Pascal Pointers and Dynamic Variables Demonstration Package

Dr. Jeffrey L. Popyack
Assistant Professor
Department of Mathematics
and Computer Science
Drexel University
Philadelphia, PA 19104

(215) 895-2668

1. Introduction:

This package is intended to provide users with a mechanism for learning to use pointers and dynamic variables in Pascal. The concept arose from the great deal of difficulty students have exhibited in learning this material in programming courses. This material is fundamental for further computer science work, yet often is not learned by students, partially because it is easier for the programmer to find an alternative way to implement his program than it is to learn the more elegant (and often more efficient) methods associated with dynamic variables.

The package is designed as follows: The user is supplied with a display which shows an initial data structure. The display is similar to that used in most Pascal textbooks. The user is allowed to enter commands from a subset of Pascal commands which use pointer variables. As each command is entered, the display is updated to demonstrate the effect of the command on the given structure. Syntax and semantic errors are detected by the interpreter, with appropriate diagnostics. Some on-line help is also provided. A set of lessons is available, each of which provides some initial structure, and asks the user to alter the structure in some way. The user is not required to perform the lessons, however; this package is designed for use as a general tool which will correctly interpret and display any sequence of pointer-type commands. This package may therefore be used by an instructor for classroom demonstration. The instructor may also use the package to create additional lessons.

The remainder of this report is organized as follows: Section 2 contains an explanation of pointers and dynamic variables in Pascal, and may be bypassed by the reader who is already familiar with this material. Section 3 contains instructions for using Pointer Demo and Section 4 shows a sample session.

2. About Pascal's Pointer Variables:

Persons familiar with the concept of dynamic variables and pointers in Pascal may proceed to Section 3.

Pascal allows the use of variable types that do not exist in some other languages, such as BASIC and FORTRAN. The user is also allowed to create new types for use in his program. One popular mechanism is the **record** structure, which allows a variable to have several components, each of which may have its own type. For instance, the declaration

```
type
     StockHolder =  record
                         name   : string[20]   ;
                         phone  : integer      ;
                         shares : integer      ;
                    end { card } ;

var
     President : StockHolder ;
     Client : array[1..1000] of StockHolder ;
```

cr
eates a type called "StockHolder", which means that any variable of this type will have three fields, containing a 20-character (or less) name, a phone number, and a number of shares held by that person. We create a variable named "President", of type "StockHolder", and refer to the various fields of this variable as "President.name", "President.phone", and "President.shares". In a like manner, we create an array, "Client", which has 1000 entries of type "StockHolder". The fields of the variable "Client[i]" are therefore referred to as "Client[i].name", etc., for any i between 1 and 1000.

Often it is difficult to judge how large the dimensions of an array need to be when the program is written. In our example, if there are only 25 clients, the remaining 975 elements of "Client" are unnecessary. Likewise, if there are 1002, 10000, or 25000 clients, the array is not large enough. A desirable feature would be the ability to leave the dimension of "Client" unspecified, and allow it to grow to whatever size is necessary when the program is run. Such a feature does not exist in Pascal, but may be emulated by using dynamic variables.

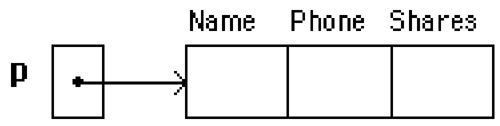
Before we may use dynamic variables, we must understand the concept of a pointer variable. A pointer variable, p, contains the address of another variable, and we say p points to that variable. We refer to this variable as p_ , which means "the variable pointed to by p". Now, suppose that, in addition to the previous type declarations, we add the declarations

```
type
      SHPtr = _StockHolder ;

var
     p : SHPtr ;
```

This declares a new type, called "SHPtr", which is a pointer to a variable of type StockHolder. Any variable of type SHPtr (for instance, p) can be assigned a value which is a pointer to a variable of type StockHolder. The variable that p points to is referred to as p_ , and has fields p_.name, p_.phone, and p_.shares.

Let us now suppose that we wish to assign p_ the values "Fred Whitman", "8675309", "23". We cannot do this yet, because the variable p_ does not exist. Although we have declared p as an SHPtr, there is no corresponding variable p_. There is only p, which can point to such a variable if it exists. In order to create such a variable *while the program is executing,* we execute the command "**new(p)**". The effect of this command is that a new variable of type SHPtr has been created, and p points to it. The variable does not have a name (such as "President", or "Client[5]"), but is referred to as p_.



W

e may now perform the assignments

```
     p_.name := 'Fred Whitman' ;
     p_.phone := 8675309 ;
     p_.shares := 23 ;
```

Because p_ did not exist before program execution began, but was created in the middle of execution, it is called a dynamic variable. The usefulness of such variables has not yet been made clear; It may seem that, in order to create a list of 1000 elements, we will need an array of 1000 SHPtr variables, performing new() 1000 times. If this were true, certainly dynamic variables would be of little use -- however, we see that it is also possible to create new SHPtr variables dynamically:

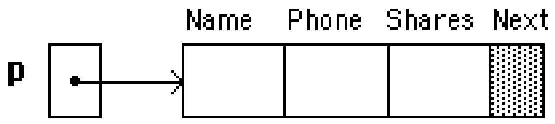
Let us revise the type "StockHolder":

```
type
     SHPtr = ^StockHolder ;
     StockHolder =   record
                         name   : string[20]   ;
                         phone  : integer      ;
```

