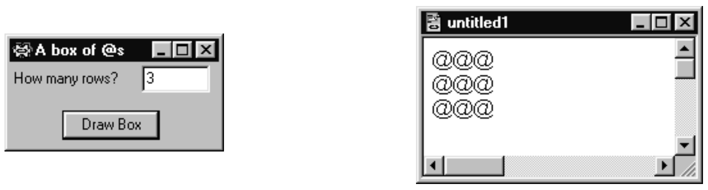
Chapter *11*

Nested Loops

Any structured statement can be nested in any other structured statement. In the same way that an IF statement can be nested inside another IF, a loop statement can be nested inside another loop. Such a configuration is called a nested loop.

Printing a box of characters

Procedure DrawBox in Listing 11.2 uses a nested loop to draw a box of @ symbols with the same number of rows and columns. It prompts the user for the number of rows in a dialog box, then creates a window that displays the box of @s. Figure 11.1 shows the input and output for the program.



```
@@@
@@@
@@@
```

Figure 11.1
The input and output for the program in Figure 11.2.

In Figure 11.2, the FOR loop with control variable i is called the outer loop, and the nested loop with control variable j is called the inner loop. The first time the outer FOR statement executes, it initializes i to 1. Then the inner loop gives the values 1, 2 and 3 in turn to j, causing the body of the inner loop to execute three times. Each time the body of the inner loop executes, it inserts a single @ symbol into the text model. The fourth time the inner FOR statement executes, it detects that the loop should terminate.

Then fm.WriteLn executes, which inserts the line character into the text model. If the line character were omitted in the model, all the @ symbols would be displayed on the same line in the view. The fm.WriteLn statement is outside the body of the inner loop, but inside the body of the outer loop.

The second time the outer FOR statement executes, it increments i to 2. The inner loop executes exactly as before, producing another sequence of three @ symbols in the text model. The same thing happens after i gets the value of 3.

```
MODULE Pbox11A;
    IMPORT Dialog, TextModels, TextViews, Views, PboxMappers;
    VAR
        d*: RECORD
            numRows*: INTEGER;
        END;

    PROCEDURE DrawBox*;
        VAR
            md: TextModels.Model;
            vw: TextViews.View;
            fm: PboxMappers.Formatter;
            i, j: INTEGER;
    BEGIN
        md := TextModels.dir.New();
        fm.ConnectTo(md);
        FOR i := 1 TO d.numRows DO
            FOR j := 1 TO d.numRows DO
                fm.WriteChar("@")
            END;
            fm.WriteLn
        END;
        vw := TextViews.dir.New(md);
        Views.OpenView(vw)
    END DrawBox;

BEGIN
    d.numRows := 0
END Pbox11A.
```

Figure 11.2
A procedure that prints a box of @ characters with nested loops.

Statement execution count

What is the statement execution count for this program? Procedure DrawBox has eight executable statements:

The executable statements of procedure DrawBox

Statement number	*Executable statement*
(1)	md := TextModels.dir.New()
(2)	fm.ConnectTo(md)
(3)	FOR i := 1 TO d.numRows DO
(4)	FOR j := 1 TO d.numRows DO
(5)	fm.WriteChar("@")
(6)	fm.WriteLn
(7)	vw := TextViews.dir.New(md)
(8)	Views.OpenView(vw)

If you trace the algorithm as described above for d.numRows having a value of 3, you will find that statements (1), (2), (7), and (8) each execute once. The outer FOR statement (3) executes four times, and the inner FOR statement (4) executes 12 times. The body of the inner loop (5) executes nine times. Statement (6), which is in the body of the outer loop but not in the body of the inner loop, executes three times. A total of 32 statements execute.

What if the value of d.numRows is n in general? Statements in a nested loop are a little more difficult to count than those in a single loop. Consider the statements in the inner loop as if they were not nested in the outer loop.

```
FOR j := 1 TO d.numRows DO
   fm.WriteChar("@")
END;
fm.WriteLn
```

(a) The FOR statement would execute $n + 1$ times, (b) the body would execute n times, and (c) the WriteChar statement would execute 1 time. But these statements are, in fact, the body of a loop that executes n times. So each one executes n times the amounts just mentioned. Namely, (a) the first statement executes $n(n + 1)$ times, (b) the body executes n^2 times, and (c) the WriteChar statement executes n times. The FOR statement of the outer loop

```
FOR i := 1 TO d.numRows DO
```

executes $n + 1$ times. Adding these terms together plus the four statements outside the loop that each execute once, yields a total of $2n^2 + 3n + 5$ for the whole program. Totals for the statement execution counts when d.numRows has values of zero, three, and n in general are summarized in Figure 11.3.

