Rudy's Delphi Corner

# Addressing pointers

> *Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover.* — Anthony Hoare

Pointers are probably among the most misunderstood and most feared data types. That is why many programmers love to avoid them.

But pointers are important. Even in languages that do not support pointers explicitly, or which make it hard to use pointers, pointers are important factors behind the scenes. I think it is very important to understand them. There are different approaches to understanding pointers.

This article was written for everyone with problems understanding or using pointers. It discusses my working view on pointers in Delphi for Win32, which may not be entirely accurate in all aspects (for instance, memory for one program is not one big block, but for most practical purposes, it helps to pretend it is). This way, pointers are easiest to understand, in my opinion.

## Memory

*You probably already know what I write in this paragraph, but it is probably good to read it anyway, since it shows my view on things, which may differ a bit from your own.*

Pointers are variables which are used to point to other variables. To explain them, it is necessary to understand the concept of a memory address and the concept of a variable. To do this, I'll first have to roughly explain computer memory.

In short, computer memory can be seen as one very long row of bytes. A byte is a small storage unit that can contain 256 separate values (0 up to 255). In current 32 bit Delphi, memory can (with a few exceptions) be seen as an array of maximum 2 gigabytes in size ($2^{31}$ bytes). What these bytes contain, depends on how the contents are interpreted, i.e. how they are used. The value of 97 can mean a byte of value 97, as well as the character `'a'`. If you combine more than one byte, you can store much larger values. In 2 bytes you can store 256*256 different values, etc.

The bytes in memory can be addressed by numbering them, starting at 0, and up to 2147483647 (assuming you have 2 gigabyte — and even if you don't have them, Windows will try to make it look as if you have them). The index of a byte in this huge array is called its address.

One could also say: a byte is the smallest addressable piece of memory.

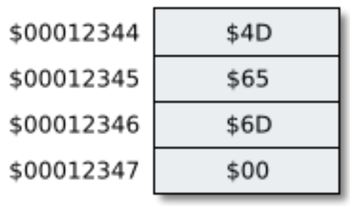
> In reality, memory is a lot more complex. There are for instance computers with bytes that are not 8 bit, which means they can contain fewer or more than 256 values, but not the computers on which Delphi for Win32 runs. Memory is managed in hardware and software, and not all memory is really existent (memory managers take care that your program doesn't notice, though, by swapping parts of memory out to or in from harddisk), but for this article, it helps to see memory as one huge block of single bytes, divided up to be used for several programs.

## Variables

A variable is a location made up of one or more bytes in this huge "array", from which you can read or to which you can write. It is identified by its *name*, but also by its *type*, its *value* and its *address*.

> *If you declare a variable, the compiler reserves a piece of memory of the appropriate size. Where this variable is stored is decided by the compiler and the runtime code. You should never make assumptions about where exactly a variable will be located.*

The type of the variable defines how the memory location is used. It defines its **size**, i.e. how many bytes it occupies, but also its **structure**. For instance, the following shows a diagram of a piece of memory. It shows 4 bytes starting at address `$00012344`. The bytes contain the values `$4D`, `$65`, `$6D` and `$00`, respectively.



*Note that although I use addresses like `$00012344` in most of the diagrams, these are completely made up, and only used to distinguish different memory locations. They do not reflect the real memory addresses, since these depend on many things, and can not be predicted.*

The type decides how these bytes are used. It can for instance be an `Integer` with value `7169357 ($006D654D)`, or an `array[0..3] of AnsiChar`, forming the C-style string `'Mem'`, or something else, like a set variable, a number of single bytes, a small record, a `Single`, part of a `Double`, etc.. In other words, the meaning of a piece of memory is not known before you know the type or types of the variable or variables stored there.

The address of a variable is the address of its first byte. In the diagram above, assuming this shows a variable of type `Integer`, its address is `$00012344`.

## Uninitialized variables

The memory for variables can be reused. The memory set aside for variables is usually only reserved as long as the program can access them. E.g., local variables of a function or procedure (I like to call both routines) are only valid as long as the routine is running. Fields of an object (which are also variables) are also only valid as long as the object "exists".

If you declare a variable, the compiler reserves the required number of bytes for that variable. But the contents may well be what was already put in these bytes before, when they were used in another function or procedure. In other words, the value of an **uninitialized** variable is *undefined* (but not necessarily *undetermined*). An example is given in the form of this simple console program:

```
program uninitializedVar;

{$APPTYPE CONSOLE}

procedure Test;
var
  A: Integer;
begin
  Writeln(A); // uninitialized yet
  A := 12345;
  Writeln(A); // initialized: 12345
end;

begin
  Test;
  Readln;
end.
```

The first value displayed (the value of the uninitialized variable A) depends on the already existing content of the memory location reserved for A. In my case, it displays the value `2147319808 ($7FFD8000)` each time, but this can be totally different on your computer. The value is undefined, because it was not initialized. In a more complex program, especially — but not only — when pointers are concerned, this is is a frequent cause of program crashes or unexpected results. The assignment initializes A with the value `12345 ($00003039)`, so that is the second value displayed.

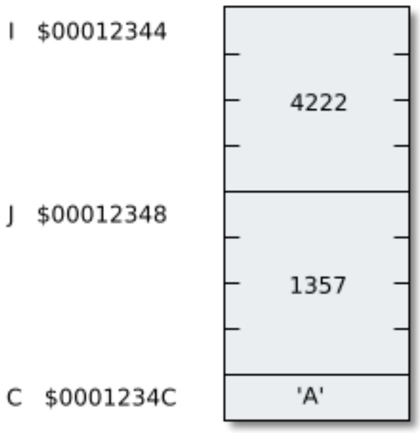
## Pointers

Pointers are also variables. But they do not contain numbers or characters, they contain the address of a memory location instead. If you see memory as an array, *a pointer can be seen as an entry in the array which contains the index of another entry in the array*.

Say I have the following declaration and initialisation:

```
var
  I: Integer;
  J: Integer;
  C: AnsiChar;
begin
  I := 4222;
  J := 1357;
  C := 'A';
```

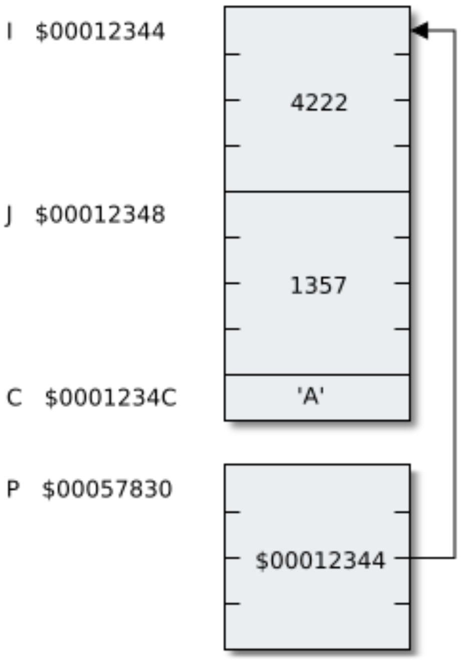
Let’s assume it results in the following memory layout:



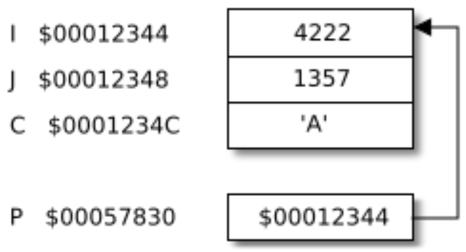
Now, after this code, assuming `P` is a pointer,

```
  P := @I;
```

I have the following situation:



*In the previous diagrams, I always showed each byte. This is generally not necessary, so the above could just as well be* *shown as:*



*This does not reflect the actual size anymore (*`C` looks *just as big as* `I` *or* `J`*), but it is good enough to understand what is going on with pointers.*

## Nil

*Thou shalt not follow the* `NULL` *pointer, for chaos and madness await thee at its end.* — Henry Spencer

`Nil` is a special pointer value. It can be assigned to any kind of pointer. It stands for the empty pointer (*nil* is Latin short for *nihil*, which means *nothing* or *zero*; others say NIL means *Not In List*). It means that the pointer has a defined state, but not that you should attempt to access the value (in C, `nil` is called `NULL` — see the quote above).

