Effective Delphi Class Engineering Part 2: Welcome To The Machine

by David Baer

In this second instalment of our series, we are going to put Delphi's object machinery under a magnifying glass. We'll be examining such things as memory management, method calling protocols and various other compiler-related issues.

There's a good analogy to the focus of this article. Everyone knows that pieces of wood can be structurally joined with nails and a hammer. To a lay person, any bunch of nails the right size can be used, right? But a professional carpenter knows there are many kinds of nails, and using an inappropriate type will at best make a job take longer. At worst, the long-term durability of the construction can be compromised.

Likewise, one can learn to write object oriented Delphi code without knowing some of the details we'll be examining. But understanding this information can make your efforts more efficient in the short term and more effective in the long term. So, grab those hammers and we'll get started.

Homework

Read the fact-filled manual (the Object Pascal Language Guide).

Object Pascal is not a complex language in comparison to some others. Nevertheless, there are hundreds of small details which, if you know and pay attention to them, can lead to more concise, efficient, durable and readable code. Certainly, you can write extensive amounts of code only using about a third of the language, but you won't know what you're missing until you survey all that's available. We'll be covering some of this material that's particularly pertinent to classes/objects, but there's much more that is worth knowing. The *OPLG* is available in Adobe Acrobat Reader format on the Delphi installation CD and also on Borland's website.

But reading the *OPLG* isn't enough. You should also acquaint yourself with all the functions and procedures available from Delphi. Knowing the many and diverse services available from these routines can save you a lot of coding. Unfortunately, the *OPLG* doesn't present this information. For that you'll need to explore the help files, and you can also learn quite a lot by reading the code and (oh, my goodness!) the many comments contained in the `Sysutils` unit.

An even better course of enlightenment might be to read Ray Lischner's *Delphi In A Nutshell* (Wiley, ISBN: 1565926595). Although I've yet to acquire my own copy, I've seen the table of contents. It looks to be the perfect source for the kind of information I'm recommending you learn and, as a bonus, it's quite reasonably priced (at least in the US).

Why am I so adamant that this is important? There are just too many reasons to fully explain. So let me enumerate just a few:

➢ Understanding the scope of names (of variables, methods, etc) will help you to write more comprehensible code.

➢ Knowing the details of the `string` data type in terms of memory allocation and reference counting will help you understand what makes for efficient code when doing character data manipulation. In the event you need to code direct Windows API calls, this knowledge is indispensable.

➢ Fortunately, Delphi doesn't require the use of pointers in most situations, but understanding how they work can turn slothful routines in performance critical sections of code into speed demons.

➢ You sometimes need to code classes for which huge numbers of instances must be created; knowing how data types are allocated can help you conserve lots of memory.

➢ Knowing the effect of using `const` in method declarations can improve performance.

The list is extensive and these are just a few examples. So, just do it: read the manual!

Units

Organize your units with attention to what's visible to the outside world, and (sometimes more importantly) what isn't.

To begin your class writing efforts, you will of course need to understand how a class is declared and supplied with an implementation (ie, the code for the class methods). This is quite straightforward and we'll get to it next. But, in addition, you'll need to understand the layout of a unit. As I pointed out last time, Delphi's IDE can let you create some sophisticated applications without ever worrying about these details. While many of you already understand this, I want to remind those who haven't given it any thought.

A class declaration is a simple construct. Listing 1 presents a unit containing several small class declarations and implementations. Examine the declaration of `TMyClass` and note that we list both class variables (sometimes referred to as member data items) and class methods (procedures and functions) under `private` and `public`.

You may also declare members (data items or methods) right after the `TMyClass = class` line, but you should avoid doing so. The visibility of such items is `public` (or `published` in some cases, but let's not get into that right now).

```
unit MyClasses;
interface
uses SysUtils;
type
  TMyOtherClass = class;
  TMyClass = class
  private
    ANumber: Integer;
    AString: String;
    AnOtherClass: TMyOtherClass;
  public
    constructor Create;
    destructor Destroy; override;
  end;
  TMyOtherClass = class
  private
    AnOtherNumber: Integer;
  public
  end;
var
  StompOnMe: Integer;
```

```
implementation
var
  MyClass: TMyClass;
{ TMyClass }
constructor TMyClass.Create;
begin
  inherited Create;
  AnOtherClass := TMyOtherClass.Create;
  AString := 'The date is ' + DateToStr(Date);
end;
destructor TMyClass.Destroy;
begin
  AnOtherClass.Free;
  inherited Destroy;
end;
initialization
  MyClass := TMyClass.Create;
finalization
  MyClass.Free;
end.
```

➤ *Listing 1*

The methods `Create` and `Destroy` are special method types, known as constructors and destructors respectively. We'll discuss those shortly.

