Application Design Implications of PASCAL/3000 Dynamic Variable Allocation Support or How to Use the HEAP

Steven Saunders
Information Networks Division
Hewlett-Packard Company
Cupertino, California

ABSTRACT

This paper is intended to introduce PASCAL/3000's dynamic variable allocation support. This introduction is used as a basis for a discussion of application design issues relating to the PASCAL/3000 support environment. The details of the PASCAL/3000 implementation which are needed to interface to existing applications are presented.

The dynamic allocation of variables is provided through memory-management routines operating on an area called the HEAP. The PASCAL language and support environment provide a means of explicitly controlling the allocation and deallocation of variables in the HEAP. These features provide the programmer with the ablity to implement designs which combine a high degree of adaptability and reliability.

Approaches for making design and implementation decisions based on the capabilities of dynamic variable allocation in PASCAL/3000 are presented. Examples of good and bad use of these capabilities are discussed. The knowledge gained in the design and implementation of the PASCAL/3000 compiler is used as the basis for this discussion.

A good working knowledge of dynamic variable allocation can provide the application designer with insights into producing designs that are more adaptable to the user's needs. The application or systems programmer can use knowledge of the PASCAL HEAP to more effectively implement quality software.

INTRODUCTION

The programming language PASCAL has a large number of features that are provided by other programming languages available on the HP3000. There are also some significant features that are unique to PASCAL, such as strong type checking and dynamic variable support features.

The strong type checking feature, which will not be covered in this paper, permits improved compiler verification of data abstractions and module interfaces.

This paper discusses the dynamic variable support features which facilitate the creation of programs which are adaptable and versatile, yet simple.

The discussion of PASCAL's dynamic variable support features is broken into three sections:
The first section introduces the concepts of dynamic variable support, and explains the terminology used in this paper. The concepts are presented in the context of the PASCAL/3000 implementation. The defining occurrence of each new word of terminology is capitalized in the text.

The second section reexamines these concepts in the context of application design. This context is used to contrast design approaches employing PASCAL's dynamic variable support to more "conventional" approaches used with languages like FORTRAN or COBOL.

The third section discusses the interaction of the language features, implementation details, and design approaches. That interaction is examined in the context of problems that application designers and implementors might encounter with the use of dynamic variables.

I. DEFINITION AND USE OF DYNAMIC VARIABLES IN PASCAL/3000

The DYNAMIC VARIABLE support feature of PASCAL allow a program to allocate on demand any number of global variables, irrespective of the program's block structure. Dynamic variables are global, in contrast to static variables that are local to the block in which they are declared. This reflects the most important aspect of dynamic variables, the separation of the declaration and allocation of storage for a variable. The PASCAL/3000 implementation provides this basic dynamic variable support along with several extensions.

Sample Program

The following sample program is used throughout this paper to illustrate PASCAL/3000 syntax. The program builds a linked list with dynamic variables, and then traverses and prints the list. The numbers contained in the comments to the left of each line are used to reference that line in the text of the paper. The lines containing "{HP}" near the right margin are extensions defined by the Hewlett-Packard PASCAL standard.

```
{LINE}
{ 0} $HEAP_DISPOSE ON, HEAP_COMPACT OFF$
{ 1} PROGRAM IUG_Example (OUTPUT);

{ 2} TYPE
{ 3}    Pointer_Type = ^ Record_Type;

{ 4}    Record_Type = RECORD
{ 5}       Integer_Field : INTEGER;
{ 6}       Pointer_Field : Pointer_Type;
{ 7}    END;

{ 8} CONST                                                    {HP}
{ 9}    Integer_Const = 27;
{10}    Pointer_Const = NIL;                                  {HP}

{11}    Record_Const = Record_Type [                          {HP}
{12}       Integer_Field : 0,                                 {HP}
{13}       Pointer_Field : Pointer_Const                      {HP}
{14}    ];                                                    {HP}

{15} VAR
{16}    Integer_Var : INTEGER;
{17}    Pointer_Var : Pointer_Type;
{18}    Record_Var  : Record_Type;

{19}    BEGIN
{20}    Integer_Var := Integer_Const;
{21}    Pointer_Var := Pointer_Const;                         {HP}
{22}    Record_Var  := Record_Const;                          {HP}

{23}    NEW (Pointer_Var);
{24}    Record_Var .Pointer_Field := Pointer_Var;

{25}    WHILE Integer_Var > 0 DO
{26}       BEGIN
{27}       Pointer_Var^ .Integer_Field := Integer_Var;
{28}       Integer_Var := PRED(Integer_Var);

{29}       NEW (Pointer_Var^ .Pointer_Field);
{30}       Pointer_Var^ Pointer_Field^ := Record_Const;       {HP}
{31}       Pointer_Var := Pointer_Var^ .Pointer_Field;
{32}       END;

{33}    Pointer_Var := Record_Var .Pointer_Field;
{34}    WHILE Pointer_Var <> NIL DO
{35}       BEGIN
{36}       WRITELN (Pointer_Var^ .Integer_Field);
{37}       Pointer_Var := Pointer_Var^ .Pointer_Field;
{38}       END;
{39}    END.
```

