Fatal Startup Error

by Brian Long

Recently I received a question emailed to *The Delphi Clinic*. Because the answer and some additional explanatory material seemed to want to take up an entire issue's *Clinic* space allocation, it seemed appropriate to split it into a separate article (as was done with a question about completely customising tooltips in Issue 16). This is one of those innocent enough questions trying to get a simple answer, but to try and justify the answer it gives me an excuse to delve under the hood and to try and explain how Delphi does some of its primary form-based operations. Here is the question:

"When I build up a form, Delphi does its usual job of adding declarations for all my objects at the top of my form's class definition. If the form represents a portion of a complex user interface, then this form class can end up getting very, very large. To try and help reduce the form unit file size I manually delete the declarations of all the purely decorative objects that I will never explicitly refer to in code, such as labels, bevels, panels and so on. However, sometimes, when I have been doing a spate of these deletions, I then get a run-time error (an `EClassNotFound` exception) as the form is being created. The message says: *Class XXXX not found* where XXXX is one of the component types I have been deleting references to. Since this only happens sometimes can you tell me the exact circumstances that control whether this error will show or not?"

As is often the case, to understand all about this error we need some background information. However to save you reading through all that information to find the actual answer, this time I'll put that first and follow it with the explanation.

If you delete all references to any specific type of component from a form class, the error will be generated. On the other hand, if you ensure you leave at least one reference to each component type used on the form, then the form will load up without failure at run-time. That's the brief version, missing out the available workaround. Now for the more elongated version...

A Form Is A Stream

To get an understanding of the problem, we need to pose an often-overlooked question. When working in the form designer, it's rather well known that the IDE is storing all the information about which components go on the form, and what their properties are, in a form file. This .DFM file is an object stream, the form and all the components on it are streamed out to the form file. An object stream file is a binary file that contains enough information for an appropriate executable file to reconstruct all the constituent objects. The generally ignored question is "How does a Delphi executable turn a DFM file into a real form instance?" This is another of those apparently innocent questions that requires a lot of digging to find the real mechanics at work. So let's start exercising our shovels.

As was explained in my article *File Handling Part 5: Streaming Components* back in Issue 10 of *The Delphi Magazine*, an object stream contains, for each streamed object, the name of the component class (as a string), followed by the name and value of each property that has a non-default value. There is no code stored in an object stream.

The whole streaming process relies on the code that implements an object already residing in the executable file (or, in the case of Delphi 3, possibly in packages used by the executable file), which explains the use of the phrase "enough information" a couple of paragraphs above.

When a component is added onto a form, Delphi does two things to the form unit. Firstly (and instantly) it adds an *object reference* to the form class through which you can refer to the run-time component. The online help refers to such object reference declarations as *instance variables*. Secondly, when you next compile or save the unit, it may update the uses clause to ensure its new object reference declaration compiles without error.

Bear in mind that Delphi has a smart linker. This means that if you have a whole bunch of units in one of your `uses` clauses, all the code from all these units does not automatically go into your final executable. Instead, only code that is explicitly or implicitly referred to (as well as code for `initialization` and `finalization` sections) is linked in.

So, take the case where you put a button on a form. Delphi inserts an instance variable `Button1` of type `TButton` and when you next compile, it adds the `StdCtrls` unit into your `uses` clause. Because of the reference to `TButton` in the instance variable declaration, the `TButton` code (or a fair portion of it) will be included in the final binary file. If you delete that declaration, then there is a possibility that all the `TButton` code will be smart-linked out of the program. That would not be helpful as then the `TButton` object could not be manufactured at run-time. Granted, other forms may well have buttons on and so *might* pull the code in, but "might" is not good enough. One form is represented by a form unit, which can be used in any project you like. Just because all the code required for a form's successful operation is coincidentally pulled into one final compiled project by the code in other units does not mean this will always be the case in all projects, so this is not an adequate situation.

So one issue related to this subject is getting the code for streamed objects into the EXE. The other issue is how the streaming system turns the string that identifies the type into an actual instance. There is no inherent way to turn a string into a class or object and so Delphi and the VCL has to do some work to allow a translation to be made.

