Manipulating Runtime Type Information

by Phil Brown

Most Delphi programmers can give a reasonable definition of the `private`, `protected` and `public` keywords. However, when asked about the meaning of the `published` keyword in a class definition, many will give the answer 'It's what appears in the object inspector at design-time.' Although this is a correct consequence of the `published` keyword, it is not a correct definition. The Delphi help file presents the following information under `Published properties|Published components`:

'The visibility rules for published components are identical to those of public components. The only difference between published and public components is that runtime type information is generated for fields and properties that are declared in a published section. This runtime type information enables an application to dynamically query the fields and properties of an otherwise unknown class type.'

So, the consequence of defining a property in the `published` section is purely and simply that Run Time Type Information (RTTI) is generated for it and is available in any Delphi program that uses the class. The Object Inspector uses this information, for example, so you can change component properties at design-time.

The RTTI stores all the appropriate information for readable, non-array properties, including the name, type and any extra data that may be useful (such as minimum values for integers, maximum lengths of strings, etc). Defining variables, functions, procedures, read-only or array properties in the `published` section is not an error but has no effect and Delphi treats the definition as if it were `public`. Furthermore, to actually get RTTI generated and stored in the program, the class (or an ancestor class) must be defined within a scope that has the `$M` (or `$TYPEINFO` verbose alternative) compiler switch turned on, as in Listing 1.

I have chosen to use the `$TYPEINFO` compiler switch in preference to `$M` as it is more descriptive and obvious when reviewing the code at a later date. Note that *any* descendants of this class will have RTTI generated for their published properties: it is not necessary to define the compiler switch again. If you wish to generate RTTI for a class, you can choose to descend your class from `TPersistent` (which is defined with the `$M` compiler switch on) rather than `TObject`; doing so provides access to a number of methods related to RTTI for object streaming. Although the Delphi help file declares that 'There is seldom, if ever, any need for an application to directly use the `$M` compiler switch', there are great benefits in using RTTI in appropriate circumstances. However, a convenient way of ensuring that RTTI is generated is to descend your class from `TPersistent`.

All forms and components have `TPersistent` as an ancestor as it is this class that provides the ability for these objects to save their state to disk, using `TFiler` descendants `TReader` and `TWriter`. It is the interoperation of these classes that saves components and forms to disk in the DFM file format using RTTI. It is possible to provide your own `TFiler` classes and then use them to store `TPersistent` descendants, but this is a convoluted process as it is has been designed more for internal rather than application use. Even so, all Delphi developers wishing to understand RTTI further should acquaint themselves with the `TPersistent` class and the methods it provides.

➤ *Listing 1: Defining a class to generate RTTI.*

```
{$TYPEINFO ON}
type
  TPerson = class (TObject)
  private
    FName: String;
    FAge: Integer;
    procedure SetAge (Value: Integer);
  Published
    // properties with RTTI follow
    property Name: String read FName write FName;
    property Age: Integer read FAge write SetAge;
  end;
{$TYPEINFO OFF}
```

Importing Data Using RTTI

One area in which RTTI can be very useful is in importing and exporting data. Most applications contain classes that represent real-world ('problem domain' or 'business') entities, such as the classic suppliers, customers and orders. Very often, these classes will be stored in a database, using any suitable means, but equally often an application must provide some way of importing or exporting this data to a data file in some format, possibly because the application needs to share data or simply to populate the database. Traditionally, this file format is customised for each application: it may be comma separated, with the data fields in a known sequence, the application may even use a binary format containing a memory image of the object or database record, or an application-specific format may be used. In these situations, each application must provide its own

means of importing and exporting data and the developer must code a new system each time it is required.

There is a common flaw with each of the data formats previously described: none of them provides for any means of version control. Using a comma separated field format, the number and sequence of the fields in the incoming file must exactly match that which is expected. If the structure varies, incorrect values will be read into fields. Furthermore, each database table (or object) would need to have import/export routines coded individually. The ideal would be a structure independent file format (and a means of importing and exporting it) which applied to all entities in the system.

The developers of Delphi had to consider the exact same issues when they were designing the means by which form and component properties were stored in the .DFM file. This file had to remain valid even if a new version of a control was released, and the file structure had to be applicable to all components without extra effort being required on the part of the developer. These requirements were met using RTTI and we can do a similar thing for our own applications.