Statement	d.numRows **= 0**	d.numRows **= 3**	d.numRows **= *n***
(1)	1	1	1
(2)	1	1	1
(3)	1	4	$n + 1$
(4)	0	12	$n(n + 1)$
(5)	0	9	n^2
(6)	0	3	n
(7)	1	1	1
(8)	1	1	1
Total:	5	32	$2n^2 + 3n + 5$

Figure 11.3
Statement execution count for the procedure DrawBox in Figure 11.2.

The squared term in the expression for the number of statements executed is typical for nested loops. It causes an estimate for the execution time that is quite different from the case where the statement count is proportional to the first power of n.

Example 11.1 Suppose it takes 400 μs to execute procedure DrawBox when d.numRows has the value of 35. If you double the value to 70, how long will it take to execute?

Setting up the precise ratio gives

$$\frac{400}{2(35)^2 + 3(35) + 5} = \frac{T}{2(70)^2 + 3(70) + 5}$$

As before, however, we can approximate by neglecting the lower-order terms in the statement execution count.

$$\frac{400}{2(35)^2} = \frac{T}{2(70)^2}$$

Again, the coefficient cancels, but this time solving for T yields $400(70/35)^2$, which is $400(4)$. That is 1600 μs, which is four times the original execution time, not just double the execution time. ▮

Such a result is expected when you think of the output. When you double the value given to d.numRows, you quadruple the number of @ symbols that need to be output, because the program prints a figure in the shape of a box with an equal number of rows and columns.

Printing a triangle

It is possible for the upper or lower limit of the control variable of the inner loop to depend on the control variable of the outer loop. Suppose you change the loops of procedure DrawBox as in Figure 11.4.

```
FOR i := 1 TO d.numRows DO
    FOR j := 1 TO i DO
        fm.WriteChar("@")
    END;
    fm.WriteLn
END;
```

Figure 11.4 An algorithm for printing a triangle of @ characters.

The control variable of the inner loop, j, now goes from 1 to i, where i is the control variable of the outer loop. When i has the value 1, the inner loop gives j values from 1 to 1. That makes the inner loop execute once. When i has value 2, the inner loop gives j values from 1 to 2, for two executions of the inner loop. The last time, j goes from 1 to 3 for three executions. The effect is to print a triangle of @ symbols. If d.numRows has a value of 3, the output will be

```
@
@@
@@@
```

A trace of the program for d.numRows having a value of 3 is shown below, assuming that statement (4) has been modified as Figure 11.4. The trace does not show the values of i or j that are irrelevant outside their loops.

A trace of the algorithm in Figure 11.4 with an input of 3

Statement executed	i	j
(1)		
(2)		
(3)	1	
(4)	1	1
(5)	1	1
(4)	1	2
(6)	1	
(3)	2	
(4)	2	1
(5)	2	1
(4)	2	2
(5)	2	2
(4)	2	3
(6)	2	
(3)	3	
(4)	3	1
(5)	3	1
(4)	3	2
(5)	3	2
(4)	3	3
(5)	3	3
(4)	3	4
(6)	3	
(3)	4	
(7)		
(8)		

You can see by simply counting this trace that the number of statements executed is 26.

determine the statement execution count of the modified program that prints the triangle for the general case when d.numRows has the value *n*, is a bit more difficult than with the previous nested loop we analyzed. The problem is that the statements in the inner loop do not simply execute *n* times more than they would execute if they were not nested. Figure 11.5 shows how to count the number of statements executed when d.numRows has the value zero, three, and *n* in general.

Statement	d.numRows **= 0**	d.numRows **= 3**	d.numRows **= *n***
(1)	1	1	1
(2)	1	1	1
(3)	1	4	$n + 1$
(4)	0	$2 + 3 + 4$	$2 + 3 + 4 + \ldots + (n + 1)$
(5)	0	$1 + 2 + 3$	$1 + 2 + 3 + \ldots + n$
(6)	0	3	n
(7)	1	1	1
(8)	1	1	1

Figure 11.5 Counting individual statements for the algorithm to draw a triangle.

Notice that the body of the inner FOR statement (5) executes once when i is 1, twice when i is 2, and three times when i is 3. Because the body statement prints a single @ symbol, these counts correspond to the fact that the first row has 1 @, the second has 2 @s, and the third has 3 @s. In general, when d.numRows has the value *n*, the number of times statement (5) executes is $1 + 2 + 3 + \ldots + n$. The discussion of the procedure in Figure 10.14 in the previous chapter notes that the formula for the sum of the first *m* integers is

$$1 + 2 + 3 + \ldots + m = \frac{m(m + 1)}{2}$$

where *m* is positive. So, the number of times statement (5) executes is $n(n + 1)/2$.

