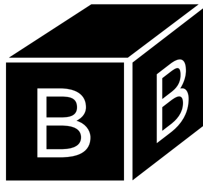


Chapter *21*

Linked Lists

Previous chapters show how Component Pascal allocates storage on the run-time stack when a proper procedure is called. First, storage is allocated for the parameters, then for the return address, and finally for the local variables of the procedure. When the program returns from a procedure, it deallocates the storage. When a function procedure is called, storage for the returned value is allocated followed by allocation for the same items as those for a proper procedure.

Component Pascal provides an alternate method for allocating and deallocating storage from main memory. It maintains a region in memory that is called the heap, which is separate from the stack. You do not control allocation and deallocation from the heap during procedure calls and returns. Instead, you allocate from the heap with the help of pointer variables. Allocation that is not triggered automatically by procedure calls is known as dynamic storage allocation.

The heap

Dynamic storage allocation

Pointers are common building blocks for implementing abstract data types and classes. This chapter presents the Component Pascal pointer type and shows how you can use pointers to implement a linked list abstract data type and a linked list class.

Pointer data types

When you declare an array, you must declare it to be an array of some type. For example, you can declare an array of integers or an array of real values. Pointers share this characteristic of arrays. When you declare a pointer, you must declare that it points to some type. The program in Figure 21.1 illustrates the Component Pascal pointer type. It shows how to declare a pointer variable and how to access the values associated with it.

The program declares type Node to be a record that contains an integer field i and a real field x. Local variable a is declared to be a pointer to a Node, that is, a pointer to a record. a is not a record. It is a pointer to a record. If a acquires a value during execution of the program, that value will not be a record. Instead, the value given to a will specify the memory location of where the record is stored, somewhere in the heap.

Figure 21.2 is a trace of the execution of the procedure in Figure 21.1. Figure 21.2(a) shows a after PointerExample1 is called. Because a is a local variable, it is allocated on the run-time stack when the procedure is called. Return address ra0 is the location of some instruction in the framework. Most local variables have unde-

fined values when they are allocated. Pointer variables, however, are initialized to the special value NIL. In Figure 21.2(a) the NIL value is shown as the dashed triangle.

```
MODULE Pbox21A;
    IMPORT StdLog;

    PROCEDURE PointerExample1*;
        TYPE
            Node = RECORD
                i: INTEGER;
                x: REAL
            END;
        VAR
            a: POINTER TO Node;
        BEGIN
            NEW(a);
            a.i := 6;
            a.x := 15.2;
            StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
            StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
        END PointerExample1;

END Pbox21A.
```

Figure 21.1
A program that illustrates the Component Pascal pointer type.

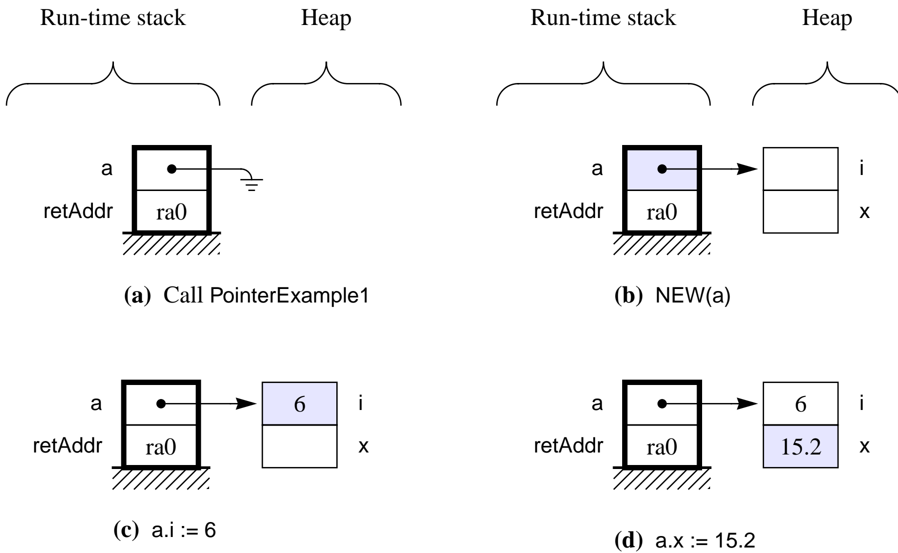


Figure 21.2
The trace of the procedure in Figure 21.1.

The first statement the program executes is

```
NEW(a)
```

The procedure NEW is a standard Component Pascal procedure that does two things:

The two actions of NEW

- It allocates storage from the heap. Because a was previously declared to be a pointer to a record with an integer and a real component, NEW(a) allocates enough memory to store a record with those components.
- It assigns to a the location of this newly allocated storage. So a now points to the location of a record.

Figure 21.2(b) indicates the effect of NEW(a). The box adjacent to the a box represents the storage allocated from the heap. The arrow pointing from the a box to the newly allocated box represents the value that NEW assigns to a.

The next statement the program executes is

```
a.i := 6
```

Figure 21.2(c) shows the effect of the assignment. In the same way that you access the field of a record by writing the name of the *record* followed by the field and separated by a period, you access the field of the record to which a pointer points by writing the name of the *pointer* followed by the field and separated by a period. Remember that a is not a record. It is a pointer to a record. Therefore, the integer field of the record to which a points gets 6.

The next statement,

```
a.x := 15.2
```

assigns a value to the real field of the record to which a points as shown in Figure 21.2(d).

The last statements

```
StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
```

simply output the following text to the Log.

```
a.i = 6
a.x = 15.2
```

Pointer assignments

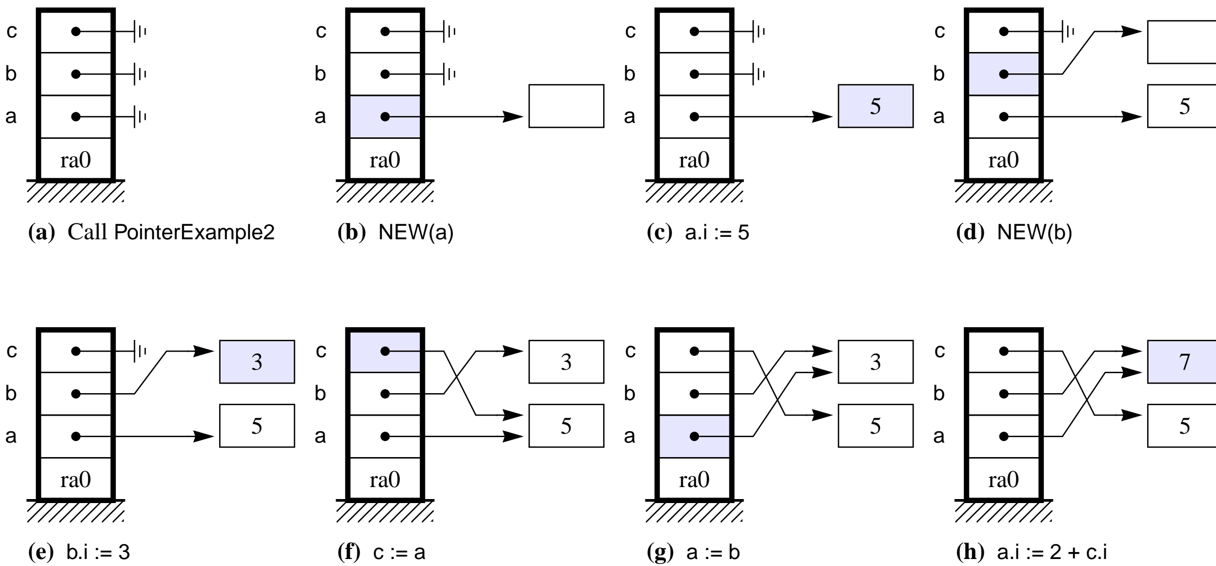
You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer “points to” an item, if you give the pointer’s value to a second pointer, the second pointer will point to the same item to which the first pointer points. The program in Figure 21.3 illustrates the effect of the assignment operation on pointers.

```
MODULE Pbox21B;
    IMPORT StdLog;

    PROCEDURE PointerExample2*;
        TYPE
            Node = RECORD
                i: INTEGER
            END;
        VAR
            a, b, c: POINTER TO Node;
        BEGIN
            NEW(a); a.i := 5;
            NEW(b); b.i := 3;
            c := a;
            a := b;
            a.i := 2 + c.i;
            StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
            StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
            StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
        END PointerExample2;

END Pbox21B.
```

Figure 21.3
The effect of the assignment operation on pointers.

The program allocates and assigns values to a.i and b.i, as Figure 21.4(a–e) shows. These operations are similar to those of the previous program.

Figure 21.4
A trace of Figure 21.3.



After b.i := 3 in Figure 21.4(e), the statement

```
c := a
```

gives the value of a to c, as Figure 21.4(f) shows. a is a pointer. Therefore c will point to the same memory location to which a points. After the assignment, c also points to the record that contains 5. Notice that the statement does not assign to c the value 5. It assigns to c the pointer to the record that contains 5.

The statement

```
a := b
```

copies the b pointer into a, as Figure 21.4(g) shows. As in all assignment statements, the previous value of a is destroyed. a no longer points to the record containing 5, but to the same record to which b points, namely the record containing 3.

The last assignment statement

```
a.i := 2 + c.i
```

contains the + arithmetic operation. Because a.i and c.i are integer variables and not pointers, the statement is legal. It adds 2 to the integer in the record to which c is pointing, 5, to get 7. The 7 is copied into the record to which a is pointing. As in all assignment statements, the original content of the memory location, 3, is destroyed.

The output statements

```
StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln;
```

produce

```
a.i = 7
b.i = 7
c.i = 5
```

Because a and b now point to the same record, the record containing 7, its value is printed twice.

Using pointers

Component Pascal allows you to declare a pointer to a record or to an array. It does not allow you to declare a pointer to any other type.

Example 21.1 The declaration

```
a: POINTER TO INTEGER;
```

is illegal because you cannot have a pointer to an integer, only to a record or an array. ∎

The period that separates the name of the pointer from the name of the field of the record to which it points is actually an abbreviation for a longer notation. In general, if pointer p points to a record r that contains field f, then

- p is a pointer
- p^ is the record r to which p points
- p^.f is field f of the record r to which p points.

The notation p.f is an abbreviation for the longer notation p^.f.

The period abbreviation for pointers to records

Example 21.2 The program statements in Figure 21.1

```
a.i := 6;
a.x := 15.2;
```

can be written in non abbreviated form with the circumflex ^ as

```
a^.i := 6;
a^.x := 15.2;
```

∎

You can output the components of a record to which a pointer points, but you cannot output the value of a pointer variable.

Example 21.3 The statement

```
StdLog.Int(a)
```

is illegal with a declared as in Figure 21.3. ∎

The only operations that are allowed on pointer data types are

The only operations allowed on pointer data types

:= assignment
= test for equality
test for inequality

Specifically, you cannot test if one pointer is greater than another, and you cannot perform mathematical operations on pointers.

Example 21.4 With a and b declared as in Figure 21.3, the test

```
IF a.i < b.i THEN
```

is legal, because you can test if an integer value is less than another integer value. However, the test

```
IF a < b THEN
```

is illegal because a and b are pointers. ∎

Example 21.5 With a declared as in Figure 21.3, the statement

```
a.i := a.i * 2
```

would be legal because a.i is an integer. On the other hand, the statement

```
a := a * 2
```

would be illegal, because you cannot multiply a pointer by 2. ∎

One of the most common errors in programming with pointers is to assume that a pointer points to a record when in fact it does not.

Example 21.6 Suppose a is declared as in Figure 21.3, and you forget to execute the NEW procedure as follows.

```
BEGIN
   a.i := 5;
```

The assignment statement will generate a trap with the error message

```
NIL dereference
```

Because a is NIL it does not point to anything, and a.i does not exist. The system protests with the run-time error message, complaining that you are referring to something with a but that a does not refer to anything. ∎

Linked lists

In practice, you frequently combine a pointer with other variables into a record. Then the pointer part of the record can point to yet another record, linking the two records together. The program in Figure 21.6 constructs a linked list of three real numbers. Each record in the linked list has two parts, a value part, which contains the value of a real number, and a next part, which points to the next record in the linked list. Figure 21.5 shows the structure of a record of type Node as declared in the program. The box labeled value will contain a real value and the box labeled next will contain a pointer value.

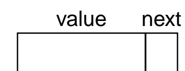


Figure 21.5
The structure of a record of type Node in Figure 21.6.

Figure 21.7 is a trace of the first part of the program in Figure 21.6, which creates a linked list. The following is a description of each statement executed by the first part of the program.

Figure 21.7(a) shows the allocated memory after the call to LinkedListExample. The program allocates storage for the local variables declared in the variable declaration part on the run-time stack. The variables first and p are not records. They are pointers to records. Initially their values are NIL.

```
MODULE Pbox21C;
   IMPORT StdLog;

   PROCEDURE LinkedListExample*;
      TYPE
         List = POINTER TO Node;
         Node = RECORD
            value: REAL;
            next: List
         END;
      VAR
         first, p: List;
      BEGIN
         (* Create linked list *)
         NEW(first); first.value := 4.5;
         p := first;
         NEW(first); first.value := 1.2;
         first.next := p; p := first;
         NEW(first); first.value := 7.3;
         first.next := p;
         (* Output linked list *)
         p := first;
         StdLog.Real(p.value); StdLog.String("  ");
         p := p.next;
         StdLog.Real(p.value); StdLog.String("  ");
         p := p.next;
         StdLog.Real(p.value); StdLog.String("  ")
      END LinkedListExample;

END Pbox21C.
```

Figure 21.6
A program that constructs a linked list of three real numbers.

