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);
```

```
    /* read in the numbers */
    for (count = 1; count <= n; ++count)  {
        printf("x = ");
        scanf("%f", &x);
        sum += x;
    }

    /* calculate the average and display the answer */
    average = sum/n;
    printf("\nThe average is %f\n", average);
}
```

Comparing this program to the corresponding programs shown in Examples 6.10 and 6.13, we again see a more concise loop specification when the `for` statement is used rather than `while` or `do - while`. Now, however, the `for` statement is somewhat more complex than in the preceding programming examples. In particular, notice that the *statement* part of the loop is now a compound statement. Moreover, we must assign an initial value to `sum` explicitly, before entering the `for` loop.

When the program is executed it will behave exactly as the earlier versions, presented in Examples 6.10 and 6.13.

6.6 NESTED CONTROL STRUCTURES

Loops, like `if - else` statements, can be *nested*, one within another. The inner and outer loops need not be generated by the same type of control structure. It is essential, however, that one loop be completely embedded within the other — there can be no overlap. Each loop must be controlled by a different index.

Moreover, nested control structures can involve both loops and `if - else` statements. Thus, a loop can be nested within an `if - else` statement, and an `if - else` statement can be nested within a loop. The nested structures may be as complex as necessary, as determined by the program logic.

EXAMPLE 6.18 Repeated Averaging of a List of Numbers Suppose we want to calculate the average of several consecutive lists of numbers. If we know in advance how many lists are to be averaged, then we can use a `for` statement to control the number of times that the inner (averaging) loop is executed. The actual averaging can be accomplished using any of the three methods presented earlier, in Examples 6.10, 6.13 and 6.17 (using a `while`, a `do - while`, or a `for` loop).

Let us arbitrarily use the `for` statement to carry out the averaging, as in Example 6.17. Thus, we will proceed in the following manner.

1. Read in a value of `loops`, an integer quantity that indicates the number of lists that will be averaged.
2. Repeatedly read in a list of numbers and determine its average. That is, calculate the average of a list of numbers for each successive value of `loopcount` ranging from 1 to `loops`. Follow the steps given in Example 6.14 to calculate each average.

Here is the actual C program.

```
/* calculate averages for several different lists of numbers */

#include <stdio.h>

main()

{
    int n, count, loops, loopcount;
    float x, average, sum;

    /* read in the number of lists */
    printf("How many lists? ");
    scanf("%d", &loops);
```

```
    /* outer loop (process each list of numbers */
    for (loopcount = 1; loopcount <= loops; ++loopcount)  {

        /* initialize and read in a value for n */
        sum = 0;
        printf("\nList number %d\nHow many numbers? ", loopcount);
        scanf("%d", &n);

        /* read in the numbers */
        for (count = 1; count <= n; ++count)  {
            printf("x = ");
            scanf("%f", &x);
            sum += x;
        }    /* end inner loop */

        /* calculate the average and display the answer */
        average = sum/n;
        printf("\nThe average is %f\n", average);

    }    /* end outer loop */
}
```

This program contains several interesting features. First, it contains two `for` statements, one embedded within the other. Each `for` statement includes a compound statement, consisting of several individual statements enclosed in braces. Also, a different index is used in each `for` statement (the indices are `loopcount` and `count`, respectively).

Note that `sum` must now be initialized within the outer loop, rather than within the declaration. This allows `sum` to be reset to zero each time a new set of data is encountered (i.e., at the beginning of each pass through the outer loop).

The input data operations are all accompanied by prompts, indicating to the user what data are required. Thus, we see pairs of `printf` and `scanf` functions at several places throughout the program. Two of the `printf` functions contain multiple `newline` characters, to control the line spacing of the output. This causes the output associated with each set of data (each pass through the outer loop) to be easily identified.

Finally, note that the program is organized into separate identifiable segments, with each segment preceded by a blank space and a comment.

When the program is executed using three simple sets of data, the following dialog is generated. As usual, the user's responses to the input prompts have been underlined.

```
How many lists? 3

List number 1
How many numbers? 4
x = 1.5
x = 2.5
x = 6.2
x = 3.0

The average is 3.300000

List number 2
How many numbers? 3
x = 4
x = -2
x = 7

The average is 3.000000
```

```
List number 3
How many numbers? 5
x = 5.4
x = 8.0
x = 2.2
x = 1.7
x = -3.9

The average is 2.680000
```

EXAMPLE 6.19 Converting Several Lines of Text to Uppercase This example illustrates the use of two different types of loops, one nested within the other. Let us extend the lowercase to uppercase conversion programs presented in Examples 6.9, 6.12 and 6.16 so that multiple lines of lowercase text can be converted to uppercase, with the conversion taking place one line at a time. In other words, we will read in a line of lowercase text, display it in uppercase, then process another line, and so on. The procedure will continue until a line is detected in which the first character is an asterisk.

We will use nested loops to carry out the computation. The outer loop will be used to process multiple lines of text. Two separate inner loops will be embedded within the outer loop. The first will read in a line of text, and the second will display the converted uppercase text. Note that these inner loops are not nested. Let us arbitrarily utilize a `while` statement for the outer loop, and a `for` statement for each of the inner loops.

In general terms, the computation will proceed as follows.

1. Assign an initial value of 1 to the outer loop index (`linecount`).
2. Carry out the following steps repeatedly, for each successive line of text, as long as the first character in the line is not an asterisk.
   (*a*) Read in a line of text and assign the individual characters to the elements of the char-type array `letter`. A line will be defined as a succession of characters that is terminated by an end-of-line (newline) designation.
   (*b*) Assign the character count (including the end-of-line character) to `tag`.
   (*c*) Display the line in uppercase, using the library function `toupper` to carry out the conversion. Then write out two newline characters so that the next line of input will be separated from the current output by a blank line, and increment the line counter (`linecount`).
3. Once an asterisk has been detected as the first character of a new line, write out `Good bye` and terminate the computation.

Here is the complete C program.

```
/* convert several lines of text to uppercase
   continue the conversion until the first character in a line is an asterisk (*) */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()
{
    char letter[80];
    int tag, count;

    while((letter[0] = getchar()) != '*')   {

        /* read in a line of text */
        for (count = 1; (letter[count] = getchar()) != EOL; ++count)
            ;
        tag = count;
```

```
        /* display the line of text */
        for (count = 0; count < tag; ++count)
            putchar(toupper(letter[count]));
        printf("\n\n");
}    /* end outer loop */

    printf("Good bye");
}
```

A typical interactive session, illustrating the execution of the program, is shown below. Note that the input text supplied by the user is underlined, as usual.

```
Now is the time for all good men to come to the aid . . .
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID . . .

Fourscore and seven years ago our fathers brought forth . . .
FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH . . .