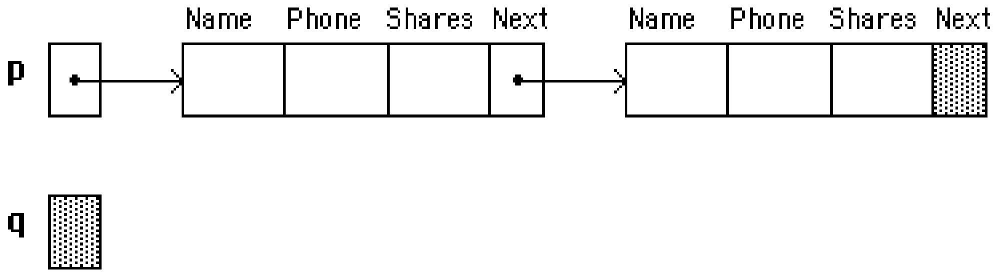
```
                    shares : integer    ;
                    next   : SHPtr      ;
                end { card } ;

    var
        p,q : SHPtr ;
```

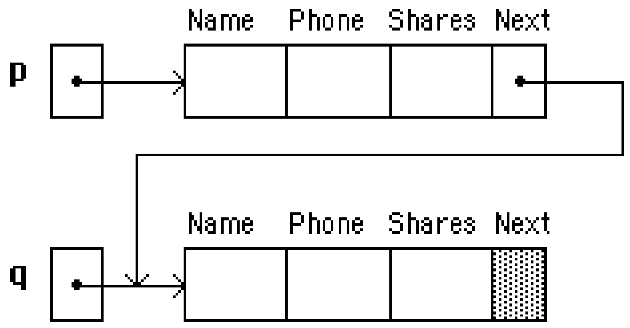
We note that only two variables, p and q, have been declared. Although the StockHolder type has been defined, there are no variables of that type in existence as our program begins. Notice, however, that we can create such a variable during execution simply by beginning with **new(p)**. What we now have is a variable p_ that contains all of the necessary StockHolder fields, plus another field, p_.next, which is a SHPtr.



This means that we can create a second variable with the command **new(p_.next)**. This yields the following structure:



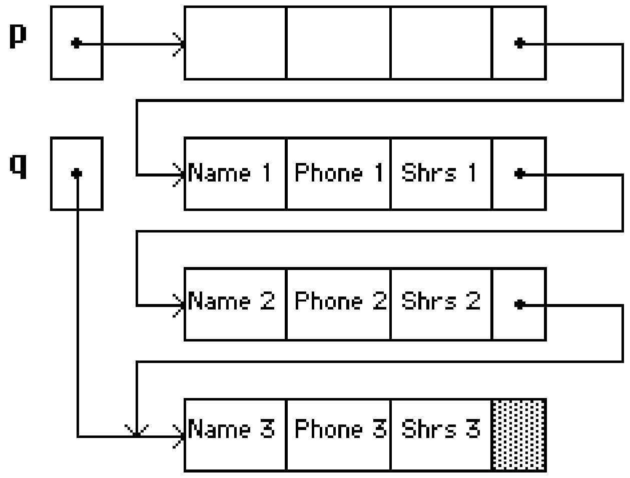
Alternatively, we could have performed the commands **new(q) ; p_.next:=q** to arrive at the same configuration. This is true because **new(q)** creates a new variable, pointed to by q, which means q holds the address of q_. Since p_.next is also a SHPtr, we see that p_.next may also hold the address of a StockHolder. Thus, by assigning **p_.next:=q** , we simply make p_.next point to q_. (If the reader finds this confusing, please consider this as justification for the need for the Pointer Demo package).



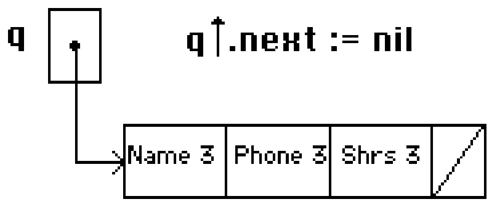
Alternate Method

We see that we can create a list of arbitrary size by performing the following section of code:

```
new(p) ;
q := p ;
while not eof do
   begin
       new(q_.next) ;
       q := q_.next ;
       readln(q_.name,q_.phone,q_.shares) ;
   end { while } ;
```



We note that the "next" field of the last item in the list has never been assigned a value. We would prefer to have a value in this field which will indicate that it points to no variable. (As always, attempts to access an uninitialized variable may cause one's program to crash. Thus, having an unassigned "next" field may lead to trouble at some point in the program.) For this reason, a special Pascal constant, **nil**, exists, which may be assigned to a pointer variable to indicate that it has no value. Thus, when the loop is completed, we may assign **q_.next := nil**.