`Nil` never points to valid memory, but since it is one well defined value, many routines can test for it (e.g. using the the `Assigned()` function). One can not test if any other value is valid. **Stale or uninitialized pointers look no different than valid pointers** (see below). There is no way to distinguish them. Program logic must always ensure that a pointer is either valid, or `nil`.

*In Delphi,* `nil` *has the value* `0`*, i.e. it points to the very first byte in memory. This is apparently a byte that will never be accessed by Delphi code. But you should generally not rely on* `nil` *being* `0`*, unless you are fully aware of what is going on behind the scenes. The value of* `nil` *could change in a later version, for one reason or other.*

## Typed pointers

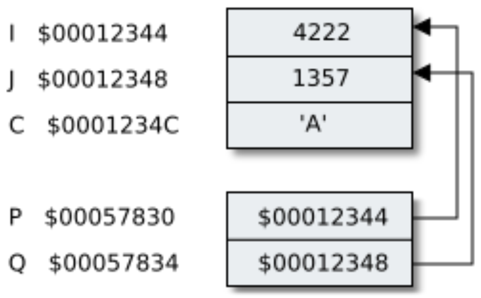
In the simple example above, `P` is of type `Pointer`. This means that `P` contains an address, but you don't know what the variable at that address is supposed to contain. That is why pointers are usually typed, i.e. the pointer is interpreted to be pointing to a memory location that is supposed to contain a certain type.

Let's assume we have another pointer, Q:

```
var
  Q: ^Integer;
```

`Q` is of type `^Integer`, which should be read as "pointer to Integer" (I was told that `^Integer` stands for ↑ `Integer`). This means that it is not an `Integer`, but points to a memory location, which is to be used as one, instead. If you assign the address of `J` to `Q`, using the `@` address operator or the functionally equivalent `Addr` pseudo-function,

```
  Q := @J; // Q := Addr(J);
```



then `Q` points to the location at address `$00012348` (it **references** the memory location identified by `J`). But since `Q` is a **typed pointer**, the compiler will treat the memory location to which `Q` points as an `Integer`. `Integer` is the **base type** of `Q`.

*Although you will hardly ever see the `Addr` pseudo-function being used, it is equivalent to `@`. `@` has the disadvantage that, when applied to complicated expressions, it is not always obvious to which part the operator applies. `Addr`, using the syntax of a function, is much less ambiguous, since the target is enclosed in parentheses `()`:*

```
  P := @PMyRec^.Integers^[6];
  Q := Addr(PMyRec^.Integers^[6]);
```

Assignment using a pointer is a bit different than direct assignment to a variable. Generally, you only have the pointer to go by. If you assign to a normal variable, you write something like:

```
  J := 98765;
```

That stores the integer `98765 (hex $000181CD)` in the memory location. But to access the memory location using `Q`, you must work *indirectly*, using the `^` operator:

```
  Q^ := 98765;
```

This is called **dereferencing**. You must follow the imaginary "arrow" to the location to which `Q` points (in other words, the `Integer` at address `$00012348`) and store it there.

*For records, the syntax allows you to omit the `^` operator, if the code is unambiguous without it. For clarity reasons, I personally always write it, though.*

*It is generally useful to define types for the pointers one is using. For instance,* `^Integer` *is not a valid parameter type declaration, so you'll have to predefine a type:*

```
type
  PInteger = ^Integer;

procedure Abracadabra(I: PInteger);
```

*In fact, the* `PInteger` *type and some other common pointer types are already defined in the Delphi runtime library (e.g. units* `System` *and* `SysUtils`*). It is custom to start the names of pointer types with the capital letter* `P` *followed by the type to which they point. If the base type is prefixed with a capital* `T`*, the* `T` *is usually omitted. Examples:*

```
type
  PByte = ^Byte;
  PDouble = ^Double;
  PRect = ^TRect;
  PPoint = ^TPoint;
```

## Anonymous variables

In the previous examples, variables were declared where they were needed. Sometimes you don't know whether you need a variable or not, or how many. Using pointers, you can have so called **anonymous variables**. You can ask the runtime to reserve a piece of memory for you, and return a pointer to it, using the `New()` pseudo-function:

```
var
  PI: PInteger;
begin
  New(PI);
```

`New()` is a compiler pseudo-function. It reserves memory for the base type of `PI`, and then points `PI` to that piece of memory (i.e. stores the address in `PI`). The variable doesn't have a name, so it is anonymous. It is only accessible indirectly, using the pointer. Now you can assign to it, pass it around to routines, and get rid of it when you don't need it, using `Dispose(PI)`:

```
  PI^ := 12345;
  ListBox1.Add(IntToStr(PI^));
  // lots of code
  Dispose(PI);
end;
```

*Instead of* `New` *and* `Dispose`*, you could also go lower level, and use* `GetMem` *and* `FreeMem`*. But* `New` *and* `Dispose` *have a few advantages. They already know the type of the pointer, and also initialize and finalize the memory location, if that is necessary. So it is advisable to use* `New` *and* `Dispose`*, instead of* `GetMem` *and* `FreeMem`*, whenever you can.*

*Always make sure that every* `New()` *is eventually followed by a* `Dispose()` *on the* ***same pointer value and type****, otherwise some variables may not be finalized properly.*

It may not be obvious how this is better than declaring a variable directly, but there are situations where this is useful, usually if you don't know how many variables you need. Think of nodes in a linked list (see below), or of a `TList`. `TList` stores pointers, and if you want to have a list of `Double` values, you simply `New()` each value and store it in the `TList`:

```
var
  P: PDouble;
begin
  while HasValues(SomeThing) do
  begin
    New(P);
    P^ := ReadValue(SomeThing);
    MyList.Add(P);
    // etc...
```

Of course you will have to `Dispose()` each value at a later stage, when the list is not used anymore.

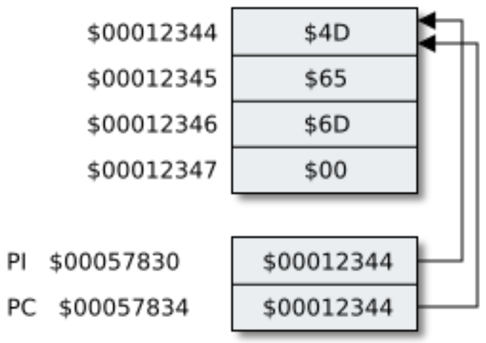
Using anonymous variables, it is easy to demonstrate that typed pointers can determine how memory is used. Two pointers with different types, pointing to the same memory, will show different values:

```
program InterpretMem;

{$APPTYPE CONSOLE}

var
  PI: PInteger;
  PC: PAnsiChar;
begin
  New(PI);
  PI^ := $006D654D;      // Bytes $4D $65 $6D $00
  PC := PAnsiChar(PI);  // Both point to same address now.
  Writeln(PI^);          // Write integer.
  Writeln(PC^);          // Write one character ($4D).
  Writeln(PC);           // Interpret $4D $65 $6D $00 as C-style string.
  Dispose(PI);
  Readln;
end.
```

`PI` fills the memory location with the value `$006D654D (7169357)`. In a diagram (note that the addresses are purely fictitious):



`PC` is then pointed to the same memory location (since the base types of the pointers are not the same, you can not just assign one to the other — you will have to cast). But `PC` is a pointer to an `AnsiChar`, so if you take `PC^`, you get an `AnsiChar`, which is the character with ASCII value `$4D`, or `'M'`.

