Blitz2D Newbies: Definitive Guide to Types

by MutteringGoblin

Types are probably the hardest thing to understand about Blitz Basic. If you're using types for the first time, you've probably got an uneasy feeling that you don't quite know what you're dealing with. You might even be avoiding types completely because you don't understand them. In this article I'm going to explain some background information about types. Once you know what's going on behind the scenes, hopefully you'll feel more confident using them. I decided not to make a complete game to go with the tutorial, because it would mean explaining a lot of stuff that's not directly related to types (there are already plenty of tutorials explaining other aspects of Blitz).

As I've said, this is a newbie tutorial, but it does assume you already understand variables, and a bit about loops and control statements (For.. Next, If.. Then etc). If you don't know a variable from a vegetable, stop now and go back to Krylar's variables tutorial. :p

Okay first things first - what's a type?

A 'basic' data type can be one of three things: an integer (whole number), a floating point number (fraction) or a string (array of text). If you've had Blitz for more than five minutes, you've probably played around with basic data types. Here's a quick reminder of how to use them:

```
i% = 2525 ; create an integer called 'i', initialise it with the number 2525
f# = 1.75 ; create a float called 'f', initialise it with the number 1.75
s$ = "hi" ; create a string called 's', initialise it with the word "hi"
Print i
Print f
Print s
WaitKey
End ; end the program, destroy all variables
```

Easy, no?

Using the **Type** keyword, you can combine basic data types to describe something more complicated (that's why types are sometimes called 'custom types'). Like functions, types are a powerful way to customise a programming language. They can be used to represent almost anything in the real world; buildings, animals - even abstract things like timers, or the points in a journey. This is what makes custom type objects so much more useful than individual variables.

You probably know how to declare a type by now - there are examples all over the place - so I'll just go over it quickly. Let's assume you want to store some information about an Alien. You need to keep track of the following things:

1. What species of alien it is.
2. Where it is located on the screen.
3. How much health it has left.
4. The direction in which it's currently travelling.
5. How fast it can move.

Since these properties are shared by all aliens, it makes sense to combine them in a declaration which says "All aliens have this stuff in common?.

```
Type Alien
        Field species$
        Field xpos,ypos
        Field health
        Field direction
        Field speed#
End Type
```

The keyword 'Type' is followed by a name of your choice (?Alien?, ?Map?, ?EditBox? or whatever the type happens to represent...) and then a list of 'Fields', which declare the properties of the type. Each field has a unique name. Notice I put type tags after 'species$' and 'speed#' to tell Blitz that they represent a string of text and a floating point number. Anything without a type tag is assumed to be an ordinary integer.

Once you've told Blitz what properties an Alien has, you can create any number of Alien *objects* that share these properties.

So what?s an object?

You can think of an object as a 'package' of data, based on a type you defined earlier in the program. If you take a few variables which are related in some way, and store them together in the computer's memory, you have an object. Objects can be any size, and they can contain any combination of basic data types.

The code involved in creating a Blitz object looks confusing because it's a mixture of BASIC and C++. In this respect Blitz is harder than other BASIC languges, where types act just like normal variables. (Blitz uses a special kind of object called ?dynamic objects?).

Creating objects - pointers and the 'New' keyword

To understand how objects are created, you have to understand pointers.

A pointer is a special kind of variable which holds the address of another variable. The 'address' is just the place in the computer's memory where the variable lives. In Blitz, all pointers point to objects - you can't make a pointer to an integer or floating point variable. With objects, everything you do is accomplished through pointers.

So why not just use ordinary variables instead of pointers?

There are two reasons. Firstly, pointers have a special ability - they can point to anything they like. This ability makes pointers much more versatile than ordinary variables, and it's also the driving force behind the infamous 'For.. Each.. Next' loop (more on that later).

The second reason for using pointers is that they make programs run faster. Remember an object can be any size. Some objects are huge, with dozens of variables inside them. A pointer is always just a few bytes, regardless of what kind of object it points to. You can save a lot of memory and CPU time by copying pointers instead of copying the objects they point to.

One last thing - a pointer can be a dead end, ie. not pointing to any object. This kind of pointer is called a 'null pointer'. Null means 'nothing'.

