CHAPTER

10

Component Architecture: VCL and CLX

IN THIS CHAPTER

Few will recall Borland's first *Object Windows Library (OWL)*, which was introduced with Turbo Pascal for Windows. OWL ushered in a drastic simplification over traditional Windows programming. OWL objects automated and streamlined many tedious tasks you otherwise were required to code yourself. No longer did you have to write huge `case` statements to capture messages or big chunks of code to manage Windows classes; OWL did this for you. On the other hand, you had to learn a new programming methodology—object-oriented programming.

Then, with Delphi 1, Borland introduced *Visual Component Library (VCL)*. The VCL was based on an object model similar to OWL's in principle but radically different in implementation. The VCL in Delphi 6 is pretty much the same as its predecessors in all previous versions of Delphi.

With Delphi 6, Borland, once again, introduced a new technology, *Component Library for Cross-Platform (CLX)*. According to Borland, CLX is "the next-generation component library and framework for developing native Linux and Windows applications and reusable components."

Both the VCL and CLX are designed specifically to work within Delphi's visual environment. Instead of creating a window or dialog box and adding its behavior in code, you modify the behavioral and visual characteristics of components as you design your program visually.

The level of knowledge required about the VCL/CLX really depends on how you use them. First, you must realize that there are two types of Delphi developers: applications developers and visual component writers. *Applications developers* create complete applications by interacting with the Delphi visual environment (a concept nonexistent in many other frameworks). These people use the VCL/CLX to create their GUI and other elements of their application such as database connectivity. C*omponent writers*, on the other hand, expand the existing VCL/CLX by developing more components. Such components are made available through third-party companies.

Whether you plan to create applications with Delphi or to create Delphi components, understanding the VCL/CLX is essential. An applications developer should know which properties, events, and methods are available for each component. Additionally, it's advantageous to fully understand the object model inherent in a Delphi application that's provided by the VCL/CLX. A common problem we see with Delphi developers is that they tend to fight the tool—a symptom of not understanding it completely. Component writers take this knowledge one step further to determine whether to write a new component or to extend an existing one by knowing how VCL/CLX works internally: how they handle messages, notifications, component ownership, parenting/ownership issues, property editors, and so on.

This chapter introduces you to the VCL/CLX. It discusses the component hierarchy and explains the purpose of the key levels within the hierarchy. It also discusses the purposes of the common properties, methods, and events that appear at the different component levels. Finally, we complete this chapter by covering *Runtime Type Information (RTTI)*.

More on the New CLX

CLX, the new cross platform library, is actually composed of four pieces. These are explained in Table 10.1.

TABLE 10.1 CLX Parts (from Delphi 6 Online Help)

Part	*Description*
VisualCLX	Native cross-platform GUI components and graphics. The components in this area might differ on Linux and Windows.
DataCLX	Client data-access components. The components in this area are a subset of the local, client/server, and n-tier based on client datasets. The code is the same on Linux and Windows.
NetCLX	Internet components including Apache DSO and CGI Web Broker. These are the same on Linux and Windows.
RTL	Runtime Library up to and including Classes.pas. The code is the same on Linux and Windows. Under Linux, this file is `BaseRTL`.

VisualCLX sits on top of the Qt framework from Trolltech. Qt is pronounced "cute" by most people, although Trolltech will tell you that it's pronounced "kyu-tee." This framework currently runs under Linux and Windows. VisualCLX is discussed in this chapter, and we cover the other CLX elements in other chapters.

What Is a Component?

Components are the building blocks developers use to design the user interface and provide some non-visual capability to their applications. As far as applications developers are concerned, a component is something developers get from the Component Palette and place on their forms. From there, they can manipulate the various properties and add event handlers to give the component a specific appearance or behavior. From the perspective of a component writer, components are objects in Object Pascal code. These objects can encapsulate the behavior of elements provided by the system (such as the standard Windows controls). Other objects can introduce entirely new visual or non-visual elements; in which case a component's code makes up the entire behavior of the component.

The complexity of components varies widely. Some components are simple; others encapsulate elaborate tasks. There's no limit to what a component can do or be made up of. You can have a simple component such as a `TLabel`, or you can have a much more complex component that encapsulates the complete functionality of a spreadsheet.

The key to understanding the VCL/CLX is to know what types of components exist. You should understand the common elements of components. You should also understand the component hierarchy and the purpose of each level within the hierarchy. The following sections provide this information.

Component Hierarchy

Figures 10.1 and 10.2 show the VCL and CLX hierarchies, respectively. You'll see that there are many similarities between both the VCL and CLX.

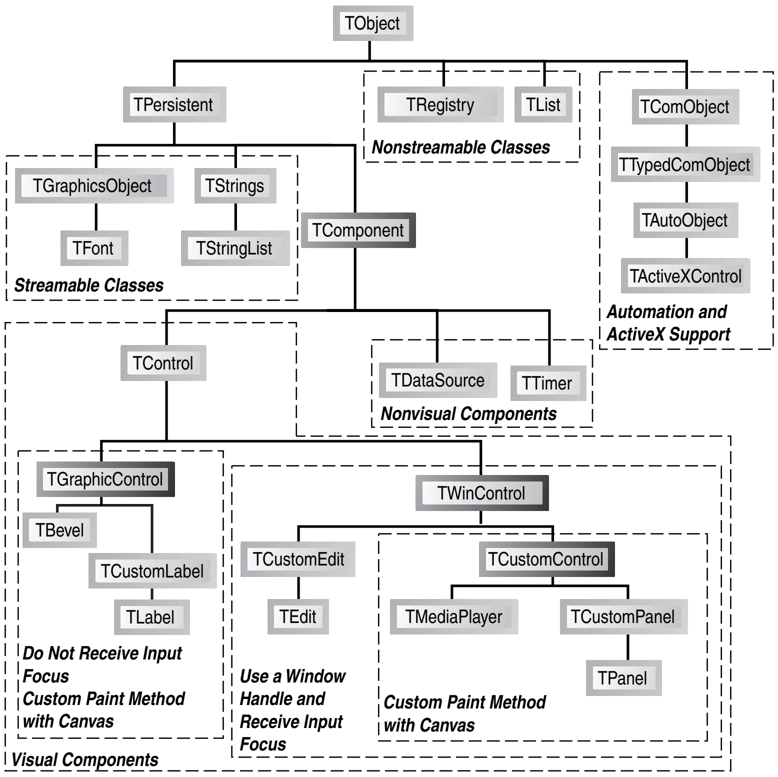


FIGURE 10.1
The VCL hierarchy.

Two types of components exist: nonvisual and visual.

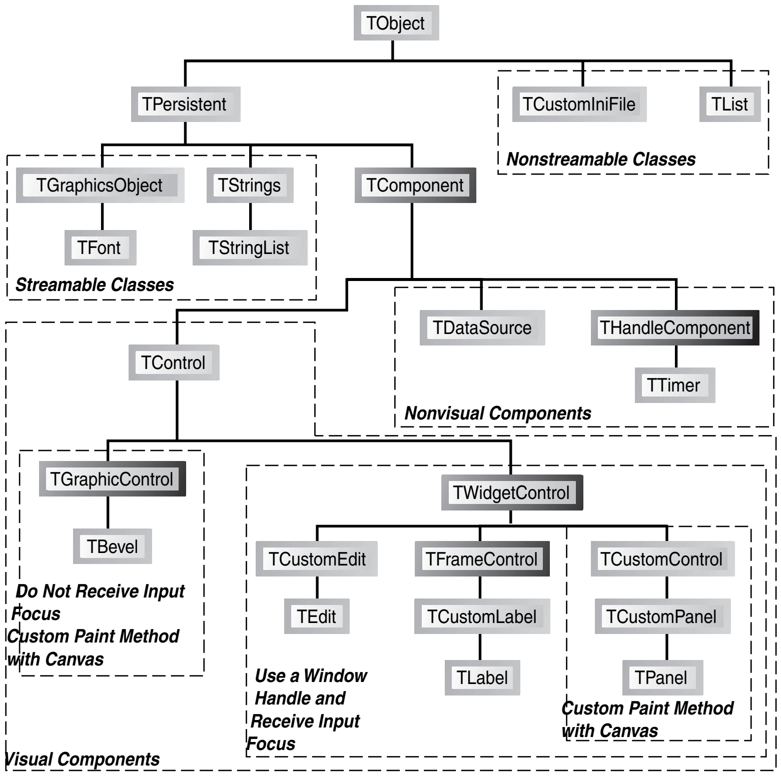


FIGURE 10.2
The CLX hierarchy.

Nonvisual Components

Nonvisual components aren't visible to the end user. These components encapsulate behavior and allow the developer to modify certain characteristics of that component through the Object Inspector at design time by modifying its properties and providing event handlers for its events. Examples of such components are `TOpenDialog`, `TTable`, and `TTimer`. As Figures 10.1 and 10.2 indicate, these nonvisual components descend directly from `TComponent`.