*
Good bye
```

It should be understood that the decision to use a `while` statement for the outer loop and `for` statements for the inner loops is arbitrary. Other loop structures could also have been selected.

Many programs involve both looping and branching. The various control structures are often nested, one within another, as illustrated in the following three examples.

EXAMPLE 6.20 Encoding a String of Characters Let us write a simple C program that will read in a sequence of ASCII characters and write out a sequence of encoded characters in its place. If a character is a letter or a digit, we will replace it with the next character in the character set, except that `Z` will be replaced by `A`, `z` by `a`, and `9` by `0`. Thus `1` becomes `2`, `C` becomes `D`, `p` becomes `q`, and so on. Any character other than a letter or a digit will be replaced by a period (.).

The computation will begin by reading in the characters. The `scanf` function will be used for this purpose. All the characters, up to but not including the newline (`\n`) character that is used to terminate the input, will be entered and stored in an 80-element, character-type array called `line`.

The characters will then be encoded and displayed individually within a `for` loop. The loop will process each of the characters in `line`, until the escape character `\0`, which designates the end of the character sequence, is encountered. (Recall that the escape sequence `\0` is automatically added at the end of each string.) Several nested `if - else` statements will be included within the loop, to carry out the appropriate encoding. Each encoded character will then be displayed using the `putchar` function.

The complete C program is shown below.

```
/* read in a string, then replace each character with an equivalent encoded character */

#include <stdio.h>

main()

{
    char line[80];
    int count;

    /* read in the entire string */

    printf("Enter a line of text below:\n");
    scanf("%[^\n]", line);
```

```
    /* encode each individual character and display it */

    for (count = 0; line[count] != '\0'; ++count)  {
        if (((line[count] >= '0') && (line[count] < '9')) ||
            ((line[count] >= 'A') && (line[count] < 'Z')) ||
            ((line[count] >= 'a') && (line[count] < 'z')))
                putchar(line[count] + 1);
        else if (line[count] == '9') putchar('0');
             else if (line[count] == 'Z') putchar('A');
                  else if (line[count] == 'z') putchar('a');
                       else putchar('.');
    }
}
```

Execution of this program generates the following representative dialog. The input provided by the user is again underlined.

```
Enter a line of text below:
The White House, 1600 Pennsylvania Avenue, Washington, DC
Uif.Xijuf.Ipvtf..2711.Qfootzmwbojb.Bwfovf..Xbtijohupo..ED
```

EXAMPLE 6.21 Repeated Compound Interest Calculations with Error Trapping In Example 5.2 we saw a complete C program to carry out simple compound interest calculations, as outlined in Example 5.1. However, the program in Example 5.2 did not allow for repetitive execution (i.e., for several successive calculations, using different input data for each calculation), nor did it attempt to detect errors in the input data. Let us now add these features to the earlier program.

In particular, let us embed the earlier calculations within a `while` statement, which will continue to execute as long as the value entered for the principal (`P`) is positive. Thus, a zero value for `P` will be interpreted as a stopping condition. We will include a message explaining the stopping condition when prompting for the value of `P`.

In addition, let us include an *error trap* that will test the value of each input quantity to determine if it is negative, since a negative value would not make any sense and should be interpreted as an error. Each test will be carried out with a separate `if` statement. If an error (i.e., a negative value) is detected, a message will be written asking the user to reenter the data.

Here is the entire C program.

```
/* simple compound interest problem */

#include <stdio.h>
#include <math.h>

main()

{
    float p,r,n,i,f;

    /* read initial value for the principal */

    printf("Please enter a value for the principal (P) ");
    printf("\n(To end program, enter 0 for the principal): ");
    scanf("%f", &p);
    if (p < 0)   {
       printf("\nERROR - Please try again: ");
       scanf("%f", &p);
    }
```

```
    while (p > 0)   {     /* main loop */

       /* read remaining input data */

       printf("\nPlease enter a value for the interest rate (r): ");
       scanf("%f", &r);
       if (r < 0)   {
          printf("\nERROR - Please try again: ");
          scanf("%f", &r);
       }
       printf("\nPlease enter a value for the number of years (n): ");
       scanf("%f", &n);
       if (n < 0)   {
          printf("\nERROR - Please try again: ");
          scanf("%f", &n);
       }

       /* calculate i, then f */

       i = r/100;
       f = p * pow((1 + i), n);

       /* display the output */

       printf("\nThe final value (F) is: %.2f\n", f);

       /* read principal for next pass */

       printf("\n\nPlease enter a value for the principal (P) ");
       printf("\n(To end program, enter 0 for the principal): ");
       scanf("%f", &p);
       if (p < 0)   {
          printf("\nERROR - Please try again: ");
          scanf("%f", &p);
       }
    }   /* end while loop */
}
```

A typical interactive session is shown below. Note that the user's responses are underlined.

```
Please enter a value for the principal (P)
(To end program, enter 0 for the principal): 1000

Please enter a value for the interest rate (r): 6

Please enter a value for the number of years (n): 20

The final value (F) is: 3207.14


Please enter a value for the principal (P)
(To end program, enter 0 for the principal): 5000

