Implementing Class-Traps For Delphi And C++Builder

Or, how to be notified of the birth and death of your objects

by Cyril Jandia

This article presents a small, quick 'n dirty unit allowing you to trap the creation and destruction process of Delphi32 and C++Builder objects. This is achieved thanks to the TClassTrap class, which offers a simple interface for encapsulating this very special service. Trapping object creation and destruction with this class gives us the opportunity to obtain object references at quite unusual moments, when we would normally say it is "too soon" or "too late." Or, put simpler, using the TClassTrap class is a cheap means of hooking calls made to Create and Free throughout our code.

TClassTrap carries out its job at what is the very beginning (ie the begin line) of a constructor or destructor's body. I said "cheap" because we don't need the source code of the classes we are interested in, nor any kind of explicit overloading of what is defined in the .dcu (.obj in C++Builder) object files we use.

Though originally designed for Delphi 2 and C++Builder 1, TClassTrap works fine with Delphi 3 as well, with minor changes to the TVmt record's definition, as I describe later.

As the title points out, what is presented applies equally to TObject derived classes written with C++Builder. To avoid annoying references to "Delphi32 and C++Builder," only the name "Delphi" will appear in the text from now.

TClassTrap In Use

As one example, we can have a solution to the problem of finding the instantiation order of controls created by "foreign" forms (ie where no source code is available). Another is how to get a reference to a "forgotten" exception object, while we are already executing the finally part of a try...finally block: remember that at that moment SysUtils.ExceptObject is nil, this is the so-called "too late" case I described. Yet another example is when you want to learn more about (possibly undocumented) objects which Delphi itself creates for its own use. Borland C++Builder presents a great opportunity for this kind of learning! Plus there are many cases where there isn't one line of source to help us.

Objects Are Everywhere

In Delphi we use objects all the time, even when just placing a TButton onto a form. There is one thing, however, which is not obvious in many cases, at least in our source code. How can we know which classes actually have instances and which do not? As far as Delphi Object Pascal is concerned, we can only know for sure whether a variable does not contain a reference to an object if its value is nil, that is it does not point to an object.

Fine, but if it is *not* nil we just don't know if it points to a valid object or not. If we called Free on the reference, it could point to no object, but it is not automatically set to nil after the call to Free. So, to be able to use a test against nil we must set our variables explicitly to nil after calling Free on objects (Listing 1).

So, we cannot know whether a non-nil value is actually a valid reference to an instance or not (ie pointing to garbage). We need something else. One idea is to carry out some kind of "instance accounting" with, say, a kind of "official account" for a particular class. Such an implementation is straightforward thanks to Delphi which provides us with the most useful TList class: see Listing 2.

This code keeps references to instances of TMyObject in MyObjectList as they are created and also ensures the list gets rid of the references as the corresponding instances to which they point are freed. So, instead of examining the value of AnObj we can ask MyObjectList for information about any instances which are still alive.

This seems to achieve what we were looking for. One possible problem is the loss of efficiency introduced by the extra code: in some intensive computing tasks this overhead may not be acceptable. But I'm sure you have found the *real* problem with this method: it's shouting from the code layout! It's related to readability and code maintenance, because object-oriented design principles strongly dissuade us from using such a programming style with

➤ *Listing 1*

```
var
  AnObj: TMyObject;
...
  if AnObj = nil then
    { we know here for sure that AnObj doesn't point to a TMyObject }
  else begin { it's not nil: let's assume AnObj points to a TMyObject }
    AnObj.Free;
    AnObj := nil; { since we plan to test safely against nil later }
  end;
```

tightly-coupled lines. Rather, we'd prefer that TMyObject register itself in MyObjectList. See Listing 3.

But now we have a new problem. We didn't really want TMyObject's implementation to change because of our need for instance accounting. Also, this technique is not of much help if we have no source code for TMyObject.

