Chapter 10

Loops

A powerful feature of all computer systems is their ability to perform repetitious tasks. Most people dislike monotonous, mechanical jobs that require little thought. Computers have the marvelous property of executing monotonous jobs without tiring or complaining. A group of statements that executes repetitively is called a loop. This chapter examines two of Component Pascal's several loop statements—the WHILE statement and the FOR statement.

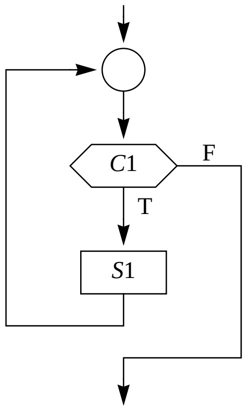


Figure 10.1
The flowchart for the WHILE statement.

The WHILE statement

WHILE statements are similar to IF statements because they both evaluate boolean expressions and execute a statement sequence if the boolean expression is true. The difference between them is that after the statement sequence executes in a WHILE statement, control is automatically transferred back up to the boolean expression for evaluation again. Each time the boolean expression is true, the body executes and the boolean expression is evaluated again. Figure 10.1 shows the flowchart for the WHILE statement

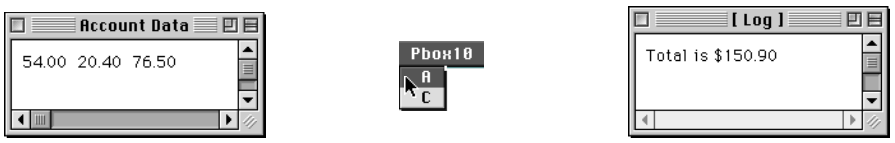
```
WHILE C1 DO
   S1
END
```

The WHILE statement tests condition C1 first. If C1 is false, it skips statement sequence S1. Otherwise, it executes statement sequence S1 and transfers control up through the collector to the test again.

The flowchart shows several important properties of the WHILE statement. First, there are two ways to reach the condition C1—from the statement immediately preceding the WHILE statement or from the body of the loop. If C1 is true the first time, it will be tested again after S1 executes. S1 must eventually do something to change the evaluation of C1. Otherwise, C1 would be true always, and the loop would execute endlessly. Second, Figure 10.1 also shows that it is possible for statement sequence S1 to never execute. It does not execute if C1 is determined to be false the first time.

The eot technique

The first program that illustrates the WHILE statement computes the sum of a list of numbers in the focus window. Each number represents the dollar balance in a customer's account. Figure 10.2 shows the input and output for this program. The input comes from the focus window as the result of a menu selection, and the output is displayed on the Log.



```
54.00  20.40  76.50
```

```
Total is $150.90
```

(a) The input window. **(b)** The menu selection **(c)** The output to the Log.

Figure 10.2 The input and output of the program in Listing 10.3.

Figure 10.2(a) shows a focus window with three real values. While this window was focused, the user selected the menu option as shown in Figure 10.2(b), which resulted in the output to the Log shown in Figure 10.2(c). The processing simply consists of adding the real values. To keep the analysis short in the following discussion only three numbers were totaled, but the program works equally well with any number of values in the focus window. The program in Listing 10.3 inputs the data from the focus window and produces the output on the Log.

As usual, sc is a variable of type PboxMappers.Scanner. sc is the object that scans the text model for the real values. This program uses the standard pattern for establishing the model from the focus window and linking sc to it.

The statement

```
sum := 0.0
```

initializes variable sum to 0.0. Then the statement

```
sc.ScanReal(balance)
```

scans the first value, which in this example is 54.00. The scan has two effects. First, because the value scanned from the model was 54.00 from the focus window, the effect is the same as the assignment statement

```
balance := 54.00
```

Second, because there was a value that got scanned, the effect is also the same as the assignment statement

```
sc.eot := FALSE
```

```
MODULE Pbox10A;
    IMPORT TextModels, TextControllers, PboxMappers, PboxStrings, StdLog;

    PROCEDURE ComputeTotal*;
        VAR
            md: TextModels.Model;
            cn: TextControllers.Controller;
            sc: PboxMappers.Scanner;
            balance: REAL;
            sum: REAL;
            sumString: ARRAY 16 OF CHAR;
    BEGIN
        cn := TextControllers.Focus();
        IF cn # NIL THEN
            md := cn.text;
            sc.ConnectTo(md);
            sum := 0.0;
            sc.ScanReal(balance);
            WHILE ~sc.eot DO
                sum := sum + balance;
                sc.ScanReal(balance)
            END;
            PboxStrings.RealToString(sum, 1, 2, sumString);
            StdLog.String("Total is $");
            StdLog.String(sumString); StdLog.Ln
        END
    END ComputeTotal;

END Pbox10A.
```

Figure 10.3
A program to find the total of all the data values in the focus window. It uses the eot technique.

The behavior of eot from PboxMappers.Scanner

You can see from the interface of PboxMappers in Figure 9.7 that every scanner has a variable exported read-only called eot, which stands for end of text. If your scanner attempts to scan an integer or real value from a text model but there are no more values left to scan, the actual parameter gets some unknown large value and sc.eot is set to true. Otherwise, the actual parameter gets the value scanned and sc.eot is set to false.

The next statement to execute is

```
WHILE ~sc.eot DO
```

Because sc.eot is false, ~sc.eot is true, and the body of the loop executes. The first statement in the statement sequence of the WHILE body is

```
sum := sum + balance
```

giving 0.0 + 54.0 to sum. The second statement in the statement sequence of the WHILE is

```
sc.ScanReal(balance)
```

which scans the next group of characters in the text, giving balance the value of 20.40 and s.eot the value of false.

Now, control returns back to the test of the WHILE loop, which is still true. So, the statement sequence executes again. The assignment statement gives sum the value of 54.0 + 20.4, which is 74.4. The sc.ScanReal(balance) statement gives balance the value of 76.5 and sc.eot the value of false.

Control again returns back to the test of the WHILE loop, which is true again. So, the statement sequence executes once more. The assignment statement gives sum the value of 74.4 + 76.5, which is 150.9. This time the sc.ScanReal(balance) statement gives sc.eot the value true because there is no more visible text after the position of the scanner.

When control returns back to the test of the WHILE loop, the condition is false, so the loop terminates, and the statements following the loop END execute. They convert the real value of sum to the corresponding string value of sumString and output the result to the Log.

Execution counts

As consumers, we are familiar with the process of evaluating products. When you choose between two automobiles to purchase, what factors influence your choice? For some people, speed and road handling may be the most important factors. Others may care about fuel economy. Some may be looking for luxury and a smooth ride. Most people are also concerned about price. These factors usually compete with one another in the car buyer’s mind. A car with much power and speed typically does not have good fuel economy. One that is luxurious does not come with a low price. In design terminology, that is a trade-off. The buyer may wish to trade off fuel economy to gain speed and power.

The space/time trade-off

The same type of problem emerges when we evaluate algorithms. Several competing factors are present. Usually, a gain of one property in an algorithm comes at the expense of another. Two important properties of an algorithm are the memory space required to store the program and its data, and the time necessary to execute the program. To compare several different algorithms that perform the same computation, we need a method of assessing these two properties. The following discussion presents a method for estimating the time necessary to execute a program.

One way to estimate the time necessary for an algorithm to execute is to count the number of statements that execute. If an algorithm has no IF statements or loops, then the number of statements that execute is simply the number of executable statements in the program listing.

If the algorithm has a loop, however, the number of statements in the listing is not equal to the number of statements executed. This is because the statements in the body may execute many times, even though they appear only once in the program listing.

Procedure ComputeTotal has 12 executable statements shown below. The declarations in the variable declaration part are not executable. Neither are the END reserved words.

The executable statements of procedure ComputeTotal

Statement number	*Executable statement*
(1)	cn := TextControllers.Focus()
(2)	IF nc # NIL THEN
(3)	md := cn.text
(4)	sc.ConnectTo(md)
(5)	sum := 0.0
(6)	sc.ScanReal(balance)
(7)	WHILE ~s.eot DO
(8)	sum := sum + balance
(9)	sc.ScanReal(balance)
(10)	PboxStrings.RealToString(sum, 1, 2, sumString)
(11)	StdLog.String("Total is $" + sumString)
(12)	StdLog.Ln

Even though the listing contains 12 executable statements, more than 12 statements execute. Statements (8) and (9) are part of a loop and may execute more than once. For the three real values in the focus window, statement (7) executes four times, and statements (8) and (9) each execute three times, as shown by the trace below.

