

Exploring Pointers in Delphi

by Craig Murphy

The subject of pointers is a traditional academic hot spot that forms part of most Pascal-orientated courses. As Delphi developers, we're often shielded from the intricacies of the underlying VCL, memory management and the WinAPI – thus a lot of the Delphi courses don't cover pointers. This is a shame, because a solid understanding of pointers makes any transition to C/C++ that little bit easier, and provides a better understanding of the application you've just written. Over the course of this article I'll cover the basics of pointers.

Introduction

During 1989 my university lecturer taught us all about arrays – static arrays. Arrays are great for storing records that are uniform in size and are of the same type. Whilst arrays are useful, they do have one flaw: any unused portion of the array is essentially wasted memory. Of course, today we could argue that dynamic arrays circumvent this flaw. True, however, if dynamic arrays avoid the need for pointers, why then does the VCL continue to use them? The answer – because they're an inherent memory management feature – working with pointers can offer performance improvements. After all, swapping two pointers is a lot faster than swapping two sizable record structures.

Pointers offer us the ability to allocate as much or as little memory as is required, there's very little wastage. Unlike arrays, pointers allow us to store any data type – we're not stuck with storing data of the same type and size.

What's in a Pointer?

A pointer is nothing more than four bytes of memory that are capable of holding a memory address.

Figure 1 shows how a pointer is represented, at a simplistic level.

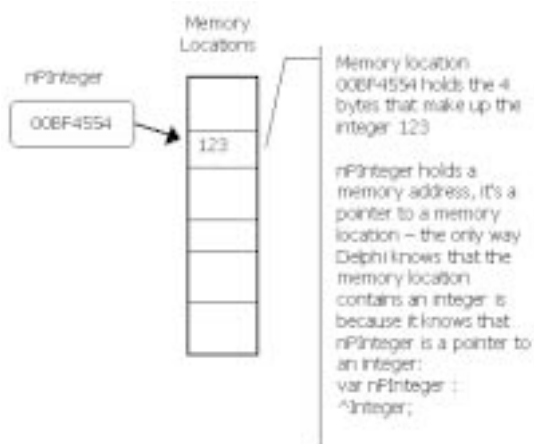


Figure 1

Memory Allocation – Delphi Methods

Delphi offers more than one means of allocating memory on the fly: there's the New and GetMem procedures. The Delphi syntax is shown below:

```
procedure New(var P: Pointer);
procedure GetMem(var P: Pointer; Size:
Integer);
```

Using the New procedure is relatively easy, as the following code fragment (Listing 1) demonstrates:

```
1:var nPInteger : ^Integer;
2:   sPtr       : String;
3:begin
4:
5:   New(nPInteger);
6:   nPInteger^ := 123;
7:   edtData.Text := IntToStr(nPInteger^);
8:
9:   FmtStr(sPtr, '%p', [nPInteger]);
10:  edtPointer.Text := sPtr;
11:
12:  edtSize.Text :=
      (IntToStr(SizeOf(nPInteger^)));
13:
14:  Dispose(nPInteger);
end;
```

Listing 1

Don't worry about the funny "hat" symbol, I'll explain that shortly. Let's go through each line:

1. Declares a variable, nPInteger, as being of type "a pointer to an Integer". At this point in time, nPInteger should be treated as un-initialised memory.
5. Calls the New procedure, which we know accepts (by var or by reference) "something of type Pointer". If we assume that the memory can be allocated, then nPInteger will point to an area of memory capable of holding an integer – it'll be 4 bytes in size.
6. Allocates the integer value 123 to "whatever nPInteger points to" – which we know to be an area of memory 4 bytes in size.
7. Take "whatever nPInteger points to", convert it into a string, and then display it in an edit control.
9. Convert the value of nPInteger (which is a pointer) into a string.
10. Display the value of nPInteger – as represented by sPtr, this is actually a memory address.
12. Display the number of bytes required by the object that nPInteger points to – in this case an integer.
14. Return the memory to the free pool – more on this later.

Some confusion arises over the positioning of the "hat" (^). In Delphi it can appear in two places:

1. Before a type name – in which case we can read it as “a pointer to a variable of that type”, e.g. `^Integer`
6. After a pointer variable, e.g. `nPInteger^` where it can be read as “the data that the pointer variable points to”

The `New` procedure automatically determines the amount of memory that needs to be allocated – it uses type information to obtain the number of bytes required.

The `GetMem` procedure is used to allocate a specific amount of memory, i.e. you must know in advance the number of bytes required. `GetMem` is particularly useful if you need to load a file into memory, as we’ll see shortly.

Hats, stars and @ signs...

We’ve seen how to allocate memory and how to use the memory – let’s see what all the pointer symbols mean.