With respect to version independence, assuming the name of a class remains the same between versions, the only data-related things that can change about it are field values and properties. It is a good general rule *never* to expose field values as simple scalar variables but as properties, even if you don't provide read and write routines. This provides for data encapsulation and future-proofs your application. An object could be described as a list of all property name and value pairs. For example, consider the `TPerson` class defined earlier. An instance of this class could be entirely represented as:

```
Name=Fred Bloggs
Age=35
```

This could be stored as a text file and later interpreted, populating the `Name` and `Age` properties of a new `TPerson` object to recreate an exact replica of our original object. Consider what now happens if we add an `Address` property to our class. If we have a text file containing the above data, it is still possible to interpret the file, setting the `Name` and `Age` properties, but leaving the `Address` property containing its default value (blank). If our new version of `TPerson` happened to drop the use of the `Age` property, our interpreting routine could recognise this fact and simply not attempt to do anything, but still could set the `Name` property that remains valid. Thus our file achieves version independence. Delphi uses the exact same scheme when storing form information in a .DFM file. If the data had been stored in a simple comma separated file, removing or adding a new property would mean that data stored in the 'old' format could not be imported without prior conversion.

```
BEGIN TPerson
Name=Fred Bloggs
Age=35
END
BEGIN TPerson
Name=John Doe
Age=48
END
```

➤ *Listing 2: A complete file describing two objects.*

There are two issues left to consider before our file format is complete: how to describe a new class (ie that the data that follows is for a `TPerson` class) and how to cope with the situation when a property changes type (eg from an integer to a double). To cope with the first issue, we can precede the above data with a delimiter that declares the class name (let's say that we will use `BEGIN` and `END` keywords to be consistent with Delphi). We will delegate the task of converting data into the correct type to the routine that must actually set the properties. Listing 2 is a complete example of a file that contains the information for importing two complete objects into the system.

So far, all we have described is a file format, which could be interpreted easily enough using traditional techniques. However, using RTTI we can write a *single* routine that can import *any* object (or sets of objects) and reuse this routine in different projects, thus reducing the effort involved each time. This routine must be able to create a class of the appropriate type given the name of the class, set a number of named properties, and finally save it. Delphi provides a couple of methods which handle the creation of a class given the class name, `RegisterClass` and `GetClass`. These only work with descendants of `TPersistent`, however, so if you want to use your own class hierarchy descended from `TObject` or some other class you will need to provide similar functionality.

Saving the object depends on each individual application. A well designed application will have a `Save` method on each main system object, which is responsible for storing that object in the host database, in which case saving an object is simple. Lesser systems may have a custom storage technique for each class and so a way of implementing this would be to have a large set of `if..then` clauses that test the class type and then execute some appropriate code. We are interested in a general purpose, application-neutral component that reads data, so we will have an event that is called each time an object has been read in. It will be the responsibility of the host application to decide what to do with each imported object.

We still need some code to actually set the appropriate property to the required value. This is where RTTI comes in: without it, the only way would be to have a large table or set of `if...then` clauses, one for each property on each object that is to be read in. This would need to be maintained as classes changed, not a task for the faint-hearted. Instead, we can request Delphi to access the information for the property with the required name, and then call a method using this information to set the property value.

The unit that provides access to the RTTI structures for each class

is called TypInfo. This small unit defines a few record structures and some procedures and functions to manipulate them. It does, however, make copious use of pointers and the implementation is mainly in assembler, so unless you are very familiar with these it is probably best not to attempt to understand how it all works.

The main function in TypInfo that we will use is GetPropInfo. The definition for this is as follows:

```
function GetPropInfo(
  TypeInfo: PTypeInfo;
  const PropName: string):
  PPropInfo;
```

This takes a pointer to a TTypeInfo structure and a property name, and returns a pointer to a TPropInfo record structure, which contains all of the information that we need. To get at the required TypeInfo structure for the object we can just use the TObject.ClassInfo method. One caveat in using this method is that for classes that do not have RTTI available, it returns nil and so we must protect ourselves against this when using the GetPropInfo function, which insists on being passed a valid TypeInfo structure.

The TypInfo unit provides a number of methods for setting a property value, dependent upon the type of the property and it is vital that the appropriate method is called. To actually set a property value, we must check the PropType member which tells us the type of the property (integer, string, enumeration etc) and then call SetStrProp, SetOrdProp, SetFloatProp or SetVariantProp. There is also a SetMethodProp procedure but this deals with methods rather than data values and so we are unlikely to require it for reading our data files.