Definition and Use of "Conventional" Static Variables

STATIC VARIABLES can be characterized as named storage areas that exist only during the execution of the procedure or function that in which they are declared. Their existence can be determined by simply looking at a program listing. This static nature means that the number of static variables and the size of each static variable are fixed when the program is compiled. Thus, the storage for static variables can be allocated when their declarations are processed. This is how PASCAL implementations handle static variables; storage for them is allocated when the block containing their declarations is entered.

The fixed number and size of static variables forces designers and implementors to wastefully reserve storage for rarely used and/or large data structures (e.g., data structures for year-end versus month-end processing). However, static variables are very useful for holding frequently computed results.

The variables declared in lines 16 through 18 in the sample program are global static variables. GLOBAL STATIC VARIABLES are the same as any other (local) static variables, except that they are allocated before a program begins execution and exist as long as it is executing.

Aspects of Dynamic Variables

The separation of declaration and allocation of storage for dynamic variables has one key implication: the number of dynamic variables is NOT fixed when the program is compiled. The size of an individual dynamic variable is fixed when the program IS compiled, just as it is for a static variable, but the number of dynamic variables can change. The changable number of dynamic variables enables application designers and implementors to provide storage for rare and/or large data structures without much effort.

A dynamic variable is defined as being pointed to by a pointer which can only point to a single unique type, which is the type of the dynamic variable. Line 3 of the sample program shows the declaration of the type "Pointer_Type" that points to dynamic variables of the type "Record_Type". Line 17 shows the declaration of "Pointer_Var," a static variable of this type. "Pointer_Var" 's value can be used to access a dynamic variable, but it is not itself a dynamic variable.

Line 6 shows a component, "Pointer_Field", of the structured type "Record_Type", whose value can be used to access a dynamic variable of the same structured type. This form of declaration can be employed to build linked data structures.

Usage Of Dynamic Variables

Dynamic variables must be explicitly allocated and deallocated by a program. Thus, the existence of dynamic variables depends on the dynamic (execution) behavior of a program. The number and arrangement of dynamic variables cannot be determined statically (by simply looking at a program listing). In contrast to static variables, dynamic variables do not have actual names. Dynamic variables' "names" are just unique values generated by the dynamic storage allocation mechanism.

Dynamic variables are allocated in PASCAL by calling the system-supplied procedure NEW. This procedure selects a storage area for the requested type of dynamic variable from an area called the HEAP. Lines 23 and 29 of the sample program show the use of New to allocate dynamic variables of the type "Record_Type". New sets the values of "Pointer_Var" and "Pointer_Field", in lines 23 and 29, respectively.

Dynamic variables are deallocated in PASCAL/3000 by one of two mechanisms. The first, standard to all PASCAL implementations, is the system-supplied procedure DISPOSE. This procedure deallocates a single dynamic variable at a time.

The second mechanism is a Hewlett-Packard PASCAL extension. This extension provides two additional system-supplied procedures, Mark and Release. The procedure MARK creates a generic pointer value that describes the state of the HEAP. The STATE OF THE HEAP can be characterized as a temporal reference point. All allocations of dynamic variables can be unambiguously classified as occurring either before or after this reference point. A GENERIC POINTER VALUE can be the value of any pointer, irrespective of the type of dynamic variable it is declared to point to. Any pointer having a generic pointer value does not point to any dynamic variable. The procedure RELEASE uses a generic pointer value created by a previous call to Mark to restore the state of the HEAP. This results in the deallocation of all dynamic variables allocated after the reference point denoted by the generic pointer value. Put very simply Release deallocates all dynamic variables allocated after the corresponding call to Mark.

