Introducing Delphi RTTI

by Marco Cantù

You might have heard that Delphi uses and manages Run Time Type Information (RTTI for short) in many complex ways, but looking in the help file and in most of the available books you'll find little or nothing on this subject. RTTI is very important in Delphi, especially in the Delphi development environment

Consider this case. You write a brand new component, with custom properties based or your own data types, and these properties are readily displayed in the Object Inspector. How does the Object Inspector know about the code you've written? How can it list the values of an enumerated type or a set type which you've defined? The magic going on here is called RTTI.

Wouldn't it be nice to be able to access similar information in a similar dynamic fashion? Sure, and this is exactly what this article is about. I won't be able to give you the ultimate word on this topic, simply because all I know is based on research and experiment. All Borland tells us about RTTI is included in the un-commented file TYPINFO.PAS.

The lack of documentation is probably because Borland wants to be free to change how RTTI works in future versions, but looking at the first two versions of Delphi I can say that this area seems to be quite stable. However, take extreme care in using this approach, since it is completely unsupported by Borland (although it has been discussed from time to time in the CompuServe Delphi forums). Notice, by the way, that in this article I'm covering Delphi 2, although most of the code will run in Delphi 1 almost unchanged.

Accessing Type Information

The basic idea of RTTI is that you can access information related to a given data type. This can be a class or a plain Pascal data type. You can also get the type information related to a property, accessing it by name.

Let's start by looking at some VCL source code. If you look for the class `TObject` in the help or in the source, you will notice an interesting but only vaguely documented method:

```
class function ClassInfo:
  Pointer;
```

This class method returns an undefined pointer, described as "a pointer to the run-time type information (RTTI) table for the object type" in the Delphi 2.01 help. This pointer actually refers to an area in memory holding the RTTI information for the class, generated automatically by the compiler. Notice that this pointer is not a class reference, this is what the `ClassType` method of `TObject` returns.

Looking Into TYPINFO.PAS

To understand the role of this `ClassInfo` pointer we have to start looking into the file TYPINFO.PAS. It turns out that this `ClassInfo` pointer is actually of type `PTypeInfo`. The same pointer is returned by another Delphi function, which allows you to access to type information of non-class data types:

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

By the way, I've found several online references claiming that you can only obtain RTTI information for classes and properties. Although this is the more common use, it is possible to get similar information for plain data types and for classes with no published members.

In Listing 1 you can see the definition of the data structure `PTypeInfo` points to, the list of enumerated values for the `Kind` field and the further details of the data types. These further specifications are used by the `TTypeData` structure.

`TTypeData` is the type of the commented `TypeData` member of the `TTypeInfo` record. You can access it either by finding your way with pointer arithmetic (looking at the actual size of the string), or by following the official approach: use the `GetTypeData` function:

```
function GetTypeData(TypeInfo:
  PTypeInfo): PTypeData;
```

This function basically returns a pointer to the last field of the `TTypeInfo` structure. What kind of type information do we get access to? This basically depends on the `TTypeKind`. The `TTypeData` structure, in fact, is a big variant record, with up to three levels of nested variant structures. Instead of showing you

➤ *Listing 1: TTypeInfo record and related types (from TYPEINFO.PAS)*

```
type
  PTypeInfo = ^TTypeInfo;
    // a pointer to TTypeInfo
  TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
   {TypeData: TTypeData}
  end;
  TTypeKind = (tkUnknown, tkInteger, tkChar,
    tkEnumeration, tkFloat, tkString, tkSet,
    tkClass, tkMethod, tkWChar, tkLString,
    tkLWString, tkVariant);
  TOrdType = (otSByte, otUByte,
    otSWord, otUWord, otSLong);
  TFloatType = (ftSingle, ftDouble,
    ftExtended, ftComp, ftCurr);
  TMethodKind = (mkProcedure, mkFunction);
  TParamFlags = set of (pfVar, pfConst, pfArray);
```

the original listing, I've tried to reformat it in a more readable way, adding some comments (following the declaration they refer to) as well. The result is in Listing 2.

Armed with this information we can start writing some helper functions and test them in simple programs. I'll get back later to the TYPINFO.PAS file to explore some further data structures and routines defined there. The files for this article include a fully commented version of TYPINFO.PAS, renamed CTYPINFO.PAS. It can be a handy reference.