Enumerated types (which includes Boolean properties) are set within the RTTI routines using integer values. The first enumerated value corresponds to integer value 0, the second to 1 and so on (these are the same values as returned by the Ord function for an enumerated value). The above technique is a reasonable way of importing data, but we can go one step further to make our input file more descriptive and more resilient in the face of changing enumerations.

Within the Object Inspector, when setting a property that is an enumerated type (such as the BorderStyle property of many components), it presents the actual enumeration name for the current property value, such as bsNone or bsSingle. This implies that the RTTI contains information for enumeration names, and this is indeed the case. The GetEnumValue function returns the ordinal value of the string value of an enumeration name passed to it, using the TPropInfo.PropType member to access the appropriate set of strings. Therefore, we can modify our import routine to cope more ably with enumerated types, so that we can interpret lines in our input file such as Sex=sxMale rather than Sex=1. This is obviously more resilient to changes in the enumeration list and more descriptive into the bargain.

We have shown how to update an object's properties using RTTI, but we have not validated the data in any way, for example to ensure that the incoming data contains reasonable values. Fortunately, there is a very simple mechanism that can be used to achieve this: property accessor routines.

When the RTTI sets a property value using the above routines, it obeys the write rule that was provided for the property, so if a procedure is used (as in TPerson.Age in the code example in Figure 1), this procedure will be called, passing the appropriate value. Therefore, simply providing standard accessor methods allows you to control the values provided through RTTI manipulation. Very often, these accessor methods will already have been provided as part of a good class design to protect property values. This validation of incoming values is a good reason for using RTTI to import data rather than a custom routine to decode a specific file format, as the same rules for correctness checking are used by both the application and the import routine.

Exporting Data Using RTTI

Exporting data is a feature required by many applications. Regardless of the format used (comma separated values, spreadsheet or a custom text or binary format), RTTI can be used for export as neatly and efficiently as for import.

If you have looked at the TypInfo unit you will see a set of converse routines to match those provided for setting property value, all beginning with the Get prefix. We will use these routines to create a TPropertyExporter component that will write out a file containing any number of objects of any type in a format suitable to be read in by the TPropertyImporter component.

The essence of the exporter component is simple: its constructor takes a filename for the destination file containing object data, and has a single method, WriteObject, which is passed the object to be written out. A property on the class, PropertyList, contains a list of names of properties of the object to be written. If empty, then the class writes out all properties for the object.

The WriteObject method is obviously the key one for this component and is split neatly into two routines. The first has the task of determining the property names to be exported if the PropertyList is empty. A list of all property names is obtained as an array of pointers to TPropInfo structures with the TypInfo.GetPropInfos routine. There is a slight complication to this routine, in that it must be passed an area of memory that it will populate. This memory area must be of the correct size, calculated by finding out the number of properties of the object using the PropCount member of the TTypeData record structure returned from GetTypeData. Essentially we need to provide a dynamically sized array from which we can then extract the property names, but we must be careful to free the memory we allocated. The listing to extract the

property names is too long to appear in full here, but the source code for the `PropertyExporter` class can be found on the disk.

The `PropertyList` now contains a list of all properties for the object that are required to be exported; this will either have been defined by the developer (by pre-populating the list), or by the system (by determining all published properties from the RTTI). The next step is to write out the object header consisting of `BEGIN` followed the class name, and then to access and output each property in turn.

Our `TPropertyExporter` class uses a routine which we have provided to convert any property value into a string. This routine accesses the property information from RTTI, determines the type of the property and then calls an appropriate routine to extract the required value and convert it into a string. For an enumerated type the `GetEnumName` routines is used to convert an enumeration integer value into a suitable string.

Some developers may be concerned about the overheads of using RTTI access routines to fetch and set property values. A look at the source code in the `TypInfo` unit should reassure them, as it consists mainly of some small and tight assembler routines. Remember also that RTTI is used extensively by the Delphi development environment and so it is very much in its interests to have efficient code. Using the access routines it is possible to update tens of thousands of properties per second, performance that even the most cynical developer should find acceptable.

Using RTTI For Database Manipulation

To further demonstrate the range of applications to which RTTI may be put, it may be useful to discuss how it may be used to write classes that can save themselves to an SQL database by generating their own SQL code at runtime. Using this technique, it is possible to write a class that is able to load and save itself to and from a database and any descendants of this class will automatically inherit this ability. Centralising access provides for very robust database manipulation, as it can be guaranteed that all appropriate fields are populated, regardless of where in the program a load or save operation is made. As each object 'knows' the fields it requires to store its property values, this scheme can even be extended to ensure that the correct fields are defined in the database during application startup, again helping to provide for a resilient application.