Visual Components

Visual components, as the name implies, are components that the end user sees. Visual components add visibility and behavior, but not necessarily interaction. These components directly descend from `TControl`. In fact, `TControl` is the class that introduces properties and methods that have to do with visibility such as Top, Left, Color, and so forth.

NOTE

You'll often see the terms *component* and *control* used interchangeably, although they're not always the same. A *control* refers to a visual user-interface element. In Delphi, controls are always components because they descend from the `TComponent` class. *Components* are the objects whose basic behavior allows them to appear on the Component Palette and be manipulated in the form designer. Components are of the type `TComponent` and aren't always controls—that is, they aren't always visual user-interface elements.

Visual components come in two flavors—those that can have focus and those that cannot.

Visible Controls That Gain Focus

Certain types of controls gain user focus. By this, we mean that the user can manipulate such controls. These types of controls are descendants of `TWinControl` (VCL) or `TWidgetControl` (CLX). `TWinControl` descendants are wrappers around Windows controls, whereas `TWidgetControl` descendants are wrappers around Qt screen objects. Characteristics of these controls are as follows:

- They can get focus and do things such as handle keyboard events.
- The user can interact with them.
- They can be containers (parents) to other controls.
- They have an associated handle (VCL) or widget (CLX).

NOTE

Both `TWinControl` and `TWidgetControl` have a property named `Handle`. `TWinControl's Handle` refers to the underlying Windows `Handle` for the control. `TWidgetControl's Handle` refers to the underlying Qt object pointer (widget). Both are named `Handle` for backward compatibility and cross compilation between CLX and VCL applications.

In Chapters 11–14, you'll learn much more about `TWinControls` and `TWidgetControls` as you learn how to create components for both VCL and CLX.

Handles

Handles are 32-bit numbers issued by Win32 that refer to certain object instances. The term *objects* here refers to Win32 objects, not Delphi objects. There are different types of objects under Win32: kernel objects, user objects, and GDI objects. Kernel objects apply to items such as events, file-mapping objects, and processes. User objects refer to window objects such as edit controls, list boxes, and buttons. GDI objects refer to bitmaps, brushes, fonts, and so on.

In the Win32 environment, every window has a unique handle. Many Windows API functions require a handle so that they know the window on which they are to perform the operation. Delphi encapsulates much of the Win32 API and performs handle management. If you want to use a Windows API function that requires a window handle, you must use descendants of `TWinControl` and `TCustomControl`, which both have a `Handle` property.

Visible Controls That Do Not Gain Focus

Other controls, although visible, don'tave the same characteristics as Windowed controls. These controls are for visibility only and are frequently referred to as *graphical* controls, which descend directly from `TGraphicControl` (see Figures 10.1 and 10.2).

Unlike windowed controls, graphical controls don't receive the input focus from the user. They are useful when you want to display something to the user but don't want the component to use up resources such as windowed controls. Graphical controls don't use Windows resources because they require no window handle (or CLX Gadget), which is also the reason they can't get focus. Examples of graphical controls are `TLabel` and `TShape`. Such controls can't serve as containers either; that is, they can't parent other controls placed on top of them. Other examples of graphical controls are `TImage`, `TBevel`, and `TPaintBox`.

The Component Structure

As we mentioned earlier, components are Object Pascal classes that encapsulate the functionality and behavior of elements developers use to add visual and behavioral characteristics to their programs. All components have a certain structure. The following sections discuss the makeup of Delphi components.

NOTE

Understand the distinction between a component and a class. A *component* is a class that can be manipulated within the Delphi environment. A *class* is an Object Pascal structure, as explained in Chapter 2, "The Object Pascal Language."

Properties

Chapter 2 introduced you to properties. Properties give the user an interface to a component's internal storage fields. Using properties, the component user can modify or read storage field values. Typically, the user doesn't have direct access to component storage fields because they're declared in the `private` section of a component's class definition.

Properties: Storage Field Accessors

Properties provide access to storage fields by either accessing the storage fields directly or through *access methods*. Take a look at the following property definition:

```
TCustomEdit = class(TWinControl)
private
  FMaxLength: Integer;
protected
  procedure SetMaxLength(Value: Integer);
...
published
  property MaxLength: Integer read FMaxLength write SetMaxLength default 0;
...
end;
```

The property `MaxLength` is the access to the storage field `FMaxLength`. The parts of a property definition consist of the property name, the property type, a `read` declaration, a `write` declaration, and an optional `default` value. The `read` declaration specifies how the component's storage fields are read. The `MaxLength` property directly reads the value from the `FMaxLength` storage field. The `write` declaration specifies the method by which the storage fields are assigned values. For the property `MaxLength`, the writer access method `SetMaxLength()` is used to assign the value to the storage field `FMaxLength`. A property can also contain a reader access method; in which case the `MaxLength` property would be declared as this:

```
property MaxLength: Integer read GetMaxLength write SetMaxLength default 0;
```

The reader access method `GetMaxLength()` would be declared as follows:

```
function GetMaxLength: Integer;
```

Property Access Methods

Access methods take a single parameter of the same type as the property. The purpose of the writer access method is to assign the value of the parameter to the internal storage field to which the property refers. The reason for using the method layer to assign values is to protect the storage field from receiving erroneous data as well as to perform various side effects, if required. For example, examine the implementation of the following `SetMaxLength()` method:

```
procedure TCustomEdit.SetMaxLength(Value: Integer);
begin
  if FMaxLength <> Value then
  begin
    FMaxLength := Value;
    if HandleAllocated then SendMessage(Handle, EM_LIMITTEXT, Value, 0);
  end;
end;
```

This method first checks to verify that the component user isn't attempting to assign the same value as that which the property already holds. If not, it makes the assignment to the internal storage field `FMaxLength` and then calls the `SendMessage()` function to pass the `EM_LIMITTEXT` Windows message to the window that the `TCustomEdit` encapsulates. This message limits the amount of text that a user can enter into an edit control. Calling `SendMessage()` in the property's writer access method is known as a *side effect* when assigning property values.

Side effects are any actions affected by the assignment of a value to a property. In assigning a value to the `MaxLength` property of `TCustomEdit`, the side effect is that the encapsulated edit control is given an entry limit. Side effects can be much more sophisticated than this.

One key advantage to providing access to a component's internal storage fields through properties is that the component writer can change the implementation of the field access without affecting the behavior for the component user.

A reader access method, for example, can change the type of the returned value to something different from the type of the storage field to which the property refers.

Another fundamental reason for the use of properties is to make modifications available to them during design time. When a property appears in the `published` section of a component's declaration, it also appears in the Object Inspector so that the component user can make modifications to this property.

You learn more about properties and how to create them and their access methods in Chapters 11, "VCL Component Building," and 13, "CLX Component Development," for VCL and CLX, respectively.

Types of Properties

The standard rules that apply to Object Pascal data types apply to properties as well. The important point about properties is that their types also determine how they're edited in the Object Inspector. Properties can be of the types shown in Table 10.2. For more detailed information, look up "properties" in the online help.

TABLE 10.2 Property Types

Property Type	*Object Inspector Treatment*
Simple	Numeric, character, and string properties appear in the Object Inspector as numbers, characters, and strings, respectively. The user can type and edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) display the value as defined in the source code. The user can cycle through the possible values by double-clicking the Value column. There's also a drop-down list that shows all possible values of the enumerated type.
Set	Properties of set types appear in the Object Inspector grouped as a set. By expanding the set, the user can treat each element of the set as a Boolean value: `True` if the element is included in the set and `False` if it's not included.
Object	Properties that are themselves objects often have their own property editors. However, if the object that's a property also has published properties, the Object Inspector allows the user to expand the list of object properties and edit them individually. Object properties must descend from `TPersistent`.
Array	Array properties must have their own property editors. The Object Inspector has no built-in support for editing array properties.

Methods

Because components are objects, they can therefore have methods. You've already seen information on object methods in Chapter 2 (that information is not repeated here). The later section "The Visual Component Hierarchy" describes some of the key methods of the different component levels in the component hierarchy.

Events

Events are occurrences of an action, typically a system action such as a button control click or a keypress on a keyboard. Components contain special properties called *events*; component users can plug code into the event (called *event handlers*) that executes when the event is invoked.

Plugging Code into Events at Design Time

If you look at the events page of a `TEdit` component, you'll find events such as `OnChange`, `OnClick`, and `OnDblClick`. To component writers, events are really pointers to methods. When users of a component assign code to an event, they create an *event handler*. For example, when you double-click an event in the Object Inspector's events page for a component, Delphi generates a method to which you add your code, such as the following code for the `OnClick` event of a `TButton` component:

```
TForm1 = class(TForm)
  Button1: Tbutton;
  procedure Button1Click(Sender: TObject);
end;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Event code goes here }
end;
```

This code is generated by Delphi.