Please enter a value for the interest rate (r): -7.5

ERROR - Please try again: 7.5
```

```
Please enter a value for the number of years (n): 12

The final value (F) is: 11908.90


Please enter a value for the principal (P)
(To end program, enter 0 for the principal): 0
```

Notice that two sets of input data are provided. The first set of data is entered correctly, resulting in a calculated future value of 3207.14 (as in Example 5.4). In the second data set, a negative value is initially supplied for the interest rate (r). This is detected as an error, resulting in an error message and a request for another value. Once the corrected value is supplied, the remaining program execution proceeds as expected.

After the second data set has been processed, the user enters a value of 0 for the principal, in response to the prompt. This causes the execution of the program to terminate.

Remember that the error trapping used in this program applies only to negative floating-point quantities entered as input data. Another type of error occurs if a letter or punctuation mark is entered for one of the required input quantities. This will produce a type mismatch in the `scanf` function, resulting in an input error. Individual compilers deal with this type of error differently, thus preventing a simple, general error trap.

The following program is more comprehensive in nature. It includes most of the programming features that we have encountered earlier in this book.

EXAMPLE 6.22 Solution of an Algebraic Equation For the more mathematically inclined reader, this example illustrates how computers can be used to solve algebraic equations, including those that cannot be solved by more direct methods. Consider, for example, the equation

$$x^5 + 3x^2 - 10 = 0.$$

This equation cannot be rearranged to yield an exact solution for x. However, we can determine the solution by a repeated trial-and-error procedure (called an *iterative* procedure) that successively refines an initial guess.

We begin by rearranging the equation into the form

$$x = (10 - 3x^2)^{1/5}$$

Our procedure will then be to guess a value for x, substitute this value into the right-hand side of the rearranged equation, and thus calculate a new value for x. If this new value is equal (or very nearly equal) to the old value, then we will have obtained a solution to the equation. Otherwise, this new value will be substituted into the right-hand side and still another value obtained for x, and so on. This procedure will continue until either the successive values of x have become sufficiently close (i.e., until the computation has *converged*), or until a specified number of iterations has been exceeded. This last condition prevents the computation from continuing indefinitely in the event that the computed results do not converge.

To see how the method works, suppose we choose an initial value of $x = 1.0$. Substituting this value into the right-hand side of the equation, we obtain

$$x = [10 - 3(1.0)^2]^{0.2} = 1.47577$$

We then substitute this new value of x into the equation, resulting in

$$x = [10 - 3(1.47577)^2]^{0.2} = 1.28225$$

Continuing this procedure, we obtain

$$x = [10 - 3(1.28225)^2]^{0.2} = 1.38344$$

$$x = [10 - 3(1.38344)^2]^{0.2} = 1.33613$$

and so on. Notice that the successive values of x appear to be converging to some final answer.

The success of the method depends on the value chosen for the initial guess. If this value is too large in magnitude, then the quantity in brackets will be negative, and a negative value cannot be raised to a fractional power. Therefore we should test for a negative value of $10 - 3x^2$ whenever we substitute a new value of x into the right-hand side.

In order to write a program outline, let us define the following symbols.

`count` = an iteration counter (`count` will increase by 1 at each successive iteration)

`guess` = the value of x substituted into the right-hand side of the equation

`root` = the newly calculated value of x

`test` = the quantity $(10 - 3x^2)$

`error` = the absolute difference between `root` and `guess`

`flag` = an integer variable that signifies whether or not to continue the iteration

We will continue the computation until one of the following conditions is satisfied.

1. The value of `error` becomes less than 0.00001, in which case we have obtained a satisfactory solution.
2. Fifty iterations have been completed (i.e., `count` = 50).
3. The variable `test` takes on a negative value, in which case the computation cannot be continued.

Let us monitor the progress of the computation by writing out each successive value of `root`.

We can now write the following program outline.

1. For convenience, define the symbolic constants `TRUE` and `FALSE`.
2. Declare all variables, and initialize the integer variables `flag` and `count` (assign `TRUE` to `flag` and 0 to `count`).
3. Read in a value for the initial `guess`.
4. Carry out the following looping procedure, while `flag` remains `TRUE`.
   (*a*) Increase the value of `count` by 1.
   (*b*) Assign `FALSE` to `flag` if the new value of `count` equals 50. This will signify the last pass through the loop.
   (*c*) Examine the value of `test`. If its value is positive, proceed as follows.
      (*i*) Calculate a new value for `root`; then write out the current value for `count`, followed by the current value for `root`.
      (*ii*) Evaluate `error`, which is the absolute value of the difference between `root` and `guess`. If this value is greater than 0.00001, assign the current value of `root` to `guess` and proceed with another iteration. Otherwise write out the current values of `root` and `count`, and set `flag` to `FALSE`. The current value of `root` will be considered to be the desired solution.
   (*d*) If the current value of `test` is not positive, then the computation cannot proceed. Hence, write an appropriate error message (e.g., `Numbers out of range`) and set `flag` to `FALSE`.
5. Upon completion of step 4, write an appropriate error message (e.g., `Convergence not obtained`) if `count` has a value of 50 and the value of `error` is greater than 0.00001.

Now let us express the program outline in the form of pseudocode, in order to simplify the transition from a general outline to a working C program.

```
#include files

#define symbolic constants

main()

{
    /* variable declarations and initialization */

    /* read input parameters */
```

```
    while (flag)  {

          /* increment count */

          /* flag becomes FALSE if count = 50 */

          /* evaluate test */

          if (test > 0)  {
             /* evaluate root */
             /* display count and loop */
             /* evaluate error */

             if (error > 0.00001) guess = root;
             else  {
                /* flag becomes FALSE */
                /* display the final answer (root and count) */
             }
          }

          else  {
             /* flag becomes FALSE */
             /* numbers out of range - write error message */
          }

    }   /* end while */

    if ((count == 50) && (error > 0.00001))
       /* convergence not obtained - write error message */
}
```

Here is the complete C program.

```
/* determine the roots of an algebraic equation using an iterative procedure */

#include <stdio.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

main()

{
    int flag = TRUE, count = 0;
    float guess, root, test, error;

    /* read input parameters */

    printf("Initial guess: ");
    scanf("%f", &guess);
    while (flag)   {                                  /* begin the main loop */
          ++count;
          if (count == 50) flag = FALSE;
          test = 10. - 3. * guess * guess;
          if (test > 0)   {                           /* another iteration */
             root = pow(test, 0.2);
             printf("\nIteration number: %2d", count);
             printf("    x= %7.5f", root);
             error = fabs(root - guess);
```

```
            if (error > 0.00001) guess = root;        /* repeat the calculation */
            else   {                                  /* display the final answer */
               flag = FALSE;
               printf("\n\nRoot= %7.5f", root);
               printf("    No. of iterations= %2d", count);
            }
         }
         else   {                                     /* error message */
            flag = FALSE;
            printf("\nNumbers out of range - try another initial guess");
         }
   }
   if ((count == 50) && (error > 0.00001))            /* another error message */
      printf("\n\nConvergence not obtained after 50 iterations");
}
```

Notice that the program contains a `while` statement and several `if - else` statements. A `for` statement could easily have been used instead of the `while` statement. Also, notice the nested `if - else` statements near the middle of the program.

The output that is generated for an initial guess of `x = 1` is shown below, with the user's responses underlined. Notice that the computation has converged to the solution `x = 1.35195` after 16 iterations. The printed output shows the successive values of x becoming closer and closer, leading to the final solution.

```
Initial guess: 1

Iteration number:  1   x=    1.47577
Iteration number:  2   x=    1.28225
Iteration number:  3   x=    1.38344
Iteration number:  4   x=    1.33613
Iteration number:  5   x=    1.35951
Iteration number:  6   x=    1.34826
Iteration number:  7   x=    1.35375
Iteration number:  8   x=    1.35109
Iteration number:  9   x=    1.35238
Iteration number: 10   x=    1.35175
Iteration number: 11   x=    1.35206
Iteration number: 12   x=    1.35191
Iteration number: 13   x=    1.35198
Iteration number: 14   x=    1.35196
Iteration number: 15   x=    1.35196
Iteration number: 16   x=    1.35195

Root= 1.35195    No. of iterations= 16
```

Now suppose that a value of `x = 10` had been selected as an initial guess. This value generates a negative number for `test` in the first iteration. Therefore the output would appear as follows.