A trace of procedure ComputeTotal with three data values in the focus window

Statement executed	sum	balance	sc.eot	sumString
(1)				
(2)				
(3)				
(4)				
(5)	0.0			
(6)	0.0	54.0	false	
(7)	0.0	54.0	false	
(8)	54.0	54.0	false	
(9)	54.0	20.4	false	
(7)	54.0	20.4	false	
(8)	74.4	20.4	false	
(9)	74.4	76.5	false	
(7)	74.4	76.5	false	
(8)	150.9	76.5	false	
(9)	150.9	?	true	
(7)	150.9	?	true	
(10)	150.9	?	true	“150.90”
(11)	150.9	?	true	“150.90”
(12)	150.9	?	true	“150.90”

So, the total number of executions is one each for statements (1), (2), (3), (4), (5), (6), (10), (11), and (12) for a total of nine, plus four for statement (7), plus three each for statements (8) and (9) for a total of six. The grand total is therefore nine plus four plus six, which is 19 statements executed.

If there were no data values in the focus window, then the trace would be as shown below, and 10 statements would execute.

Statement executed	sum	balance	sc.eot	sumString
(1)				
(2)				
(3)				
(4)				
(5)	0.0			
(6)	0.0	?	true	
(7)	0.0	?	true	
(10)	0.0	?	true	“0.00”
(11)	0.0	?	true	“0.00”
(12)	0.0	?	true	“0.00”

A trace of procedure ComputeTotal with no data values in the focus window

If the focus window contained n data values, then a total of $3n + 10$ statements would execute as shown in Figure 10.4.

Statement	No data values	Three data values	*n* data values
(1)	1	1	1
(2)	1	1	1
(3)	1	1	1
(4)	1	1	1
(5)	1	1	1
(6)	1	1	1
(7)	1	4	$n + 1$
(8)	0	3	n
(9)	0	3	n
(10)	1	1	1
(11)	1	1	1
(12)	1	1	1
Total:	10	19	$3n + 10$

Figure 10.4 Statement execution count for the procedure ComputeTotal in Figure 10.3.

Execution time estimates

You can use the general expression for the statement count, $3n + 10$, to estimate the execution time for a large number of data values, given the execution time for a small number of data values.

Example 10.1 Suppose you execute procedure ComputeTotal with 100 data values and it takes 140 μs (140 microseconds, which is 140×10^{-6} seconds). The problem is to estimate how long it would take to execute the program with 1000 data values.

Assuming that each executable statement takes the same amount of time to execute, simply form the ratio

$$\frac{140}{3 \times 100 + 10} = \frac{T}{3 \times 1000 + 10}$$

where T is the time to execute with 1000 data values. Solving for T gives

$$\frac{140}{310} = \frac{T}{3010}$$

or $T = 1359\ \mu\text{s} = 0.001359\ \text{s}$. ▪

The time of 0.00136 seconds is only an estimate. Each statement in procedure ComputeTotal does not execute in the same amount of time. Remember that the compiler must translate the Component Pascal source statements to object statements in machine language before it can execute. Typically, the compiler translates one source statement to more than one object statement. It may translate one source statement into three object statements and another source statement into five object statements. Furthermore, even the object statements do not execute in equal amounts of time. Under these circumstances, it is unreasonable to expect each source instruction to execute in the same amount of time.

The following example shows why the estimate works so well in practice. In dealing with large numbers of data, say hundreds or thousands for the value of *n*, the additive constants are insignificant to the final result and can be ignored.

Example 10.2 In the previous example, ignoring the additive constant, 10, in the expression can be justified because 310 is about equal to 300, and 3010 is about equal to 3000. Assuming that $3n + 10$ is approximately equal to $3n$, forming the ratio and solving for T then yields

$$\frac{140}{3 \times 100} = \frac{T}{3 \times 1000}$$

$$\frac{140}{100} = \frac{T}{1000}$$

or $T = 1400\ \mu\text{s} = 0.00140\ \text{s}$, which is not too different from our original estimate. ▪

Estimating the execution time by ignoring the additive constant and the coefficients

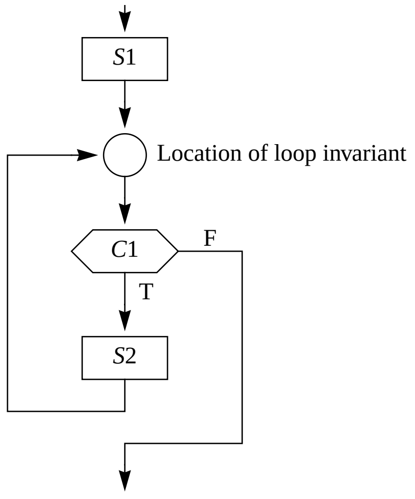
Notice that when you ignore the additive constant, the coefficient of *n*, which is 3, cancels in the ratio. Why is the coefficient of *n* unimportant in the estimate of the execution time for 1000 data values? Because for these large amounts of data, namely $n = 100$ and $n = 1000$, the number of statements executed is just about directly proportional to *n*. That implies that doubling the number of data values will double the number of statements executed, hence it will double the execution time. Or, as in this problem, multiplying the number of data values by 10 multiplies the execution time by 10.

Although the coefficient of *n* is unimportant in estimating the execution time for one algorithm with different amounts of data, it is important in comparing two different algorithms for the same job. If one algorithm requires $4n + 5$ statements to execute, and another algorithm to do the same processing requires $7n + 5$ statements to execute, the first will execute faster than the second with the same amount of data.

Loop invariants

A loop invariant is an assertion at the beginning of a loop. Because it is an assertion, it is a statement that is true at a specific point in a program. Figure 10.5 shows the point at which a loop invariant is true.

Figure 10.5 The location of the loop invariant for a WHILE loop.



(a) Flowchart.

```
Statement 1
(* Location of loop invariant *)
(* Loop invariant is true. *)
WHILE Condition1 DO
    Statement2
END
(* Loop invariant is true and Condition1 is false. *)
```

(b) Source code.

You can see from Figure 10.5(a) that there are two ways to get to the loop invariant. You can get to it from above by executing the statement sequence *S*1, or you can get to it from below by executing the statement sequence *S*2 in the body of the loop.

In the program of Figure 10.3, the statement sequence *S*1 is

```
sum := 0.0;
sc.ScanReal(balance)
```

the condition *C*1 is

```
~sc.eot
```

and the statement sequence *S*2 is

```
sum := sum + balance;
sc.ScanReal(balance)
```

For this program, the loop invariant is

- sum is the total of all the values scanned, not including the current value scanned into balance.

The loop invariant for Figure 10.3

To prove that a statement is a loop invariant, you must show two things:

Proving a loop invariant

- The statement is true initially because of the execution of *S*1.
- The statement is true at the end of each loop because of the execution of *S*2.

Now consider the proposed loop invariant for Figure 10.3. Because of the execution of *S*1, sum has value 0.0 and balance has value 54.00 assuming the values shown in Figure 10.2. But 0.0 *is* the total of all the values scanned, not including the 54.00 scanned into balance. It follows that the first part of the proof is true. To prove the second part, consider how *S*2 executes. *C*1 must be true at the beginning of *S*2. That is, sc.eot must be false. But regardless of whether you get to the body of the loop from above or from below, the statement

```
sc.ScanReal(balance)
```

was just executed. So, you know that you just did a scan into balance, after which sc.eot is false. Therefore, balance contains a valid real number. The statement sequence *S*2 adds the scanned value to sum, then does a scan. So after *S*2 executes, sum is once again the total of all the values scanned, not including the current value scanned into balance. It follows that the second part of the proof is true.

Consider what must be the case when a WHILE loop eventually terminates.

When a WHILE loop terminates

- The loop invariant is true.
- The loop condition is false.

The loop condition must be false, because that is the only way the loop can terminate. For the program of Figure 10.3, you know that after the loop terminates sum is the total of all the values scanned, not including the current value scanned into balance and that sc.eot is true. Because sc.eot is true, you know that the current value of balance should not be added to sum, which now contains the correct value.

Using loop invariants

This algorithm illustrates a common programming technique. Generally, when you program with loops, you should try to formulate a useful loop invariant. Establish the invariant before the loop executes the first time. Then, design the body of the loop so that each time it executes, the loop invariant becomes true when you reach the condition at the top of the loop from below. That is, you must write the body of the loop in such a way to reestablish the loop invariant.

The concept of a loop invariant may seem like much ado about nothing, or it may seem to be making a complicated point about something that appears simple. Using loop invariants to design programs is frequently a useful design technique that you will find aids in reasoning about your loops.