`PC` is a special case, though, since the type `PAnsiChar`, although it is actually just a pointer to `AnsiChar`, is treated a little differently than most other pointer types. I explain this in another article. `PC`, if not derefenced, is generally seen as a pointer to a piece of text ending in a zero character `#0`, and `Writeln(PC)` will display the text formed by the bytes `$4D $65 $6D $00`, which is `'Mem'`.

*When thinking about pointers, and especially about complex pointer situations, I usually have a piece of paper and a pen or pencil handy, to draw the kind of diagrams you see in this article. I give variables made up addresses too (they don't have to be 32 bit, addresses like* `30000`, `40000`, `40004`, `40008` *and* `50000` *are good enough to follow what is going where).*

## Bad, bad pointers

If used properly, pointers are very useful and flexible tools. But when you make a mistake, they can be a big problem. That is another reason why many people avoid pointers as much as they can. Some of the most common errors are described here.

## Uninitialized pointers

Pointers are variables, and like any variable, they should be initialized, either by assigning another pointer value to them, or by using `New` or `GetMem` or some such:

```
var
  P1: PInteger;
  P2: PInteger;
  P3: PInteger;
  I: Integer;
begin
  I := 0;
  P1 := @I;  // OK: using @ operator
  P2 := P1;  // OK: assign other pointer
  New(P3);   // OK: New
  Dispose(P3);
end;
```

If you simply declare, say, a `PInteger`, but don't initialize it, the pointer probably contains some random bytes, i.e. it points to a random place somewhere in memory.

If you start accessing that random place in memory, ugly things can happen. If the memory is outside the memory reserved for your application, you most likely get an Access Violation, a program crash. But if the memory is part of your program, and you write to it, you could be overwriting data that may perhaps not be changed. If the data is used in another part of the program, at a later point in time, the results your program produces may be wrong. Such errors can be extremely hard to find.

So actually, if you get an AV or some other kind of obvious crash, you can actually be glad (except if it ruined your hard disk, perhaps). Your program crashed, and that is bad, but such errors are easy to debug and correct. But if you produce bad data and bad results, the problem may be much worse, and you may not even notice it, or only much later. That is why you must use pointers with extreme caution. Always meticulously check for uninitialized pointers.

*Accessing an uninitialized pointer is* undefined behaviour*. I am particularly fond of the term "nasal demons", from an entry in the comp.std.c Usenet newsgroup, where someone wrote: "When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose".*

## Stale pointers

Stale pointers are pointers that were once valid, but have gone bad. This can happen when the memory to which a pointer points is freed and reused.

One common cause of stale pointers is when memory is freed (disposed), but the pointer to it is still used after that moment. To prevent that, some programmers always set pointers to `nil`, after the memory is freed. They will add tests for nil, before they access memory. In other words, nil is used as some kind of flag to mark the pointer as invalid. This is one approach, but not always the best.

Another often seen error is having more than one pointer to some piece of memory, and then freeing it using one of these pointers. Even if you `nil` that pointer, the other pointers will still contain the address of that piece of freed memory. If you are lucky, you get an "Invalid pointer" error, but what should happen is undefined.

A third, similar problem is pointing a pointer to volatile data, i.e. data that may disappear anytime. One big mistake is for instance a function returning a pointer to local data of a routine. Once the routine has ended, that data is gone, it does not exist anymore. A classic (but rather stupid, I know) example:

```
function VersionData: PAnsiChar;
var
  V: array[0..11] of AnsiChar;
begin
  CalculateVersion(V);
  Result := V;
end;
```

`V` will be placed on the **processor stack**. This is a piece of memory that is reused for local variables and parameters for every running function, and which also contains sensitive data like return addresses for function calls. The result now points to `V` (`PAnsiChar` can point to an array directly, see the article I mentioned). As soon as `VersionData` ends, the stack is modified by the next routine that runs, so whatever was calculated by `CalculateVersion` is gone again, and the pointer now points to the new contents of that particular part of the stack.

A similar problem is pointing a PChar to a string, but that is also discussed in the article about PChars. Pointing to an element of a dynamic array is of the same class, since the dynamic array may be moved entirely, if the array got too small and SetLength was used.

## Using the wrong base type

The fact that pointers can point at any memory location, and that two pointers of a different type can point to the same location, means that you can access the same memory location in different ways. Using a pointer to Byte (^Byte), you could change the individual bytes of an integer or some other type.

But you can also overwrite or overread. For instance, if you access a location which is only supposed to contain a Byte with a pointer to Integer, you could overwrite 4 bytes, not only the byte that is reserved, but also the 3 consecutive locations, since the compiler would treat those 4 as an integer. Also, if you read from a byte location, you might read too much:

```
var
  PI: PInteger;
  I, J: Integer;
  B: Byte;
begin
  PI := PInteger(@B);
  I := PI^;
  J := B;
end;
```

`J` will have the proper value, because the compiler will add code to expand the single byte to a (4-byte) `Integer` by padding the integer with zero bytes. But `I` will not. It will contain the byte, and the 3 bytes following it, together forming some undefined value.

Pointers also allow you to set a variable’s value without assigning to the variable itself. This can be the cause of a lot of frustration during debugging. You know that a variable contains a wrong value, but can’t find the spot in code where you assign that value to the variable, because it was set through a pointer.

## Owners and orphans

Pointers can not only have a different base type, they can also have different ownership semantics. If you allocate memory, using `New` or `GetMem` or one of the other routines for more specialised tasks, you are the **owner** of that memory. It is best, if you want to hold on to that memory, to tuck the pointer into a safe place. The pointer is your only access to the memory, and if the address gets lost, you have no way to access or free the memory anymore. One rule is that who allocates memory should also free it, so it is your responsibility to take care it is always possible. Well designed programs always consider this.

*It is very important to understand ownership. Who owns memory must always free it. You can delegate this task, but you must ensure that it is done properly.*

One common error is to use a pointer to allocate memory, and then to re-use the pointer, to allocate some more memory, or to point it to some other memory. The pointer, which first contained the address of the first chunk, will now contain the address of the newest chunk, and the old address is forever lost. There is no useful way to ever find back where that memory was allocated. The memory is **orphaned**. No one can access it, no one can take care of it anymore. It will present a so called **memory leak**.

Here is a simple example taken (with permission of the author) from Borland’s newsgroups:

```
var
```

```
  bitdata: array of Byte;
  pbBitmap: Pointer;
begin
  SetLength(bitdata, nBufSize);
  GetMem(pbBitmap, nBufSize);
  pbBitmap := Addr(bitdata);
  VbMediaGetCurrentFrame(VBDev, @bmpinfo.bmiHeader, @pbBitmap, nBufSize);
```

Actually, this code does quite a few confusing things. `SetLength` allocates bytes for `bitdata`. For some reason, the programmer then uses `GetMem` to allocate the same amount of bytes for `pbBitmap`. But then he immediately *sets pbBitmap to another address*, which makes the memory he just allocated with `GetMem` unreachable for any code (`pbBitmap` was the only way to reach it, and now doesn't point to it anymore). In other words, we have a memory leak.