Plugging Code into Events at Runtime

It becomes clear how events are method pointers when you assign an event handler to an event programmatically. For example, to link your own event handler to an `OnClick` event of a `TButton` component, you first declare and define the method you intend to assign to the button's `OnClick` event. This method might belong to the form that owns the `TButton` component, as shown here:

```
TForm1 = class(TForm)
  Button1: TButton;
...
private
  MyOnClickEvent(Sender: TObject); // Your method declaration
end;
...
{ Your method definition below }
procedure TForm1.MyOnClickEvent(Sender: TObject);
begin
  { Your code goes here }
end;
```

The preceding example shows a user-defined method called `MyOnClickEvent()` that serves as the event handler for `Button1.OnClick`. The following line shows how you assign this method to the `Button1.OnClick` event in code, which is usually done in the form's `OnCreate` event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.OnClick := MyOnClickEvent;
end;
```

This technique can be used to add different event handlers to events, based on various conditions in your code. Additionally, you can disable an event handler from an event by assigning `nil` to the event, as shown here:

```
Button1.OnClick := nil;
```

Assigning event handlers at runtime is essentially what happens when you create an event handler through Delphi's Object Inspector—except that Delphi generates the method declaration. You can't just assign any method to a particular event handler. Because event properties are method pointers, they have specific method signatures, depending on the type of event. For example, an `OnMouseDown` method is of the type `TMouseEvent`, a procedure definition shown here:

```
TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; Shift:
  TShiftState; X, Y: Integer) of object;
```

Therefore, the methods that become event handlers for certain events must follow the same signature as the event types. They must contain the same type, number, and order of parameters.

Earlier, we said that events are properties. Similar to data properties, events refer to private data fields of a component. This data field is of the procedure type, such as `TMouseEvent`. Examine this code:

```
TControl = class(TComponent)
private
  FOnMouseDown: TMouseEvent;
protected
  property OnMouseDown: TMouseEvent read FOnMouseDown write FOnMouseDown;
public
end;
```

Recall the discussion of properties and how they refer to private data fields of a component. You can see how events, being properties, refer to private method pointer fields of a component.

You learn more about creating events and event handlers in Chapters 11 and 13.

Streamability

One characteristic of components is that they must have the capability to be streamed. *Streaming* is a way to store a component and information regarding its properties' values to a file. Delphi's streaming capabilities take care of all this for you. In fact, the DFM file created by Delphi is nothing more than a resource file containing the streamed information on the form and its components as an RCDATA resource. As a component writer, however, you must sometimes go beyond what Delphi can do automatically. The streaming mechanism of Delphi is explained in greater depth in Chapter 12, "Advanced VCL Component Building."

Ownership

Components have the capability of owning other components. A component's owner is specified by its `Owner` property. When a component owns other components, it's responsible for freeing the components it owns when it's destroyed. Typically, the form owns all components that appear on it. When you place a component on a form in the form designer, the form automatically becomes the component's owner. When you create a component at runtime, you must pass the ownership of the component to the component's `Create` constructor; it's assigned to the new component's `Owner` property. The following line shows how to pass the form's implicit `Self` variable to a `TButton.Create()` constructor, thus making the form the owner of the newly created component:

```
MyButton := TButton.Create(self);
```

When the form is destroyed, the `TButton` instance to which `MyButton` refers is also destroyed. This is handled internally in the VCL. Essentially, the form iterates through the components referred to by its `Components` array property (explained in more detail shortly) and destroys them.

It's possible to create a component without an owner by passing `nil` to the component's `Create()` method. However, when this is done, it's your responsibility to destroy the component programmatically. The following code shows this technique:

```
MyTable := TTable.Create(nil)
try
  { Do stuff with MyTable }
finally
  MyTable.Free;
end;
```

When using this technique, you should use a `try..finally` block to ensure that you free up any allocated resources if an exception is raised. You wouldn't use this technique except in specific circumstances when it's impossible to pass an owner to the component.

Another property associated with ownership is the `Components` property. The `Components` property is an array property that maintains a list of all components belonging to a component. For example, to loop through all the components on a form to show their classnames, execute the following code:

```
var
  i: integer;
begin
  for i := 0 to ComponentCount - 1 do
     ShowMessage(Components[i].ClassName);
end;
```

Obviously, you'll probably perform a more meaningful operation on these components. The preceding code merely illustrates the technique.

Parenthood

Not to be confused with ownership is the concept of *parenthood*. Components can be *parents* to other components. Only windowed components such as `TWinControl` and `TWidgetControl` descendants can serve as parents to other components. Parent components are responsible for calling the child component methods to force them to draw themselves. Parent components are responsible for the proper painting of child components. A component's parent is specified through its `Parent` property.

A component's parent doesn't necessarily have to be its owner. It's perfectly legal for a component to have different parents and owners.

The Visual Component Hierarchy

Remember from Chapter 2 that the abstract class `TObject` is the base class from which all classes descend (see Figures 10.1 and 10.2).

As a component writer, you don't descend your components directly from `TObject`. The VCL already has `TObject` class descendants from which your new components can be derived. These existing classes provide much of the functionality you require for your own components. Only when you create noncomponent classes do your classes descend from `TObject`.

`TObject`'s `Create()` and `Destroy()` methods are responsible for allocating and deallocating memory for an object instance. In fact, the `TObject.Create()` constructor returns a reference to the object being created. `TObject` has several functions that return useful information about a specific object.

The VCL uses most of `TObject`'s methods internally. You can obtain useful information about an instance of a `TObject` or `TObject` descendant such as the instance's class type, classname, and ancestor classes.

CAUTION

Use `TObject.Free` instead of `TObject.Destroy`. The `free` method calls `destroy` for you but first checks to see whether the object is `nil` before calling `destroy`. This method ensures that you won't generate an exception by attempting to destroy an invalid object.

The `TPersistent` Class

The `TPersistent` class descends directly from `TObject`. The special characteristic of `TPersistent` is that objects descending from it can read their properties from and write them to a stream after they're created. Because all components are descendants of `TPersistent`, they are all streamable. `TPersistent` defines no special properties or events, although it does define some methods that are useful to both the component user and writer.

`TPersistent` Methods

Table 10.3 lists some methods of interest defined by the `TPersistent` class.

TABLE 10.3 Methods of the `TPersistent` Class

Method	*Purpose*
`Assign()`	This public method allows a component to assign to itself the data associated with another component.
`AssignTo()`	This protected method is where `TPersistent` descendants must implement the VCL definition for `AssignTo()`. `TPersistent` raises an exception when this method is called. `AssignTo()` is where a component can assign its data values to another instance or class—the reverse of `Assign()`.
`DefineProperties()`	This protected method allows component writers to define how the component stores extra or unpublished properties. This method is typically used to provide a way for a component to store data that's not of a simple data type, such as binary data.

The streamability of components is described in greater depth in Chapter 12, "Working with Files," from *Delphi 5 Developer's Guide* on the CD-ROM. For now, it's enough to know that components can be stored and retrieved from a disk file by means of streaming.

The `TComponent` Class

The `TComponent` class descends directly from `TPersistent`. `TComponent`'s special characteristics are that its properties can be manipulated at design time through the Object Inspector and that it can own other components.

Nonvisual components also descend from `TComponent` so that they inherit the capability to be manipulated at design time. A good example of a nonvisual `TComponent` descendant is the `TTimer` component. `TTimer` components aren't visual controls, but they are still available on the Component Palette.

TComponent defines several properties and methods of interest, as described in the following sections.

TComponent Properties

The properties defined by TComponent and their purposes are shown in Table 10.4.

TABLE 10.4 The Special Properties of TComponent

Property Name	*Purpose*
Owner	Points to the component's owner.
ComponentCount	Holds the number of components that the component owns.
ComponentIndex	The position of this component in its owner's list of components. The first component in this list has the value 0.
Components	A property array containing a list of components owned by this component. The first component in this list has the value 0.
ComponentState	This property holds the current state of a component of the type TComponentState. Additional information about TComponentState can be found in the online help and in Chapter 11.
ComponentStyle	Governs various behavioral characteristics of the component. csInheritable and csCheckPropAvail are two values that can be assigned to this property; both values are explained in the online help.
Name	Holds the name of a component.
Tag	An integer property that has no defined meaning. This property shouldn't be used by component writers—it's intended to be used by application writers. Because this value is an integer type, pointers to data structures—or even object instances—can be referred to by this property.
DesignInfo	Used by the form designer. Do not access this property.

TComponent Methods

TComponent defines several methods having to do with its capacity to own other components and to be manipulated on the form designer.

TComponent defines the component's Create() constructor, which was discussed earlier in this chapter. This constructor is responsible for creating an instance of the component and giving it an owner based on the parameter passed to it. Unlike TObject.Create(), TComponent.Create() is virtual. TComponent descendants that implement a constructor must declare the Create() constructor with the override directive. Although you can declare other constructors on a component class, `TComponent.Create()` is the only constructor VCL will use to create an instance of the class at design time and at runtime when loading the component from a stream.

The `TComponent.Destroy()` destructor is responsible for freeing the component and any resources allocated by the component.

The `TComponent.Destroying()` method is responsible for setting a component and its owned components to a state indicating that they are being destroyed; the `TComponent.DestroyComponents()` method is responsible for destroying the components. You probably won't have to deal with these methods.

The `TComponent.FindComponent()` method is handy when you want to refer to a component for which you know only the name. Suppose you know that the main form has a `TEdit` component named `Edit1`. When you don't have a reference to this component, you can retrieve a pointer to its instance by executing the following code:

```
EditInstance := FindComponent.('Edit1');
```

In this example, `EditInstance` is a `TEdit` type. `FindComponent()` will return `nil` if the name doesn't exist.

The `TComponent.GetParentComponent()` method retrieves an instance to the component's parent component. This method can return `nil` if there is no parent to a component.

The `TComponent.HasParent()` method returns a Boolean value indicating whether the component has a parent component. Note that this method doesn't refer to whether this component has an owner.

The `TComponent.InsertComponent()` method adds a component so that it's owned by the calling component; `TComponent.RemoveComponent()` removes an owned component from the calling component. You wouldn't normally use these methods because they're called automatically by the component's `Create()` constructor and `Destroy()` destructor.

The `TControl` Class

The `TControl` class defines many properties, methods, and events commonly used by visual components. For example, `TControl` introduces the capability for a control to display itself. The `TControl` class includes position properties such as `Top` and `Left` as well as size properties such as `Width` and `Height`, which hold the horizontal and vertical sizes. Other properties include `ClientRect`, `ClientWidth`, and `ClientHeight`.

`TControl` also introduces properties regarding appearances and accessibility, such as `Visible`, `Enabled`, and `Color`. You can even specify a font for the text of a `TControl` through its `Font` property. This text is provided through the `TControl` properties `Text` and `Caption`.

TControl also introduces some standard events, such as the mouse events OnClick, OnDblClick, OnMouseDown, OnMouseMove, and OnMouseUp. It also introduces drag events such as OnDragOver, OnDragDrop, and OnEndDrag.

TControl isn't very useful at the TControl level. You'll never create descendants of TControl.

Another concept introduced by TControl is that it can have a parent component. Although TControl might have a parent, its parent must be a TWinControl (VCL) or a TWidgetControl (CLX). Parent controls must be *windowed* controls. The TControl introduces the Parent property.

Most of Delphi's controls are derived from TControl's descendants: TWinControl and TWidgetControl.

The TWinControl and TWidgetControl

Standard controls descend from the classes TWinControl for VCL controls and TWidgetControl for CLX controls. These controls are the user-interface objects you see in most applications. Items such as edit controls, list boxes, combo boxes, and buttons are examples of these controls. Because Delphi encapsulates the behavior of standard controls instead of using Windows or Qt API level functions to manipulate them, you use the properties provided by each of the various control components.

The three basic characteristics of these controls are that they have a Windows handle, can receive input focus, and can be parents to other controls. CLX controls don't have a window handle; rather, they have an object pointer that accomplished the same thing. You'll find that the properties, methods, and events belonging to these controls support focus changing, keyboard events, drawing of controls, and other necessary functions.

An applications developer primarily uses TWinControl/TWidgetControl descendants. A component writer must understand these controls and their descendants in much greater depth.

TWinControl/TWidgetControl Properties

TWinControl and TWidgetControl define several properties applicable to changing the focus and appearance of the control. In the remaining text, we'll refer only to TWinControl although it will also be applicable to TWidgetControl.

The TWinControl.Brush property is used to draw the patterns and shapes of the control (See Chapter 8, "Graphics Programming with GDI and Fonts," in *Delphi 5 Developer's Guide* on this book's CD-ROM.)

TWinControl.Controls is an array property that maintains a list of all controls to which the calling TWinControl is a parent.

The `TWinControl.ControlCount` property holds the count of controls to which it is a parent.

`TWinControl.Ctl3D` is a property that specifies whether to draw the control using a three-dimensional appearance.

The `TWinControl.Handle` property corresponds to the handle of the Windows object that the `TWinControl` encapsulates. This is the handle you would pass to Win32 API functions requiring a window handle parameter.

`TWinControl.HelpContext` holds a help context number that corresponds to a help screen in a help file. This is used to provide context-sensitive help for individual controls.

`TWinControl.Showing` indicates whether a control is visible.

The `TWinControl.TabStop` property holds a Boolean value to determine whether a user can tab to the said control. The `TWinControl.TabOrder` property specifies where in the parent's list of tabbed controls the control exists.

`TWinControl` Methods

The `TWinControl` component also offers several methods that have to do with window creation, focus control, event dispatching, and positioning. There are too many methods to discuss in depth in this chapter; however, they're all documented in Delphi's online help. We'll list only those methods of particular interest in the following paragraphs.

Methods that relate to window creation, re-creation, and destruction apply mainly to component writers and are discussed in Chapter 11. These methods are `CreateParams()`, `CreateWnd()`, `CreateWindowHandle()`, `DestroyWnd()`, `DestroyWindowHandle()`, and `RecreateWnd()` for VCL. For CLX's `TWidgetControl`, these methods are `CreateWidget()`, `DestroyWidget()`, `CreateHandle()`, and `DestroyHandle()`.

Methods having to do with window focusing, positioning, and alignment are `CanFocus()`, `Focused()`, `AlignControls()`, `EnableAlign()`, `DisableAlign()`, and `ReAlign()`.

`TWinControl` Events

`TWinControl` introduces events for keyboard interaction and focus change. Keyboard events are `OnKeyDown`, `OnKeyPress`, and `OnKeyUp`. Focus-change events are `OnEnter` and `OnExit`. All these events are documented in Delphi's online help.

The `TGraphicControl` Class

`TGraphicControls`, unlike `TWinControls`, don't have a window handle and therefore can't receive input focus. They also can't be parents to other controls. `TGraphicControls` are used when you want to display something to the user on the form, but you don't want this control to function as a regular user-input control. The advantage of `TGraphicControls` is that they don't request a handle from Windows that uses up system resources. Additionally, not having a window handle means that `TGraphicControls` don't have to go through the convoluted Windows paint process. This makes drawing with `TGraphicControls` much faster than using the `TWinControl` equivalents.

`TGraphicControls` can respond to mouse events. Actually, the `TGraphicControl` parent processes the mouse message and sends it to its child controls.

`TGraphicControl` allows you to paint the control and therefore provides the property `Canvas`, which is of the type `TCanvas`. `TGraphicControl` also provides a `Paint()` method that its descendants must override.

The `TCustomControl` Class

You might have noticed that the names of some `TWinControl` descendants begin with `TCustom`, such as `TCustomComboBox`, `TCustomControl`, `TCustomEdit`, and `TCustomListBox`.

Custom controls have the same functionality as other `TWinControl` descendants, except that with specialized visual and interactive characteristics, custom controls provide you with a base from which you can derive and create your own customized components. You provide the functionality for the custom control to draw itself if you're a component writer.

Other Classes

Several classes aren't components but serve as supporting classes to the existing component. These classes are typically properties of other components and descend directly from `TPersistent`. Some of these classes are of the type `TStrings`, `TCanvas`, and `TCollection`.

The `TStrings` and `TStringLists` Classes

The `TStrings` abstract class gives you the capability to manipulate lists of strings that belong to a component such as a `TListBox`. `TStrings` doesn't actually maintain the memory for the strings (that's done by the native control that owns the `TStrings` class). Instead, `TStrings` defines the methods and properties to access and manipulate the control's strings without having to use the control's set of API level functions and messages.

Notice that we said `TStrings` is an abstract class. This means that `TStrings` doesn't really implement the code required to manipulate the strings—it just defines the methods that must be there. It's up to the descendant components to implement the actual string-manipulation methods.

To explain this point further, some examples of components and their `TStrings` properties are `TListBox.Items`, `TMemo.Lines`, and `TComboBox.Items`. Each of these properties is of the type

TStrings. You might wonder, if their properties are TStrings, how can you call methods of these properties when these methods have yet to be implemented in code? That's a good question. The answer is that, even though each of these properties is defined as TStrings, the variable to which the property refers (TListBox.FItems, for example) was instantiated as a descendant class. To clarify this, FItems is the private storage field for the Items property of TListBox:

```
TCustomListBox = class(TWinControl)
 private
   FItems: TStrings;
```

NOTE

Although the class type shown in the preceding code snippet is a TCustomListBox, the TListBox descends directly from TCustomListBox in the same unit and therefore has access to its private fields.

The unit StdCtrls.pas, which is part of the Delphi VCL, defines a descendant class TListBoxStrings, which is a descendant of TStrings. Listing 10.1 shows its definition.

LISTING 10.1 The Declaration of the TListBoxStrings Class

```
TListBoxStrings = class(TStrings)
  private
    ListBox: TCustomListBox;
  protected
    procedure Put(Index: Integer; const S: string); override;
    function Get(Index: Integer): string; override;
    function GetCount: Integer; override;
    function GetObject(Index: Integer): TObject; override;
    procedure PutObject(Index: Integer; AObject: TObject); override;
    procedure SetUpdateState(Updating: Boolean); override;
  public
    function Add(const S: string): Integer; override;
    procedure Clear; override;
    procedure Delete(Index: Integer); override;
    procedure Exchange(Index1, Index2: Integer); override;
    function IndexOf(const S: string): Integer; override;
    procedure Insert(Index: Integer; const S: string); override;
    procedure Move(CurIndex, NewIndex: Integer); override;
end;
```

`StdCtrls.pas` then defines the implementation of each method of this descendant class. When `TListBox` creates its class instances for its `FItems` variable, it actually creates an instance of this descendant class and refers to it with the `FItems` property:

```
constructor TCustomListBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
  // An instance of TListBoxStrings is created
  FItems := TListBoxStrings.Create;
  ...
end;
```

We want to make it clear that although the `TStrings` class defines its methods, it doesn't implement these methods to manipulate strings. The `TStrings` descendant class does the implementation of these methods. This is important if you're a component writer because you must know how to perform this technique as the Delphi components did it. It's always good to refer to the VCL or CLX source code to see how Borland performs these techniques when you're unsure.

If you're not a component writer but want to manipulate a list of strings, you can use the `TStringList` class, another descendant of `TStrings`, with which you can instantiate a completely self-contained class. `TStringList` maintains a list of strings external to components. The best part is that `TStringList` is totally compatible with `TStrings`, which means that you can directly assign a `TStringList` instance to a control's `TStrings` property. The following code shows how you can create an instance of `TStringList`:

```
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;
```

To add strings to this `TStringList` instance, do the following:

```
MyStringList.Add('Red');
MyStringList.Add('White');
MyStringList.Add('Blue');
```

If you want to add these same strings to both a `TMemo` component and a `TListBox` component, all you have to do is take advantage of the compatibility between the different components' `TStrings` properties and make the assignments in one line of code each:

```
Memo1.Lines.Assign(MyStringList);
ListBox1.Items.Assign(MyStringList);
```

You use the `Assign()` method to copy `TStrings` instances instead of making a direct assignment such as `Memo1.Lines := MyStringList`.

Table 10.5 shows some common methods of `TStrings` classes.

TABLE 10.5 Some Common `TStrings` Methods

`TStrings` *Method*	*Description*
`Add(const S: String): Integer`	Adds the string `S` to the string's list and returns the string's position in the list.
`AddObject(const S: string; AObject: TObject): Integer`	Appends both a string and an object to a string or string list object.
`AddStrings(Strings: TStrings)`	Copies strings from one `TStrings` to the end of its existing list of strings.
`Assign(Source: TPersistent)`	Replaces the existing strings with those specified by the `Source` parameter.
`Clear`	Removes all strings from the list.
`Delete(Index: Integer)`	Removes the string at the location specified by `Index`.
`Exchange(Index1, Index2: Integer)`	Switches the location of the two strings specified by the two index values.
`IndexOf(const S: String): Integer`	Returns the position of the string `S` on the list.
`Insert(Index: Integer; const S: String)`	Inserts the string `S` into the position in the list specified by `Index`.
`Move(CurIndex, NewIndex: Integer)`	Moves the string at the position `CurIndex` to the position `NewIndex`.
`LoadFromFile(const FileName: String)`	Reads the text file, `FileName`, and places its lines into the string list.
`SaveToFile(const FileName: string)`	Saves the string list to the text file, `FileName`.

The `TCanvas` Class

The `Canvas` property, of type `TCanvas`, is provided for windowed controls and represents the drawing surface of the control. `TCanvas` encapsulates what's called the *device context* of a window. It provides many of the functions and objects required for drawing to the window's surface. (Chapter 8, "Graphics Programming with GDI and Fonts," of *Delphi 5 Developer's Guide* on this book's CD-ROM goes into detail about the `TCanvas` class.)

Runtime Type Information

Back in Chapter 2 you were introduced to Runtime Type Information (RTTI). This chapter delves much deeper into the RTTI innards that will allow you to take advantage of RTTI beyond what you get in the normal usage of the Object Pascal language. In other words, we're going to show you how to obtain type information on objects and data types much similar to the way the Delphi IDE obtains the same information.

So how does RTTI manifest itself? You'll see RTTI at work in at least two areas with which you normally work. The first place is right in the Delphi IDE, as stated earlier. Through RTTI, the IDE magically knows everything about the object and components with which you work (see the Object Inspector). Actually, there's more to it than just RTTI. But for the sake of this discussion, we're covering only the RTTI aspect. The second area is in the runtime code that you write. Already, in Chapter 2 you read about the `is` and `as` operators.

Let's examine the `is` operator to illustrate typical usage of RTTI.

Suppose that you need to make all `TEdit` components read-only on a given form. This is simple enough—just loop through all components, use the `is` operator to determine whether the component is a `TEdit` class, and then set the `ReadOnly` property accordingly. Here's an example:

```
for i := 0 to ComponentCount - 1 do
    if Components[i] is TEdit then
      TEdit(Components[i]).ReadOnly := True;
```

A typical usage for the `as` operator would be to perform an action on the `Sender` parameter of an event handler, where the handler is attached to several different components. Assuming that you know that all components are derived from a common ancestor whose property you want to access, the event handler can use the `as` operator to safely typecast `Sender` as the desired descendant, thus surfacing the wanted property. Here's an example:

```
procedure TForm1.ControlOnClickEvent(Sender: TObject);
var
  i: integer;
begin
 (Sender as TControl).Enabled := False;
end;
```

These examples of *typesafe programming* illustrate enhancements to the Object Pascal language that indirectly use RTTI. Now let's look at a problem that would call for direct usage of RTTI.

Suppose you have a form containing components that are data aware and components that aren't data aware. However, you need to perform some action on the data-aware components only. Certainly you could loop through the `Components` array for the form and test for each data-aware component type. However, this could get messy to maintain because you would have to test against every type of data-aware component. Also, you don't have a base class to test against that's common to only data-aware components. For instance, something such as `TDataAwareControl` would have been nice, but it doesn't exist.

A clean way to determine whether a component is data aware is to test for the existence of a `DataSource` property. You are sure that this property exists for all data-aware components. To do this, however, you need to use RTTI directly.

The following sections discuss RTTI in more depth to give you the background knowledge needed to solve problems such as the one mentioned earlier.

The `TypInfo.pas` Unit: Definer of Runtime Type Information

Type information exists for any object (a descendant of `TObject`). This information exists in memory and is queried by the IDE and the Runtime Library to obtain information about objects. The `TypInfo.pas` unit defines the structures that allow you to query for type information. The `TObject` methods shown in Table 10.6 are repeated from Chapter 2.

TABLE 10.6 `TObject` Methods

Function	*Return Type*	*Returns*
`ClassName()`	`string`	The name of the object's class
`ClassType()`	`TClass`	The object's type
`InheritsFrom()`	`Boolean`	Boolean to indicate whether the class descends from a given class
`ClassParent()`	`TClass`	The object Cancestor's type
`InstanceSize()`	`word`	The size, in bytes, of an instance
`ClassInfo()`	`Pointer`	A pointer to the object's in-memory RTTI

For now, we want to focus on the `ClassInfo()` function, which is defined as follows:

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

This function returns a pointer to the RTTI for the calling class. The structure to which this pointer refers is of the type `PTypeInfo`. This type is defined in the `TypInfo.pas` unit as a pointer to a `TTypeInfo` structure. Both definitions are given in the following code as they appear in `TypInfo.pas`:

```
PPTypeInfo = ^PTypeInfo;
  PTypeInfo = ^TTypeInfo;
  TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
   {TypeData: TTypeData}
  end;
```

The commented field, `TypeData`, represents the actual reference to the type information for the given class. The type to which it actually refers depends on the value of the `Kind` field. `Kind` can be any of the enumerated values defined in the `TTypeKind`:

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

Take a look at the `TypInfo.pas` unit at this time to examine the subtypes to some of the preceding enumerated values to get yourself familiar with them. For example, the `tkFloat` value can be further broken down into one of the following:

```
TFloatType = (ftSingle, ftDouble, ftExtended, ftComp, ftCurr);
```

Now you know that `Kind` determines to which type `TypeData` refers. The `TTypeData` structure is defined in `TypInfo.pas`, as shown in Listing 10.2.

LISTING 10.2 The `TTypeData` Structure

```
PTypeData = ^TTypeData;
TTypeData = packed record
  case TTypeKind of
    tkUnknown, tkLString, tkWString, tkVariant: ();
    tkInteger, tkChar, tkEnumeration, tkSet, tkWChar: (
        OrdType: TOrdType;
        case TTypeKind of
          tkInteger, tkChar, tkEnumeration, tkWChar: (
            MinValue: Longint;
            MaxValue: Longint;
            case TTypeKind of
              tkInteger, tkChar, tkWChar: ();
              tkEnumeration: (
                BaseType: PPTypeInfo;
                NameList: ShortStringBase));
          tkSet: (
            CompType: PPTypeInfo));
    tkFloat: (FloatType: TFloatType);
    tkString: (MaxLength: Byte);
    tkClass: (
        ClassType: TClass;
        ParentInfo: PPTypeInfo;
        PropCount: SmallInt;
        UnitName: ShortStringBase;
       {PropData: TPropData});
    tkMethod: (
      MethodKind: TMethodKind;
      ParamCount: Byte;
```

LISTING 10.2 Continued

```
      ParamList: array[0..1023] of Char
      {ParamList: array[1..ParamCount] of
        record
          Flags: TParamFlags;
          ParamName: ShortString;
          TypeName: ShortString;
        end;
        ResultType: ShortString});
    tkInterface: (
        IntfParent : PPTypeInfo; { ancestor }
        IntfFlags : TIntfFlagsBase;
        Guid : TGUID;
        IntfUnit : ShortStringBase;
       {PropData: TPropData});
    tkInt64: (
        MinInt64Value, MaxInt64Value: Int64);
  end;
```

As you can see, the `TTypeData` structure is really just a big variant record. If you're familiar with working with variant records and pointers, you'll see that dealing with RTTI is really simple. It just seems complex because it's an undocumented feature.

NOTE

Often, Borland doesn't document a feature because it might change between versions. When using features such as the undocumented RTTI, realize that your code might not be fully portable between versions of Delphi.

At this point, we're ready to demonstrate how to use these structures of RTTI to obtain type information.

Obtaining Type Information

To demonstrate how to obtain Runtime Type Information on an object, we've created a project whose main form is defined in Listing 10.3.

LISTING 10.3 Main Form for `ClassInfo.dpr`

```
unit MainFrm;

interface
```

Listing 10.3 Continued

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, DBClient, MidasCon, MConnect;

type

  TMainForm = class(TForm)
    pnlTop: TPanel;
    pnlLeft: TPanel;
    lbBaseClassInfo: TListBox;
    spSplit: TSplitter;
    lblBaseClassInfo: TLabel;
    pnlRight: TPanel;
    lblClassProperties: TLabel;
    lbPropList: TListBox;
    lbSampClasses: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure lbSampClassesClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo;

{$R *.DFM}

function CreateAClass(const AClassName: string): TObject;
{ This method illustrates how you can create a class from the class name. Note
  that this requires that you register the class using RegisterClasses() as
  shown in the initialization method of this unit. }
var
  C : TFormClass;
  SomeObject: TObject;
begin
  C := TFormClass(FindClass(AClassName));
  SomeObject := C.Create(nil);
  Result := SomeObject;
end;
```

LISTING 10.3 Continued

```
procedure GetBaseClassInfo(AClass: TObject; AStrings: TStrings);
{ This method obtains some basic RTTI data from the given object and adds that
  information to the AStrings parameter. }
var
  ClassTypeInfo: PTypeInfo;
  ClassTypeData: PTypeData;
  EnumName: String;
begin
  ClassTypeInfo := AClass.ClassInfo;
  ClassTypeData := GetTypeData(ClassTypeInfo);
  with AStrings do
  begin
    Add(Format('Class Name:     %s', [ClassTypeInfo.Name]));
    EnumName := GetEnumName(TypeInfo(TTypeKind), Integer(ClassTypeInfo.Kind));
    Add(Format('Kind:           %s', [EnumName]));
    Add(Format('Size:           %d', [AClass.InstanceSize]));
    Add(Format('Defined in:     %s.pas', [ClassTypeData.UnitName]));
    Add(Format('Num Properties: %d',[ClassTypeData.PropCount]));
  end;
end;

procedure GetClassAncestry(AClass: TObject; AStrings: TStrings);
{ This method retrieves the ancestry of a given object and adds the
  class names of the ancestry to the AStrings parameter. }
var
  AncestorClass: TClass;
begin
  AncestorClass := AClass.ClassParent;
  { Iterate through the Parent classes starting with Sender's
    Parent until the end of the ancestry is reached. }
  AStrings.Add('Class Ancestry');
  while AncestorClass <> nil do
  begin
    AStrings.Add(Format('    %s',[AncestorClass.ClassName]));
    AncestorClass := AncestorClass.ClassParent;
  end;
end;


procedure GetClassProperties(AClass: TObject; AStrings: TStrings);
{ This method retrieves the property names and types for the given object
  and adds that information to the AStrings parameter. }
var
  PropList: PPropList;
```

LISTING 10.3 Continued

```
  ClassTypeInfo: PTypeInfo;
  ClassTypeData: PTypeData;
  i: integer;
  NumProps: Integer;
begin

  ClassTypeInfo := AClass.ClassInfo;
  ClassTypeData := GetTypeData(ClassTypeInfo);

  if ClassTypeData.PropCount <> 0 then
  begin
    // allocate the memory needed to hold the references to the TPropInfo
    // structures on the number of properties.
    GetMem(PropList, SizeOf(PPropInfo) * ClassTypeData.PropCount);
    try
      // fill PropList with the pointer references to the TPropInfo structures
      GetPropInfos(AClass.ClassInfo, PropList);
      for i := 0 to ClassTypeData.PropCount - 1 do
        // filter out properties that are events ( method pointer properties)
        if not (PropList[i]^.PropType^.Kind = tkMethod) then
          AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
          PropList[i]^.PropType^.Name]));

      // Now get properties that are events (method pointer properties)
      NumProps := GetPropList(AClass.ClassInfo, [tkMethod], PropList);
      if NumProps <> 0 then begin
        AStrings.Add('');
        AStrings.Add('   EVENTS   ================ ');
        AStrings.Add('');
      end;
      // Fill the AStrings with the events.
      for i := 0 to NumProps - 1 do
          AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
          PropList[i]^.PropType^.Name]));

    finally
      FreeMem(PropList, SizeOf(PPropInfo) * ClassTypeData.PropCount);
    end;
  end;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
```

LISTING 10.3 Continued

```
  // Add some example classes to the list box.
  lbSampClasses.Items.Add('TApplication');
  lbSampClasses.Items.Add('TButton');
  lbSampClasses.Items.Add('TForm');
  lbSampClasses.Items.Add('TListBox');
  lbSampClasses.Items.Add('TPaintBox');
  lbSampClasses.Items.Add('TMidasConnection');
  lbSampClasses.Items.Add('TFindDialog');
  lbSampClasses.Items.Add('TOpenDialog');
  lbSampClasses.Items.Add('TTimer');
  lbSampClasses.Items.Add('TComponent');
  lbSampClasses.Items.Add('TGraphicControl');
end;

procedure TMainForm.lbSampClassesClick(Sender: TObject);
var
  SomeComp: TObject;
begin
  lbBaseClassInfo.Items.Clear;
  lbPropList.Items.Clear;

  // Create an instance of the selected class.
  SomeComp := CreateAClass(lbSampClasses.Items[lbSampClasses.ItemIndex]);
  try
    GetBaseClassInfo(SomeComp, lbBaseClassInfo.Items);
    GetClassAncestry(SomeComp, lbBaseClassInfo.Items);
    GetClassProperties(SomeComp, lbPropList.Items);
  finally
    SomeComp.Free;
  end;
end;

initialization
begin
  RegisterClasses([TApplication, TButton, TForm, TListBox, TPaintBox,
    TMidasConnection, TFindDialog, TOpenDialog, TTimer, TComponent,
    TGraphicControl]);
end;

end.
```

NOTE

CLX versions of the RTTI demos shown here reside on the CD-ROM under the subdirectory `CLX` for this chapter.

This main form contains three list boxes. `lbSampClasses` contains classnames for a few sample objects whose type information we'll retrieve. On selecting an object from `lbSampClasses`, `lbBaseClassInfo` will be populated with basic information about the selected object, such as its size and ancestry. `lbPropList` will display the properties belonging to the selected object from `lbSampClasses`.