So then, how about an idea which may look somewhat crazy at first: could we hook into Delphi's own mechanism of object creation and destruction? After all, Windows itself has in many circumstances forced us to carry this out (see the Windows API reference on *Hooking functions*). Couldn't we use an analogous technique with Delphi Object Pascal and/or its compiler?

Inside the unit SYSTEM.PAS we find lots of information on object creation and destruction, including the VMT, or Virtual Method Table.

Secrets Of The VMT

Strongly typed, compiled object oriented languages which support polymorphism often use what Borland Object Pascal people call a Virtual Method Table. This special compiler generated data structure is very handy for implementing polymorphic classes, that is, classes the instances of which have an actual type known only at run time. In fact, it seems to be the most straightforward way for implementing polymorphism efficiently. Thus statically (ie when parsing the source code) references to polymorphic objects are of the type of a well-known common ancestor, let's say TVehicle. But, dynamically (ie at run time) those references actually point to objects of various derived types, such as TMotorBike, TCar and so on. Each of these instances has a hidden pointer to the VMT of its actual run time class type: approximately equal to a TClass value, also known as the "metaclass" (more on this later). Note that no matter how similar two derived class types sharing a common ancestor can be, the compiler will generate two distinct VMTs. Delphi seems to do this like other languages, but even a little bit more since *any* Delphi class has a VMT. This is because the ultimate ancestor, TObject, from which all Delphi classes derive, already has a VMT with reserved entries for some predefined virtual methods it defines. Well known to us is the Destroy destructor which is what we must override to customise the instance destruction process properly. Listing 4 shows Delphi 2's VMT layout, as presented in Ray Lischner's book *Secrets of Delphi 2*.

Note I have modified slightly Ray's definition of the VMT record layout to support Delphi 3's.

➤ Listing 2

```
unit u_MyObj;
...
var
  MyObjectList: TList{of TMyObject};
  AnObj: TMyObject;
...
  AnObj := TMyObject.Create;
  try
    if MyObjectList.IndexOf(AnObj) < 0 then MyObjectList.Add(AnObj);
    ...
  finally
    if MyObjectList.IndexOf(AnObj) >= 0 then
      MyObjectList.Delete(MyObjectList.IndexOf(AnObj));
    AnObj.Free;
  end;
...
initialization
  MyObjectList := TList.Create;
finalization
  if MyObjectList.Count > 0 then
    { error: some TMyObject's haven't be freed!
      equally, we would have forget to set to nil somewhere... };
  MyObjectList.Free;
end;
```

➤ Listing 3

```
constructor TMyObject.Create;
begin
  inherited Create;
  ...
  if MyObjectList.IndexOf(Self) < 0 then MyObjectList.Add(Self);
  ...
end;
...
destructor TMyObject.Destroy;
var SelfIndex: Integer;
begin
  ...
  SelfIndex := MyObjectList.IndexOf(Self);
  if SelfIndex >= 0 then MyObjectList.Delete(SelfIndex);
  ...
  inherited Destroy;
end;
```

➤ Listing 4

```
type
  PVmt = ^TVmt;
  TVmt = record
{$IFDEF VER100} // D3 specific: both fields prepended since Object Pascal 9.x
(D2, BCB)
    Vmt: Pointer; // most likely related to use of ClassParent field below(?)
    IntfTable: Pointer; // for D3's COM interfaces support stuff
{$ENDIF}
    AutoTable: Pointer;
    InitTable: Pointer;
    TypeInfo: Pointer;
    FieldTable: Pointer;
    MethodTable: Pointer;
    DynMethodTable: Pointer;
    ClassName: PShortString;
    InstanceSize: Cardinal;
    ClassParent: Pointer;
{$IFDEF VER100} // D3 specific: field inserted since Object Pascal 9.x (D2, BCB)
    SafeCallExceptionMethod: Pointer;
{$ENDIF}
    DefaultHandler: Pointer;
    NewInstance: Pointer; // what we'll hook
    FreeInstance: Pointer;
    Destroy: Pointer; // what we'll hook
  end;
```

Normally, this should remain ok with Delphi 2 and C++Builder as well since they defined, respectively, VER90 and VER93 (Delphi 1 was VER80, that is Object Pascal version 8 after Borland Pascal 7).

