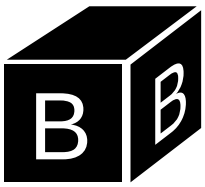


Chapter *15*

One-Dimensional Arrays

Recall that abstraction involves the suppression of detail. The collection of a group of items is generally the first step toward abstraction. The previous chapter presented proper procedures and function procedures, which are collections of program statements executed when one procedure calls another. Declaring a procedure creates a new statement that the calling procedure can use. One advantage of such declarations is that if different people design the calling procedure and the called procedure, the person who writes the calling procedure does not need to know about the collection of statements in the called procedure. This is particularly true when the called procedure is in a different module from the calling procedure. The collection of statements is a step toward program abstraction.

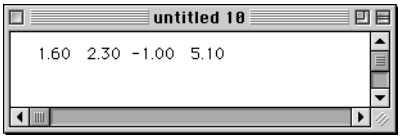
Program abstraction

We have already seen that records are collections of values, each of which may have different types. Arrays are also a collection of values. Unlike records, the values in an array must all be the same type. For example, it is possible for a record to have both an integer field and a real field. However, an array of integer values cannot contain a real value. In this chapter, you will learn how to declare and manipulate arrays. In the same way that the collection of statements is a step towards program abstraction, the collection of values is a step toward data abstraction.

Data abstraction

Array input/output

Figure 15.1 shows the input and output windows of a program that displays the input values in reverse order. Procedure ReverseReals in Figure 15.2 inputs four values from the focus window and outputs them in reverse order in the new window. It declares list to be an array of four real values. The procedure is invoked by a menu selection not shown in the figure.



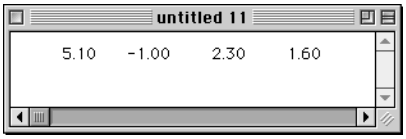


Figure 15.1
The input and output for the procedure of Figure 15.2.

```
MODULE Pbox15A;
   IMPORT TextModels, TextViews, Views, TextControllers, PboxMappers;

   TYPE
      Real4 = ARRAY 4 OF REAL;

   PROCEDURE ReverseReals*;
      VAR
         mdIn: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         list: Real4;
         i: INTEGER;
         mdOut: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         mdIn := cn.text;
         sc.ConnectTo(mdIn);
         FOR i := 0 TO 3 DO
            sc.ScanReal(list[i])
         END;
         mdOut := TextModels.dir.New();
         fm.ConnectTo(mdOut);
         FOR i := 3 TO 0 BY -1 DO
            fm.WriteReal(list[i], 8, 2)
         END;
         vw := TextViews.dir.New(mdOut);
         Views.OpenView(vw)
      END
   END ReverseReals;

END Pbox15A.
```

Figure 15.2
A program to reverse four real values in the focus window.

You could reverse the four values by declaring four variables, say list1, list2, list3, and list4. You could read them in with four sc.ScanReal statements and write them out with four fm.WriteReal statements. The disadvantage of this approach is that it is not feasible for large data sets. Would you like to write a program with this approach to reverse 100 values?

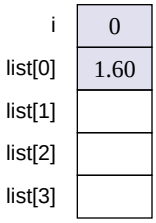
In this procedure, list is declared to have type Real4, which is declared to be

```
ARRAY 4 OF REAL
```

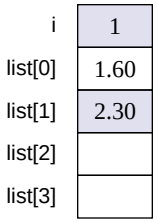
The declaration means that the array variable, list, contains four real values indexed from 0 to 3. The four values are referred to by list[0], list[1], list[2], and list[3]. An element of the array, say list[2], is also called a subscripted variable because of its similarity to subscripted variables in mathematics. In mathematical notation, if a variable

x is subscripted, you refer to its values by x_0, x_1, x_2, and x_3. Component Pascal syntax calls for the subscripts to be enclosed in square brackets. An array variable is also called a vector. An individual compartment that contains a value is called a cell of the array. Figure 15.3 shows the array list and the variable i allocated on the run-time stack. To keep the figure simple, the MVC variables are not shown.

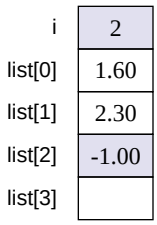
(a) Initially.



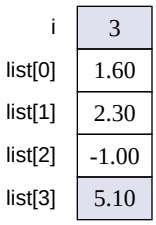
(b) After first scan.



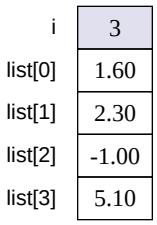
(c) After second scan.



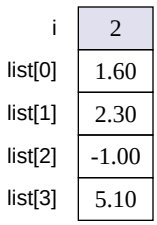
(d) After third scan.



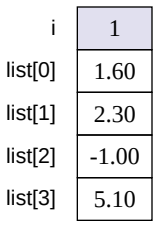
(e) After fourth scan.



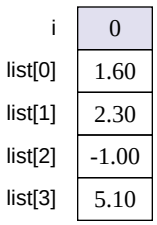
(f) Output list[3].



(g) Output list[2].



(h) Output list[1].



(i) Output list[0].

Figure 15.3
A trace of procedure ReverseReals in Figure 15.2.

The program works by setting up the usual MVC variables to scan from the model shown in the focus window. The first FOR loop

```
FOR i := 0 TO 3 DO
```

initializes i to 0. Then, the first time through the loop

```
sc.ScanReal(list[i])
```

scans the first value 1.60 from the text model into list[0], because the current value of

i is 0. Figure 15.3(b) shows the result of the scan. The next time through the loop i has value 1, so the effect of

```
sc.ScanReal(list[i])
```

is to scan the next value from the text model into list[1] as Figure 15.3(c) shows. Similarly, the third and fourth values are scanned into list[2] and list[3].

The second FOR loop outputs the values in reverse order to the text model for the output window. It initializes i to 3 and executes

```
fm.WriteReal(list[i], 8, 2)
```

which has the effect of writing list[3] to the output model, because the current value of i is 3. Notice in Figure 15.3(f), that the values maintain their order in the list array. The next time through the loop i has value 2, so the statement

```
fm.WriteReal(list[i], 8, 2)
```

has the effect of writing list[2] to the text model as Figure 15.3(g) shows. Similarly, list[1] and list[0] are written without having their order altered in the array.

Using arrays

You must remember that an array like list contains a collection of values, not just one value. To assign a value to a cell of an array you must specify which cell gets the value.

Example 15.1 The statement

```
list := -1.0
```

where list is declared as it is in procedure ReverseReals is illegal, even though –1.0 has type real. The problem is that the cell of list is not specified. On the other hand,

```
list[2] := -1.0
```

is legal. The assignment statement gives the value of –1.0 to the third cell of list. ∎

The index of list in procedure ReverseReals has a range of 0 to 3. During execution, the computer checks whether the index is within the allowable range whenever a reference to a cell of an array is made. If it is not, a trap occurs with an appropriate error message.

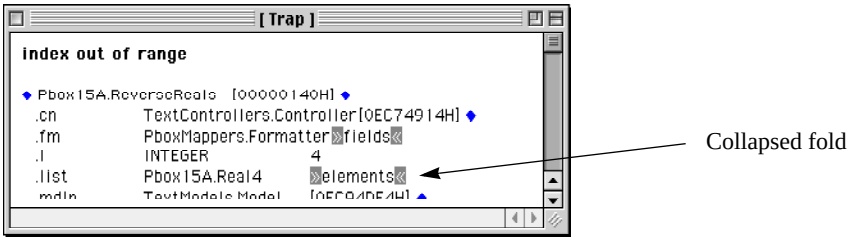
Example 15.2 If you erroneously change the second FOR statement in Figure 15.2 to

```
FOR i := 4 TO 0 BY -1 DO
```

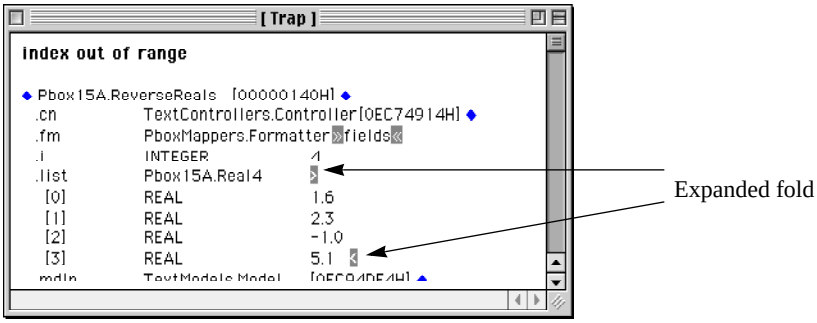
you will get a trap when the statement

```
fm.WriteReal(list[i], 8, 2)
```

executes because the first time through the loop i is not between 0 and 3. Figure 15.4 shows the top part of the trap window that results from the error. You can see in part (a) that the values of the list are not visible because they are hidden in the collapsed fold. Expanding the fold as in part (b) shows the value of the array.