Three helper procedures are used to obtain class information:

- `GetBaseClassInfo()`—Populates a string list with basic information about an object, such as its type, size, defining unit, and number of properties
- `GetClassAncestry()`—Populates a string list with the object names of a given object's ancestry
- `GetClassProperties()`—Populates a string list with the properties and their types for a given class

Each procedure takes an object instance and a string list as parameters.

As the user selects one of the classes from `lbSampClasses`, its `OnClick` event, `lbSampClassesClick()`, calls a helper function, `CreateAClass()`, which creates an instance of a class given the name of the class type. It then passes the object instance and the appropriate `TListBox.Items` property to be populated.

TIP

The `CreateAClass()` function can be used to create any class by its name. However, as demonstrated, you must make sure that any classes passed to it have been registered by calling the `RegisterClasses()` procedure.

Obtaining Runtime Type Information for Objects

`GetBaseClassInfo()` passes the return value from `TObject.ClassInfo()` to the function `GetTypeData()`. `GetTypeData()` is defined in `TypInfo.pas`. Its purpose is to return a pointer to the `TTypeData` structure based on the class whose `PTypeInfo` structure was passed to it (see Listing 10.2). `GetBaseClassInfo()` simply refers to the various fields of both the `TTypeInfo` and `TTypeData` structures to populate the `AStrings` string list. Note the use of the function

GetEnumName() to return the string for an enumerated type. This is also a function of RTTI defined in TypInfo.pas. Type information on enumerated types is discussed in a later section.

TIP

Use the GetTypeData() function defined in TypInfo.pas to return a pointer to the TTypeInfo structure for a given class. You must pass the result of TObject.ClassInfo() to GetTypeData().

TIP

You can use the GetEnumName() function to obtain the name of an enumeration value as a string. GetEnumValue() returns the enumeration value given its name.

Obtaining the Ancestry for an Object

The GetClassAncestry() procedure populates a string list with the classnames of the given object's ancestry. This is a simple operation that uses the ClassParent() class procedure on the given object. ClassParent() will return a TClass reference to the given class's parent or nil if the top of the ancestry is reached. GetClassAncestry() simply walks up the ancestry and adds each classname to the string list until the top is reached.

Obtaining Type Information on Object Properties

If an object has properties, its TTypeData.PropCount value will contain the number of properties it has. There are several approaches you can use to obtain the property information for a given class—we demonstrate two.

The GetClassProperties() procedure begins much like the previous two methods in that it passes the ClassInfo() result to GetTypeData() to obtain the reference to the TTypeData structure for the class. It then allocates memory for the PropList variable based on the value of ClassTypeData.PropCount. PropList is defined as the type PPropList. PPropList is defined in TypInfo.pas as follows:

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

The TPropList array stores pointers to the TPropInfo data for each property. TPropInfo is defined in TypInfo.pas as follows:

```
  PPropInfo = ^TPropInfo;
  TPropInfo = packed record
```

```
    PropType: PPTypeInfo;
    GetProc: Pointer;
    SetProc: Pointer;
    StoredProc: Pointer;
    Index: Integer;
    Default: Longint;
    NameIndex: SmallInt;
    Name: ShortString;
  end;
```

`TPropInfo` is the Runtime Type Information for a property.

`GetClassProperties()` uses the `GetPropInfos()` function to fill this array with pointers to the RTTI information for all properties for the given object. It then loops through the array and writes out the name and type for the property by accessing that property's type information. Note the following line:

```
if not (PropList[i]^.PropType^.Kind = tkMethod) then
```

This is used to filter out properties that are events (method pointers). We populate these properties last, which allows us to demonstrate an alternative method for retrieving property RTTI. In the final part of the `GetClassProperties()` method, we use the `GetPropList()` function to return the `TPropList` for properties of a specific type. In this case, we want only properties of the type `tkMethod`. `GetPropList()` is also defined in `TypInfo.pas`. Refer to the source commentary for additional information.

TIP

Use `GetPropInfos()` when you want to retrieve a pointer to the property Runtime Type Information for *all* properties of a given object. Use `GetPropList()` if you want to retrieve the same information, except for properties of a specific type.

Figure 10.3 shows the output of the main form with Runtime Type Information for a selected class.

Checking for the Existence of a Property for an Object

Earlier we presented the problem of needing to check for the existence of a property for a given object. Specifically, we were referring to the `DataSource` property. Using functions defined in `TypInfo.pas`, we could write the following function to determine whether a control is data aware:

```
function IsDataAware(AComponent: TComponent): Boolean;
var
  PropInfo: PPropInfo;
```

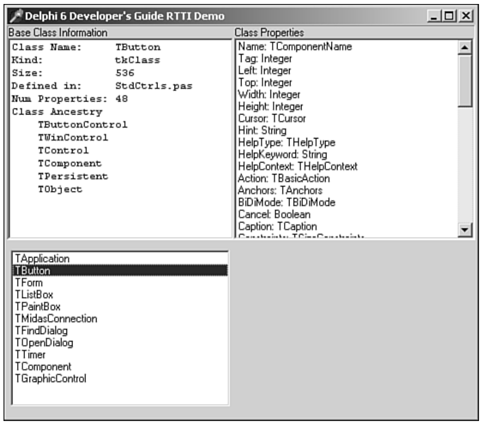
```
begin
  // Find the property named datasource.
  PropInfo := GetPropInfo(AComponent.ClassInfo, 'DataSource');
  Result := PropInfo <> nil;

  // Double check, make sure it descends from TDataSource
  if Result then
    if not ((PropInfo^.Proptype^.Kind = tkClass) and
        (GetTypeData(PropInfo^.PropType^).ClassType.InheritsFrom(TDataSource)))
       then
      Result := False;
end;
```

Here, we're using the `GetPropInfo()` function to return the `TPropInfo` pointer on a given property. This function returns `nil` if the property doesn't exist. As an additional check, we make sure that the property named `DataSource` is actually a descendant of `TDataSource`.

We also could have written this function more generically to check for the existence of any property by its name, like this:

```
function HasProperty(AComponent: TComponent; APropertyName: String): Boolean;
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComponent.ClassInfo, APropertyName);
  Result := PropInfo <> nil;
end;
```

Note, however, that this works only on published properties. RTTI doesn't exist for unpublished properties.



```
Class Name:     TButton
Kind:           tkClass
Size:           536
Defined in:     StdCtrls.pas
Num Properties: 48
Class Ancestry
    TButtonControl
    TWinControl
    TControl
    TComponent
    TPersistent
    TObject
```

```
Name: TComponentName
Tag: Integer
Left: Integer
Top: Integer
Width: Integer
Height: Integer
Cursor: TCursor
Hint: String
HelpType: THelpType
HelpKeyword: String
HelpContext: THelpContext
Action: TBasicAction
Anchors: TAnchors
BiDiMode: TBiDiMode
Cancel: Boolean
Caption: TCaption
```

```
TApplication
TButton
TForm
TListBox
TPaintBox
TMidasConnection
TFindDialog
TOpenDialog
TTimer
TComponent
TGraphicControl
```

FIGURE 10.3
Output of a class's Runtime Type Information.

Obtaining Type Information on Method Pointers

Runtime Type Information can be obtained on method pointers. For example, you can determine the type of method (procedure, function, and so on) and its parameters. Listing 10.4 demonstrates how to obtain Runtime Type Information for a selected group of methods.

LISTING 10.4 Obtaining Runtime Type Information for Methods

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, DBClient, MidasCon, MConnect;

type

  TMainForm = class(TForm)
    lbSampMethods: TListBox;
    lbMethodInfo: TMemo;
    lblBasicMethodInfo: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure lbSampMethodsClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo, DBTables, Provider;

{$R *.DFM}

type
  // It is necessary to redefine this record as it is commented out in
  // typinfo.pas.