```
Initial guess: 10
Numbers out of range - try another initial guess
```

It is interesting to see what happens when the initial guess is once again chosen as `x = 1`, but the maximum number of iterations is changed from 50 to 10. You are encouraged to try this and observe the result.

you should underdstand that there are many other iterative methods for solving algebraic equations. Most converge faster than the method described above (i.e., they require fewer iterations to obtain a solution), though the mathematics is more complicated.

6.7 THE switch STATEMENT

The switch statement causes a particular group of statements to be chosen from several available groups. The selection is based upon the current value of an expression which is included within the switch statement.

The general form of the switch statement is

```
switch (expression) statement
```

where *expression* results in an integer value. Note that *expression* may also be of type char, since individual characters have equivalent integer values.

The embedded *statement* is generally a compound statement that specifies alternate courses of action. Each alternative is expressed as a group of one or more individual statements within the overall embedded *statement*.

For each alternative, the first statement within the group must be preceded by one or more *case labels* (also called *case prefixes*). The case labels identify the different groups of statements (i.e., the different alternatives) and distinguish then from one another. The case labels must therefore be unique within a given switch statement.

In general terms, each group of statements is written as

```
case expression :
    statement 1
    statement 2
    . . . . .
    statement n
```

or, when multiple case labels are required,

```
case expression 1 :
case expression 2 :
    . . . . .
case expression m :
    statement 1
    statement 2
    . . . . .
    statement n
```

where *expression 1*, *expression 2*, . . . , *expression m* represent constant, integer-valued expressions. Usually, each of these expressions will be written as either an integer constant or a character constant. Each individual *statement* following the case labels may be either simple or complex.

When the switch statement is executed, the *expression* is evaluated and control is transferred directly to the group of statements whose case-label value matches the value of the *expression*. If none of the case-label values matches the value of the *expression*, then none of the groups within the switch statement will be selected. In this case control is transferred directly to the statement that follows the switch statement.

EXAMPLE 6.23 A simple switch statement is illustrated below. In this example, choice is assumed to be a char-type variable.

```
switch (choice = getchar())  {

case 'r':
case 'R':
     printf("RED");
     break;
```

```
case 'w':
case 'W':
     printf("WHITE");
     break;

case 'b':
case 'B':
     printf("BLUE");
}
```

Thus, RED will be displayed if choice represents either r or R, WHITE will be displayed if choice represents either w or W, and BLUE will be displayed if choice represents either b or B. Nothing will be displayed if any other character has been assigned to choice.

Notice that each group of statements has two case labels, to account for either upper or lowercase. Also, note that each of the first two groups ends with the break statement (see Sec. 6.8). The break statement causes control to be transferred out of the switch statement, thus preventing more than one group of statements from being executed.

One of the labeled groups of statements within the switch statement may be labeled default. This group will be selected if none of the case labels matches the value of the *expression*. (This is a convenient way to generate error messages or error correction routines.) The default group may appear anywhere within the switch statement—it need not necessarily be placed at the end. If none of the case labels matches the value of the *expression* and the default group is not present (as in the above example), then no action will be taken by the switch statement.

EXAMPLE 6.24 Here is a variation of the switch statement presented in Example 6.23.

```
switch (choice = toupper(getchar()))  {

case 'R':
     printf("RED");
     break;

case 'W':
     printf("WHITE");
     break;

case 'B':
     printf("BLUE");
     break;

default:
     printf("ERROR");
}
```

The switch statement now contains a default group (consisting of only one statement), which generates an error message if none of the case labels matches the original *expression*.

Each of the first three groups of statements now has only one case label. Multiple case labels are not necessary in this example, since the library function toupper causes all incoming characters to be converted to uppercase. Hence, choice will always be assigned an uppercase character.

EXAMPLE 6.25 Here is another typical switch statement. In this example flag is assumed to be an integer variable, and x and y are assumed to be floating-point variables.

```
switch (flag)  {

case -1:
     y = abs(x);
     break;

case 0:
     y = sqrt(x);
     break;

case 1:
     y = x;
     break;

case 2:
case 3:
     y = 2 * (x - 1);
     break;

default:
     y = 0;
}
```

In this example y will be assigned some value that is related to the value of x if `flag` equals −1, 0, 1, 2 or 3. The exact relationship between y and x will depend upon the particular value of `flag`. If `flag` represents some other value, however, then y will be assigned a value of 0.

Notice that the case labels are numeric in this example. Also, note that the third group of statements has two case labels, whereas each of the other groups have only one case label. And finally, notice that a default group (consisting of only one statement) is included within this `switch` statement.

In a practical sense, the `switch` statement may be thought of as an alternative to the use of nested `if - else` statements, though it can only replace those `if - else` statements that test for equality. In such situations, the use of the `switch` statement is generally much more convenient.

EXAMPLE 6.26 Calculating Depreciation Let us consider how to calculate the yearly depreciation for some depreciable item, such as a building, a machine, etc. There are three commonly used methods for calculating depreciation, known as the *straight-line* method, the *double-declining-balance* method, and the *sum-of-the-years'-digits* method. We wish to write a C program that will allow us to select any one of these methods for each set of calculations.

The computation will begin by reading in the original (undepreciated) value of the item, the life of the item (i.e., the number of years over which it will be depreciated) and an integer that indicates which method will be used. The yearly depreciation and the remaining (undepreciated) value of the item will then be calculated and written out for each year.

The *straight-line* method is the easiest to use. In this method the original value of the item is divided by its life (total number of years). The resulting quotient will be the amount by which the item depreciates each year. For example, if an $8000 item is to be depreciated over 10 years, then the annual depreciation would be $8000/10 = $800. Therefore, the value of the item would decrease by $800 each year. Notice that the annual depreciation is the same each year when using straight-line depreciation.

When using the *double-declining-balance* method, the value of the item will decrease by a constant *percentage* each year. Hence the actual amount of the depreciation, in dollars, will vary from one year to the next. To obtain the depreciation factor, we divide 2 by the life of the item. The depreciation factor is multiplied by the value of the item *at the beginning of each year* (not the original value of the item) to obtain the annual depreciation.

Suppose, for example, that we wish to depreciate an $8000 item over 10 years, using the double-declining-balance method. The depreciation factor will be 2/10 = 0.20. Hence the depreciation for the first year will be 0.20 × $8000 = $1600. The second year's depreciation will be 0.20 × ($8000 − $1600) = 0.20 × $6400 = $1280; the third year's depreciation will be 0.20 × $5120 = $1024, and so on.

In the *sum-of-the-years'-digits* method the value of the item will decrease by a percentage that is *different* each year. The depreciation factor will be a fraction whose denominator is the sum of the digits from 1 to n, where n represents the life of the item. If, for example, we consider a 10-year lifetime, the denominator will be $1 + 2 + 3 + \cdots + 10 = 55$. For the first year the numerator will be n, for the second year it will be $(n - 1)$, for the third year $(n - 2)$, and so on. The yearly depreciation is obtained by multiplying the depreciation factor by the original value of the item.

To see how the sum-of-the-years'-digits method works, we again depreciate an \$8000 item over 10 years. The depreciation for the first year will be $(10/55) \times \$8000 = \$1454.55$; for the second year it will be $(9/55) \times \$8000 = \$1309.09$; and so on.

Now let us define the following symbols, so that we can write the actual program.

`val` = the current value of the item
`tag` = the original value of the item (i.e., the original value of `val`)
`deprec` = the annual depreciation
`n` = the number of years over which the item will be depreciated
`year` = a counter ranging from 1 to `n`
`choice` = an integer indicating which method to use

Our C program will follow the outline presented below.

1. Declare all variables, and initialize the integer variable `choice` to 0 (actually, we can assign any value other than 4 to `choice`).
2. Repeat all of the following steps as long as the value of `choice` is not equal to 4.
   (*a*) Read a value for `choice` which indicates the type of calculation to be carried out. This value can only be 1, 2, 3 or 4. (Any other value will be an error.)
   (*b*) If choice is assigned a value of 1, 2 or 3, read values for `val` and `n`.
   (*c*) Depending on the value assigned to `choice`, branch to the appropriate part of the program and carry out the indicated calculations. In particular,
      (*i*) If `choice` is assigned a value of 1, 2 or 3, calculate the yearly depreciation and the new value of the item on a year-by-year basis, using the appropriate method indicated by the value of `choice`. Print out the results as they are calculated, on a year-by-year basis.
      (*ii*) If `choice` is assigned a value of 4, write out a "goodbye" message and end the computation by terminating the `while` loop.
      (*iii*) If `choice` is assigned any value other than 1, 2, 3 or 4, write out an error message and begin another pass through the `while` loop.

Now let us express this outline in pseudocode.