(a) The values of list are hidden in the fold.



(b) Expanding the fold to see the values of list.

Figure 15.4
The trap window when your index is out of range.

The index is not limited to a constant or a single variable. It can be any arbitrary expression as long as the expression has type integer.

Example 15.3 The assignment statement

```
list[3 * i - 5] := 1.6
```

gives the value of 1.6 to list[1] if i has the value of 2.

Example 15.4 Suppose list gets the values

1.6 2.3 –1.0 5.1

from the model behind the focus window, as in procedure ReverseReals. The code

fragment

```
FOR i := 2 TO 5 DO
   fm.WriteReal(list[i MOD 4], 6, 1)
END
```

would output

```
 -1.0   5.1   1.6   2.3
```

These are the values of list[2], list[3], list[0], and list[1].

Memory allocation for arrays

If you do not know exactly how many data items will be in the array, you must allocate more space than you would reasonably expect. Figure 15.5 shows the input and output windows for a program that performs processing similar to that in Figure 15.2. The figure shows the output window displaying seven values from the focus window in reverse order, but the program will work for up to 1024 values in the focus window. Procedure ReverseReals in Figure 15.6 shows the technique. It allocates storage for 1024 real values in list, even though the focus window may contain fewer values.

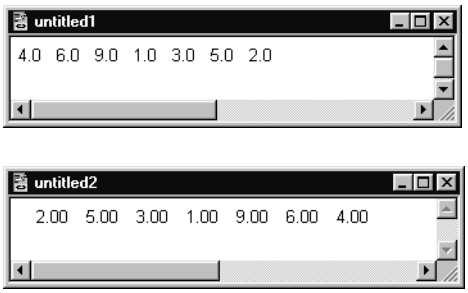


Figure 15.5
The input and output for the procedure of Figure 15.6.

The trace of the run-time stack for this program would be very large indeed. It would have 1024 cells just for list, including list[0], list[1], and so on, to list[1023]. In this example the focus window contained only seven real values. That means that the program did not use 1017 values. They remained undefined throughout the program execution and represent wasted memory.

The program inputs the values into the real array with procedure ScanRealVector from module PboxMappers. The documentation for ScanRealVector is

```
PROCEDURE (VAR s: Scanner) ScanRealVector (OUT v: ARRAY OF REAL; OUT numItm: INTEGER), NEW
Pre
s is connected to a text model.    20
Sequences of characters scanned represent in-range real or integer values.    21
Number of values in text model <= LEN(v).    Index out of range.
Post
v gets all the values scanned up to the end of the text model to which s is connected.
numItm gets the number of integer values scanned.
The values are stored at v[0..numItm - 1].
```

Both v and numItm are called by result. That means that they each refer to their corresponding actual parameter, and that their initial values when the procedure is called can be considered undefined. You can see that this is the case, because neither list nor numItems has been given any values before the call to ScanRealVector. The effect of the procedure is to change the values of both list and numItems. ScanRealVector is programmed to scan real values from a text model, skipping over any spaces, tabs, or line characters, until the end of the text is reached. It puts the values in vector v and in the process counts the number of values scanned and puts the value of the count in numItm. In the end, the values have been placed in v[0] to v[numItm - 1].

The type, ARRAY 1024 OF REAL, in the declaration of the array specifies a fixed number of elements in the array. You may be tempted to circumvent the problem of wasted memory by declaring list as

```
list: ARRAY numItems OF REAL
```

An illegal declaration

but this declaration is illegal because numItems is a variable. You must have a constant expression in the declaration of the size of your array. Storage allocation for variables in the procedure occurs before the first statement executes. The procedure cannot wait until numItems gets a value from

```
sc.ScanRealVector(list, numItems)
```

before allocating memory.

Open arrays

It is legal to declare a formal array parameter v with a fixed number of cells. For example, procedure ScanRealVector could have been declared as

```
(VAR s: Scanner) ScanRealVector (OUT v: ARRAY 1024 OF REAL; OUT numItm: INTEGER)
```

instead of as

```
(VAR s: Scanner) ScanRealVector (OUT v: ARRAY OF REAL; OUT numItm: INTEGER)
```

Both of these declarations are legal.

```
MODULE Pbox15B;
   IMPORT TextModels, TextViews, Views, TextControllers, PboxMappers;

   TYPE
      Real1024 = ARRAY 1024 OF REAL;

   PROCEDURE ReverseReals*;
      VAR
         mdIn: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         list: Real1024;
         numItems: INTEGER;
         i: INTEGER;
         mdOut: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         mdIn := cn.text;
         sc.ConnectTo(mdIn);
         sc.ScanRealVector(list, numItems);
         mdOut := TextModels.dir.New();
         fm.ConnectTo(mdOut);
         FOR i := numItems - 1 TO 0 BY -1 DO
            fm.WriteReal(list[i], 6, 2)
         END;
         vw := TextViews.dir.New(mdOut);
         Views.OpenView(vw)
      END
   END ReverseReals;

END Pbox15B.
```

Figure 15.6
A program to reverse any number of real values.

If the server module PboxMappers were designed with v having 1024 cells, as in the first declaration above, the program in Figure 15.6 would compile and run with no apparent difference to the user. An array specified without the number of elements it contains, as in the specification of v in the second declaration above, is known as an open array. A formal parameter list is one of the few places you can specify an open array. The advantage of specifying an open array in the formal parameter list of a procedure is that it makes the procedure more general than if you commit to an array of fixed size.

The advantage of open arrays

Example 15.5 Suppose the server module PboxMappers were designed with 1024 cells for v, as in the first declaration above. If you wanted the program of Figure 15.6 to process up to 2048 real values by declaring type

```
Real2048 = ARRAY 2048 OF REAL;
```

and variable

```
list: Real2048;
```

it would not compile. There would be a type conflict between actual parameter list, which would be an ARRAY 2048 OF REAL, and formal parameter v, which would be an ARRAY 1024 OF REAL. Because v is declared as an open array in procedure ScanRealVector, however, the procedure will work correctly regardless of the number of cells allocated for the actual parameter. ∎

One of the preconditions of procedure ScanRealVector is

```
Number of values in text model <= LEN(v).    Index out of range.
```

The LEN function

LEN is a built-in Component Pascal function that returns the length of an array regardless of the number of cells that are occupied by meaningful values. It is particularly useful in procedures that have open arrays in their parameter lists.

Example 15.6 The value of LEN(v) in procedure ScanRealVector is 1024 when it is called from procedure ReverseReals in Figure 15.6. ∎

In the program of Figure 15.6, if there are no more than 1024 values in the text model then numItems will get a value less than LEN(v). The program will execute with no ill effects. It will simply have some unused cells in the list array. But, if there are more than 1024 values in the text model the scanner will trap with an "index out of range" error message.

Wasted memory is a common problem in array processing and does not have a simple solution. With some programs you will know ahead of time exactly how much data must be processed and exactly how large to declare your array. With other programs, however, you will not know. In that case, you must decide what is reasonable for the problem at hand and for the main memory size of your computer.

A problem-solving technique

One skill you should develop is the ability to manipulate the elements of an array. Typically you will be confronted with a problem that requires the elements to be rearranged somehow, and you must write the statements that perform the re-arrangement. *Analysis* is determining the manipulation from given program statements, while *design* is determining the program statements from a given desired manipulation.

Analysis versus design

There are two approaches to program design problems. One approach is to go from the specific to the general. This technique involves generalizing from a small number of known patterns to a single general pattern. Another approach is to derive the general pattern using the methods of formal logic. These two approaches are at opposite ends of the inductive/deductive reasoning spectrum. Practitioners of each approach sometimes disparage the opposite approach. Both, however, are valuable

and should be mastered by the professional software designer. The usual practice is to use the generalization technique to determine the code initially, then use formal methods to prove that what you have written is correct.