The **nil** constant is very useful -- it signifies the end of the list. We may traverse the list until its end, by repeatedly checking for a pointer with value **nil**:

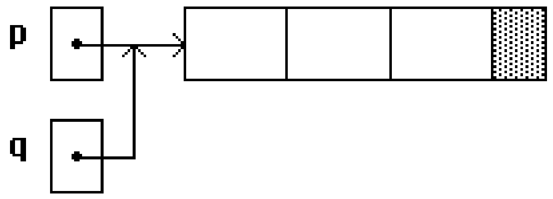
```
q := p ;
while q<>nil do
    q := q_.next ;
```

One further point is of note: We see that at the end of input, we have a linked list of StockHolders, where p points to the beginning of the list. Actually, p points to an unused node, which is then followed by the real list. If we want to bypass this node, we may add the command p:=p^.next. Thus, p will point to the real list of StockHolders. This causes the node originally pointed to by p to hang in limbo -- no SHPtr points to it, and so we may wish to remove this variable altogether. For this reason, a **dispose** command also exists which destroys a variable created by the **new** command. Essentially, dispose(p) removes the variable pointed to by p. Thus, instead of simply performing **p:=p_.next**, we should perform **q:=p; p:=p_.next; dispose(q)** . We see that repeated application of this process can be used to remove the entire list from memory, if desired. This can be very useful for programs requiring the use of many long lists, which, due to space limitations, cannot all exist in main memory at the same time. In such cases, it often occurs that some of the lists are only needed during certain phases of program execution; thus, by creating one list dynamically, using it where needed, then destroying it, we are able to create the second list without exceeding memory restrictions.

2.1 A detail not covered by the Pascal Standard

According to the rules discussed previously, what should happen when the following commands are executed?

```
new(p) ;
q := p ;
dispose(q) ;
```



According to the description of **dispose**, this would cause the variable q_ (which also is p_) to be destroyed. Yet, what is the value of p, now that p_ has been destroyed? Does it point into nothingness? Is it **nil**? Is it unassigned? This is a difficult philosophical question, which is made more difficult by the fact that it is not specified in the Pascal standard. It is unreasonable for us to expect that p will be made **nil** automatically -- this would require q_ to 'know' which variables point at it, and there could be thousands of such variables. Similarly, it would be very wasteful to have to locate all variables pointing to q_, then assign them **nil**. The best we can hope for is that p will be undefined. But, if this is the case, what does a reference to p mean? Should it cause an error, or perhaps should it simply allow the transfer of an address that is meaningless?

Because of the difficulties associated with this problem, system designers have adopted different approaches to handling it. In some cases, nothing extra is done, with the effect that p still contains the address it contained before, and thus p_ may be used as though nothing ever happened, as long as the space it occupies is not reassigned with another **new** statement. If that space is reassigned, p may automatically point to it, or may point to the wrong part of it, in which case the program will crash. (This is the technique used by PRIME Pascal). In other cases, dispose may not be used on a variable which is pointed to by more than one pointer. This eliminates the problem, although implementation of this procedure requires more work, and the program will still crash. (This is the technique used by MacPascal). In still other cases, the dispose command is never actually implemented, and is simply treated as a placebo! (This is how it is treated by Lisa Pascal).

For the purposes of providing a clear demo, the Pointer Demo Package uses the approach taken by MacPascal -- a variable may only be disposed if it is pointed to by exactly one variable. We note that it is probably a good idea for programmers to avoid the circumstance of trying to dispose of a variable with multiple pointers anyway, as the outcome is unpredictable, and the procedure is error-prone.

3. Using the Pointer Demo package:

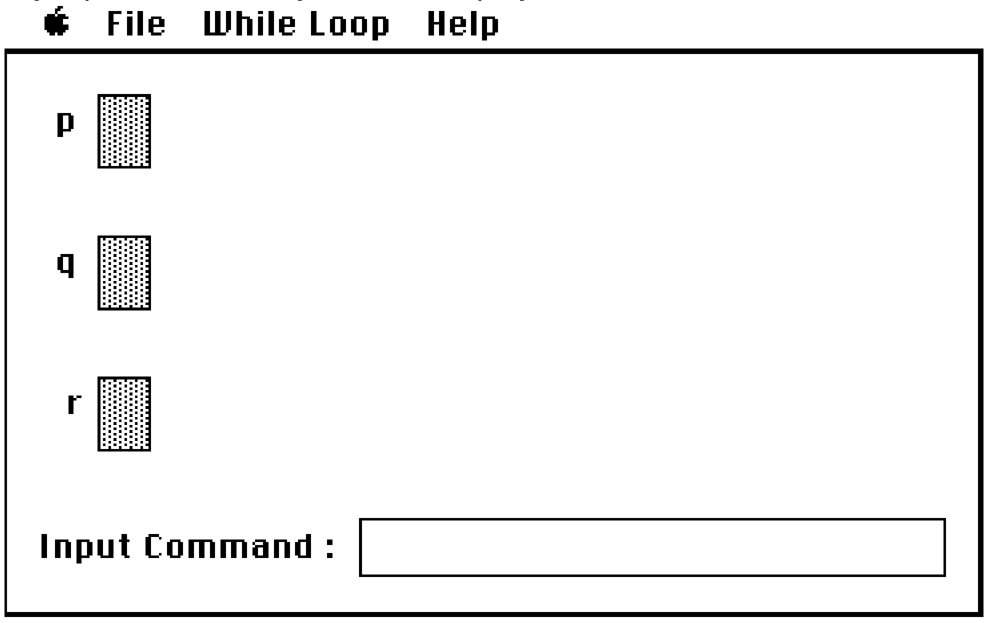
The variables p,q, and r are defined by

```
type NodePtr = ^Node;      Node = record
                                      field : char ;
                                      next  : NodePtr ;
                                  end ;
var p,q,r : NodePtr ;
```

The commands available to you are of the form

```
new(<NodePtr>)
dispose(<NodePtr>)
<NodePtr> := <NodePtr>
<NodePtr> := nil
<Node>^.field := <Node>^.field
<Node>^.field := '<char>'
<Node> := <Node>
```

Upon opening the application, the user is presented with a display depicting the three unassigned variables, p, q, and r. (Unassigned pointer variables are always shown as grey). The prompt **Enter Command** and a text box are visible. Any input from the keyboard is displayed in the command box.