RTTI For Ordinal Types

The simplest techniques relate to extracting information about ordinal types, including simple ones (eg integers and characters), enumerated types and sets. Once you understand how the `TTypeData` structure is arranged, there is little more to say. So we can start looking into the helper routines, then the example program.

The first routine is the `ShowOrdinal` procedure (see Listing 3), which adds RTTI information for enumerated data types to a list of strings (a `TStrings` object) passed as a parameter. The first parameter is a `PTypeInfo` pointer.

Besides the basic information, I add the minimum and maximum values if the data type is not a set and add specific values for enumerated and set types. Enumerated types have a base type and a list of values. The base type is the type this enumeration was built from. Usually this is the type itself, but in some rare cases (such as the `TBorderStyle` type) the base type is an enumeration with more values. Notice that the `BaseType` field is of type `PTypeInfo`, so we can access its name using the pointer de-reference operator (^). To show the list of possible values of the enumerated type, I use a second helper routine, `ListEnum` (which is described later).

For a set, I basically show the data of the base type by calling the `ShowOrdinal` procedure again. This is simple, because the `CompType` field stores the `PTypeInfo` pointer I need to pass to the procedure.

Notice that in Delphi 2 *only* you can refer to a field of a structure without using the de-reference operator, even if you are using a pointer. Instead of `pti^.Kind` you can write `pti.Kind` producing exactly the same effect.

Enumerated Type Values

In the code above I use the `GetEnumName` function. It requires as parameters the type information of the data type (obtained by calling the `TypeInfo` function, passing the enumerated data type as parameter) and the value of the enumeration we are looking for. For example, I can write:

```
GetEnumName(
  TypeInfo(TFormStyle), 0)
```

to retrieve the `fsNormal` string. The code in the procedure above, for the `TTypeKind` and `TOrdType` enumerated types is only slightly more complex, also because we have to cast the value of the enumerated type to an integer, as in:

```
GetEnumName(TypeInfo(TFormStyle),
  Integer(fsNormal))
```

which returns the string corresponding to the enumerated value passed as a parameter. Notice that the reverse operation is possible as well, by calling the `GetEnumValue` function.

What is interesting is that knowing the minimum and maximum value of an enumerated type we can easily list all the names. This is

➤ *Listing 2: The TTypeData structure*

```
type
  PTypeData = ^TTypeData;
  // a pointer to TTypeData
  TTypeData = packed record
    case TTypeKind of
      tkUnknown: ();  // no information
      tkLString: ();  // no information
      tkLWString: ();  // no information
      tkVariant: ();  // no information
      tkInteger: (
        OrdType: TOrdType;
        MinValue: Longint;
        MaxValue: Longint);
      tkChar, tkWChar: (
        OrdType: TOrdType;
        MinValue: Longint;
        MaxValue: Longint);
      tkEnumeration: (
        OrdType: TOrdType;
        MinValue: Longint;
        MaxValue: Longint;
        BaseType: PTypeInfo;
        // the original type definition
        NameList: ShortString);
        // the enumeration names (see GetEnumName)
      tkSet: (
        OrdType: TOrdType;
        CompType: PTypeInfo);
        // the enumerated type the set is built from
      tkFloat: (
        FloatType: TFloatType);
      tkString: (
        MaxLength: Byte);
      tkClass: (
        ClassType: TClass;
        // the class reference
        ParentInfo: PTypeInfo;
        // the parent type information
        PropCount: SmallInt;
        // the number of properties
        UnitName: ShortString
        // the unit defining the class type
       {PropData: TPropData});
       { the properties data: to access this information
         call procedure GetPropInfos or function GetPropList}
      tkMethod: (
        MethodKind: TMethodKind;
        // mkProcedure, mkFunction
        ParamCount: Byte;
        // the number of parameters
        ParamList: array[0..1023] of Char
        // the parameters list, better described as:
       {ParamList: array[1..ParamCount] of
          record
            Flags: TParamFlags;
            ParamName: ShortString;
            TypeName: ShortString;
          end;
        ResultType: ShortString});
        // the return type
  end;
```

what the procedure `ListEnum` does in a few lines of code (Listing 4).

The ORDTYPE Example

With this code available we can build a very simple example showing information about ordinal types. We only need to provide a component with a list of strings (a list box or a memo will do) for the output and a way to select a data type.