Creating a linked list

Figure 21.7(b) shows the effect of the procedure call NEW(first). Procedure NEW does two things. First, it allocates enough storage from the heap for a record of type Node. Then it changes the value of first to point to the record just allocated. It also initializes the value of link to NIL. All pointers are automatically initialized to NIL whenever they are allocated, whether on the stack or from the heap. Then, first.value := 4.5 stores value 4.5 in the value part of the node to which first points.

Figure 21.7(c) shows the effect of the statement

```
p := first
```

This is a pointer assignment. It makes p point to the same node to which first points. Because first points to the node containing 4.5, p will point to the same node after the assignment.

Figure 21.7(d) shows that procedure call

```
NEW(first)
```

allocates another record from the heap and sets first to point to the newly allocated

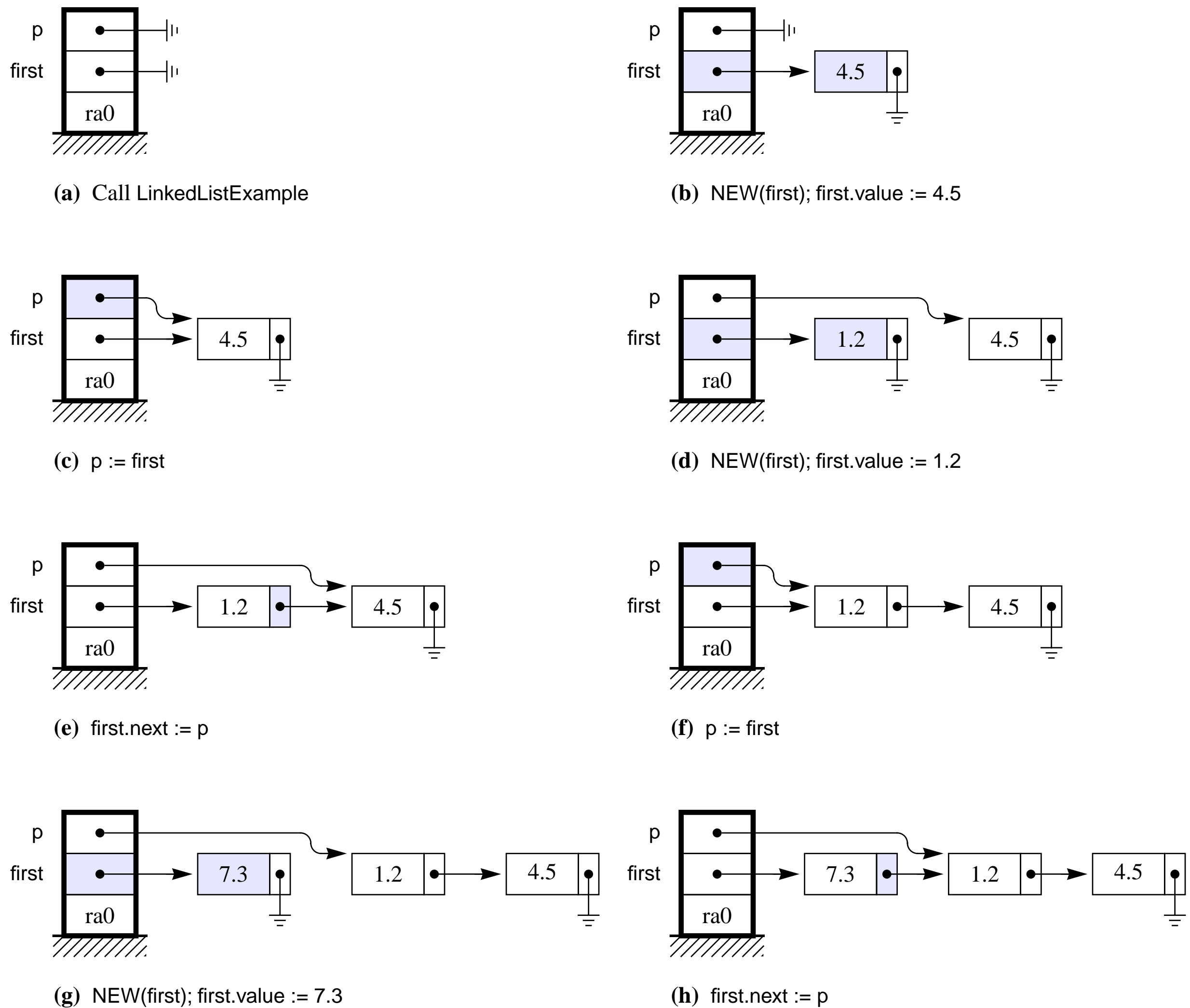


Figure 21.7
The trace of Figure 21.6 to create a linked list.

record. Then,

```
first.value := 1.2
```

sets the value part of the record to which first points, to 1.2.

Figure 21.7(e) shows the effect of the assignment statement

```
first.next := p
```

This is another pointer assignment. It makes first.next point to the same record to which p points. You can see from the figure that this statement is responsible for linking the 1.2 node to the 4.5 node.

Figure 21.7(f) shows the effect of the pointer assignment

```
p := first
```

This assignment statement makes p point to the same thing to which first points, namely, the newly allocated record.

Figure 21.7(g) shows that the procedure call

```
NEW(first)
```

allocates another record from the heap and sets first to point to the newly allocated record. Furthermore,

```
first.value := 7.3
```

sets the value part of that record to 7.3.

Figure 21.7(h) shows how the statement

```
first.next := p
```

links the 7.3 node to the 1.2 node.

Now the linked list is complete. The first record containing 7.3 in its value part is linked to the second record, which contains 1.2 in its value part. The second record is, in turn, linked to the third which contains 4.5 in its value part.

Outputting a linked list

The last part of the program outputs the linked list to the Log. Figure 21.8 is a trace. The idea is for p to start at the beginning of the list and to advance through it by way of the next field. Following is a description of each statement executed.

Figure 21.8(a) shows the effect of the assignment statement

```
p := first
```

Because p and first are both pointers, the assignment makes p point to the same record to which first points, namely the first record of the linked list. It is similar to an initializing statement. Because p.value is the value field of the first record, the StdLog.Real call outputs 7.3.

Figure 21.8(b) shows how the statement

```
p := p.next
```

advances p to the next record of the linked list. p.value is now the value of the second record. The StdLog.Real call outputs 1.2.

Figure 21.8(c) shows how the same statement

```
p := p.next
```

advances p to the next record of the linked list again. p.value is now the value of the last record. The StdLog.Real call outputs 4.5.

The action of the pointer p is typical of algorithms that must process information from every node in a linked list. You initialize a local pointer variable p to point to the first node in the list. You process all the nodes of the list by putting the statement

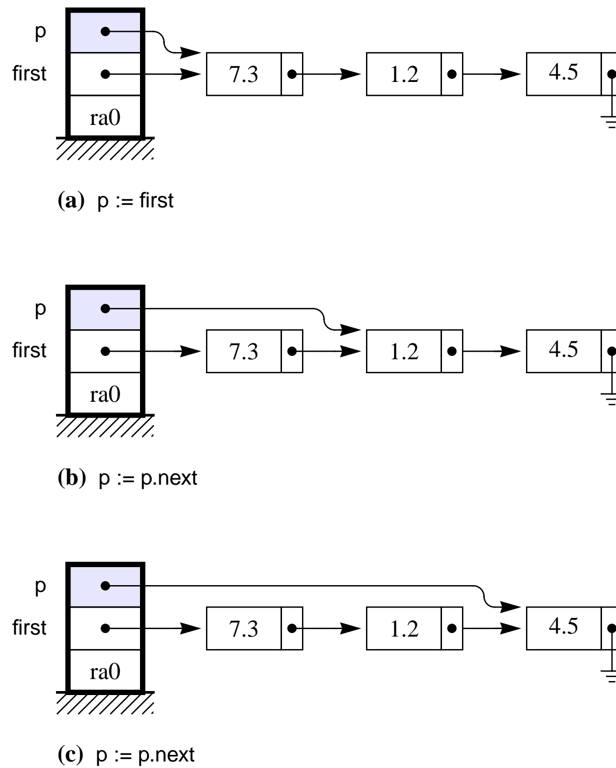


Figure 21.8
The trace of the procedure in Figure 21.6 to output a linked list.

p := p.next in the body of a loop. Each time the loop executes, p advances to the next node of the linked list. Also in the body of the loop are any statements that process the data in the value part of the node. How would such a loop terminate? When p points to the last node of the linked list. That happens when p.next = NIL. The statements to output the linked list could be written with a single loop as follows.

```
p := first;
WHILE p # NIL DO
   StdLog.Real(p.value); StdLog.String("  ");
   p := p.next
END
```

This loop works correctly even if the list is empty, because an empty list would have first equal to NIL. The test for a WHILE statement is at the beginning of the loop, so the test would be false the first time and the body of the loop would never execute.

In Figure 21.6, first and p are local variables that are allocated on the run-time stack. When the procedure terminates, first and p are all deallocated and so no longer point to any of the nodes of the linked list. So how do the nodes of the linked list get deallocated? They stay allocated until the heap runs out of memory and the execution of a NEW statement requires memory from the heap. At that point, the Component Pascal system initiates an operation known as *garbage collection*. The garbage collection algorithm sweeps through all the nodes in the heap and deallocates all

Automatic garbage collection

those that are unreachable from any active pointers. This automatic garbage collection feature is a major advantage of Component Pascal over many other programming languages. Languages that do not have automatic garbage collection require the programmer to deallocate any unused nodes. It is easy to make programming errors in the deallocation of dynamic memory, because the effects of incorrect deallocation frequently do not show up immediately.

Class assignments

There are two basic relationships in object-oriented design—class composition and inheritance. The examples in this section assume that Alpha, Beta, and Gamma are each a record type and are related by the second relationship, inheritance. Specifically, Beta inherits from Alpha, and Gamma inherits from Alpha. The Unified Modeling Language (UML) symbol for inheritance is the triangle. Figure 21.9 illustrates the object-oriented relationship of inheritance between Alpha, Beta, and Gamma in a UML class diagram. The Component Pascal term for inheritance is extension. Beta is an extension of Alpha. Beta is called the subclass, and Alpha is called the superclass.

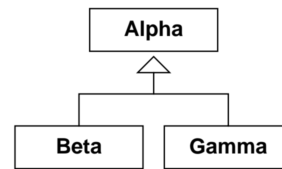


Figure 21.9
The object-oriented relationship of inheritance between Alpha, Beta, and Gamma.

The examples also assume that AlphaPtr, BetaPtr, and GammaPtr are pointers to Alpha, Beta, and Gamma respectively and that alphaPtr, betaPtr, and gammaPtr are variables of type AlphaPtr, BetaPtr, and GammaPtr respectively as follows.

```
TYPE
   AlphaPtr = POINTER TO Alpha;
   BetaPtr = POINTER TO Beta;
   GammaPtr = POINTER TO Gamma;
VAR
   alphaPtr: AlphaPtr;
   betaPtr: BetaPtr;
   gammaPtr: GammaPtr;
```

The word class is object-oriented terminology for type, and the word object is object-oriented terminology for variable. The statement, "Variable alphaPtr has type AlphaPtr", becomes in object-oriented terminology, "Object alphaPtr is an instantiation of class AlphaPtr".

The idea behind inheritance is that Alpha is the more general class and Beta is the more specific class. The fundamental class assignment rule is that you can assign the specific to the general, but you cannot assign the general to the specific.

The fundamental class assignment rule

Example 21.7 With alphaPtr and betaPtr declared as above, and Beta inheriting from Alpha as in Figure 21.9, the assignment

```
betaPtr := alphaPtr
```

is *not* legal, but the assignment

```
alphaPtr := betaPtr
```

is legal. ∎

In Example 21.7, what is the type of alphaPtr after the legal assignment? It would appear from its declaration that it has type AlphaPtr, but because it has just been assigned betaPtr, it would appear to have beta's type, which is BetaPtr. In fact, it has both. Its static type is AlphaPtr and its dynamic type is BetaPtr. In computer science terminology, the word static means something that happens or is determined at compile time, and the word dynamic means something that happens or is determined at execution time. The compiler can determine the static type of alphaPtr from its declaration in the VAR section. But, the assignment to alphaPtr does not occur until the program is executing.

The meaning of static and dynamic

It is possible that at some later time alphaPtr could get the value of gammaPtr, at which time its dynamic type would change to GammaPtr. To morph is to change from one object into another object, as in the metamorphosis of a caterpillar into a butterfly. In object-oriented terminology, alpha is polymorphic because it can change from being a beta object to being a gamma object.

Polymorphism

From the fundamental class assignment rule it follows that an object's dynamic type is either the same as its static type or is more specific than its static type. Suppose you have one object with some static type, and another object of a more general static type but whose dynamic type is the same as the static type of the first object. You want to assign the second object to the first. According to the static types the assignment violates the fundamental class assignment rule. But if the dynamic type of the second object is the same as the static type of the first object it seems that you should be able to make the assignment.