Furthermore, the inner FOR statement (4) executes twice when i is 1, three times when i is 2, and four times when i is 3. The reason statement (4) executes more frequently than statement (5) is the extra test it must perform after the last execution of the body of the loop. So, the total number of times statement (4) executes in general is

$$2 + 3 + 4 + \ldots + (n + 1)$$

But, this expression is equal to

$$-1 + 1 + 2 + 3 + 4 + \ldots + (n + 1)$$

which in turn is equal to

$$-1 + \frac{(n + 1)[(n + 1) + 1]}{2}$$

using the above formula for the sum of the first *m* integers with $m = (n + 1)$. Multiplying the numerator and combining terms gives a count for statement (4) of

$$\frac{1}{2}n^2 + \frac{3}{2}n$$

Figure 11.6 summarizes the statement execution counts for the algorithm to print a triangle in the cases where d.numRows equals zero, three, and n in general. The interesting conclusion to this analysis is that the execution time is still quadratic in n, just as with the algorithm to print a square.

Statement	d.numRows **= 0**	d.numRows **= 3**	d.numRows **= *n***
(1)	1	1	1
(2)	1	1	1
(3)	1	4	$n + 1$
(4)	0	9	$\frac{1}{2}n^2 + \frac{3}{2}n$
(5)	0	6	$\frac{1}{2}n^2 + \frac{1}{2}n$
(6)	0	3	n
(7)	1	1	1
(8)	1	1	1
Total:	5	26	$n^2 + 4n + 5$

Figure 11.6
Statement execution count for the algorithm to print a triangle.

Example 11.2 Suppose it takes 200 μs to execute this program when n has the value of 35. If you double the value to 70, how long will it take to execute?

Using the approximate ratio,

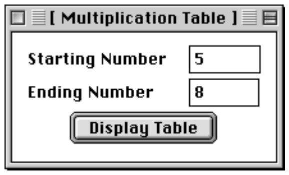
$$\frac{200}{(35)^2} = \frac{T}{(70)^2}$$

Solving for T yields $200(70/35)^2 = 200(4) = 800$ μs. So the quadratic nature of the statement execution count again predicts that the execution time is four times the original execution time when n is doubled. ▮

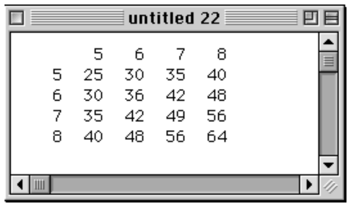
A multiplication table

Our last program in this chapter is a stepwise refinement problem that requires nested loops. The problem is to input two numbers and output a multiplication table for the values between the two numbers. The input will be taken from a dialog box and the output will be to a new window as Figure 11.7 shows.

First Pass—The input is a pair of numbers. That calls for two integer variables, d.startNum and d.endNum. The output consists of five rows for the inputs 5 and 8. The first row is different from the next four rows because it is the heading of the table. The other four rows contain the actual products. Regardless of the specific values input, the output will be one heading row and several rows of products. The integer variable, i, will represent the row number and will vary from startNum to endNum.



Starting Number 5
Ending Number 8



```
        5   6   7   8
    5  25  30  35  40
    6  30  36  42  48
    7  35  42  49  56
    8  40  48  56  64
```

Figure 11.7
The input and output for a program that displays a multiplication table.

```
*Input* d.startNum, d.endNum
*Output the first row*
FOR i := d.startNum TO d.endNum DO
    *Output row with* i *as first number*
END
```

Second Pass—The first row is a list of numbers from d.startNum to d.endNum. However, it contains some extra space before the first number, which must be output before the numbers in the first row are output. Otherwise the numbers in the first row will not be aligned over the columns properly. The numbers themselves can be output with a single FOR statement.

Each row in the body of the multiplication table starts with the single value of i. The rest of the row is the product of the number, i, with another number, j, that varies from d.startNum to d.endNum. Expanding on the first pass, the second pass is

```
*Input* d.startNum, d.endNum
*Output the space at the beginning of the first row*
FOR j := d.startNum TO d.endNum DO
    *Output* j
END
FOR i := d.startNum TO d.endNum DO
    *Output* i
    FOR j := d.startNum TO d.endNum DO
        *Output* i * j
    END
END
```