Now, let's talk about the overall organization of the unit. The examples in Listing 1 are pretty conventional. We declare the two classes in the `interface` section of the unit. In doing so, they are available for use by any other unit, including `MyClasses` in its `uses` clause. But there's no requirement to do so. For example, if we had intended `TMyOtherClass` to act solely to support `TMyClass`, we could have declared it in the unit's `implementation` section. Doing that would make it invisible to the outside world.

But, in this case, we can't do that, because `TMyClass` contains a reference to `TMyOtherClass`. Object Pascal is a one-pass compiler and demands that types be declared before they are referenced. In this example, we could make things right by declaring `TMyOtherClass` first. But what if `TMyOtherClass` also referenced `TMyClass`? They can't both be first! Instead, we use the device known as a *forward declaration*. We first provide the one-line declaration of `TMyOtherClass`, and provide the full declaration later.

It would be very useful if we could declare a class in the `interface` section that privately used an internal class defined in the `implementation` section (that internal class then being completely out of sight to users of the 'owner' class). Unfortunately, Object Pascal demands that a forward class declaration in the `interface` section must also have its full declaration appear in that section. Normally this is not a huge annoyance, but it would be nice if the language could be accommodating in this case.

Class Unit Placement

Place collaborating classes in separate units.

For all the talk of private and public so far, we've avoided one important fact. The private members of a class are effectively public to other classes defined in the same unit. Sometimes this can be a great benefit, especially where internal helper classes are involved. But it can also lead to unintentional breaches in an encapsulation strategy.

It's often far from obvious how a collection of collaborating classes should be distributed with respect to unit assignments. Rather than trying to define a set of guidelines, let's just consider an example from the VCL.

The unit dbctrls.pas contains a variety of data-aware visual controls like `TDBEdit`, `TDBCheckbox`, etc. These are related in a sibling-like fashion. It's perfectly reasonable for them to share a unit dedicated to that 'theme'. On the other hand, the db.pas unit contains the definition of `TDataSource` and many other related classes.

The control classes in dbctrls.pas do not collaborate with each other, but they do collaborate with the data source facilities in db.pas. Size considerations aside (these are both pretty hefty units to begin with), it would have been a mistake to combine the classes in dbctrls.pas and db.pas into a single unit. The reason is that it would be very easy to write code in a control method that unintentionally accesses private members in the data source class, or vice versa.

While this might not badly compromise the integrity of the whole at first, it can make it extremely difficult to reassign the classes to separate units later on (as you might wish to do if a unit starts to become too large and ungainly to be manageable).

You can always combine classes into one unit after they reach a degree of maturity and stability. But attempting to go the other way can reveal a host of unanticipated dependencies that are difficult to untangle. Trust me on this one, it's a lesson I learned the hard way.

Memory Utilisation

If you understand where your data is at all times, you'll keep it (and yourself) out of trouble.

Let's continue with Listing 1 to explore memory allocation issues. We'll start with simple variables, and get to objects in a moment. You may have noticed that there is one `var` item declared in the `interface` section and another in the `implementation`. Now, I'm not promoting the former as good practice! It should be avoided wherever possible (the name of that item might suggest why).

You might be thinking, 'How can this be bad practice? Delphi does it!' And indeed it does, by placing a form object reference in `interface` when the IDE generates a new form. This is another of those compromises Delphi uses to achieve RAD-ness. The first thing I usually do with a new form unit is to delete the reference (and mark the form as non-auto-create), and I know a fair number of seasoned Delphi coders do the same.

In general, global variables invite abuse, and you're usually better off avoiding them. In the grand scheme of things, Delphi does expose a few of these (apart from the aforementioned form references). But these are class references like `Application` and `Screen`, which is entirely justifiable. Your program code frequently needs access to global information. Encapsulating that information in a class is a far better way to control it than just leaving it exposed for abuse. A global variable cannot be read-only to the general public. A class data member can be (through the use of properties, the subject of the next instalment).

One very important point to consider is that `var` items declared at the unit level occur exactly once in an executing application. You could create multiple instances of `TMyClass` and each instance would have its own allocation of its data members. But `var` items (and `const` variables) declared at the unit level (as distinct from `var` items declared in a procedure or function) are singletons.

Unlike `var` items in the `interface` section, `var` items in the `implementation` section are not nearly so dangerous. Indeed, they can be quite useful. Object Pascal provides no form of class variable, something available in certain other OO languages. A class variable is one that also exists as a singleton in an executing application, but may only be accessed via a class or object reference. As such, these may be private as well, and can be very useful in numerous situations. But an `implementation` `var` item serves this purpose almost as well.