Here are the steps of the generalizing technique:

- *Step 1*—Write some specific initial values for the array in a trace.
- *Step 2*—Perform the manipulation by changing the values in the trace, one at a time.
- *Step 3*—For each change, write a specific assignment statement that will produce the change.
- *Step 4*—Discover a pattern in the indices of the assignment statements you wrote. Generalize from the specific statements to a loop containing arrays with variables in the subscripts.

The last step is usually the hardest.

The following discussion presents a series of problems that require you to design a program or code fragment that manipulates the values of an array. Each problem is developed to show how you might use the generalizing technique.

The rotate left problem

The first illustration of this problem-solving technique is to rotate the elements of an array to the left. The leftmost element will rotate to the rightmost spot. For example, suppose list and numItems are declared as in procedure ReverseReals in Figure 15.6, numItems has the value 4, and list has the values

5.0 –2.3 8.0 0.1

Then, after the rotation, list should have the values

–2.3 8.0 0.1 5.0

Now you apply the four steps of the problem-solving technique.

Steps 1 and 2—In these steps, you write the values in a table and perform the changes one at a time.

	list[0]	list[1]	list[2]	list[3]
Original values	5.0	–2.3	8.0	0.1
Change list[0]	–2.3	–2.3	8.0	0.1
Change list[1]	–2.3	8.0	8.0	0.1
Change list[2]	–2.3	8.0	0.1	0.1
Change list[3]	–2.3	8.0	0.1	5.0

Step 3—For each change, you must write a specific assignment statement that will produce the change.

	list[0]	list[1]	list[2]	list[3]
Original values	5.0	–2.3	8.0	0.1
list[0] := list[1]	–2.3	–2.3	8.0	0.1
list[1] := list[2]	–2.3	8.0	8.0	0.1
list[2] := list[3]	–2.3	8.0	0.1	0.1
list[3] := ?	–2.3	8.0	0.1	?

But here you have a problem. You want list[3] to get the old value of list[0]. But if you write

```
list[3] := list[0]
```

then list[3] will get the current value of list[0], which is –2.3, not 5.0. The solution is to employ a temporary real variable, say temp, which saves the old value of list[0]. Here is a revised trace:

	temp	list[0]	list[1]	list[2]	list[3]
Original values		5.0	–2.3	8.0	0.1
temp :=list[0]	5.0	5.0	–2.3	8.0	0.1
list[0] := list[1]	5.0	–2.3	–2.3	8.0	0.1
list[1] := list[2]	5.0	–2.3	8.0	8.0	0.1
list[2] := list[3]	5.0	–2.3	8.0	0.1	0.1
list[3] := temp	5.0	–2.3	8.0	0.1	5.0

Step 4—In this step, you discover a pattern in the indices of the assignment statements you wrote. The pattern in the indices just presented is

```
0   1
1   2
2   3
```

The index on the right of the assignment statement is one more than the index on the left. So the generalization is

```
temp := list[0];
FOR i := 0 TO 2 DO
   list[i] := list[i + 1]
END;
list[3] := temp
```

in the case where the array has four elements. In the more general case where there are numItems values, the statements are

```
temp := list[0];
FOR i := 0 TO numItems - 2 DO
   list[i] := list[i + 1]
END;
list[numItems - 1] := temp
```

```
MODULE Pbox15C;
   IMPORT TextModels, TextViews, Views, TextControllers, PboxMappers;

   TYPE
      Real1024 = ARRAY 1024 OF REAL;

   PROCEDURE RotateLeft (VAR v: ARRAY OF REAL; numItm: INTEGER);
      VAR
         i: INTEGER;
         temp: REAL;
   BEGIN
      ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
      IF numItm > 1 THEN
         temp := v[0];
         FOR i := 0 TO numItm - 2 DO
            v[i] := v[i + 1]
         END;
         v[numItm - 1] := temp
      END
   END RotateLeft;

   PROCEDURE ProcessRotation*;
      VAR
         mdIn: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         list: Real1024;
         numItems: INTEGER;
         mdOut: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         mdIn := cn.text;
         sc.ConnectTo(mdIn);
         sc.ScanRealVector(list, numItems);
         mdOut := TextModels.dir.New();
         fm.ConnectTo(mdOut);
         fm.WriteRealVector(list, numItems, 6, 2);
         RotateLeft(list, numItems);
         fm.WriteLn; fm.WriteLn;
         fm.WriteRealVector(list, numItems, 6, 2);
         vw := TextViews.dir.New(mdOut);
         Views.OpenView(vw)
      END
   END ProcessRotation;

END Pbox15C.
```

Figure 15.7
A program with a procedure to rotate the elements in an array.

Figure 15.7 shows this algorithm implemented in a procedure called RotateLeft. Figure 15.8 shows the input and output windows of the program. It generalizes the problem to work for any number of elements, with seven elements shown in the figure.

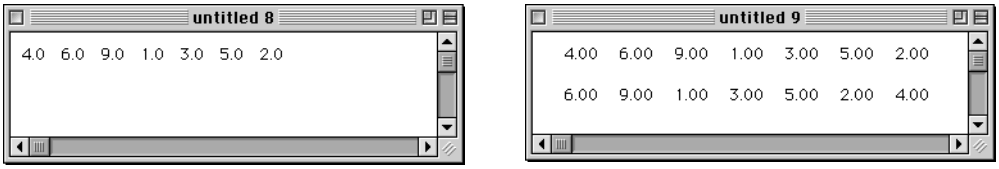


Figure 15.8
The input and output for the procedure of Figure 15.7.

Procedure RotateLeft has formal parameter v that corresponds to actual parameter list and formal parameter numItm that corresponds to actual parameter numItems. The precondition for RotateLeft to work correctly is that numItm have a value between 0 and LEN(v). As is the case with procedure ScanRealVector, v is an open array so that procedure RotateLeft could work with an array of any length.

Example 15.7 Suppose you have two types declared as follows.

```
TYPE
   Real128 = ARRAY 128 OF REAL;
   Real1024 = ARRAY 1024 OF REAL;
```

and two local arrays

```
VAR
   myArray: Real128;
   yourArray: Real1024;
```

If you want to rotate each array and you do not write your procedure with an open array, you must write *two* procedures, one with declaration

```
PROCEDURE MyRotateLeft (VAR v: Real128; numItm: INTEGER)
```

that you call with

```
MyRotateLeft (myArray, myNumItems)
```

and one with declaration

```
PROCEDURE YourRotateLeft (VAR v: Real1024; numItm: INTEGER)
```

that you call with

```
YourRotateLeft (yourArray, yourNumItems)
```

However, if you have the open array in the formal parameter list you only need to write *one* procedure as in Figure 15.7 and call it with either array as

```
RotateLeft (myArray, myNumItems);
RotateLeft (YourArray, yourNumItems)
```

It is usually best to not commit to a fixed array length when you design a procedure to process an array. Most server modules provide procedures with open array parameters as does ScanRealVector as shown on page 319. Because v is an open array, your client module can call it with an actual parameter having any length you desire. Because of the advantage of using open arrays you should get in the habit of using them in the formal parameter lists of your procedures.

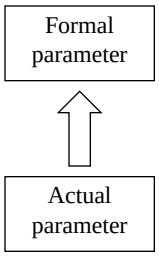
Call by constant reference

The program uses the procedure WriteRealVector from PboxMappers. Here is its documentation.

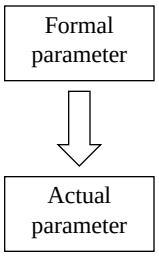
```
PROCEDURE (VAR f: Formatter) WriteRealVector (IN v: ARRAY OF REAL; numItm, minWidth, dec: INTEGER),
    NEW
Pre
f is connected to a text model.    20
numItm <= LEN(v).    Index out of range.
Post
The first numItm values of v are written to the text model to which f is connected,
each with a field width of minWidth and dec places past the decimal point. If minWidth
is too small to contain a value of v it expands to accommodate the value.
```

The calling procedure supplies WriteRealVector with the vector to write v, the number of items in v to write numItm, the field width minWidth, and the number of places past the decimal point dec. The designation IN signifies call by constant reference.