Delphi inserts all the instance variables as fields in the unnamed section (also known as the default section) of the form class. This default section has the same semantic implications as the published section, so all these fields are published.

RTTI And Other Internal Tables

When properties are published by a component (and a form is an example of a component), the compiler generates RTTI (run-time type information) for their types. This is so the object inspector can analyse, edit and store their values in the form file, and the streaming system can set their values when loading a form.

But all these instance variables are not properties. For other published items, typically found in forms and data modules, which include data fields and methods (ie component references and event handlers) the compiler performs a different task.

The address and name of published methods are stored in a *method table* (used by the class functions TObject.MethodAddress and TObject.MethodName). The names and relative addresses of published data fields are stored in a *field table* (used by the TObject.FieldAddress function) and a list of class references for their types is stored in a *field class table*. All these tables (like virtual method tables, dynamic method tables and the type information table) are implemented on a per-class basis.

When a form gets created, its constructor attempts to initialise all the components that sit upon it by loading the form file stream from the executable's resource table. This is done with the Classes unit routine ReadComponentRes. Assuming the resource (named the same as the form class) is found, this routine then reads through the stream creating objects and setting properties.

When it reads a component class name, it attempts to turn the string into a class reference (via which it can call the constructor) by using the form's field class table. It loops through each class reference in the table, calling the class function ClassName and comparing the result with the read string. If it finds a match, it calls the constructor to make a new instance and then reads and sets all the appropriate properties.

The job of assigning this newly created component to the relevant form data field is left to both the component and the form in combination. When the component is constructed and passed an Owner parameter, it calls the target Owner's InsertComponent method. InsertComponent will try to locate a data field with a name that matches the value of the component's Name property (using FieldAddress). If it finds one, it sets its value accordingly.

So to recap, the compiler builds a field class table for each form (as well as all other components if they have published data fields as opposed to properties) and the streaming system relies on this to translate from a string into a class reference.

As you have found, if you delete all references to a particular component type from a form class then that type will not be found in the form's field class table and so you get the *Class not found* exception. So leaving at least one reference to the component type in the form's default section would deal with the situation.

An Alternative Remedy

But there is another way. This other way caters for situations that component writers might face. If someone is writing a component (say, called TComposite) that is made out of several constituent components (say, a TLabel and a TEdit), it might be desirable for the label and edit to be stored in and read from the form file, but not available in the Object Inspector (and so with no declarations in the form class). So when a TComposite component is placed on a form, there will only be one reference added to the form class and that will be for a TComposite object. This means there is a potential for the field class table to have no record of TLabel and TEdit.

To cater this possibility, Delphi supports a global class list, as well as each of these form-local lists. To get a class onto the global list, call RegisterClass, passing in the class reference; for example, RegisterClass(TButton). To get multiple classes into the global list, use RegisterClasses; for instance, RegisterClasses([TLabel, TEdit]). These calls will usually be made in an appropriate unit's initialisation section to ensure the classes are in the global list before any streams are read. If the stream reading mechanics cannot find a class in a field class table, the global class list will be checked using the FindClass routine. If FindClass fails to locate the target class, it generates an EClassNotFound exception (which is where your error is coming from). The online help for RegisterClass summarises this lot with:

"Call RegisterClass to register a class with the streaming system. Form classes and component classes that are referenced in a form declaration (instance variables) are automatically registered. Any other classes used by an application must be explicitly registered by calling RegisterClass. When classes are registered, the class type can be obtained from the class name by calling the FindClass or GetClass function. If a class is not registered, GetClass returns Nil when passed the class name, while FindClass raises an exception."

Historically, during the development of the original Delphi, there were no field class tables manufactured. Instead, for each form, Delphi maintained a unit initialisation section that contained a call to RegisterClasses.

Each time you added a new component onto the form, the component type was added into the open array passed to `RegisterClasses`. Clearly Borland's R&D decided this was too unsightly and went for a behind the scenes approach instead (with no form unit source code generated, it becomes much easier on the eye). Incidentally, this pre-release behaviour explains the otherwise misleading error message *Error in module Unit1: Call to* `RegisterClasses` *is missing or incorrect* which occurs if you damage the final end of a Delphi 1 form unit (Delphi 2 fixed this oversight).