Before we look at object memory issues, let's briefly discuss where data resides in general. Let's keep it simple for a moment and just consider simple variables (like 4-byte integers). A variable will exist in one of two places: on the heap or on a stack. The heap is just that: a heap of memory, which is used for all data storage except for those items using a stack.

Stacks are a little more complicated. For every thread, a contiguous block of memory is available for use in the calling and executing of procedures and functions. These are true stacks in that they grow when a procedure is called, grow some more when that procedure calls another, shrink when the called procedure is finished, etc. There are three basic uses for stack storage: parameters passed to a called procedure, storage for `var` items in functions and procedures, and working storage for things like intermediate results in expression evaluations.

A point of interest here is `var` items in procedures and functions. You can't code for very long in Delphi before you realize that these are not initialised when a procedure is entered. An exception to this is reference counted variables like strings (which must be initialised if reference counting is to work in the first place).

Turning our attention back to the heap, the good news is that most items residing in the heap *are* initialised to binary zeroes. This includes unit level `var` items and class data members (but does *not* include dynamically allocated records, so beware).

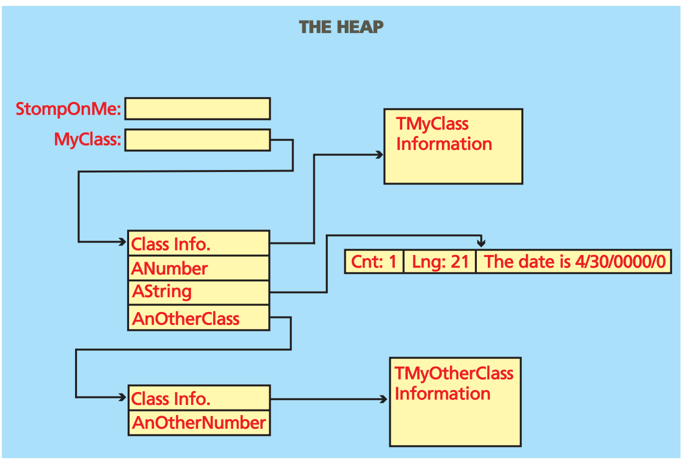
Now, before we tie this all together, let's consider a couple of additional things. The first is the memory allocation for strings. String variables are four-byte pointers, which point to a block of data containing information about the string, as well as the string data itself. That block consists of three contiguous 4-byte pieces (allocation size, reference count and current length) followed by the string data. This block will always be in the heap (although multiple string variables might point to this chunk, courtesy of reference counting). A string variable actually points to the start of the string data.

An object reference variable is just a 4-byte pointer to the actual instance data. The object reference variable may be either in a stack or in the heap, depending on where it is declared. When an object is created, a block of storage for all its data members is acquired in the heap (and helpfully initialised to binary zeroes). The data members occupy contiguous storage, which is preceded by one 4-byte pointer to a block of class information. One such block of class information exists for each class type in the compiled program. For now, we're not going to get into this subject, but we'll look at it in detail in a later instalment.

So, the data members of a class instance occupy contiguous storage, right? Well, not exactly. The basic data members of the class instance do, but if these data members are strings, for example, or references to other class instances, the composite collection of data may reside in many places. Understanding this is a key to understanding what's involved with object assignment. We'll look at that subject below.

Figure 1 shows the main heap allocations that are happening in `MyClasses`. In that unit, one instance of `TMyClass` is created in the `intialization` section for the

➤ *Figure 1*



purposes of illustration. `TMyClass` creates an instance of `TMyOtherClass` during its creation process, so we have two classes in existence, as illustrated.

Constructors

Give your objects a good start in life with well-designed constructors.

There is a special type of class method, called a constructor, which is used to create a class instance. Calling the constructor creates a class instance (ie, an object). Usually, the reference pointer returned from the call is assigned to a class reference variable. Constructors are conventionally named `Create`, but they can be given any name you wish (not that you'd want to buck convention without a good reason).

You do not always need to provide a constructor for your class implementation. If your class is simple and contains only simple data types as member data items, the Delphi object machinery takes care of everything for you. You can think of this (for now anyway) as a default service which allocates the necessary memory block for your member data.

More complex objects may need some custom initialization to be performed when being created. For example, if the object uses an internal helper class, it's often appropriate to just create an instance of the helper class during its own creation. You can see this being done in the constructor for `TMyClass`.