PASCAL supplies a generic pointer value NIL which can and should be used to indicate pointers that are not currently pointing to any allocated dynamic variable.

The value of any dynamic variable can be inspected or modified by DEREFERENCING any pointer pointing to that dynamic variable. The up-arrow "^" or the at-sign "@" are used to syntactically denote dereferencing. Line 30 of the sample program shows an assignment to the dynamic variable pointed to by the "Pointer_Field" component of the dynamic variable pointed to by the static variable "Pointer_Var". All dereferences take this form of starting the dereferencing sequence with some static pointer variable (e.g., "Pointer_Var" in line 27). Lines 31, 33, 36, and 37 show the use of dereferencing to inspect the value of a component of a dynamic variable.

Implementation of Dynamic Variables in PASCAL/3000

The HEAP area of any PASCAL/3000 program is the DL-DB area of the stack segment of the process executing that program. Static variables are stored in the DB-S area of the same stack segment. The value of a pointer to an allocated dynamic variable is the word address of the first word of that dynamic variable.

The generic pointer values created by Mark are of a form known only to the PASCAL/3000 implementation. The generic pointer value Nil is equal to the word address +32767, the theoretical upper limit of a HP3000 stack segment.

The allocation of dynamic variables in PASCAL/3000 can involve one or two methods of "finding" storage space for a dynamic variable. The basic method essentially amounts to moving DL futher away from DB to get the needed storage area.

The second method requires that the compiler option "HEAP_DISPOSE ON" be specified (e.g., line 0 of sample program). If it is, a free list of deallocated areas is searched for the first area large enough to store a dynamic variable of the type requested. If this search fails then the basic method is used.

The deallocation of dynamic variables in PASCAL/3000 by the Dispose procedure depends on compiler options as shown in the decision table below:

$HEAP_DISPOSE ON$	True	True	False
$HEAP_COMPACT ON$	True	False	-
Do nothing			X
Insert area into free list	X	X	
Combine w/ adjacent free areas	X		

It should be noted that the "Do nothing" action in the table above is what the Dispose procedure does in many PASCAL implementations. The "Insert area into free list" and "Combine w/adjacent free areas" actions require the system to have one (1) word of overhead for each dynamic variable allocated.

The deallocation of dynamic variables in PASCAL/3000 by the Release procedure amounts to moving DL to where it was when Mark was called. The operation of this procedure is independent of all compiler options.

Limitations of Dynamic Variables in PASCAL/3000

The basic limitations of dynamic variables in PASCAL/3000 are that the number of variables that can be allocated is limited, and that each has its size fixed when the program is compiled. The limited number is the result of the HEAP residing entirely within the stack segment, which is limited to 32,767 words. That is felt to be the most realistic design choice simply because of the large overhead associated with randomly accessing data not stored in the stack segment. All implementations have some sort of upper limit on their dynamic storage, and this is the upper limit that makes sense on the HP3000. This limitation makes it necessary to design and implement most applications with some use of dynamic variable deallocation.

The second limitation is common to all PASCAL implementations and can be overcome by proper design.

II. DYNAMIC VARIABLE DESIGN CONSIDERATIONS AND IMPLICATIONS

The most effective way to use dynamic variables in an application is to consider their use when designing the application. The two key aspects of this approach are decomposing a program's data into indivisible data items and choosing between alternative allocation/deallocation models. The flexibility and expressiveness of dynamic variables are also important design considerations.

Constrasting Concepts of What a Data Item Is

The data items used in applications written in languages without dynamic variables tend to be thought of as counters, temporaries, buffers, and tables. The first three of these can be easily implemented as static variables. But data items used as tables can have limited flexibility if implemented as static arrays, excessive implementation complexity if they are implemented as adjustable or virtual arrays, and poor performance if implemented as files.

The problem is not the limitations of these methods of implementing tables. The problem is simply that thinking of data items as tables does not always reflect the reality of application's intended function. Instead of thinking of a data item as a monolithic table, a designer could decompose it into small pieces, each piece representing a "chunk" of information. Each of these pieces would be related to the other pieces in well-defined ways. This way of organizing information lends itself to the dynamic variable approach. The benifits of this approach are that the number of dynamic variables is not fixed as is the number of elements in a static array, and that well-defined relationships can be easily implemented by pointers.