We saw a polymorphic object instance has an embedded pointer to the TClass designating record of its actual metaclass (ie a TVmt), its "family" name if you prefer. Well it's almost true. Delphi objects have such a pointer, but this one actually points to just past the end of the per-class fixed-size TVmt record, instead of pointing to this record's beginning. The location pointed to is where pointers to programmer-defined virtual methods are stored, consecutively, using four bytes each (they are 32-bit pointers). So, if we cast a TClass value to a pointer we get the place where Delphi's compiler stored the pointers to all the virtual methods this TClass has either introduced itself, or inherited from its direct ancestor.

Step-By-Step Virtual Method Call

Calling a virtual method on an object consists of:

1. Determining statically (ie at compile time) what one could call the "virtual index" of the method: such a constant index is unique to a virtual method name used throughout a whole set of ancestor and descendant classes linked together by direct inheritance.
2. Getting the 32-bit pointer to the end of the object's TClass designating record (ie a TVmt) a pointer embedded in the object itself, at offset 0.
3. Retrieving the pointer to the virtual method from the array pointed to by 2. using 1.
4. Carrying out the call to the code pointed to by 3.

Note that though this scheme seems lengthy, in fact the compiler generates quite straightforward code for these four steps. So the code remains fast: when compared to a static method call, a virtual method call overhead is not a relevant issue nowadays. Anyway, the biggest known cost of this call speed issue is experienced with dynamic method calls. But that's another story: see Ray Lischner's book for further details.

The undocumented Assembly View in Delphi 32 is very helpful for a better understanding of all this: switch it on and off with the following REG_SZ typed key in the Windows Registry:

```
HK_CURRENT_USER\Software\
  Borland\Delphi\[2|3].0\
  Debugging\EnableCPU
```

setting it to 1 or 0 respectively.

So, how does Delphi manage object creation and destruction using the two TVmt fields NewInstance and FreeInstance. After studying the contexts in which these identifiers appear and how they are used, we can deduce quite easily which very specific role they play in the creation and destruction process. First, it is important to understand they do not handle the entire process alone. Thus, when we write something like TMyObject.Create, in addition to the code generated by the compiler for preparing the access to the metaclass value (TMyObject), there are also two other routines involved that are out of direct reach: _ClassCreate and _ClassDestroy. It seems they have to do with the handling of an exception frame to be used in case of failure. Only then does the code pointed to by NewInstance and FreeInstance carry out the real job, either allocating and zeroing the dynamic memory chunk required by a fresh class instance before executing the first statement of the class constructor, or deallocating an existing class instance just after the last statement of the class destructor has been executed. It's worth noting that Delphi defines default implementations of these methods, so that, with rare exceptions, all classes use the same code when it is time to New or FreeInstance a chunk of dynamic memory.

A Useful Plug

As just stated, we cannot force Delphi to create or destroy objects another way, at least at the level _ClassCreate and _ClassDestroy are called, because at this level we are still under the control of the compiler generated code. But it would be really cool to have a means of "plugging" our own pointers into the two "slots" NewInstance and FreeInstance represented within the TVmt record. You guessed it, we do have one, it is called VirtualProtect. This function of the Win32 API allows a process to modify the protection attributes of pages of its virtual address space. With VirtualAlloc we can modify the protection attributes of the pages in which our classes' VMTs reside, for instance from PAGE_EXECUTE_READ to PAGE_READWRITE.

Win32 has the reputation of being a more secure, powerful API than Win16 could ever be, a main concern of which is to prevent applications from corrupting each other's data. It may sound strange therefore to read that Win32 doesn't forbid self-modifying code for a process. It is not that strange. In fact, self-modifying code is allowed for a process because Win32 considers it is the process's own responsibility to decide to change page protection attributes if so desired, but only for pages belonging to its own virtual address. So that a process with default application-level privileges cannot change the protection attributes of another process's pages. In short, Win32 assumes every process knows exactly what it is doing within its own address space and why, but the same process cannot decide changes in another one's. Thus, when we want, with the help of the TClassTrap class to plug "foreign" pointers into the NewInstance and FreeInstance slots of an arbitrary TVmt record, we are modifying our own code.