Using the Pbox scanners

Recall the rule for assignment statements that allows you to assign an integer value to a real variable, but does not allow you to assign a real value to an integer variable. A similar rule applies to a Pbox scanner. You can scan an integer value into a real variable, but you cannot scan a real value into an integer variable. Here is the documentation for ScanReal from module PboxMappers.

```
PROCEDURE (VAR s: Scanner) ScanReal (OUT x: REAL), NEW
Pre
s is connected to a text model.    20
Characters scanned represent a real or integer value.    21
Post
~s.eot
    x gets the next real or integer value scanned.
s.eot
    x gets MAX(REAL)
```

It shows that the text scanned can represent integer or real. If it is something else, such as a letter, a trap will be generated with error number 21.

Example 10.3 In Figure 10.2, if the input is

```
54  20.40  76.50
```

with the first number written as an integer, the program will execute correctly. However, if the input is

```
$54.00  $20.40  $76.50
```

a trap will be generated because of the dollar signs. ■

The documentation for ScanInt is

```
PROCEDURE (VAR s: Scanner) ScanInt (OUT n: INTEGER), NEW
Pre
s is connected to a text model.    20
Characters scanned represent an integer value.    21
Post
~s.eot
    n gets the next integer value scanned.
s.eot
    n gets MAX(INTEGER)
```

It shows that the text scanned must represent an integer.

Example 10.4 Suppose numEmp is an integer variable that is supposed to represent the number of employees in a company, and your program executes

```
sc.ScanInt(numEmp)
```

with the text

```
147.0
```

in the focus window. The program will trap with error number 21 because of the decimal point in the text. ▮

The documentation for both ScanInt and ScanReal shows that when sc.eot is true, that is, when no value is scanned because the end of text has been reached, the actual parameter gets its maximum possible value. See Example 4.15, page 60, for the definition of the MAX function.

If you ever stop doing a scan before the scanner reaches the end of the text model, the remaining values are simply not scanned or processed by your program.

Example 10.5 Suppose score is an integer variable and you execute the loop

```
sc.ScanInt(score);
WHILE ~sc.eot & (score <= 100) DO
    StdLog.String("score = "); StdLog.Int(score); StdLog.Ln;
    sc.ScanInt(score)
END
```

with the input containing the text

```
78  94  85  73  75  200 80  79
```

The program will scan the first five numbers and print them on the Log. Then it will scan the 200, but the condition will be false, because even though ~sc.eot is true, score <= 100 is false. The loop will not print the 200 to the Log, nor will it ever scan the 80 or the 79. ▮

Computing the average

Suppose you want to compute the average of the balances in the accounts. You would need not only their sum, but the number of accounts as well. The algorithm in Figure 10.6 uses another integer variable, numAccts, to count how many data values are in the focus window.

```
sum := 0.0;
numAccts := 0;
sc.ScanReal(balance);
WHILE ~sc.eot DO
    sum := sum + balance;
    INC(numAccts);
    sc.ScanReal(balance)
END;
IF numAccts > 0 THEN
    Output sum / numAccts
ELSE
    Output a no accounts message
END
```

Figure 10.6
An algorithm to find the average of all the data values in the focus window.

This algorithm illustrates a common programming technique. To determine how many times a loop executes, initialize a counting variable, in this algorithm numAccts, outside the loop to zero. Each time you scan a real value in the statement sequence of the loop, increment the counting variable by one. When the loop terminates, the value of the counting variable will be the number of values scanned by the loop. Before dividing by numAccts to compute the average, you must test to make sure that it is not zero. Otherwise your program may attempt a division by zero, which would cause a program trap by your user.

Finding the largest

The algorithm in Figure 10.7 finds the largest number from the focus window that contains integer values. If the input in the focus window is

```
73   80   -18   68   92   75
```

then the output is 92.

The two variables, num and largest, are integers. The algorithm works by scanning the first value from the text model into num. If the text model is empty the boolean sc.eot will be set to true, because a scan was attempted at the end of the text. In that case, the algorithm outputs a message indicating that the focus window is empty.

```
sc.ScanInt(num)
IF sc.eot THEN
    Output empty window message
ELSE
    largest := num
    sc.ScanInt(num)
    WHILE ~sc.eot DO
        IF num > largest THEN
            largest := num
        END
        sc.ScanInt(num)
    END
    Output largest
END
```

Figure 10.7
An algorithm to find the largest value in the focus window.

On the other hand, if sc.eot is false then an integer value has been scanned into num, 73 in this example. The first statement in the ELSE part initializes largest to the first value scanned. So now, both largest and num have the value 73. Then, the sc.ScanInt statement before the WHILE attempts to scan the second value from the text model. In this example, it would scan the 80 into num and set sc.eot false, because the scan was successfully executed before the end of text.

The first time the body of the loop executes, num is greater than largest, because it has the value 80. So, largest gets the value 80 from num. Then, num gets the next value from the focus window, –18. At this point, the WHILE statement is about to execute. You should be able to formulate the loop invariant for this program.

The loop invariant for Figure 10.7

- largest has the largest of all the values scanned, not including the current value scanned into num.

Can you see that this loop invariant is true just before the loop executes even for the first time?

The second time through the loop, the true alternative of the IF statement does not execute, because the value of num, which is now –18, is not greater than the value of largest, which is 80. Had the value of num been greater than 80, largest would have acquired that value and would still be the largest number scanned thus far. When the loop terminates the loop invariant will still be true. That is, variable largest has the largest value scanned so far, except for the last value scanned into variable num. Because no value is scanned into num when the scanner is at the end of text, largest will contain the largest of all the integer values in the focus window.

This algorithm illustrates a common programming technique. To save a value through successive loop iterations, declare a variable and initialize it appropriately. In the body of the loop, update the value with an assignment statement in the alternative of an IF statement as needed.

Real Expressions

Real values are approximate

You must be careful when you test real expressions in WHILE statements. Unlike integer values, real values have fractional parts, which the computer can store only approximately in main memory. The approximate nature of real values can cause endless loops if you do not design your WHILE tests properly.

Example 10.6 The following code fragment, where r is a real variable, is an endless loop:

```
r := 0.6;
WHILE r # 1.0 DO
   r := r + 0.1
END
```

It would seem that after r is initialized to 0.6, the loop would increase it to 0.7, 0.8, 0.9, and 1.0, at which point the loop would terminate. The problem is that r is never exactly 1.0 after those calculations. After four executions of the loop the value of r will be approximately one, not exactly one. ▮

The problem in Example 10.6 is that r was tested for strict inequality. In general, you should use the following rule for testing real values:

- Never test a real expression for strict equality, =, or strict inequality, #.

Tests for real values should always contain a less than or a greater than part.

Example 10.7 The previous example could be coded

```
r := 0.6;
WHILE r <= 0.95 DO
    r := r + 0.1
END
```

which would increase r to 0.7, 0.8, 0.9, and 1.0, at which point the loop would terminate. ▮

Minimizing the number of multiplications

The Component Pascal language does not have an operator that raises a value to a power. However, the Math library module has the function

```
PROCEDURE IntPower (x: REAL; n: INTEGER): REAL
```

For example, the expression $7x^3$ where x is a real variable can be written

```
7 * Math.IntPower (x, 3)
```

Without the function, the expression would be written 7 * x * x * x, which requires three multiplications. With the function, three multiplications are necessary inside the function itself. Multiplication of real values is one of the most time-consuming operations that the CPU can do. Polynomial expressions, which are sums of terms such as $7x^3$, are especially time consuming when they occur in loops that execute repeatedly. A common technique to minimize the computation time is to completely factor such expressions to reduce the number of real multiplications required. This technique will be used in the program in Figure 10.11.

Example 10.8 Suppose you need to evaluate $7x^3 + 2x^2 + 8x + 5$. Without factoring, the corresponding Component Pascal expression is

```
7 * x * x * x + 2 * x * x + 8 * x + 5
```

which requires six multiplications and three additions. On the other hand, if you completely factor the expression as $((7x + 2)x + 8)x + 5$ then the corresponding Component Pascal expression is

```
((7 * x + 2) * x + 8) * x + 5
```

which requires only three multiplications and three additions. ▮

The bisection algorithm

A numerical method is an algorithm that calculates a value or set of values that approximates the solution of a mathematical problem. The program in Figure 10.11 is a numerical method that computes one root of the cubic equation $x^3 - x^2 - 4x + 2 = 0$ with the bisection algorithm.

Figure 10.8 is a graph of the function $f(x) = x^3 - x^2 - 4x + 2$. The roots of the cubic equation are the values of x for which $f(x) = 0$. Figure 10.8 shows that $f(x)$ is zero for three different values of x: (a) between –2.0 and –1.0, (b) between 0.0 and

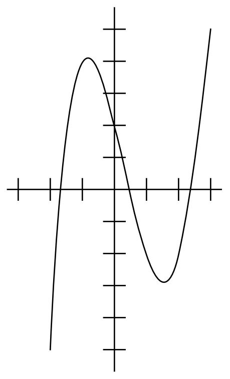


Figure 10.8
A graph of the function
$f(x) = x^3 - x^2 - 4x + 2$

1.0, and (c) between 2.0 and 3.0. Although this cubic equation has three roots, cubic equations in general can have from one to three roots. The program will determine the root of the equation that lies between $x = 2$ and $x = 3$.