This is not to imply that all tables should be replaced

with structures composed of linked dynamic variables. But any "table" data items that must support dynamic insertion and/or deletion of "element" data items are good candidates for implementation with dynamic variables.

The application implementor could, independently of the designer, convert any static table to a structure composed of dynamic variables. This will work acceptably in some cases, but fail in others (e.g., converting a randomly-accessed table to a simple linked list). The designer can, as part of the decomposition process, make suggestions on the use of dynamic variables. But an even more promising benefit of using dynamic variables is that the whole structure of an application could be improved. A designer that understands how dynamic variables permit an adaptive implementation will no doubt create more versatile designs.

Models of Dynamic Variable Allocation/Deallocation

The limitations of the PASCAL/3000 implementation require that most applications employing dynamic variable allocation also employ some form of dynamic deallocation. This requirement can be met by considering allocation/deallocation models as part of the application design. The choice of the proper model can maximize the number of available dynamic variables while minimizing the system overhead. Four basic models will be presented here, in order of increasing flexibility and overhead.

1. *Fire Sale Model*

The FIRE SALE MODEL, the simplest model, does not require any system support of deallocation. The name of this model is used in analogy to the nonreturnable nature of items purchased in a fire sale. An application employing the fire sale model can maintain its own free lists, one list for each type of dynamic variable. This model is useful for designing applications that will allocate new data items, but seldom, if ever, need to deallocate them. An example of an application employing the model is a PERT analysis program. The program builds the PERT graph and then analyzes it. The fire sale model makes use of the following compiler options and system-supplied procedures:

```
Compiler Options    - $HEAP_DISPOSE OFF$
System Procedures   - New
```

2. *Stack Model*

The STACK MODEL of deallocation requires that the system can save and restore the state of the HEAP. An application employing the stack model can maintain its own free lists. This model is useful for designing applications that will allocate new data items in groups and then will deallocate the groups in reverse order of allocation. An example of an application employing this model is the PASCAL/3000 compiler. The compiler groups the allocation of data items based on the block structure of the source program, processing and then deallocating the innermost block first. The stack model makes use of the following compiler options and system-supplied procedures:

```
Compiler Options    - $HEAP_DISPOSE OFF$
System Procedures   - New
                      Mark/Release
```

3. *Pool Model*

The POOL MODEL of deallocation requires that the system is able to place deallocated dynamic variables in a pool of free storage and allocate new dynamic variables from this pool. An application employing the pool model should not maintain its own free lists. This model is useful for designing applications that will allocate and deallocate data items more or less simultaneously. An example of an application employing this model would be a shop floor simulation program. The simulation would allocate, process, and deallocate job, task, and event data items in an interleaved manner. The PASCAL/3000 implementation will support two variations of this model. The first does not combine contiguous free storage blocks, and will only work well when very few unique types of dynamic variables are used. This variation of the pool model makes use of the following compiler options and system-supplied procedures:

```
Compiler Options    - $HEAP_DISPOSE ON$
                      $HEAP_COMPACT OFF$

System Procedures   - New
                      Dispose
```

The second variation does combine contiguous free storage blocks, and will work well in all cases, but suffers more execution time overhead. This variation of the pool model makes use of the following compiler options and system-supplied procedures:

```
Compiler Options     - $HEAP_DISPOSE ON$
                       $HEAP_COMPACT ON$
System Procedures    - New
                       Dispose
```

4. *Hybrid Model*

The HYBRID MODEL of deallocation requires that the system support both the stack and pool models of deallocation. An application employing the hybrid model should not maintain its own free lists. This model is useful for designing aplications that will allocate and deallocate data items simultaneously as well as in groups. An example of an application employing this model would be a natural language query processor. The program would allocate and deallocate data items to build a model of the world and the necessary queries. When a set of queries is completed, the world model data items would be deallocated as a group. The hybrid model makes use of the following compiler options and system-supplied procedures:

```
Compiler Options     - $HEAP_DISPOSE ON$
                       $HEAP_COMPACT ON$
System Procedures    - New
                       Mark/Release
                       Dispose
```

The User's Input CAN Determine The Number of Data Items

The major advantage that the use of dynamic variables can offer to the designer is adaptability. For example, someone designing an information retrieval system for employee data would not have to make arbitrary decisions about the maximum number of employee dependents that the system could handle. Rather, the designer would simply describe the employee and dependent data items and their relationship. With dynamic allocation, the required number would be allocated when the application was run.

