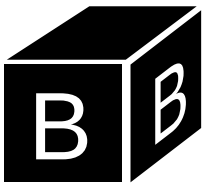


Chapter 7

Abstract Stacks and Lists

The concept of abstraction is pervasive in many of the sciences. Although abstraction has many nuances, one that is of primary importance in computer science is the idea of hidden detail. The modules of Component Pascal provide abstraction for the programmer. The interface of a module shows those elements of the module that are exported and therefore available for other modules to import. All parts of the module that are not exported are hidden details.

Abstraction as hidden detail

Two abstractions that a module can supply are the abstract data structure (ADS) and the abstract data type (ADT). A module that supplies an ADS contains an entity that is useful for the programmer to manipulate. An example of an ADS that your programs have manipulated is the Log. The module StdLog permits you to send strings, integer values, and real values to the Log. The details of how the information gets displayed on the Log are hidden. The programmer only needs to know how to use procedures such as String and Int that are listed in the interface of StdLog. One characteristic of an ADS is that only one entity or data structure is provided by the exporting module. There is only one Log in the BlackBox system.

Abstract data structures

In contrast to a module that supplies an ADS, a module that supplies an ADT exports a type. Modules that use an ADT can declare variables to be of that type. Because a module is free to declare more than one variable to be of the exported type, it can have more than one data structure.

Abstract data types

The BlackBox framework provides the programmer both kinds of entities—ADSs and ADTs. Whether it provides an ADS or an ADT depends on whether it is designed to provide one data structure for the programmer to manipulate, or a type so the programmer can create and manipulate as many data structures needed to solve the problem at hand.

When to use an ADS versus an ADT

This chapter describes two abstractions—stacks and lists. For each of these two abstractions, it illustrates how it could be provided as an ADS and as an ADT. These abstractions are not part of the standard BlackBox distribution, but are contained in the Pbox project for use with this book. This chapter shows the power of abstraction by allowing you to use the data structures without knowing the details of how they are implemented. After you learn more about the Component Pascal language, later chapters will reveal the details that are hidden behind the interfaces for these abstractions.

Stacks

The LIFO property of a stack

A stack is also called a last in, first out (LIFO) list. It is a structure that stores values. Two operations access the values stored on a stack—push and pop. You can visualize a stack as a spring-loaded stack of dishes in a restaurant. When the busboy puts a clean dish on top of the stack, the weight of the dish pushes the stack. If he puts yet another dish on top, it will push down the stack a bit further. If a waitress needs a dish, she takes one from the top of the stack. In other words, the last dish put on the stack is the first one out when someone retrieves a dish.

The push and pop operations

In abstract form, a stack is a list of values with the operations *push* for storage and *pop* for retrieval. Figure 7.1 shows a sequence of operations on a stack in abstract form. In the figure, the push operation places a value on the stack. d.x is a real variable that is not a part of the stack. The pop operation gives the value to d.x from the top of the stack.

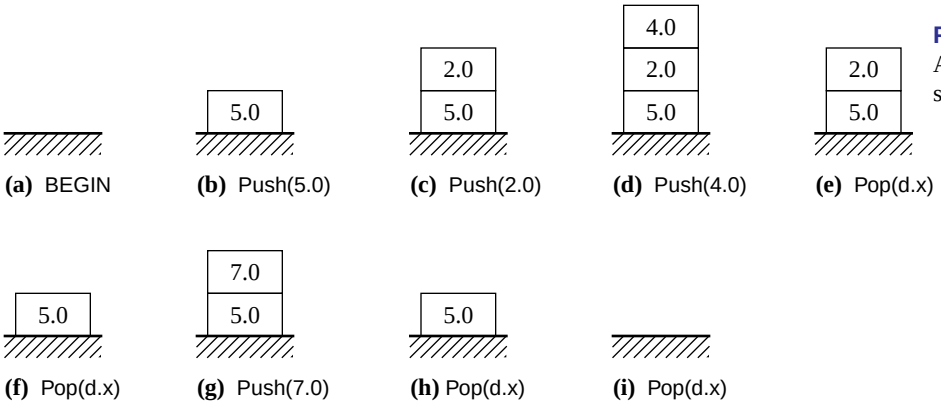


Figure 7.1
A sequence of operations on a stack.

The figure shows the values 5.0, 2.0, and 4.0 pushed onto the stack. The first pop operation in (e) gives the value of 4.0 to d.x. Because 4.0 was the last value in, it is the first value out. Figure 7.1(f) shows the stack partially empty, with only 5.0. Then 7.0 is pushed onto the stack. The next pop operation in (h) gives 7.0 to d.x. Because 7.0 was the last value in, it is the first value out.

Evaluating postfix expressions

Stacks are common in computer systems. One application of stacks is in the processing of arithmetic expressions. When you write a Component Pascal expression such as 3 + 5 in a program, the Component Pascal compiler must translate it to machine language. Then the machine language program executes.

There are three kinds of arithmetic notation:

- Infix 3 + 5
- Prefix + 3 5
- Postfix 3 5 +

Infix notation is the notation you learned as a child. The plus operator is between the operands 3 and 5. In prefix notation, the operator precedes its operands, and in post-

fix notation, the operator follows its operands.

The expressions you write in a Component Pascal program are infix expressions. Unfortunately, infix expressions are difficult to evaluate when the program executes in machine language. It is easier for the computer to evaluate a postfix expression. Component Pascal compilers convert infix expressions to postfix. Then, when the machine-language version of the program executes, it evaluates the postfix expression.

The evaluation of a postfix expression requires a stack of operands. The algorithm for evaluating a postfix expression is

The algorithm for evaluating a postfix expression

- Scan the postfix expression from left to right.
- If you encounter an operand, push it onto the stack.
- If you encounter an operator, apply the operator to the top two operands of the stack. Replace the two operands with the result of the operation.
- After scanning the entire postfix expression, the stack should have one item, the value of the expression.

Example 7.1 Here is a trace of the evaluation for the postfix expression

1 6 + 5 2 − ×

In this trace, the bottom of the stack is on the left.

Stack	*Expression*
empty	1 6 + 5 2 − ×
1	6 + 5 2 − ×
1 6	+ 5 2 − ×
7	5 2 − ×
7 5	2 − ×
7 5 2	− ×
7 3	×
21	empty

The algorithm first pushes 1, then 6. It encounters the plus operator and applies it to 1 and 6, replacing them with 7 on the stack. It pushes 5 and 2, encounters the minus operator and replaces the 5 and 2 with their difference on the stack. Then it encounters the multiply operator. It applies it to 7 and 3, producing the final result of 21. ∎

Translation from infix to postfix

A computer system must solve two basic problems to process an expression from a Component Pascal program. First, it must translate the infix expression to postfix, and second, it must evaluate the postfix expression. The previous algorithm showed the evaluation of a postfix expression. The following discussion shows how to translate from infix to postfix.

Example 7.2 Five examples of infix expressions and their corresponding postfix

expressions are

Infix	*Postfix*
2 + 3	2 3 +
2 × 5 + 3	2 5 × 3 +
2 + 5 × 3	2 5 3 × +
2 × 3 + 5 × 4	2 3 × 5 4 × +
2 + 3 × 5 + 4	2 3 5 × + 4 +

You can verify that they are equivalent by evaluating the postfix expression according to the evaluation algorithm. ∎

Two different postfix expressions can be equivalent to the same infix expression.

Example 7.3 The postfix expressions

5 3 × 2 +

and

2 5 3 × +

are both equivalent to the infix expression

2 + 5 × 3

However, the operands of the first postfix expression (5, 3, 2) are in a different order from the operands of the infix expression (2, 5, 3). ∎

One property of the postfix expressions in Example 7.2 is that their operands are all in the same order as the operands of the equivalent infix expressions. The translation algorithm that follows has the same property.

Maintain the order of the operands.

A characteristic of infix that is not shared by postfix is the precedence of the operators. In the infix expression 2 + 5 × 3, the multiplication is performed before the addition because multiplication has a higher precedence than addition. In postfix, however, there is no operator precedence. The order in which an operation is performed is determined strictly by the position of the operator in the postfix expression. That is one reason computers can evaluate postfix expressions more easily than infix expressions.

Postfix has no operator precedence.

In the translation of the preceding expressions from infix to postfix, only the placement of the operators is different. In fact, the multiplication operators occur before the addition operators, because multiplication has a higher precedence than addition in the infix expression. An algorithm that translates from infix to postfix only needs to shift the operators to the right and possibly reorder them. The order of the operands remains unchanged.

The following algorithm for translating an expression from infix to postfix uses a stack to temporarily store the operators until they can be inserted further to the right into the postfix expression.

The algorithm for translating from infix to postfix

- Scan the infix expression from left to right.
- If the item is an operand, move it directly to the postfix expression.
- If the item is an operator, compare it with the operator on top of the stack:
  - If the operator on top of the stack has a precedence lower than that of the item just encountered in the infix expression or if the stack is empty, push the item just encountered onto the stack.
  - If the operator on top of the stack has a precedence higher than or equal to that of the item just encountered in the infix expression, pop items off the stack. Place them in the postfix expression until either the precedence of the top operator is less than the precedence of the item or the stack is empty. Then push the item onto the stack.
- After the entire infix expression has been scanned, pop any remaining operators left on the stack and put them in the postfix expression.

Because the operands pass directly to the postfix expression, they will maintain their order. The algorithm allows an operator to be pushed onto the stack only if the stack top contains an operator of lower precedence. Therefore, the stack will always have the operators with the highest precedence near the top.