In fact, there are a few more errors. `bitdata` is a dynamic array, and taking the address of `bitdata` only takes the address of a pointer, instead of the address of the first byte of the buffer (see further below, *dynamic arrays*). Also, since `pbBitmap` is already a pointer, it is wrong to use the @ operator on it, in the function call.

Better code would have been:

```
var
  bitdata: array of Byte;
  pbBitmap: Pointer;
begin
  if nBufSize > 0 then
  begin
    SetLength(bitdata, nBufSize);
    pbBitmap := Addr(bitdata[0]);
    VbMediaGetCurrentFrame(VBDev, @bmpinfo.bmiHeader, pbBitmap, nBufSize);
  end;
```

or even:

```
var
  bitdata: array of Byte;
begin
  if nBufSize > 0 then
  begin
    SetLength(bitdata, nBufSize);
    VbMediaGetCurrentFrame(VBDev, @bmpinfo.bmiHeader, @bitdata[0], nBufSize);
  end;
```

This may seem a trivial problem, but in more complex code, this can easily happen.

Note that a pointer does not have to own memory. Pointers are often used to iterate over arrays (see below), or to access parts of a structure. If you did not allocate memory with them, there is no need to hold on to them. They are only used as temporary throw-away variables.

## Pointer arithmetic and arrays

> *You can either have software quality or you can have pointer arithmetic, but you cannot have both at the same time.* — Bertrand Meyer

Delphi allows some simple manipulations of a pointer. Of course you can assign to them, and compare them for equality (`if P1 = P2 then`) or inequality, but you can also increment and decrement them, using `Inc` and `Dec`. The neat thing is that these increments and decrements are *scaled by the size of the base type of the pointer*. An example (note that I set the pointer to a fake address. As long as I don't access anything with it, nothing bad will happen):

```
program PointerArithmetic;

{$APPTYPE CONSOLE}

uses
  SysUtils;

procedure WritePointer(P: PDouble);
begin
  Writeln(Format('%8p', [P]));
end;

var
  P: PDouble;

begin
```

```
  P := Pointer($50000);
  WritePointer(P);
  Inc(P);    // 00050008 = 00050000 + 1 * SizeOf(Double)
  WritePointer(P);
  Inc(P, 6); // 00050038 = 00050000 + 7 * SizeOf(Double)
  WritePointer(P);
  Dec(P, 4); // 00050018 = 00050000 + 3 * SizeOf(Double)
  WritePointer(P);
  Readln;
end.
```

The output is:

```
00050000
00050008
00050038
00050018
```

The utility of this is to provide sequential access to arrays of such types. Since (one-dimensional) arrays contain consecutive items of the same type — i.e. if one element is at address `N`, then the next element is at address `N + SizeOf(element)` —, it makes sense to use this to access items of an array in a loop. You start with the base address of the array, at which you can access the first element. In the next iteration of the loop, you increment the pointer to access the next element of the array, and so on, and so forth:

```
program IterateArray;

{$APPTYPE CONSOLE}

var
  Fractions: array[1..8] of Double;
  I: Integer;
  PD: ^Double;

begin
  // Fill the array with random values.
  Randomize;
  for I := Low(Fractions) to High(Fractions) do
    Fractions[I] := 100.0 * Random;

  // Access using pointer.
  PD := @Fractions[Low(Fractions)];
  for I := Low(Fractions) to High(Fractions) do
  begin
    Write(PD^:9:5);
    Inc(PD); // Point to next item
  end;
  Writeln;

  // Conventional access, using index.
  for I := Low(Fractions) to High(Fractions) do
    Write(Fractions[I]:9:5);
  Writeln;

  Readln;
end.
```

Incrementing a pointer is, at least on older processors, probably slightly faster than multiplying the index with the size of the base type and adding that to the base address of the array for each iteration.

In reality, the effect of doing it this way is not nearly as big as you might expect. First, modern processors have special ways of addressing the most common cases using an index, so there is no need to update the pointer too. Second, the compiler will generally optimize indexed access into the pointer using version anyway, if this is more beneficial. And in the above, the gain found by using a slightly more optimized access is largely overshadowed by the time it takes to perform the Write().

As you can see in the program above, you can easily forget to increment the pointer inside the loop. And you must either use `for-to-do` anyway, or use another way or counter to terminate the loop (which you must then also decrement and compare manually). IOW, the code using the pointer is generally much harder to maintain. Since it is not faster anyway, except perhaps in a very tight loop, I would be very wary of using that kind of access in Delphi. Only do this if you have profiled your code and found pointer access to be beneficial and necessary.

## Pointers to arrays

But sometimes you only have a pointer to access memory. Windows API functions often return data in buffers, which then contain arrays of a certain size. Even then, it is probably easier to cast the buffer to a pointer to an array than to use Inc or Dec. An example:

```
type
  PIntegerArray = ^TIntegerArray;
  TIntegerArray = array[0..65535] of Integer;
var
  Buffer: array of Integer;
  PInt: PInteger;
  PArr: PIntegerArray;
  ...
  // Using pointer arithmetic:
  PInt := @Buffer[0];
  for I := 0 to Count - 1 do
  begin
    Writeln(PInt^);
    Inc(PInt);
  end;

  // Using array pointer and indexing:
  PArr := PIntegerArray(@Buffer[0]);
  for I := 0 to Count - 1 do
    Writeln(PArr^[I]);
  ...
end;
```

## Delphi 2009

In Delphi 2009 and later, pointer arithmetic, as usable for the `PChar` type (and `PAnsiChar` and `PWideChar`), is now also possible for other pointer types. When and where this is possible is governed by the new `$POINTERMATH` compiler directive.

Pointer arithmetic is generally switched off, but it can be switched on for a piece of code using `{$POINTERMATH ON}`, and off again using `{$POINTERMATH OFF}`. For pointer types compiled with pointer arithmetic (pointer math) turned on, pointer arithmetic is generally possible.

Currently, besides `PChar`, `PAnsiChar` and `PWideChar`, the only other type for which pointer arithmetic is enabled by default is the `PByte` type. But switching it on for, say, `PInteger` would simplify the code above considerably:

```
{$POINTERMATH ON}
var
  Buffer: array of Integer;
  PInt: PInteger;
  ...
  // Using new pointer arithmetic:
  PInt := @Buffer[0];
  for I := 0 to Count - 1 do
    Writeln(PInt[I]);
  ...
end;
{$POINTERMATH OFF}
```

So there is no need for the declaration of special `TIntegerArray` and `PIntegerArray` types to be able to access the type as an array anymore. Alternatively, instead of `PInt[I]`, the `(PInt + I)^` syntax could have been used, with the same result.

*Apparently, in* Delphi 2009*, the new pointer arithmetic doesn't work as intended for pointers to* generic *types yet. Whatever type the parametric type is instantiated as, indices are not scaled by* `SizeOf(T)`*, as expected.*

## References

Many types in Delphi are in fact pointers, but pretend not to be. I like to call these types **references**. Examples are dynamic arrays, strings, objects and interfaces. These types are all pointers behind the scenes, but with some extra semantics and often also some hidden content.