The flexibility of dynamic variables was a key tool in designing the PASCAL/3000 compiler. The compiler was designed not to have any arbitrary limits save for the HEAP size limit. Thus, a programmer need not, for example, be worried that Case statements could have no more than 1023 case label values. This general freedom from limits without increased complexity could greatly enhance the usability and useful life of many applications.

Natural Implementation of Algorithms and Data Structures

The implementation of many algorithms and the data structures they operate on is significantly easier with dynamic variables. This is simply because many algorithms for peforming operations on complex data structures were designed with dynamic variables in mind (e.g., insertion to, deletion from, and searching of height-balanced binary trees). Thus designers, and especially implementors, can take advantage of the work of others to achieve better results without having to "re-invent the wheel." As mentioned before, the results of decomposing an application's data items can be readily expressed with dynamic variables.

Direct Representation of Data Item Relationships

The representation of the relationship between two or more static variables can only be described by the logic of the program. The relationship between two or more dynamic variables can be partially represented by the declaration of pointers as components of the dynamic variables. Thus, much of the relationship information for dynamic variables can be maintained inside the variable, while the same information for static variables is maintained outside of the variables. All this, plus the added adaptability resulting from the data item decomposition design approach make dynamic variables a superior way to represent data item relationships.

III. DYNAMIC VARIABLE PITFALLS TO BE AVOIDED

Dynamic variables can provide improvements in software quality, but they can also be used in ways that can seriously degrade quality. These pitfalls can take the following forms:

- A design using a bad choice of allocation/deallocation model.
- Interfacing with external routines that do not respect the integrity of the HEAP area.
- Design or implementation flaws resulting in attempts to access deallocated variables.

- Deliberate or accidential modification of a pointer by accessing the wrong variant of a record.
- Simply attempting to allocate more variables than can fit into a HP3000 stack segment.

While these are not all the possible pitfalls, experience shows they are the most common.

Poor Selection of Allocation/Deallocation Model

The selection of an allocation/deallocation model that is poorly matched to the application design can increase the complexity or reduce the dynamic variable capacity of the application program. If a program employs a fire sale or stack model, uses a great many dynamic variable types and maintains free lists for each of them, then the complexity of the program must increase and more storage may be wasted in free lists than would be used by the system overhead from employing a pool or hybrid model. The solution would be to switch from the fire sale or stack model to either the pool or hybrid model (e.g., use $HEAP_DISPOSE ON$).

The reverse could also be true, that only a few dynamic variable types would need free lists, each containing a few deallocated variables. The solution would then be to switch to a model that used the "HEAP_DISPOSE OFF" compiler option.

Using External Routines That Alter DL-DB Area

The PASCAL/3000 implementation assumes it has total control over the DL-DB area. But many library or utility routines implemented in SPL/3000 make free use of this area (e.g., VPlus/3000 and DSG/3000 previously used this area). The solution was for PASCAL/3000 to provide the intrinsics GETHEAP and RTNHEAP that function just like New and Dispose. These intrinsics can only solve the problem if the routines were designed, or can be modified, to request storage in independent chunks whose individual location is unimportant. There is one further constraint on using this solution with either the stack or hybrid allocation/deallocation models: all areas that an external routine allocates after Mark is called must be deallocated before the corresponding call to Release (e.g., a correct sequence is Mark, GetHeap, RtnHeap, Release).

Accessing Deallocated Dynamic Variables

Dynamic variables that have been deallocated either by Dispose or Mark/Release cannot be accessed. That is the rule, but given that pointer values in PASCAL/3000 are implemented as word addresses, any pointer to a deallocated dynamic variable that has not been modified still points to "something"! Some form of this DANGLING POINTER PROBLEM exists in almost all PASCAL implementations (e.g., it does not exist for implementations that only support the fire sale model). The software failures (bugs) caused by this problem range from bounds violations to obscure, seemingly random, failures in totally unrelated parts of a program. Clearly, at least from the experience of implementing the PASCAL/3000 compiler, this problem must be designed out, not debugged out. The solution to this problem is very application-dependent. It represents the dynamic variable pitfall that the designer should be most concerned about. Below is an incomplete list of several partial solutions that can be incorporated into applications as the designer sees fit.