Example 21.8 Suppose that betaTwoPtr has declaration

```
VAR
   betaTwoPtr: BetaPtr;
```

and the assignment statement

```
alphaPtr := betaTwoPtr
```

executes. This assignment statement is legal by the class assignment rule, because alphaPtr is more general than betaTwoPtr. Furthermore, alphaPtr now has dynamic type BetaPtr. It seems as though the assignment

```
betaPtr := alphaPtr
```

should now be legal, because the dynamic type of alphaPtr is the same as the static type of betaPtr. ∎

Component Pascal provides a way. To make the assignment, you must append a type guard to the variable on the right hand side of the assignment statement, which consists of an extension (that is, a subclass) of its static type enclosed in parentheses. The assignment statement will compile correctly only if the type guard is an extension of the expression that it guards.

Type guards

Example 21.9 With the above declarations,

```
alphaPtr (BetaPtr)
```

is a valid type guard, because type BetaPtr is a subclass of AlphaPtr, which is the type of alphaPtr. However, the type guard

```
betaPtr (AlphaPtr)
```

will not compile, because AlphaPtr is not a subclass of BetaPtr. ∎

Example 21.10 Because a type is a subclass of itself, the type guard

```
alphaPtr (AlphaPtr)
```

although not very useful, is legal. ∎

To determine if an assignment statement will compile, the compiler treats the guarded variable as if it had the type of the guard and applies the class assignment rule.

Example 21.11 With alphaPtr and betaPtr declared as above, the assignment statement

```
betaPtr := alphaPtr (BetaPtr)
```

will compile, because if alphaPtr had the type BetaPtr it would compile. ∎

If variable v has type guard T, the guarded expression v(T) asserts that the dynamic type of v is T or an extension (that is, a subclass) of T. The program will trap if during execution the dynamic type of v is not T or a subclass of T.

Example 21.12 The assignment statement in Example 21.11 will always compile successfully. If during execution alphaPtr first gets betaTwoPtr as in Example 21.8 the assignment will also execute successfully. However, if during execution alphaPtr first gets gammaPtr (a legal assignment by the class assignment rule) then the assignment statement in Example 21.11 will trap. ∎

The ANYREC record

Component Pascal has a special record named ANYREC, which is the most general record of all. Any record that does not inherit from any other record automatically inherits from ANYREC. So, even though it is not shown in the UML diagram of Figure 21.9, class Alpha inherits from ANYREC.

Example 21.13 Suppose Alpha and Beta have the inheritance relationship of Figure 21.9, and alphaPtr and betaPtr are declared as above. If you declare

```
VAR
   deltaPtr: POINTER TO ANYREC;
```

then the assignments

```
deltaPtr := alphaPtr;
deltaPtr := betaPtr
```

are legal. The second assignment is based on the transitive property of inheritance. That is, because Beta inherits from Alpha, and Alpha inherits from ANYREC, Beta inherits from ANYREC. So, deltaPtr is more general than betaPtr and can get its value. The assignment

```
alphaPtr := deltaPtr (AlphaPtr)
```

will compile successfully, but will execute successfully only if the dynamic type of deltaPtr is AlphaPtr or a subclass of AlphaPtr. ▮

A circular linked list ADT

It is possible to package a linked list as an abstract data structure, an abstract data type, or a class. The same advantages and disadvantages apply to any ADS, ADT, and class. Namely, an ADS is appropriate when there is only one data structure. To implement an ADS, you put the data structure in the server module and usually export only the procedures that operate on the data structure. An ADT and a class are appropriate when the client module might want more than one instance of the data structure. The server module exports the type of the data structure so the client module is free to declare as many data structures as needed.

Figure 21.10 shows the interface of a list abstract data type. It has two interesting features. First, it is a circular list in which the next field of the last node is not NIL, but points back to the first node in the list. Second, the value part of each node is not limited to any particular type like REAL or INTEGER. Instead, the value part is a pointer to ANYREC. The client is free to define any record it desires to store in the value part of each node. The ADT maintains a current position in the circular list, which the client can change via the GoNext procedure.

```
DEFINITION PboxCListADT;

   TYPE
      CList = POINTER TO Node;

   PROCEDURE Clear (OUT lst: CList);
   PROCEDURE Empty (lst: CList): BOOLEAN;
   PROCEDURE GoNext (VAR lst: CList);
   PROCEDURE Insert (VAR lst: CList; val: POINTER TO ANYREC);
   PROCEDURE NodeContent (lst: CList): POINTER TO ANYREC;

END PboxCListADT.
```

Figure 21.10
The interface of the circular list abstract data type.

Like all interfaces, the one in Figure 21.10 hides the details of the implementa-

tion. A client can use the procedures with the circular list without knowing anything about how they are implemented. Here are the specifications of the procedures.

The documentation of the clear procedure is

```
PROCEDURE Clear (OUT lst: CList)
post
lst is cleared to the empty list.
```

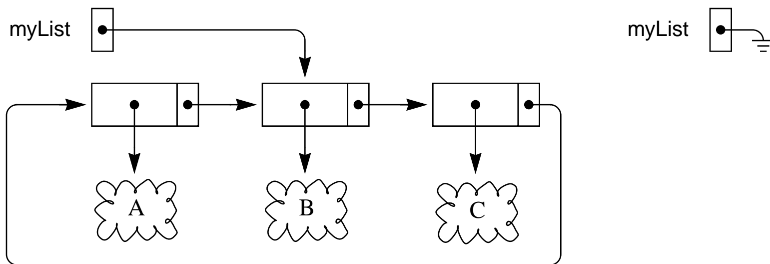
As an example of what procedure Clear does, suppose you have a variable named myList that has been declared as follows

```
VAR
   myList: PboxCListADT.CList;
```

and has the structure shown in Figure 21.11(a) where it contains three items. The clouds in Figure 21.11(a) represent records of some unknown type to which the pointers to ANYREC point. If you execute the statement

```
PboxCListADT.Clear(myList)
```

then myList is cleared to the empty list, as shown in Figure 21.11(b).



(a) Before PboxCListADT.Clear(myList)

myList

(b) After PboxCListADT.Clear(myList)

Figure 21.11
The result of calling procedure Clear.

The documentation of the procedure Empty is

```
PROCEDURE Empty (lst: CList): BOOLEAN
post
Returns TRUE if lst is empty. Otherwise returns FALSE.
```

If myList has the structure of Figure 21.11(a), then PboxCListADT.Empty(myList) returns FALSE, but if myList has the structure of Figure 21.11(b) it returns TRUE.

Procedure GoNext advances the list to the next node in the list. Its documentation is

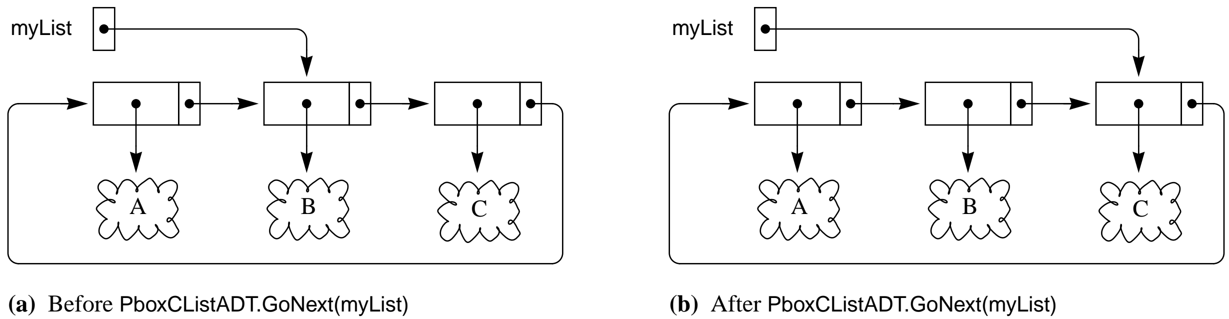
PROCEDURE **GoNext** (VAR lst: CList)
pre
lst is not empty. 20
post
The next location in lst is designated as the current item.

Figure 21.12 shows the effect of executing

PboxCListADT.GoNext(myList)

One more execution of GoNext would make myList point to the node whose value part points to the A record.

Figure 21.12
The result of calling procedure GoNext.



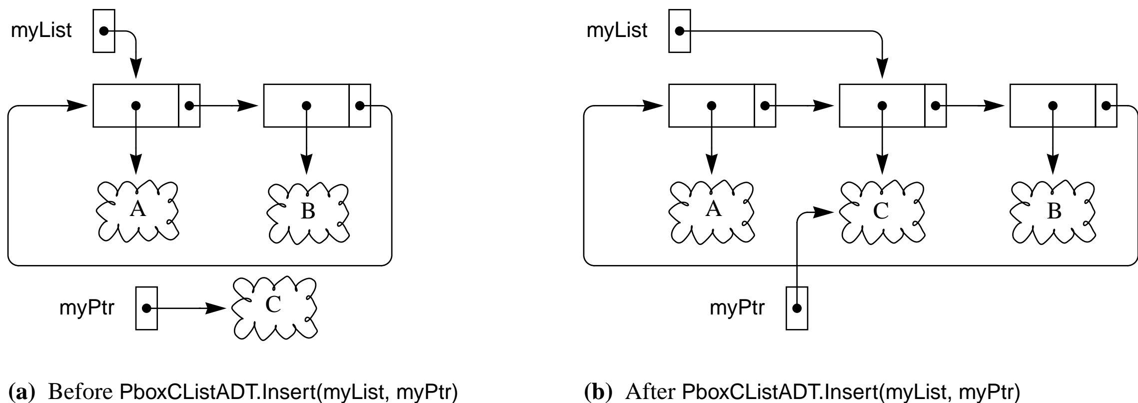
(a) Before PboxCListADT.GoNext(myList)

(b) After PboxCListADT.GoNext(myList)

Procedure Insert assumes that val is a pointer to some record. It inserts a new node into the list whose value part points to the same record that val points to. Its documentation is

PROCEDURE **Insert** (VAR lst: CList; val: POINTER TO ANYREC)
post
Value val is inserted in lst after the current item, and it becomes the current item.

Figure 21.13
The result of calling procedure Insert.



(a) Before PboxCListADT.Insert(myList, myPtr)

(b) After PboxCListADT.Insert(myList, myPtr)

Figure 21.13 shows the effect of executing the statement

```
PboxCListADT.Insert(myList, myPtr)
```

Before execution, myPtr points to some record. After execution, myPtr still points to the record but a new node is created and inserted into the circular list. The value part of the new node also points to the record.

The last procedure NodeContent returns the value part of the current node, that is, the node to which lst points. Its documentation is

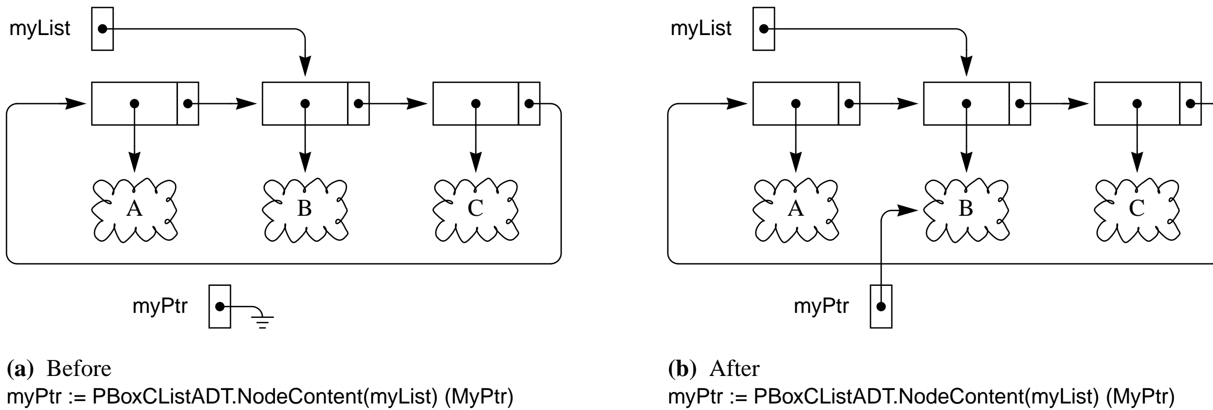
```
PROCEDURE NodeContent (lst: CList): POINTER TO ANYREC
pre
lst is not empty.    20
post
Returns the content from the current item of lst.
```

The purpose of NodeContent is for the programmer to retrieve the content of the current node. For example, suppose myPtr is a pointer of type MyPtr with an initial value of NIL as in Figure 21.14(a). After executing the statement

```
myPtr := PboxCListADT.NodeContent(myList) (MyPtr)
```

myPtr points to the record to which the value part of the current node in the list points as in Figure 21.14(b). Note the required use of the type guard (MyPtr), because myPtr is more specific than POINTER TO ANYREC. The programmer can then use myPtr to access the various fields of the record.

Figure 21.14
The result of calling procedure NodeContent when myList initially has the structure of Figure 21.11(a).



(a) Before
myPtr := PBoxCListADT.NodeContent(myList) (MyPtr)

(b) After
myPtr := PBoxCListADT.NodeContent(myList) (MyPtr)

Figure 21.15 is the dialog box for a program that uses the circular list in PboxCListADT. The program stores a list of books where the type Book is defined as

```
TYPE
    String64 = ARRAY 64 OF CHAR;
    Book = POINTER TO RECORD
        title: String64;
        author: String64;
        price: REAL
    END;
```

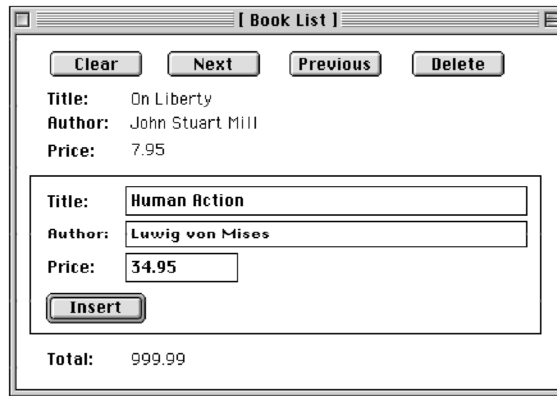


Figure 21.15 The dialog box for a program that uses the circular list from module PboxCListADT.