Example 7.4 Here is a trace of the translation process according to the previous algorithm:

Postfix output	*Stack*	*Infix input*
empty	empty	2 + 3 × 5 + 4
2	empty	+ 3 × 5 + 4
2	+	3 × 5 + 4
2 3	+	× 5 + 4
2 3	+ ×	5 + 4
2 3 5	+ ×	+ 4
2 3 5 ×	+	+ 4
2 3 5 × +	empty	+ 4
2 3 5 × +	+	4
2 3 5 × + 4	+	empty
2 3 5 × + 4 +	empty	empty

When the algorithm gets the multiplication operator, it compares it with the addition operator on top of the stack. Multiplication has a higher precedence than addition. Therefore, it puts the multiplication operator on the stack. After it sends the 5 operand to the postfix expression, the multiplication operator follows it. So when you evaluate the postfix expression, you will multiply 3 by 5 before adding the result to 2. ∎

Postfix expressions have no parentheses.

Another reason why postfix expressions are easier to evaluate than infix expressions is that postfix expressions have no parentheses. Infix expressions can have parentheses. When they do, all of the operations inside the parentheses must be performed before the operations outside. Converting an infix expression with parentheses is only slightly more complicated than converting an expression without

parentheses.

Example 7.5 Here are some examples of infix expressions with parentheses and the corresponding postfix expressions.

Infix	*Postfix*
2 × (7 + 3)	2 7 3 + ×
2 × (7 + 3 × 4)	2 7 3 4 × + ×
2 × (7 × 3 + 4)	2 7 3 × 4 + ×
2 + (7 × 3 + 4)	2 7 3 × 4 + +
2 × (7 × (3 + 4) + 5)	2 7 3 4 + × 5 + ×

Again, the order of the operands is the same. You should evaluate these expressions to convince yourself that they are equivalent. ∎

When a left parenthesis is detected in the left to right scan, it marks the starting point of a substack within the main stack. It is as if a new expression is to be evaluated, the expression within the parentheses. The algorithm pushes the left parenthesis onto the stack to mark the beginning of the substack. It converts the subexpression using the substack. When it encounters the matching right parenthesis, it pops the operators off the substack and places them in the postfix expression until it reaches the left parenthesis. It discards the pair of parentheses and continues converting.

Example 7.6 Here is an example of the translation process for an infix expression containing parentheses:

Postfix output	*Stack*	*Expression*
empty	empty	2 × (7 + 3 × 4) + 6
2	empty	× (7 + 3 × 4) + 6
2	×	(7 + 3 × 4) + 6
2	× (	7 + 3 × 4) + 6
2 7	× (	+ 3 × 4) + 6
2 7	× (+	3 × 4) + 6
2 7 3	× (+	× 4) + 6
2 7 3	× (+ ×	4) + 6
2 7 3 4	× (+ ×	) + 6
2 7 3 4 ×	× (+	) + 6
2 7 3 4 × +	× (	) + 6
2 7 3 4 × +	×	+ 6
2 7 3 4 × + ×	empty	+ 6
2 7 3 4 × + ×	+	6
2 7 3 4 × + × 6	+	empty
2 7 3 4 × + × 6 +	empty	empty

When the algorithm encounters the left parenthesis, it simply pushes it onto the stack with the multiplication operator. It continues the conversion, placing the addi-

tion and multiplication operators on the stack. When it encounters the right parenthesis, the algorithm knows it is at the end of the subexpression between the two parentheses. It pops the multiplication and addition operators off the stack. Then the algorithm discards the pair of parentheses and continues the conversion.

The stack abstract data structure

Figure 7.2 is the interface of the stack data structure provided by module PboxStackADS. It contains five exported items—one constant and four procedures. The constant is named capacity and gives the maximum number of items that can be stored on the stack. Constants are similar to variables except that their values cannot be changed with an assignment statement.

You cannot change the value of a constant.

```
DEFINITION PboxStackADS;

   CONST
      capacity = 8;

   PROCEDURE Clear;
   PROCEDURE NumItems (): INTEGER;
   PROCEDURE Pop (OUT val: REAL);
   PROCEDURE Push (val: REAL);

END PboxStackADS.
```

Figure 7.2
The interface of the stack abstract data structure.

As usual, you can view the interface by highlighting PboxStackADS in a display of text on some window and selecting Info→Interface. The interface gives you the names of all the items exported by the module. For the procedures, it gives you the formal parameters including their types, so you will know what type the corresponding actual parameters must have.

Inspecting the interface

The interface does not provide you with a description of what each procedure does, although you can usually surmise that from its name. For example, it is obvious from its name that procedure Pop pops a value off a stack. In case you need more information about the items exported from a module than what is supplied in its interface, you can highlight the module name and select Info→Documentation. If the programmer of the module has supplied corresponding documentation in the Docu folder, a window will open the documentation in browser mode. The documentation fragments in the following discussion are available on-line by highlighting PboxStackADS and selecting Info→Documentation.

Inspecting the documentation

Here is the documentation for capacity.

CONST **capacity**
The maximum number of real values in the stack.

There is a limit to how many items the user can put on this stack. You can see from the interface that a maximum of eight real values are allowed.

Procedure Clear has this documentation.

```
PROCEDURE Clear
Post
The stack is cleared to the empty stack.
```

The word Post in the documentation indicates a postcondition. In general, the documentation of a procedure in BlackBox consists of the procedure's preconditions and postconditions. A precondition states what must be true *before* the procedure executes in order for it to execute correctly. A postcondition states what must be true *after* the procedure executes if all its preconditions were true beforehand. Procedure Clear has no preconditions and one postcondition, which specifies that regardless of the initial state of the stack it will be empty after procedure Clear executes.

Preconditions and postconditions

The specification for procedure NumItems is

```
PROCEDURE NumItems (): INTEGER
Post
Returns the number of elements in the stack.
```

You can tell from its signature that it is a function procedure, not a proper procedure, and it has no parameters in its formal parameter list. When you execute NumItems, it returns the number elements in the stack regardless of the initial state of the stack.

Procedure Pop has a precondition as well as a postcondition.

```
PROCEDURE Pop (OUT val: REAL)
Pre
0 < NumItems()   20
Post
An item is removed from the top of the stack and val gets its value.
```

The precondition requires that the number of elements in the stack be greater than 0 in order to guarantee the post condition after you execute Pop. This is a reasonable precondition, because how can you pop an element off a stack if there are no elements in the stack to begin with? The integer 20 that follows the precondition is an error number that is displayed if the precondition is ever violated. One of the programming style convention for Component Pascal in the BlackBox framework is that the error numbers for precondition violations begin with integer 20. The next section gives an example that shows what happens when you violate a precondition.

A precondition programming style

The specification for procedure Push reveals that it also has a precondition.

```
PROCEDURE Push (val: REAL)
Pre
NumItems() < capacity   20
Post
val is pushed onto the top of the stack.
```

The precondition states that the number of items on the stack must be less than the maximum number allowed. That is, there must be room for at least one more item if you want to push an item onto the top of the stack.

Formal parameters val in Pop and val in Push show that each of these procedures expects a single real actual parameter. The word OUT before a formal parameter in

OUT parameters

the parameter list of a procedure indicates two things:

- The value of the parameter is considered to be undefined when the procedure is called.
- The procedure will change the value of the actual parameter.

Figure 7.1(e) illustrates these two points. Before the pop operation, the value of actual parameter val is considered undefined. After the pop operation, val has the value 4.0 from the top of the stack. Procedure Pop changes the value of val to 4.0.

Figure 7.3 shows a dialog box that illustrates the use of the stack ADS provided by PboxStackADS. To push a value onto the stack, the user enters a real value in the text field and then pushes the button labeled Push. To pop a value off the stack, the user pushes the button labeled Pop, after which the dialog box displays the value popped in the region to the right of the Pop button. The dialog box shows the current number of items on the stack after each push and pop operation. The button labeled Clear Stack removes all items off the stack. The box in Figure 7.3 shows the result of pressing the Pop button after pushing 5.0, 2.0, and 4.0. It corresponds to the state of the stack in Figure 7.1(e).

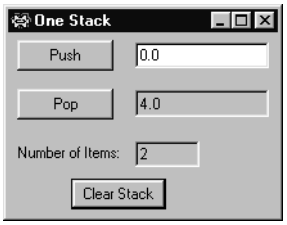


Figure 7.3
The dialog box that goes with the program in Figure 7.4.

Figure 7.4 is the listing of a program that implements the dialog box of Figure 7.3. It imports PboxStackADS and contains the usual record named d whose fields are linked to the dialog box. The initialization part of the module assigns the appropriate procedures to the procedure fields of d. Rather than initialize the default values of the remaining fields of d individually, the initialization part of the module simply calls procedure Clear, which initializes the stack and sets the proper default values.

```
MODULE Pbox07A;
   IMPORT Dialog, PboxStackADS;

   VAR
      d*: RECORD
         valuePushed*, valuePopped-: REAL;
         numItems-: INTEGER;
      END;

   PROCEDURE Push*;
   BEGIN
      PboxStackADS.Push(d.valuePushed);
      d.numItems := PboxStackADS.NumItems();
      Dialog.Update(d)
   END Push;

   PROCEDURE Pop*;
   BEGIN
      PboxStackADS.Pop(d.valuePopped);
      d.numItems := PboxStackADS.NumItems();
      Dialog.Update(d)
   END Pop;
```

Figure 7.4
A program that uses the stack abstract data structure.

```
    PROCEDURE ClearStack*;
    BEGIN
        PboxStackADS.Clear;
        d.valuePushed := 0.0; d.valuePopped := 0.0;
        d.numItems := 0;
        Dialog.Update(d)
    END ClearStack;

BEGIN
    ClearStack
END Pbox07A.
```

Figure 7.4
Continued.