Third Pass—The third pass is the complete Component Pascal module in Figure 11.8. It has several output details not in the previous passes to make the numbers in the table line up properly. It prints each number in a field width of four. Four spaces must precede the first heading line so the numbers in the heading will line up with the proper columns. There is a special character constant named digitspace that you can import from module TextModels. The amount of space occupied by a TextModels.digitspace is exactly equal to the amount of space occupied by a decimal digit. Unless the font is monospaced, as Courier is, the normal space character may not occupy the same amount of space as a decimal digit.

```
MODULE Pbox11B;
    IMPORT Dialog, TextModels, TextViews, Views, PboxMappers;
    VAR
        d*: RECORD
            startNum*, endNum*: INTEGER;
        END;

    PROCEDURE MakeMultiplicationTable*;
        VAR
            md: TextModels.Model;
            vw: TextViews.View;
            fm: PboxMappers.Formatter;
            i, j: INTEGER;
    BEGIN
        md := TextModels.dir.New();
        fm.ConnectTo(md);
        FOR j := 1 TO 4 DO
            fm.WriteChar(TextModels.digitspace)
        END;
        FOR j := d.startNum TO d.endNum DO
            fm.WriteInt(j, 4)
        END;
        fm.WriteLn;
        FOR i := d.startNum TO d.endNum DO
            fm.WriteInt(i, 4);
            FOR j := d.startNum TO d.endNum DO
                fm.WriteInt(i * j, 4)
            END;
            fm.WriteLn
        END;
        vw := TextViews.dir.New(md);
        Views.OpenView(vw)
    END MakeMultiplicationTable;

BEGIN
    d.startNum := 0; d.endNum := 0
END Pbox11B.
```

Figure 11.8
A program that produces the multiplication table shown in Figure 11.7.

A convention for naming integer variables

This program uses a common programming convention for naming integer variables that are used to process a table. i is usually used as the control variable for the row loop, and j is usually used for the column loop. Later chapters continue this convention, and you should adopt it as well.

Exercises

1. Plot the total statement execution counts for procedure DrawBox (Figure 11.2) and its modification to print a triangle (Figure 11.4) on the same graph for values of *n* from 0 to 6. What shape does each graph have?

2. What is the total statement execution count of the code fragment

```
Statement 1 ;
FOR i := 1 TO d.num DO
    Statement 2 ;
    FOR j := 1 TO i DO
        Statement 3 ;
        Statement 4
    END
END
```

if d.num has the following values? Show the count for each statement in a table similar to Figure 11.6.

(a) 0 **(b)** 3 **(c)** *n*

3. What is the total statement execution count of the code fragment

```
Statement 1 ;
FOR i := 2 TO d.num DO
    Statement 2 ;
    FOR j := 1 TO i DO
        Statement 3
    END
END
```

if d.num has the following values? Show the count for each statement in a table similar to Figure 11.6. It may help you to first write the statement counts similar to the way they are written in Figure 11.5.

(a) 1 **(b)** 4 **(c)** *n*

4. What is the total statement execution count of the code fragment

```
Statement 1 ;
FOR i := 1 TO d.num DO
    Statement 2
    FOR j := -i TO i DO
        Statement 3
    END
END
```

if d.num has the following values? Show the count for each statement in a table similar to Figure 11.6. It may help you to first write the statement counts similar to the way they are written in Figure 11.5.

(a) 0 **(b)** 3 **(c)** *n*

You may need to use this formula for the sum of odd integers.

$1 + 3 + 4 + \ldots + (2n - 1) = n^2$ for positive n

Problems

5. Write a program that asks the user to enter in a dialog box the number of rows and columns of a block of @s to be printed. Display the box in a new window. For example, if the user enters 3 for the number of rows and 5 for the number of columns, the new window should contain the following text:

```
@@@@@
@@@@@
@@@@@
```

6. Write a program that asks the user to enter in a dialog box the number of rows and columns of an empty block of zeros to be printed. Display the box in a new window. For example, if the user enters 5 for the number of rows and 9 for the number of columns, the new window should contain the text shown below. You will need to output the TextModels.digitspace character inside the box for the right side of the box to line up properly.

```
000000000
0       0
0       0
0       0
000000000
```

7. Write a program that asks the user to enter in a dialog box the number of zeros on the base of a triangle, then prints a right triangle of zeros with a vertical right side. For example, if the user enters 5 for the number of rows, the new window should contain the text shown below. The leading spaces must be the TextModels.digitspace character for the right side of the triangle to be vertical.

```
    0
   00
  000
 0000
00000
```

8. Write a program that asks the user to enter in a dialog box the number of zeros on the base of a triangle, then prints a symmetric triangle. For example, if the user enters 7 for the number of zeros on the base the new window should contain the text

```
   0
  000
 00000
0000000
```

and if the user enters 8 the new window should contain the text

```
   00
  0000
 000000
00000000
```

The leading spaces must be the TextModels.digitspace character for the triangle to be symmetric.

9. Write a program that asks the user to enter in a dialog box a positive one-digit integer and prints a triangle of digits from one up to the digit entered. For example, if the user enters 4 the new window should display

```
1
22
333
4444
```

If the user enters a number other than a positive one-digit integer the new window should display an appropriate error message.