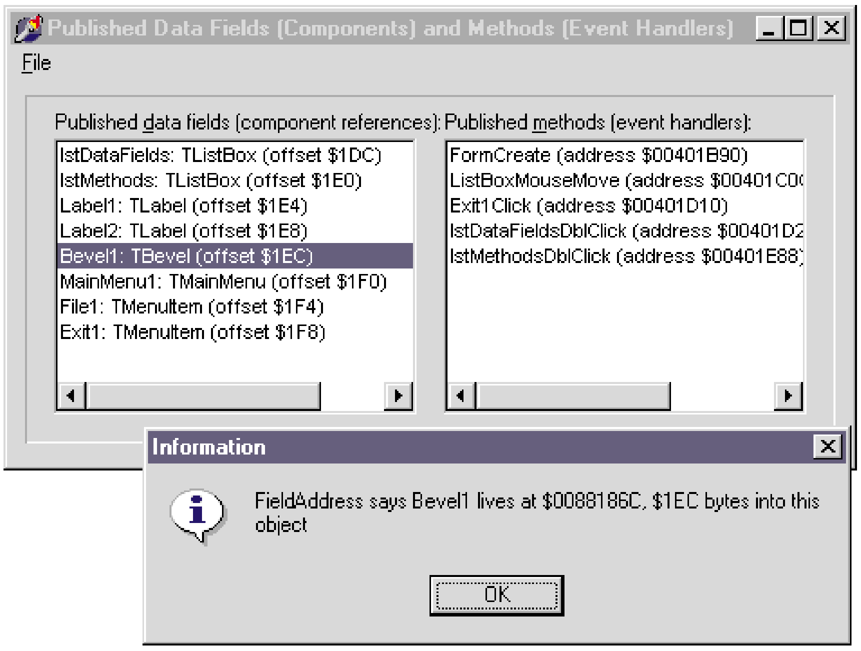
Of course by ensuring the form unit contains the class name of all components, either in the form class definition, or in a `RegisterClasses` statement, we can rest assured that the code for the component class will definitely not be smart-linked out. So all bases are covered.

Getting Your Hands Slightly Dirty?

The business of reading a form resource for the purpose of creating all the constituent components and setting the properties can be analysed by examining the source code in the VCL. You can follow the logic through in the `Classes` unit source if you have it. `ReadComponentRes` (or in the case of 32-bit Delphi the internal version, `InternalReadComponentRes`) creates some type of stream to map onto the resource (`THandleStream` in 16-bit, `TResourceStream` in 32-bit). The stream's `ReadComponent` method is called with the form reference passed as a parameter. `TStream.ReadComponent` creates a `TReader` and calls the `ReadRootComponent` method, again passing the form reference.

The `TReader` then does some initialisation stuff and calls the forms `ReadState` method. The form inherits a `ReadState` method from `TComponent`, this calls the `TReader`'s `ReadData` method, which then calls its `ReadDataInner` method. Are you keeping up?

`TReader.ReadDataInner` reads and sets all the form properties



Published Data Fields (Components) and Methods (Event Handlers)

Published data fields (component references):

```
lstDataFields: TListBox (offset $1DC)
lstMethods: TListBox (offset $1E0)
Label1: TLabel (offset $1E4)
Label2: TLabel (offset $1E8)
Bevel1: TBevel (offset $1EC)
MainMenu1: TMainMenu (offset $1F0)
File1: TMenuItem (offset $1F4)
Exit1: TMenuItem (offset $1F8)
```

Published methods (event handlers):

```
FormCreate (address $00401B90)
ListBoxMouseMove (address $00401C0
Exit1Click (address $00401D10)
lstDataFieldsDblClick (address $00401D2
lstMethodsDblClick (address $00401E88)
```

Information

FieldAddress says Bevel1 lives at $0088186C, $1EC bytes into this object

➤ *Figure 1*

and then repeatedly calls `TReader.ReadComponent`, passing in `nil`. `ReadComponent` reads in the component class name string and name and then calls the nested `CreateComponent` procedure. `CreateComponent` calls `FindFieldClass` to translate the class string into a class reference. `FindFieldClass` searches the form's field class table, and if necessary calls `FindClass`. If a class reference is found, its `Create` method is called to construct an instance of the class (otherwise `FindClass` generates an exception).