Let's create an empty Alien pointer.

```
Local MyPointer.Alien
```

I've given it the name 'MyPointer' to show what kind of variable it is, but you can call your pointers anything you like. The important thing to note is the dot, followed by the type name 'Alien'. The dot indicates a pointer type, and the word 'Alien' tells Blitz what kind of pointer it is. Provided you have a definition of 'Alien' somewhere in your code, this should compile with no errors.

Confused? Okay, let's compare the Alien pointer to a string variable...

```
Local MyString$
```

They're doing exactly the same job. A type name with a dot before it denotes a pointer, just like a dollar denotes a string. It's difficult to get your head around because they look different, but essentially, a dollar$ sign does the same job as a hash# or a dot Alien. It just tells Blitz what you intend to use the variable for.

Now it's time to create an Alien object. You do this using the **New** keyword. The next piece of code is a bit confusing so I'll try and pick it apart as thoroughly as I can.

```
MyPointer.Alien = New Alien
```

This is why everyone avoids types when they first start using Blitz! To make things easier, you can ignore everything before the equals sign for a minute. We'll concentrate on the last part, 'New Alien', and what it does. This is sensible anyway, because Blitz executes the line from right to left.

'New' is an instruction to the computer, telling it to allocate memory for a new object. The actual amount of memory allocated will depend on the type of object. An Alien object needs about 24 bytes.

When the computer sees a New command it does several things:

1. It looks at the type name (in this case 'Alien') and finds out how much memory that particular type of object needs.
2. It reserves the right amount of memory for the new object.
3. It 'zeroes out' the object's fields. That means setting every value in the object to 0.
4. It ?returns? the memory address of the new object.

Now go back to the first part of the statement, 'MyPointer.Alien'. This is an Alien pointer declaration. So what do pointers have to do with creating objects?

Well, the New command does one last thing before it's job is complete. It returns the address of the object it just created. In ordinary English that means it copies the address into whatever pointer you provided. This ensures you have immediate access to your new Alien object.

If you're confused, try reading the line of code from right to left. First the New command does its thing, and spits out the address of the new Alien. Then the equals sign takes that address, and copies it into the empty Alien pointer.

So now our pointer has an Alien to call its own. Of course the Alien is empty. It has no name, no coordinates, no speed... so it's time to fill the object with information. With a basic variable you would do it like this:

```
speed# = 2.5
```

But remember we're not dealing with an Alien - we have the address of the Alien. Anything we do to the object has to be done indirectly, through its pointer. This is called indirection. In Blitz it ?s very easy; in fact it happens automatically...

```
MyPointer\Species$ = "purple tentacled"
```

That's all there is to it. To change the contents of an object, you apply a left slash to the object's pointer, and add the name of the field you want to access. What the above line really means is: *Find the object pointed at by MyPointer, and change its Species$ field to "purple tentacled"*. You can fill up all the other fields in exactly the same way.

NOTE: You don't have to use the type tag '.Alien' on the end of all your Alien pointers, except when you first declare them. But you can always use them if you like, to make the code clearer. With a type tag, the line we just saw would look like this:

```
MyPointer.Alien\Species$ = "purple tentacled"
```

Working with multiple objects

Here's where things get interesting. I'm going to create 3 separate Alien objects to show you how Blitz deals with lists and loops, and other Type commands like Insert, Before, After, First, Last and Delete.

The next program uses all the stuff we've talked about so far - types, pointers, indirection and the New command.

```
; Direction Constants
Const U = 0 ; Up
Const L = 1 ; Left
Const R = 2 ; Right
Const D = 3 ; Down


Type Alien
        Field species$
        Field xpos,ypos
        Field health
        Field direction
        Field speed#
End Type

;------------------------------------
MyPointer.Alien = New Alien
MyPointer\Species$ = "purple tentacled"
MyPointer\xpos = 0
MyPointer\ypos = 0
MyPointer\health = 100
MyPointer\direction = D
MyPointer\speed# = 1.0
;------------------------------------
MyPointer.Alien = New Alien
MyPointer\Species$ = "green blob"
MyPointer\xpos = 256
MyPointer\ypos = 256
MyPointer\health = 100
MyPointer\direction = R
MyPointer\speed# = 1.0
;------------------------------------
MyPointer.Alien = New Alien
MyPointer\Species$ = "space platypus"
MyPointer\xpos = 512
MyPointer\ypos = 64
MyPointer\health = 100
MyPointer\direction = R
MyPointer\speed# = 2.0
;------------------------------------
End
```