To execute the command, the user enters a "Return". If there are syntax errors in the command, an error message is displayed. If there are no syntax errors, but the command refers to an invalid variable (such as p_.next, when p is **nil** or unassigned), an error message will be displayed. If there are no errors, the command will be executed, and the corresponding action will be depicted on the display.

The effects of various command types are explained below:

new(p) A node appears, with p pointing to it. The fields of p_ are unassigned, as is depicted by the color grey. Note that the field p_.next is unassigned, not **nil**. This is an important distinction. If no more space is available, an error message is displayed.

dispose(p) If p_ exists, and is not pointed to by any other pointer variable, p_ will be removed, and p will become unassigned. If some other variable points to p_, an error message is displayed. In some versions of Pascal, disposing p_ is handled differently when other pointers exist to p_. (See Section 2.1 for an explanation).

p := nil p is assigned the value nil, which is designated on the display by a diagonal line through p.

p := q If q_ exists, or q is **nil**, p is assigned the appropriate value. If q is unassigned, an error occurs. If p_ exists before the command is executed, the pointer is lost. If p_ is not pointed to by any other variables, it becomes a dangling node: there is no way to access this node.

p := p_.next If p is nil or unassigned, (and therefore, p_ does not exist), an error occurs. Otherwise, execution is identical to the **p:=q** case above.

p_.next:=q If p is nil or unassigned, an error occurs. Otherwise, same as above.

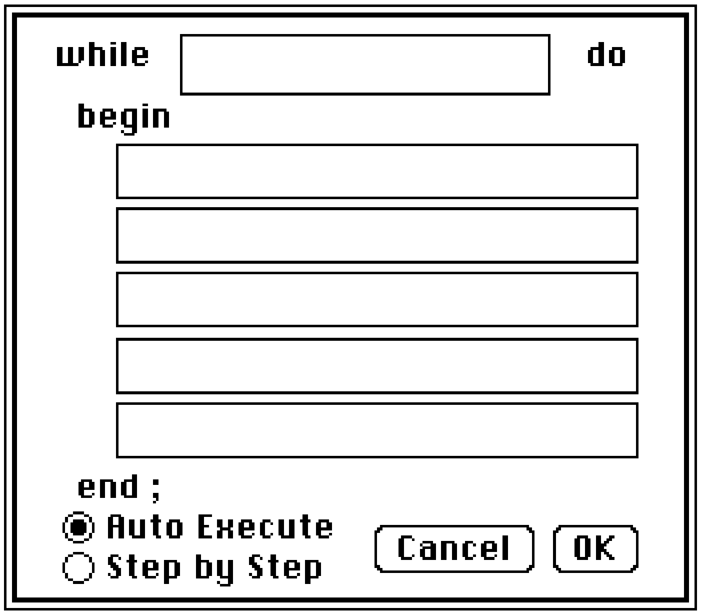
p_.field:='a' If p is nil or unassigned, an error occurs. Otherwise, a small rounded rectangle containing the letter **a** appears in the field portion of p_.

p_.field := q_.field
If p or q is nil or unassigned, an error occurs. If q_ exists, but q_.field is unassigned, an error occurs. If p_ exists, and q_.field contains a value, the appropriate assignment occurs.

p_:=q_ p_ and q_ must exist, and both q_.field and q_.next must contain values, or an error occurs. If these conditions are met, p_ receives a copy of q_, i.e., p_.field is assigned q_.field, and p_.next is assigned q_.next.

Using the While Loop:

In order to traverse linked lists, a looping control structure is most desirable. For this reason, the Pointer Demo package contains a loop structure. In order to set up a **while** loop, select **Set Up Loop** from the **While Loop** menu. A loop template appears on the right of the screen:



We may now enter a loop condition, and assignment statements which comprise the loop body. In order to move the insertion point from one text entry box to the next, either select the appropriate box with the cursor, or hit the **Tab** or **Return** key to move to the next box. Any of the legal Pascal commands discussed previously may be used in the loop body; the loop condition may be any legal Pascal condition, subject to these restrictions:

1) No precedence of operators is assumed for **AND**, **OR**, and **NOT**. Thus, full parenthesization is necessary. (For instance, this statement will be assumed syntactically incorrect:

while (p<>q) and (p^.next<>nil) or (r=q) do

2) Do not attempt to compare two character constants, e.g.,

while '3' <> '5' do

For online help, select **Conditions** from the **Help** menu.

The user may also select whether to execute the loop automatically or step by step. The choice is made by selecting one of the "radio buttons" in the bottom left corner of the **while** dialog.

When satisfied with the loop setup, the user may select OK, in which case the statements are checked for syntactic correctness until an error is found. If an error is found, it will be identified in an Alert box. Otherwise, the **while** dialog will disappear.

When ready to execute the loop that has been designed, the user may select **Execute Loop** from the **While Loop** menu. If **Auto Execute** has been chosen from the **while** dialog, the user will see instant manipulation of the data structure which faithfully reproduces the effects of the loop. If any run-time errors are detected, such as *attempt to access a nonexisting or unitialized variable, attempt to create a new variable when no space exists,* or *infinite loop,* an error message appears and looping is terminated.