Virtual Constructors

It is because this requirement of a generic approach of calling the constructor through some returned class reference that explains why component constructors must be polymorphic. The `FindFieldClass` method returns a `TPersistentClass` (declared as `class of TPersistent`). This is typecast into a `TComponentClass` (declared as `class of TComponent`) and through this class reference the `Create` constructor is called. Of course, the supplied class reference may actually be a `TButton` reference or a `TEdit` reference or some other reference. Thanks to the polymorphic requirement of class constructors, the right one is sure to be called.

If you could think of some requirement for gaining access to the local field class table, or indeed the field table or method table, you are free to do so. It is not necessarily that obvious how to write code to talk to these compiler generated structures but it is certainly possible. The TblTest.Dpr project on the disk does exactly this. It has a load of code in a unit `TblInfo` that gives access to the all three of these tables and, where appropriate, navigate from one entry to the next. The project uses these routines to list out all the published fields and methods of the main form in a couple of listboxes upon form creation. Figure 1 shows the output and some verification that the displayed information is correct. Remember that the `TObject` methods `FieldAddress`, `MethodAddress` and `MethodName` make use of these tables in order to work.

The code in the unit is a little unpleasant in places, but you will be pleased to hear that I avoided using assembler (which the runtime library and VCL code chooses to use). Not only is there a lot of pointer manipulation going on but there is plenty of conditional compilation to ensure everything works in Delphi 1, 2 and 3. I won't bother listing any code here in the magazine pages, it is not that interesting. Suffice it to say that if you are interested you will need to have a look at the files supplied on the disk.

Last Words

Some final comments on the questioner's approach to reducing the size of form units. He was manually deleting object references from the form unit. This leaves the object in existence with all its properties intact, including the Name property. If you are never going to refer to these components, then leaving all their Name property values unchanged will cause a certain amount of unwanted information to be left in the form file.

As an alternative, you can set the Name property to an empty string in the Object Inspector, which (when you press Enter or click on something else) will automatically remove the object reference from the editor. The IDE normally keeps the object reference identifier and the Name property in sync: if you change the property, the identifier gets updated. Changing the Name to an empty string forces the IDE to delete the object reference. The Name string will now take no space in the form file, and therefore the executable will be that much smaller.

If you delete the object reference then the component will still have its original name, although there is no object reference form data field through which you can talk to it. But if you found you needed to, you could use the form's FindComponent method, which takes a name string and attempts to locate a component that it owns whose Name property matches. For example:

```
(FindComponent('Label1') as
  TLabel).Caption := 'Found it';
```

If you delete the Name property value, then this will not work.

Also, if the Name property remains set, the Object Inspector is happy to continue making event handlers for the component's events when you double click next to the event (or select the event and press Ctrl-Enter). It auto-names the handlers, so an OnClick event handler for a TLabel component Label1 will be Label1Click.

With the Name property blanked out, the auto-naming process fails somewhat and the Object Inspector complains: *Cannot create a default method name for an unnamed component.* You can circumvent this problem by simply making up your own event handler name and typing it next to the event on the Object Inspector and pressing Enter.

Of course in an event handler for an unnamed component, you can reference the component using the Sender parameter (which is passed to practically all event handlers and represents the component that triggered the event):

```
procedure TForm1.UnnamedLabelClick(
  Sender: TObject);
begin
  (Sender as TLabel).Caption :=
    'It was clicked';
end;
```

So to summarise what has gone on over the last few pages, the program will run successfully if you leave at least one reference to each component type in each form or alternatively make use of RegisterClasses. En route we have examined the underlying mechanisms employed by the streaming system to take a stream file linked into the program as a resource and turn it into a bunch of real objects. Finally, for form unit size reduction we have seen one or two points that might dictate whether you manually delete component references from the form class or instead set the component's Name property to an empty string.

Brian Long is a UK-based freelance Delphi and C++ Builder consultant and trainer. He is available for bookings and can be contacted by email at brian@blong.com

Copyright © 1998 Brian Long
All rights reserved