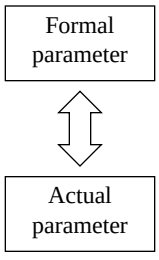
The purpose of call by constant reference is identical to the purpose of call by value. Namely, the calling procedure desires to give a value to the called procedure. In that sense we can revise Figure 12.17 to include call by constant reference as shown in Figure 15.9



(a) Call by value and call by constant reference.



(b) Call by result.



(c) Call by reference.

Figure 15.9
The flow of information for four different calling mechanisms, including call by constant reference.

If call by constant reference has the same effect as call by value, then why does Component Pascal provide both calling mechanisms? The answer is efficiency. In fact, procedure WriteRealVector would work if v were called by value. However, remember that the array corresponding to the actual parameter has 1024 elements. In call by value, the formal parameter gets the value of the actual parameter. Hence, all 1024 elements would be pushed onto the run-time stack when the procedure is called. That action would take much time and would consume much space on the stack. It would be more efficient if v were called by reference. Then only a single cell on the run-time stack would be necessary, and it would contain a reference to the actual parameter.

But the purpose of call by reference is for information to flow in both directions, as shown in Figure 15.9(c). In procedure WriteRealVector, the information is only supposed to flow in one direction as in Figure 15.9(a). So in call by constant reference, Component Pascal pushes a reference to the actual parameter on the run-time stack. At the same time the compiler forbids the called procedure to change the value of the formal parameter. The result is to achieve the effect of call by value but with the efficiency of call by reference.

Now you may be asking, If call by constant reference has the efficiency advantage of call by reference and the effect of call by value, why have call by value in the first place? The answer is twofold. First, call by value has the advantage that the actual parameter can be any expression of the proper type. It need not be a single variable. In call by constant reference the actual parameter must be a single variable.

Figure 15.10
Guidelines for using the four parameter calling mechanisms.

	Integers, reals, booleans, pointers, short arrays and short records	Long arrays and long records
Call by value Default	Common. Use when the actual parameter should *not* change. Actual parameter can be an expression.	Not common. Inefficient procedure call. Use call by constant reference instead.
Call by constant reference IN	Not common (illegal for all types except arrays and records). Inefficient procedure execution. Use call by value instead.	Common. Use when the actual parameter should *not* change. Actual parameter must be a variable.
Call by result OUT	Common. Use when the actual parameter *should* change and its initial value is *undefined*. Actual parameter must be a variable.	
Call by reference VAR	Common. Use when the actual parameter *should* change and its initial value is *defined*. Actual parameter must be a variable.	

Second, the efficiency in call by constant reference is in the procedure call. It is actually less efficient after the procedure call and during execution of the called procedure. It is therefore more efficient in total to use call by value for integers, reals, booleans, pointers (described later in this book), and short arrays and records, and to use call by constant reference for long arrays and records. Figure 15.10 summarizes all these ideas for the four calling mechanisms.

Finding the largest value

The next illustration of this problem-solving technique involves finding the largest value of an array. For example, in the function procedure

```
PROCEDURE Maximum (IN v: ARRAY OF REAL; numItm: INTEGER): REAL;
```

if v has the values

5.0 –2.3 8.0 0.1

and numItm has the value 4, then the procedure should return the value 8.0.

The basic idea is the same as the algorithm to find the largest value in the focus window. That algorithm saves the largest value found so far in a variable. Each time the loop executes, the algorithm scans a new value from the model displayed in the focus window. If the value scanned is greater than the largest found to that point, the algorithm updates the variable with the newly scanned value. The algorithm we will now discuss uses the same logic, but it compares largest with the items of the array one at a time.

The first three steps of the problem-solving technique require you to write some specific initial values for the array in a trace table. Then change the values one at a time, and for each change, write a specific assignment statement that will produce the change. The following trace shows one possibility.

	largest	v[0]	v[1]	v[2]	v[3]
Original values		5.0	–2.3	8.0	0.1
largest :=v[0]	5.0	5.0	–2.3	8.0	0.1
IF v[1] > largest THEN	5.0	5.0	–2.3	8.0	0.1
update largest	5.0	5.0	–2.3	8.0	0.1
IF v[2] > largest THEN	5.0	5.0	–2.3	8.0	0.1
update largest	8.0	5.0	–2.3	8.0	0.1
IF v[3] > largest THEN	8.0	5.0	–2.3	8.0	0.1
update largest	8.0	5.0	–2.3	8.0	0.1

You must now discover a pattern in the indices and generalize. The pattern in the indices in the comparisons is

1
2
3

So the statements, one of which is a loop, are

```
largest := v[0];
FOR i := 1 TO 3 DO
    IF v[i] > largest THEN
        largest := v[i]
    END
END;
```

in the case where the array has four elements. In the more general case where there are numItm values, you should replace the constant 4 by numItm - 1. Figure 15.11 is the completed function. A precondition for the function to work is that numItm be strictly greater than zero. Otherwise there is no largest element and it would not make sense to call the procedure. The precondition should be verified in the calling procedure.

```
PROCEDURE Maximum (IN v: ARRAY OF REAL; numItm: INTEGER): REAL;
    VAR
        i: INTEGER;
        largest: REAL;
BEGIN
    ASSERT((0 < numItm) & (numItm <= LEN(v)), 20);
    largest := v[0];
    FOR i := 1 TO numItm - 1 DO
        IF v[i] > largest THEN
            largest := v[i]
        END
    END;
    RETURN largest
END Maximum;
```

Figure 15.11
A function that returns the largest element in an array.

What is the total statement execution count for the algorithm of Figure 15.11? ASSERT statements do not count for execution purposes, because they do no data processing. Their purpose is to specify a procedure, which would execute the same without them. Clearly the initialization of largest executes one time. If numItm has the value *n*, the FOR statement executes *n* times. Furthermore, every time the body of the FOR loop executes, the test of the nested IF statement executes. But how many times does the assignment statement

```
largest := v[i]
```

execute? The answer is, It depends. You cannot predict exactly how many times it executes, because you do not know how many times the IF test will be true. In the best case, the IF test will never be true and the assignment will never execute. In the worst case, the IF test will always be true and the assignment will always execute. The average case depends on the original arrangement of the data values in v, and is somewhere between the best case and the worst case. It is an exercise for the student (Exercise 8) to determine the execution count.

Definition of best-case and worst-case execution count

Exchanging the largest with the last

The next problem is to switch the largest value of an array with the last value. For example, if list is declared as before with the same initial values

```
5.0   –2.3   8.0   0.1
```

then, after the processing, the values should be

```
5.0   –2.3   0.1   8.0
```

For a first attempt, you might try to find the largest number using function Maximum above. Namely, suppose you have computed that largest has the value 8.0. Now, how would you make the exchange? The following trace shows the specific statements.

	temp	largest	v[0]	v[1]	v[2]	v[3]
Original values		8.0	5.0	–2.3	8.0	0.1
temp := v[3]	0.1	8.0	5.0	–2.3	8.0	0.1
v[3] := largest	0.1	8.0	5.0	–2.3	8.0	8.0
v[2] := temp	0.1	8.0	5.0	–2.3	0.1	8.0

How do you generalize the last assignment statement in the trace? Where did the 2 in v[2] come from? The problem is that we have the value of the largest element, when what we really need is the index of the largest element to make the exchange.

So, instead of saving the largest value in the array, we must save the *index* of the largest value in the array. An integer variable, say indexMax, will save the value of the index of the largest element found so far. The following trace shows the specific statements to compute indexMax.

	indexMax	v[0]	v[1]	v[2]	v[3]
Original values		5.0	–2.3	8.0	0.1
indexMax := 0	0	5.0	–2.3	8.0	0.1
IF v[1] > v[indexMax] THEN					
update indexMax	0	5.0	–2.3	8.0	0.1
IF v[2] > v[indexMax] THEN					
update indexMax	2	5.0	–2.3	8.0	0.1
IF v[3] > v[indexMax] THEN					
update indexMax	2	5.0	–2.3	8.0	0.1

The statements in the form of a loop are

```
indexMax := 0;
FOR i := 1 TO 3 DO
   IF v[i] > v[indexMax] THEN
      indexMax := i
   END
END;
```

which is valid when there are four items in the list. In the general case, you must replace the 3 by numItem - 1. The algorithm is shown as the procedure in Figure 15.12. If numItem is not greater than one, there is no need to exchange at all. The procedure works fine even when given an array with one item or an empty array.

```
PROCEDURE LargestLast (VAR v: ARRAY OF REAL; numItm: INTEGER);
   VAR
      i, indexMax: INTEGER;
      temp: REAL;
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   IF numItm > 1 THEN
      indexMax := 0;
      FOR i := 1 TO numItm - 1 DO
         IF v[i] > v[indexMax] THEN
            indexMax := i
         END
      END;
      temp := v[numItm - 1];
      v[numItm - 1] := v[indexMax];
      v[indexMax] := temp
   END
END LargestLast;
```

Figure 15.12
A procedure that exchanges the largest element in an array with the last element.