If the user has chosen **Step By Step** execution, the message **Hit Return Key** appears in place of the **Enter Command** prompt, and the loop condition appears in the command box. Striking the return key causes the condition to be evaluated. If the condition is false the text "--DONE--" appears in the command box. Otherwise, each subsequent strike of the return key brings the next command (or condition) to the command box, and the currently displayed command to be executed. At the end of the loop body, the condition again appears in the command box. This procedure is repeated until either an error occurs or the condition evaluates as false. In either case, the loop is terminated, and the **Enter Command** prompt reappears beside the command box.

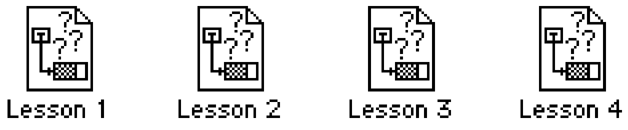
Running a Lesson:

A small set of "lessons" is supplied with the package. These lessons consist of an initial configuration and a set of instructions. The user is asked to manipulate the configuration to produce some other structure. To run one of the lessons, select **Read Lesson** from the **File** menu. The current configuration will be removed, the new initial configuration will be installed, and the instructions will appear. The user is free to attempt any commands to manipulate the structure. If necessary, the instructions may be recalled by choosing **Instructions** from the **Help** menu.

Creating a Lesson:

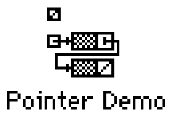
The instructor may use the package to create lessons for student use. To do this, select **Create Lesson** from the **File** menu. The screen is cleared, and a new menu, labeled **Lesson**, appears. The user creates an initial configuration by entering commands in the usual fashion. All commands entered while in "Create Mode" are saved, except for those that result in an error message. The user continues entering commands until the initial structure is complete. At this point, select **Save as Initial Structure** from the **Lesson** menu. A dialog box entitled **Make Instructions** appears. The user enters a brief set of instructions, then selects **OK**. If satisfied with the lesson, the user may choose **Save Lesson** from the **Lesson** menu. Otherwise, select **Cancel Lesson**. (This may be done at any time during the process of creating a lesson). For online help, select **Lessons** from the **Help** menu.

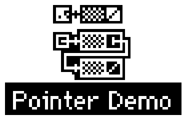
Lessons appear on the disk with a special icon (shown below). You may invoke the Pointer Demo by opening or double-clicking on a lesson icon. This lesson will then provide the initial configuration for the demo.



4. Sample Session

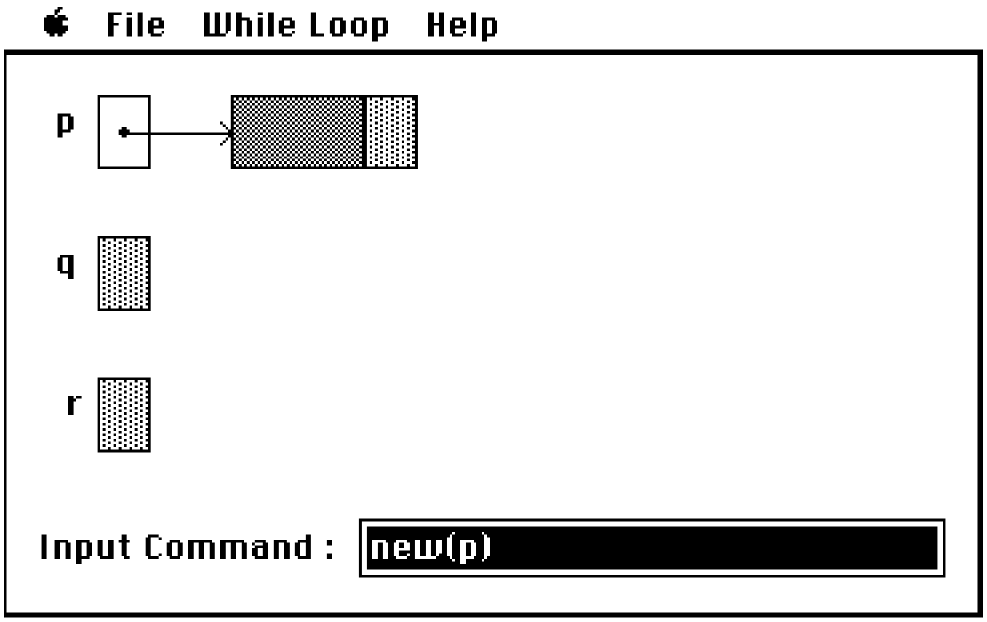
1. **Start up the application.** Click on the Pointer Demo icon:





When the item is selected, in will appear like this _:
Select **Open** from the File menu, or "double click" on the icon.

2. Type **new(p)**, followed by a carriage return. A new record is created, and p points to it. The grey boxes labeled q and r indicate that q and r are *unassigned* variables of type NodePtr.