```
#include files

main()

{
    /* variable declarations and initialization */

    while (choice != 4)  {

          /* generate menu and read choice */

          if (choice >= 1 && choice <= 3)
             /* read val and n */

          switch (choice)  {

          case  1:        /* straight-line method */

                /* write out title */
```

```
                /* calculate depreciation */

                /* for each year:
                        calculate a new value
                        write out year, depreciation, value */

            case  2:        /* double-declining-balance method */

                /* write out title */

                /* for each year:
                        calculate depreciation
                        calculate a new value
                        write out year, depreciation, value */

            case  3:        /* sum-of-the-years'-digits method */

                /* write out title */

                /* tag original value */

                /* for each year:
                        calculate depreciation
                        calculate a new value
                        write out year, depreciation, value */

            case  4:        /* end of computation */

                /* write "goodbye" message */

                /* write out title */

            default:        /* generate error message */

                /* write error message */
            }
        }
    }
```

Most of the pseudocode is straightforward, though a few comments are in order. First, we see that a `while` statement is used to repeat the entire set of calculations. Within this overall loop, the `switch` statement is used to select a particular depreciation method. Each depreciation method uses a `for` statement to carry out the required calculations.

At this point it is not difficult to write a complete C program, as shown below.

```
/* calculate depreciation using one of three different methods */

#include <stdio.h>

main()
{
    int n, year, choice = 0;
    float val, tag, deprec;

    while (choice != 4)  {

        /* read input data */

        printf("\nMethod: (1-SL  2-DDB  3-SYD  4-End) ");
        scanf("%d", &choice);
        if (choice >= 1 && choice <= 3)  {
           printf("Original value: ");
           scanf("%f", &val);
```

```
            printf("Number of years: ");
            scanf("%d", &n);
        }

        switch (choice)  {

        case 1:         /* straight-line method */

            printf("\nStraight-Line Method\n\n");
            deprec = val/n;
            for (year = 1; year <= n; ++year)  {
                val -= deprec;
                printf("End of Year %2d", year);
                printf("  Depreciation: %7.2f", deprec);
                printf("  Current Value: %8.2f\n", val);
            }
            break;

        case 2:         /* double-declining-balance method */

            printf("\nDouble-Declining-Balance Method\n\n");
            for (year = 1; year <= n; ++year)  {
                deprec = 2*val/n;
                val -= deprec;
                printf("End of Year %2d", year);
                printf("  Depreciation: %7.2f", deprec);
                printf("  Current Value: %8.2f\n", val);
            }
            break;

        case 3:         /* sum-of-the-years'-digits method */

            printf("\nSum-Of-The-Years\'-Digits Method\n\n");
            tag = val;
            for (year = 1; year <= n; ++year)  {
                deprec = (n-year+1)*tag / (n*(n+1)/2);
                val -= deprec;
                printf("End of Year %2d", year);
                printf("  Depreciation: %7.2f", deprec);
                printf("  Current Value: %8.2f\n", val);
            }
            break;

        case 4:         /* end of computation */

            printf("\nGoodbye, have a nice day!\n");
            break;

        default:         /* generate error message */

            printf("\nIncorrect data entry - please try again\n");
        }    /* end switch */
    }    /* end while */
}
```

The calculation of the depreciation for the sum-of-the-years'-digits method may be somewhat obscure. In particular, the term `(n-year+1)` in the numerator requires some explanation. This quantity is used to count *backward* (from `n` down to 1) as `year` progresses *forward* (from 1 to `n`). These declining values are required by the sum-of-the-years'-digits method. We could, of course, have set up a backward-counting loop instead, i.e.

```
for (year = n; year >= 1; --year)
```

but then we would have required a corresponding forward-counting loop to write out the results of the calculations on a yearly basis. Also, the term `(n*(n+1)/2)` which appears in the denominator is a formula for the sum of the first n digits; i.e., `1 + 2 + . . . + n`.

The program is designed to be run interactively, with prompts for the required input data. Notice that the program generates a *menu* with four choices, to calculate the depreciation using one of the three methods or to end the computation. The computer will continue to accept new sets of input data, and carry out the appropriate calculations for each data set, until a value of 4 is selected from the menu. The program automatically generates an error message and returns to the menu if some value other than 1, 2, 3 or 4 is entered in response to the menu request.

Some representative output is shown below. In each case, an $8000 item is depreciated over a 10-year period, using one of the three methods. The error message that is generated by an incorrect data entry is also illustrated. Finally, the computation is terminated in response to the last menu selection.

```
Method: (1-SL  2-DDB  3-SYD  4-End) 1
Original value: 8000
Number of years: 10

Straight-Line Method

End of Year  1  Depreciation: 800.00  Current Value: 7200.00
End of Year  2  Depreciation: 800.00  Current Value: 6400.00
End of Year  3  Depreciation: 800.00  Current Value: 5600.00
End of Year  4  Depreciation: 800.00  Current Value: 4800.00
End of Year  5  Depreciation: 800.00  Current Value: 4000.00
End of Year  6  Depreciation: 800.00  Current Value: 3200.00
End of Year  7  Depreciation: 800.00  Current Value: 2400.00
End of Year  8  Depreciation: 800.00  Current Value: 1600.00
End of Year  9  Depreciation: 800.00  Current Value:  800.00
End of Year 10  Depreciation: 800.00  Current Value:    0.00