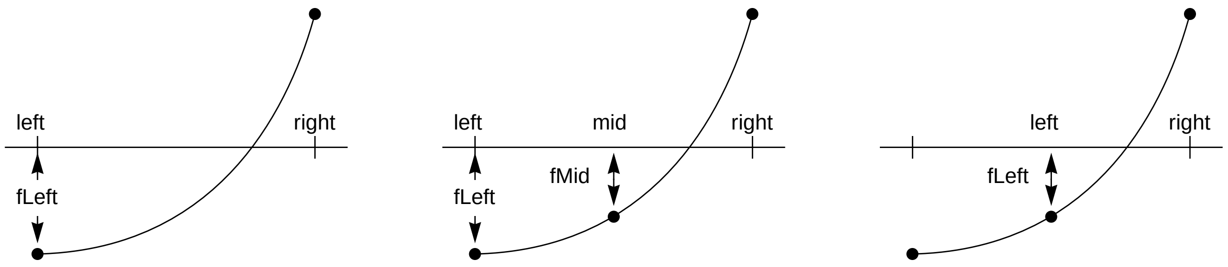
In the bisection algorithm, the variable left is a value of *x* that lies to the left of the root and the variable right is a value of *x* that lies to the right of the root. This algorithm initializes left to 2.0 and right to 3.0. Then, as Figure 10.9(a) shows, it calculates the variable fLeft as

```
fLeft := f(left)
```

The next step is to compute the value of *x* that is the midpoint between left and right. As Figure 10.9(b) shows, the algorithm gives that value of *x* to the variable mid and computes fMid as

```
fMid := f(mid)
```

The value of fMid determines whether the root lies to the left or right of mid. If fLeft and fMid have the same sign, then the root lies to the right of mid. Otherwise, the root lies to the left of mid. In the figure, fLeft and fMid have the same sign, because both are negative. Therefore, the root lies to the right of mid.



(a) Before the loop executes the first time. **(b)** Computation of mid and fMid. **(c)** Updating left and fLeft.

Figure 10.9 The bisection algorithm to find a root of *f*(*x*).

If the root lies to the right of mid, the algorithm changes the value of left and fLeft by

```
left := mid
fLeft := fMid
```

as Figure 10.9(c) shows. The root is still between left and right. Had the root lain to the left of mid, the algorithm would have changed the value of right by

```
right := mid
```

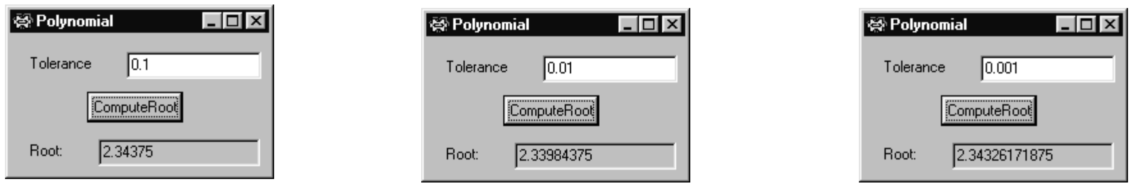
so that the root would still be between left and right.

The bisection algorithm continues to find the midpoint between left and right.

Each time it updates left or right, it decreases the interval between them such that the root is still in the interval. The loop invariant is the assertion that *the root is between* left *and* right. The loop terminates when left and right get close enough to satisfy a tolerance limit set by the user.

The loop invariant for the bisection algorithm

Figure 10.10 shows three executions of a program that implements the bisection algorithm. When the user enters 0.1, the bisection method calculates the root as 2.34375. It is accurate to the nearest tenth, so the last four digits, 4375, may not be significant. When the user enters 0.01, the loop executes more times and calculates the root as 2.33984375, a more accurate value for the root than 2.34375. The smaller the tolerance, the more times the loop executes and the more accurate is the value for the root. But then the program runs longer. So there is a trade-off between the accuracy of the solution and the execution time.



```
Tolerance  0.1
Root:      2.34375

Tolerance  0.01
Root:      2.33984375

Tolerance  0.001
Root:      2.34326171875
```

Figure 10.10
Three executions of the bisection algorithm of Listing 10.11.

Procedure ComputeRoot in Listing 10.11 implements the bisection algorithm in Component Pascal. The real value for the tolerance entered by the user is stored in d.tolerance. As long as the length of the interval is greater than the tolerance entered by the user, the loop executes. Each time the loop executes, the program halves the interval, which guarantees that the loop will eventually terminate.

If the user enters zero for the tolerance, the loop will execute endlessly, because continually halving the interval never permits it to reach zero. If the user enters a negative tolerance, the loop will execute endlessly because the boolean expression in the WHILE statement will always be true. The program could be improved by testing the tolerance for negative or zero values and not allowing them.

```
MODULE Pbox10B;
    IMPORT Dialog;
    VAR
        d*: RECORD
            tolerance*: REAL;
            root-: REAL
        END;

    PROCEDURE ComputeRoot*;
        CONST
            a3 = 1.0; a2 = -1.0; a1 = -4.0; a0 = 2.0;
        VAR
            left, fLeft: REAL;
            mid, fMid: REAL;
            right: REAL;
    BEGIN
        left := 2.0;
        fLeft := ((a3 * left + a2) * left + a1) * left + a0;
        right := 3.0;
        (* Assert: root is between left and right *)
        WHILE ABS(left - right) > d.tolerance DO
            mid := (left + right) / 2.0;
            fMid := ((a3 * mid + a2) * mid + a1) * mid + a0;
            IF fLeft * fMid > 0.0 THEN
                (* Assert: root is between mid and right *)
                left := mid;
                fLeft := fMid
            ELSE
                (* Assert: root is between left and mid *)
                right := mid
            END
        END;
        d.root := (left + right) / 2.0;
        Dialog.Update(d)
    END ComputeRoot;

BEGIN
    d.tolerance := 1.0;
    d.root := 0.0
END Pbox10B.
```

Figure 10.11
Computation of the root of a polynomial equation with the bisection algorithm.

Stepwise refinement

The next example of the WHILE loop illustrates a software development technique known as stepwise refinement. The data for this problem consists of a focus window containing an employee ID number followed by two real values that represent the number of hours worked per week and the hourly pay rate for that employee. The program must output to the Log a table with the employee ID number, hours worked, and weekly pay with the possibility of overtime. It must also determine the average salary of all the employees as well as the number of employees who earned overtime. For example, if the focus window contains the text

```
"123-A6002"  35.0  13.00
"123-A6517"  45.0  10.00
"561-B3882"  40.0  12.50
"561-B4559"  40.0  11.00
"561-B7384"  50.0  10.00
```

then the output to the Log should be

```
123-A6002    35.0      455.00
123-A6517    45.0      475.00
561-B3882    40.0      500.00
561-B4559    40.0      440.00
561-B7384    50.0      550.00
Average wages: 484.00
Number with overtime: 2
```

Stepwise refinement is based on the concept of abstraction. The idea is to not be concerned with all the details that are necessary for the final Component Pascal program, but instead to focus on the logic at a higher level of abstraction. The process consists of a number of steps or passes at the problem. At each pass, you get to a lower level of abstraction until you reach the final pass which produces the complete program. What follows is a description of the passes for a stepwise refinement solution for the above problem.

The first step is to determine the variables in the VAR section. Remember that input, processing, and output are the three major parts of a program. That gives a hint of the variables required.

Input—The input consists of a sequence of lines, each one of which contains a string and two real values. Hence you will need an array of characters, say empID, and two real variables, say hours and rate, for the input. You will need a scanner sc to input their values. While it is true that you will need a model and a view to get the input from the window, you should not be concerned with those details until the very last pass.

Processing—To compute the average, you must compute each wage and divide the total wages by the number of employees. Hence you will need real variables, wages and totalWages, to store the computed wage for an individual and for the total of all the wages. An integer variable, numEmp, will count the number of employees. The integer variable numOvertime will keep track of the number of employees who worked overtime.

Output—The three variables, hours, rate, and wages, will be used to output a single line in the report. aveWages will be a real variable for outputting the average wage at the bottom of the report. The value of numOvertime will also appear at the bottom.