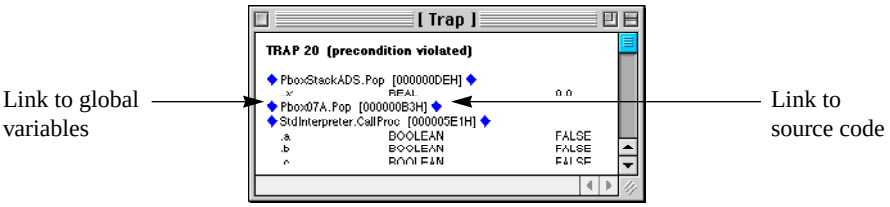
The trap window

You can easily violate a precondition with the dialog box in Figure 7.3 by simply pressing the button labeled Clear Stack followed by pressing the button labeled Pop. Because Clear Stack makes the stack empty, when you attempt to execute procedure Pop its precondition

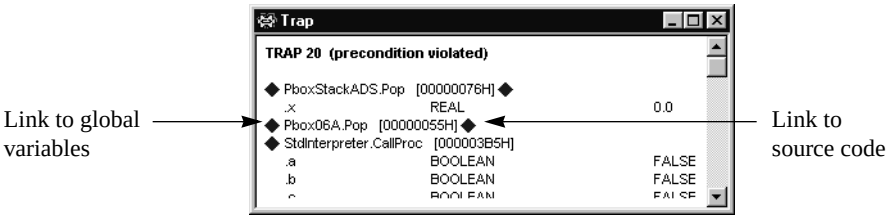
```
0 < NumItems()
```

will be false, because NumItems will return 0. BlackBox responds to this condition with the trap window, whose purpose is to provide information to the programmer about the cause of some error condition. Figure 7.5 shows the top part of the trap window generated from the above scenario.

The purpose of the trap window



(a) MacOS

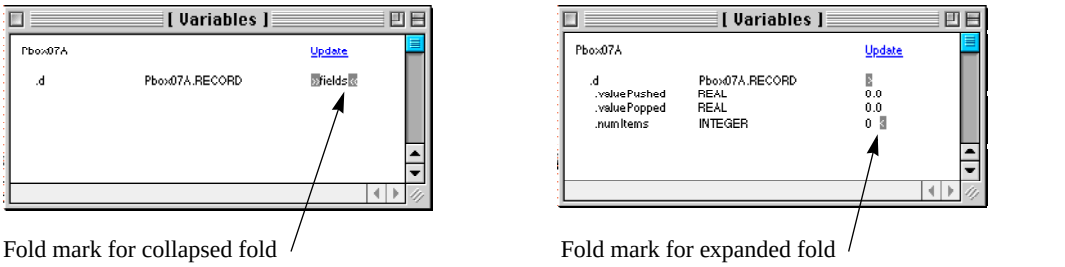


(b) MSWindows

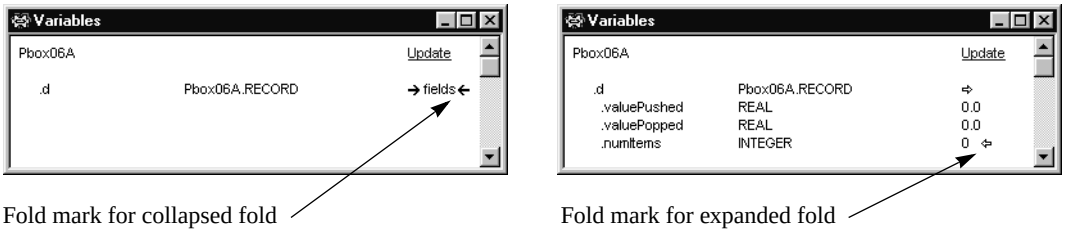
Figure 7.5
A trap window for the stack ADS program.

The first line of the trap window gives a description for the cause of the trap. In this case the cause is that a precondition was violated, specifically the precondition with error number 20. The lines with the diamond marks in front of them are the names of the procedures that are executing when the trap occurs. You can see from the second line in the trap window that procedure PboxStackADS.Pop was executing when its precondition was violated. It was called by Pbox07A.Pop, which was in turn called by StdInterpreter.CallProc, and so on. If you scroll down the trap window you can see that the procedures you write are the last of a long line of procedures that the framework calls before eventually calling yours. Information in the lower procedures is useful only to the programmers of the BlackBox framework and can be ignored.

If you ever get a trap window with a precondition violation, it is easy to read the corresponding documentation. Simply highlight the top procedure name in the trap window (which is possible even though the trap window is in browser mode) and select Info→Documentation. If the programmer has supplied a corresponding Docu file, the framework will open the documentation and you can read the specification. In Figure 7.5, if you highlight PboxStackADS.Pop and select Info→Documentation you will see the specification that describes the precondition for procedure Pop.



(a) MacOS



(b) MSWindows

Figure 7.6
Global variables generated from the trap window of Figure 7.5.

Each procedure that is executing when the trap occurs has a leading mark, which links to the global variables of the module, and a trailing mark, which links to the source code of the executing procedure. When you click on the leading mark a new

window is activated that shows the values of any global variables. Clicking on the leading mark of Pbox07A.Pop shown in Figure 7.5 produces the window shown in Figure 7.6. The global variable in Pbox07A is record d, the interactor for the dialog box.

Link to global variables

Recall that a record is a collection of values, not necessarily of the same type. Record d has three values—valuePushed of type real, valuePopped of type real, and numItems of type integer. When the window that displays d is activated you cannot see the three fields that comprise d because they are contained in a collapsed fold. A fold is a section of text enclosed by fold marks. When the fold is expanded the text is visible. When it is collapsed the text is invisible. You can expand and collapse the text between fold marks by clicking on the marks. Figure 7.6 shows the fold marks for the collapsed fold and the expanded fold which results from clicking on the mark. You can see by inspecting the values in the expanded fold that the type of numItems is integer and its value was 0 when the trap occurred.

Folds

Before you experiment with the link to the source code in Figure 7.5, you should insure that the window for the source code of module Pbox07A is *not* opened. When that is the case and you click on the link to the source code in the trap window, BlackBox will open a window with the source code in it. It will highlight the statement that was executing when the trap occurred. Figure 7.7 shows the result of clicking the link to the source code in Figure 7.5. You can see from the figure that not only is the window opened, but the text PboxStackADS.Pop(d.valuePopped) is highlighted. The highlighting shows you that the call to procedure Pop was executing when the trap occurred.

Link to source code

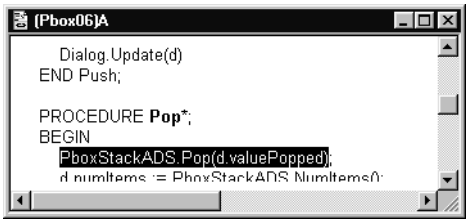


Figure 7.7
Source code window opened by the link to the source code in Figure 7.5.

The trap window not only gives you links to global variables and source code, it also gives you the types and values of formal parameters and local variables. Unlike global variables and source code, formal parameters and local variables are not accessed via links, but are provided directly in the trap window. Figure 7.5 shows a value of 0 for formal parameter x in procedure PboxADS.Pop. Local variables include a and b in StdInterpreter.CallProc, as well as many others not shown in the figure. Much of this information is helpful when you encounter errors in your program. You should develop skill in using the information provided by the trap window to correct your programs.

The stack abstract data type

Previous chapters showed how to program with several types, such as INTEGER and REAL. Because these types are provided by the Component Pascal language, they

are also known as *primitive* types. Component Pascal requires that all primitive types be indicated by spelling them with all uppercase letters. Most programming language designers recognize that programmers will frequently need types other than those provided by the language. Therefore, they provide a facility for programmers to define their own types, known as *programmer-defined* types. The style convention for programmer-defined types is to capitalize the first letter.

Primitive types

Programmer-defined types

The dialog box for the stack abstract data structure provided only one stack for the user to manipulate. The distinguishing feature of an abstract data *type* in contrast to an abstract data *structure* is that a module with an ADT exports a type. Hence, the importing module can declare several variables of the exported type, and therefore have several data structures to manipulate. The listing in Figure 7.8 shows the interface of a module that provides a stack ADT.

```
DEFINITION PboxStackADT;

   CONST
      capacity = 8;

   TYPE
      Stack = RECORD END;

   PROCEDURE Clear (VAR s: Stack);
   PROCEDURE NumItems (IN s: Stack): INTEGER;
   PROCEDURE Pop (VAR s: Stack; OUT val: REAL);
   PROCEDURE Push (VAR s: Stack; val: REAL);

END PboxStackADT.
```

Figure 7.8
The interface of the stack abstract data type.

The interface of the stack ADT shows the same constant for the capacity of a stack and the same four procedures—Clear, NumItems, Pop, and Push. However, an additional item is exported—the type Stack. The interface shows that the type stack is a record, but it appears to have no fields. If you examine MODULE PboxStackADT, you will undoubtedly see some fields in the record for the type. However, none of them are exported. This is yet another example of hidden detail in the concept of abstraction. The interface hides the details of the implementation of the stack. As a programmer who wants to use the type Stack, you only need to be concerned with the behavior of the procedures, and the specifications for using the parameters in the parameter lists.

If you inspect the documentation of module PboxStackADT you will find that the description of Stack is

TYPE **Stack**
The stack abstract data type supplied by PboxStackADT.

Stack is a programmer-defined type supplied by the module for client modules to use. In the same way that you can declare a real variable in Component Pascal with the code fragment

```
VAR
   myReal: REAL;
```

you can declare a stack variable with the code fragment

```
VAR
   myStack: Stack;
```

The parameter lists for the stack ADT procedures are identical to those for the stack ADS except for the addition in each one of a formal parameter of type Stack. The additional parameter is necessary because a module that uses this ADT may have more than one stack variable. If you want to push a value onto a stack, you must give not only the value you want to push but the stack on which you want to push it. Here are the specifications for Clear and Pop, which describe how formal parameter s is used by the client module.

```
PROCEDURE Clear (VAR s: Stack)
Post
Stack s is cleared to the empty stack.

PROCEDURE Pop (VAR s: Stack; OUT val: REAL)
Pre
0 < NumItems(s)   20
Post
An item is removed from the top of stack s and val gets its value.
```

Specifications for the other procedures are similar.