- Dynamic arrays
- Multi-dimensional dynamic arrays
- Strings
- Objects
- Turbo Pascal objects
- Interfaces
- Reference parameters
- Untyped parameters

What distinguishes references from pointers is:

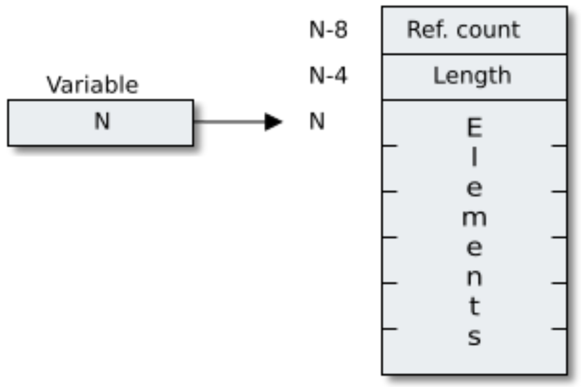
- **References are immutable.** You can not increment or decrement a reference. References point to certain structures, but never into them, like for instance the pointers that point into an array, in the examples above.
- **References do not use pointer syntax.** This hides that they *are* in fact pointers, and makes them hard to understand for many, who do not know this, and therefore do things with them they would better not do.

*Do not confuse such references with C++'s* reference types*. These are different in many ways.*

## Dynamic arrays

Before Delphi 4, dynamic arrays were not a feature of the language, but they existed as a concept. A dynamic array is a block of memory that is allocated and managed via a pointer. The dynamic array can grow or shrink. This means in fact, that memory of the new required size is allocated, the contents of the old block that are to be preserved are copied over to the new block, the old block is freed, and the pointer is then set to point to the new block.

The dynamic array types (e.g. `array of Integer`) in Delphi do the same. But the runtime library adds special code that manages each access and assignment. At the memory location **below** the address to which the pointer points, there are two more fields, the number of elements allocated, and the *reference count*.



If, as in the diagram above, `N` is the address in the dynamic array variable, then the *reference count* is at address `N-8`, and the number of allocated elements (the *length indicator*) at `N-4`. The first element is at address `N`.

For each added reference (e.g. assignment, parameter passing, etc.), the reference count is incremented, and for each removed reference (e.g. when a variable goes out of scope, or when an object containing a dynamic array reference is freed, or when a variable containing a dynamic array reference is re-assigned to nil or another array) it is decremented.

*Accessing dynamic arrays in low level routines like* `Move` *or* `FillChar`*, or other routines accessing entire arrays, like* `TStream.Write` *is often done wrongly. For a normal array (often this is called a* static *array, to distinguish it from a* dynamic *array), the variable is identical with the block of memory. For a dynamic array, this is not the case (see diagram). So a routine that wants to access the elements of the array as a block, should not reference the dynamic array variable,* *but the first element of the array instead.*

```
var
  Items: array of Integer;
  ...
  // Wrong: address of Items variable is passed
  MyStream.Write(Items, Length(Items) * SizeOf(Integer));
  ...
  // Correct: address of first element is passed
  MyStream.Write(Items[0], Length(Items) * SizeOf(Integer));
```

*For* static *arrays, the address of the first element is identical with the address of the array, so in other words, passing* `Items[0]` *also works if* `Items` *is a static array. If you* ***get used to always passing the first element of an array****, then you can't go wrong, no matter whether the array is dynamic or static.*

*Note that in the above,* `Stream.Write` *uses untyped* `var` *parameters, which are also references. They will be discussed below.*

## Multi-dimensional dynamic arrays

The above discusses one-dimensional dynamic arrays. But dynamic arrays can also be multi-dimensional. Well, at least syntactically, since in reality, they are not. A multidimensional dynamic array is in fact a one-dimensional array of pointers to other one-dimensional arrays.

Say, we have these declarations:

```
type
  TMultiIntegerArray = array of array of Integer;

var
  MyIntegers: TMultiIntegerArray;
```

Now that looks like a multi-dimensional array, and it can indeed be accessed like `MyIntegers[0, 3]`. But in fact, the declaration of the type should be read like this (I'm taking some liberties with syntax here):

```
type
  TMultiIntegerArray = array of (array of Integer);
```

Or, to make it a little more explicit, that is in fact the same as the following:

```
type
  TSingleIntegerArray = array of Integer;
  TMultiIntegerArray = array of TSingleIntegerArray;
```

As you can see, a *TMultiIntegerArray* is in fact a one-dimensional array of *TSingleIntegerArray* pointers. This means that a *TMultiIntegerArray* is not stored as one block of memory arranged in rows and columns, but that it is in fact a *ragged array*, i.e. each entry is simply a pointer to another array, and each of these sub-arrays can have a different size. So instead of

```
  SetLength(MyIntegers, 10, 20);
```

(which would allocate 10 `TSingleIntegerArray`s of 20 integers each, so this is, on the surface, a rectangular array), you can access and change each of the subarrays:

```
SetLength(MyIntegers, 10);
SetLength(MyIntegers[0], 40);
SetLength(MyIntegers[1], 31);
// etc...
```

## Strings

> *Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.*
> — Stan Kelly-Bootle

Strings are in many ways the same as dynamic arrays. They are also reference counted, they have a similar internal structure, with a reference count and a length indicator stored below the string data (at the same offsets).

The differences are in syntax and semantics. You can not set a string to `nil`, you set it to `''` (empty string) to clear it. Strings can also be constants (the reference count is -1, which is a special value for the runtime routines: they will not try to increment or decrement it, or free the string). The first element is at index `1`, unlike dynamic arrays, where the first element is always at index `0`.

More about strings can be found in my article about PChars and strings.

## Objects

In the desktop compilers, objects — or, more accurate, class instances — have no compiler generated lifetime management. Their internal structure is simple. Each class instance contains, at offset 0 (i.e. at the address to which the reference points) a pointer to the so-called VMT table. This is a table with a pointer for each virtual method of the class. At negative offsets from the table, a lot of other information about the class is present. I will not go into that in this article. There is one VMT table for each class (not for each object!).

*In the mobile compilers (e.g. for Android or iOS), objects do have a compiler generated lifetime management, called ARC. I will not dicsuss that here, except to say that the lifetime management is quite similar to that of interfaces.*

Classes that implement interfaces also have similar pointers to tables that contain pointers to the methods that implement the interface, one for each implemented interface. These tables also contain some extra info at negative offsets. At which offset into the object these pointers are stored depends on the fields already present in the ancestor class. The compiler knows this.

After the VMT pointer and any interface table pointers, the fields of the object are stored, just like in a record.

RTTI data and other info about a class is obtained by following the reference to the object, which also points to the VMT pointer, and then following that pointer to the VMT. The compiler then knows where to find the rest of the data, usually also through complicated structures containg pointers to other structures, sometimes even recursively.