Method: (1-SL  2-DDB  3-SYD  4-End) 2
Original value: 8000
Number of years: 10

Double-Declining-Balance Method

End of Year  1  Depreciation:1600.00  Current Value: 6400.00
End of Year  2  Depreciation:1280.00  Current Value: 5120.00
End of Year  3  Depreciation:1024.00  Current Value: 4096.00
End of Year  4  Depreciation: 819.20  Current Value: 3276.80
End of Year  5  Depreciation: 655.36  Current Value: 2621.44
End of Year  6  Depreciation: 524.29  Current Value: 2097.15
End of Year  7  Depreciation: 419.43  Current Value: 1677.72
End of Year  8  Depreciation: 335.54  Current Value: 1342.18
End of Year  9  Depreciation: 268.44  Current Value: 1073.74
End of Year 10  Depreciation: 214.75  Current Value:  858.99

Method: (1-SL  2-DDB  3-SYD  4-End) 3
Original value: 8000
Number of years: 10

Sum-of-the-Years'-Digits Method
```

```
End of Year  1  Depreciation:1454.55  Current Value: 6545.45
End of Year  2  Depreciation:1309.09  Current Value: 5236.36
End of Year  3  Depreciation:1163.64  Current Value: 4072.73
End of Year  4  Depreciation:1018.18  Current Value: 3054.55
End of Year  5  Depreciation: 872.73  Current Value: 2181.82
End of Year  6  Depreciation: 727.27  Current Value: 1454.55
End of Year  7  Depreciation: 581.82  Current Value:  872.73
End of Year  8  Depreciation: 436.36  Current Value:  436.36
End of Year  9  Depreciation: 290.91  Current Value:  145.45
End of Year 10  Depreciation: 145.45  Current Value:    0.00

Method: (1-SL  2-DDB  3-SYD  4-End) 5

Incorrect data entry - please try again

Method: (1-SL  2-DDB  3-SYD  4-End) 4

Goodbye, have a nice day!
```

Notice that the double-declining-balance method and the sum-of-the-years'-digits method result in a large annual depreciation during the early years, but a very small annual depreciation in the last few years of the item's lifetime. Also, we see that the item has a value of zero at the end of its lifetime when using the straight-line method and the sum-of-the-years'-digits method, but a small value remains undepreciated when using the double-declining-balance method.

6.8 THE `break` STATEMENT

The `break` statement is used to terminate loops or to exit from a switch. It can be used within a `for`, `while`, `do - while`, or `switch` statement.

The `break` statement is written simply as

```
break;
```

without any embedded expressions or statements.

We have already seen several examples of the use of the `break` statement within a `switch` statement, in Sec. 6.7. The `break` statement causes a transfer of control out of the entire `switch` statement, to the first statement following the `switch` statement.

EXAMPLE 6.27 Consider once again the `switch` statement originally presented in Example 6.24.

```
switch (choice = toupper(getchar()))  {

case 'R':
     printf("RED");
     break;

case 'W':
     printf("WHITE");
     break;

case 'B':
     printf("BLUE");
     break;

default:
     printf("ERROR");
     break;
}
```

Notice that each group of statements ends with a `break` statement, in order to transfer control out of the `switch` statement. The `break` statement is required within each of the first three groups, in order to prevent the succeeding groups of statements from executing. The last group does not require a `break` statement, since control will automatically be transferred out of the `switch` statement after the last group has been executed. This last `break` statement is included, however, as a matter of good programming practice, so that it will be present if another group of statements is added later.

If a `break` statement is included in a `while`, `do - while` or `for` loop, then control will immediately be transferred out of the loop when the `break` statement is encountered. This provides a convenient way to terminate the loop if an error or other irregular condition is detected.

EXAMPLE 6.28 Here are some illustrations of loops that contain `break` statements. In each situation, the loop will continue to execute as long as the current value for the floating-point variable `x` does not exceed 100. However, the computation will break out of the loop if a negative value for `x` is detected.

First, consider a `while` loop.

```
scanf("%f", &x);
while (x <= 100)  {
      if (x < 0)    {
         printf("ERROR - NEGATIVE VALUE FOR X");
         break;
      }
      /* process the nonnegative value of x */
      . . . . .
      scanf("%f", &x);
}
```

Now consider a `do - while` loop that does the same thing.

```
do  {
       scanf("%f", &x);
       if (x < 0)  {
          printf("ERROR - NEGATIVE VALUE FOR X");
          break;
    }
    /* process the nonnegative value of x */
    . . . . .
} while (x <= 100);
```

Finally, here is a `for` loop that is similar.

```
for (count = 1; x <= 100; ++count)   {
    scanf("%f", &x);
    if (x < 0)  {
       printf("ERROR - NEGATIVE VALUE FOR X");
       break;
    }
    /* process the nonnegative value of x */
    . . . . .
}
```

In the event of several nested `while`, `do - while`, `for` or `switch` statements, a `break` statement will cause a transfer of control out of the immediate enclosing statement, but not out of the outer surrounding statements. We have seen one illustration of this in Example 6.26, where a `switch` statement is embedded within a `while` statement. Another illustration is shown below.

EXAMPLE 6.29 Consider the following outline of a `while` loop embedded within a `for` loop.

```
for (count = 0; count <= n; ++count)  {
    . . . . .
    while (c = getchar() != '\n')  {
          if (c = '*')  break;
          . . . . .
    }
}
```

If the character variable `c` is assigned an asterisk (`*`), then the `while` loop will be terminated. However, the `for` loop will continue to execute. Thus, if the value of `count` is less than `n` when the breakout occurs, the computer will increment `count` and make another pass through the `for` loop.

6.9 THE `continue` STATEMENT

The `continue` statement is used to *bypass* the remainder of the current pass through a loop. The loop does *not* terminate when a `continue` statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. (Note the distinction between `continue` and `break`.)

The `continue` statement can be included within a `while`, a `do - while` or a `for` statement. It is written simply as

```
continue;
```

without any embedded statements or expressions.

EXAMPLE 6.30 Here are some illustrations of loops that contain `continue` statements.

First, consider a `do - while` loop.

```
do  {
       scanf("%f", &x);
       if (x < 0)  {
          printf("ERROR - NEGATIVE VALUE FOR X");
          continue;
    };

    /* process the nonnegative value of x */

       . . . . .
} while (x <= 100);
```

Here is a similar `for` loop.

```
for (count = 1; x <= 100; ++count)   {
    scanf("%f", &x);
    if (x < 0)  {
       printf("ERROR - NEGATIVE VALUE FOR X");
       continue;
    }
```

```
    /* process the nonnegative value of x */

    . . . . .
}
```

In each case, the processing of the current value of x will be bypassed if the value of x is negative. Execution of the loop will then continue with the next pass.

It is interesting to compare these structures with those shown in Example 6.28, which make use of the `break` statement instead of the `continue` statement. (Why is a modification of the `while` loop shown in Example 6.28 not included in this example?)

EXAMPLE 6.31 Averaging a List of Nonnegative Numbers In Example 6.17 we saw a complete C program that uses a `for` loop to calculate the average of a list of n numbers. Let us now modify this program so that it processes only nonnegative numbers.

The earlier program requires two minor changes to accommodate this modification. First, the `for` loop must include an `if` statement to determine whether or not each new value of x is nonnegative. A `continue` statement will be included in the `if` statement to bypass the processing of negative values of x. Secondly, we require a special counter (`navg`) to determine how many nonnegative numbers have been processed. This counter will appear in the denominator when the average is calculated (i.e., the average will be determined as `average = sum/navg`).

Here is the actual C program. It is interesting to compare it with the program shown in Example 6.17.