The formal parameters of the stack ADT show four different calling modes that Component Pascal provides:

The four calling modes of Component Pascal

- Default Call by value
- IN Call by constant reference
- OUT Call by result
- VAR Call by reference

Parameter val in Push uses the default call by value. Parameter s in NumItems uses the IN mode known as call by constant reference. Parameter val in Pop uses the OUT mode known as call by result. And parameter s in ClearStack uses the VAR mode known as call by reference.

There are some subtle differences between these four modes that later chapters investigate. For now, you should recognize one important property of these modes:

Changing versus not changing the value of the actual parameter.

- In call by value (default) and call by constant reference (IN), the procedure does *not* change the value of the actual parameter.
- In call by result (OUT) and call by reference (VAR), the procedure *does* change the value of the actual parameter.

You can understand how this property applies to the parameters of the procedures in the stack ADT. When you call procedure Push, you give a value for val that you want to be pushed onto the stack. You do *not* want the procedure to change the value

of val. When you call procedure NumItems, you give a stack for which you want the number of items. You do *not* want the procedure to change the stack. These two modes are call by value and call by constant reference, in which the actual parameter does not change.

When you call procedure Pop, you provide a variable to receive the popped value. You *do* want the procedure to change the value of the variable you provide. When you call procedure ClearStack, you provide the stack variable that you want to be cleared. You *do* want the procedure to change the stack. These two modes are call by result and call by reference, in which the actual parameter does change.

Figure 7.9 shows a dialog box that illustrates the facility of multiple data structures with the stack ADT. It lets the user manipulate two stacks labeled Stack A and Stack B. After the user enters a value in the text box he has the option of pushing the value onto Stack A or onto Stack B. Similarly, the box has two buttons for the pop operation—one for popping from Stack A and one for popping from Stack B—and two displays for the number of items in each of the stacks. The clear button clears both stacks.

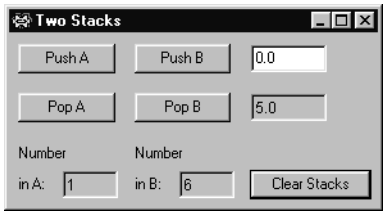


Figure 7.9
The dialog box for manipulating two stacks.

Figure 7.10 shows a program for the dialog box of Figure 7.9. As usual, the box for user input of a real value corresponds to a real exported field valuePushed in the interactor d. The real value popped and the integer number of items in stack A and B are exported read-only, because the program computes their values.

In contrast to the program that used the stack ADS, this program includes two global variables in addition to the interactor d. StackA and StackB are both declared to be of type PboxStackADT.Stack. This is possible only because the interface of Figure 7.8 shows that Stack is a type exported by module PboxStackADT. Note how the stacks themselves are contained in module Pbox07B as global variables, while there is no global stack in module Pbox07A. The procedures in module Pbox07A manipulate one stack that is contained in PboxStackADS.

Now, what happens when the user wants to push the value 3.5 onto stack A? She enters 3.5 into the input text box and presses the button labeled Push A. The programmer has linked this button to procedure PushA in module Pbox07B. The first statement of this procedure is the call

```
PboxStackADT.Push(stackA, d.valuePushed)
```

the actual parameter stackA corresponds to formal parameter s, and actual parameter d.valuePushed corresponds to formal parameter val. So, the value 3.5 is pushed onto stackA. Remember that val is called by value. This operation does *not* change the value of d.valuePushed. Also, s is called by reference. This operation *does* change stackA, because it now has 3.5 on its top whereas before the call it did not.

```
MODULE Pbox07B;
   IMPORT Dialog, PboxStackADT;

   VAR
      d*: RECORD
         valuePushed*, valuePopped-: REAL;
         numItemsA-, numItemsB-: INTEGER;
      END;
      stackA, stackB: PboxStackADT.Stack;

   PROCEDURE PushA*;
   BEGIN
      PboxStackADT.Push(stackA, d.valuePushed);
      d.numItemsA := PboxStackADT.NumItems(stackA);
      Dialog.Update(d)
   END PushA;

   PROCEDURE PushB*;
   BEGIN
      PboxStackADT.Push(stackB, d.valuePushed);
      d.numItemsB := PboxStackADT.NumItems(stackB);
      Dialog.Update(d)
   END PushB;

   PROCEDURE PopA*;
   BEGIN
      PboxStackADT.Pop(stackA, d.valuePopped);
      d.numItemsA := PboxStackADT.NumItems(stackA);
      Dialog.Update(d)
   END PopA;

   PROCEDURE PopB*;
   BEGIN
      PboxStackADT.Pop(stackB, d.valuePopped);
      d.numItemsB := PboxStackADT.NumItems(stackB);
      Dialog.Update(d)
   END PopB;

   PROCEDURE ClearStacks*;
   BEGIN
      PboxStackADT.Clear(stackA);
      PboxStackADT.Clear(stackB);
      d.valuePushed := 0.0; d.valuePopped := 0.0;
      d.numItemsA := 0; d.numItemsB := 0;
      Dialog.Update(d)
   END ClearStacks;

BEGIN
   ClearStacks
END Pbox07B.
```

Figure 7.10
A program that uses the stack abstract data type.

The second statement in procedure PushA is

```
d.numItemsA := PboxStackADT.NumItems(stackA)
```

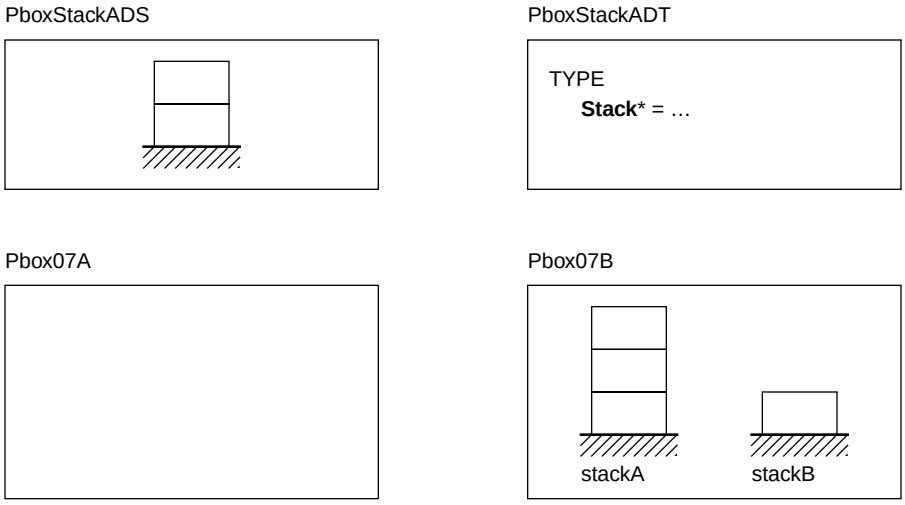
The statement calls function NumItems, which returns the number of items in the stack specified by its parameter, stackA in this case. d.numItemsA gets the value returned. Function NumItems in module PboxStackADS has no parameters, because there is only one stack in an abstract data structure. When the programmer requests the number of items in a stack, it is understood to be the number of items in the one stack supplied by the ADS. With the abstract data type in module PboxStackADT, however, the programmer must specify the stack for which the number of elements is desired.

The last statement in procedure PushA is

```
Dialog.Update(d)
```

which, as usual, makes the change to d.numItemsA visible in the dialog box.

Figure 7.11 illustrates the difference between the stack ADS of Pbox07A and the stack ADT of Pbox07B. With the ADS, there is only one data structure, and it is contained in the exporting module. When the importing procedure manipulates a stack it always manipulates this one data structure. With the ADT, the exporting procedure exports a type, and the data structures are contained in the importing module. When the importing module manipulates a stack, it must specify via a parameter which stack is to be manipulated.



(a) Abstract data structure

(b) Abstract data type

Figure 7.11
The difference between an abstract data structure and an abstract data type.

The list abstract data structure

Like a stack, a list is a data structure that stores values. A list, however, is more flexible than a stack, because when you store an element in a list you are not limited to storing it at one end of the structure. Furthermore, when you remove an element from a list, you can remove it from any location as well. To have the capability of storing and retrieving from any location in a list, you must have a means of specifying an arbitrary location in the list. Each element has associated with it an integer called its *position*. The position of the first element of a list is 0. Figure 7.12 shows the container for a list of at most eight elements. The first element is stored at position 0, and if the list were full the last element would be stored at position 7. The previous sections described stacks as containers of real numbers. In practice, data structures can store values of any type. In this and the following sections, the list data structures will store values of type string.

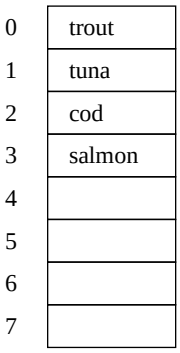


Figure 7.12
A list that contains a maximum of eight strings.

Figure 7.13 is the interface for the list abstract data structure. As with the stack, the list is not part of the standard BlackBox system and must be accessed from the Pbox project. The interface contains a constant capacity of eight, which is the maximum number of elements in the list. Even though this is an ADS as opposed to an ADT, a type is nevertheless exported. However, type T is *not* the type of the list. It is the type of each *element* in the list. In this interface, type T appears as a type of some formal parameters.

```
DEFINITION PboxListADS;

   CONST
      capacity = 8;
   TYPE
      T = ARRAY 16 OF CHAR;

   PROCEDURE Clear;
   PROCEDURE Display;
   PROCEDURE GetElementN (n: INTEGER; OUT val: T);
   PROCEDURE InsertAtN (n: INTEGER; IN val: T);
   PROCEDURE Length (): INTEGER;
   PROCEDURE RemoveN (n: INTEGER);
   PROCEDURE Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN);

END PboxListADS.
```

Figure 7.13
The interface of the list abstract data structure.

