```
        ////////    //    //    //
       //    //    ///   //    //
      //    //    ////  //    //
     ////////    // // //    //
    //          //  ////    //
   //          //   ///    //
  //          //    //    ///////
```

Pascal NewsLetter
Issue #1
May, 1990

Editor: Pete Davis

The Programmer's Forum BBS is the home of
PNL. It can be reached in Washington, DC at
(202)966-3647. Information is available
through the following locations:

FidoNet: Pete Davis@1:109/138
GEnie: PDAVIS5
BitNet: HJ647C@GWUVM & UE356C@GWUVM

Table Of Contents

Introduction

Well, welcome to the premier issue of the Pascal News Letter. Since this is the first issue, it's important that the purpose of the newsletter be stated. I run a bulletin board in Washington, and I often come across newsletters and magazine. I have yet to find any geared towards Pascal, though. (Save for a few which aren't really worthy of mention.) Because I use Pascal quite often, I became frustrated that there wasn't a free source of information. Sure, I could go out and buy some magazines that are Pascal oriented, but when there are newsletters like CNews, and MicroCornucopia about, why should I have to bother with that? There should be a Pascal oriented one. Well, now there is.

My main purpose with the newsletter is to provide a good place for the solutions to common problems. Many people have questions regarding pascal and have a lot of problems getting those questions answered. There are also a lot of people with fantastic concepts and ideas waiting to be passed around, but no means to pass it around. There will now be a way. Because this is the first issue, all articles are written by myself, this is also why it might be a bit skimpy. Hopefully this will improve with further issues. It is my hope that people with interesting ideas and concepts will pass them along to me, preferably with an article

attached.

Most of the articles are geared towards Turbo Pascal. This is not meant as a limitation, however. Turbo Pascal is about as standard as the IBM compatible pascal compilers get. Many of the ideas will be portable to other versions of pascal, if possible.

Like many of you, I'm sure, I'm a busy person. Doing this newsletter will take a significant amount of my time, but from this, I would like to explain my articles a bit. Complete pieces of code are not always provided. Instead, partial pieces of code and algorithms will make up a large part of my articles. (I am only speaking for myself and not others, who in the future may provide full pieces of code for the newsletter.) My purpose is to get the concepts and ideas across and let you, the reader, implement them in a way that suits you.

I am very fond of feedback, and would love to get a message now and then about what you, the reader, thinks about PNL, and what you would like to see in future issues. Please feel free to send your feedback, suggestions, and articles, especially, to any of the above addresses. As editor, you may expect several topical changes to your articles, but nothing major.

Pete Davis

Editor

Generic Structures in Turbo Pascal

I have to thank my current professor, Raymond Thomas for the ideas and concepts covered in this article. Although I had thought only a bit about the generic structures, his class forced me too look at them more closely.

The structure provided in our class was a generic linked list, but the idea can be carried out into several other implementations. The idea is to have a unit of code that can be re-used for different types of data. For example, one could write a unit that performs stack operations of POP and PUSH, but have it be able to work with integers, strings, real numbers, or even records. This is very useful if you want to write units and distribute them in the public domain or as shareware, and have people be able to use them for themselves. Such procedures as generic linked lists, stacks, queues, B-Trees, sorting routines, etc...

There are several features of Turbo Pascal that make this possible. Among them is the untyped pointer, the SizeOf function, the Move, GetMem, and FreeMem procedures, and the untyped parameter.

The heart of any generic structure is going to be the source of information about the structure and the information holder. In our example, it is going to be the head of the stack:


```
type
  StackPtr = ^StackItem;
  StackItem = record
    NextItem : StackPtr;
    DataItem : pointer;
  end;

  HeadPtr = ^Header;
  Header = record
    Size  : word;
    Top   : StackPtr;
  end;
```

Now for a bit of an explanation of our stack. For the most part, it follows the standard structure of a linked list implementation of a stack. The main differences one notices are the Size (in the Header record) and DataItem (in the StackItem record). Size is the size of the data items being placed in the stack. This is where Turbo Pascal's

SizeOf function comes in handy. To show how this works, I will present the StackInit procedure.

```
procedure InitStack(var H_Ptr : Header; ItemSize : word);

begin
  New(H_Ptr);
  H_Ptr^.Size := ItemSize;
  H_Ptr^.Top  := nil;
end;
```