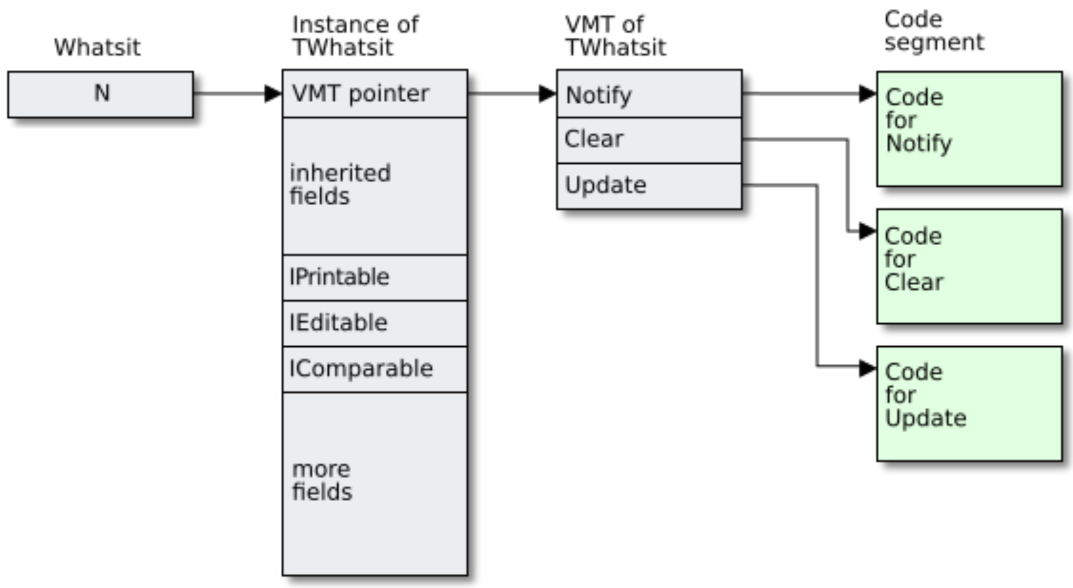
An example follows. Assume the following declaration:

```
type
  TWhatsit = class(TAncestor, IPrintable, IEditable, IComparable)
  public
    // other field and method declarations
    procedure Notify(Aspect: TAspect); override;
    procedure Clear; override;
    procedure Edit;
    procedure ClearLine(Line: Integer);
    function Update(Region: Integer): Boolean; virtual;
    // etc...
  end;

  var
    Whatsit: TWhatsit;

  begin
    Whatsit := TWhatsit.Create;
```

Then the object layout will be something like:



## Turbo Pascal objects

Although they were deprecated at the time Delphi 1.0 was released, I want to mention old Turbo-Pascal-style `object` types (declared with the `object` keyword, instead of the `class` keyword). There are still some libraries that use them.

These TP object types are more or less like records with methods, but with inheritance and properties (properties were added in Delphi, Turbo Pascal did not have them). The layout is like that for a record. There is no inherent VMT pointer at the beginning of the object. Only if an object has virtual or dynamic methods, a new (hidden) field for the VMT pointer is added (this VMT pointer can be added in a descendant object, so it is not necessarily at offset 0).

Another difference with `class` types is that TP-style `object` types are ***not*** reference types and that they are not necessarily allocated on the heap. Like records, they can be used as value types, or they can be used with pointers. For these types, the `New` pseudo-procedure accepts a second parameter, a constructor call, and `Dispose` a destructor call. This type does not have something like `TObject` as a base and is much like the class and struct types in the C++ language. This also means that they are subject to object slicing when used as value types, like in C++.

A simple example:

```
type
  POldStyle = ^TOldStyle;
  TOldStyle = object
    ...
    constructor Init(Value: Integer); // typical name for a constructor in Turbo Pascal
    destructor Done; virtual;         // typical name for a destructor in Turbo Pascal
    procedure Add(Extra: Integer);
  end;

...

var
  TurboObj: POldStyle;
begin
  New(TurboObj, Init(17));
  TurboObj^.Add(33);
  ...
  Dispose(TurboObj, Done);
end.
```

*Although they have been deprecated for a long time already, old TP-style objects still work (but they may be a little buggy, and bugs in these deprecated types don't seem to be a priority for the Delphi dev-team), even in the 64 bit compilers. But I would not recommend their use unless you already have code that uses them. Either use records with methods, or proper class types. The documentation also says:* Object types are supported for backward compatibility only. Their use is not recommended.

*To find out more about these types, read the documentation, either in the built-in help, or online, in the Delphi docwiki.*

## Interfaces

Interfaces are in fact a collection of methods. Internally, they are pointers to pointers to an array of pointers to code. Assume we have the following declarations:

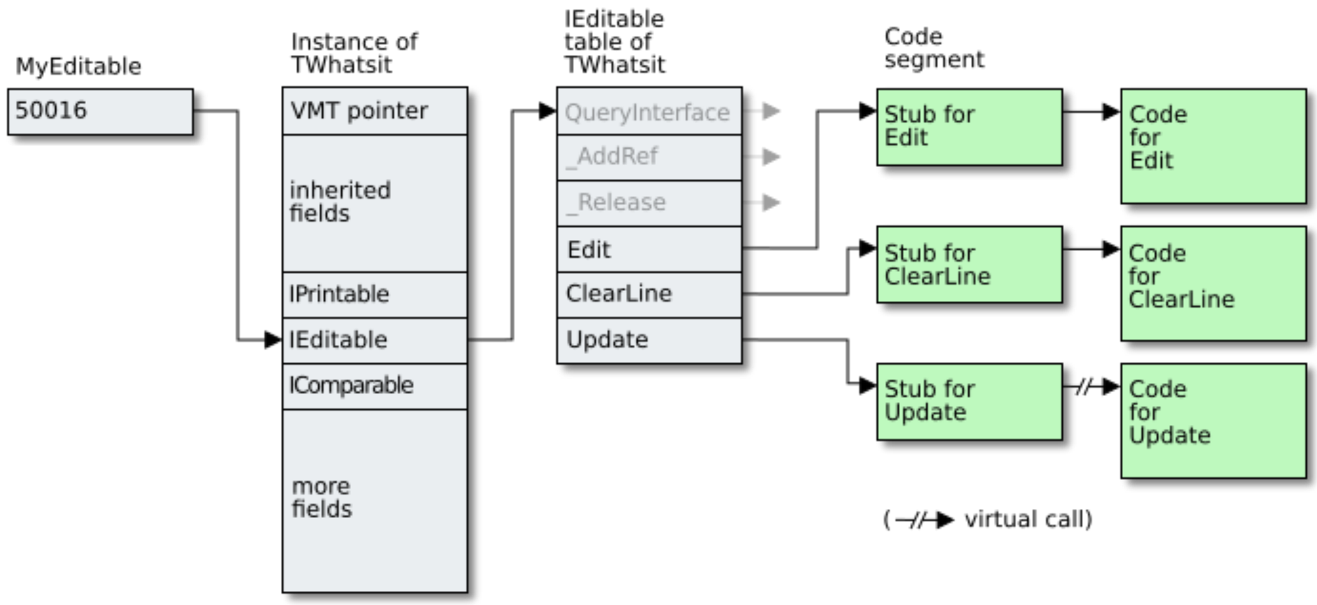
```
type
  IEditable = interface
    procedure Edit;
    procedure ClearLine(Line: Integer);
    function Update(Region: Integer): Boolean;
  end;

  TWhatsit = class(TAncestor, IPrintable, IEditable, IComparable)
  public
    procedure Notify(Aspect: TAspect); override;
    procedure Clear; override;
    procedure Edit;
    procedure ClearLine(Line: Integer);
    function Update(Region: Integer): Boolean; virtual;
    // etc...
  end;

var
  MyEditable: IEditable;

begin
  MyEditable := TWhatsit.Create;
```

Then the relationship between interface, implementing object, implementing class and methods looks like this:



The variable `MyEditable` points to the `IEditable` pointer in the object created by `TWhatsit.Create`. Note that `MyEditable` does not point to the start of the object, but to an offset into it. The `IEditable` pointer in the object then points to a table of pointers, one for each method in the interface. Each of the entries points to a piece of stub code. This piece of code adjusts the `Self` pointer (which is in fact the value of `MyEditable`) to point to the start of the object, by subtracting the offset of the `IEditable` pointer in the object from the passed pointer, and then calls the real method. There is a stub for each implementation of a method of each interface implemented by the class.

An example: Say the instance is at address 50000, and the pointer for the implementation of `IEditable` by `TWhatsit` is at offset 16 of each instance. Then `MyEditable` will contain 50016. The `IEditable` pointer at 50016 then points to the table for that interface as implemented in that class (say, at address 30000), and then that points to the stub (say, at address 60000). The stub will see the value (which is passed as `Self` parameter) 50016, subtract the offset 16 and get 50000. This is the address of the implementing object. The stub then calls that real method, passing 50000 as the `Self` parameter.