The ADS could have been written without T by writing ARRAY 16 OF CHAR everywhere T appears in a parameter list. One advantage of using T is that the parameter lists are shorter because of the abbreviation that T provides. Another, more important, advantage is for the programmer of the ADS. To change the type of the elements stored in the list you would need to change only one line of code. For example, to store a list of reals you would need to change only the line

```
T = ARRAY 16 OF CHAR;
```

to

```
T = REAL;
```

The documentation for T is

```
TYPE T
The type of each element in the list.
```

The specification for procedure Clear is

```
PROCEDURE Clear
Post
The list is initialized to the empty list.
```

It has no precondition. The post condition is that the list is cleared.

Example 7.7 If you execute Clear when the list has the four elements as in Figure 7.12, then all four elements will be removed from the list. ∎

Procedure Length is a function that returns the length of the list. Its specification is

```
PROCEDURE Length (): INTEGER
Post
Returns the number of elements in the list.
```

Example 7.8 If you call Length() when the list has the elements as in Figure 7.12, the function will return 4, because there are four elements in the list. ∎

Procedure InsertAtN inserts an element at position n, shifting the elements below to make room for the new element. It has two preconditions.

```
PROCEDURE InsertAtN (n: INTEGER; IN val: T)
Pre
0 <= n    20
Length() < capacity    21
Post
val is inserted at position n in the list, increasing Length() by 1.
If n > Length(), val is appended to the list.
```

The first precondition states that the value you supply for n cannot be negative. This is a reasonable precondition, because n is the position in the list at which element val is inserted. Because there are no negative positions, you are not allowed to provide a negative position. The second precondition states that the length of the list must be less than its maximum capacity. This is a reasonable precondition, because if the list is already filled to capacity there will be no room for an additional element.

Example 7.9 If the list has the elements as in Figure 7.12, and you execute

```
InsertAtN(2, myString)
```

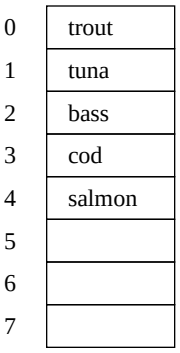
where myString has the value “bass”, then the list will be changed as in Figure 7.14(a). If instead you execute

```
InsertAtN(0, myString)
```

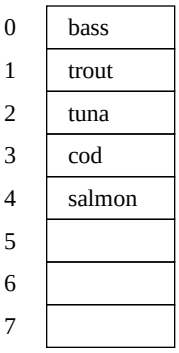
then the list will be changed as in Figure 7.14(b). And if instead you execute

```
InsertAtN(7, myString)
```

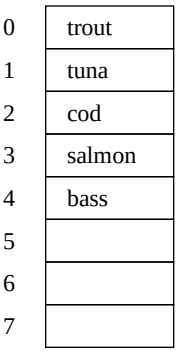
a precondition is not violated. Element “bass” is inserted at position 4 at the end of the list as shown in Figure 7.14(c). ∎



(a) After inserting at position 2.



(b) After inserting at position 0.



(c) After inserting at position 7.

Figure 7.14
The list of Figure 7.12 after executing the statements of Example 7.9.

Procedure RemoveN removes the element from position n, shifting the elements below to take up the room vacated by the removed element. Here is its specification.

```
PROCEDURE RemoveN (n: INTEGER)
Pre
0 <= n   20
Post
If n < Length(), the element at position n in the list is removed.
Otherwise, the list is unchanged.
```

The precondition does not allow you to supply a negative value for position n.

Example 7.10 If the list has the elements as in Figure 7.12, and you execute

```
RemoveN(2)
```

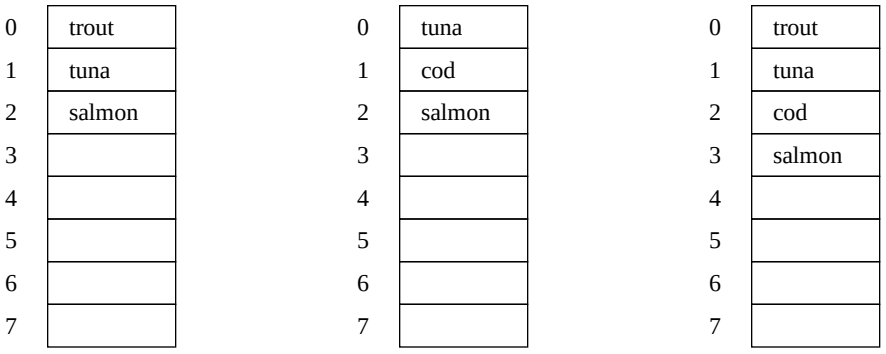
then the list will be changed as in Figure 7.15(a). If instead you execute

```
RemoveN(0)
```

then the list will be changed as in Figure 7.15(b). And if instead you execute

```
RemoveN(7)
```

a precondition is not violated. The list simply remains unchanged as shown in Figure 7.15(c).



(a) After removing from position 2.

(b) After removing from position 0.

(c) After removing from position 7.

Figure 7.15
The list of Figure 7.12 after executing the statements of Example 7.10.

Procedures InsertAtN and RemoveN alter the content of the list. The list ADS provides two procedures that allow you to query the list without altering its content. With procedure GetElementN, you supply a position n, and it gives you the element at that position. Procedure Search does the reverse. You give it an element, and it returns the position of that element in the list. It includes a boolean parameter fnd, which is set to FALSE if the element is not found in the list. In that case, the value returned for n is not defined. Here are the specifications of GetElementN and Search.

```
PROCEDURE GetElementN (n: INTEGER; OUT val: T)
Pre
0 <= n    20
n < Length()    21
Post
val gets the data value of the element at position n of the list.
Note: 0 is the position of the first element in the list.

PROCEDURE Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN)
Post
If srchVal is in the list, fnd is set to TRUE and n is set to the first position where srchVal is found.
Otherwise, fnd is set to FALSE and n is undefined.
```

Example 7.11 If the list has the elements as in Figure 7.12, and you execute

```
GetElementN(2, myString)
```

then myString will get the value “cod”. Execution of

```
GetElementN(4, myString)
```

violates a precondition and generates a trap. ∎

Example 7.12 If the list has the elements as in Figure 7.12, myString has value “cod”, position is an integer variable, found is a boolean variable, and you execute

```
Search(myString, position, found)
```

then found will get TRUE and position will get the value 2. If you execute the same statement when myString has value “catfish”, found will get FALSE and position will get some undefined value. Note that the position of the last element, “salmon”, is 3, not 4, because the position of the first element, “trout”, is 0, not 1. ∎

The specification of procedure Display is

```
PROCEDURE Display
Post
The list is output to the Log, one element per line with each element preceded by its position.
```

Display has no preconditions.

Example 7.13 If the list has the elements as in Figure 7.12, and you execute procedure Display, the following text will be printed on the Log.

```
0 trout
1 tuna
2 cod
3 salmon
```

∎

Figure 7.16 shows a dialog box that allows the user to manipulate a list of strings. The state of the dialog box corresponds to a list that contains four elements as in Figure 7.12. The user has previously entered the elements by pressing the Insert button. She queried the list by asking for the location of element “salmon”, which the dialog box gives as position 3, and for the retrieval of the element at position 1, which the dialog box gives as “tuna”. With each insert and deletion of an element the current number of items is displayed.

The listing in Figure 7.17 is a program that implements the dialog box in Figure 7.16. It is a straightforward mapping from input and output values of the dialog box to variables in the d interactor, and from the buttons of the dialog box to exported procedures in the module.

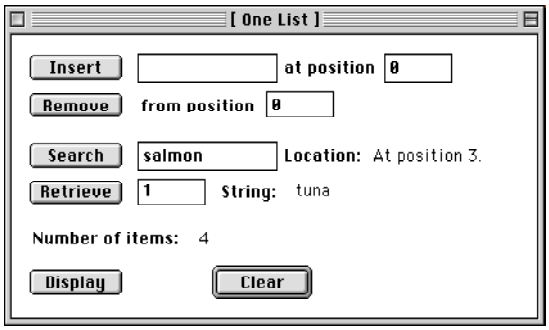


Figure 7.16
The dialog box for manipulating a list.

```
MODULE Pbox07C;
   IMPORT Dialog, PboxListADS, PboxStrings;

   TYPE
      String32 = ARRAY 32 OF CHAR;

   VAR
      d*: RECORD
         insertT*: PboxListADS.T; insertPosition*: INTEGER;
         removePosition*: INTEGER;
         searchT*: PboxListADS.T; searchPosition-: String32;
         retrievePosition*: INTEGER; retrieveT-: PboxListADS.T;
         numItems-: INTEGER;
      END;

   PROCEDURE InsertAt*;
   BEGIN
      PboxListADS.InsertAtN(d.insertPosition, d.insertT);
      d.numItems := PboxListADS.Length();
      Dialog.Update(d)
   END InsertAt;

   PROCEDURE RemoveFrom*;
   BEGIN
      PboxListADS.RemoveN(d.removePosition);
      d.numItems := PboxListADS.Length();
      Dialog.Update(d)
   END RemoveFrom;
```

Figure 7.17
A program that uses the list abstract data structure to implement the dialog box of Figure 7.16.