Rewriting NewInstance And FreeInstance

It is one thing to find a API trick for redirecting compiler-generated pointers to NewInstance and FreeInstance within an (also) compiler-generated TVmt record, but it is another thing to implement properly our own versions of both class functions in order to manage the list of instances for a particular TClass. For the latter task, we have to rely on another trick, a Delphi Object Pascal trick more precisely. What we want is a simple way of redefining both NewInstance and FreeInstance class methods and at the same time connect the corresponding TVmt record's slots of the class to trap to these redefinitions. The problem is the phrase: "at the same time." It is not a good idea. Even more, we simply cannot do it that way.

In fact, we have no other choice than proceeding in at least two steps. First, we redefine NewInstance and FreeInstance in a generic fashion (see TTrappedObject in the implementation part of the ObjTraps unit, on the disk of course). Then and only then we plug the resulting code pointers into the NewInstance and FreeInstance slots of the class to trap (see TClassTrap.SetTrapProc and TClassTrap.SetMagicHooks in the code on the disk).

The first step is a static one, in other wards it is done at compile time of our unit, once for all. The second is handled at run time when the programmer will decide to register a new class-trap for a particular TClass value. Hence the source code found in the unit OBJTRAPS.PAS unit on the disk.

It's Up To You Now

Listing 5 shows a basic class trapping example. I'm sure you'll find plenty of uses for TClassTrap, be it for debugging purposes, for class instance accounting in simulation models, for Delphi experts, or for anything else: Delphi programming is only limited by our own imagination!

Acknowledgements

Very special thanks are due to: Roy Nelson of ETT for his support on pre-releases of this article, Hervé for his many good questions and remarks, and "ChD" for constructive criticism of early code versions. You all three have been of great help. I must also say Ray Lischner's book *Secrets of Delphi 2* has inspired me a lot.

After two years as a technical support engineer at Borland France Support Department for support contracts, Cyril Jandia, AKA FLFan, has recently re-oriented himself towards training on Borland products. He's currently preparing a home for his fiancée Caroline, still too many miles away, of whom he impatiently awaits the arrival in the capital. Both love England and its culture and hope to spend more time in the country soon!

Recommended Reading

- *Object-Oriented Software Construction*, Bertrand Meyer, Prentice-Hall, 1988 (French version by InterEditions, Paris, 1990).
- *Eiffel — The Language*, Bertrand Meyer, Prentice-Hall, 1991 (French version by InterEditions, Paris).
- *Delphi 2 — Programmation Avancée*, Dick Lantim, Eyrolles, 1996 (great on Win32's IPC, the OpenTools API and more).
- *Delphi 2 Developer's Guide*, Xavier Pacheco and Steve Teixeira, SAMS, 1996.
- *The Revolutionary Guide To Delphi 2*, various authors, Wrox Press, 1996.
- *Secrets of Delphi 2*, Ray Lischner, Waite Group Press, 1996 (Valuable insights on the VMT, RTTI, etc and confirms what we can learn by ourselves from long journeys deep into the RTL/VCL source, simply a "must have" book!).

➤ *Listing 5*

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)     Memo1: TMemo;
  private
  public
  end;
var Form1: TForm1;
implementation
uses ObjTraps;
{$R *.DFM}
procedure MemoTrap(const trap: TClassTrap;
  const obj: TObject; op: TObjectOperation);
begin
  if op = ooCreate then
    MessageBox(0, 'TMemo created', '', MB_OK)
  else
    MessageBox(0, 'TMemo freed', '', MB_OK);
end;
initialization
  MakeTraps([TMemo], MemoTrap);
end.
```