The tentative variable declaration part now looks like this:

```
VAR
    sc: PboxMappers.Scanner;
    empID: ARRAY 16 OF CHAR;
    hours, rate: REAL;
    wages, totalWages, aveWages: REAL;
    numEmp, numOvertime: INTEGER;
```

In this problem, the number of variables and their types were fairly easy to determine before writing the logic of the program. With some problems it is not always possible to determine the variables beforehand. In general, you should determine the principal variables of the program at an early stage of the stepwise refinement. Then, augment the variable declaration part with new variables as refinement progresses.

First pass—The program in Figure 10.3 showed the technique of processing a set of values using the eot technique. The coding pattern is to perform a scan before the loop. In the body of the loop, process the data that was just read. After processing, scan the model for the next values as the last statements in the loop. Using this pattern, the first pass is the following:

```
Initialize variables
Input empID, hours, rate
WHILE ~sc.eot DO
    Process empID, hours, rate
    Input empID, hours, rate
END
Compute the average
Output aveWages, numOvertime
```

Second pass—Each pass in a stepwise refinement solution should concentrate on one aspect of the problem. This problem requires a table with a list of values and summary information at the bottom. The second pass will solve the table output part of the problem. Two kinds of lines appear in the body of the report, one for those who worked overtime and one for those who did not. This requires an IF statement in the body of the loop. The summary information appears once at the bottom of the report. The output statements for the summary must therefore be after the END of the WHILE loop. Here is the second pass:

```
Initialize variables
Input empID, hours, rate
WHILE ~sc.eot DO
    IF employee did not work overtime THEN
        Compute wages without overtime
    ELSE
        Compute wages with overtime
    END
    Output empID, hours, wages
    Input empID, hours, rate
END
Compute the average
Output aveWages, numOvertime
```

Third pass—This pass will solve the problem of computing the average and the number who worked overtime. The average is the sum of the wages divided by the number of employees. You can compute the sum by the technique of Figure 10.6. That example initialized the variable sum to 0.00 before the WHILE loop. Each time the loop executed, sum increased by the value input from the file. In this problem, you can initialize and increase totalWages the same way.

You can compute the number of employees using numEmp as a counting variable. Initialize numEmp to zero before the WHILE loop. Each time the body of the loop executes, increment numEmp by one. After the loop has terminated, the value of numEmp will equal the number of times the loop was executed, which equals the number of times a line was processed, which equals the number of employees.

Similarly, you can initialize numOvertime to zero before the loop. But now you only want to increment numOvertime by one if the employee worked overtime. The third pass is

```
totalWages := 0.0
numEmp := 0
numOvertime := 0
Input empID, hours, rate
WHILE ~s.eot DO
    IF employee did not work overtime THEN
        wages := hours * rate
    ELSE
        wages := 40.0 * rate + (hours - 40.0) * 1.5 * rate
        INC(numOvertime)
    END
    totalWages := totalWages + wages
    INC(numEmp)
    Output empID, hours, wages
    Input empID, hours, rate
END
IF numEmp > 0 THEN
    aveWages := totalWages / numEmp
ELSE
    aveWages := 0.00
END;
Output aveWages, numOvertime
```

Fourth Pass—This pass is the complete Component Pascal program shown in Figure 10.12.

This program shows several common stepwise refinement characteristics. In every pass except the last one, all the irrelevant details of input and output should be suppressed. The first three passes of this example used the pseudocode statements *Input* and *Output*. Only on the last pass were they converted to Component Pascal scans and StdLog procedures with formatting details.

```
MODULE Pbox10C;
    IMPORT TextModels, TextControllers, PboxMappers, PboxStrings, StdLog;

    PROCEDURE ProcessPayroll*;
        VAR
            md: TextModels.Model;
            cn: TextControllers.Controller;
            sc: PboxMappers.Scanner;
            empID: ARRAY 16 OF CHAR;
            hours, rate: REAL;
            wages, totalWages, aveWages: REAL;
            numEmp, numOvertime: INTEGER;
            outString: ARRAY 32 OF CHAR;
    BEGIN
        cn := TextControllers.Focus();
        IF cn # NIL THEN
            md := cn.text;
            sc.ConnectTo(md);
            totalWages := 0.0; numEmp := 0; numOvertime := 0;
            sc.ScanString(empID); sc.ScanReal(hours); sc.ScanReal(rate);
            WHILE ~sc.eot DO
                IF hours <= 40 THEN
                    wages := hours * rate
                ELSE
                    wages := 40.0 * rate + (hours - 40.0) * 1.5 * rate;
                    INC(numOvertime)
                END;
                StdLog.String(empID);
                PboxStrings.RealToString(hours, 8, 1, outString); StdLog.String(outString);
                PboxStrings.RealToString(wages, 12, 2, outString); StdLog.String(outString);
                StdLog.Ln;
                totalWages := totalWages + wages;
                INC(numEmp);
                sc.ScanString(empID); sc.ScanReal(hours); sc.ScanReal(rate)
            END;
            IF numEmp > 0 THEN
                aveWages := totalWages / numEmp
            ELSE
                aveWages := 0.00
            END;
            StdLog.String("Average wages: ");
            PboxStrings.RealToString(aveWages, 1, 2, outString); StdLog.String(outString); StdLog.Ln;
            StdLog.String("Number with overtime: ");
            PboxStrings.IntToString(numOvertime, 1, outString); StdLog.String(outString); StdLog.Ln;
        END
    END ProcessPayroll;

END Pbox10C.
```

Figure 10.12
A payroll report with summary information

Each pass should isolate and solve one specific part of the problem. In this program, the parts solved by each pass were the following:

- *First pass* input from the focus window
- *Second pass* output of the table
- *Third pass* computation of summary values
- *Fourth pass* Component Pascal details

The strategy here is to divide and conquer. If you have a large problem that you do not know how to solve, divide it into smaller subproblems that you can solve. Stepwise refinement gives you a framework for dividing a problem into smaller parts.

Another tip with stepwise refinement is to work it out on your text editor, not on paper. With each pass you can expand one pseudocode statement into several statements that are closer to Component Pascal, a job more easily accomplished on a screen than on a piece of paper. At the end of the last pass, the Component Pascal program will be on your disk ready to compile.

The structured programming theorem

Component Pascal provides several loop statements other than the WHILE statement, one of which is the FOR statement. Theoretically, there is no reason to provide any loop other than the WHILE. An important computer science theorem about the power of the WHILE statement coupled with the IF statement is known as the structured programming theorem, proved by Corrado Bohm and Guiseppe Jacopini in 1966. They proved mathematically that any algorithm, no matter how large or complicated, can be written with only three control statements—sequence, which is one statement following another, the IF statement, and the WHILE statement.

The structured programming theorem was proved by Bohm and Jacopini.

According to the structured programming theorem, Component Pascal really does not need to provide the CASE statement, for example. You can imagine that any program with a CASE statement could be written to perform the identical processing using an IF statement with several ELSIF parts. By the same token, any program with a FOR statement can be written to perform the identical processing using a WHILE statement. Nevertheless, statements like CASE and FOR are provided, not because of any additional power they give to the programmer, but because they are convenient.

The FOR statement

Suppose you want to compute the sum of all the integers from 1 to 100. You could use a WHILE loop, as in Figure 10.13. sum and i are variables of type INTEGER. i is called the control variable of the loop because its value controls when the loop terminates.

```
sum := 0
i := 1
WHILE i <= 100 DO
    sum := sum + i
    INC(i)
END
Output sum
```

Figure 10.13
An algorithm for the sum of consecutive integers with a WHILE loop.

If you execute the above algorithm, it will output 5050, which is the sum $1 + 2 + 3 + \ldots + 100$.

The sequence of steps

- Initialize a variable.
- Test the variable at the beginning of a loop.
- Execute the body of the loop.
- Increment the variable.

occurs frequently in programs. The FOR statement automatically initializes a control variable, tests it at the beginning of the loop, and increments it after executing the body of the loop. The program in Listing 10.14 is the above algorithm written in Component Pascal. It finds the sum of consecutive integers between one and an ending value entered by the user, but it is written with a FOR loop in place of the WHILE loop. Figure 10.15 shows the dialog box for this program.

When procedure ComputeSum executes, sum gets 0. Then, the statement

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

executes. The words FOR, TO, and DO are Component Pascal reserved words. The assignment statement between the reserved words FOR and TO gives an initial value to the control variable of the FOR statement. In this statement, i, the control variable, gets the initial value of 1.

The FOR statement then compares the current value of i with the expression after the reserved word TO. If the value of the control variable is greater than the expression, the loop terminates. Otherwise, the loop body executes. In this statement, the value of the control variable, 1, is not greater than the value of the expression, 100. The loop body executes, which adds 1 to sum.