Initializing in decreasing order

The previous problems were rearrangements of existing values in the array. Some problems call for initializing the values in the array. Here is an example. Suppose list is an array of integers. If numItem has the value 5, you must initialize list to

4 3 2 1 0

In general, the values should be in decreasing order, with the first value equal to the number of items minus 1 and the last value equal to 0.

The specific assignment statements for five elements are

```
v[0] := 4
v[1] := 3
v[2] := 2
v[3] := 1
v[4] := 0
```

You must discover the general relationship between the pairs

0 4
1 3
2 2
3 1
4 0

Each time the first integer in the pair increases by one, the second integer decreases by one. If you write the FOR loop

```
FOR i := 0 TO 4 DO
   v[i] := some expression
END
```

the expression must decrease as i increases. An expression with -i satisfies that requirement. As i increases, -i decreases. Namely, the expression 4 - i works for five elements. When i has the value 0, 4 - i has the value 4. When i has the value 4, 4 - i has the value 0.

For four elements the pattern is

0 3
1 2
2 1
3 0

and the expression is 3 - i. For general values of numItm, the expression is

```
numItm - i - 1
```

When i has the value 0, numItm - i - 1 has the value numItm - 1. So, v[0] := numItm - 1. When i has the value numItm - 1, numItm - i - 1 has the value 0. So, v[numItm - 1] := 0. The algorithm in procedure form is in Figure 15.13.

```
PROCEDURE Initialize (OUT v: ARRAY OF INTEGER; numItm: INTEGER);
   VAR
      i: INTEGER;
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   FOR i := 0 TO numItm - 1 DO
      v[i] := numItm - i - 1
   END
END Initialize;
```

Figure 15.13
A procedure to initialize the elements of an array to a decreasing sequence.

Another approach to the same problem is to note that the expression always starts with the value of numItem - 1 and decreases by one. You can declare an integer variable, j, and initialize it to numItem - 1. Decrement the value of j each time through the loop and simply give the value of j to v[i]. The algorithm in procedure form is in Figure 15.14.

```
PROCEDURE Initialize (OUT v: ARRAY OF INTEGER; numItm: INTEGER);
   VAR
      i, j: INTEGER;
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   j := numItm - 1;
   FOR i := 0 TO numItm - 1 DO
      v[i] := j;
      DEC(j)
   END;
END Initialize;
```

Figure 15.14
A procedure that performs the identical processing to that in Figure 15.13.

Character arrays

Recall from Chapter 4 that arrays of characters are terminated with the sentinel value 0X. In that respect they are different from arrays of other types because other arrays are not so terminated. However, like arrays of other types, arrays of characters can be manipulated by subscripting.

The $ string selector

The $ symbol denotes the Component Pascal string selector, which affects the processing of arrays of characters, but not arrays of other types. The $ symbol, when it follows the name of a variable of type array of character, signifies all the values from the first up to and including the cell containing the 0X sentinel character. The purpose of the $ string selector is to make string assignments more efficient than would otherwise be the case.

Example 15.8 Suppose you declare the type

```
TYPE
   String16 = ARRAY 16 OF CHAR;
```

and you have two variables declared as

```
VAR
   d*: RECORD
      stringIn*: String16;
      stringOut-: String16
   END;
```

The value of d.stringIn is "each" and you want to assign it to d.stringOut, so you write

```
d.stringOut := d.stringIn
```

Without the $ symbol after d.stringIn, the values represented by d.stringIn include all 16 cells of the array. The assignment statement would cause 16 values to be assigned to d.stringOut as Figure 15.15(a) shows. This is clearly a waste of time, because for an array of characters all the values after the 0X sentinel are irrelevant. With the $

symbol after d.stringIn, the values represented by d.stringIn$ consists of the five values d.stringIn[0], d.stringIn[1], …, d.stringIn[4]. The assignment statement

```
d.stringOut := d.stringIn$
```

only needs to give five values to d.stringOut as Figure 15.15(b) shows instead of 16 values.

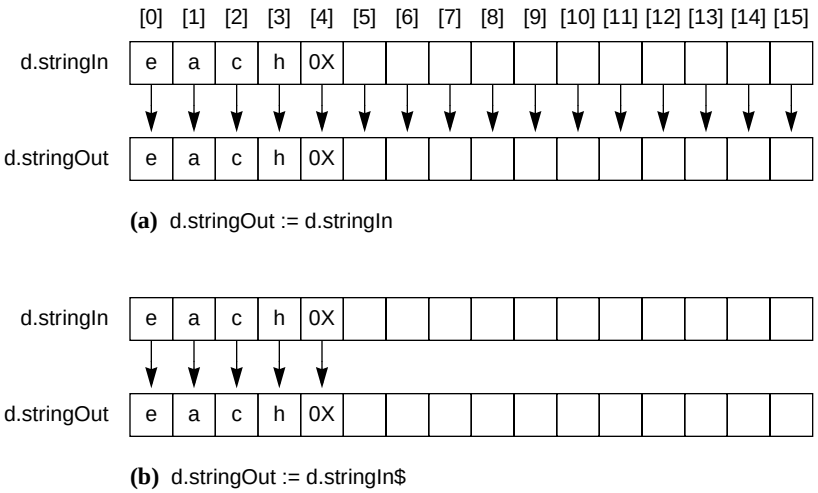


Figure 15.15
The effect of the $ when specifying an array of characters.

LEN returns the length of an array regardless of the number of cells that are occupied by meaningful values. It works with arrays of characters the same way it works with arrays of other types.

Example 15.9 Within procedure RotateLeft in Figure 15.7, the function call

```
LEN(v)
```

would return 1024 because that is the length of the actual parameter, list. It is irrelevant that there may be only seven cells used as in Figure 15.8. The LEN function cannot detect which cells are used and which are not.

Example 15.10 If d.stringIn is declared as in Example 15.8 and has value "each", the function call

```
LEN(d.stringIn)
```

returns 16 because that is the number of cells in the array d.stringIn. Again, it is irrelevant that only five cells are used.

Using LEN and $ together

When you use the LEN function in conjunction with the $ string selector, the function is able to determine which cells are used, and returns the length of the string, that is the number of characters it contains. LEN is only able to achieve this effect with arrays of characters, because arrays of other types are not terminated with 0X. Another interpretation of LEN when used with the $ string selector is the index of the 0X sentinel.

Example 15.11 If d.stringIn is declared as in Example 15.8 and has value "each", the function call

```
LEN(d.stringIn$)
```

returns 4 because that is the number of characters in the array d.stringIn. You can see from Figure 15.15 that 4 is also the index of the 0X sentinel. ∎

Here is a program that illustrates string manipulation by subscripting. Figure 15.16 is the dialog box.

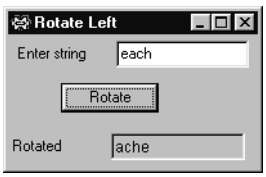


Figure 15.16
The dialog box for a program that rotates an array of characters to the left.

Figure 15.17 shows the program that implements the dialog box. It declares a type String16 to be an array of 16 characters. Although 16 characters are reserved for d.stringIn, the variable can hold a maximum of 15 characters because of the need for the terminal 0X character.

The assignment statement

```
d.stringOut := d.stringIn$
```

in procedure ProcessRotation assigns one array to another using the $ string selector to eliminate any unnecessary assignments beyond the 0X sentinel. Procedure RotateLeft uses the function LEN together with the $ string selector to rotate the characters in the actual parameter d.stringOut.