3. Type **q:=nil** . The slash indicates that q has a *nil* value.

4. Type **new(r)** . A new record is created, and r points to it.

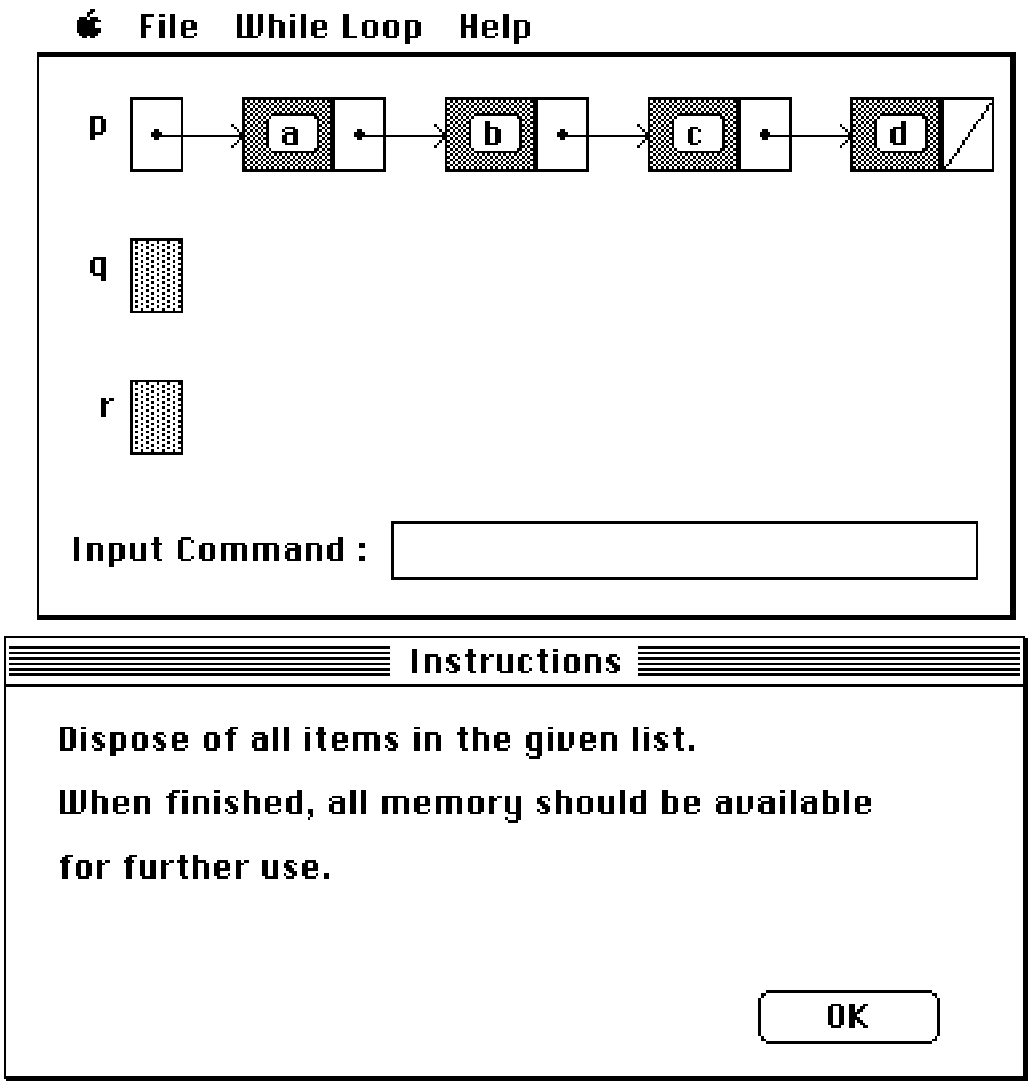
5. Type **new(r^.next)** . A new record is created, and r_.next points to it. (The ^ symbol is used to mean _).

6. Type **p^.next:=r^.next^.next** . This causes an error, because r_.next_.next is unassigned. You will be informed of your error. Type **p^.next:=r^.next** . This causes the previously unassigned pointer of record p_ to point to the same record r_.next points to.

7. Type **p^.field := '1'** . The value '1' appears in the field segment of p_ .

8. This may be a good place for you to determine if you know what is going on. Determine how to modify the structure you now have to create a list with three elements in it, with values '1', '2', '3', in order. If you have forgotten the syntax of commands, or the type declarations from the beginning of the program, select **Commands** from the **Help** menu.

9. Suppose you would like to try one of the lessons that have been created for use with this package. Select **Read Lesson** from the **File** Menu, then select **Lesson 6**. A structure is displayed on the screen, along with instructions:

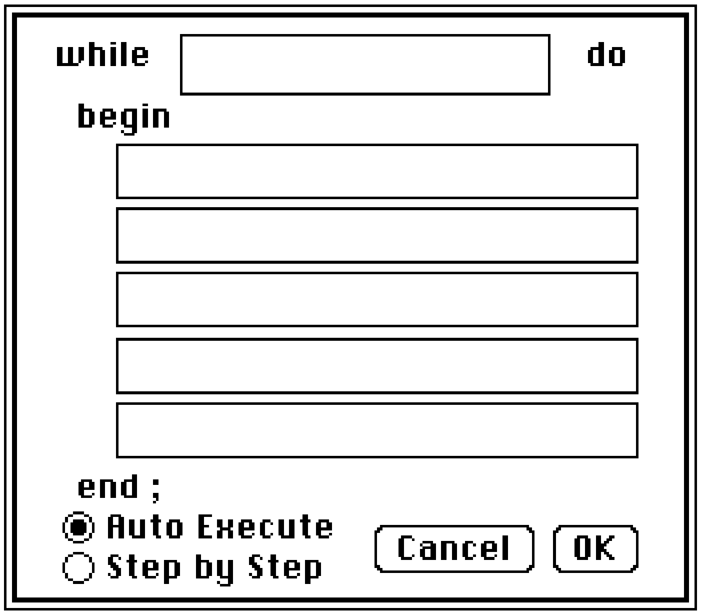


Dispose of all items in the given list.

When finished, all memory should be available

for further use.