In the diagram, for clarity, I omitted the stubs for `QueryInterface`, `_AddRef` and `_Release`.

*Can you see why I like to use pencil and paper sometimes?* `;-)`

## Reference parameters

Reference parameters are often called `var` parameters, but also `out` parameters are reference parameters.

Reference parameters are parameters where not the value of the actual parameter is taken and passed to the routine, but its address instead. An example:

```
procedure SetBit(var Int: Integer; Bit: Integer);
begin
  Int := Int or (1 shl Bit);
end;
```

This is functionally equivalent to the following:

```
procedure SetBit(Int: PInteger; Bit: Integer);
begin
  Int^ := Int^ or (1 shl Bit);
end;
```

There are a few differences, though:

- You do not use pointer syntax. Using the name of the parameter automatically dereferences the parameter, IOW using the name of the parameter means accessing the target, not the pointer.
- Reference parameters can not be modified. Using the name of the parameter uses its target, you can't assign to the pointer, or increment or decrement it.
- You must pass something that has an address, i.e. has an actual memory location, unless you use a casting trick. So to a reference parameter of type Integer, you can't for instance pass `17`, `98765`, or `Abs(MyInteger)`. It must be a variable (this includes elements of arrays, fields of records or objects, etc.).
- Actual parameters must be of the same type as the declared parameters, i.e. you can not pass a `TEdit` if you declared the parameter as a `TObject`. To avoid this, you can only use *untyped reference parameters* instead (see below).

Syntactically, it may seem simpler to use reference parameters then to use pointer parameters. But one should be aware of some peculiarities. To pass pointers, one must increase the level of indirection by one. In other words, if you have a pointer `P` to an integer, to pass that, you must syntactically pass `P^`:

```
var
  Int: Integer;
  Ptr: PInteger;
  Arr: array of Integer;
begin
  // Initialisation of Int, Ptr and Arr not shown...
  SetBit(Ptr^, 3);    // Ptr is passed
  SetBit(Arr[2], 11); // @Arr[2] is passed
  SetBit(Int, 7);     // @Int is passed
```

## Untyped parameters

Untyped parameters are also reference parameters, but they can either be `var`, `const` or `out`. You can pass any type, which makes it easier to write routines that accept almost anything, of any size or type, but this means you must either have a way to pass the type with it, as another parameter, or the routine is thus, that the type doesn't matter. To access the parameter, you'll have to cast it.

Internally, untyped parameters are passed as pointers too. Two examples follow. The first is of a generic routine that fills any buffer with a range of bytes, i.e. the type of the variable passed in the buffer doesn't matter:

```
// Example of routine where type doesn't matter
procedure FillBytes(var Buffer; Count: Integer;
  Values: array of Byte);
var
  P: PByte;
  I: Integer;
  LenValues: Integer;
begin
  LenValues := Length(Values);
  if LenValues > 0 then
  begin
    P := @Buffer; // Treat buffer as array of byte.
    I := 0;
    while Count > 0 do
    begin
      P^ := Values[I];
      I := (I + 1) mod LenValues;
      Inc(P);
      Dec(Count);
    end;
  end;
end;
```

The second is a method of a `TIntegerList`, a descendant of a generic `TTypedList`:

```
function TIntegerList.Add(const Value): Integer;
begin
  Grow(1);
  Result := Count - 1;
  // FInternalArray: array of Integer;
  FInternalArray[Result] := Integer(Value);
end;
```

As you can see, to use the pointer, you must take the address of the parameter, although in fact the parameter is already the pointer you want. Again, the level of indirection is off by one.

To access the referenced target, you simply use it as for normal reference parameters, but you must cast to a type, so the compiler knows how to dereference the pointer.

I already mentioned the level of indirection. This can be seen if you want to initialize a dynamic array with `FillBytes`. To do that, you don't pass the variable, but the first element of the array. In fact, you can also pass the first element of a static array to achieve the same. So, if you are passing arrays to untyped reference parameters, IMO, your best choice is always to pass the first element, instead of the array, unless you really want to mess up the array pointer of a dynamic array.

## Procedural types

Procedural types are in fact variables that hold typed pointers to procedures or functions. There are several such types:

- Plain procedural types
- Method types (events)
- Anonymous methods

These are implemented quite differently, and one should not confuse these types. They are all reference types, although one **can** (and sometimes must) use the `@` operator with the first kind, plain procedural types.

### Plain procedural types

These reference global (i.e. non-method) procedures and functions. This is useful if you must call different functions in a similar context. For instance, the (non-generic) `TList.Sort` method (in `System.Classes`) is passed a `TListSortCompare` procedural type parameter which compares two items (passed as pointers) of the list. This allows different ways to compare the items, so the list can be sorted differently.

I will try to explain them with another example:

```
program ProceduralTypes;

{$APPTYPE CONSOLE}

uses
  SysUtils, Math;

type
  TOperation = function(Left, Right: Integer): Integer;

// Now use the type:
function MapOperation(const A, B: array of Integer; Operation: TOperation): TArray<Integer>;
var
  I: Integer;
begin
  SetLength(Result, Math.Min(Length(A), Length(B)));
  for I := 0 to High(Result) do
    Result[I] := Operation(A[I], B[I]); // Call the passed function
end;

// Utility function to print an array.
procedure PrintArray(const A: array of Integer);
var
  I: Integer;
begin
  for I in A do
    Write(I, ' ');
  Writeln;
end;

// The following functions will be mapped to the Operation parameter of the previous function.
procedure Addition(Left, Right: Integer): Integer;
begin
  Result := Left + Right;
end;

procedure Subtraction(Left, Right: Integer): Integer;
begin
  Result := Left - Right;
end;

var
  Adds, Subs: TArray<Integer>;

begin
  // Pass Addition as the Operation parameter: items will be added pairwise.
  Adds := MapOperation([1, 2, 3, 4], [5, 6, 7, 8], Addition);
  PrintArray(Adds);

  // Pass Subtraction as the Operation parameter: items will be subtracted pairwise.
  Subs := MapOperation([1, 2, 3, 4], [5, 6, 7, 8], Subtraction); // Subtracts the items in the arrays
  PrintArray(Subs);
end.
```

The output is, as expected:

```
6 8 10 12
-4 -4 -4 -4
```

The first line shows the additions of corresponding items in the two arrays (1+5, 2+6, 3+7, 4+8), each the result of passing them pairwise to the `Operation` parameter, which references the `Addition` function. In the second line, the results are subtractions of the same values (1−5, 2−6, 3−7, 4−8), because in the second call, the `Operation` parameter references the `Subtraction` function.

## Signature

You can only assign functions or procedures that have the same *signature* as the declared procedural type. The *signature* is the combination of parameter and return types. In the example, the signature is, in pseudo-code: `function(Integer, Integer): Integer`. The names of the function and the parameters do not matter. Only the combination of the types must be the same.

### Only global procedures and functions

*This is important and not knowing this is the cause of many mistakes I've seen:* **You can only assign global procedures or functions to plain procedural types**. You can **not** assign instance methods of objects, because these have a hidden `Self` parameter, which must be a valid object. You can, though, assign `static` methods; these do not have a `Self` parameter and are, internally, just global procedures or functions too.