1. Never have more than one pointer to any dynamic variable.
2. Never have pointer components of dynamic variables point to other dynamic variables allocated after successive calls to Mark.
3. Doubly-link all data structures so that all pointers to a dynamic variable can be set to Nil before the variable is deallocated.
4. Maintain a count of the number of pointers to a dynamic variable in that variable and never deallocate it if the count is greater than 1.
5. Study, document, and analyze the deallocation portions of an application very carefully.
6. Never deallocate any dynamic variable.

Pointers in the Variant Parts of Records

The dangling pointer problem's first cousin is the CLOBBERED POINTER PROBLEM. The problem can occur only when a pointer is a component of the variant part of a record and some component of another variant is modified. This is the worst of the variant record problems because the destruction can spread with very little trace of the source of the bug. For example, an ordinal component of one variant is modified and then a pointer in another variant is dereferenced to modify the dynamic variable it points to. This dynamic variable may now be the record length component of a file control block. The next Read call for this file will fail with an strange error.

The solution is to always use tag fields when declaring variant record types and always check the tag field before inspecting or modifing any component of the associated variant. A simpler but less-widely-useful solution is to never declare any pointer components in any variant part.

Allocating Too Many Dynamic Variables

If dynamic variables are used to implement very versatile applications, then these applications can be made to allocate more dynamic variables than will fit into a stack segment. This HEAP OVERFLOW condition can only be relieved by the deallocation of some dynamic variables. The condition can be prevented by reducing the number or the size of dynamic variables that need to be allocated. These two solutions, reacting to and preventing HEAP overflow, are explored below.

The first requirement of any design that attempts to

handle HEAP overflow is that the application must be notified of the condition. This is provided by PASCAL/3000 through the MPE library trap mechanism. The application designates a library trap handler procedure which is called when an allocation attempt causes a HEAP overflow. The trap handler procedure can take whatever action is deemed appropriate by the designer. For example, when the PASCAL/3000 compiler detects a HEAP overflow, it simply reports the condition and terminates processing.

The design flexibility of preventing a HEAP overflow is much greater than reacting to it, but it can increase application complexity. The approaches for preventing HEAP overflow fall into two broad classes: improving allocation/deallocation efficiency, and allocating some dynamic variables outside the HEAP.

The improvement of allocation/deallocation efficiency depends on reducing the lifespans, peak number, and sizes of dynamic variables used by an application. Reducing the lifespans can usually be best accomplished by using individual deallocation (Dispose) instead of group deallocation (Mark/Release). Reducing the peak or maximum number of dynamic variables allocated is very application-dependent and requires modelling alternative designs. Reducing the size of dynamic variables can be accomplished by impoving the decomposition of the data items (safe way) and/or by allocating the smallest variant needed in each case (dangerous way).