Procedure RotateLeft in Figure 15.17 is similar to the RotateLeft in Figure 15.7. Instead of passing numItm to indicate the number of items in the array, this procedure uses the LEN function to determine the number of characters in the string. The algorithm uses the LEN function three times, which is actually inefficient. Each time you call LEN with an array of characters using the $ symbol, it must execute a loop starting at the beginning of the array and repeating until the 0X sentinel is reached.

```
MODULE Pbox15D;
   IMPORT Dialog, PboxStrings;
   TYPE
      String16 = ARRAY 16 OF CHAR;
   VAR
      d*: RECORD
         stringIn*: String16;
         stringOut-: String16
      END;

   PROCEDURE RotateLeft (VAR str: ARRAY OF CHAR);
      VAR
         i: INTEGER;
         temp: CHAR;
   BEGIN
      IF LEN(str$) > 1 THEN
         temp := str[0];
         FOR i := 0 TO LEN(str$) - 2 DO
            str[i] := str[i + 1]
         END;
         str[LEN(str$) - 1] := temp
      END
   END RotateLeft;

   PROCEDURE ProcessRotation*;
   BEGIN
      d.stringOut := d.stringIn$;
      RotateLeft(d.stringOut);
      Dialog.Update(d)
   END ProcessRotation;

BEGIN
   d.stringIn := "";
   d.stringOut := ""
END Pbox15D.
```

Figure 15.17
A program with a procedure to rotate the elements in an array.

Figure 15.18 shows another version of procedure RotateLeft, which does not use LEN at all. It tests for the 0X symbol directly with a WHILE loop instead of a FOR loop. The algorithm first checks for str[0] equal to the sentinel. If they are equal, str is the empty string and no rotation is necessary. Otherwise str has at least one character and you can do the rotation. Without the IF test, the WHILE loop would execute even if str were the empty string. Eventually the value of i could reach 15, at which point str[i + 1] would be out of range and would generate a trap.

```
PROCEDURE RotateLeft (VAR str: ARRAY OF CHAR);
   VAR
      i: INTEGER;
      temp: CHAR;
BEGIN
   IF str[0] # 0X THEN
      temp := str[0];
      i := 0;
      WHILE str[i + 1] # 0X DO
         str[i] := str[i + 1];
         INC(i)
      END;
      str[i] := temp
   END
END RotateLeft;
```

Figure 15.18
A more efficient version of RotateLeft.

★ Specifications for arrays

Arrays are frequently processed with FOR statements. Because GCL has no **for** statement, such statements are written with the equivalent **while**. You translate from CP to GCL by making the initialization, test, and increment, which is implicit in the CP FOR statement, explicit.

Example 15.12 The translation of the CP code fragment

```
FOR i := 0 TO numItms - 2 DO
   v[i] := v[i + 1]
END
```

to GCL using *n* for numItms is

```
i := 0 ;
do i ≤ n – 2 →
   v[i] := v[i + 1] ;
   i := i + 1
od
```

Formal specifications for arrays frequently require universal quantification, ∀, to denote a relationship that is true for all the elements of the array. As an example, consider procedure RotateLeft from Figure 15.7. The body of the procedure is

```
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   IF numItm > 1 THEN
      temp := v[0];
      FOR i := 0 TO numItm - 2 DO
         v[i] := v[i + 1]
      END;
      v[numItm - 1] := temp
   END
END RotateLeft;
```

The purpose of the IF statement is to guarantee that numItm is greater than one. To write a specification for the inner code fragment

```
temp := v[0];
FOR i := 0 TO numItm - 2 DO
   v[i] := v[i + 1]
END;
v[numItm - 1] := temp
```

you can use the fact that numItm must be greater than one by making it a precondition. Using *t* as an abbreviation for temp, the above code in GCL is

$t := v[0]$;
$i := 0$;
do $i \le n - 2 \rightarrow$
  $v[i] := v[i + 1]$;
  $i := i + 1$
od;
$v[n - 1] := t$

which we will abbreviate as *S*.

Now, the precondition for the Hoare triple $\{P\}S\{Q\}$ is *P*, and it will contain the conjunct $n > 1$. But how can you specify the postcondition *Q*? The purpose of the procedure is to shift the elements of the array to the left, with the first element moving to the end of the array. You need a rigid variable **V** to specify the initial value of the array *v*. The precondition *P* is

$$1 < n \le len(v) \wedge (\forall i \mid 0 \le i < n : v[i] = \mathbf{V}[i])$$

For the postcondition, you need to state that all the elements starting from the second are shifted down one slot, and the first is shifted to the end. Using universal quantification again, the postcondition *Q* is

$$(\forall i \mid 0 \le i < n - 1 : v[i] = \mathbf{V}[i + 1]) \wedge v[n - 1] = \mathbf{V}[0]$$

The complete formal specification in the form of the Hoare triple $\{P\}S\{Q\}$ is

$\{1 < n \le len(v) \wedge (\forall i \mid 0 \le i < n : v[i] = \mathbf{V}[i])\}$
$v := ?$
$\{(\forall i \mid 0 \le i < n-1 : v[i] = \mathbf{V}[i+1]) \wedge v[n-1] = \mathbf{V}[0]\}$

To prove that statement S satisfies the specification, you prove using formal methods the validity of the Hoare triple

$\{1 < n \le len(v) \wedge (\forall i \mid 0 \le i < n : v[i] = \mathbf{V}[i])\}$
$t := v[0];$
$i := 0;$
do $i \le n-2 \rightarrow$
  $v[i] := v[i+1];$
  $i := i+1$
od;
$v[n-1] := t$
$\{(\forall i \mid 0 \le i < n-1 : v[i] = \mathbf{V}[i+1]) \wedge v[n-1] = \mathbf{V}[0]\}$

Because so many specifications require you to set up a rigid variable for an array, it is convenient to have an abbreviation for the fully quantified expression. From now on, this book will assume that

$v = \mathbf{V}$

is an abbreviation for

$(\forall i \mid 0 \le i < n : v[i] = \mathbf{V}[i])$

where n is the number of elements in the array. With this abbreviation, the above formal specification is written

$\{1 < n \le len(v) \wedge v = \mathbf{V}\}$
$v := ?$
$\{(\forall i \mid 0 \le i < n-1 : v[i] = \mathbf{V}[i+1]) \wedge v[n-1] = \mathbf{V}[0]\}$

Sometimes a rigid variable is not required, particularly if the elements of the array do not change.

Example 15.13 Procedure Maximum in Figure 15.11 does not change the value of v. The ASSERT statement guarantees that numItm is greater than zero, which becomes the precondition. It is an exercise for the student to translate the code fragment

```
largest := v[0];
FOR i := 1 TO numItm - 1 DO
   IF v[i] > largest THEN
      largest := v[i]
   END
END
```

from CP to GCL. Assuming g stands for largest and n stands for numItm, the formal specification states that g is one of the elements of v and that it is the largest element in v as follows.

$\{0 < n \le len(v)\}$
$g := ?$
$\{(\exists i \mid 0 \le i < n : g = v[i]) \wedge (\forall i \mid 0 \le i < n : g \ge v[i])\}$ ∎

The algorithm for putting the largest element at the end of the array assumes that the values in the array after S executes are a rearrangement of the values before S executes. In general, a rearrangement of values is called a permutation.

Example 15.14 If an array of values before S executes is

5 2 7 4

and the values after S executes is

4 2 7 5

then the final value are a permutation of the initial values. ∎

Let f to denote the first index and l the last index in the range $[f..l]$. To specify that the values in array b between $b[f]$ and $b[l]$ are a permutation of the corresponding values in array a, this book will use the predicate $perm(a,b,f,l)$. It takes some care to define this predicate formally. If there are no duplicated values in a or in b, then the definition of the predicate is

$(\forall i \mid f \le i \le l : (\exists j \mid f \le j \le l : a[i] = b[j]))$

The definition is more difficult to write if there are duplicated values in a or in b, and will not be given here. It is left as an exercise to use $perm(a,b,f,l)$ to write the formal specification for the algorithm to put the largest element at the end of the array.

Exercises

1. Predict the output of Figure 15.2 if the two loops are modified as follows:

(a)

```
FOR i := 0 TO 3 DO
    sc.ScanReal(list[i])
END;

FOR i := 0 TO 3 DO
    fm.WriteReal(list[i], 8, 2)
END;
```

(b)

```
FOR i := 3 TO 0 BY -1 DO
    sc.ScanReal(list[i])
END;

FOR i := 0 TO 3 DO
    fm.WriteReal(list[i], 8, 2)
END;
```

(Continued on next page.)