In examining that method, you'll see a call to `inherited Create`. Understanding its purpose requires an explanation of class inheritance, a topic we'll be covering in a future article. For now, let me just suggest that you always supply a call to `inherited Create` as the first statement in your constructor.

There is only one situation I know of where the 'inherited call first' rule will get you into trouble: in a `TThread` descendant, where the thread is created as non-suspended. In that case, the `inherited` call needs to be the last statement in the constructor.

Destructors

Let your objects experience a dignified end with appropriate destructors.

The final moments of an object's life may require wrap-up processing, which is the role of the destructor. For example, if your initialisation created an internal object, it's normal to destroy that internal object in the destructor. But, like constructors for simple classes, you don't always need to provide a destructor. Where there is no internal cleanup needed, the default services will take care of de-allocating the storage block used for member data.

In `TMyClass` you'll see a call to `inherited Destroy` in the destructor. Once again, just take it on faith for the moment that you should always include a line like this as the last statement of all destructors. Destructors must be named `Destroy`, and they must always be declared with the `override` directive (an explanation will have to wait for another instalment).

A perennial point of confusion for neophyte Delphi developers is that of `Free` versus `Destroy`. Code that needs to destroy an object should always call `Free`, not `Destroy`, as can be seen in the `finalization` section of `MyClasses`. `Free` offers a shortcut that saves a lot of repetitive coding. Attempting to destroy an object that was never created to begin with must be avoided (unless you like access violations). `Free` first tests the object reference for `nil` and then calls `Destroy`, but only if the reference is not `nil`. This isn't an ironclad protection mechanism. There are many ways to get into trouble with inappropriate destruction activity, but it offers some respite.

For now, just consider `Free` to be another default service of the object machinery. Although you will frequently need to supply a `Destroy` method, you should never supply a `Free` method implementation.

Self-Awareness

Class methods execute in a very self-ish manner; which is the main point, actually!

Your class method code will normally make reference to the member data of a class instance. How does the executable know which instance is being referenced? It's not obvious by examining the source code, because the compiler is supplying a most helpful invisible assist.

When coding a method call, you do it with a statement that includes a class reference variable, followed by a period, followed by the method name (and parameters, if appropriate). What's actually happening is quite straightforward. The compiler-generated code appends that class reference as a final 'invisible' parameter to the call. That reference is a pointer to the class instance's data member storage block.

Code in a method that accesses member data does so by simply naming the member data variable. The generated code uses the hidden instance reference (again, it's just a pointer) to locate the item in memory. The pointer value can be explicitly accessed in method code by using the reserved word `Self`. This can occasionally be necessary if name collisions occur. For example, when a method parameter is named `Thing` and a class data member has the same name, use of the identifier `Thing` in the method code will reference the parameter. To reference the class data member, you can use `Self.Thing`. In my opinion, using different identifiers is a much better approach.

One final point: it is not the responsibility of your class to protect from erroneous use by class clients invoking methods with bad object references. For example, this line of code:

```
if Self = nil then Exit;
```

is a pathetic exercise in futility. Your class cannot protect itself against this kind of misuse, so don't even bother to try.

Creating And Destroying Objects

You don't have room service: you need to serve up your own objects.

You don't have maid service: you are expected to clean up after yourself (some of the time, anyway).

In Object Pascal, objects must always be explicitly created. Novice users may be forgiven for not realizing this rather obvious necessity since, once again, Delphi's RAD capabilities nicely hide this requirement. After all, because all components are class instances we don't have to write `Create` calls for the components in our code, do we?

It is true that all component instances are OP objects, but the VCL provides the creation services in the case of components on forms and data modules when those forms and data modules are created. The cleverly engineered input streaming mechanism automates this for us.

But, for all other cases, explicit creation is a necessity. This applies to users of your classes, and it applies to you when your class uses internal classes. For small objects, or objects that will always be needed, it's often convenient to simply create those internal classes right in the constructor. Occasionally, an internal class that is 'expensive' to create (using large amounts of memory or having a lengthy creation time) is only occasionally needed. In this case, deferring the creation of that internal class until needed is a better approach.

You are also generally expected to free the objects you create. I suspect that many seasoned developers approach this like I do: if I write a statement that creates an object, I immediately go to the appropriate place in the code and write the call to `Free` on that object. For classes using internal classes, freeing the instances of the internal class is usually best done in the destructor. In fact, if a class uses even one internal class, a destructor will almost always be required, if for no other purpose than ensuring the `free` takes place.

Failing to free objects results in a memory leak (ie, unused but non-reclaimable memory left in the heap). Memory leaks are not always insidious. If an object is to live right up to the end of the execution of the program, failing to free it does no harm. The result is sometimes called a benign leak. The OS will reclaim the storage of abandoned object when the program is terminated, so there is no harm done.