The approach of allocating dynamic variables outside the HEAP results in increased application complexity because much of the work performed by system-supplied procedures and language syntax must be duplicated in the application. The most flexible approach is to adopt coding conventions that permit migration of dynamic variables from inside to outside the HEAP with little modification to the application. The sample program shown below illustrates a coding convention that allows dynamic variables to be allocated inside or outside of the HEAP. The main advantage of this coding convention is that it localizes any changes to three procedures and one function for each dynamic variable type (e.g., _New, _Dispose, _Modify, & _Access). The main disadvantage of this convention is that it necessitates a function call, a procedure call, and moving the value of an entire dynamic variable five (5) times just to modify a single component of it. This convention basically replaces dereferences with procedure and function calls. The sample shows identical operations on two dynamic variable types, prefixed "type1_" and "type2_", one allocated in a file and the other allocated in the HEAP.

```
PROGRAM Allocate_Outside_Heap;
TYPE
   type1_Ptr = 0 .. 32000;
   type1_Rec = RECORD
      data : BOOLEAN;
   END;

   type2_Ptr = ^ type2_Rec;
   type2_Rec = RECORD
      data : BOOLEAN;
   END;
CONST
   type1_Nil   = 0;
   type1_Const = type1_Rec [data: FALSE];

   type2_Nil   = NIL;
   type2_Const = type2_Rec [data: FALSE];
VAR
   type1_Heap : RECORD
      Heap_Limit : type1_Ptr;
      Heap_Store : FILE OF type1_Rec;
   END;

   type1_Ptr1, type1_Ptr2: type1_Ptr;
   type2_Ptr1, type2_Ptr2: type2_Ptr;


   PROCEDURE type1_New (VAR Ptr: type1_Ptr);
      BEGIN
      WITH type1_Heap DO
```

```
      BEGIN
      Heap_Limit := SUCC (Heap_Limit);
      Ptr := Heap_Limit;
      END;
   END;

PROCEDURE type2_New (VAR Ptr: type2_Ptr);
   BEGIN
   NEW (Ptr);
   END;


PROCEDURE type1_Dispose (VAR Ptr: type1_Ptr);
   BEGIN
   Ptr := type1_Nil;
   END;

PROCEDURE type2_Dispose (VAR Ptr: type2_Ptr);
   BEGIN
   DISPOSE (Ptr);
   Ptr := type2_Nil;
   END;


FUNCTION type1_Access (Ptr: type1_Ptr): type1_Rec;
VAR
   type1_Temp1 : type1_REC;

   BEGIN
   WITH type1_Heap DO
      BEGIN
      ASSERT (Ptr <> type1_Nil, 0);
      ASSERT (Ptr <= Heap_Limit, 1);
      READDIR (Heap_Store, Ptr, type1_Temp1);
      type1_Access := type1_Temp1;
      END;
   END;

FUNCTION type2_Access (Ptr: type2_Ptr): type2_Rec;
   BEGIN
   type2_Access := Ptr^;
   END;


PROCEDURE type1_Modify (Ptr: type1_Ptr; Rec: type1_Rec);
   BEGIN
   WITH type1_Heap DO
      BEGIN
      ASSERT (Ptr <> type1_Nil, 0);
      ASSERT (Ptr <= Heap_Limit, 1);
      WRITEDIR (Heap_Store, Ptr, Rec);
      END;
   END;

PROCEDURE type2_Modify (Ptr: type2_Ptr; Rec: type2_Rec);
   BEGIN
   Ptr^ := Rec;
   END;
```

```
PROCEDURE Example1_Procedure (Ptr1, Ptr2: type1_Ptr);
VAR
   type1_Temp1 : type1_Rec;

   BEGIN
   type1_Temp1 := type1_Access (Ptr1);
   WITH type1_Access (Ptr2) DO
      BEGIN
      (* omitted *)
      END;
   type1_Modify (Ptr1, type1_Temp1);
   type1_Modify (Ptr2, type1_Access (Ptr1));
   END;

PROCEDURE Example2_Procedure (Ptr1, Ptr2: type2_Ptr);
VAR
   type2_Temp1 : type2_Rec;

   BEGIN
   type2_Temp1 := type2_Access (Ptr1);
   WITH type2_Access (Ptr2) DO
      BEGIN
      (* omitted *)
      END;
   type2_Modify (Ptr1, type2_Temp1);
   type2_Modify (Ptr2, type2_Access (Ptr1));
   END;


BEGIN
WITH type1_Heap DO
   BEGIN
   Heap_Limit := type1_Nil;
   OPEN (Heap_Store);
   END;

type1_New (type1_Ptr1);
type1_Modify (type1_Ptr1, type1_Const);
type1_New (type1_Ptr2);
type1_Modify (type1_Ptr2, type1_Const);
Example1_Procedure (type1_Ptr1, type1_Ptr2);
type1_Dispose (type1_Ptr1);
type1_Dispose (type1_Ptr2);

type2_New (type2_Ptr1);
type2_Modify (type2_Ptr1, type2_Const);
type2_New (type2_Ptr2);
type2_Modify (type2_Ptr2, type2_Const);
Example2_Procedure (type2_Ptr1, type2_Ptr2);
type2_Dispose (type2_Ptr1);
type2_Dispose (type2_Ptr2);
END.
```

SUMMARY

This paper introduced dynamic variables as supported by the PASCAL/3000 implementation. The concepts and models needed to use dynamic variables in designing more adaptive applications were covered. Finally, some possible pitfalls designers and implementors using dynamic variables might encounter were discussed. The interested reader might consider the following two publications.

for experts:

Pascal/3000 Program Language Reference Manual
Hewlett-Packard Company
Part No. 32106-900001

for beginners:

Programming in PASCAL with PASCAL/1000
Peter Grogono
Addison-Wesley Publishing Company Inc.

Special thanks to Wendy Peikes for her editoral suggestions.