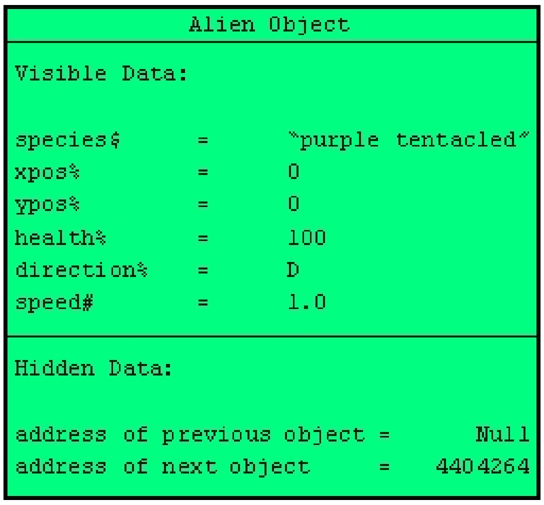
Notice how I used a single Alien pointer to create all three Alien objects. This is an example of how pointers can change their value. The pointer holds the address of each object for long enough to fill its fields, then it moves on to a new object.

When the code has been executed, the pointer contains the address of the last Alien object that was created, the "Space Platypus".

NOTE: When you create a new object with a pointer that is already in use, *the existing object is not destroyed*. The pointer moves on, but the old object still exists somewhere in the computer's memory. In C++, it's easy to create a 'hanging' object, an object that can't be accessed because you've lost its address. Luckily, you can't make this mistake in Blitz. :)

Here are some diagrams to show what happens in the computer's memory when you run the Alien program.

Memory location 4404224:



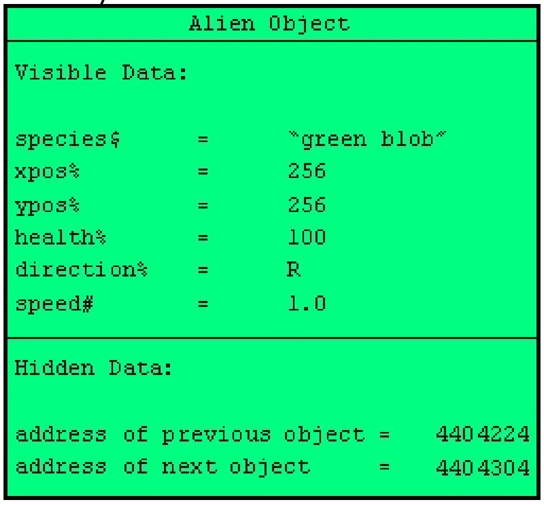
```
                Alien Object
Visible Data:

species$      =      "purple tentacled"
xpos%         =      0
ypos%         =      0
health%       =      100
direction%    =      D
speed#        =      1.0

Hidden Data:

address of previous object =      Null
address of next object     =   4404264
```

Memory location 4404264:



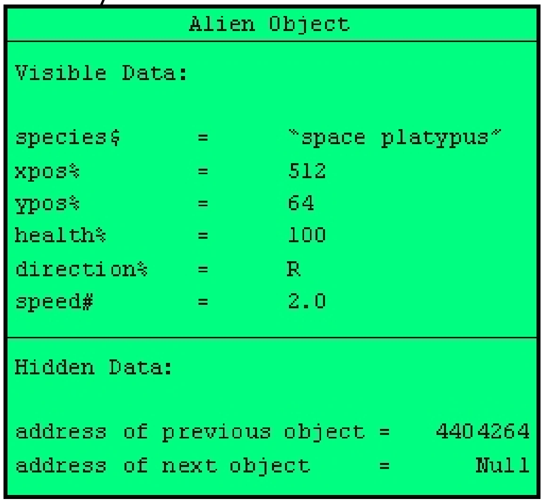
```
                Alien Object
Visible Data:

species$      =      "green blob"
xpos%         =      256
ypos%         =      256
health%       =      100
direction%    =      R
speed#        =      1.0

Hidden Data:

address of previous object =   4404224
address of next object     =   4404304
```