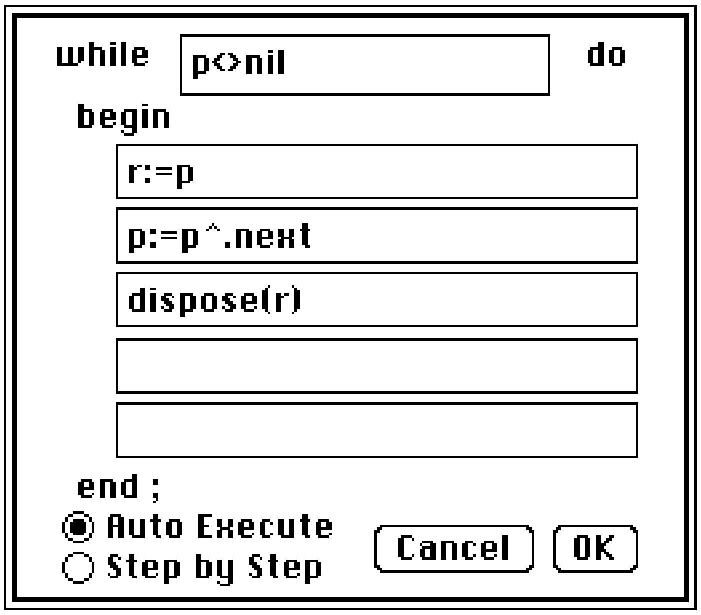
10. This exercise is best performed by repetition. Select **Set Up Loop** from the **While Loop** menu. The following figure appears at the right of the screen:



11. We wish to enter this sequence:

```
while p<>nil do
   begin
      r := p ;
      p := p_.next ;
      dispose(r) ;
   end ;
```

At present, the first text box (corresponding to the loop condition) is active. Type **p<>nil**. In order to move to the next box, hit the "Return" or "Tab" key, or click in this box with the mouse. When this box is selected, type **r := p**. Again, you may optionally enter a semicolon at the end of any command line. Enter **p:=p_.next** and **dispose(r)** in the next two boxes. Leave the remaining three text boxes blank.



```
while  p<>nil  do
  begin
    r:=p
    p:=p^.next
    dispose(r)


  end ;
```

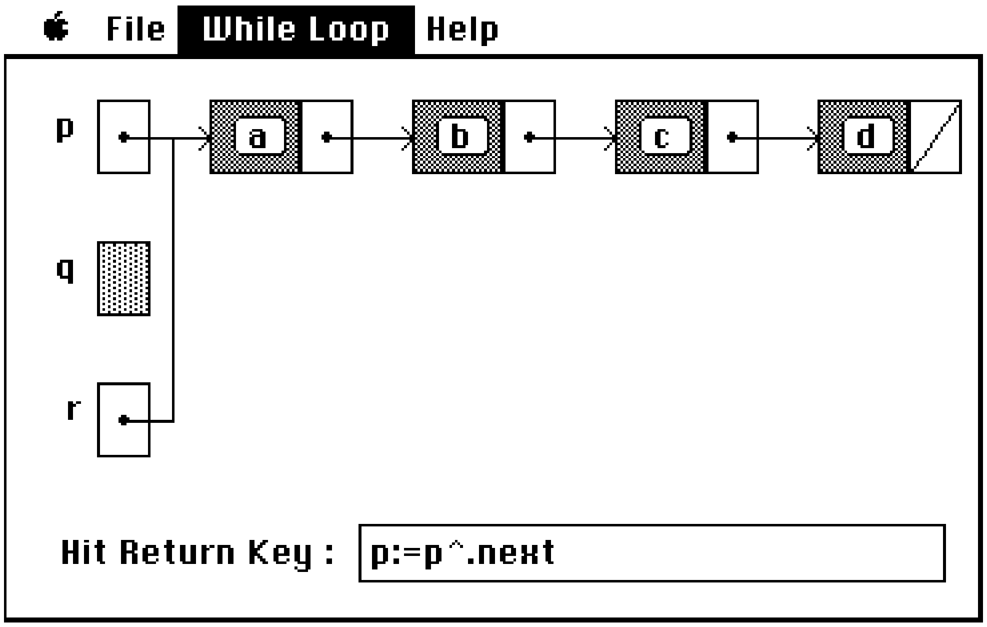
(● Auto Execute) ○ Step by Step

12. We may choose to execute this loop either Automatically or Step-by-Step. Select Step-by-Step by clicking on the item that says **Step by Step**.

13. Now that the loop has been set up as you desire, click the **OK** button. Each command you have entered is checked for syntax errors. If any error is found, an error message will appear, and the line in which it appears will be highlighted. After correcting the error, click OK again. If no errors are found, the **While Loop** dialog will disappear, which means you are ready to execute the loop.

14. Select **Execute Loop** from the **While Loop** menu. Since you have chosen to execute this loop Step by Step, the message **Hit Return Key:** appears in the display where the message **Enter Command:** appeared previously. Also, the condition **p<>nil** appears in the command box. This indicates that the next step to be executed will be to check the condition **p<>nil**. To execute this step, hit the "Return" key. If the condition is true, the first command from the loop body will appear in the command box. If not, the message "-- DONE --" will appear.