Essentially, the technique is to use RTTI to extract a list of all properties for an object and to build up a valid SQL statement defining these property names and values. In order for this technique to work, the class must be compiled with RTTI generated, all data that is required to be saved must be exposed in the `published` section and the database must be defined with a table matching the class name (or simple derivative,

```
type
  TPet = (Dog, Cat, Rabbit, Hamster);
  TPets = set of TPet;
  TComplexObject = class
  private
    FPets: TPets;
    function GetPets: String;
    procedure SetPets (Value: String);
  published
    property Pets: TPets read FPets write FPets stored False;
    property PetStr: String read GetPets write SetPets;
  end;
function TComplexObject.GetPets: String;
begin
  Result := '';
  if Dog in Pets then Result := Result + 'D';
  if Cat in Pets then Result := Result + 'C';
  if Rabbit in Pets then Result := Result + 'R';
  if Hamster in Pets then Result := Result + 'H';
end;
procedure TComplexObject.SetPets (Value: String);
begin
  FPets := [];
  if Pos ('D', Value) <> 0 then FPets := FPets + [Dog];
  if Pos ('C', Value) <> 0 then FPets := FPets + [Cat];
  if Pos ('R', Value) <> 0 then FPets := FPets + [Rabbit];
  if Pos ('H', Value) <> 0 then FPets := FPets + [Hamster];
end;
```

➤ *Listing 3: Providing textual representations of complex properties.*

say by stripping the leading T) and with fields that match the published properties in name and type. Fortunately, most of these requirements are easily achieved and desirable.

We will provide a new abstract class, called TSQLObject, which provides protected SQLInsert, SQLUpdate and SQLSelect routines, which return a string containing an appropriate SQL command. Any object that descends from TSQLObject that publishes its properties and has access to a database connection will automatically be able to load and save itself. Note that any class would need to provide an appropriate WHERE clause for the SQLUpdate command, as the choice of primary key is table specific. Of course, it is entirely possible (and desirable) to provide a descendant class of TSQLObject which provides standardised Load and Save methods, with knowledge about primary keys and then descend all your application classes from this.

For example, our TPerson class earlier could generate the following SQL statements (using the first published property as the primary key, and the class name less the leading T for the table name):

```
INSERT INTO Patient (Name, Age)
  VALUES ("Fred Bloggs", 35)
UPDATE Patient SET Name="Fred
  Bloggs", Age=35 WHERE
  Name="Fred Bloggs"
```

A TSQLObject is provided on the disk to demonstrate the possibilities. As all load and save operations require generating a list of published properties, this is cached in a private string list the first time it is required and re-used for all future operations. This is a useful performance optimisation.

Manipulating Complex Properties

In the examples above, all published properties are used for load and save operations. It is possible that a class has some properties that are inherently complex, such as arrays, sets or even other classes. In these cases there is no built-in mechanism to obtain a string representation of the property, and you might think that it is not possible to use these RTTI mechanisms to manipulate them. In these situations, a very elegant solution presents itself: the naturally complex property should not be published, but public, and a new property provided in the published section exclusively for the purpose of presenting a string representation of the complex property. This property should use read and write accessor routines and these should simply be used to provide a convenient textual representation of the complex property.

It may also be true that a property is required to be published that is not required to be saved to a file or database. As all of our routines so far have manipulated the entire set of published properties, how can we prevent these undesired properties from appearing in our output files or SQL commands? We could use a naming scheme to identify them, but there is a much more elegant way. Each property declaration can optionally be followed by a stored False directive, which is used for components to indicate properties that should be displayed in the Object Inspector but not actually streamed to disk. We can use this information for exactly the same purpose for our published properties, and ignore those that are defined stored False. The IsStoredProp function in TypInfo returns a Boolean value which we can respond to appropriately whenever we use a published property. The general technique of manipulating complex properties is demonstrated in Listing 3.

Conclusion

This article has demonstrated how to generate, access and manipulate Run Time Type Information. Although designed primarily for streaming forms and components to and from disk at design-time within the Delphi environment, RTTI can also be used to provide runtime objects with flexible capabilities for a variety of purposes. Using it appropriately can result in smaller programs that are quicker to develop, more robust and require less maintenance and the ease of use makes it a valid technique for all Delphi developers.

Phil Brown is a senior consultant with Informatica Consultancy & Development, specialising in OO systems design and training. When not orienting objects he enjoys sampling fine wine. Contact him as phil@informatica.uk.com