A good solution, of course, is to have a list box or a combo box for the selection. However, if we add the names of the data types in this list, then there is no way to retrieve the type information. One possible solution is to create a list with both the class names and the class information. This is easy to do because the `TStrings` and `TStringList` classes allow us to attach objects to the strings we store. Since these objects are simply 32-bit pointers, we can store any other similar pointer instead, casting it to `TObject`. We can write something like:

```
ListBox1.Items.AddObject(
  'Integer',
  TObject(TypeInfo(Integer)))
```

Since we'd like to add many data types, a better approach is to wrap this statement inside a form method, which takes a `PTypeInfo` parameter and adds the type name and the type information to the listbox:

```
procedure TForm1.AddType(
  pti: PTypeInfo);
begin
  ListBox1.Items.AddObject(
    pti^.Name, TObject(pti))
end;
```

➤ *Listing 4*

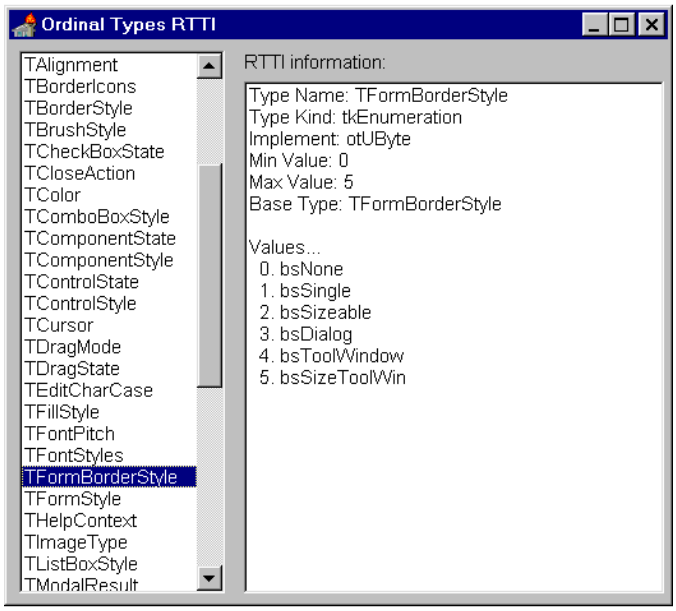
```
procedure ListEnum(
  pti: PTypeInfo;
  sList: TStrings);
var I: Integer;
begin
  with GetTypeData(pti)^ do
    for I := MinValue to
      MaxValue do
      sList.Add('  ' +
        IntToStr(I) + ': ' +
        GetEnumName(pti, I));
end;
```

When the form is created, the ORDTYPE program simply calls this `AddType` method about 50 times in the `FormCreate` method, with both system and VCL types, as in:

```
AddType(TypeInfo (Boolean));
AddType(TypeInfo (TAlignment));
```

Of course you can add some more system types (provided they are ordinal types) and many more VCL types. I've just picked a few. Now when the user clicks on an item of the list box, we can write the code shown in Listing 5 to retrieve the pointer to the type information and update the list.

The result of all of this work is the form you can see in Figure 1. In this case the program shows the RTTI information of an enumerated type, but you should try it with ordinal values and sets as well.



➤ *Figure 1: The list of possible values of an ordinal type as shown by ORDTYPE*

➤ *Listing 3: The ShowOrdinal helper routine*

```
procedure ShowOrdinal(pti: PTypeInfo; sList: TStrings);
var ptd: PTypeData;
begin
  // protect against misuse
  if not (pti^.Kind in
    [tkInteger, tkChar, tkEnumeration, tkSet, tkWChar]) then
    raise Exception.Create('Invalid type information');
  // get a pointer to the TTypeData structure
  ptd := GetTypeData(pti);
  // access the TTypeInfo structure
  sList.Add('Type Name: ' + pti^.Name);
  sList.Add('Type Kind: ' +
    GetEnumName(TypeInfo(TTypeKind),    Integer(pti^.Kind)));
  // access the TTypeData structure
  sList.Add('Implement: ' +
    GetEnumName(TypeInfo(TOrdType), Integer(ptd^.OrdType)));
  // a set has no min and max
  if pti^.Kind <> tkSet then begin
    sList.Add('Min Value: ' + IntToStr(ptd^.MinValue));
    sList.Add('Max Value: ' + IntToStr(ptd^.MaxValue));
  end;
  // add the enumeration base type
  // and the list of the values
  if pti^.Kind = tkEnumeration then begin
    sList.Add('Base Type: ' + (ptd^.BaseType)^.Name);
    sList.Add('');
    sList.Add('Values...');
    ListEnum(pti, sList);
  end;
  // show RRTI info about set base type
  if  pti^.Kind = tkSet then begin
    sList.Add('');
    sList.Add('Set base type information...');
    ShowOrdinal(ptd^.CompType, sList);
  end;
end;
```