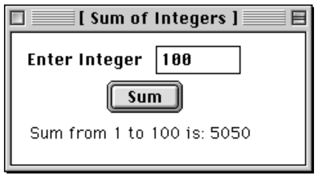
Control returns to the top of the loop. The FOR statement automatically increments the value of i with the equivalent of INC(i). It then compares the value of i with the expression after the reserved word TO. Because the current value of i, which is 2, is not greater than the value of the expression, which is 100, the body of the loop executes again. The loop continues executing, with i getting the values 1, 2, 3, and so on, to 100. After it executes with i having the value 100, the loop terminates.

```
MODULE Pbox10D;
   IMPORT Dialog, PboxStrings;
   VAR
      d*: RECORD
         num*: INTEGER;
         message-: ARRAY 64 OF CHAR
      END;

   PROCEDURE ComputeSum*;
      VAR
         sum, i: INTEGER;
         intString: ARRAY 16 OF CHAR;
   BEGIN
      sum := 0;
      FOR i := 1 TO d.num DO
         sum := sum + i
      END;
      PboxStrings.IntToString(d.num, 1, intString);
      d.message := "Sum from 1 to " + intString + " is: ";
      PboxStrings.IntToString(sum, 1, intString);
      d.message := d.message + intString;
      Dialog.Update(d)
   END ComputeSum;

BEGIN
   d.num := 0;
   d.message := ""
END Pbox10D.
```

Figure 10.14
Computing the sum of the first d.num integers with a FOR loop.



```
Sum from 1 to 100 is: 5050
```

Figure 10.15
The dialog box for the program of Figure 10.14.

You should be able to determine the statement execution count for this program. A total of $2n + 7$ statements execute, where n is the value input for d.num. That is $2(100) + 7$, or 207 statement executions for the computation of the sum of the first 100 integers.

The algorithm of Figure 10.14 may be a good illustration of the FOR statement, but it is not a good solution to the problem. The formula, $m(m + 1)/2$, gives the sum of the first m integers directly, as will be shown later. The following simpler algorithm solves the same problem. Assuming that the *Output* statement requires the same five statements as in Figure 10.14, this algorithm requires only six statement executions regardless of the value input for d.num.

```
sum := d.num * (d.num + 1) / 2
Output sum
```

Figure 10.16
A better algorithm for the sum of consecutive integers.

Using FOR Statements

The EBNF definition of the FOR statement is

```
FOR Ident ":=" Expr TO Expr [BY ConstExpr] DO StatementSeq END
```

If you use the [BY ConstExpr] option, the value of the control variable is changed by ConstExpr instead of by one.

The control variable can have steps other than 1.

Example 10.9 The following code fragment

```
FOR i := 1 TO d.num DO
    StdLog.Int(i); StdLog.String("  ")
END
```

where d.num and i are variables of type integer, outputs

```
1  2  3  4  5
```

to the Log if the value of d.num is 5. But the code fragment

```
FOR i := d.num TO 1 BY -1 DO
    StdLog.Int(i); StdLog.String("  ")
END
```

outputs

```
5  4  3  2  1
```

∎

The Component Pascal language report defines the FOR statement

```
FOR v := beg TO end BY step DO
    statements
END
```

to be equivalent to

```
temp := end;
v := beg;
IF step > 0 THEN
    WHILE v <= temp DO
        statements;
        v := v + step
    END
ELSE
    WHILE v >= temp DO
        statements;
        v := v + step
    END
END
```

where temp has the same type as v, and step must be a nonzero constant expression. If step is not specified, it is assumed to be 1. As is the case for the WHILE statement, it is possible for the body of the FOR statement to never execute.

It is possible for the body of the FOR statement to never execute.

Example 10.10 In the two code fragments of the previous example, if the value of d.num is zero, neither fragment will produce any output. ▮

In the EBNF definition of the FOR statement, either expression Expr can have any value—positive, negative, or zero.

Example 10.11 The code fragment

```
FOR i := d.num TO 5 DO
    StdLog.Int(i); StdLog.String("  ")
END
```

outputs

```
-3  -2  -1  0  1  2  3  4  5
```

if the value of d.num is –3. It outputs

```
3  4  5
```

if the value of d.num is 3. ▮

Do not change the value of the control variable in the body of a FOR statement.

It is frequently useful to use the control variable in an expression. Although it is legal to change the value of the control variable in the body of a FOR loop, it is extremely bad practice to do so. If you are ever tempted to change the value of the control variable in a FOR loop, you should redesign your algorithm using a WHILE loop instead of a FOR loop.

Example 10.12 This code fragment

```
FOR i := 1 TO d.num DO
    j := 2 * i - 1;
    StdLog.Int(j); StdLog.String("  ")
END
```

is legal and produces the output

```
1  3  5  7  9
```

if the value of d.num is 5. It is both legal and good practice to use i in the expression on the right side of the assignment statement. ▮

Example 10.13 The code fragment

```
FOR i := 1 TO d.num DO
    StdLog.Int(i); StdLog.String("  ")
    INC(i)
END
```

is legal but is extremely bad practice because the value of i is changed by the INC procedure. Never do this, even though the compiler allows it! ▮

The control variable is limited to integer or character type. Specifically, it cannot be of type real.

The control variable cannot be real.

Example 10.14 The following code fragment

```
FOR level := 0.5 to 6.5 DO
    StdLog.Real(level)
END
```

where level is a real variable is illegal because the control variable cannot be real. ▮

Example 10.15 The code fragment

```
FOR ch := ‘a’ TO ‘z’ DO
    StdLog.Char(ch)
END
```

where ch is a variable of type CHAR is legal and produces the following output on the Log

```
abcdefghijklmnopqrstuvwxyz
```
▮

★ The guarded command do statement

The statement that corresponds to the CP WHILE statement is the **do** statement in GCL. As with the **if** statement, the **do** statement uses a guarded command. The CP statement

```
WHILE C1 DO
    S1
END
```

is written in GCL as

do $C1 \rightarrow S1$ **od**

Example 10.16 The algorithm of Figure 10.6 to find the average is written in GCL as

```
s := 0.0; nA := 0; sc.ScanR(b);
do ¬sc.eot →  s := s + b; nA := nA + 1; sc.ScanR(b) od
if  nA > 0 → Output s / nA
 []  nA ≤ 0 → Output a no accounts message
fi
```
▮

Exercises

1. **(a)** What is a loop invariant? **(b)** What two things must you show to prove that a statement is a loop invariant? **(c)** What must be the case when a WHILE loop terminates?

2. For the algorithm in Figure 10.6 that computes the average of the accounts do the following: **(a)** Draw a flowchart. **(b)** Determine the total statement execution count if there are three data values. Include only the statements in the figure and ignore any statements that would be in a complete Component Pascal procedure. **(c)** Determine the total statement execution count if there are *n* data values. **(d)** If the algorithm executes in 50 μs for 200 data values, estimate the execution time for 10,000 data values from (c). Use both the exact computation and the approximate computation and compare how close they are by computing their percentage difference.

3. For the algorithm in Figure 10.7 that finds the largest value, do the following: **(a)** Draw a flowchart. **(b)** Determine the total statement execution count if there are three data values. Include only the statements in the figure and ignore any statements that would be in a complete Component Pascal procedure. **(c)** Determine the total statement execution count if there are *n* data values, assuming that the body of the nested IF statement executes every time. This is called the "worst case" time. **(d)** Determine the execution count assuming that the first number in the list is the largest. This is called the "best case" time. **(e)** If the algorithm executes in 120 μs for 80 data values, estimate the execution time for 5000 data values assuming the count in part (c). Use the exact computation. **(f)** Work part (e) assuming the count in part (d). **(g)** Give the percentage difference between the worst case and the best case times.

4. State the loop invariant for the WHILE loop in procedure ComputeTotal in Figure 10.3. The loop invariant will be a statement about the state of sum.

5. State the loop invariant for the WHILE loop in the algorithm to compute the average in Figure 10.6. The loop invariant will include statements about the states of both sum and numAccts.

6. Tell how many multiplication and addition steps are required to evaluate

$$ax^4 + bx^3 + cx^2 + dx + e$$

Factor the expression so that it requires only four multiplication and four addition steps.

7. Determine the output to the Log of the code fragment.

```
sum := 0;
WHILE i < 9 DO
    sum := sum + i;
    INC(i)
END;
StdLog.String("sum = "); StdLog.Int(sum)
```

for the following initial values of i.

(a) 5 **(b)** 8 **(c)** 9 **(d)** 0

8. Answer these questions for each of the Component Pascal code fragments. (1) What is the first value added to sum? (2) What is the last value added to sum? (3) What is the value of a at the termination of the loop? (4) How many times does the body of the loop execute?