A typical call to the InitStack procedure would be like this:

```
InitStack(H, SizeOf(MyData));
```


Now, let's examine the StackItem record. The DataItem field provides only a generic pointer. Here is where the big difference between the Generic Data Structure and the Pre-Defined Data Structure shows itself. Because DataItem is a

Generic pointer, it has no size associated with it. One is unable to use the New and Dispose procedures for the individual data items in the stack. Instead, the New and Dispose procedures are used to handle the StackItem records, while the GetMem and FreeMem procedures are used to handle the data in those StackItem records. Here is a sample of a Push procedure:

```
procedure Push(H_Ptr : Header; var Data);

{ Notice that Data is an un-typed variable }

var
  ANode : StackPtr;  { Our temporary node }

begin
  { Allocate memory for the node itself. }
  New(ANode);

  { Allocate space for the user's data and set a
    pointer to it in ANode.                    }
  GetMem(ANode^.DataItem, H_Ptr^.Size);

  { Since it's a stack, and it's the newest item,
    have it point to the previous top of the stack }
  ANode^.NextItem := H_Ptr^.Top;

  { and have the new top point to our new node }
  H_Ptr^.Top      := ANode;

  { Now physically move the data from it's current
    unprotected space, and into the space acquired for it.}
  Move(Data, ANode^.NextItem^, H_Ptr^.Size);
end;
```

Ok, so now we have our Push procedure. The comments are a little thick, but they can be removed to make it look nicer. I don't really think I need to go into too much discussion of the operations in the Push procedure, for the most part they are pretty straight forward. I would like to caution about the use of the Move procedure, however. You need to know exactly where you are moving your data to and from. It might take a little work to get exactly what you want.

Now, what good is a stack that you can only put information into, and not get any back? Not much, so here is the Pop procedure:

```
procedure Pop(H_Ptr : Header; var Data);

{ It would be nice to have the Pop procedure be a function
  returning the value, but we can't return an untyped
value                                    }

var
  ANode : StackPtr;   { Our temporary node }
```

```
begin
  { First, make sure we're working with a
    non-empty stack!                    }
  if not StackEmpty(H_Ptr) then
    begin

      { Set ANode to the top of the list }
      ANode := H_Ptr^.Top;

      { Have the Top now point to the second item
       in the list                             }
      H_Ptr^.Top := ANode^.NextItem;

      { Move the contents of the data to the user's
        variable.                              }
      Move(ANode^.DataItem^, Data, H_Ptr^.Size);
```



```
      { Return our data's memory and our node
        itself to the heap.                   }
      FreeMem(ANode^.DataItem, H_Ptr^.Size);
      Dispose(ANode);
    end
  else
    ... Error routine goes here ...
end;
```

Well, that just about covers the basics of our generic data structure. I left out some of the routines, but they should be pretty easy to add. For example, the EmptyStack function would return a boolean value of true if H_Ptr^.Top was nil. So, I'll leave the rest of it to you. There are a multitude of possibilities and uses for generic data structures. It's just one more way to make your routines 'user friendly'.

Turbo Pascal Program Optimization

This article will cover a portion of program optimization in Turbo Pascal. I say it covers a portion of optimization only because optimization is such a vast subject that it would be impossible to cover completely in a single article. I also gear it towards Turbo Pascal, but most of the tips here will apply to almost all pascal compilers.

WHERE TO OPTIMIZE:

The most important step in optimizing a program is deciding where to optimize. A good programmer is compelled to optimize as he/she writes the code. This is a good programming practice, which once you know most of the important optimizations secrets, is easy to implement as you code. A big problem occurs when you try to optimize after writing the code. The reason this is a problem is one tends to try to optimize every bit of code. This is incredibly time consuming and usually results in only mild improvements. Knowing where to optimize, however, can save a lot of time in coding and a lot of time in execution.

There are a lot of places where optimization will make huge improvements in code speed. Learning to recognize where your program is spending it's time is easy once you know

what to look for. First of all, loops! Loops should stick out like a sore thumb when optimizing a program. A user recently came into work with a program that he thought wasn't working. He complained that he had let it run for fifteen minutes and still didn't get his results. Although his program was only about 20 lines of code, it would take at least 2 hours to run. Why? He had 4 loops. Each loop was inside another loop. That adds up pretty quickly. Each one looped 100 times. Now, let's figure out how many loops that is: 100 * 100 * 100 * 100 = 100,000,000 loops total. To make things worse, he had some pretty heavy calculations inside the inner loop. Ok, so loops are definitely one place to look for optimization.