➤ *Listing 5*

```
procedure TForm1.Listbox1Click(
  Sender: TObject);
var
  pti: PTypeInfo;
begin
  pti := PTypeInfo(
    ListBox1.Items.Objects[
     Listbox1.ItemIndex]);
  ListBox2.Items.Clear;
  ShowOrdinal(pti,
    ListBox2.Items);
end;
```

Special Functions For Cursors And Colors

If you run ORDTYPE and look at the `TColor` and `TCursor` data types, you'll notice that they are simply indicated as numeric values, not enumerated types, although their behavior in the Object Inspector might seem to indicate that. However, the programs displays the list of cursors or colors anyway (see Figure 2). How can this happen?

➤ *Listing 6*

```
if ListBox1.Items [ListBox1.ItemIndex] = 'TColor' then begin
  ListBox2.Items.Add('');
  ListBox2.Items.Add('Values...');
  GetColorValues(AddToList);
end;
if ListBox1.Items [ListBox1.ItemIndex] = 'TCursor' then begin
  ListBox2.Items.Add('');
  ListBox2.Items.Add('Values...');
  GetCursorValues(AddToList);
end;
```

➤ *Listing 7: The ShowMethod helper routine*

```
procedure ShowMethod(pti: PTypeInfo; sList: TStrings);
var
  ptd: PTypeData;
  pParam: PParamData;
  nParam: Integer;
  Line: string;
  pTypeString, pReturnString: ^ShortString;
begin
  // protect against misuse
  if pti^.Kind <> tkMethod then
    raise Exception.Create('Invalid type information');
  // get a pointer to the TTypeData structure
  ptd := GetTypeData(pti);
  // 1: access the TTypeInfo structure
  sList.Add('Type Name: ' + pti^.Name);
  sList.Add('Type Kind: ' + GetEnumName(TypeInfo(TTypeKind), Integer(pti^.Kind)));
  // 2: access the TTypeData structure
  sList.Add('Method Kind: ' +
    GetEnumName(TypeInfo(TMethodKind), Integer(ptd^.MethodKind)));
  sList.Add('Number of parameter: ' + IntToStr(ptd^.ParamCount));
  // 3: access to the ParamList. Get initial pointer and reset parameters counter
  pParam := PParamData(@(ptd^.ParamList));
  nParam := 1;
  // loop until all parameters are done
  while nParam <= ptd^.ParamCount do begin
    // read the information
    Line := 'Param ' + IntToStr(nParam) + ' > ';
    // add type of parameter
    if pfVar in pParam^.Flags then
      Line := Line + 'var ';
    if pfConst in pParam^.Flags then
      Line := Line + 'const ';
    // get the parameter name
    Line := Line + pParam^.ParamName + ': ';
    // one more type of parameter
    if pfArray in pParam^.Flags then
      Line := Line + ' array of ';
    // the type name string must be located...
    // moving a pointer past the params and the string (including its size byte)
    pTypeString := Pointer(Integer(pParam) +
      sizeof(TParamFlags) + Length(pParam^.ParamName) + 1);
    // add the type name
    Line := Line + pTypeString^;
    // finally, output the string
    sList.Add(Line);
    // move pointer to next structure, past the two strings (including size byte)
    pParam := PParamData(Integer(pParam) + sizeof(TParamFlags) +
      Length(pParam^.ParamName) + 1 + Length(pTypeString^) + 1);
    // increase the parameters counter
    Inc(nParam);
  end;
  // show the return type if a function
  if ptd^.MethodKind = mkFunction then begin
    // at the end, instead of a param data, there is the return string
    pReturnString := Pointer(pParam);
    sList.Add('Returns > ' + pReturnString^);
  end;
end;
```

Of course, it is possible to retrieve a lot of information about these two data types, using special functions. To get a list of cursors or a list of colors, you can call the `GetCursorValues` and `GetColorValues` procedures respectively. These two procedures behave like enumerated functions: they require a method as parameter, which is called once for each value in the list. Here is the method passed to the procedures:

```
procedure TForm1.AddToList(
  S: String);
begin
  ListBox2.Items.Add(S);
end;
```

and Listing 6 shows the code added to the `ListBox1Click` method of the form.