Developers frequently refer to the operator for “getting at the actual data that a pointer points” to as “the hat” operator. Getting at the actual data is known as de-referencing. The operator is actually a circumflex: `^`.

Delphi doesn’t use the C/C++ de-reference operator, the asterisk or star (`*`) apart from when it’s used for arithmetic multiplication. De-referencing in C/C++ is a little simpler than it is in Delphi, as the following C example demonstrates, Listing 2:

```
{
int *n;

n = malloc(sizeof(int));

*n = 123;

printf("%d", *n);
}
```

Listing 2

The code above performs a similar task to the Delphi snippet we saw earlier. Notice how the de-reference operator (`*`) is always in the same place.

The [Delphi] address-of operator – the `@` sign - is a slightly different beast and is covered later in this article.

Memory De-allocation

Obviously, for every memory allocation there should be a corresponding de-allocation. Delphi provides us with two procedures for achieving this:

```
procedure Dispose(var P: Pointer);
procedure FreeMem(var P: Pointer[; Size:
Integer]);
```

From a programmatic aspect, we would use `New` & `Dispose` together and `GetMem` & `FreeMem` together. If you refer back to Listing 1, line 14 demonstrates the use of `Dispose`.

When it all goes wrong

The scaremongers will have us believe that pointers are difficult to work with and are hard to debug. Yes, it’s true that they carry a health warning, but so do most of the techniques that we take for granted.

To protect your pointer-based code (and other code for that matter), use `try...finally`. Listing 3 presents a simple example that loads a `.pas` file into memory, then copies it into a `TMemo`. There are nested `try..finally` statements – the first statement cleans up if there is a problem with the file. The second statement cleans up if the memory allocation encountered problems.

```
var
  F: file;
  Index,Size: Integer;
  Buffer: PChar;

begin
  AssignFile(F, 'unit1.pas');
  Reset(F, 1);
  try
    Size := FileSize(F);
    GetMem(Buffer, Size);
    edtFileSize.Text:=IntToStr(Size);
    try
      BlockRead(F, Buffer^, Size);

      Index:=0;

      while Index < Size do begin
        Memo.Text:=Memo.Text + Buffer[Index];
        Inc(Index);
      end;

    finally
      FreeMem(Buffer);
    end;
  finally
    CloseFile(F);
  end;

end;
```

Listing 3

It’s likely that you’ll be allocating and de-allocating memory in different places. Thus, you might find yourself de-allocating the same block of memory more than once. Delphi will raise an exception, `EInvalidPointer`,

```
var nPFreeTwice : ^Integer;
begin
  New(nPFreeTwice);
  nPFreeTwice^:=123;
  Dispose(nPFreeTwice);
  Dispose(nPFreeTwice);
end;
```

Listing 4

Attempting to allocate an amount of memory more than once is a little harder to trap.

```
New(nPGetTwice);
FmtStr(sPtr, '%p', [nPGetTwice]);
ShowMessage(sPtr);

New(nPGetTwice);
FmtStr(sPtr, '%p', [nPGetTwice]);
ShowMessage(sPtr);
```

Listing 5

Listing 5, when executed, allocates some memory then displays the address of the allocated memory on the screen. As Figures 3 and 4 show, after two allocations we have two blocks of memory – their memory addresses are shown. However we can only access the latter block, we’re unable

to work with the first block. If we can't work with the first block of memory, we can't Dispose of it either – we've got a memory leak. Listing 5 demonstrates the code required to create a memory leak.

Figure 2



Figure 3



Procedure Pointers

Every time you add a button to a form, and then assign an OnClick event handler you're actually using a procedure pointer. Procedure pointers are just like interfaces – they are a specification. They are ideal if you have a library procedure or function that needs to execute some standard code as well as executing some code unique to the application that uses the library. In this instance, the application would provide the address of a procedure that performs the unique tasks; the library procedure then calls that procedure.

Figure 4 represents this notion. At stage 1 the application registers the custom procedure, probably during initialisation. At stage 2, the application calls the library procedure/function. The library procedure/function knows that it can't do everything that the application requires – so at stage 3, it calls the custom procedure. This situation is likely to arise frequently. I've omitted stage 4 – the application should really unregister the procedure too.

Perhaps you've written a multi-lingual application? The library code doesn't change, but the application itself must do something specific for each language it has been built to handle. Procedure pointers are the ideal solution.