Memory location 4404304:



```
                Alien Object
Visible Data:

species$      =      "space platypus"
xpos%         =      512
ypos%         =      64
health%       =      100
direction%    =      R
speed#        =      2.0

Hidden Data:

address of previous object =   4404264
address of next object     =      Null
```

Blitz Basic stores objects as 'linked lists'. You don't actually have to worry about linked lists in Blitz - they happen automatically. But it's useful to learn a bit about them, because understanding linked lists will make types seem less mysterious.

What's a linked list?

Linked lists are one of the big advantages of using custom types. The thinking goes like this: because an object can be any size, and have any number of fields, you can use a couple of fields to store the addresses of other objects.

You'll see on the diagrams that every object has two 'hidden' fields, which are pointers to other objects. This is called a 'doubly linked list', because it's linked in two directions.

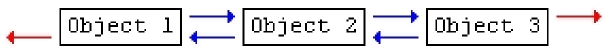
Every time you declare a new Type, Blitz adds the two hidden pointer fields. They're not like the pointers you use in your programs, because there's no way to access them directly. But, like bacteria and Tony Blair's conscience, you know they're there. (Okay that was a bad example...)

Whenever you make a new object, it's automatically added to the end of the list. Blitz hides these kind of details so you don't have to worry about them.

So why are linked lists good? Three reasons...

1. They're fast.
2. They let you keep track of your objects. When all objects of a type are chained together, there's no danger of any objects going missing.
3. They're convenient. To update all your objects, you just need to find the first object in the list. By following the pointers in each object, you'll eventually come to the end of the list, visiting all other objects along the way.

Here's another diagram showing the relationship between the three Alien objects. The blue arrows show the links between objects, and the red arrows indicate null pointers, which point to nothing.



If you look closely at that diagram and the previous ones you'll get a good idea of what's going on.

Any list of three or more objects is a complete list. There's the middle object ("green blob") which has valid pointers in both directions, and the two objects on either end which each have one valid pointer, and one null pointer.

Looping through objects with a 'For.. Each.. Next' loop

The 'For.. Each.. Next' loop is one of Blitz's more confusing features. It lets you use a single pointer to make changes to an entire collection of objects. It looks like this:

```
For A.Alien = Each Alien
        Print A\Species$
Next
WaitKey
```

The thing that throws people is the 'Each' part. How can one Alien equal each Alien? It doesn't make a lot of sense. Until you remember that 'A.Alien' is an Alien pointer, and pointers can change their value. Let's rewrite that code to make it more obvious...

```
For pointer.Alien = Each Alien
        Print pointer\Species$
Next
WaitKey
```

Now it's easier to understand. When the For Each Next loop executes, the sequence of events goes like this...

1. Find the address of the first Alien object
2. Copy that object's address into our 'pointer' variable.
3. Print the species of the Alien object.
4. Check the hidden 'next' pointer in the object.
5. If the hidden pointer is valid, go back to stage 2. Otherwise end the loop.

Add this loop to the Alien program and run it. As if by magic you'll see the names of all three aliens displayed on the screen.

Other type commands: 'Insert', 'Before', 'After', 'First', 'Last' and 'Delete'

We're going to use a For loop to find a particular Alien in our list, the "green blob". Once we've found green blob, we'll make a copy of his address so we can do stuff with him later.

```
Global GreenBlob.Alien ; reserve a pointer for the alien

For pointer.Alien = Each Alien
        If pointer\Species$ = "green blob" Then GreenBlob = pointer
Next
```