To handle colors and cursors there are other special support functions, such as `ColorToString` and `CursorToString`, to turn a numeric value into a string and `StringToColor` and `StringToCursor` to obtain the numeric value corresponding to a string.

RTTI For Method Pointers

Accessing the RTTI information related to ordinal types is not too difficult, once you understand the basic concepts. Looking into method pointer types gets slightly more complex. The example I am going to build (METHTYPE) is very similar to the ORDTYPE program.

Again, the form has two list boxes, one with the type names and the other displaying the RTTI information. The first list box is filled as before by calling the `AddType` method, which adds the type name to the list and type data to the corresponding item of the `Objects` array property. However, the initialization now takes place with method pointers, as in:

```
AddType(
  TypeInfo (TNotifyEvent));
AddType(
  TypeInfo (TFindMethodEvent));
```

When the user selects a new item in this listbox, the second listbox is filled with information, by calling

the ShowMethod helper routine. So all the code is actually in one place: ShowMethod. This is another complex procedure, as you can see from the source code in Listing 7.

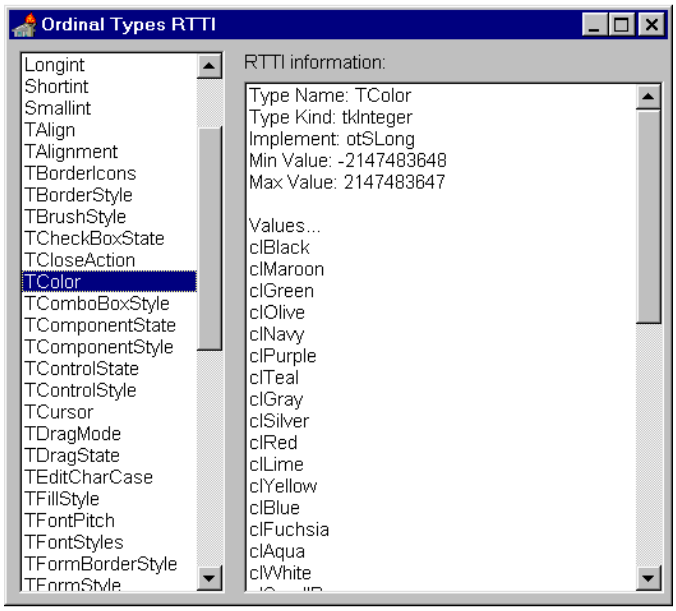
The first part of ShowMethod resembles ShowOrdinal in the last example. Once we've displayed the basic type information, we can start looking at the specific data. In this case we retrieve the kind of method (either procedure or function) and the number of parameters. Accessing the parameters information is where things get a little bit confused. You can go back to the code of the TTypeData structure (Listing 2) to see the original description. Since the actual structure of the parameters data was commented, I've re-defined it as:

```
type
  TParamData = record
    Flags: TParamFlags;
    ParamName: ShortString;
    TypeName: ShortString;
  end;
  PParamData = ^TParamData;
```

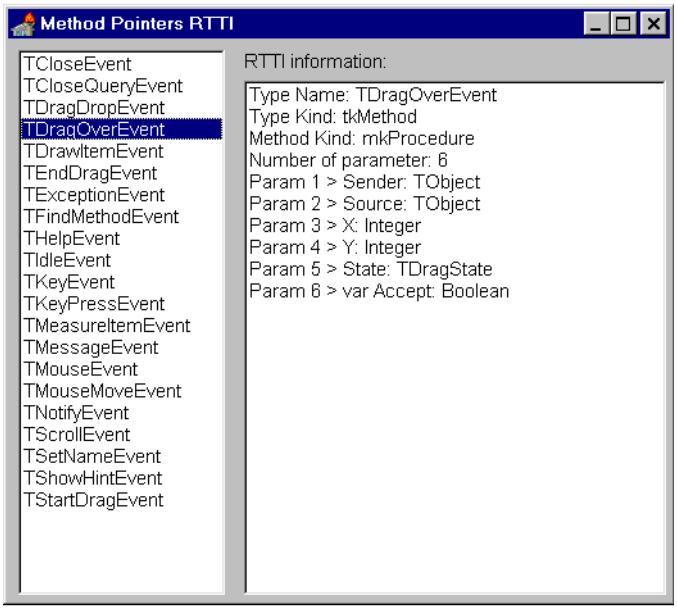
In practice, for each parameter there is the flag, followed by two strings. The problem is that the physical length of each string corresponds to the actual length of its data. So we have to look to the length byte at the beginning (these are short strings, that is traditional Pascal strings) to see where the next string begins. In practice, we cannot use the TypeName field at all, since the type definition suggests a fixed string length. Instead, once we have a pointer to the beginning of the structure, we can get a pointer to the TypeName string with the following code:

```
pTypeString :=
  Pointer(Integer(pParam) +
  sizeof(TParamFlags) +
  Length(pParam^.ParamName) +
  1);
```