15. The command **r := p** appears in the command box. Execute this command by hitting the "Return" key. When you do this, watch on the display as r suddenly points to p_. The next command, **p := p_.next**, appears in the command box.



p:=p^.next

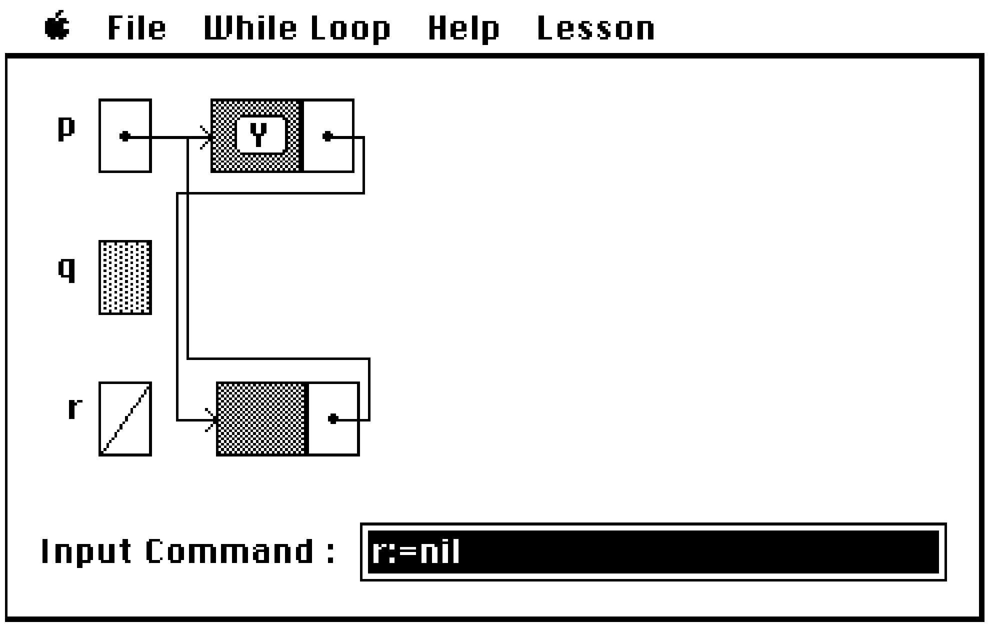
Re peatedly hit the "Return" key and watch as each succeeding command is executed. When the loop finally terminates, the message "--DONE--" will appear, and the **Hit Return Key:** message will be restored to **Enter Command:**.

16. To see how the Auto Execute mode differs from Step by Step mode, select **Lesson 6** again. Select **Set Up Loop** again, and enter the same condition and commands. This time, however, choose **Auto Execute** instead of **Step By Step**. When your commands are entered correctly, click the **OK** button. Now, when you select **Execute Loop** from the **While Loop** menu, you will see the results as the loop is executed quickly, with no further input from you.

17. If, at any time during the demo, you wish to restart with a clean slate, select **Clear** from the **File** menu. To restart a lesson from the beginning, select that lesson from the **File** menu.

18. To create a lesson, begin by selecting **Create Lesson** from the **File** menu. A new menu, **Lesson**, appears on the menu bar. Commands you enter will be recorded.

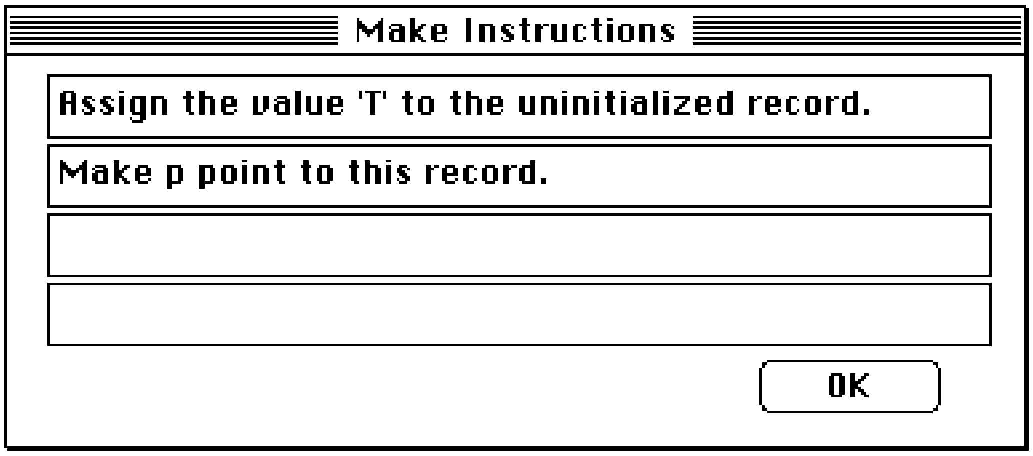
19. Enter the commands, **new(p)**, **new(r)**, **p^.field:='Y'**, **p^.next:=r**, **r^.next:=p**, **r:=nil**. If you err while entering a command, and receive an error message, don't despair. The only commands to be recorded are the ones which create the structure shown on the screen.



```
r:=nil
```

20. Save the structure by selecting **Save as Initial Structure** from the **Lesson** menu. A dialog box labeled **Make Instructions** appears.

21. Enter the following instructions: **Assign the value 'T' to the uninitialized record. Make p point to this record.** Click the OK button when the instructions are finished.



Assign the value 'T' to the uninitialized record.

Make p point to this record.

22. You may now save the lesson. Select **Save Lesson** from the **Lesson** menu. You will be asked to give the lesson a name. To check whether the lesson has been saved as you intended it, select **Read Lesson**, and choose the lesson you just created.

23. To exit the demo, select **Quit**.