A book is a pointer to a record with three fields—one for the title of the book, one for the author, and one for the price. The dialog box allows the user to enter information into each of these fields, and then insert the book into the circular list. The dialog box represents a situation where the user has entered the information for the book *Human Action*, inserted it, and then pressed the Next button to display the next book in the list, which is *On Liberty*. Figure 21.16 shows the corresponding structure of the circular list.

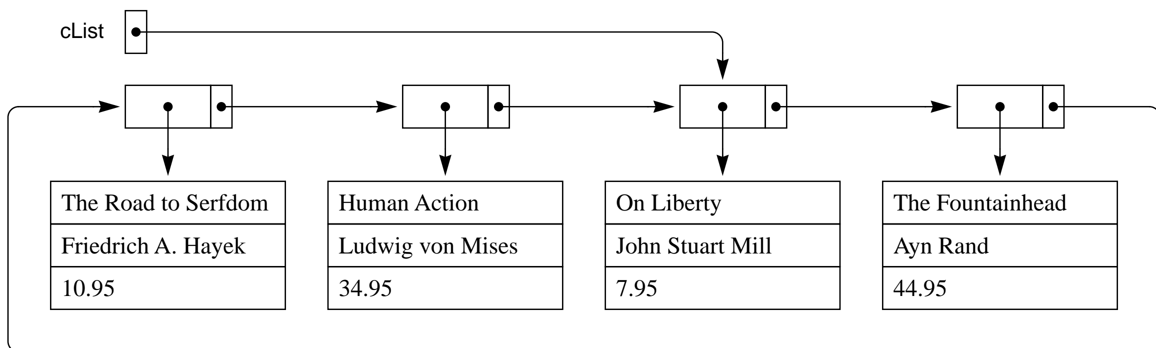


Figure 21.16 The circular list corresponding to the dialog box in Figure 21.15.

The dialog box has five buttons. The Clear button will obviously execute a procedure that calls PboxCListADT.Clear, the Next button will execute a procedure that calls PboxCListADT.GoNext, and the Insert button will execute a procedure that calls PboxCListADT.Insert. The Previous button allows the user to display the content of

the previous node in the list and the Delete button allows the user to delete the current node from the list. These features cannot be implemented directly with the procedures of Figure 21.10. Implementation of additional procedures to provide these capabilities is left as a problem for the student at the end of this chapter. Also left as a problem for the student is computation of the total price of all the books in the linked list, which has a stub value of 999.99 in Figure 21.15.

Figure 21.17 is the program that implements the dialog box of Figure 21.15. The module contains a global variable cList, which is the circular list of books, along with the usual d interactor for the dialog box.

```
MODULE Pbox21D;
   IMPORT Dialog, PboxCListADT;

   TYPE
      String64 = ARRAY 64 OF CHAR;
      Book = POINTER TO RECORD
         title: String64;
         author: String64;
         price: REAL
      END;

   VAR
      d*: RECORD
         titleOut-, authorOut- : String64;
         priceOut-: REAL;
         titleIn*, authorIn* : String64;
         priceIn*: REAL;
         total-: REAL
      END;
      cList: PboxCListADT.CList;

   PROCEDURE ClearDialog;
   BEGIN
      d.titleOut := ""; d.authorOut := ""; d.priceOut := 0.0;
      d.titleIn := ""; d.authorIn := ""; d.priceIn := 0.0;
      d.total := 0.0
   END ClearDialog;

   PROCEDURE SetBookOut (b: Book);
   BEGIN
      d.titleOut := b.title;
      d.authorOut := b.author;
      d.priceOut := b.price
   END SetBookOut;

   PROCEDURE SetTotal;
   BEGIN
      (* A problem for the student *)
      d.total := 999.99
   END SetTotal;
```

Figure 21.17
The program that implements the dialog box of Figure 21.15.

```
    PROCEDURE Clear*;
    BEGIN
        ClearDialog;
        PboxCListADT.Clear(cList);
        Dialog.Update(d)
    END Clear;

    PROCEDURE Next*;
        VAR
            book: Book;
    BEGIN
        IF ~PboxCListADT.Empty(cList) THEN
            PboxCListADT.GoNext(cList);
            book := PboxCListADT.NodeContent(cList) (Book);
            SetBookOut(book);
            Dialog.Update(d)
        END
    END Next;

    PROCEDURE Previous*;
    BEGIN
        (* A problem for the student *)
    END Previous;

    PROCEDURE Delete*;
    BEGIN
        (* A problem for the student *)
    END Delete;

    PROCEDURE Insert*;
        VAR
            book: Book;
    BEGIN
        NEW(book);
        book.title := d.titleIn;
        book.author := d.authorIn;
        book.price := d.priceIn;
        PboxCListADT.Insert(cList, book);
        SetBookOut(book);
        SetTotal;
        Dialog.Update(d)
    END Insert;

BEGIN
    Clear
END Pbox21D.
```

Figure 21.17
Continued.

Procedure Clear

Execution of the Clear procedure is straightforward. It calls ClearDialog, which clears the fields in the dialog box, then calls the Clear procedure for the circular list to make cList empty. It then updates the dialog box to make the changes visible.

Note that procedure Clear is executed when the module is loaded.

Procedure Next

Procedure Next first checks if cList is empty. If it is, nothing happens. Otherwise, the precondition for GoNext is satisfied, and it executes

```
PboxCListADT.GoNext(cList)
```

which makes cList point to the next node in the linked list. Now the procedure needs to access the fields of the book contained in the value part of the node so it can display them on the dialog box. It gets the information by executing

```
book := PboxCListADT.NodeContent(cList) (Book)
```

where book is a local variable of type Book. The type guard (Book) is necessary because NodeContent returns a pointer to ANYREC, which is more general than Book. Now that Book has a value the procedure call

```
SetBookOut(book)
```

puts its values in the d record, after which time the dialog is updated to make the changes visible.

Procedure Insert

Procedure Insert also has a local variable book of type Book. First it executes

```
NEW(book)
```

which creates a new record with three fields—title, author, and price—and sets book to point to the newly allocated record. Then, the statements

```
book.title := d.titleIn;
book.author := d.authorIn;
book.price := d.priceIn
```

transfer the data from the input fields of the dialog box to the newly allocated record. The procedure call

```
PboxCListADT.Insert(cList, book)
```

inserts the new book into the list as shown in Figure 21.13, and

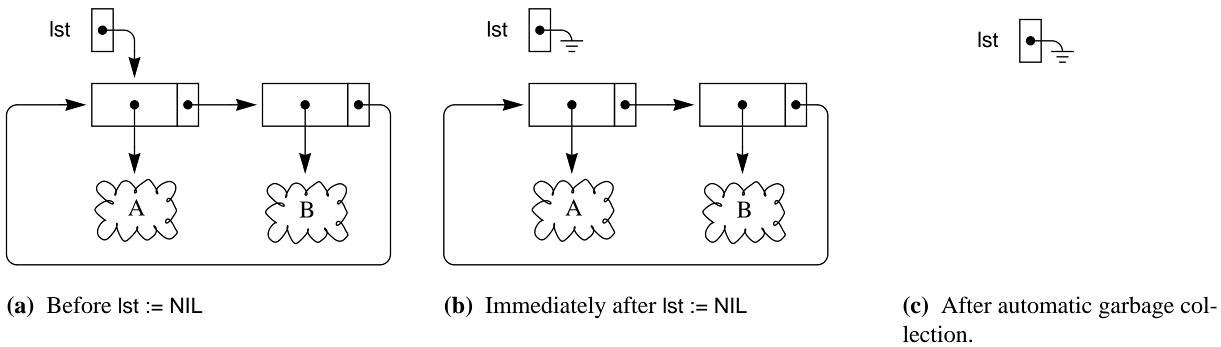
```
SetBookOut(book);
SetTotal
```

sets the output fields in the dialog box, after which they are updated to make the changes visible.

Figure 21.18 shows the implementation of the circular linked list. The exported type CList is a pointer to Node where Node has a value field and next field as does the node in Figure 21.5, page 469. The difference here is that the value part is a pointer to ANYREC.

```
MODULE PboxCListADT;

    TYPE
        CList* = POINTER TO Node;
        Node = RECORD
            value: POINTER TO ANYREC;
            next: CList
        END;

    PROCEDURE Clear* (OUT lst: CList);
    BEGIN
        lst := NIL
    END Clear;

    PROCEDURE Empty* (lst: CList): BOOLEAN;
    BEGIN
        RETURN lst = NIL
    END Empty;

    PROCEDURE GoNext* (VAR lst: CList);
    BEGIN
        ASSERT (lst # NIL, 20);
        lst := lst.next
    END GoNext;

    PROCEDURE NodeContent* (lst: CList): POINTER TO ANYREC;
    BEGIN
        ASSERT (lst # NIL, 20);
        RETURN lst.value
    END NodeContent;

    PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
        VAR
            temp: CList;
    BEGIN
        IF lst = NIL THEN
            NEW(lst);
            lst.value := val;
            lst.next := lst
        ELSE
            temp := lst.next;
            NEW(lst.next);
            lst := lst.next;
            lst.value := val;
            lst.next := temp
        END
    END Insert;

END PboxCListADT.
```

Figure 21.18
Implementation of the circular list PboxCListADT.CList.



(a) Before lst := NIL **(b)** Immediately after lst := NIL **(c)** After automatic garbage collection.

Figure 21.19
Implementation of procedure Clear with automatic garbage collection.

Procedure Clear simply sets lst to NIL. Something important is going on behind the scenes here—the process known as automatic garbage collection. Figure 21.19 shows the effect of automatic garbage collection. Immediately after execution of lst := NIL, the nodes of the circular list are still allocated from the heap. However, there is no way they can be used, because there are no pointers from any program that link to them. Periodically the BlackBox framework detects every allocated piece of memory in the heap that cannot be reached from any active pointers, and automatically returns each one to the heap so the storage can be reused. If Component Pascal did not provide for automatic garbage collection, procedure Clear would be more complicated. It would require a loop to advance a pointer through the list, individually returning each unused node to the heap. Most modern programming languages have automatic garbage collection, but a few older programming languages still in widespread use do not.

Procedure Empty simply executes

Procedure PboxCListADT.Empty

```
RETURN lst = NIL
```

If list lst is empty, lst equals NIL and the boolean expression lst = NIL is true. Procedure GoNext implements its precondition with the ASSERT statement, and then executes

Procedure PboxCListADT.GoNext

```
lst := lst.next
```

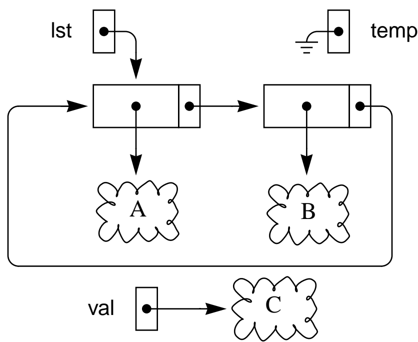
in the usual manner. Formal parameter lst is called by reference because the actual parameter must change. Procedure NodeContent is another one-liner. After its precondition test it simply returns lst.value. See Figure 21.14 for the effect.

Procedure PboxCListADT.NodeContent

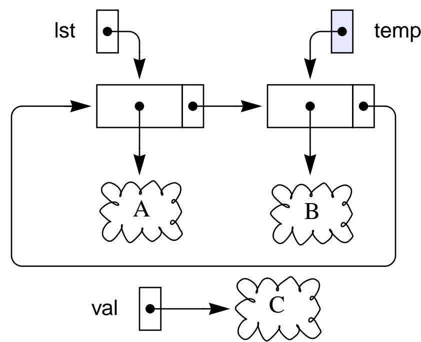
Procedure Insert takes a pointer to an existing record, creates a new node in the circular list, and sets the value part of the new node to point to the same record. There are two cases depending on whether lst is empty. Figure 21.20 shows the nonempty case with the pointers labeled with their formal parameters. It corresponds to Figure 21.13 where the pointers are labeled with the corresponding actual parameters.

Here is how procedure Insert works. Figure 21.20(a) shows the initial configuration with lst pointing to one of the nodes in the circular list and val pointing to a record that needs to be inserted. The first statement

Procedure PboxCListADT.Insert

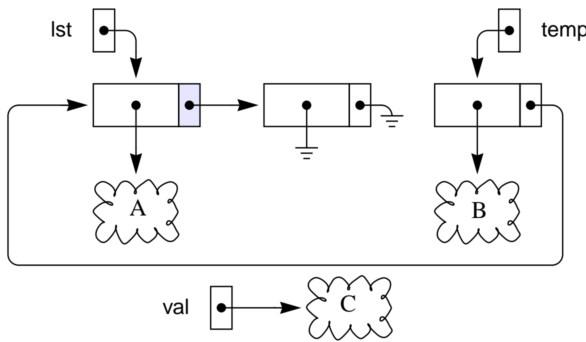


(a) Initial.

