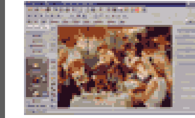


Delphi Informant
Features
Case Studies
News
New Products
Book Reviews
Product Reviews
Opinion
Back Issues
Search
Downloads
Article files
Third-party files
Upload a File

FREE Trial Issue
New Subscription
Renew Subscription
Delphi CD-ROM
Report Problems
Change of Address
Seminars
Delphi Development Seminars NEW!
Informant
ICG News
Contact Us
Advertise with Us
Write for Us

DBNavigator

Delphi 3 / RTTI

By Cary Jensen, Ph.D.

Run-time Type Information

An Introduction to Delphi's Undocumented RTTI

This article provides an introduction to run-time type information (RTTI), which is information the Delphi compiler stores in the code segment of your compiled project. This information is associated with published properties of a class, and it provides a mechanism for treating the symbolic information associated with your types as strings. One example of how RTTI impacts your everyday use of Delphi is the Object Inspector. The Object Inspector displays the names of published properties. This information is retrieved using RTTI.

Delphi ships with a unit, named typinfo.pas, that contains the RTTI functions and procedures, as well as type declarations used by these functions. By adding this unit to your unit's **uses** clause, you can call these functions to access RTTI. At a minimum, you should consider taking a look at the typinfo.pas file in Delphi's \Source\VCL directory.

This article demonstrates several uses of RTTI. One note of caution is in order, however. Borland has specifically not documented the typinfo unit, and reserves the right to change it in any new release of Delphi. It is essential that Borland maintain this right, since the RTTI features are critical to the operation of Delphi itself. As a result, it's possible that if you use RTTI in your application, subsequent changes to the typinfo unit in a new version of Delphi will require you to make modifications to your programs if you want to recompile them under the new version.

Getting RTTI for Enumerated Types

Enumerated types are used throughout Delphi. An example of this is the *TCommonAvi* enumerated type, which declares the valid values for the *CommonAVI* property of an Animate control. The following is the *TCommonAvi* declaration from Delphi 3's comctrls unit:

Delphi Development Seminars
April 17-21, CA,
May 22-26, Wash. DC
REGISTER TODAY!

More By This Author
- Interfaces Revisited : Part II: Interface References versus Object References
- Interfaces Revisited: Part I : Declarations, Implementation, and Method Name Resolution
- Delphi Frames : Understanding Delphi 5's New Visual Container Class
- The Data Module Designer : Delphi 5's Gift to Database Programmers
- Delphi 5 : A First Look at the New Features

Latest Features
- Delphi in the Office
- Parsing the Web
- *TChart* Actions
- CORBA: Part II
- Interfaces Revisited

Article Rating
Rate this article on a scale from 0 to 5
5 Best
4
3
2
1
0 Worst
Submit

Email

Tell a friend about this article!

```
type TCommonAVI = (aviNone, aviFindFolder, aviFindFile,
  aviFindComputer, aviCopyFiles, aviCopyFile,
aviRecycleFile, aviEmptyRecycle, aviDeleteFile);
```

about this article!

RTTI permits you to do two things with enumerated types. You can retrieve strings that contain the name of each value in the enumerated type, and you can identify the ordinal position within the enumerated type of one of its valid values using a string representation of the value.

You obtain a string equivalent of an enumerated type value using the *GetEnumName* function:

```
function GetEnumName(TypeInfo: PTypeInfo;
                     Value: Integer): string;
```

The first argument is a pointer to the enumerated type's RTTI information, and the second argument is the ordinal position of the value within the enumerated type. *GetEnumName* returns a string representing the corresponding enumerated type value.

You get the ordinal position of an enumerated type value based on a string using the *GetEnumValue* function:

```
function GetEnumValue(TypeInfo: PTypeInfo;
  const Name: string): Integer;
```

Like *GetEnumName*, the first argument is a pointer to the RTTI information. The second argument is a string that represents the enumerated type value. This function returns the ordinal position of the corresponding value.