(a)
```
sum := 0;
a := 50;
WHILE a < 100 DO
    INC(a, 2);
    sum := sum + a
END
```

(b)
```
sum := 0;
a := 50;
WHILE a < 100 DO
    sum := sum + a;
    INC(a, 2)
END
```

(c)
```
sum := 0;
a := 50;
WHILE a <= 100 DO
    INC(a, 2);
    sum := sum + a
END
```

9. What is the value of count after the statements are executed?

```
i := 15;
count := 0;
WHILE i <= 1000 DO
    INC(i, 2);
    INC(count)
END
```

10. Determine the output to the Log of the following code fragment.

```
sum := 0;
FOR i := 5 to d.num DO
    sum := sum + i
END;
StdLog.String("sum = "); StdLog.Int(sum)
```

for the following values of d.num.

(a) 10 **(b)** 6 **(c)** 5 **(d)** 4

11. Determine the total statement execution count of the code fragment in the previous exercise if d.num has the value *n*. Assume that *n* is greater than or equal to 5.

12. Determine the output to the Log of the following code fragment.

```
sum := 0;
FOR i := 10 TO d.num BY -1 DO
    sum := sum + i
END;
StdLog.String("sum = "); StdLog.Int(sum)
```

for the following values of d.num.

(a) 5 **(b)** 9 **(c)** 10 **(d)** 11

13. If procedure ComputeSum in Figure 10.14 takes 40 μs to execute when d.num has the value of 100, how long does it take to execute if d.num has the value of 150? Use both the exact computation and the approximate computation and compare how close they are by computing their percentage difference.

14. The Component Pascal language report defines the FOR statement in terms of an equivalent WHILE statement. **(a)** Using the definition, predict the output of the following code fragment.

```
last := 10;
FOR i := 0 TO last BY 2 DO
    StdLog.String("i = "); StdLog.Int(i); StdLog.Ln;
    StdLog.String("last = "); StdLog.Int(last); StdLog.Ln;
    INC(last)
END
```

(b) Execute the code fragment. Was your prediction correct?

15. Write the algorithm of Figure 10.7 to find the largest value in GCL.

16. Write the algorithm of Figure 10.13 to find the sum of consecutive integers in GCL.

Problems

17. Write a Component Pascal program that outputs to the Log the average of a list of values that are displayed in the focus window, or a statement that there are no values in the window.

18. Write a Component Pascal program that outputs to the Log the maximum integer value from a list of integers in the focus window. Output an appropriate message if no integer values are in the focus window.

19. Modify the program in Figure 10.11 so that it also outputs the number of times the WHILE loop executes. Run the modified program five times with the following tolerances: 1e–2, 1e–3, 1e–4, 1e–5, 1e–6. Graph the number of times the loop executes (y-axis) versus the tolerance (x-axis). What mathematical relationship did you discover between these two quantities? Hint: Use semilog graph paper or, equivalently, let each x-axis division be 10 times greater than the previous x-axis division. For what values of the tolerance, if any, will the loop never execute?

20. Modify the program in Figure 10.11 to always print the solution to four places past the decimal point and to print an error message if the user enters a negative number or zero for the tolerance.

21. Write a program to compute and output to a dialog box the sum of all positive even integers less than or equal to a value entered by the user in the dialog box.

22. Write a program to compute and output to a dialog box the sum of all positive odd integers less than or equal to a value entered by the user in the dialog box.

23. The focus window contains a list of integers. Write a program that counts how many even integers there are in the window. For example, if the focus window contains

```
5   38   1   -45   21   -7   12   5
```

the program should display a dialog box that says “There are 2 even integers in the window”. You can test if an integer `n` is even with `~ODD(n)`.

24. The focus window contains a list of integers. Write a program that counts how many positive integers are in the window. For example, if the focus window contains

```
5   38   1   -45   21   -7   12   5
```

the program should display a dialog box that says “There are 6 positive integers in the window”.

25. A linear sequence is a list of integers, each of which is a constant integer increment of the previous integer. For example,

```
3   6   9   12   15   18
```

is a linear sequence because each number is three plus the previous one. The constant increment need not be three, however. Write a program that inputs a list of numbers from the focus window and displays on a dialog box whether the sequence is linear, and the constant increment if it is. For example, the dialog box for the above list should state, “The sequence is linear with an increment of 3.” The message for the sequence,

```
3   6   9   12   16   19
```

should state, “The sequence is not linear.” Consider the empty list and any list with only one integer to be not linear.

26. A geometric sequence is a list of integers, each of which is a constant integer multiple of the previous integer. For example,

```
3   6   12   24   48   96
```

is a geometric sequence because each number is two times the previous one. The constant multiple need not be two, however. Write a program that inputs a list of numbers from the focus window and displays on a dialog box whether the sequence is geometric, and the integer multiple if it is. For example, the dialog box for the above list should state, “The sequence is geometric with a multiple of 2.” The message for the sequence,

```
3   6   12   24   46   92
```

should state, “The sequence is not geometric.” Consider the empty list and any list with only one integer to be not geometric.

27. A salesperson gets a 5% commission on sales of $1000 or less, and a 10% commission on sales in excess of $1000. For example, a sale of $1300 earns him $80—that is $50 on the first $1000 of the sale and $30 on the $300 in excess of the first $1000. The focus window contains a salesperson’s ID number (string) and his sales amount (real) on each line. Write a program that outputs to the Log a report containing the ID number, the amount of sales, and the commission for each salesperson. At the bottom of the report, print the ID number of the salesperson who sold the most. For example, if the focus window contains

```
"EM-00134"     580.00
"EM-01209"     600.00
"EM-00030"    1000.00
"EM-02238"    1200.00
"EM-09411"     800.00
"EM-02344"    1150.00
```

the report on the Log should be

```
EM-00134       580.00    29.00
EM-01209       600.00    30.00
EM-00030      1000.00    50.00
EM-02238      1200.00    70.00
EM-09411       800.00    40.00
EM-02344      1150.00    65.00
Highest sales ID: EM-02238
```

28. The price per Frisbee depends on the quantity ordered, as indicated in Figure 10.17. The focus window contains a list of numbers that represent order quantities of Frisbees. Write a program that outputs to the Log a report of order quantities and the cost per order. At the bottom of the report print the total cost of all the orders. For example, if the focus window contains

```
50   150   20   200   300   1   250   100
```

the report on the Log should be

```
   50        250.00
  150        450.00
   20        100.00
  200        500.00
  300        600.00
    1          5.00
  250        625.00
  100        300.00
Total: 2830.00
```

Quantity	**Price per Frisbee**
0 – 99	$ 5.00
100 – 199	3.00
200 – 299	2.50
300 or more	2.00

Figure 10.17 The price schedule for Problem 28.

29. An instructor determines the total score for each student according to the weights of Figure 10.18. A total of 90 to 100 is a grade of A, 80 to 89 is a B, and so on for C, D, and F. Each line of the focus window contains four real values that are the homework, exam, and final exam scores in that order. Write a program that outputs to the Log a table, each line of which contains

- The four values of each score
- The weighted total
- The letter grade

At the bottom of the report print the average of the weighted totals.

Score item	Percent toward total
Homework	15
Exam 1	25
Exam 2	25
Final exam	35

Figure 10.18 The grading weights for Problem 29.

30. The number of runs for each team in each inning of a Dodgers versus Giants baseball game is in the focus window. The Dodgers bat first. A complete game lasts nine innings. After the last Dodger out in the top of the ninth inning, if the Giants are ahead they do not bat in the bottom of the ninth. The game is over and the Giants win. In that case, no value is in the focus window for the number of Giants runs in the bottom of the ninth, not even a zero.

If it is a tie game or if the Dodgers are ahead after their last out, the Giants bat in the bottom of the ninth. After the last Giant out, whoever is ahead wins. If it is a tie, the game goes into extra innings, after which the same termination conditions apply as in the ninth inning.

Write a program that continues to input the runs for each inning until one team wins. Announce the winner, the score, and the number of innings played. Assume there are no input errors in the focus window. The program must not attempt to read past the end of text. Do not use the scanner eot variable.

The loop termination condition for this problem cannot be conveniently written as a single expression. Declare a boolean variable named over that indicates when the game is over. Initialize it to false before entering your WHILE loop, which should execute once for each inning. The test for termination should be on ~over. The processing at the bottom of the loop body should be a computation for determining if the game is over.

31. If a volleyball team serves the ball and wins the volley, they get 1 point and they get to serve again. If they serve the ball and lose the volley, the opposing team does not get a point. But the opposing team does win the right to serve next. The first team to get 15 points wins the game, except that they must win by at least 2 points. If the score is 15 to 14, play continues until one team is ahead by 2.