```
/* calculate the average of the nonnegative numbers in a list of n numbers */

#include <stdio.h>

main()

{
    int n, count, navg = 0;
    float x, average, sum = 0;

    /* initialize and read in a value for n */
    printf("How many numbers? ");
    scanf("%d", &n);

    /* read in the numbers */
    for (count = 1; count <= n; ++count)  {
        printf("x = ");
        scanf("%f", &x);
        if (x < 0) continue;
        sum += x;
        ++navg;
    }

    /* calculate the average and write out the answer */
    average = sum/navg;
    printf("\nThe average is %f\n", average);
}
```

When the program is executed with nonnegative values for x, it behaves exactly like the earlier version presented in Example 6.17. When some of the x's are assigned negative values, however, the negative values are ignored in the calculation of the average.

A sample interactive session is shown below. As usual, the user's responses are underlined.

```
How many numbers? 6
x = 1
x = -1
x = 2
x = -2
x = 3
x = -3

The average is 2.000000
```

This is the correct average of the positive numbers. Note that the average would be zero if all of the numbers had been averaged.

6.10 THE COMMA OPERATOR

We now introduce the comma operator (,) which is used primarily in conjunction with the `for` statement. This operator permits two different expressions to appear in situations where only one expression would ordinarily be used. For example, it is possible to write

`for (`*`expression 1a, expression 1b`*`; `*`expression 2`*`; `*`expression 3`*`) `*`statement`*

where *expression 1a* and *expression 1b* are the two expressions, separated by the comma operator, where only one expression (*expression 1*) would normally appear. These two expressions would typically initialize two separate indices that would be used simultaneously within the `for` loop.

Similarly, a `for` statement might make use of the comma operator in the following manner.

`for (`*`expression 1`*`; `*`expression 2`*`; `*`expression 3a, expression 3b`*`) `*`statement`*

Here *expression 3a* and *expression 3b*, separated by the comma operator, appear in place of the usual single expression. In this application the two separate expressions would typically be used to alter (e.g., increment or decrement) two different indices that are used simultaneously within the loop. For example, one index might count forward while the other counts backward.

EXAMPLE 6.32 Searching for Palindromes A *palindrome* is a word, phrase or sentence that reads the same way either forward or backward. For example, words such as *noon*, *peep*, and *madam* are palindromes. If we disregard punctuation and blank spaces, then the sentence *Rise to vote, sir!* is also a palindrome.

Let us write a C program that will enter a line of text containing a word, a phrase or a sentence, and determine whether or not the text is a palindrome. To do so, we will compare the first character with the last, the second character with the next to last, and so on, until we have reached the middle of the text. The comparisons will include punctuation and blank spaces.

In order to outline a computational strategy, let us define the following variables.

`letter` = a `character`-type array containing as many as 80 elements. These elements will be the characters in the line of text.

`tag` = an `integer` variable indicating the number of characters assigned to `letter`, excluding the escape character `\0` at the end.

`count` = an `integer` variable used as an index when moving forward through `letter`.

`countback` = an integer variable used as an index when moving backward through `letter`.

`flag` = an `integer` variable that will be used to indicate a true/false condition. True will indicate that a palindrome has been found.

`loop` = an `integer` variable whose value will always equal 1, thus appearing always to be true. The intent here is to continue execution of a main loop, until a particular stopping condition causes a breakout.

We can now outline our overall strategy as follows.

1. Define the symbolic constants EOL (end-of-line), TRUE and FALSE.
2. Declare all variables and initialize loop (i.e., assign TRUE to loop).
3. Enter the main loop.
   (*a*) Assign TRUE to flag, in anticipation of finding a palindrome.
   (*b*) Read in the line of text on a character-by-character basis, and store in letter.
   (*c*) Test to see if the uppercase equivalents of the first three characters are E, N and D, respectively. If so, break out of the main loop and exit the program.
   (*d*) Assign the final value of count, less 1, to tag. This value will indicate the number of characters in the line of text, not including the final escape character \0.
   (*e*) Compare each character in the first half of letter with the corresponding character in the second half. If a mismatch is found, assign FALSE to flag and break out of the (inner) comparison loop.
   (*f*) If flag is TRUE, display a message indicating that a palindrome has been found. Otherwise, display a message indicating that a palindrome has not been found.
4. Repeat step 3 (i.e., make another pass through the outer loop), thus processing another line of text.

Here is the corresponding pseudocode.

```
#include files

#define symbolic constants

main()

{
   /* declare all variables and initialize as required */

   while (loop)  {

      flag = TRUE;    /* anticipating a palindrome */

      /* read in a line of text and store in letter */

      /* break out of while loop if first three characters
         of letter spell END (test uppercase equivalents) */

      /* assign number of characters in text to tag */

      for ((count = 0, countback = tag); count <= (tag - 1)/ 2;(++count, --countback)) {

          if (letter[count] != letter[countback])  {
             flag = FALSE;

             /* not a palindrome - break out of for loop */
          }
      }

      /* display a message indicating whether or not letter contains a palindrome */

   }
}
```

The program utilizes the comma operator within a for loop to compare each character in the first half of letter with the corresponding character in the second half. Thus, as count increases from 0 to (tag - 1) / 2, countback decreases from tag to (tag / 2) + 1. Note that integer division (resulting in a truncated quotient) is involved in establishing these limiting values.

Also, observe that there will be two distinct comma operators within the `for` statement. Each comma operator and its associated operands are enclosed in parentheses. This is not necessary, but it does emphasize that each operand pair comprises one argument within the `for` statement.

The complete C program is shown below.

```
/* search for a palindrome */

#include <stdio.h>
#include <ctype.h>

#define EOL  '\n'
#define TRUE 1
#define FALSE 0

main()
{
    char letter[80];
    int tag, count, countback, flag, loop = TRUE;

    /* main loop */

    while (loop)   {
       flag = TRUE;

       /* read the text */

       printf("Please enter a word, phrase or sentence below:\n");
       for (count = 0; (letter[count] = getchar()) != EOL; ++count)
           ;
       if ((toupper(letter[0]) == 'E') && (toupper(letter[1]) == 'N') &&
           (toupper(letter[2]) == 'D')) break;
       tag = count - 1;

       /* carry out the search */

       for ((count = 0, countback = tag); count <= tag/2;
           (++count, --countback))   {

              if (letter[count] != letter[countback])   {
                 flag = FALSE;
                 break;
              }
       }

       /* display message */

       for (count = 0; count <= tag; ++count)
           putchar(letter[count]);
       if (flag) printf(" IS a palindrome\n\n");
       else printf(" is NOT a palindrome\n\n");
    }
}
```

A typical interactive session is shown below, indicating the type of output that is generated when the program is executed. As usual, the user's responses are underlined.