Don't just look at loops, though, look at what's inside a loop. Sometimes redundant data is placed inside loops that can be taken out of the loop. Declaring a variable inside a loop that is always the same:

example ->

```
for x:=1 to 80 do
  begin
    y:=1
    gotoxy(x,y);
    write('x');
  end;
```

Here we have the variable y being declared 80 times, while it never changes. This can be fixed by declaring y before entering into the loop. This is a very obvious example, but sometimes it isn't quite so obvious.

SPEED vs. SIZE

Optimization really covers two areas: Speed and Size. Unfortunately, optimizing for one usually ends up making the other worse. There are some exceptions, however, like passing variable parameters as opposed to value parameters as we'll cover later. Speed is almost always of primary importance and I will usually emphasize it.

VALUE vs. VARIABLE PARAMETERS

When writing procedures or functions, there is usually a little thought about whether to use variable or value parameters. The normal train of thought is that you pass a variable parameter only if that value will change in the procedure or function. Now lets look at what actually goes on behind the scenes. When a parameter is a value parameter, there are no changes made to the actual variable itself. This means that an exact copy of the variable needs to be made in memory. With a character or byte value, this isn't real significant, but what if you're passing an array of 1000 integers. That means an exact copy of 2000 bytes needs to be copied in memory. This not only takes time but it also takes up quite a bit of memory. If your routine is recursive or occurs inside a loop, you are looking at a serious decrease in speed and a huge increase in memory use. One way

to repair this is to pass variable parameters whenever possible. There are some times when it is impossible to pass a variable parameter. In these cases you'll be forced to pass a value parameters.

For those unfamiliar with variable and value parameters, here are two examples:

```
procedure ShowIt1(S : string);

begin
  write(S);
end;

procedure ShowIt2(var S : string);

begin
```

```
  write(S);
end;
```

The first procedure, ShowIt1, uses a value parameter. This means that an exact copy of the string S has to be made in memory. Since a string can be up to 255 characters, this can add up.

The second procedure uses a variable parameter. Instead of passing a complete copy of the variable, a pointer to the string S itself is passed. Since this pointer is only 4 bytes long, you can make a very significant improvement.

COMPILER DIRECTIVES

When testing a new program, it is very important to have compiler options, such as range checking and stack

checking active. Truth is, though, that these options add quite a bit of time-consuming code to your programs. Of course, it is good to have them in when writing the first copy of your program, but when compiling a final version, it is a good idea to disable these options. The results are a significant increase in speed, and a nice cut in the size of the final program.

IF/THEN vs. IF/THEN/ELSE vs. CASE

The last point of optimization that I will cover in this issue is the decision structures. Here is a short piece of code. The variable C is type char:

```
1>  ---- IF/THEN ----
if C = 'A' then writeln('You Win!');
if C = 'B' then writeln('You Lose!');
if C = 'C' then writeln('Tie Game!');

2> ---- IF/THEN/ELSE ----
if C = 'A' then writeln('You Win!') else
if C = 'B' then writeln('You Lose!') else
if C = 'C' then writeln('Tie Game!');

3> ---- CASE ----
case C of
  'A' : writeln('You Win!');
  'B' : writeln('You Lose!');
  'C' : writeln('Tie Game);
end;
```

The first example is the slowest of the three. It is also the least clear of the code. It is rarely a required structure, and can usually be replace by the more efficient

IF/THEN/ELSE structure. When possible, though, one should use the CASE structure. It is the best method for something like the above coding. It is faster and requires less memory.

Conclusion

Well, like I said in the beginning, this is a little skimpy, but it is the first issue. I hope to have a more full second issue. If you have some good ideas, please submit them. Since this is the end of the spring semester, I will be looking at quite a bit more time as summer approaches. So, with my extra time and, hopefully, your submissions, the second and later issues will be larger. I am also planning on including complete pieces of code with the newsletter itself.

Please send your submissions to the addresses provided. I hope you enjoy this and I look forward to your comments!