Figure 7.17
Continued.

```
    PROCEDURE SearchFor*;
    VAR
        found: BOOLEAN;
        position: INTEGER;
    BEGIN
        PboxListADS.Search(d.searchT, position, found);
        IF found THEN
            PboxStrings.IntToString(position, 1, d.searchPosition);
            d.searchPosition := "At position " + d.searchPosition + "."
        ELSE
            d.searchPosition := "Not in list."
        END;
        Dialog.Update(d)
    END SearchFor;

    PROCEDURE RetrieveFrom*;
    BEGIN
        PboxListADS.GetElementN(d.retrievePosition, d.retrieveT);
        Dialog.Update(d)
    END RetrieveFrom;

    PROCEDURE DisplayList*;
    BEGIN
        PboxListADS.Display;
        Dialog.Update(d)
    END DisplayList;

    PROCEDURE ClearList*;
    BEGIN
        PboxListADS.Clear;
        d.insertT := ""; d.insertPosition := 0;
        d.removePosition := 0;
        d.searchT := ""; d.searchPosition := "";
        d.retrievePosition := 0; d.retrieveT := "";
        d.numItems := PboxListADS.Length();
        Dialog.Update(d)
    END ClearList;

BEGIN
    ClearList
END Pbox07C.
```

As an example of the mapping, consider the first line of the dialog box that allows the user to insert an element into the list. The user must enter a string into the first box and an integer representing the location of where to insert the string into the second box. These two values require two fields in the d interactor, both exported read-write because the user enters the values. The corresponding fields in the d interactor are

```
insertT*: PboxListADS.T; insertPosition*: INTEGER
```

The type of insertT is PboxListADS.T, which we know from the interface in Figure 7.13 is the same as ARRAY 16 OF CHAR. This type permits the user to enter a string of up to 15 characters. The button labeled Insert is obviously linked to the procedure named InsertAt.

How does this procedure work? Suppose that the list has the elements as in Figure 7.12 and the user has entered bass in the first box and 2 in the second box. These actions give the value "bass" to d.insertT and 2 to d.insertPosition. When the user clicks the button labeled Insert, procedure InsertAt executes because that is the procedure to which the button is linked.

The first statement of procedure InsertAt is

```
PboxListADS.InsertAtN(d.insertPosition, d.insertT)
```

Actual parameter d.insertPosition corresponds to formal parameter n in the interface of Figure 7.13, and actual parameter d.insertT corresponds to formal parameter val. The procedure inserts "bass" before the element at position 2, as shown in Figure 7.14(a).

Because of this change in the list, the number of elements needs to be updated. The second statement of procedure InsertAt is

```
d.numItems := PboxListADS.Length()
```

which changes the value of d.numItems to 5 to reflect the new length of the list. This new value does not become visible to the user until the third statement of procedure InsertAt

```
Dialog.Update(d)
```

executes.

As another example, consider execution of procedure SearchFor. Suppose the user enters "salmon" in the input field adjacent to the button labeled Search as in Figure 7.16. When she presses the button, procedure SearchFor executes. Its first statement

```
PboxListADS.Search(d.searchT, position, found)
```

is a call to procedure Search in module PboxListADS with actual parameters d.searchT, position, and found. Before the procedure call, d.searchT has the value "salmon", and position and found are local variables that have undefined values. During execution, procedure Search gives found the value TRUE and position the value 3, because it found the string "salmon" at position 3 of the list. The next statement in procedure SearchFor is the IF statement

```
IF found THEN
   PboxStrings.IntToString(position, 1, d.searchPosition);
   d.searchPosition := "At position " + d.searchPosition + "."
ELSE
   d.searchPosition := "Not in list."
END
```

Its purpose is to set the value of d.searchPosition, whose type is String32.

The programmer defined the type String32 earlier in the listing

Defining character array types

```
TYPE
   String32 = ARRAY 32 OF CHAR;
```

to be an array of 32 characters. String32 is a Component Pascal identifier chosen by the programmer. The module would have worked just as well had the programmer not defined the String32 type and declared d.searchPosition directly as

```
searchT*: PboxListADS.T; searchPosition-: ARRAY 32 OF CHAR
```

Although there is no advantage in this program to declare the String32 type, in certain situations when arrays of characters are assigned to each other or compared to each other it is necessary to do so. This book generally follows the practice of defining such types from now on.

Because found has value TRUE, the THEN part of the IF statement executes and the ELSE part is skipped. The two statements in the THEN part

```
PboxStrings.IntToString(position, 1, d.searchPosition);
d.searchPosition := "At position " + d.searchPosition + "."
```

convert the integer value 3 to the string value “3”, storing the string value in d.searchPosition, then convert the string value “3” to the string value “At position 3.”, which gets stored in d.searchPosition. Finally, the last statement of SearchFor

```
Dialog.Update(d)
```

makes the value of d.searchPosition visible in the dialog box.

The workings of the other buttons are similar. You can view the dialog box as the specification of the program. The way to specify what a program should do is to sketch the dialog box first. You then set up the input and output values in the box to correspond to fields of the interactor d, and the buttons of the dialog box to correspond to exported procedures. In the program of Figure 7.17, there is a close correlation between the task that must be accomplished by a button and the services provided by the server module PboxListADS. The program consists of simply linking the controls of the dialog box correctly and calling the appropriate exported procedure.

The list abstract data type

The listing of Figure 7.18 is the interface for the list abstract data type. You have surely surmised by now that the difference between a list ADS and a list ADT is that the list ADT exports the list type, which permits the importing module to declare more than one list data structure.

```
DEFINITION PboxListADT;

   CONST
      capacity = 8;

   TYPE
      List = RECORD END;
      T = ARRAY 16 OF CHAR;

   PROCEDURE Clear (VAR lst: List);
   PROCEDURE Display (IN lst: List);
   PROCEDURE GetElementN (IN lst: List; n: INTEGER; OUT val: T);
   PROCEDURE InsertAtN (VAR lst: List; n: INTEGER; IN val: T);
   PROCEDURE Length (IN lst: List): INTEGER;
   PROCEDURE RemoveN (VAR lst: List; n: INTEGER);
   PROCEDURE Search (VAR lst: List; IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN);

END PboxListADT.
```

Figure 7.18
The interface of the list abstract data type.

Comparing Figure 7.18 with Figure 7.13, the list ADT exports the additional type

```
List = RECORD END
```

that is not exported by the list ADS. This is the type of the list itself. You can tell from the interface that a list is a record, that is, a collection of values which do not necessarily have identical types. However, the fields of the exported record are not shown in the interface. They are hidden in the black box of the implementation—yet another example of the pervasive concept of abstraction. At this stage of our understanding, we do not need to know how the list is implemented. We only need to know how to use the procedures.

Apart from the type of the list that is exported, every other item in the list ADT of Figure 7.18 is identical in name and purpose to the corresponding item in the list ADS. The only difference in the parameter lists of the procedures in the ADT is the addition of a formal parameter to specify on which list the operation is to be performed. For example, the specification of procedure InsertAtN for the list ADS is

```
PROCEDURE InsertAtN (n: INTEGER; IN val: T)
Pre
0 <= n   20
Length() < capacity   21
Post
val is inserted at position n in the list, increasing Length() by 1.
If n > Length(), val is appended to the list.
```

while the specification of procedure InsertAtN for the list ADT is

```
PROCEDURE InsertAtN (VAR lst: List; n: INTEGER; IN val: T);
Pre
0 <= n    20
Length(lst) < capacity    21
Post
val is inserted at position n in list lst, increasing Length(lst) by 1.
If n > Length(lst), val is appended to lst.
```

You can see that the precoditions and postconditions are identical, except that the postcondition for the list ADS says that val is inserted in *the* list, whereas the postcondition for the list ADT says that val is inserted in list lst. The pre- and postconditions for the other procedures of the list ADT are similar in the same way to those of the list ADS.

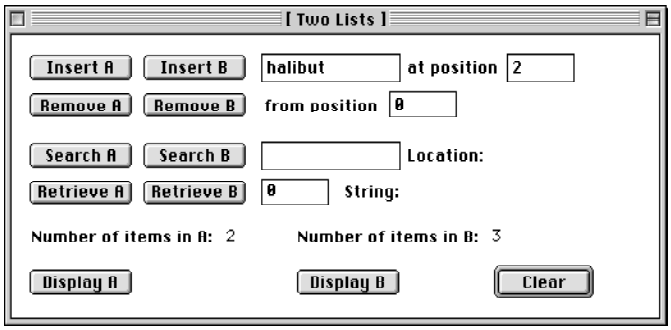


Figure 7.19
The dialog box for manipulating two lists.

Figure 7.19 shows a dialog box that allows the user to manipulate two lists. The listing in Figure 7.20 is an implementation of the dialog box based on the list ADT.

```
MODULE Pbox07D;
   IMPORT Dialog, PboxListADT, PboxStrings;

   TYPE
      String32 = ARRAY 32 OF CHAR;

   VAR
      d*: RECORD
         insertT*: PboxListADT.T; insertPosition*: INTEGER;
         removePosition*: INTEGER;
         searchT*: PboxListADT.T; searchPosition-: String32;
         retrievePosition*: INTEGER; retrieveT-: PboxListADT.T;
         numItemsA-, numItemsB-: INTEGER;
      END;
      listA, listB: PboxListADT.List;
```

Figure 7.20
A program that uses the list abstract data type

Figure 7.20
Continued.

```
PROCEDURE InsertAtA*;
BEGIN
   PboxListADT.InsertAtN(listA, d.insertPosition, d.insertT);
   d.numItemsA := PboxListADT.Length(listA);
   Dialog.Update(d)
END InsertAtA;

PROCEDURE InsertAtB*;
BEGIN
   PboxListADT.InsertAtN(listB, d.insertPosition, d.insertT);
   d.numItemsB := PboxListADT.Length(listB);
   Dialog.Update(d)
END InsertAtB;

PROCEDURE RemoveFromA*;
BEGIN
   PboxListADT.RemoveN(listA, d.removePosition);
   d.numItemsA := PboxListADT.Length(listA);
   Dialog.Update(d)
END RemoveFromA;

PROCEDURE RemoveFromB*;
BEGIN
   PboxListADT.RemoveN(listB, d.removePosition);
   d.numItemsB := PboxListADT.Length(listB);
   Dialog.Update(d)
END RemoveFromB;

PROCEDURE SearchForA*;
VAR
   found: BOOLEAN;
   position: INTEGER;
BEGIN
   PboxListADT.Search(listA, d.searchT, position, found);
   IF found THEN
      PboxStrings.IntToString(position, 1, d.searchPosition);
      d.searchPosition := "At position " + d.searchPosition + "."
   ELSE
      d.searchPosition := "Not in list."
   END;
   Dialog.Update(d)
END SearchForA;
```