The 'If' statement checks each Alien's Species$ field. If it matches the string "green blob", the address of the Alien is copied into the GreenBlob pointer. To demonstrate the other Blitz type commands, I'm going to re-write the Alien program, giving a unique pointer to every Alien.

```
; Direction Constants
Const U = 0 ; Up
Const L = 1 ; Left
Const R = 2 ; Right
Const D = 3 ; Down

Type Alien
        Field species$
        Field xpos,ypos
        Field health
        Field direction
        Field speed#
End Type

;-------------------------------------
Global PurpleTentacled.Alien = New Alien ; *mumble*
PurpleTentacled\Species$ = "purple tentacled"
PurpleTentacled\xpos = 0
PurpleTentacled\ypos = 0
PurpleTentacled\health = 100
PurpleTentacled\direction = DOWN
PurpleTentacled\speed# = 1.0
;-------------------------------------
Global GreenBlob.Alien = New Alien ; *cough*
GreenBlob\Species$ = "green blob"
GreenBlob\xpos = 256
GreenBlob\ypos = 256
GreenBlob\health = 100
GreenBlob\direction = Right
GreenBlob\speed# = 1.0
;-------------------------------------
Global SpacePlatypus.Alien = New Alien ; *splutter*
SpacePlatypus\Species$ = "space platypus"
SpacePlatypus\xpos = 512
SpacePlatypus\ypos = 64
SpacePlatypus\health = 100
SpacePlatypus\direction = Right
SpacePlatypus\speed# = 2.0
;-------------------------------------
End
```

Important: You wouldn't normally give each alien its own pointer. Pointers can change their value, so you could end up getting very confused when your GreenBlob pointer points to a SpacePlatypus. I'm only doing it here to help explain the rest of the type commands.

Now you have three Alien pointers, you can use other Blitz commands like **Insert** and **After** to rearrange the list.

```
Insert GreenBlob After SpacePlatypus
```

Now the GreenBlob is at the end, and the SpacePlatypus is in the middle. Blitz automatically updates the hidden pointers in all affected objects. Nothing gets moved around in the computer's memory - only the values of the pointers are changed, giving the illusion that the list has been 'shuffled'. To show what has happened, put a For Each Next loop after the Insert command, to print out the Aliens' species. The **Before** command works in a similar way to the After command.

One thing to beware of - if you use the Before command on the first object in a list, or the After command on the last object, they will both return a null pointer (because obviously there's nothing before the beginning, or after the end). Always check for null pointers before trying to access an object's fields. If you don't, Blitz will eventually give you the dreaded 'Object does not exist' message.

There's another use for Before and After:

```
MyPointer.Alien = Before SpacePlatypus
Print MyPointer\Species$
```

Here, Blitz finds the object that comes before SpacePlatypus in the list, and copies its address into MyPointer. You can string these commands together, like this:

```
MyPointer.Alien = After After PurpleTentacled
Print MyPointer\Species$
```

This returns the object *after* the object after PurpleTentacled. As we've only made three objects, that will be the last object in the list.

The **First** and **Last** commands are self-explanatory. They return the address of the first object you created, and the last one (unless you've moved objects around with the Insert command). You can use First along with After and Null to cycle through objects. It's like doing a 'For Each Next' loop manually.

```
MyPointer.Alien = First Alien ; grab the first Alien's address

While MyPointer <> Null ; while MyPointer is a valid pointer
        Print MyPointer\Species$
        MyPointer = After MyPointer ; move forward through the list
Wend
```

The final type command is **Delete**. It frees the memory taken up by an object. You should Delete objects whenever they outlive their use, especially if you're creating new objects all the time. To delete an object, just apply the Delete command to its pointer. Delete is nearly always used after some kind of conditional check, like this:

```
If MyPointer\xpos >= GraphicsWidth() + 32
        Delete MyPointer
EndIf
```

Here the object is deleted if its image has disappeared off the edge of the screen. Note that you are deleting the object, *not* the pointer. The pointer becomes null, but you can still carry on using it if you set it to point to a new object.

To quickly delete all objects of a particular type, you can use the Each command.

```
Delete Each Alien ; Delete the entire list of Alien objects
```

Common mistakes with types

1) Using the wrong operators

To access the fields of an object via its pointer, you need to use the \ operator. It's easy to use the / (division) operator by mistake. If you have ever programmed in C, you might find yourself using the dot operator to try and access an object's fields. This is wrong too. In Blitz Basic, the dot operator indicates a pointer.