Besides the cast to integer to perform pointer arithmetic and the final cast back to a generic pointer, this statement adds to the pParam pointer the size of the flags, plus the length of the string, plus its length byte. Similar pointer arithmetic is done to move to the data structure of the next parameter, or to the final string holding the return type which is available at the end of the parameters list.



➤ *Figure 2: The list of colors displayed by ORDTYPE*



➤ *Figure 3: METHTYPE program, showing RTTI information for method pointer types*

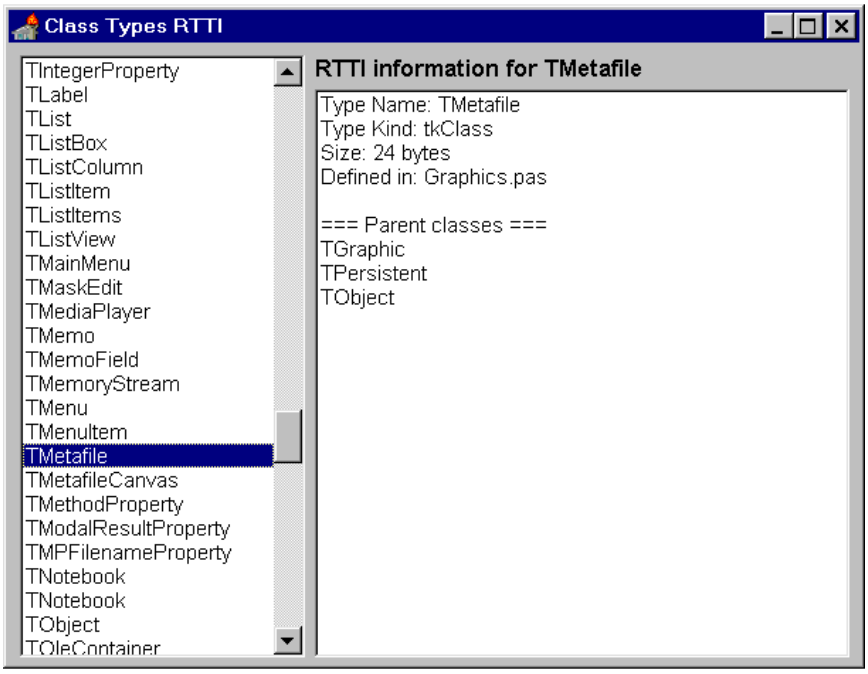
You can see an example of the output of the METHTYPE program in Figure 3. Most of the methods listed refer to procedures. Only the THelpEvent type refers to a function. Of course, you can look into the Delphi help files or the VCL source code and add more method pointer types to the list.

RTTI For Classes

After looking at the RTTI information for ordinal types and method pointers, we are now ready to look into the last and most complex area: RTTI information for classes. We'll build an example with the same user interface as the last two and write a helper routine to show class RTTI data inside a TStrings object.

The first list box has class names and pointers to the RTTI information, but this time I've added more than 200 classes. Selecting an item from the list box will call yet another helper routine, ShowClass, which has the same structure as the procedures we've already seen. The first part (see Listing 8) should be familiar, the rest requires some comments.

In Listing 8 I use the ClassType member of the TTypeData structure to show the size of the instances of that class. Having this information we can show additional class data.



➤ *Figure 4*

```
procedure ShowClass(pti: PTypeInfo; sList: TStrings);
var
  ptd: PTypeData;
  ParentClass: TClass;
begin
  // protect against misuse
  if pti^.Kind <> tkClass then
    raise Exception.Create('Invalid type information');
  // get a pointer to the TTypeData structure
  ptd := GetTypeData(pti);
  // access the TTypeInfo structure
  sList.Add('Type Name: ' + pti^.Name);
  sList.Add('Type Kind: ' +
    GetEnumName(TypeInfo(TTypeKind), Integer(pti^.Kind)));
  // access the TTypeData structure
  sList.Add('Size: ' + IntToStr(ptd^.ClassType.InstanceSize) + ' bytes');
  sList.Add('Defined in: ' + ptd^.UnitName + '.pas');
  // add the list of parent classes (if any)
  ParentClass := ptd^.ClassType.ClassParent;
  if ParentClass <> nil then begin
    sList.Add('');
    sList.Add('=== Parent classes ===');
    while ParentClass <> nil do begin
      sList.Add(ParentClass.ClassName);
      ParentClass := ParentClass.ClassParent;
    end;
  end;
end;
```