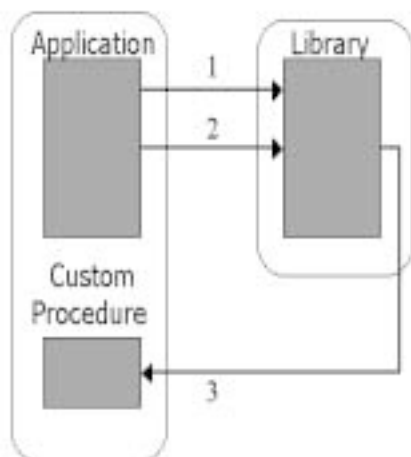


Figure 4

Procedure pointers don't just apply to procedures – they apply to functions too. The following code snippet demonstrates the use of procedure pointers (applied to functions) and the @ operator.

```

1:procedure TForm1.btnProcedureClick(Sender:
                                TObject);
2:
3:type pf= function (x,y : Integer): Integer;
4:
5:var pAdd,pSubtract : pf;
6:
7:   function Add(x,y : Integer) : Integer;
8:   begin
9:       Add:=x+y;
10:  end;
11:
12:   function Subtract(x,y : Integer) :
                                Integer;
13:   begin
14:       Subtract:=x-y;
15:  end;
16:
17:begin
18:
19:   pAdd:=@Add;
20:
21:   ShowMessage('Add: ' +
                inttostr(pAdd(1,2)));
22:
23:   pSubtract:=@Subtract;
24:
25:   ShowMessage('Subtract: ' +
                inttostr(pSubtract(2,1)));
26:
27:end;

```

Listing 6

3. Defines a type called pf, which is a pointer to a function that takes two parameters of type integer and returns an integer.
5. Declares two variables, pAdd and pSubtract that are of type pf.
7. Declares the Add function.
12. Declares the Subtract function.
19. Assigns the address of the Add function to the variable pAdd. Thus pAdd now points to the address of the Add function.
21. Calls the function that pAdd points to – notice how there's no need to de-reference pAdd.
23. Assigns the address of the Subtract function to the variable pSubtract. Thus pSubtract now points to the address of the Subtract function.
25. Calls the function that pSubtract points to – again, there's no need to de-reference pSubtract.

If you don't feel comfortable with the @ operator, you can easily replace it with the Addr() function:

```
pAdd:=Addr(Add);
```

System.pas defines a number of pointer types – they're perhaps a little more readable than their traditional declarations:

```

PSmallInt   = ^SmallInt;
PInteger    = ^Integer;
PSingle     = ^Single;
PDouble     = ^Double;
PDate       = ^Date;

```

Two hats are better than one?

My mother taught me that it's rude to point. However, do two pointers make everything polite? Well, maybe not, but they can be of use – especially for memory management, as we'll see shortly.

A “pointer to a pointer” is known as double indirection, and is de-referenced using two ^ operators: ^^.

```

1:var nPInteger : PInteger;
2:  ppInt      : ^PInteger;
3:begin
4:
5:  New(nPInteger);
6:
7:  nPInteger^ := 999;
8:
9:  ppInt := @nPInteger;
10:
11: ShowMessage(IntToStr(ppInt^^))
end;

```

Listing 7

Figure 5 depicts listing 7.

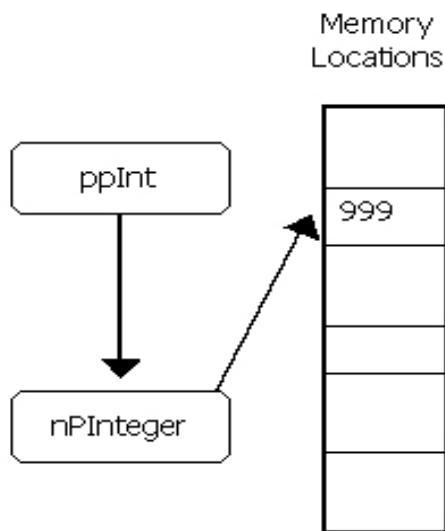


Figure 5

You're possibly wondering why we might need double indirection. As I mentioned earlier, a common use is memory management. A typical memory manager may hold a list of pointers to blocks of memory. Those blocks of memory can be freed, and reduced or extended in size – thus we have a variable sized amount of heap memory. If we simply gave our client application access to the memory blocks, the memory manager would have to notify the client application every time the heap memory shrunk or grew – because the client application has pointer variables that point to the heap memory. Whilst I'm using the memory manager as an example, the code I'll present will be just enough to demonstrate what's happening – obviously presenting a complete memory manager is beyond the scope of an article about pointers!

To elegantly deal with this problem, our client application maintains a pointer (1) to a pointer (2) that points into the heap memory. The client manages pointer (1), whereas the memory manager handles changes to pointer (2). Figure 6 represents this concept.