(c)

```
FOR i := 0 TO 3 DO
    sc.ScanReal(list[i])
END;

FOR i := 4 TO 7 DO
    fm.WriteReal(list[i MOD 4], 8, 2)
END;
```

(d)

```
FOR i := 5 TO 8 DO
    sc.ScanReal(list[i MOD 4])
END;

FOR i := 0 TO 3 DO
    fm.WriteReal(list[(i + 1) MOD 4], 8, 2)
END;
```

2. Suppose i and n are integers and v is an ARRAY 10 OF REAL in the following code:

```
i := n - 1;
WHILE i >= 0 DO
    StdLog.Real(v[i]); StdLog.String("  ");
    DEC(i, 2)
END
```

(a) What is the output to the Log if n is 6 and the values of v are

4.0 3.0 5.1 1.0 –7.0 8.5

(b) What is the output to the Log if n is 7 and the values of v are

4.0 3.0 5.1 1.0 –7.0 8.5 2.0

3. If i is an integer and v is an array of integers, what is the output of the following code?

```
FOR i := 0 TO 3 DO
    v[i] := 2 * i
END;
FOR i := 3 TO 1 BY -1 DO
    v[i] := v [i - 1] + 1
END;
FOR i := 0 TO 3 DO
    StdLog.Int(v[i]); StdLog.String("  ")
END
```

4. How many statements does procedure RotateLeft of Figure 15.7 execute if the value of numItm is *n*? Count only the statements in procedure RotateLeft. Do not include the statements in the calling procedure.

5. Your friend writes the following statements in procedure RotateLeft of Figure 15.7.

```
FOR i := 0 TO numItm - 1 DO
    IF i = 0 THEN
        temp := v[0]
    ELSE
        v[i - 1] := v[i]
    END
END;
v[numItm - 1] := temp
```

(a) Does your friend's code work correctly? **(b)** How many statements execute if the value of numItm is *n*? Compare this count with that of the previous exercise.

6. Your friend writes the following statements in procedure RotateLeft of Figure 15.7.

```
temp := v[numItm - 1];
FOR i := 0 TO numItm - 2 DO
    v[i + 1] := v[i]
END;
v[0] := temp
```

and renames the procedure RotateRight. If the program runs with the values

5.0 –2.3 8.0 0.1

for v, what is the output?

7. Determine the statement execution count for both versions of procedure Initialize in Figure 15.13 and Figure 15.14 if the value of numItm is *n*.

8. Suppose the value of numItm in function procedure Maximum in Figure 15.11 is *n*. Assume that *n* is greater than 1. **(a)** How must the data be arranged initially for the best case to occur? **(b)** What is the total statement execution count for this function in the best case? **(c)** How must the data be arranged initially for the worst case to occur? **(d)** What is the total statement execution count for this function in the worst case?

9. Suppose the value of numItm in procedure LargestLast in Figure 15.12 is *n*. Assume that *n* is greater than 1. **(a)** How must the data be arranged initially for the best case to occur? **(b)** What is the total statement execution count for this procedure in the best case? **(c)** How must the data be arranged initially for the worst case to occur? **(d)** What is the total statement execution count for this procedure in the worst case?

10. The expression $v = \mathbf{V}$ where *v* and **V** are arrays of *n* elements is an abbreviation for what quantified expression?

11. **(a)** The predicate $perm(a,b,f,l)$ where *a* and *b* are arrays of elements in the range [*f..l*]is an abbreviation for what quantified expression assuming there are no duplicate values in *a* or in *b*? **(b)** Write a list of four integers for *a* that contains one duplicated value and a list of four integers for *b* that contains no duplicated values, such that the values in *a* and *b* satisfy the quantified expression and such that the values in *b* are not a permutation of the values in *a*.

12. Translate the code fragment in Example 15.13 from CP to GCL.

13. **(a)** Translate the code fragment from procedure LargestLast in Figure 15.12

```
indexMax := 0;
FOR i := 1 TO numItm - 1 DO
    IF v[i] > v[indexMax] THEN
        indexMax := i
    END
END;
temp := v[numItm - 1];
v[numItm - 1] := v[indexMax];
v[indexMax] := temp
```

from CP to GCL. You will not need variable temp, because GCL has the multiple assignment statement. **(b)** Write the formal specification for the code fragment. The postcondition will have two conjuncts, one to state that the final values of *v* are a rearrangement of the initial values, and one to state that the largest element is in the last position. You may use the predicate $perm(a,b,f,l)$.

14. Write the formal specification for the statements S in procedure Initialize in Figure 15.13.

```
PROCEDURE Initialize (OUT v: ARRAY OF INTEGER; numItm: INTEGER)
...
BEGIN
    ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
    S
END Initialize
```

Abbreviating *n* for numItm, S must set the elements of *v*[0..*n*–1] to a decreasing sequence of integers, each value one less that the preceding value, and ending with 0.

15. Write the formal specification for the statements S in procedure RotateRight in Problem 20.

```
PROCEDURE RotateRight (VAR v: ARRAY OF REAL; numItm: INTEGER)
...
BEGIN
    ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
    IF numItm > 1 THEN
        S
    END
END RotateRight
```

16. Write the formal specification for the statements S in procedure Reverse in Problem 22.

```
PROCEDURE Reverse (VAR v: ARRAY OF REAL; numItm: INTEGER)
...
BEGIN
    ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
    IF numItm > 1 THEN
        S
    END
END Reverse
```

17. Write the formal specification for the statements S in procedure FirstOdd in Problem 24. Use the symbol *r* for the integer value returned. You may need to use existential quantification ∃ to state that there does not exist any odd integers before the index returned.

```
PROCEDURE FirstOdd (IN v: ARRAY OF INTEGER; numItm: INTEGER): INTEGER
...
BEGIN
    ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
    S
END FirstOdd
```

18. Write the formal specification for the statements S in IsPalindrome in Problem 29. Use the symbol *b* for the boolean value returned and *n* as an abbreviation for LEN(str$).

```
PROCEDURE IsPalindrome (str: ARRAY OF CHAR): BOOLEAN
...
BEGIN
    S
END IsPalindrome
```

Problems

19. The focus window contains a list of real numbers. Write a procedure activated from a menu selection that inputs the real numbers into an array and then does the following: outputs every other number starting with the first, outputs every other number starting with the second, outputs every negative number, and outputs how many negative numbers were in the list. If the focus window contains

```
5.1  23.2  -6.2  1.0  -19.6  -13.0  4.8
```

Your one procedure should create a new window with all the following output.

```
Every other one from first:
  5.1   -6.2  -19.6    4.8

Every other one from second:
  23.2    1.0  -13.0

Every negative:
  -6.2  -19.6  -13.0

The list has 3 negative numbers.
```

You should write a single exported procedure. It is not necessary to have separate procedures for each output.

20. Declare

```
PROCEDURE RotateRight (VAR v: ARRAY OF REAL; numItm: INTEGER)
```

with the same parameter list as RotateLeft in Figure 15.7. Your procedure should rotate the numbers to the right instead of to the left. Test the procedure in a program similar to Figure 15.7. Implement the appropriate precondition on numItm.

21. Declare

```
PROCEDURE Rotate2Left (VAR v: ARRAY OF REAL; numItm: INTEGER)
```

with the same parameter list as RotateLeft in Figure 15.7 that rotates the items two places to the left instead of only one. For example, if v has the same initial values as in Figure 15.8, its values after the procedure is called should be

9.0 1.0 3.0 5.0 2.0 4.0 6.0

Use only one loop, and do not call RotateLeft. Test the procedure in a program similar to Figure 15.7. Implement the appropriate precondition on numItm, which is the same as the precondition for RotateLeft.

22. Declare

```
PROCEDURE Reverse (VAR v: ARRAY OF REAL; numItm: INTEGER)
```

with the same formal parameter list as procedure RotateLeft in Figure 15.7. The procedure should reverse the elements in the array. Implement the appropriate precondition on numItm. Test the procedure in a program similar to Figure 15.7. Display the values both before and after the call to procedure Reverse. Do not use any output statements in procedure Reverse.