➤ *Listing 8: The ShowClass helper routine (in its first version)*

```
procedure TForm1.ListBox2Click(Sender: TObject);
var Text: string;
    Index: Integer;
begin
  // get the current item
  Text := ListBox2.Items[ListBox2.ItemIndex];
  // search the first listbox
  Index := ListBox1.Items.IndexOf(Text);
  // if found, it was a parent class: show RTTI
  if Index >= 0 then begin
    ListBox1.ItemIndex := Index;
    ListBox1Click(ListBox1);
  end;
end;
```

➤ *Listing 9*

What I find more interesting to do, instead, is to show the list of parent classes.

Notice that amongst the RTTI information we can show the unit which defined the data structure, available only for classes and used by Delphi to automatically add referenced units as soon as you add a component to a form.

In Figure 4 you can see an example of the output of a class with its parent classes listed. What is nice in this example is that a user can simply click on one of the names of the parent classes to jump to the RTTI information for that class. This is accomplished by checking if the first list box contains a string corresponding to the string a user has clicked onto (see Listing 9).

Besides showing some interesting RTTI for class types, CLASSTYP also has a neat user interface. But there is more we want to add. As discussed in the next section, this program is capable of showing information about properties as well.

Getting A Properties List

If you look back at the `TTypeData` record definition, we find:

```
type
  TPropData = packed record
    PropCount: Word;
    PropList: record end;
   {PropList:
      array[1..PropCount]
      of TPropInfo}
  end;
```

The `TPropData` structure is actually seldom used. To access this information we use one of the following routines defined in TYPINFO.PAS:

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

These two routines fill the `PropList` parameter with a list of pointers to properties RTTI information. `GetPropInfos` retrieves the properties, while `GetPropList` allows you to

specify a filter on the kind of properties you are interested in and returns the number of properties matching the criteria. The information we are accessing is available only for `published` properties: `protected` or `public` properties generate no RTTI.

The `PropList` parameter is similar to the last parameter of the `TPropData` record above:

```
type
  PPropList = ^TPropList;
  TPropList = array[0..16379]
    of PPropInfo;
```

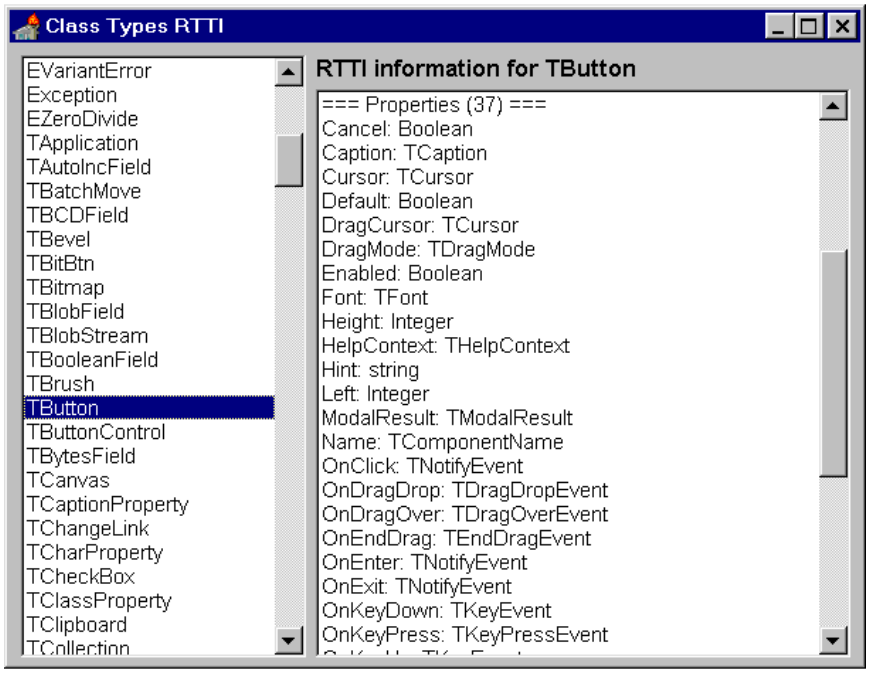
In practice, `TPropList` is a list of pointers to properties RTTI information and `PPropList` is a pointer to the list of pointers. These definitions and the `TPropData` record all refer to another data type, `TPropInfo`. A pointer to this same data type is returned by a third property information access function, `GetPropInfo`, which extracts the `PProfInfo` pointer for a specific property passed by name:

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

This function is used to access a specific property, passed by string. For the moment, we can turn our attention to the `TPropInfo` data structure, shown in Listing 10.