The relationship between these two function calls is demonstrated with the following code segment:

```
var
  s: string;
  i: Integer;
begin
  s := GetEnumName(TypeInfo(TCommonAvi),3);
// Displays aviFindComputer.
  ShowMessage(s);
  i := GetEnumValue(TypeInfo(TCommonAvi),s);
// Displays 3.
  ShowMessage(IntToStr(i));
```

As mentioned, these functions require a pointer to the RTTI for an enumerated type. Since the pointer for a particular type may change from one version of Delphi to another, you must use the *TypeInfo* function to retrieve this information. *TypeInfo* takes a single argument, the type of the enumerated type whose RTTI pointer you want to return:

```
function TypeInfo(TypeIdent): Pointer;
```

The use of these functions is demonstrated in the ANIMATE project, which also demonstrates the use of the Animate component from the Win32 page of the component palette. (This project, and the other demonstration projects discussed in this article, are available on diskette and for

download; see end of article for details.) Figure 1 shows the main form of this project as it might appear while running. There's a ComboBox that lists the various valid values for the *TAnimate.CommonAVI* property. In most applications, you would have populated this ComboBox using its *Items* property at design time.

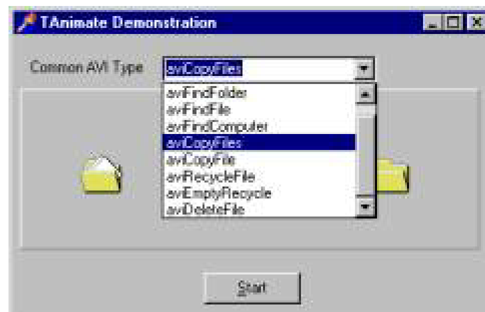


Figure 1: The *TCommonAVI* values displayed in the ComboBox are discovered at run time using RTTI.

In this project however, the ComboBox is loaded dynamically from RTTI using the *GetEnumName* function from within the form's *OnCreate* event handler (see Figure 2). A **for** loop iterates through the *TCommonAvi* enumerated type. For each ordinal position, the *GetEnumName* function is called, and the returned value is added to the ComboBox's *Items* property.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ca: TCommonAvi;
begin
// For each value of the TCommonAVI enumerated type.
for ca := Low(TCommonAvi) to High(TCommonAvi) do
    // Get string equivalent of the enumerated type value.
    ComboBox1.Items.Add(GetEnumName(TypeInfo(TCommonAvi),
                        Ord(ca)));
// Initialize ComboBox to the first item in the list.
  ComboBox1.ItemIndex := 0;
end;
```

Figure 2: The ComboBox is loaded dynamically from RTTI using the *GetEnumName* function from within the form's *OnCreate* event handler.

RTTI is used again to assign the value selected in the ComboBox to the Animate's *CommonAVI* property. This is performed from the ComboBox's *OnChange* event handler, shown in Figure 3. This code includes an additional step for the purpose of clarity. Specifically, the value returned by *GetEnumValue* is assigned to an intermediate variable named *ValueOrd*. This variable is then cast as a *TCommonAVI* type. Instead of using the variable *ValueOrd*, the value returned by the *GetEnumValue* could have been cast directly, permitting these two steps to be represented by a single statement.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
var
  ValueOrd: Integer;
begin
if Animate1.Active then
    begin
      Button1.Caption := '&Start';
      Animate1.Stop;
    end;
// Get the ordinal position of the value associated
// with the selected string in the ComboBox.
  ValueOrd := GetEnumValue(TypeInfo(TCommonAvi),
                ComboBox1.Items[ComboBox1.ItemIndex]);
// Cast this ordinal value to the TCommonAVI type.
  Animate1.CommonAVI := TCommonAVI(ValueOrd);
end;
```

Figure 3: Assigning a value to the Animate component's *CommonAVI* property using the ComboBox's *OnChange* event handler.

Getting Object Property Listings

As you learned earlier, it's possible to get the names of published properties using RTTI. This is done by populating a *PPropList* using a call to *GetPropList*:

```
function GetPropList(TypeInfo: PTypeInfo;
  TypeKinds: TTypeKinds; PropList: PPropList): Integer;
```

A *PPropList* is an array of *TPropInfo* records, and each record holds information about a particular property. This record includes fields such as *Name* and *PropType*. The first argument is the *TypeInfo*. Unlike *GetEnumName*, for which you must use *TypeInfo*, there is a second, alternative way to get the *PTypeInfo* argument. Since this function is used on objects, you can use the *ClassInfo* property as this first argument. The second argument is a set of the property types. Following is the declaration of the *TTypeKind* enumerated type, which defines the valid values for this set:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration,
  tkFloat, tkString, tkSet, tkClass, tkMethod, tkWChar,
  tkLString, tkWString, tkVariant, tkArray, tkRecord,
  tkInterface);
```

This is the *TTypeKinds* declaration:

```
TTypeKinds = set of TTypeKind;
```

Other useful *TTypeKind*-related declarations in this unit include the following:

```
const
  tkAny = [Low(TTypeKind)..High(TTypeKind)];
  tkMethods = [tkMethod];
  tkProperties = tkAny - tkMethods - [tkUnknown];
```

The third argument of *GetPropList* is the *PPropList* that is populated with the property information. Finally, *GetPropList* returns an integer representing the number of properties returned in the *PPropList*.

The use of *GetPropList* is demonstrated in the PROPLIST project. The main form for this project is shown in Figure 4. This form includes a ComboBox, whose contents are populated at run time with the form's *OnCreate* event handler with the names of the components appearing on the form:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  ComboBox1.Items.Clear;
for i := 0 to Form1.ComponentCount -1 do
    ComboBox1.Items.Add(Form1.Components[i].Name);
  ComboBox1.Text := '';
```

```
end;
```

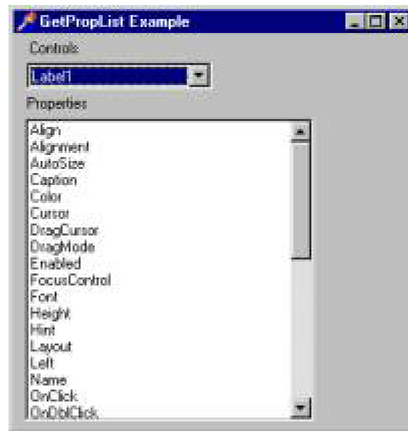


Figure 4: The names of the published properties of a class selected in the **Controls** ComboBox are displayed in a ListBox. These values are discovered using *GetPropList*.

The ListBox is populated with the property names of the object selected in the ComboBox. This operation is performed from the ComboBox's *OnChange* event handler, as shown in Figure 5.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
var
  PropList: PPropList;
  i: Integer;
  CompName: string;
begin
  PropList := AllocMem(SizeOf(PropList^));
  i := 0;
  CompName := ComboBox1.Items[ComboBox1.ItemIndex];
  ListBox1.Items.Clear;
  try
    GetPropList(FindComponent(CompName).ClassInfo,
                tkProperties + [tkMethod], PropList);
    while (PropList^[i] <> nil) and
  (i < High(PropList^)) do begin
      ListBox1.Items.Add(PropList^[i].Name);
      Inc(i);
    end;
  finally
    FreeMem(PropList);
  end;
end;
```

Figure 5: The ListBox is populated with the property names of the object selected in the ComboBox. This operation is performed from the ComboBox component's *OnChange* event handler.

This code is generic, in that the *PTypeInfo* is extracted using the *ClassInfo* property of a component, a pointer to which is returned using the *FindComponent* method. *FindComponent* returns a reference to an instance of an object based on a string, which in this case is the selected component name in the ComboBox. Since *FindComponent* returns a *TComponent* reference, and *Name* is a property of *TComponent*, it's unnecessary in this example to cast the reference returned by *FindComponent* to another class.

Using RTTI with Properties

Polymorphism permits you to treat objects that descend from different classes similarly. For example, you can access the *Name* property of any component that descends from *TComponent* using a *TComponent* reference. This is exactly what's being done in the following code, which comes from the PROPLIST example described earlier. It's used to populate the

ComboBox with a list of the form's component names.

```
for i := 0 to Form1.ComponentCount -1 do
  ComboBox1.Items.Add(Form1.Components[i].Name);
```

However, the ability to treat these components polymorphically in this way is possible only when the property (or method) being accessed is visible in the shared ancestor class. *Name*, in this example, is declared **published** in *TComponent*, and therefore satisfies this requirement.

When two or more objects have the same property, but that property is not declared as **public** or **published** in a common ancestor, you cannot access it polymorphically using a reference to the ancestor. An example of a property such as this is *Color*. The *Color* property of both the *TEdit* and *TMemo* classes is inherited from *TControl*. However, this property is declared as **protected** in *TControl*. The *Color* property is re-declared as **published** in the *TEdit* and *TMemo* class definitions, respectively. Since *TEdit* and *TMemo* do not share this property in an ancestor class with sufficient visibility to be accessed at run time, it isn't possible to treat these two classes polymorphically with respect to the *Color* property. For example, the following code generates a compiler error:

```
for i := 0 to Self.ControlCount - 1 do
  Self.Controls[i].Color := clTeal;
```

By comparison, the *Hint* property, which is declared as **published** in *TControl*, can be treated polymorphically for any *TControl* descendant. For example, the following code compiles properly:

```
for i := 0 to Self.ControlCount - 1 do
  Self.Controls[i].Hint := 'hi' + IntToStr(i);
```

Fortunately, RTTI provides a mechanism that permits you to access **published** properties in a generic fashion across classes, even when those classes do not share a visible inherited version of the property. This mechanism is provided through a series of procedures with names like *SetOrdProp*, *SetStrProp*, *SetMethodProp*, and so forth.

These set methods require a *PPropInfo* reference to the property. This reference is generated by a call to *GetPropInfo*:

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

You then pass this *PPropInfo* reference to an appropriate set method. The following is *SetOrdProp*, which can be used with any Integer or Longint value:

```
procedure SetOrdProp(Instance: TObject;
  PropInfo: PPropInfo; Value: Longint);
```

The use of *GetPropInfo* and *SetOrdProp* are demonstrated in the PROPINFO project. This project includes a button with the following *OnClick* event handler attached to it:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  PropInfo: PPropInfo;
  i: Integer;
begin
if ColorDialog1.Execute then
    for i := 0 to Self.ControlCount - 1 do begin
      PropInfo := GetPropInfo(Self.Controls[i].ClassInfo,
                              'Color');
      if Assigned(PropInfo) then
        SetOrdProp(Self.Controls[i], PropInfo,
                   ColorDialog1.Color);
    end;
end;
```

When you click this button, the **for** loop iterates through all *TControl* descendants on the form. For those controls that have *PropInfo* for a *Color* property, *SetOrdProp* is called to color the control. Figure 6 shows how this form looks before clicking the button, and Figure 7 shows how it looks after the button has been clicked.

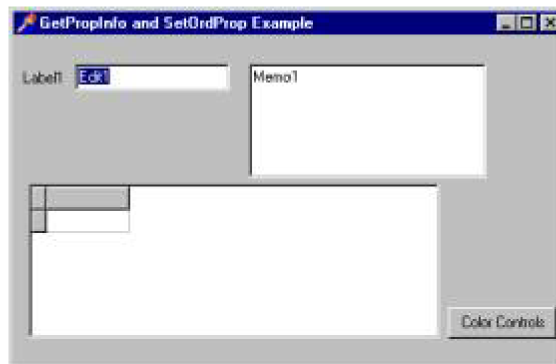


Figure 6: The PropInfo main form when it's first displayed.

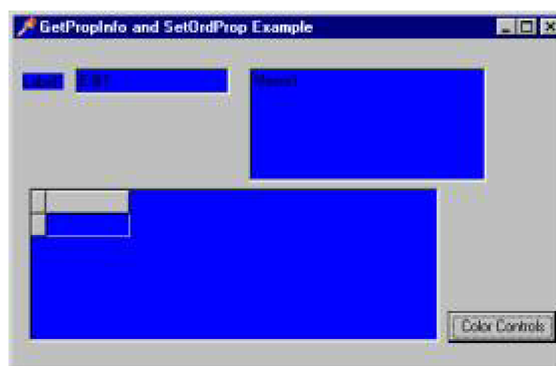


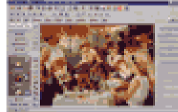
Figure 7: The PropInfo main form after clicking the button labeled **Color Controls** and selecting the color *clBlue*.

Conclusion

RTTI is information about the published properties of your classes that the compiler stores in your executable. Using the procedures and functions of the typinfo unit, you can extract this information at run time. When used effectively, RTTI can simplify your code and reduce your reliance on string constants that can be difficult to maintain.

The projects referenced in this article are available for download.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including Delphi in Depth [Osborne McGraw-Hill, 1996]. He is also a Contributing Editor of Delphi Informant, and was a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. For information concerning Jensen Data Systems' Delphi consulting and training services, visit the Jensen Data Systems Web site at http://idt.net/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at mailto:cjensen@compuserve.com.



INFORMANT COMMUNICATIONS GROUP

Informant Communications Group, Inc.
10519 E. Stockton Blvd., Suite 100
Elk Grove, CA 95624-9703
Phone: (916) 686-6610 • Fax: (916) 686-8497

Copyright © 2000 Informant Communications Group. All Rights Reserved. • Site Use Agreement • Send feedback to the Webmaster • Important information about privacy