```
Please enter a word, phrase or sentence below:
TOOT

TOOT IS a palindrome
```

```
Please enter a word, phrase or sentence below:
FALSE

FALSE is NOT a palindrome

Please enter a word, phrase or sentence below:
PULLUP

PULLUP IS a palindrome

Please enter a word, phrase or sentence below:
ABLE WAS I ERE I SAW ELBA

ABLE WAS I ERE I SAW ELBA IS a palindrome

Please enter a word, phrase or sentence below:
END
```

Remember that the comma operator accepts two distinct expressions as operands. These expressions will be evaluated from left to right. In situations that require the evaluation of the overall expression (i.e., the expression formed by the two operands and the comma operator), the type and value of the overall expression will be determined by the type and value of the right operand.

Within the collection of C operators, the comma operator has the lowest precedence. Thus, the comma operator falls within its own unique precedence group, beneath the precedence group containing the various assignment operators (see Appendix C). Its associativity is left to right.

6.11 THE goto STATEMENT

The goto statement is used to alter the normal sequence of program execution by transferring control to some other part of the program. In its general form, the goto statement is written as

```
goto label;
```

where *label* is an identifier that is used to label the target statement to which control will be transferred.

Control may be transferred to any other statement within the program. (To be more precise, control may be transferred anywhere within the current *function*. We will introduce functions in the next chapter, and discuss them thoroughly in Chapter 7.) The target statement must be labeled, and the label must be followed by a colon. Thus, the target statement will appear as

```
label: statement
```

Each labeled statement within the program (more precisely, within the current function) must have a unique label; i.e., no two statements can have the same label.

EXAMPLE 6.33 The following skeletal outline illustrates how the goto statement can be used to transfer control out of a loop if an unexpected condition arises.

```
/* main loop */

scanf("%f", &x);
while (x <= 100)   {
      . . . . .
      if (x < 0) goto errorcheck;
      . . . . .
      scanf("%f", &x);
}
```

```
    . . . . .
/* error detection routine */
errorcheck: {
               printf("ERROR - NEGATIVE VALUE FOR X");
               . . . . .
            }
```

In this example control is transferred out of the `while` loop, to the compound statement whose label is `errorcheck`, if a negative value is detected for the input variable `x`.

The same thing could have been accomplished using the `break` statement, as illustrated in Example 6.28. The use of the `break` statement is actually the preferred approach. The use of the `goto` statement is presented here only to illustrate the syntax.

All of the popular general-purpose programming languages contain a `goto` statement, though modern programming practice discourages its use. The `goto` statement was used extensively, however, in early versions of some older languages, such as Fortran and BASIC. The most common applications were:

1. Branching around statements or groups of statements under certain conditions.
2. Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during the current pass.
3. Jumping completely out of a loop under certain conditions, thus terminating the execution of a loop.

The structured features in C enable all of these operations to be carried out without resorting to the `goto` statement. For example, branching around statements can be accomplished with the `if - else` statement; jumping to the end of a loop can be carried out with the `continue` statement; and jumping out of a loop is easily accomplished using the `break` statement. The use of these structured features is preferrable to the use of the `goto` statement, because the use of `goto` tends to encourage (or at least, not discourage) logic that skips all over the program whereas the structured features in C require that the entire program be written in an orderly, sequential manner. For this reason, *use of the* `goto` *statement should generally be avoided.*

Occasional situations do arise, however, in which the `goto` statement can be useful. Consider, for example, a situation in which it is necessary to jump out of a doubly nested loop if a certain condition is detected. This can be accomplished with two `if - break` statements, one within each loop, though this is awkward. A better solution in this particular situation might make use of the `goto` statement to transfer out of both loops at once. The procedure is illustrated in the following example.

EXAMPLE 6.34 Converting Several Lines of Text to Uppercase Example 6.19 presents a program to convert several successive lines of text to uppercase, processing one line of text at a time, until the first character in a new line is an asterisk (*). Let us now modify this program to detect a break condition, as indicated by two successive dollar signs (`$$`) anywhere within a line of text. If the break condition is encountered, the program will print the line of text containing the dollar signs, followed by an appropriate message. Execution of the program will then terminate.

The logic will be the same as that given in Example 6.19, except that an additional loop will now be added to test for two consecutive dollar signs. Thus the program will proceed as follows.

1. Assign an initial value of 1 to the outer loop index (`linecount`).
2. Carry out the following steps repeatedly, for successive lines of text, as long as the first character in the line is not an asterisk.
   (*a*) Read in a line of text and assign the individual characters to the elements of the char-type array `letter`. A line will be defined as a succession of characters that is terminated by an end-of-line (i.e, a *newline*) designation.
   (*b*) Assign the character count, including the end-of-line character, to `tag`.

(*c*) Display the line in uppercase, using the library function `toupper` to carry out the conversion. Then display two newline characters (so that the next line of input will be separated from the current output by a blank line), and increment the line counter (`linecount`).

(*d*) Test all successive characters in the line for two successive dollar signs. If two successive dollar signs are detected, then display a message indicating that a break condition has been found and jump to the terminating condition at the end of the program (see below).

3. Once an asterisk has been detected as the first character of a new line, write out "Good bye." and terminate the computation.

Here is the complete C program.

```
/* convert several lines of text to uppercase

   Continue conversion until the first character in a line is an asterisk (*).
   Break out of the program sooner if two successive dollar signs ($$) are detected */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()

{
    char letter[80];
    int tag, count, linecount = 1;

    while ((letter[0] = getchar()) != '*')   {

          /* read in a line of text */
          for (count = 1; (letter[count] = getchar()) != EOL; ++count)
              ;
          tag = count;

          /* display the line of text */
          for (count = 0; count < tag; ++count)
              putchar(toupper(letter[count]));
          printf("\n\n");
          ++linecount;

          /* test for a break condition */
          for (count=1; count < tag; ++count)
              if (letter[count-1] == '$' && letter[count] == '$')   {
                 printf("BREAK CONDITION DETECTED - TERMINATE EXECUTION\n\n");
                 goto end;
              }
    }
    end: printf("Good bye");
}
```

It is interesting to compare this program with the corresponding program presented earlier, in Example 6.19. The present program contains an additional `for` loop embedded at the end of the `while` loop. This `for` loop examines consecutive pairs of characters for a break condition (`$$`), after the entire line has already been written out in uppercase. If a break condition is encountered, then control is transferred to the final `printf` statement (`"Good bye"`) which is now labeled `end`. Note that this transfer of control causes a breakout from the `if` statement, the current `for` loop, and the outer `while` loop.

You should run this program, using both the regular terminating condition (an asterisk at the start of a new line) and the breakout condition. Compare the results obtained with the output shown in Example 6.19.