Figure 7.20
Continued.

```
    PROCEDURE SearchForB*;
    VAR
        found: BOOLEAN;
        position: INTEGER;
    BEGIN
        PboxListADT.Search(listB, d.searchT, position, found);
        IF found THEN
            PboxStrings.IntToString(position, 1, d.searchPosition);
            d.searchPosition := "At position " + d.searchPosition + "."
        ELSE
            d.searchPosition := "Not in list."
        END;
        Dialog.Update(d)
    END SearchForB;

    PROCEDURE RetrieveFromA*;
    BEGIN
        PboxListADT.GetElementN(listA, d.retrievePosition, d.retrieveT);
        Dialog.Update(d)
    END RetrieveFromA;

    PROCEDURE RetrieveFromB*;
    BEGIN
        PboxListADT.GetElementN(listB, d.retrievePosition, d.retrieveT);
        Dialog.Update(d)
    END RetrieveFromB;

    PROCEDURE DisplayListA*;
    BEGIN
        PboxListADT.Display(listA);
    END DisplayListA;

    PROCEDURE DisplayListB*;
    BEGIN
        PboxListADT.Display(listB);
    END DisplayListB;

    PROCEDURE ClearLists*;
    BEGIN
        PboxListADT.Clear(listA); PboxListADT.Clear(listB);
        d.insertT := ""; d.insertPosition := 0;
        d.removePosition := 0;
        d.searchT := ""; d.searchPosition := "";
        d.retrievePosition := 0; d.retrieveT := "";
        d.numItemsA := 0; d.numItemsB := 0;
        Dialog.Update(d)
    END ClearLists;

BEGIN
    ClearLists
END Pbox07D.
```

The structure of the module is characteristic of those that import a type. There are two global variables, listA and listB, that correspond to the two lists that are manipulated by the user. They are declared as

```
listA, listB: PboxListADT.List;
```

That is, each list has the type that is exported by module PboxListADT. When the user clicks the button labeled Insert A, procedure InsertAtA executes. Its first statement is

```
PboxListADT.InsertAtN(listA, d.insertPosition, d.insertT)
```

The first actual parameter listA corresponds to the first formal parameter lst. This is how the calling procedure tells the called procedure on which list the operation is to be performed.

Queues

The FIFO property of a queue

A queue is also called a first in, first out (FIFO) list. It is similar to a stack in that it stores values in a list in the order it receives them, but different in that data is retrieved from the opposite end of the list. The two operation on a queue are *enqueue* and *dequeue*, which correspond to push and pop for a stack.

You can visualize a queue as a line of people at a ticket window. When another person comes to buy a ticket, she goes to the back of the line, corresponding to the enqueue operation. When the clerk at the window can serve another person he helps the person from the front of the line, corresponding to the dequeue operation. The first person to enter the queue is the first one to be served.

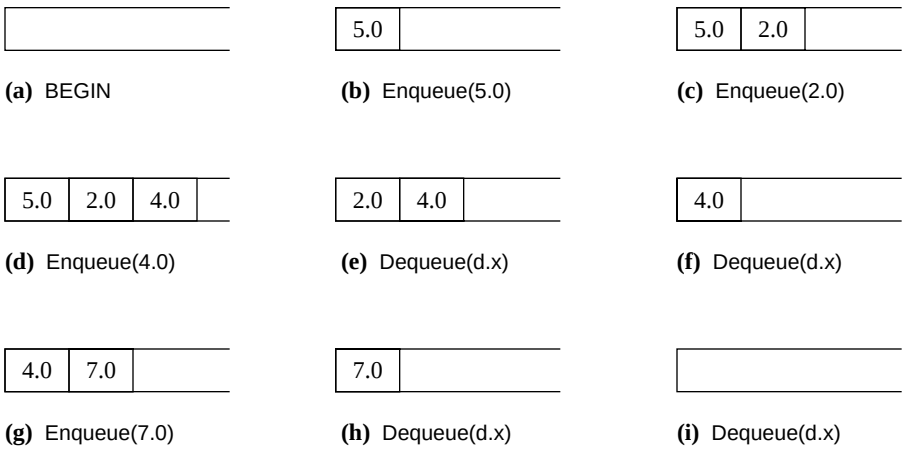


Figure 7.21
A sequence of operations on a queue.

Figure 7.21 shows a sequence of operations on a queue in abstract form. It corresponds to the sequence in Figure 7.1 for a stack, but Push is replaced by Enqueue and Pop is replaced by Dequeue. In the figure, values 5.0, 2.0, and 4.0 are added to the queue. The first delete operation gives the value of 5.0 to val. Contrast this with the first pop operation in Figure 7.1, which gives 4.0 to val. In Figure 7.21, because 5.0 was the first value in, it is the first value out.

The Pbox project does not provide a queue for you to manipulate. However, you can simulate the behavior of a queue with the list ADS or ADT provided by Pbox. The implementations of the queue ADS and ADT are problems at the end of this chapter.

Design by contract

Most of the programming problems in the early part of this book require you to write client modules. Later problems near the end require you to write server modules as well. Sometimes in a commercial software development environment, the same programmer writes the client module and the server module. However, for large projects, it is more common for one person to program the server modules that are imported by a different client programmer. In that situation, the interface becomes a *contract* between the two programmers. The interface specifies the name of the procedures and the number and types of their parameters. The documentation, which always includes the interface, also states any preconditions that must be satisfied if the procedure is to execute correctly. As with all written contracts, the specification of the precondition is in the form of a guarantee. The programmer of the server module is making a guarantee that *if* the programmer of the client ensures that the precondition is not violated *then* the procedure will execute correctly.

Bulletproof programs

A common use of the IF statement is to prevent a trap from ever occurring in a user's program. Such programs are sometimes called *bulletproof*. You have probably experienced commercial programs that are not bulletproof. The fortunate user gets a dialog box with a message like, "Your application program has terminated because an error of type -781 has occurred," or something else equally meaningless. The unfortunate user must restart the computer. Programmers of client modules make their programs bulletproof by reading the contract suppled by the interface and using the IF statement to ensure that their programs never violate the contract. Problem 27 requires you to make the program of Figure 7.4 bulletproof using IF statements in the client module. Such a program will prevent the user from ever experiencing the trap window of Figure 7.5.

Chapter 17 shows how the ASSERT procedure implements the precondition in the server module by bringing up the trap window when the precondition is violated. Later in this book, you will learn how to program server modules that are imported by client modules. When you work those problems, you will also be writing the client modules that import the server modules. You may be tempted to simplify your programming by putting IF statements that make the program bulletproof in your server module. That is a bad habit. You should always keep in mind that in a commercial software development environment, the programmer of the server is usually not the same person as the programmer of the client.

It is the job of the programmer of the server to establish the contract in the interface with preconditions for the procedures. She implements the preconditions with the ASSERT procedure. It is the job of the programmer of the client to make the application program bulletproof. He makes it bulletproof with the IF statement. This software practice is summarized by the design-by-contract rule, which states

The design-by-contract rule

- IF in the client.
- ASSERT in the server.

It may seem unnecessarily complicated for you to do in two places what you could more easily do in one place. But, because the programmer of the server is usually not the same person as the programmer of the client, you should get into the habit of writing your modules according to the design-by-contract rule.

★ Formal specifications

The preconditions and postconditions for procedures in BlackBox documentation correspond to the preconditions and postconditions for the *Hoare triple* of formal methods. Formally, the three parts of a Hoare triple are:

The Hoare triple

- the precondition, P
- the statement, S
- the postcondition, Q

which is usually written $\{P\}S\{Q\}$ in formal methods notation. The precondition and postcondition are conditions that can be either true or false. The interpretation of Hoare triple $\{P\}S\{Q\}$ is, "If P is true and you execute S, then Q is guaranteed to be true." The statement S can represent a single programming statement or, more generally, a sequence of statements. In this chapter, S represents the sequence of programming statements in the implementation of the procedure. Because the interface hides the implementation, you cannot see S from any of the program listings in this chapter.

The interpretation of a Hoare triple

Sometimes the sequence of statements that S represents is supposed to change some value, or set of values, or data structure. To indicate what is to be changed, S is written as if it were a single assignment statement, even though it still represents a set of statements in general. The left side of the assignment is a variable or set of variables to be changed and the right side of the assignment is a question mark. The question mark signifies that the formal specification does not state how the values of the variables are to be changed. It only states which variables are to be changed.

Example 7.14 The documentation of RemoveN for the list ADT

```
PROCEDURE RemoveN (VAR lst: List; n: INTEGER)
Pre
0 <= n   20
Post
If n < Length(lst), the element at position n in list lst is removed.
Otherwise, the list is unchanged.
```

is written formally as

$\{0 \le n \wedge Length(lst) = \mathbf{L}\}$
$lst := ?$
$\{(n < \mathbf{L} \Rightarrow \text{Element at } n \text{ is removed} \wedge Length(lst) = \mathbf{L} - 1) \wedge (n \ge \mathbf{L} \Rightarrow lst \text{ is unchanged})\}$

Rigid variables

The letter **L**, called a rigid variable, is used to save the initial value of the function $Length(lst)$ in the Hoare triple. If n is less than the initial length of the list, whatever the length is before execution of the procedure, and the preconditions are satisfied, then after execution the length of the list will be its initial value minus one. S is the expression $lst := ?$, which indicates that *lst* is the data structure to be changed. ∎

The weakest possible precondition is true.

If the documentation has no precondition, then the formal precondition is the weakest possible precondition, namely *true*.