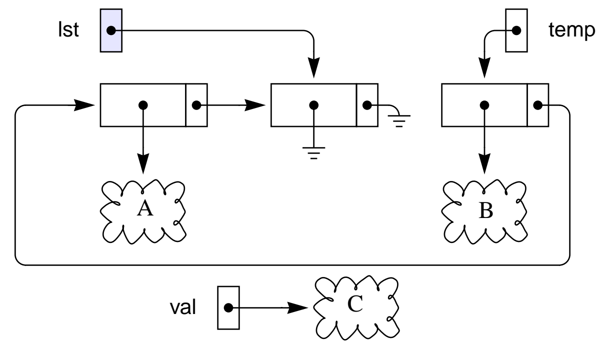


(b) temp := lst.next.

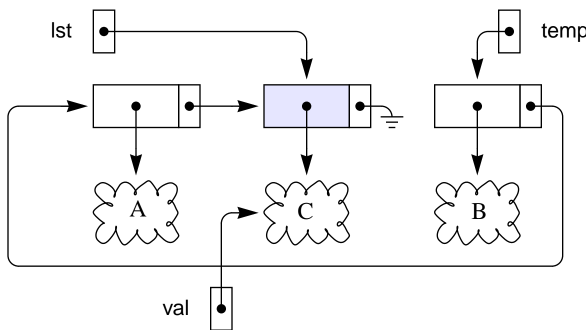
Figure 21.20
Execution of procedure PboxCListADT.Insert with a nonempty list.



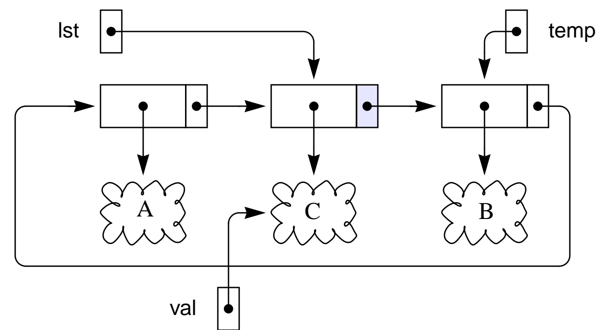
(c) NEW(lst.next).



(d) lst := lst.next.



(e) lst.value := val.



(f) lst.next := temp.

```
temp := lst.next
```

sets the temporary pointer to point to the node after the current one as in Figure 21.20(b). It is necessary to have a pointer to this node, because the link between it and the original one will be broken. Figure 21.20(c) shows the effect of

```
NEW(lst.next)
```

As always, NEW does two things—allocates storage from the heap and makes its parameter point to allocated storage. In this case, lst.next is the parameter. Its type is Node, so storage for a Node is allocated and lst.next points to it. The newly allocated node has two pointer fields that Component Pascal initializes to NIL. You can see from the figure that lst.next no longer points to the node to which temp points. The next statement

```
lst := lst.next
```

brings lst over to the new node as in Figure 21.20(d). Figure 21.20(e) shows the effect of

```
lst.value := val
```

which is yet another pointer assignment. It makes lst.value point to the same record to which val points. This is how the record is placed into the list. Figure 21.20(f) shows how

```
lst.next := temp
```

links the new current node to the next node. It now becomes clear why you need temp to be able to link up the new node to the next one.

A circular doubly-linked list

To implement the Previous and Delete buttons on the dialog box of Figure 21.15 you will need to modify module PboxCListADT. To change the current position to the previous node it is convenient for each node to have a link not only to the next node but to the previous one as well. The definition of a node is augmented to

```
Node = RECORD
    prev: CList;
    value: POINTER TO ANYREC;
    next: CList
END;
```

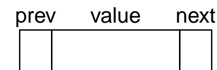


Figure 21.21
The structure of a record of type Node for a doubly-linked list.

as in Figure 21.21. Figure 21.22 shows the doubly linked version of the circular list in Figure 21.11.

Because of the extra prev link in each node you will need to add some processing to procedure PboxCListADT.Insert. The prev link of the next node must be made to

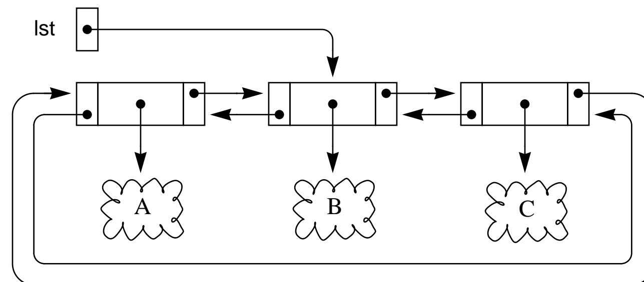


Figure 21.22
A circular doubly-linked list that corresponds to the singly-linked list of Figure 21.11

point to the newly inserted node, and the prev link of the newly inserted node must be made to point to the old current node.

With the prev link it is simple to implement a procedure that changes the current position to the previous node. Use the technique as in PboxCListADT.GoNext but with the prev link instead of the next link. It is more complicated but possible to implement the Previous button of the dialog box in Figure 21.15 without modifying PboxCListADT at all. You can find the previous node by looping all the way around the list until you get to the node just before the current one. Implementation of the Previous button with this approach is a problem for the student at the end of the chapter.

A procedure that deletes the current node should have as its precondition that the list is not empty, because it is impossible to delete a node from an empty list. There is a special case if the list contains a single node, because after the deletion the list is empty. Figure 21.23 shows what you must do to the nonempty list of Figure 21.22 to delete the current node. Make the links of the previous and the next node bypass the current node, and set lst to the previous node. After automatic garbage collection, the node will be reclaimed. It is not necessary to change the links of the deleted node. The fact that it is possible to get *from* the deleted node *to* the list is irrelevant. What matters is that it is impossible to get *to* the deleted node from any accessible pointer. The inaccessibility of the deleted node is a sufficient condition for automatic garbage collection. The garbage collector will not necessarily collect the nodes to which the deleted pointers point. In Figure 21.23, record B is also inaccessible and will be garbage collected along with its list node.

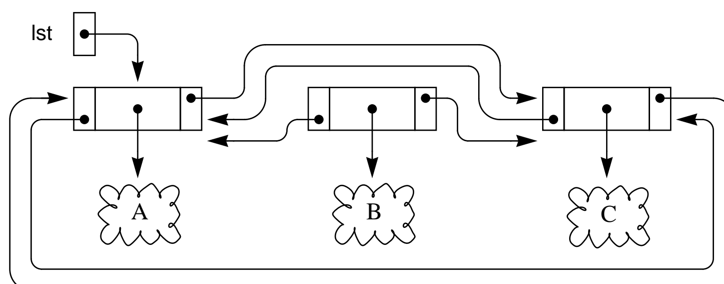


Figure 21.23
Deleting a node from a doubly linked circular list with the list initially as in Figure 21.22.

Record assignment

Both arrays and records are collections of values. With arrays, the values must all be of the same type but with records they need not be. If you have two array variables of the same type, say a and b, and you make an assignment between them, say a := b, then every element of b gets copied to the corresponding element of a. Figure 4.17(a), page 68, shows such a copy for arrays of characters.

It is possible to assign a whole record to another one, provided they have the same type. In the same way that assignment of one array to another causes every element of the array to be copied, assignment of one record to another causes every field to be copied.

All fields get copied in a record assignment

Example 21.14 Suppose type Composer is a record with two fields, and composerA and composerB are declared as follows.

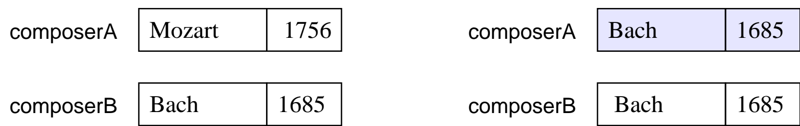
```
TYPE
   Composer = RECORD
      name: ARRAY 32 OF CHAR;
      birthYear: INTEGER
   END;
VAR
   composerA, composerB: Composer;
```

If composerA and composerB have the values in Figure 21.24(a), and you make the assignment

```
composerA := composerB
```

then both fields of composerB get copied to composerA as in Figure 21.24(b). That is, the assignment is equivalent to

```
composerA.name := composerB.name;
composerA.birthYear := composerB.birthYear
```



(a) Before composerA := composerB. **(b)** After composerA := composerB.

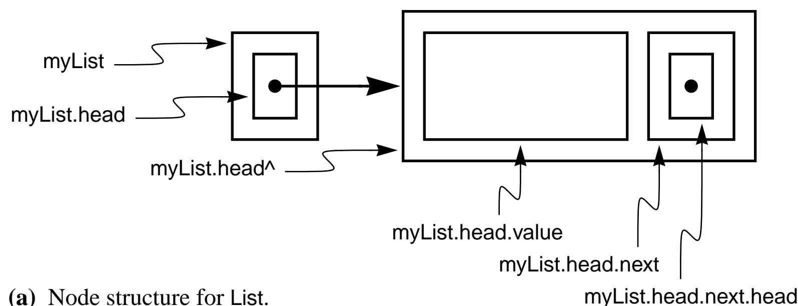
Figure 21.24
Record assignment.

The declaration of CList in PboxCListADT defines CList to be a pointer to a node. The following section declares List from module PboxLListObj to be a linked list that has the same interface as the list presented in Chapter 7 (whose implementation is shown in Chapter 17). Rather than implement List as an ADT like CList, the following section implements List as a class. Because it is implemented as a class, List is

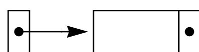
not defined to be a pointer, but a record. Here is the declaration.

```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```

Type List is a record containing only one field named head, which is a pointer to a Node. As usual, a Node is a record with two fields—value and next—and next has type List. With this setup, List is not a pointer. It is a record that contains a pointer. Furthermore, the next field is not a pointer. It is also a record that contains a pointer. That makes next a record inside of a record. Figure 21.25(a) shows the structure of a node for List assuming that myList is a variable of type List. To keep the diagrams simple, the abbreviation in part (b) will be used to represent the structure in part (a) throughout the remainder of this book.



(a) Node structure for List.



(b) Abbreviated diagram of the same structure as in (a).

Figure 21.25 Diagrams for the node structure of List.

Suppose myList and yourList are both variables of type List. The assignment statement

```
myList := yourList
```

is a record assignment. Therefore, every field of record yourList gets copied to the corresponding field of myList. But there is only one field in the record, namely head, which is a pointer. Therefore, the above record assignment is equivalent to the pointer assignment

```
myList.head := yourList.head.
```

A linked list class

The remainder of this chapter describes how a linked list can be implemented as a class. Recall from Chapter 9 that the two primary advantages of using a class instead of an ADT are the object-oriented features of *inheritance* and *class composition*. Chapters 23 and 24 describe these object-oriented features. In this chapter, there is no direct advantage to implementing the linked list as a class because those features of the class are not used.

Inheritance and class composition

The purpose for introducing the linked list as a class is to learn some of the details of how to program with objects. The culmination of our study of object-oriented programming is a design pattern known as the state pattern presented in Chapter 24. That chapter presents yet another implementation of the linked list based on an object-oriented property of inheritance known as polymorphism. The state pattern design technique is a modification of the class implementation of this chapter. So, you may view this class implementation as an opportunity to learn some more details of how to program with objects. It is a preliminary step in the direction of more advanced object-oriented programming. Figure 21.26 shows the interface of PboxLListObj, a linked list packaged as a class.

```
DEFINITION PboxLListObj;

  TYPE
    T = ARRAY 16 OF CHAR;
    List = RECORD
      (VAR lst: List) Clear, NEW;
      (IN lst: List) Display, NEW;
      (IN lst: List) GetElementN (n: INTEGER; OUT val: T), NEW;
      (VAR lst: List) InsertAtN (n: INTEGER; IN val: T), NEW;
      (IN lst: List) Length (): INTEGER, NEW;
      (VAR lst: List) RemoveN (n: INTEGER), NEW;
      (IN lst: List) Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW
    END;

END PboxLListObj.
```

Figure 21.26
The interface of the linked list class PboxLListObj.

Compare Figure 21.26 with Figure 7.18, page 139, which is the interface of PboxListADT, a list abstract data type. Because PboxListADT implements its list with an array it has a capacity, which the linked implementation does not have. This is a major advantage of a linked data structure compared to an array-based data structure. With an array, you must declare the maximum amount of storage you will need even if you do not use all the storage. With a linked structure, you need not commit to a maximum size. The data structure can grow to fit the data as long as storage is available in the heap. Heap storage is, in turn, limited only by the amount of physical storage available on your computer.

A major advantage of a linked implementation over an array implementation

The procedures for PboxLListObj in Figure 21.26 perform the same operations as the procedures for PboxListADT in Figure 7.18. Procedure Clear initializes a list to the empty list. Display displays the content of the list on the Log. GetElementN returns the element at position *n* in a list assuming that the first element is at position

0. InsertAtN provides a value and a location of where to insert the value into a list. Length returns the number of elements in a list. RemoveN supplies a position in a list and removes the element at that position from the list. Search supplies a search value and sets fnd to false if the value is not in the list. Otherwise it sets fnd to true and n to the position of the first occurrence of that value in the list. Figure 21.27 is the documentation for PboxLListObj.

TYPE **List**
The linked list class supplied by PboxLListObj.

TYPE **T**
The type of each element in the list, a string of at most 15 characters.

PROCEDURE (VAR lst: List) **Clear**
Post
List lst is initialized to the empty list.

PROCEDURE (IN lst: List) **Display**
Post
List lst is output to the Log, one element per line with each element preceded by its position.

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

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

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

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

Figure 21.27
The documentation of the linked list class PboxLListObj.

PROCEDURE (IN lst: List) **Search** (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN)
Post
If srchVal is in list lst, 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.

Figure 21.28 shows the dialog box of a program that uses the linked list class of Figure 21.26. It is identical to the dialog box of Figure 7.19 except that the number of items in each list (999) is not correct. It is your job to complete the implementation of the linked list class to display the correct number of items in the list.

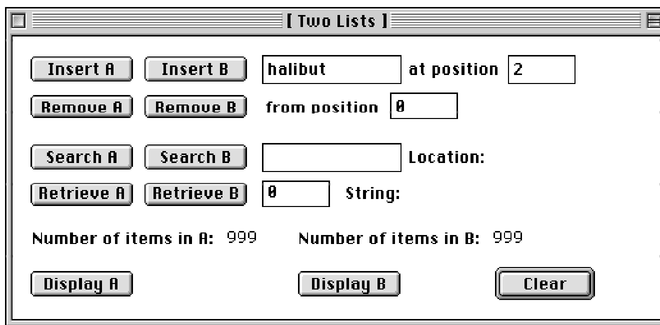


Figure 21.28 The dialog box for manipulating two lists.

Figure 21.29 shows the module that produces the dialog box of Figure 21.28. The interface in Figure 21.26 shows that two types are exported by the linked list class—List, which corresponds to the linked list, and T, which corresponds to the type of the value that is stored in each cell of the list. In this linked list, the type stored in each cell is a string of up to 15 characters. You can see this type PboxLListObj.T for the fields in the d record that correspond to the controls of the dialog box used for input and output of values with this type.

Figure 21.29 The program for the dialog box of Figure 21.28.