These two lines of code are okay:

```
MyPointer\Speed# = 2.5 ; The quick version
MyPointer.Alien\Speed# = 2.5 ; The explicit version
```

These are both wrong:

```
MyPointer/Speed# = 2.5 ; Wrong - uses the division operator
MyPointer.Speed# = 2.5 ; Also wrong - this line implies that MyPointer is a pointer to a Speed object
```

2) Copying objects

Say you want to make an identical copy of an Alien object. You might try something like this:

```
Copy.Alien = Original.Alien
```

It doesn't work! All this does is copy the second pointer into the first. You end up with two pointers sharing the same Alien. To copy a whole object, first create a new empty object, and then copy each field in turn.

```
Copy.Alien = New Alien
Copy\Species$ = Original\Species$
Copy\xpos = Original\xpos
Copy\ypos = Original\ypos
; and so on...
```

3) Manipulating pointers

In some languages you can cause havoc by adding and subtracting values from pointers. Blitz doesn't let you do this. Neither will it let you Print out the value of the pointer. This makes programming in Blitz safer. (If you want pointers that behave like ordinary integers, you can use the highly classified 'Object' and 'Handle' commands, described in the Undocumented section of this site.) The following operators can be applied directly to pointers:

```
pointer1 = pointer2 ; copy pointer values
If pointer1 = pointer2 ; check if two pointers share the same object
EndIf
If pointer1 <> Null ; check if a pointer is valid
EndIf
```

4) Returning pointers from functions

Sometimes you might want to write a function (a sort of customised Blitz command) to find a particular object in a list. To return a pointer to the object, you have to give the function name an appropriate type tag.

```
Function GetFirstAlien.Alien()
Local pointer.Alien = First Alien
Return pointer
End Function
```

The name 'GetFirstAlien.Alien' tells Blitz that the function's result will be an Alien pointer of some kind (although it could be a null pointer).

5) Arrays of objects

An array is a list of variables that can be accessed with a ?subscript?. You can think of the subscript as a kind of ID number. To create (or ?dimension?) an array of a basic data type like strings, you would use something like the following:

```
Dim StringCollection$(100)
```

Now you have a collection of empty strings, any one of which can be accessed with its subscript, as shown here:

```
StringCollection$(50) = ?Have you seen my shoes??
```

That puts a useful piece of text in string number 50.

You can also do this with pointers, but the syntax is a bit tricky until you get used to it. Have a look:

```
Dim AlienPointerCollection.Alien(100)
```

Notice the ?dot Alien? on the end of the array name. Don?t make the mistake of putting it after the subscipt like this:

```
Dim AlienPointerCollection(100).Alien ; Wrong!
```

There?s an important difference between type arrays and ordinary arrays. Unlike strings, an array of pointers can?t be used straight away. They all begin life as null pointers. If you want usable objects right away, you can use a ?For.. Next? loop to create a new empty object at each position in the array:

```
Dim AlienPointerCollection.Alien(100)

For j = 0 To 100
AlienPointerCollection(j) = New Alien
Next
```

Here the variable ?j? goes in place of the subscript. Since ?j? cycles through every number between 0 and 100, it can automatically create a new Alien object at each position in the array. Very useful!

To access the fields of an object in an array, just use the \ operator as you would with any other pointer, but remember to include a subscript number as well:

```
AlienPointerCollection(50)\Species = ?George W. Bush?
```

Don?t worry if you don?t understand this - it takes a bit of time and experimentation to get used to arrays of objects. If the array stuff gives you a headache, just try using ordinary pointers first, and come back to arrays later.

Glossary

- Array - A list of variables where each variable has a unique number to identify it
- Float - A number with a decimal point
- Initialise - to give a variable an initial (starting) value.
- Instance - another word for 'object'
- Integer - a whole number
- Loop - a way of telling the computer to follow the same set of instructions a number of times
- Null - the word used to describe a pointer with no value
- Object - a chunk of memory containing various related pieces of data (based on a type)
- Operator - A built in function like + - = / which has an effect on a variable
- Pointer - a variable that contains the address of another variable
- String - an array of characters
- Type - Any kind of data you use in your program (integers, floats, Aliens...)

I hope you found this tutorial useful. I'd like to thank to Mark Tiffany, Foppy, Morduun, BasicGamer(2?) and everyone else who helped me to understand types. :)

- You can discuss this article with others by clicking HERE.
- For the printable version of this tutorial, please click HERE.