This structure reveals a lot of information about properties. It includes the name of the property, the index of its name (probably referring to a list of names, I guess this is an optimization to save memory), an index of the property and a pointer to the RTTI information of the property type.

Then there are three pointers to the methods used to operate on the property (if defined). These are the methods in the read, write, and stored sections of the property definition. We can use them to test if they are defined and to retrieve the memory address, the method pointer. Unfortunately we cannot get their names, because it is very uncommon to use published methods for property access.



➤ *Figure 5: CLASSTYP*

```
type
  PPropInfo = ^TPropInfo;
  TPropInfo = packed record
    PropType: PTypeInfo; // property type RTTI
    GetProcn: Pointer; // read method
    SetProc: Pointer; // write method
    StoredProc: Pointer; // store method
    Index: Integer; // property index
    Default: Longint; // default value (odd type)
    NameIndex: SmallInt; // index of the name
    Name: ShortString; // name
  end;
```

➤ *Listing 10*

```
var
  ppi: PPropInfo;
  pProps: PPropList;
  nProps, I: Integer;
...
// add the list of properties (if any)
nProps := ptd^.PropCount;
if nProps > 0 then begin
  // format the initial output
  sList.Add('');
  sList.Add('=== Properties(' + IntToStr(nProps) + ') ===');
  // allocate the required memory
  GetMem(pProps, sizeof(PPropInfo) * nProps);
  // protect the memory allocation
  try
    // fill the TPropList structure pointed to by pProps
    GetPropInfos(pti, pProps);
    // sort the properties
    SortPropList(pProps, nProps);
    // show name and data type of each property
    for I := 0 to nProps - 1 do begin
      ppi := pProps [I];
      sList.Add(ppi^.Name + ': ' + ppi^.PropType.Name);
    end;
  finally
    // free the allocated memory
    FreeMem(pProps, sizeof(PPropInfo) * nProps);
  end;
end;
```

➤ *Listing 11: Additions to the ShowClass method used to show properties information*

The final information is the default value of the property. The strange thing here is the data type of this field: `LongInt`. In fact, the actual meaning of this value is determined by the data type of the property. So it is possible to use it only after typecasting it to the proper type (although I am not going to use it in the example).

After this long introduction we are ready to look into the final part of the `ShowClass` procedure for our CLASSTYP example. The code reads the number of properties from the `TPropData` structure and if it finds any it outputs an initial line, then displays a line describing each property. The properties information is retrieved by calling the `GetPropInfos` function, passing as the `PPropInfo` parameter a block of memory allocated with the proper size (the size of the pointers of the list multiplied by the number of properties). The code added to `ShowClass` is in Listing 11, with a couple of specific local variables at the beginning.

Inside the `try-finally` block (also used to free the memory in case of an exception), we get the properties information, then output each name and data type inside a `for` loop. In between these statements we call `SortPropList` to sort the properties alphabetically.

I found `SortPropList` while looking at the `implementation` portion of the `TypInfo` unit. In fact it is not exported, but we can simply borrow its code.

That's all for the CLASSTYP program. You can see an example of the output with the list of properties in Figure 5. However, you should really try running the program to get an idea of its effect.

Conclusion

This example ends this article. We have delved into TYPINFO.PAS and built simple programs to show RTTI information for ordinal types, method pointers and classes. There is a lot more to say about RTTI. Actually the most important use of these constructs is to access component properties dynamically, by name, rather than with the usual compiled code, but I'll have to leave that to a future article...

Marco Cantù, author of the book *Mastering Delphi 2* (Sybex), is working an a new advanced book titled *Delphi Developer's Handbook* (SYBEX). Besides writing, he enjoys speaking at conferences and consulting on advanced features of Delphi. Contact him at 100273.2610@compuserve.com, or check his home page at
http://ourworld.compuserve.com/homepages/marcocantu