Nevertheless, a good case can be made to explicitly free all objects. There are several commercial and freeware utilities available that can detect memory leaks and I highly recommend that you get in the habit of using one. If you do, then you'll see the wisdom of avoiding even benign leaks. They may cause no performance degradation, but they still show up in a leak detector report. As such, their presence can make identifying true leak conditions more difficult.

There's a common case in which you are relieved of the responsibility of freeing objects. Some classes offer ownership services, wherein they take on the responsibility of freeing objects when they themselves are destroyed. 'Owned' objects have something else looking after their cleanup, by VCL convention. Component classes are the main example of this. The `TComponent` constructor has a parameter named `AOwner`, which is a good clue that this service is being provided.

Additionally, some container classes offer this cleanup service. The `TObjectList` class (in the source file contnrs.pas) is an example. It has a property, `OwnsObjects`, which may be set to `True` to indicate that the auto-freeing service is in effect.

One final point: if it's a good practice to free objects, it's a terrible one to get sloppy and attempt to free them more than once. Doing so will inevitably result in an access violation. If there's a danger of executing a `Free` more than once, make certain you set the object reference to `nil` immediately after the `Free` call (or use Delphi 5's `FreeAndNil` routine to do both operations in a single statement).

Alternate Constructors

Be creative and use multiple constructors where appropriate.

Supplying a single constructor for most classes will usually be all that's needed. But there's no restriction on having multiple constructors, and these can be quite useful in some situations.

In particular, you may wish to supply a simple constructor that gets the basic object set up in a largely non-initialised state. You may also then supply additional constructors with parameters that may be used to specify initial values of data members. This provides a nice flexibility to either create objects for later use (using the simple constructor) or objects that are immediately usable after a single call to the constructor.

You have two choices in providing multiple constructors. Number one is to give each constructor a unique name, and that's that. Your second choice is to name them all the same (`Create` would be a very good choice in this case), and use the `overload` directive. Listing 2 illustrates the two alternatives in action.

Object Replication

You don't have a copy machine: when cloning objects, you need to provide the cloning code in your class.

I think the main benefit of understanding memory usage in object creation, which we spent more than a little time on above, is that you can readily grasp the requirements for copying objects. To begin with, it will be easy to avoid a frequent misconception of neophyte Delphi practitioners.

➤ *Listing 2*

```
...
public
  constructor Create;
  constructor CreateInitialized(Count: Integer; const Text: String);
...
public
  constructor Create; overload;
  constructor Create(Count: Integer; const Text: String);
    overload;
```

```
var
  MyObject: TMyClass;
  MyOtherObject: TMyClass;
...
  MyObject := TMyClass.Create;
  MyOtherObject := MyObject;
```

➤ *Listing 3*

Specifically, the code in Listing 3 assigns the pointer of one object reference to another. That's all! It does not get you a second instance of the object. You have nothing more than two object references pointing to the same instance of the object. If you need to offer replication services, you must provide one or more methods to perform the value assignments.

So, why can't the compiler be smarter and provide some automatic mechanisms to do this sort of thing? The reason is that it cannot possibly know how much is to be copied. If we have a simple class that contains, for example, nothing but integer data members, then this would be no problem. But objects frequently contain internal objects. OK, that makes the job harder, but you could still argue that the compiler should offer this sort of service.

However, objects also frequently contain references to external objects. If we clone an object, do we want to copy the reference only (two objects now pointing to a shared single instance of the referenced object), or do we want to create a clone of the referenced object? The answer will depend on circumstances. Even if the compiler was smart enough to generate this extremely tricky code, it cannot possibly know which of two possibilities is desired.

On the other hand, as a class designer, you should have a precise idea about what a copy operation means. Like it or not, it'll be up to you to write the code to make it happen.

By convention, copying routines are named `Assign` in Delphi classes (not that there's any language requirement for this name). An `Assign` method will typically supply a single parameter with which to specify the 'from' object, and the method will be called on a reference to the 'to' object.

In practice, `Assign` can be used to not only clone objects, but to copy common data between objects of differing types. We will return to this subject in a later instalment, when we delve into the subject of polymorphism. You'll be amazed at the wonderful things you can accomplish with `Assign`.

Next Time

We'll undertake a thorough examination of one of Object Pascal's most elegant features: properties.

David Baer is Senior Architectural Engineer at StarMine. Yes, that's right, he's got a new job at an exciting startup in San Francisco (but he wants to assure everyone that he has *not* been dot-commandeered). Contact him at dbaer@starmine.com