```
MODULE Pbox21E;
   IMPORT Dialog, PboxLListObj, PboxStrings;

   TYPE
      String32 = ARRAY 32 OF CHAR;

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

Figure 21.29
Continued.

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

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

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

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

PROCEDURE SearchForA*;
VAR
    found: BOOLEAN;
    position: INTEGER;
BEGIN
    listA.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 SearchForA;
```

Figure 21.29
Continued.

```
    PROCEDURE SearchForB*;
    VAR
        found: BOOLEAN;
        position: INTEGER;
    BEGIN
        listB.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 SearchForB;

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

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

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

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

    PROCEDURE ClearLists*;
    BEGIN
        listA.Clear; listB.Clear;
        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 Pbox21E.
```

The two lists processed by the module are the global lists declared as

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

This client module can have more than one data structure because the type of the data structure PboxLListObj.List is exported by the server module. The lists are global because they must persist between the clicks of the buttons in the dialog box.

To invoke a method for the linked list the client module uses the syntax appropriate for objects. For example, in procedure InsertAtA, listA is an object. The relevant method exported by the server module is InsertAtN, whose documentation specifies

PROCEDURE (VAR lst: List) **InsertAtN** (n: INTEGER; IN val: T)

The receiver is (VAR lst: List), which acts like one of the formal parameters, except that it is placed before the method name instead of after it along with the other formal parameters. The corresponding method call as shown in Figure 21.29 is

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

In the same way that d.insertPosition is the actual parameter that corresponds to formal parameter n, and d.insertT is the actual parameter that corresponds to formal parameter val, listA is the actual parameter that corresponds to formal parameter lst. Unlike the other actual parameters, the object listA comes before the name of the method, it is not enclosed by parentheses, and it is separated from the method name by a period.

Figure 21.30 is a partial implementation of the linked list class. Most of the methods are left as problems for the student. The statements in the procedures that are not completed are known as *stubs*. Their purpose is to allow the module to be compiled before all the procedures are completed. For example, the statement

The purpose of a stub

```
(* A problem for the student *)
RETURN 999
```

in method Length is a stub. The RETURN statement is necessary for the module PboxLListObj to compile. Placing stubs in some of the procedures allows the other procedures in the module to be compiled and tested. When you complete procedure Length you should delete its stub.

Figure 21.30
Implementation of the linked list class that is used in Figure 21.29.

```
MODULE  PboxLListObj;
    IMPORT StdLog;

    TYPE
        T* = ARRAY 16 OF CHAR;
        List* = RECORD
            head: POINTER TO Node
        END;
        Node = RECORD
            value: T;
            next: List
        END;

    PROCEDURE (VAR lst: List) Clear*, NEW;
    BEGIN
        lst.head := NIL
    END Clear;

    PROCEDURE (IN lst: List) Display*, NEW;
        VAR
            p: List;
            i: INTEGER;
    BEGIN
        i := 0;
        p := lst;
        WHILE p.head # NIL DO
            StdLog.Int(i); StdLog.String(" "); StdLog.String(p.head.value); StdLog.Ln;
            INC(i);
            p := p.head.next
        END
    END Display;

    PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
        VAR
            p: List;
            i: INTEGER;
    BEGIN
        ASSERT(0 <= n, 20);
        p := lst;
        FOR i := 1 TO n DO
            ASSERT(p.head # NIL, 21);
            p := p.head.next
        END;
        ASSERT(p.head # NIL, 21);
        val := p.head.value
    END GetElementN;
```

Figure 21.30
Continued.

```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;
        i: INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN (* Insert at beginning *)
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;

PROCEDURE (IN lst: List) Length* (): INTEGER, NEW;
BEGIN
    (* A problem for the student *)
    RETURN 999
END Length;

PROCEDURE (VAR lst: List) RemoveN* (n: INTEGER), NEW;
BEGIN
    (* A problem for the student *)
END RemoveN;

PROCEDURE (IN lst: List) Search* (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW;
BEGIN
    (* A problem for the student *)
    fnd := FALSE
END Search;

END PboxLListObj.
```

The declaration of the linked list from module PboxLListObj in Figure 21.30 is

```
List* = RECORD
   head: POINTER TO Node
END;
Node = RECORD
   value: T;
   next: List
END;
```

which is shown in Figure 21.25. Enclosing the head pointer within a record is necessary because the linked list is implemented as a class, and some of the methods for the class would be impossible to implement otherwise. Like the declaration for CList, however, the type of the next field of the Node is a List.

Why not structure the node for the class the same way as for CList? Because of restrictions that Component Pascal puts on the receiver of a method. The type of the receiver must be either

Restrictions on the receiver of a method

- a pointer to a record called by value (default),
- a record called by reference (VAR), or
- a record called by constant reference (IN).

It would be legal to declare the linked list class as follows

```
List* = POINTER TO Node;
Node = RECORD
   value: T;
   next: List
END;
```

which is similar to the way it is declared in Figure 21.18 for CList. The problem is with the implementation of some of the methods, such as procedure Clear. This procedure clears a list by setting the pointer to NIL. To change the value of the pointer requires the pointer be called by reference in procedure Clear, as Figure 21.30 shows. It would be impossible to call the pointer by reference in the receiver, because pointers are restricted to call by value in the receiver.

Method PboxLListObj.Clear

Method PboxLListObj.Clear works just like its counterpart PboxCListADT.Clear. The CList procedure sets

```
lst := NIL
```

while the corresponding List method sets

```
lst.head := NIL
```

In both cases, automatic garbage collection reclaims any inaccessible nodes resulting from the NULL assignment.

Method PboxLListObj.GetElementN

Method GetElementN takes as input an integer n and list lst, and retrieves the element val from the node at position n in lst. Figure 21.31(a) shows listA before execution of procedure GetElementN. Figure 21.31(b) shows the usual memory allocation

on the run-time stack when a call to listA.GetElementN(d.retrievePosition, d.retrieveT) executes assuming that the user has entered 1 for d.retrievePosition. Formal parameter lst is called by constant reference, and so gets a reference to listA. val is called by result, so it gets a reference to its actual parameter d.retrieveT, which is not shown in the figure.

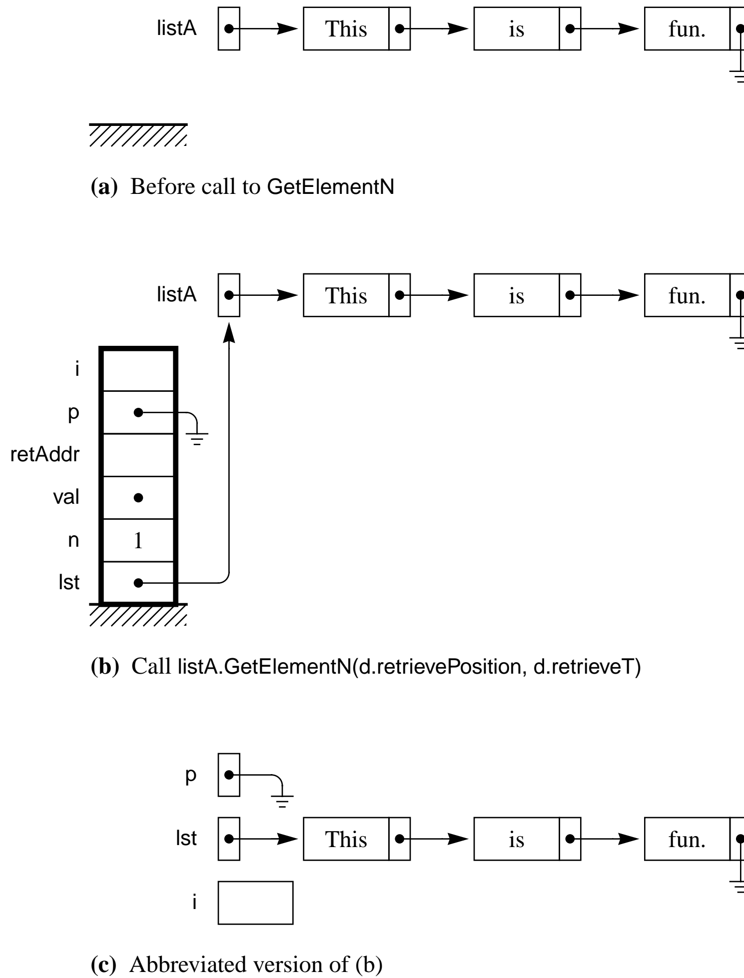


Figure 21.31 Memory allocation when method GetElementN is called.

To keep things simple, the following diagrams will frequently show only the relevant formal parameters and local variables as in Figure 21.31(c). Even though formal parameters and local variables are allocated on the run-time stack, the structure of the stack itself is not shown. The receiver lst is a record that contains the single link head. Rather than show the actual parameter, the figures will show the formal parameter lst, and label the pointer to the first node as lst.head as in Figure 21.31(c). Likewise, the label for the formal parameter p will be the pointer that is in the one field of the record, p.head.

The preconditions of procedure GetElementN as specified in the documentation are

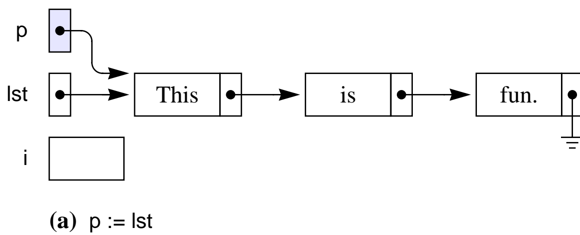
```
0 <= n    20
n < lst.Length()    21
```

The first statement in the implementation of the procedure

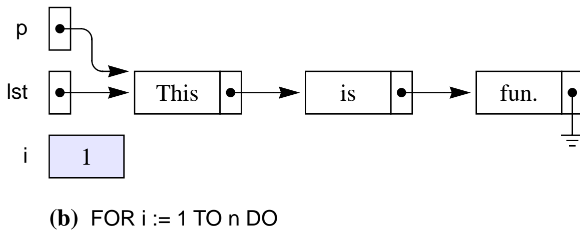
```
ASSERT(0 <= n, 20)
```

insures that n is greater than zero. If it is not, a trap will execute with the appropriate message that a precondition was violated and will display the identifying number 20.

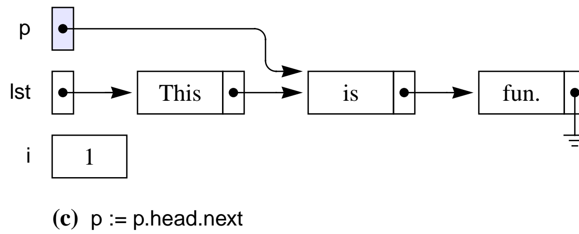
The technique for retrieving the element at position n is to initialize List variable p to list lst, and then advance it through the list n times. Figure 21.32 shows the sequence of steps to retrieve the element at position 1. The record assignment



(a) p := lst



(b) FOR i := 1 TO n DO



(c) p := p.head.next

Figure 21.32
A trace of procedure GetElementN.