### The @ operator

You can use the `@` operator on both sides of the assignment. This makes sense if the procedural type is used in an expression. The Delphi documentation gives a nice example:

```
var
   F, G: function: Integer;
   I: Integer;
   function SomeFunction: Integer;
     ...
   F := SomeFunction;      // assign SomeFunction to F
   I := F;
   ...
   if F = SomeFunction then
```

Because calling parameterless functions in Pascal and Delphi does not require parentheses (in other words, you don't have to use `F()` to call `F`), using procedural types can be ambiguous. The `if` clause above compares the Integer **results** of calling `F` and `SomeFunction`. if you want to know if `SomeFunction` was assigned to `F`, you must use the `@` operator:

```
   if @F = @SomeFunction then
```

If you want to know the address of the *variable* `F`, you should use a double `@`:

```
   Writeln(Format('%p', [@@F]));
```

*To avoid an ambiguous situation, if you really want to be sure you are calling the procedural type and comparing the Integer* results *of calling the functions, you can make the calls explicit using parentheses:*

```
  if F() = SomeFunction() then
```

## Method Pointers

Method pointers are often used for events. Events in the VCL or FireMonkey are in fact method pointer properties. One well known event type is for instance:

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
```

This is used for many of the VCL events, e.g. the `OnClick` event of a control. The syntax differs from plain procedural types by the `of object` part. One could say that `of object` is part of the signature: `procedure(TObject) of object`.

These types are not just pointers. They contain information about the address of the code *and* about the specific object on which they are called. Internally, a method pointer is realized as a `TMethod` record:

```
type
```

```
  PMethod = ^TMethod;
  TMethod = record
    Code, Data: Pointer;
  public
    class operator Equal(const Left, Right: TMethod): Boolean; inline;
    class operator NotEqual(const Left, Right: TMethod): Boolean; inline;
    class operator GreaterThan(const Left, Right: TMethod): Boolean; inline;
    class operator GreaterThanOrEqual(const Left, Right: TMethod): Boolean; inline;
    class operator LessThan(const Left, Right: TMethod): Boolean; inline;
    class operator LessThanOrEqual(const Left, Right: TMethod): Boolean; inline;
  end;
```

In older versions of Delphi, there are no `class operators`, so there they are just:

```
type
  PMethod = ^TMethod;
  TMethod = record
    Code, Data: Pointer;
  end;
```

The `Data` pointer points to the class instance (object) on which the method is called, and the `Code` pointer points to the code of the method. Using the type is like calling a plain procedural type:

```
procedure TButton.Click;
begin
  if Assigned(OnClick) then
    OnClick(Self);          // Self (the current button) is passed as the Sender parameter here.
end;
```

There is another, hidden parameter (like for all methods), which is taken from the `Data` part of the `TMethod` (for instance the current form). This is passed as the `Self` of the event handler.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Self.Caption := TButton(Sender).Caption;
end;
```

`Button1Click` has two parameters: the hidden `Self` parameter (the instance of `TForm1`) and the explicit `Sender` parameter (the button that was clicked). So the above sets the `Caption` of the form to the `Caption` of the button that was clicked.

### The @ operator

Like for plain procedural types, you can use the `@` operator to compare two method types. This does not just compare the `Code` parts, but also the `Data` parts, so in this case `@` does not just return a pointer, it returns a `TMethod`.

### Static methods

Static methods of a class do not have an implicit `Self` parameter. They are, except for the scope, the same as plain global functions or procedures. They can only be assigned to plain procedural types, not to method types.

## Anonymous methods

These are, by far, the most complicated but also the most powerful of the procedural types.

They are declared as follows:

```
type
  TAnonymousOperation = reference to function(Left, Right: Integer): Integer;
```

On the surface, they are like plain procedural types. But they are more powerful than that. They can capture every part of the context in which they are defined that they use.

This is an article about pointers, so I will not explain more and just refer to the built-in help or the Delphi docwiki once again. It contains some nice examples, similar to the code about plain procedural types above.

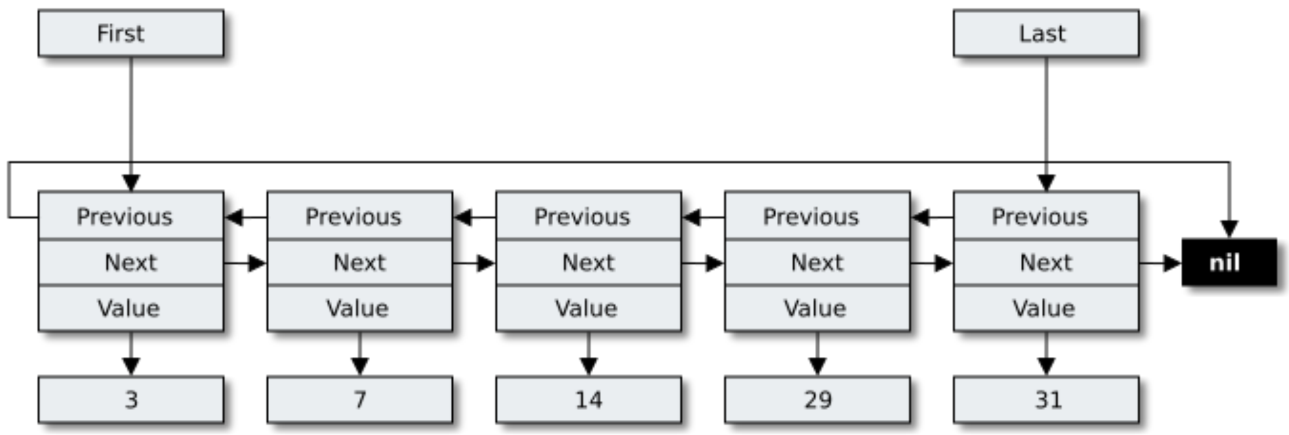
### Implementation

That these are not just simple pointers should be clear: something must keep (*capture*) all the variables the anonymous method refers to (for instance the `y` parameter of the outer function in the first example in the docwiki article ). Under the surface, an anonymous method is implemented as a class that has member fields for all the variables it captures. To manage the lifetime, the class implements an interface, and the interface reference is what is stored in the anonymous method variable. The interface has an `Invoke` method with the same signature as the declared anonymous method. This is the method that is actually called when the anonymous method is invoked.

## Data structures

Pointers are used extensively in data structures like linked lists, all kinds of trees and hierarchies, etc. I will not discuss these here. Suffice it to say that such advanced structures could not be done without pointers or references, not even in languages which officially don't even use pointers, like Java (well, as far as I know). If you really want to know more about such structures, you'll have to read one of the many textbooks on that subject.

I will give one simple example diagram of a data structure which relies heavily on pointer use, a linked list:



If you have such structures, it usually pays out to encapsulate the inner workings in a class, so that your use of pointers can be reduced to the implementation of the class, and they don't have to show up in the public interface of it. Pointers are powerful, but also hard to use, and if you can avoid using them, then do.

## Conclusion

I tried to give you my view on pointers. There are other approaches, but IMO, the one using diagrams (no matter how simple) with arrows is a good way to understand complex pointer problems, like the one above, or to understand how interface variables, objects, classes and code are linked. This does not mean that I draw diagrams of every simple problem. Only of the more complex ones, using pointers.

What this article tries to show is that pointers are everywhere, even if you don't always see them. That is not a reason to get paranoid, but the understanding of pointers as the underlying mechanism is, in my opinion, a prerequisite to avoiding a lot of mistakes.

I hope I was able to give you some useful tips. I'm sure that this article is incomplete, and I'd love suggestions on improvement. Just e-mail me.

*Rudy Velthuis*

---

### Standard Disclaimer for External Links

## Disclaimer and Copyright

Last update: Feb. 20, 2019

Back to top