Example 7.15 The documentation of Length for the list ADT

PROCEDURE **Length** (IN lst: List): INTEGER
Post
Returns the number of elements in list lst.

is written formally as

$\{true\}S\{Length(lst) = \text{The length of } lst\}$ ∎

Formal specifications

In the previous examples of the Hoare triple, the symbol S stands for the statements in the implementation of the procedure. The abstraction of the interface hides the details of the statements from the client program. A Hoare triple $\{P\}S\{Q\}$ consisting of a given precondition P and postcondition Q, but an unknown S, is called a *formal specification*. The software designer who writes the procedure treats the formal specification as a contract. She assumes that the client program will insure that the precondition is true, then writes the statements that will make the postcondition true after the procedure executes.

Another use of the Hoare triple is to define the assignment statement. When used this way, S is not hidden as it is in a formal specification, but is the assignment statement. Formally, the assignment statement is defined in terms of the Hoare triple and textual substitution, which also uses the := symbol. If E is an expression and R is a postcondition, then the assignment statement $x := E$ is defined as the Hoare triple

Formal definition of assignment

$\{R[x := E]\}x := E\{R\}$

where the := symbol in the precondition signifies textual substitution. Textual substitution of E for x in the postcondition produces the weakest precondition for the Hoare triple.

Example 7.16 The definition of $x := x + 2$ for the postcondition $x > 0$ is the Hoare triple

$\{x + 2 > 0\}x := x + 2\{x > 0\}$

You can see that this Hoare triple is valid because if $x + 2 > 0$ is true before the assignment statement executes, that is, if $x > -2$, then the postcondition $x > 0$ is guaranteed to be true afterwards. Of course, many other Hoare triples have this same property. For example, the Hoare triple

$\{x - 1 > 0\}x := x + 2\{x > 0\}$

is also valid, because if $x - 1 > 0$ is true before execution then the postcondition $x > 0$ is guaranteed to be true afterwards. Even though the second Hoare triple is valid, it is not useful in defining the assignment statement. The precondition of the first Hoare triple $x + 2 > 0$ is weaker than that of the second $x - 1 > 0$ because it puts less of an initial restriction on *x*. It only requires that *x* be greater than –2, while the other precondition requires that *x* be greater than 1. ∎

Exercises

1. Evaluate the following postfix expressions.

(a) 2 5 1 3 + − ×
(b) 2 5 1 + 3 − ×
(c) 2 5 + 1 − 3 ×
(d) 1 1 1 1 1 1 − − − − −

2. Convert the following infix expressions to postfix.

(a) $a + b - c \times d$
(b) $x \times (z - y) /$ w
(c) $F - G \times (H + I \times J)$
(d) $p \times (q \times (r + s / t) + u) + v$

3. Trace the execution of the algorithm for converting the following infix expression to postfix. Show the contents of the stack at each step of the conversion as in Example 7.6.

(a) 5 + 2 − 6 × 4
(b) 2 × (3 + 4 × 5 + 6)

4. Inspect the documentation of module TextModels and answer the following questions about procedure Delete.

(a) What does procedure Delete do?
(b) How many preconditions does it have?
(c) What is the first precondition?
(d) How many postconditions does it have?

5. What two things does the word OUT signify when it is placed before a formal parameter?

6. Identify whether each of the following parameter passing indicators is call by value, call by result, call by reference, or call by constant reference.

(a) IN
(b) OUT
(c) default
(d) VAR

7. Complete the following statements by choosing the correct option in parentheses.

(a) In call by value the procedure (does / does not) change the value of the actual parameter.
(b) In call by result the procedure (does / does not) change the value of the actual parameter.
(c) In call by reference the procedure (does / does not) change the value of the actual parameter.
(d) In call by constant reference the procedure (does / does not) change the value of the actual parameter.

8. Suppose PboxStackADS were written as a stack of strings instead of a stack of reals. Use the style of the interface in Figure 7.13 to write the interface for such a stack.

9. Suppose PboxStackADT were written as a stack of strings instead of a stack of reals. Use the style of the interface in Figure 7.18 to write the interface for such a stack.

10. What is the design-by-contract rule?

11. **(a)** What is the interpretation of the Hoare triple $\{P\}S\{Q\}$? **(b)** What is a formal specification?

12. Write the formal specification for **(a)** PboxListADT.Clear, **(b)** PboxStackADT.Pop. For part (b) you will need to use two rigid variables, one to save the initial value of the number of elements in the stack and one to save the initial value of the top of the stack. You may use $TopOf(s)$ in the precondition to indicate the value of the top of stack s. Note that PboxStackADT.Pop changes both s and val.

13. What is the weakest precondition for the assignment statement $x := x - 5$ whose post-condition is $x < 20$?

Problems

14. Construct a four-function reverse polish notation (RPN) real calculator as shown in the dialog box of Figure 7.22. The calculator has an internal stack and uses postfix notation to evaluate the result. When the user presses the Enter button push the value onto the stack and display the value in the field labeled Top. The button labeled "-" pops one item off the stack into a temporary variable, then pops a second item off the stack into another temporary variable. It subtracts the first from the second (notice the order), pushes the difference onto the stack, and displays it in the field labeled Top. For example, to evaluate the postfix expression 2 3 4 + × you would perform the sequence: type 2, press Enter, type 3, press Enter, type 4, press Enter, press +, press *. The figure shows the dialog box just before pressing * in the above sequence. Use the abstract data structure PboxStackADS.

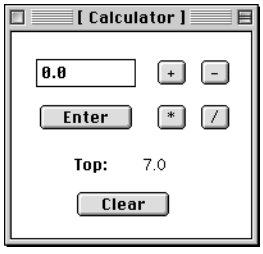


Figure 7.22
The four-function RPN calculator for Problem 14.

15. Do Problem 14, but use the abstract data type from PboxStackADT.

16. Expand your program of Problem 14 to make it a full-featured scientific calculator. Include the trigonometric functions sine, cosine, tangent and their inverses, and the natural logarithm and its inverse. Include a button for the user to enter π and compute the

square and the square root. Get the value of π from the Math module. Use the abstract data structure PboxStackADS.

17. Do Problem 16, but use the abstract data type from PboxStackADT.

18. The abstract list is more general than the abstract stack, because you can insert and remove from any location instead of only from the top. In PboxListADS, if you call procedure InsertAtN and give n the value 0 then the item will always be prepended to the front of the list. Furthermore, if you call procedure RemoveN and give n the value 0, then the item will always be removed from the front of the list. Write a program that implements a dialog box that looks and behaves exactly like Figure 7.3, but import PboxListADS instead of PboxStackADS. To pop a value with the procedures of PboxListADS you will need to get the value with procedure GetElementN before you remove it with RemoveN.

19. The abstract list is more general than the abstract stack, because you can insert and remove from any location instead of only from the top. In PboxListADT, if you call procedure InsertAtN and give the n the value 0 then the item will always be prepended to the front of the list. Furthermore, if you call procedure RemoveN and give n the value 0, then the item will always be removed from the front of the list. Write a program that implements a dialog box that looks and behaves exactly like Figure 7.9, but import PboxListADT instead of PboxStackADT. To pop a value with the procedures of PboxListADT you will need to get the value with procedure GetElementN before you remove it with RemoveN.

20. Implement a dialog box that looks and behaves like Figure 7.9, but with one additional button labeled "A to B". When the user presses this button an item should be popped off of Stack A and pushed onto Stack B. The text fields for Push and Pop should not change. Only the fields for the number of items in Stacks A and B should change. Use a temporary variable called temp in your procedure AToB to store the value between the pop and push operations.

21. Implement a dialog box that looks and behaves like Figure 7.3, but with one additional button labeled "Swap Top". If the stack contains two or more items when the user presses this button, the top two items on the stack should be exchanged. 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 two temporary variables called temp1 and temp2 in your procedure SwapTop to store the two values between the pop and push operations. Import module PboxStackADS.

22. Work Problem 21, but import module PboxStackADT.

23. Implement a dialog box that looks and behaves like Figure 7.16, but with one additional button labeled "Front to Back". If the list contains two or more items when the user presses this button, the item at the front of the list should be moved to the back of the list. Otherwise, the list should remain unchanged. All the text fields for the dialog box should not change, nor should the field for the number of items change. Use a temporary variable called temp in your procedure FrontToBack to store the front values between the remove and insert operations. Import module PboxListADS.

24. Work Problem 23, but import module PboxListADT.

25. In PboxListADS, if you call procedure InsertAtN and give the n the value PboxListADS.capacity then the item will always be appended to the rear of the list. Furthermore, if you call procedure RemoveN and give n the value 0, then the item will always be removed from the front of the list. Write a program that implements a dialog box that looks exactly like Figure 7.3, but with the Push button relabeled Enqueue, the Pop button relabeled Dequeue, the Clear Stack button relabeled Clear Queue, and the window title relabeled One Queue. The buttons must implement the FIFO policy as shown in Figure 7.21. The type of the values stored should be PboxListADS.T instead of REAL.

26. In PboxListADT, if you call procedure InsertAtN and give the n the value PboxListADS.capacity then the item will always be appended to the rear of the list. Furthermore, if you call procedure RemoveN and give n the value 0, then the item will always be removed from the front of the list. Write a program that implements a dialog box that looks exactly like Figure 7.9, but with the Push buttons relabeled Enqueue, the Pop buttons relabeled Dequeue, the Clear Stacks button relabeled Clear Queues, and the window title relabeled Two Queues. The buttons must implement the FIFO policy as shown in Figure 7.21. The type of the values stored should be PboxListADT.T instead of REAL.

27. Inspect the preconditions of all the procedures that are called in module Pbox07A of Figure 7.4 for the stack ADS and modify the module to insure that a trap can never occur. If the user enters a value in a dialog box that would violate a precondition when a button is pressed, the program should simply do nothing to the stack or the dialog box.

28. Do Problem 27 but with module Pbox07B in Figure 7.10 for the stack ADT.

29. Do Problem 27 but with module Pbox07C in Figure 7.17 for the list ADS.

30. Do Problem 27 but with module Pbox07D in Figure 7.20 for the list ADT.