```
p := lst
```

in Figure 21.32(a) is equivalent to

```
p.head := lst.head
```

and initializes p.head to point to the first element of the list. The first execution of

```
FOR i := 1 TO n DO
```

in Figure 21.32(b) initializes i to 1 and causes the body of the loop to execute. The first statement in the body of the loop is the assertion

```
ASSERT(p.head # NIL, 21)
```

which will trigger a trap if p.head equals NIL. If n is less than the length of the list, then p will eventually get the NIL value from the last node in the list, producing the trap. The ASSERT statement implements the precondition n < lst.Length(). The second statement in the body of the loop

```
p := p.head.next
```

in Figure 21.32(c) advances p.head through the linked list. In this example, p.head only advances once to get to the node at position 1, but in general it will advance n times to get to position n. The assertion preceding this statement guarantees that p.head is not NIL and, therefore, that p.head.value exists. The last statement in the procedure

```
val := p.head.value
```

gives the value part of the node pointed to by p.head to formal parameter val.

It is important to be aware of the types involved when dealing with linked structures. In the assignment statement

```
p := p.head.next
```

the relevant types are

- p is a record (which is also a List)
- p.head is a pointer to a Node
- p.head.next is a record (which is also a List)

So, the assignment is a record assignment, or, equivalently, a List assignment. Because the record contains a single field named head, which is a pointer, you could write the same assignment as

```
p.head := p.head.next.head
```

However, an assignment statement such as

```
p := p.head
```

would be illegal due to type conflict, because you would be trying to assign a pointer to a record.

Method PboxLListObj.InsertAtN

Method InsertAtN takes as input an integer n and a string value val, and inserts a new node with val into list lst at position n. Figure 21.33 is a trace of the procedure call to insert the word “such” at position 2. The figure uses the abbreviated style without showing the run-time stack for simplicity.

The first statement

```
ASSERT(0 <= n, 20)
```

corresponds directly to the precondition 0 <= n of the procedure. Unlike procedure GetElementN, which requires n to be less than the length of the list, InsertAtN permits n to be greater than or equal to the length of the list. If it is, the new node is appended to the end of the list.

The processing for the case of the empty list or when n has the value of zero is different from the case when the list has at least one node and n is greater than zero. Figure 21.33 shows a list with three elements and an insertion at position 2. A trace of the procedure for the other case is left as an exercise for the student.

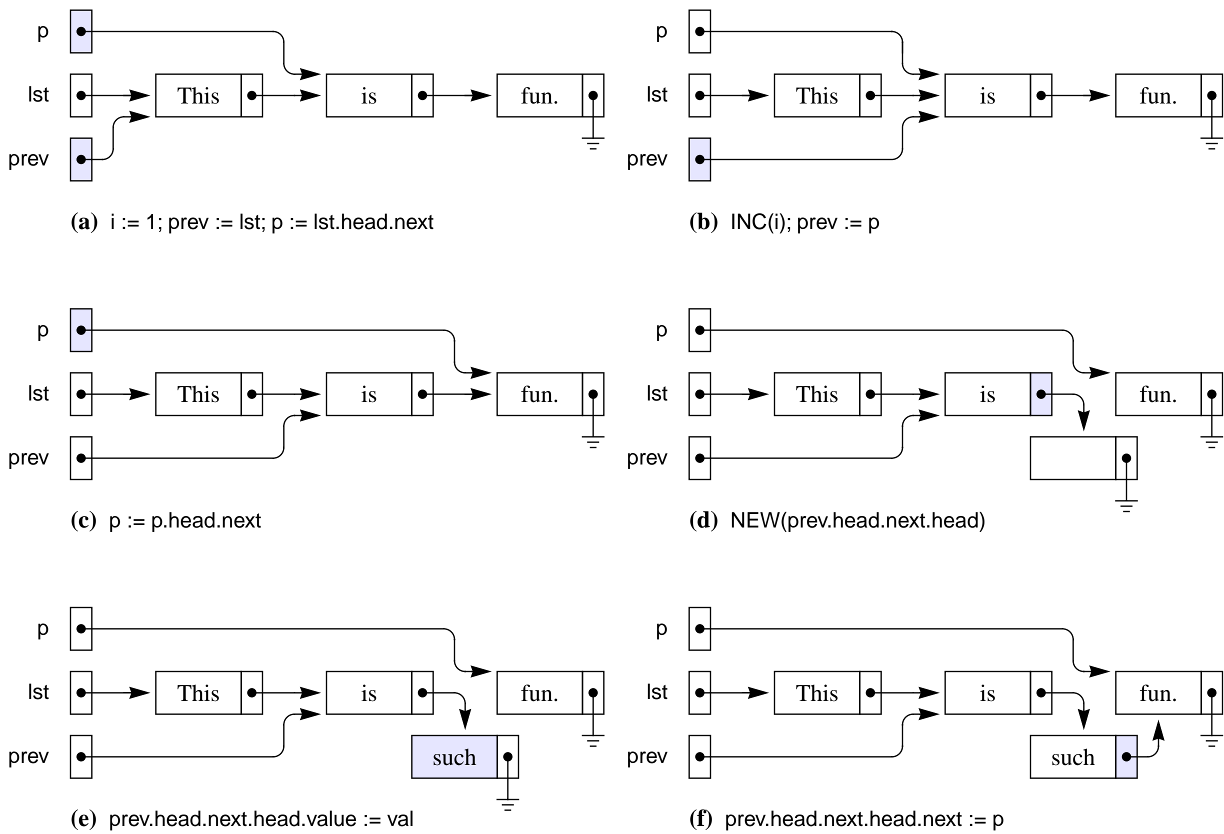


Figure 21.33 Execution of procedure InsertAtN to insert the word "such" at position 2.

Figure 21.33(a) shows the list after the first three statements in the ELSE part of the procedure execute. To keep the figure simple, storage for local variable i is not shown. The statement

```
prev := lst
```

makes pointer prev.head point to the same node to which lst.head points, namely the

node that contains “This”. The statement

```
p := lst.head.next
```

makes pointer p.head point to the same node to which lst.head.next.head points, namely the node that contains “is”. The program is guaranteed to not produce a trap from a NIL pointer reference with this assignment, because the IF part of the test must be false, which implies that lst.head is not NIL. Therefore, lst.head points to a node and the next part of that node exists. It is true that if the list consists of a single node that p.head would get NIL from this assignment, but such an assignment would not produce a trap.

These statements set up the loop invariant, which is that p.head points to the node at position i, and prev.head points to the previous node. Specifically, the value of i is 1, p.head points to the node at position 1, and prev.head points to the previous node at position 0. The body of the loop processes the variables by increasing i while maintaining the loop invariant.

Establish the loop invariant

The test at the beginning of the loop is

```
WHILE (i < n) & (p.head # NIL) DO
```

By De Morgan’s law the loop will terminate when (i >= n) OR (p.head = NIL). In fact, because the loop increments i only by one each time through the loop, the loop will terminate when (i = n) OR (p.head = NIL).

Figure 21.33(b) and (c) show the effect of one execution of the loop body. The statements

```
INC(i);
prev := p
```

increment i by one and advance prev.head to point to the next node. Then, the assignment statement

```
p := p.head.next
```

makes p.head point to the next node in the list. As with the initialization statements before the loop, you are guaranteed that this assignment will not produce a NIL reference trap. This time it is the WHILE test that guarantees no trap because you can execute the body only if (p.head # NIL). So, you know that p.head points to a node and that p.head.next exists.

Figure 21.33(c) shows that the loop invariant is maintained. Specifically, the value of i is 2, p.head points to the node at position 2, and prev.head points to the previous node at position 1. At this time the loop terminates because i equals n. Because the loop body maintains the invariant, you know that when the loop terminates that (i = n) OR (p.head = NIL) and that p.head points to the node at position i, and prev.head points to the previous node. The remaining task is to allocate a new node and splice it into the linked list between the nodes pointed to by prev.head and p.head.

Reestablish the loop invariant

Figure 21.33(d) shows the effect of the statement

```
NEW(prev.head.next.head);
```

to allocate a new node from the heap. Figure 21.33(e) shows how

```
prev.head.next.head.value := val;
```

sets the value part of the node to the string entered by the user. The next statement illustrated in Figure 21.33(f)

```
prev.head.next.head.next := p
```

links the new node to the remainder of the list.

With these pointer manipulations you can see why the algorithm needs to keep track of the previous node prev and the next node p. The link of prev must be changed to point to the newly allocated node. And the link of the newly allocated node must be set to point to the node to which p.head points.

Method PboxLListObj.RemoveN

Figure 21.34 shows the sequence of events that must transpire to implement RemoveN. In Figure 21.34(a) the algorithm initializes prev.head to point to the node at position 0 in the list and p.head to point to the node at position 1. A loop is required to search for the node that contains the search word. The body of the loop contains statements to advance p and prev.