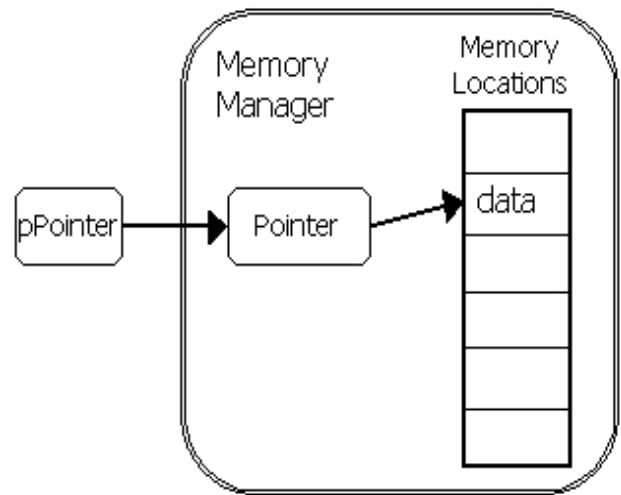


Figure 6

Listing 8 simulates what goes on when the memory manager adjusts the pointer it has. To make it clear what has happened, I've changed the data too – the original integer value was 999, after the memory manager's pointer has been updated, the value has been changed to 123. Obviously this wouldn't happen in a real scenario.

```

1:var nPInteger : PInteger;
2:  nPOtherInteger : PInteger;
3:  ppInt      : ^PInteger;
4:begin
5:  New(nPInteger);
6:
7:  nPInteger^ := 999;
8:
9:  ppInt := @nPInteger;
10:
11: ShowMessage(IntToStr(ppInt^^));
12:
13:
14: New(nPOtherInteger);
15:
16: nPOtherInteger^ := 123;
17:
18: nPInteger := nPOtherInteger;
19:
20: ShowMessage(IntToStr(ppInt^^));
21:end;

```

Listing 8

If I explain what's happening line-by-line, it becomes clear that we've found a use for double-indirection:

1. Declares a pointer to an integer
2. Declares another pointer to an integer
3. Declares a pointer to a pointer to an integer
7. Whatever nPInteger is pointing to, assign it the value 999
9. Assign the address of nPInteger to ppInt
11. Whatever ppInt is pointing to, de-reference it and convert it to a string for display
16. Whatever nPOtherInteger is pointing to, assign it the value 123
18. nPInteger now points at whatever nPOtherInteger is pointing at

20. However, ppInt hasn't changed, it still points at the address of nPInteger, so whatever ppInt is pointing to (nPInteger), de-reference it and convert it to a string from display

Figure 7 depicts what happened at line 18 – the memory manager changes the pointer it has (perhaps because the heap has been compacted or an item has grown/shrunk), and the data has moved.

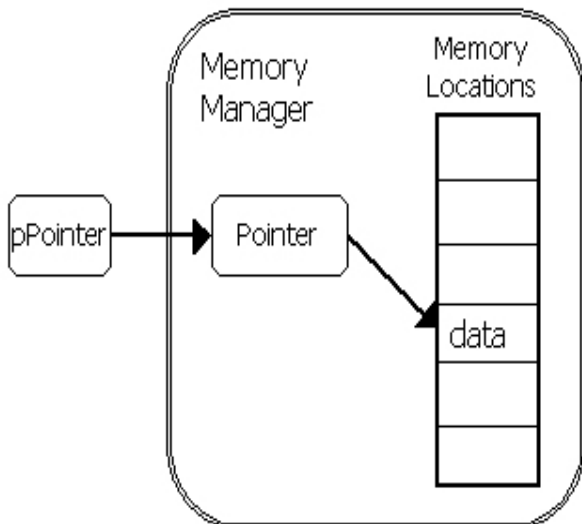


Figure 7

Summary

I hope that pointers are now a little friendlier than they were at the start of this article. Over the course of this article I've taken a look at pointers in Delphi and their uses, and covered some of the problems you might experience. Whilst the VCL makes life easier by providing abstractions that hide pointers from us, the time will come when we'll find ourselves looking under the hood – and then we'll need to know about pointers! If you're keen to learn more about pointers, take a look inside classes.pas – the TList object is a good place to start.

All the source code for this article is available for download from my web site: <http://www.isleofjura.demon.co.uk/bug>



*Craig works as an Enterprise Developer (and **Dilbert** Evangelist!) for Currie & Brown (<http://www.currieb.com>) – their primary business is quantity surveying, cost management and project management. He can be reached via e-mail at: Craig.Murphy@currieb.co.uk or*

Craig@isleofjura.demon.co.uk



Angus' Nifty Tips

If you want to give your users the opportunity to connect to network resources

from within your program, all you have to do is call the "WNetConnectionDialog()" Win32 API function:

```
WNetConnectionDialog(0, RESOURCETYPE_DISK );
```