The focus window contains a sequence of 1’s and 0’s for a volleyball team that serves first in a game. A 1 represents winning the volley and a 0 represents losing the volley. For example, the sequence at the beginning of the file

```
1 1 0 1 0 0
```

represents

- Winning a volley and a point
- Winning a volley and a point
- Losing the serve
- Winning the serve back
- Losing the serve
- Losing the volley and a point

after which the team is ahead, 2 to 1.

Write a program that continues to scan the 1's and 0's until the game is over. Output the score and state whether the team won or lost. Assume there are no input errors in the focus window. The program must not attempt to read past the end of text. Do not use the scanner eot variable.

The loop termination condition for this problem cannot be conveniently written as a single expression. Declare a boolean variable named over that indicates when the game is over. Initialize it to false before entering your WHILE loop, which should execute once for each input value. The test for termination should be on ~over. The processing at the bottom of the loop body should be a computation for determining if the game is over.

32. A businessman wants to claim depreciation of an asset with the straight line method. If the asset has a useful life of *n* years, then $1/n$ of its original value is subtracted each year. Write a program that asks for the asset's value and its useful life span in a dialog box, and outputs a depreciation schedule to the Log using the straight line method. For example, if the user enters 1000.00 for the value of the asset and 5 years for the lifesaving the report on the Log should be

```
0    1000.00
1     800.00
2     600.00
3     400.00
4     200.00
5       0.00
```

33. A businessman wants to claim depreciation of an asset with the double declining balance method. If the asset has a useful life of *n* years, then $2/n$ times its current value is subtracted each year. Write a program that asks for the asset's value and its useful life-span, and outputs a depreciation schedule using the double declining balance method. For example, if the user enters 1000.00 for the value of the asset and 5 years for the lifesaving the report on the Log should be

```
0    1000.00
1     600.00
2     360.00
3     216.00
4     129.60
5      77.76
```

34. The factorial of an integer *n* is

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

For example, the factorial of 4 is 24, because $4 \cdot 3 \cdot 2 \cdot 1 = 24$. Zero factorial is defined to be one. Write a program that asks the user to input a nonnegative integer in a dialog box, then computes and outputs the factorial of that number in the dialog box. Use an integer field in the dialog box to output the factorial. If the number entered is negative, compute nothing and do not change any fields in the dialog box.

35. Suppose that *x* is a real nonzero number and *n* is an integer. Then *x* raised to the *n*th power, written mathematically as x^n, means

$$\begin{array}{rl} x \cdot x \cdot \ldots \cdot x & \text{if } n > 0 \\ 1.0 & \text{if } n = 0 \\ 1.0 / (x \cdot x \cdot \ldots \cdot x) & \text{if } n < 0 \end{array}$$

where there are *n x*'s in the first and last expressions. Without using the Math module, write a program that inputs a real number and an integer in a dialog box and raises the real number to the power indicated by the integer. For example, if the user enters 2.0 for the real number and -3 for the power, the dialog box should display 0.125 for the power. Use a real field in the dialog box to output the power.

36. The base of the natural logarithms, *e*, is approximated with four terms as

$$\frac{1}{1} + \frac{1}{1(1)} + \frac{1}{1(1)(2)} + \frac{1}{1(1)(2)(3)}$$

Notice that the fourth term is 1/3 times the previous term. In general, the *n*th term is 1/(*n* – 1) times the previous term. Write a program that asks the user to input the number of terms in a dialog box and outputs the approximation of *e*. Use two variables in additional to your control variable—sum, which represents the sum computed so far, and term, which represents the value of the current term. Initialize sum to the first term outside the loop, and start the loop with the second term. For example, if the user enters 4 for the number of terms, the value 2.6666… should be output. Use a real field in the dialog box to output the approximation of *e*. If the number entered is negative or zero, compute nothing and do not change any fields in the dialog box.

37. The average or mean of *n* numbers, $x_1, x_2, \ldots, x_n$, is

$$\begin{aligned} \bar{x} &= \frac{x_1 + x_2 + \ldots + x_n}{n} \\ &= \frac{1}{n} \sum_{i=1}^{n} x_i \end{aligned}$$

The standard deviation, a measure of how scattered the *n* numbers are, is defined as

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

when the numbers are a random sample of a population. If the numbers are all close to each other, they will all be close to the mean, their differences from the mean will be small, and the standard deviation will be small.

(a) The focus window contains a sequence of real values. Write a program that computes and outputs to a dialog box the standard deviation of the real numbers from the definition. Display the result with four places past the decimal point, and write an error message if there is only one value or no values in the focus window. You will need one loop to compute the mean, followed by a second execution of ConnectTo to position the scanner at the beginning of the text model again, followed by another loop for the squares of the differences from the mean.

(b) A mathematically equivalent formula for the standard deviation is

$$s = \sqrt{\frac{\sum_{i=1}^{n}x_i^2 - n\bar{x}^2}{n-1}}$$

(Can you derive this formula from the definitions of the mean and standard deviation?) This formula only requires one loop, because you can accumulate the sum of the squares at the same time you are accumulating the sum for the mean. Write a program that computes and outputs to a dialog box the standard deviation from this formula using only one loop. Display the result with four places past the decimal point, and write an error message if there is only one value or no values in the focus window.

38. An integer greater than 1 is prime if the only positive integers that divide it are 1 and the number itself. For example, 13 is prime because it is divisible only by 1 and 13, while 15 is not prime because it is divisible by 1, 3, 5, and 15. Write a Component Pascal program that asks the user to input a positive integer in a dialog box and then outputs a message to the dialog box indicating whether the integer is prime.

39. Write a program that asks the user to enter a positive integer in a dialog box, then outputs to the Log all the positive factors of that number. If the number entered is less than 1, compute nothing and output an appropriate message to the Log. For example, if the input is 15 then the output should be

```
1   3   5   15
```

40. The first two Fibonacci numbers are 0 and 1. The third Fibonacci number is the sum of the first pair, 0 plus 1, which is 1. The fifth is the sum of the previous pair, 1 plus 2, which is 3. The first seven Fibonacci numbers are

```
0   1   1   2   3   5   8
```

each number being the sum of the two previous numbers. Write a program that asks the user to enter an integer in a dialog box, and outputs to the Log that many Fibonacci numbers. If the number entered is less than 2, compute nothing and output an appropriate message to the Log. For example, if the user enters 7 the program should output the seven numbers listed above.

41. Implement a dialog box that looks and behaves like Figure 7.9, but with one additional button labeled “A Gets B”. When the user presses this button Stack A should be cleared and then given a copy of Stack B. The text fields for Push and Pop should not change. Only the field for the number of items in stack A should change. Use a temporary stack variable called tempStack in your procedure AGetsB to first store the value of Stack B in reverse order by popping all the items from Stack B into it. Then, clear Stack A and copy all the items from tempStack into it and back into Stack B. Import module PboxStackADT.

42. Work Problem 41, but import module PboxStackObj.

43. Implement a dialog box that looks and behaves like Figure 7.19, but with one additional button labeled “A Gets B”. When the user presses this button List A should be cleared and then given a copy of List B. All the text fields for the dialog box should not change. Only the field for the number of items in List A should change. Import module PboxListADT. Use a temporary variable called temp of type PboxListADT.T in your procedure AGetsB to store the value between the retrieve and insert operations.

44. Implement a dialog box that looks and behaves like Figure 7.3, but with one additional button labeled “Top To Bottom”. If the stack contains two or more items when the user presses this button, the top item on the stack should be moved to the bottom of the stack. Otherwise, the stack should remain unchanged. The text fields for Push and Pop should not change, nor should the field for the number of items change. Use a temporary variable called temp to store the top value of the stack and another temporary variable called tempStack to store the remainder of the stack in your procedure TopToBottom. Import module PboxStackADS.

45. Work Problem 44, but import module PboxStackObj.

46. Implement a dialog box that looks and behaves like Figure 7.19, but with one additional button labeled “A Append B”. If the length of List A plus the length of List B is greater than the capacity of a list when the user presses this button, nothing should happen. Otherwise, a copy of List B should be appended to the end of List A. All the text fields for the dialog box should not change. Only the field for the number of items in List A should change. Import module PboxListADT. Use a temporary variable called temp of type PboxListADT.T in your procedure AAppendB to store the value between the retrieve and insert operations.

47. Implement a dialog box that looks and behaves like Figure 7.16, but with one additional button labeled “Reverse”. When the user presses this button, all the items of the list should be rearranged in reverse order. All the text fields for the dialog box should not change. Import module PboxListADS. Use two local variables, i initialized to 0 and j initialized to Length() - 1. While i is less than j, switch the items at positions i and j followed by INC(i) and DEC(j). Use a temporary variable called temp of type PboxListADS.T in your procedure ReverseA to store the value between the retrieve, remove, and insert operations

48. Work Problem 47, but import module PboxListADT.