```
prev := p;
p := p.head.next
```

When p.head points to the node to be removed, prev.head points to the node before it, as shown in Figure 21.34(b). Figure 21.34(c) shows how to remove the node from the list. The algorithm merely changes the link part of the preceding node to point to the node after the one to be deleted. Implementation of the procedure to delete a node is a problem for the student at the end of this chapter.

Design trade-offs with linked lists

What are the advantages of using linked lists with pointers? After all, Figure 17.8 shows all the operations on the list data structure implemented with arrays. The linked implementation of lists has two advantages over the array implementation.

Flexibility of storage allocation

The first advantage is the flexibility of dynamic storage allocation. With arrays you must allocate enough memory for the maximum problem size you expect to encounter. With dynamic storage allocation you always have the entire heap from which to allocate another element.

This advantage is particularly important in problems with many lists. Suppose you have three lists—a, b, and c. One time when you run the program, list a may have 10,000 elements, and lists b and c only a few. The next time, list b may have 10,000, and lists a and c only a few. If you implement the lists with arrays, you would need to allocate 10,000 elements for a, b, and c, for a total of 30,000 elements, to account for the possibility of any of the three lists having a maximum of 10,000 elements. Your computer would need storage for 30,000 elements to run the program.

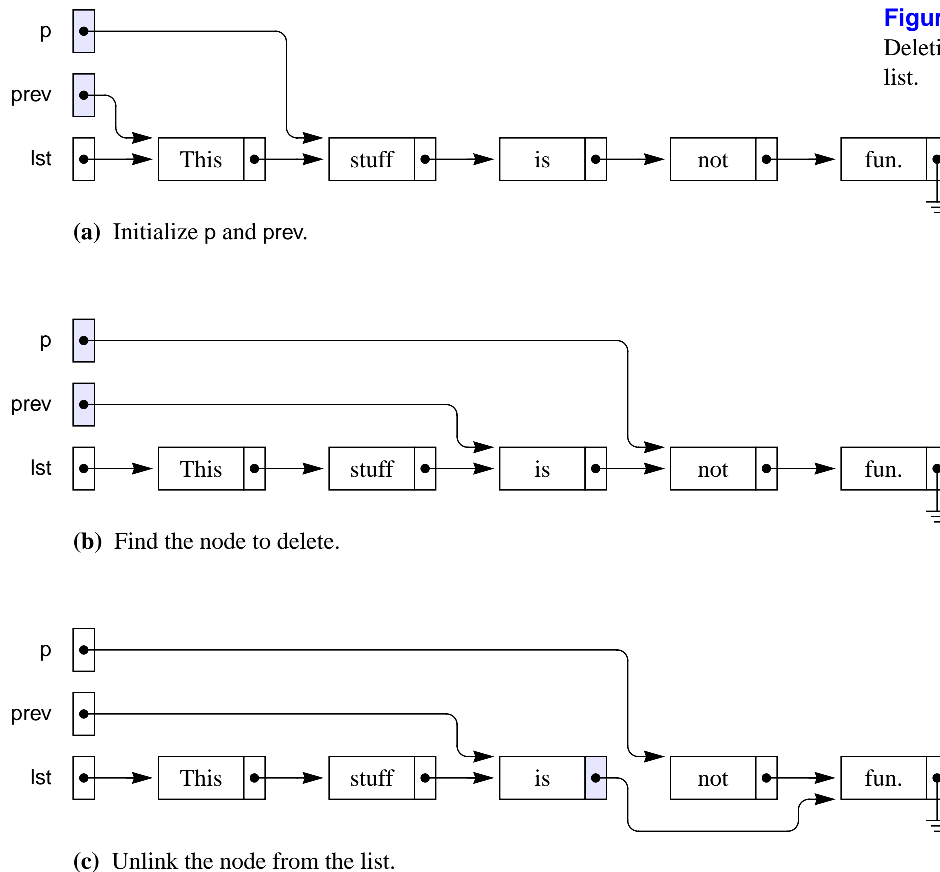


Figure 21.34
Deleting a node in a linked list.

But if you implement the lists with dynamic storage allocation, you do not need to declare the maximum size of each list. Your computer would need storage for only a few more than 10,000 elements regardless of which list used most of them. Using pointers, different lists use storage from the same heap. The net result is that the linked list implementation can require less storage because of the flexibility of dynamic storage allocation from the heap.

Speed of insertions and deletions

The second advantage of a linked implementation is the speed of insertions and deletions in long lists. To delete item 10 in a 100-item array, a, requires you to shift a[11] to a[10], a[12] to a[11], a[13] to a[12], and so on. But to delete item 10 in a 100-item linked list only requires you to change the link field of item 9 to point to item 11. You need not make any shifts.

Sequential access

The disadvantage of linked lists is their sequential nature. The only way to access an element in the middle of a list is to start at the beginning and sequentially advance a pointer through all the intermediate nodes. Arrays, however, are random access data structures. That is, you can access the element at position i of array a directly by subscripting a[i]. Direct access with subscripting is what allows for efficient searching and sorting. For example, you cannot do a binary search of a linked list because you cannot access the middle of the list in one step the way you can with

an array.

So, whether to use an array implementation or a linked implementation of a list depends on the use to which it will be put. If speed of insertions and deletions is important and if flexibility of memory allocation is important use a linked implementation. If speed of searching and sorting is important use an array implementation. Such considerations arise frequently in the study of data structures. You typically have more than one implementation at your disposal and the implementation you choose will involve trade-offs that require you to take into account the use to which the data structure will be put.

Trade-offs with data structures

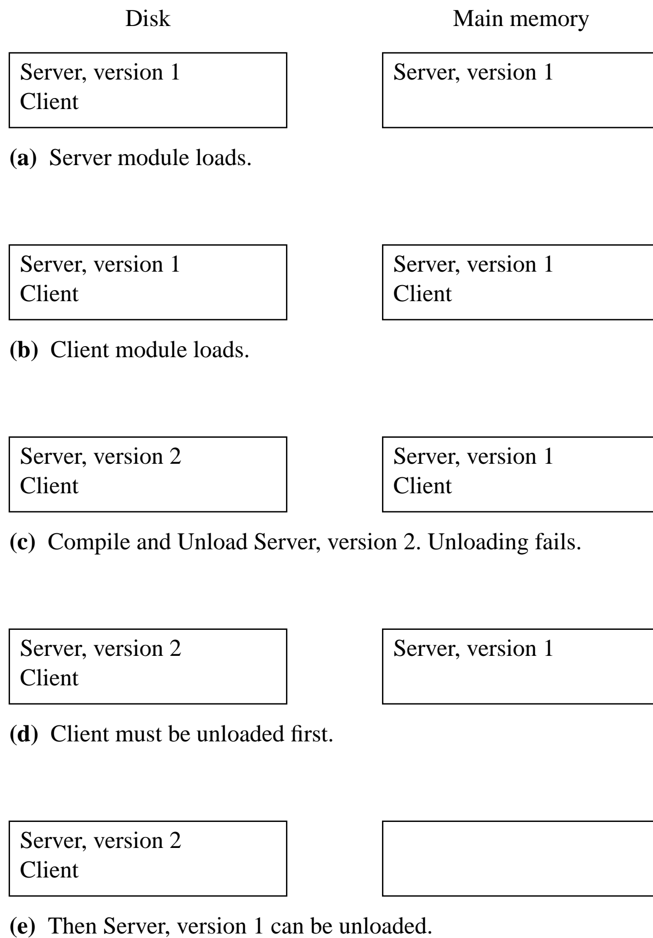


Figure 21.35
Developing a server module.

Unloading

The problems at the end of this chapter require you to program a server module that is imported by a client module. Usually, the client module will be written or modi-

fied only once, and you will spend most of your program development effort on the server module. Because the BlackBox framework is based on dynamic linking and loading, you must take care with the unloading process. Suppose you write some code in a server module, call it version 1, and test it with the client module. Figure 21.35 shows a likely scenario.

Figure 21.35(a) and (b) show that before a client module can be loaded from disk to main memory, all modules that it imports, that is, the server modules, must be loaded first. The framework cannot allow a module to be loaded without the modules that it uses (that is, imports) to also be loaded. Therefore, when you test a server module, the server loads first followed by the client that uses it.

Figure 21.35(c) shows what happens if you make a change to your server module, call it version 2, and select Dev→Compile And Unload. The compile may be successful, but the unload will fail because the system cannot unload the server module while the client is still in main memory. To unload a server you must first unload all the clients that import it.

How to unload a module

There are several ways to unload a module. If the focus window contains the source code for the module you want to unload, you can simply select Dev→Unload and that module will be unloaded. Alternatively, you can type the name of the module in the Log or in some other text window and highlight it with your mouse. Then select Dev→Unload Module List.

If you ever want to see which modules are currently loaded you can select Info→Loaded Modules. A window will appear that contains a list of all the currently loaded modules, the number of bytes each one occupies in main memory, the number of clients for each server module, and the date and time of the compile and of the load. Because the list of modules in the window has every client listed before its servers, you can even highlight the list in the window in order to execute Dev→Unload Module List.

Exercises

1. Suppose the variable declaration part of a Component Pascal program declares a, b, c, and d to each be a pointer to a record as in Figure 21.3. What does each of the following code fragments output to the Log?

(a)
```
NEW(a);
NEW(b);
a.i := 5;
b.i := 6;
a := b;
StdLog.Int(a.i);StdLog.Ln;
StdLog.Int(b.i);StdLog.Ln;
```

(b)
```
NEW(a);
a.i := 7;
NEW(b);
b.i := 2;
c := a;
d := b;
a := b;
StdLog.Int(a.i);StdLog.Ln;
StdLog.Int(b.i);StdLog.Ln;
StdLog.Int(c.i);StdLog.Ln;
StdLog.Int(d.i);StdLog.Ln;
```

(c)
```
NEW(a);
a.i := 9;
b := a;
a.i := b.i + a.i;
StdLog.Int(a.i);StdLog.Ln;
StdLog.Int(b.i);StdLog.Ln;
```

(d)
```
NEW(a);
a.i := 2;
NEW(b);
b.i := 3;
c := a;
a := b;
b := c;
StdLog.Int(a.i);StdLog.Ln;
StdLog.Int(b.i);StdLog.Ln;
StdLog.Int(c.i);StdLog.Ln;
```

2. Suppose p, q, r, and last have type List as in Figure 21.6 and have the values in Figure 21.36. Draw the figure after execution of the following instructions.

(a)
q := r;
p := last

(b)
q := q.next;
p := p.next

(c)
q := q.next.next;
p := p.next.next

(d)
NEW(q)
q.next := r;
q.value := 5.0;
r := q;
q := NIL

(e)
r.next := q.next;
q := NIL

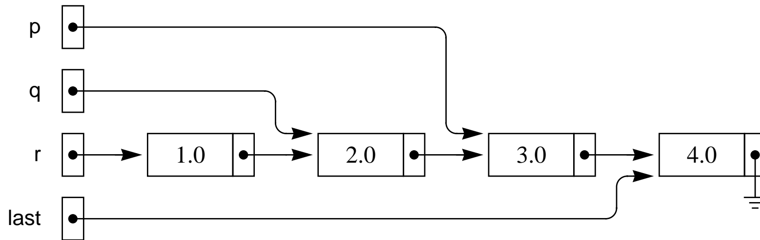


Figure 21.36
The linked list for Exercises 2, 3, and 6.

3. Suppose p, q, r, and last have type List as in Figure 21.6 and have the values in Figure 21.36. State whether each of the following instructions produces a NIL pointer reference error. For those that do, explain why.

(a)
p := NIL;
p := p.next

(b)
p := p.next;
p := p.next

(c)
last := last.next;
last := last.next

4. In computer science terminology, **(a)** what is the meaning of the word static? **(b)** What is the meaning of the word dynamic?

5. Assuming the declarations of the types and variables in the section Class assignments, page 474, state whether each of the following assignments will or will not compile correctly. For those that will not compile correctly, explain why not.

(a) alphaPtr := gammaPtr
(b) alphaPtr := gammaPtr (GammaPtr)
(c) alphaPtr := gammaPtr (AlphaPtr)
(d) gammaPtr := alphaPtr
(e) gammaPtr := alphaPtr (GammaPtr)
(f) gammaPtr := alphaPtr (AlphaPtr)
(g) betaPtr := gammaPtr
(h) betaPtr := gammaPtr (BetaPtr)
(i) deltaPtr := gammaPtr
(j) gammaPtr := deltaPtr (GammaPtr)

6. Suppose p, q, r, and last have type List as in Figure 21.30 and have the values in Figure 21.36 assuming the abbreviation of Figure 21.25(b). Draw the figure after execution of the following instructions.

(a)
```
q.head := r.head;
p := last
```

(b)
```
q := q.head.next;
p := p.head.next
```

(c)
```
q := q.head.next.head.next;
p := p.head.next.head.next
```

(d)
```
NEW(q.head)
q.head.next := r;
q.head.value := 5.0;
r := q;
q.head := NIL
```

(e)
```
r.head.next := q.head.next;
q.head := NIL
```

7. Suppose p and q have type List as in Figure 21.30. State whether each of the following statements will or will not compile correctly. For those that will not compile correctly, explain why not.

(a) p := q **(b)** p.head := q.head **(c)** p^.head := q^.head
(d) p := q.head **(e)** p.next := q.next **(f)** p.head.next := q
(g) p.head^.next := q **(h)** NEW(p) **(i)** NEW(p.head)

8. **(a)** What types are receivers restricted to in Component Pascal? **(b)** For each of the types in (a), which parameter calling mechanisms are allowed?

Problems

9. Modify the PboxStackADT to implement the stack with a linked list instead of an array. The interface for your modified stack should be similar to the interface in Figure 7.8 except that it should not export a capacity, because you will be able to allocate as many nodes as you wish from the heap. Define a stack to be a pointer to a node as follows.

```
Stack* = POINTER TO Node;
Node = RECORD
    value: REAL;
    next: Stack
END;
```

Your interface should have the following specifications.

```
PROCEDURE Clear (VAR s: Stack);
PROCEDURE NumItems (s: Stack): INTEGER;
PROCEDURE Pop (VAR s: Stack; OUT x: REAL);
PROCEDURE Push (VAR s: Stack; x: REAL);
```

In each of these procedures, s will point to the node at the top of the stack. The next field of the node will point to the node that is second from the top, and so on. Test your implementation with the program of Figure 7.10.

10. It is possible for the user to generate a trap when she executes module Pbox21E, Figure 21.29. Rewrite the module to make it bulletproof so that no trap will be generated regardless of the user input. If an input value would generate a trap, program the module to do nothing.

11. Implement procedure SetTotal in Figure 21.17 to compute the total price for all the books in the circular list. The empty list is a special case, for which the total is 0.00. If there is at least one book in the list you should initialize a local variable, say p, of type CList to cList. Leaving cList unchanged, progress around the circular list with p using GoNext(p) to advance through the circular list until p equals cList. At each position in the circular list accumulate the total from the price of the current book. You will need to use a local variable book of type Book as in PROCEDURE Next on page 483 to access the price field of the book record.

12. Implement the Previous button of the dialog box in Figure 21.15 by completing procedure Previous in Figure 21.17. Do not modify PboxCListADT.

13. Implement the Delete button of the dialog box in Figure 21.15 by completing procedure Delete in Figure 21.17. Add a procedure to PboxCListADT also named Delete, which is called by procedure Delete in Figure 21.17. The precondition for procedure Delete is that the list is not empty. The postcondition is for the new current position to be the old previous one. Do not change the declaration for Node in PboxCListADT.

14. Implement the Previous button of the dialog box in Figure 21.15 by completing procedure Previous in Figure 21.17. Add a prev link to the PboxCListADT node as described in the section, A circular doubly-linked list, page 488. Add a procedure to PboxCListADT named GoPrev, which is called by procedure Previous. The precondition for GoPrev is that the list is not empty.

15. Implement the Delete button of the dialog box in Figure 21.15 by completing procedure Delete in Figure 21.17. Do this problem only after you have completed Problem 14 for the doubly-linked list. Add a procedure to PboxCListADT also named Delete, which is called by procedure Delete in Figure 21.17. The precondition for procedure Delete is that the list is not empty. The postcondition is for the new current position to be the old previous one as shown in Figure 21.23.

16. This problem is for you to complete the following methods for the linked list class in module PboxLListObj. Test your procedures with the program in Figure 21.29 using the dialog box of Figure 21.28. For each of the following procedures, implement any preconditions with the appropriate ASSERT or HALT procedure. Do not use recursion in any of your implementations.

(a) (VAR lst: List) RemoveN (n: INTEGER), NEW
(b) (IN lst: List) Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW
(c) (IN lst: List) Length (): INTEGER, NEW

17. Change the implementation of procedure Display in PboxLListObj as follows.

```
PROCEDURE (IN lst: List) Display*, NEW;
BEGIN
    lst.DisplayRecursive(0)
END Display;
```

Then, define DisplayRecursive as

PROCEDURE (IN lst: List) DisplayRecursive (n: INTEGER), NEW

which recursively outputs a list starting with its first element numbered n. Do not use a loop.

18. Work Problem 17, but output the list in reverse order.

19. Work Problem 16(b) to write the method Search for the linked list class, but do it recursively without a loop. Define SearchN as

PROCEDURE (IN lst: List) SearchN (IN srchVal: T; VAR n: INTEGER; OUT fnd: BOOLEAN), NEW;

whose signature differs from that of Search only by n being called by reference (VAR) instead of called by result (OUT). The programmer of the client in Figure 21.29 must see the same interface for your recursive version of PboxLListObj and must not need to initialize the value of n before it calls the server method. In the implementation of Search, initialize n to 0 then call SearchN, which assumes that the initial value of n is defined. If lst is not empty and the value field of its first node does not equal srchVal, then SearchN must call itself recursively. If srchVal is at position n in the next list of the first node of lst, then it is at position n + 1 in lst. In that case, SearchN must increment n.

20. Work Problem 16(c) to write the method Length for the linked list class, but do it recursively without a loop. The base case is for the empty list.

21. This problem requires you to add the following methods to module PboxLListObj in Figure 21.30. Test your procedures by importing them into the module of Figure 21.29. Augment the dialog box of Figure 21.28 to test the procedures.

(a) PROCEDURE (VAR lst: List) **Copy*** (listB: List), NEW
Create a new list, lst, which is a copy of listB. You must allocate new copies of all the nodes of listB. You are not allowed to simply set lst.head to point to the same head node that listB.head points to.

(b) PROCEDURE (VAR lst: List) **Append*** (listB: List), NEW
Append a copy of ListB to the end of ListA. You must allocate new copies of all the nodes of listB. You are not allowed to simply set the head field of the last node of lst to point to the same head node that listB.head points to.

(c) PROCEDURE (VAR lst: List) **Merge*** (listB: List), NEW
Merge a copy of listB with lst starting with the first word in lst. For example, if lst is the list

Now the for

and listB is the list

is time action

then after the procedure is called, lst should be the list

```
Now is the time for action
```

Be sure your procedure works correctly if `lst` has more words than `listB` or if `listB` has more words than `lst`. You must allocate new copies of all the nodes of `listB`. You are not allowed to alter the original nodes of `listB`.