23. Declare

```
PROCEDURE Shuffle (IN vIn: ARRAY OF REAL; OUT vOut: ARRAY OF REAL; numItm: INTEGER)
```

The procedure should shuffle the values like a perfect shuffle of a card deck. Split the deck into two equal stacks and build the shuffled deck by alternately taking cards from the top of each stack. If the list vIn is

1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

then after the shuffle the list vOut should be

1.0 6.0 2.0 7.0 3.0 8.0 4.0 9.0 5.0

Test the procedure in a program similar to Figure 15.7, but with two different lists, one for input and one for output. Implement the appropriate precondition on numItm. Display the values of each list after the call to procedure Shuffle. Do not use any output statements in procedure Shuffle.

24. Declare

```
PROCEDURE FirstOdd (IN v: ARRAY OF INTEGER; numItm: INTEGER): INTEGER
```

The function should return a value that is the index of the first odd integer in the list, or –1 if there are no odd integers. For example, if v contains

8 6 2 7 3 1 4 9 5

the function should return 3, because the first odd integer, 7, is at v[3]. Implement the appropriate precondition on numItm. Test your function by taking the input from the focus window and displaying the result in the Log. Do not use any input or output statements in function FirstOdd.

25. Declare

```
PROCEDURE InitOneZero (OUT v: ARRAY OF INTEGER; numItm: INTEGER)
```

that sets the values of v to alternating ones and zeros. For example, if numItm is 7, the values of v should be

1 0 1 0 1 0 1

Implement the appropriate precondition on numItm. Test your procedure with a dialog box that inputs the number of items. When the user clicks the initialize button, initialize the array, then output it to the Log. Do not include any output statements in InitOneZero.

26. Declare

PROCEDURE InitPairs (OUT v: ARRAY OF INTEGER; numItm: INTEGER)

that sets the values of v to alternating pairs of ones and zeros. For example, if numItm is 7, the values of v should be

1 1 0 0 1 1 0

Implement the appropriate precondition on numItm. Test your procedure with a dialog box that inputs the number of items. When the user clicks the initialize button, initialize the array, then output it to the Log. Do not include any output statements in InitPairs. Hint: Because the pattern repeats every four times, consider the MOD 4 operation.

27. The program in Figure 14.5 generates a random sequence of integers between 0 and 9. With a seed of 2346, the integers 4, 6, and 2 each occur twice and the integers 1, 3, and 8 do not appear as Figure 14.4 shows. Write a procedure

InitRandom (OUT v: ARRAY OF INTEGER; numItm: INTEGER)

that puts random integer values between 0 and numItm - 1 in the first numItm cells of v without any repeating values. Initialize the list with sequential integer values, then make one sweep through the list to exchange the content of each cell with the content of another cell chosen at random. For example, if numItm is 7, initialize v to

0 1 2 3 4 5 6

Then interchange v[0] with another cell chosen at random, v[1] with another cell chosen at random, and so on. Implement the appropriate precondition on numItm. Test your procedure with a dialog box that inputs the number of items and gives the user the option to set the seed. When the user clicks the compute button, initialize the array, then output it to the Log. Do not include any output statements in InitRandom.

28. A list of numbers is said to have a run if several identical values are adjacent to each other. For example, the list of numbers

12 3 3 3 3 3 16 3 4 16 9 4 4

has a run of five 3's and another run of two 4's. Declare

PROCEDURE LongestRun (IN v: ARRAY OF INTEGER; numItm: INTEGER): INTEGER

that returns the length of the longest run in a list of integers. For example, the function should return 5 if v has the above values. Do not use more than one loop. Be sure to consider the case where the longest run is at the end of the list. Assert as part of your precondition that there is at least one item in the list. Test your procedure by taking the input values from the focus window with a menu selection. Verify in the calling procedure that your precondition is met. Output the length of the longest run on the Log. Do not include any output statements in procedure LongestRun.

29. A palindrome is a word that is the same spelled backward or forward. For example, radar is a palindrome but bulb is not, because in reverse order it would be blub. Declare

```
PROCEDURE IsPalindrome (str: ARRAY OF CHAR): BOOLEAN
```

to determine if str is a palindrome. Test it with a dialog box that prompts the user to input a word and outputs whether it is a palindrome. Assume the user will not enter any spaces within or before the word. Consider the empty string to be a palindrome.

30. Declare

```
PROCEDURE Collapse (VAR str: ARRAY OF CHAR)
```

that eliminates all the spaces in str. Test it with a dialog box that prompts the user to input a phrase and outputs the modified phrase without the spaces. For example, if the user enters "a head" your procedure should change it to "ahead". Use only one loop, not nested, and collapse all leading and trailing spaces. Do not include any output statements in procedure Collapse. Test your procedure in a program similar to that in Figure 15.17.

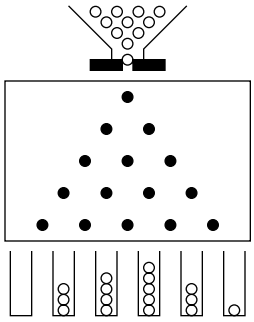
31. Write procedure RotateLeft of Figure 15.18 with a REPEAT loop in place of the WHILE loop. Test your procedure in a program similar to that in Figure 15.17 with a dialog box for the user to enter a string. Be sure to test your program for the case where the user enters the empty string, a string with only one character, and a string with several characters.

32. Declare

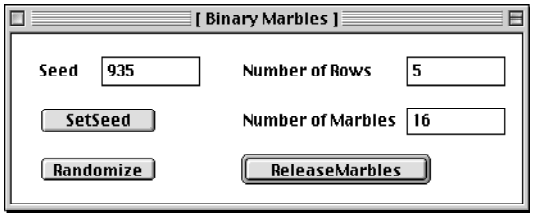
```
PROCEDURE MyToUpper (from: ARRAY OF CHAR; OUT to: ARRAY OF CHAR)
```

that does the identical processing as PboxStrings.ToUpper. Test your procedure in a program similar to that in Figure 15.17 with a dialog box for the user to enter a string. Do not import any module except for Dialog.

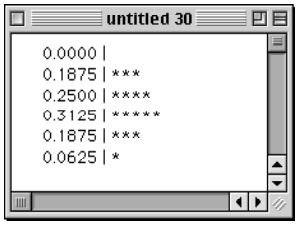
33. Figure 15.19(a) shows a funnel that contains a bunch of marbles at the top of a board with rows of pegs. When the door at the bottom of the funnel opens a marble is released and hits the top peg. The probability is 0.5 that it will bounce off the peg to the right and 0.5 that it will bounce to the left. If it bounces to the right, it will hit a peg on the second row, at which time the probability is again 0.5 for bouncing to the left and 0.5 to the right. The marble continues to hit one peg in each row with probability 0.5 of bouncing to the left and 0.5 to the right. After hitting a peg in the bottom row, the marble is caught in one of the buckets below the last row of pegs. Each bucket is so narrow that the marbles are stacked one on top of the other in the bucket.



(a) The funnel, marbles, pegs and buckets.



(b) The dialog box.



(c) The output.

Figure 15.19
Problem 33.

(a) Write a program that simulates the marbles rolling down the board, hitting the pegs, and landing in the buckets. Implement the dialog box of Figure 15.19(b) to allow the user to input the number of rows of pegs and the number of marbles to release through the funnel. Display the output in a new window like that of Figure 15.19(c) showing the percentage of the number of marbles in each bucket to four places past the decimal point and a histogram with an asterisk symbol for each marble caught in the bucket.

(b) Assuming that the buckets in Figure 15.19(a) are numbered starting with 0 for the leftmost bucket, 1 for the next, and so on up to n for the rightmost bucket, the probability of a single marble landing in bucket number k is

$$P(n, k) = \frac{n!}{k!(n-k)!2^n} \qquad 0 \le k \le n$$

where n is the number of rows of pegs. The percentages that your program computes and displays in the output window should be close to the above theoretical probability. Add to your program a computation of the squared error defined as

$$S(n) = \sum_{i=0}^{n} [P(n, i) - E(n, i)]^2$$

where $E(n, i)$ are the experimental values that you computed as the ratios in the first part of your program. Output the squared error to six places past the decimal point at the bottom of the output window. Experiment with different numbers of marbles for a fixed number of rows. In general, what happens to the squared error as the number of marbles increases?