  PParamRecord = ^TParamRecord;
  TParamRecord = record
    Flags:     TParamFlags;
    ParamName: ShortString;
```

LISTING 10.4 Continued

```
    TypeName:  ShortString;
  end;

procedure GetBaseMethodInfo(ATypeInfo: PTypeInfo; AStrings: TStrings);
{ This method obtains some basic RTTI data from the TTypeInfo and adds that
  information to the AStrings parameter. }
var
  MethodTypeData: PTypeData;
  EnumName: String;
begin
  MethodTypeData := GetTypeData(ATypeInfo);
  with AStrings do
  begin
    Add(Format('Class Name:     %s', [ATypeInfo^.Name]));
    EnumName := GetEnumName(TypeInfo(TTypeKind), Integer(ATypeInfo^.Kind));
    Add(Format('Kind:           %s', [EnumName]));
    Add(Format('Num Parameters: %d',[MethodTypeData.ParamCount]));
  end;
end;

procedure GetMethodDefinition(ATypeInfo: PTypeInfo; AStrings: TStrings);
{ This method retrieves the property info on a method pointer. We use this
  information to reconstruct the method definition. }
var
  MethodTypeData: PTypeData;
  MethodDefine:   String;
  ParamRecord:    PParamRecord;
  TypeStr:        ^ShortString;
  ReturnStr:      ^ShortString;
  i: integer;
begin
  MethodTypeData := GetTypeData(ATypeInfo);

  // Determine the type of method
  case MethodTypeData.MethodKind of
    mkProcedure:      MethodDefine := 'procedure ';
    mkFunction:       MethodDefine := 'function ';
    mkConstructor:    MethodDefine := 'constructor ';
    mkDestructor:     MethodDefine := 'destructor ';
    mkClassProcedure: MethodDefine := 'class procedure ';
    mkClassFunction:  MethodDefine := 'class function ';
  end;
```

LISTING 10.4 Continued

```
  // point to the first parameter
  ParamRecord    := @MethodTypeData.ParamList;
  i := 1; // first parameter

  // loop through the method's parameters and add them to the string list as
  // they would be normally defined.
  while i <= MethodTypeData.ParamCount do
  begin
    if i = 1 then
      MethodDefine := MethodDefine+'(';

    if pfVar in ParamRecord.Flags then
      MethodDefine := MethodDefine+('var ');
    if pfconst in ParamRecord.Flags then
      MethodDefine := MethodDefine+('const ');
    if pfArray in ParamRecord.Flags then
      MethodDefine := MethodDefine+('array of ');
//  we won't do anything for the pfAddress but know that the Self parameter
//  gets passed with this flag set.
{
    if pfAddress in ParamRecord.Flags then
      MethodDefine := MethodDefine+('*address* ');
}
    if pfout in ParamRecord.Flags then
      MethodDefine := MethodDefine+('out ');


    // Use pointer arithmetic to get the type string for the parameter.
    TypeStr := Pointer(Integer(@ParamRecord^.ParamName) +
      Length(ParamRecord^.ParamName)+1);

    MethodDefine := Format('%s%s: %s', [MethodDefine, ParamRecord^.ParamName,
      TypeStr^]);

    inc(i); // Increment the counter.

    // Go the next parameter. Notice that use of pointer arithmetic to
    // get to the appropriate location of the next parameter.
    ParamRecord := PParamRecord(Integer(ParamRecord) + SizeOf(TParamFlags) +
      (Length(ParamRecord^.ParamName) + 1) + (Length(TypeStr^)+1));

    // if there are still parameters then setup
    if i <= MethodTypeData.ParamCount then
```

LISTING 10.4 Continued

```
    begin
      MethodDefine := MethodDefine + '; ';
    end
    else
      MethodDefine := MethodDefine + ')';
  end;

  // If the method type is a function, it has a return value. This is also
  // placed in the method definition string. The return value will be at the
  // location following the last parameter.
  if MethodTypeData.MethodKind = mkFunction then
  begin
    ReturnStr := Pointer(ParamRecord);
    MethodDefine := Format('%s: %s;', [MethodDefine, ReturnStr^])
  end
  else
    MethodDefine := MethodDefine+';';

  // finally, add the string to the listbox.
  with AStrings do
  begin
    Add(MethodDefine)
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Add some method types to the list box. Also, store the pointer to the RTTI
    data in listbox's Objects array }
  with lbSampMethods.Items do
  begin
    AddObject('TNotifyEvent', TypeInfo(TNotifyEvent));
    AddObject('TMouseEvent', TypeInfo(TMouseEvent));
    AddObject('TBDECallBackEvent', TypeInfo(TBDECallBackEvent));
    AddObject('TDataRequestEvent', TypeInfo(TDataRequestEvent));
    AddObject('TGetModuleProc', TypeInfo(TGetModuleProc));
    AddObject('TReaderError', TypeInfo(TReaderError));
  end;
end;

procedure TMainForm.lbSampMethodsClick(Sender: TObject);
begin
  lbMethodInfo.Lines.Clear;
  with lbSampMethods do
```

LISTING 10.4 Continued

```
  begin
    GetBaseMethodInfo(PTypeInfo(Items.Objects[ItemIndex]), lbMethodInfo.Lines);
    GetMethodDefinition(PTypeInfo(Items.Objects[ItemIndex]),
      lbMethodInfo.Lines);
  end;
end;

end.
```

In Listing 10.4, we populate a list box, `lbSampMethods`, with some sample method names. We also store the references to those methods' RTTI data in the `Objects` array of the list box. We do this by using the `TypeInfo()` function, which is a special function that can retrieve a pointer to Runtime Type Information for a given type identifier. When the user selects one of these methods, we use that RTTI data from the `Objects` array to retrieve and reconstruct the method definition from the information we have about the method and its parameters in the RTTI data. Refer to the listing's commentary for further information.

TIP

Use the `TypeInfo()` function to retrieve a pointer to the compiler-generated Runtime Type Information for a given type identifier. For example, the following line retrieves a pointer to the RTTI for the `TButton` type:

```
  TypeInfoPointer := TypeInfo(TButton);
```

Obtaining Type Information for Ordinal Types

We've already covered the more difficult pieces to RTTI. However, you can also obtain RTTI for ordinal types. The following sections illustrate how to obtain RTTI data on integer, enumerated, and set types.

Type Information for Integer Types

Obtaining type information for integer types is simple. Listing 10.5 illustrates this process.

LISTING 10.5 Obtaining Runtime Type Information for Integers

```
procedure TMainForm.lbSampsClick(Sender: TObject);
var
  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;
```

LISTING 10.5 Continued

```
  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
begin
  memInfo.Lines.Clear;
  with lbSamps do
  begin

    // Get the TTypeInfo pointer
    OrdTypeInfo := PTypeInfo(Items.Objects[ItemIndex]);
    // Get the TTypeData pointer
    OrdTypeData := GetTypeData(OrdTypeInfo);

    // Get the type name string
    TypeNameStr := OrdTypeInfo.Name;
    // Get the type kind string
    TypeKindStr := GetEnumName(TypeInfo(TTypeKind),
Integer(OrdTypeInfo^.Kind));

    // Get the minimum and maximum values for the type
    MinVal := OrdTypeData^.MinValue;
    MaxVal := OrdTypeData^.MaxValue;


    // Add the information to the memo
    with memInfo.Lines do
    begin
      Add('Type Name: '+TypeNameStr);
      Add('Type Kind: '+TypeKindStr);

      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));
    end;
  end;
end;
```

Here, we use the `TypeInfo()` function to obtain a pointer to the `TTypeInfo` structure for the `Integer` data type. We then pass that reference to the `GetTypeData()` function to obtain a pointer to the `TTypeData` structure. We use both those structures to populate a list box with the integer's RTTI. See the demo named `IntegerRTTI.dpr` in the directory for this chapter on the CD-ROM accompanying this book for a more detailed demonstration.

Type Information for Enumerated Types

Obtaining RTTI for enumerated types is just as easy as it is for integers. In fact, you'll see that Listing 10.6 is almost identical to Listing 10.5, with the exception of the additional `for` loop to show the values of the enumeration type.

LISTING 10.6 Obtaining RTTI for an Enumerated Type

```
procedure TMainForm.lbSampsClick(Sender: TObject);
var
  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;

  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
  i: integer;
begin
  memInfo.Lines.Clear;
  with lbSamps do
  begin

    // Get the TTypeInfo pointer
    OrdTypeInfo := PTypeInfo(Items.Objects[ItemIndex]);
    // Get the TTypeData pointer
    OrdTypeData := GetTypeData(OrdTypeInfo);

    // Get the type name string
    TypeNameStr := OrdTypeInfo.Name;
    // Get the type kind string
    TypeKindStr := GetEnumName(TypeInfo(TTypeKind),
Integer(OrdTypeInfo^.Kind));

    // Get the minimum and maximum values for the type
    MinVal := OrdTypeData^.MinValue;
    MaxVal := OrdTypeData^.MaxValue;


    // Add the information to the memo
    with memInfo.Lines do
    begin
      Add('Type Name: '+TypeNameStr);
      Add('Type Kind: '+TypeKindStr);

      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));
```

LISTING 10.6 Continued

```
      // Show the values and names of the enumerated types
      if OrdTypeInfo^.Kind = tkEnumeration then
        for i := MinVal to MaxVal do
          Add(Format('  Value: %d   Name: %s', [i,
            GetEnumName(OrdTypeInfo, i)]));

    end;
  end;
end;
```

You'll find a more detailed demo named `EnumRTTI.dpr` on the CD-ROM in the directory for this chapter.

Type Information for Set Types

Obtaining RTTI for set types is only slightly more complex than the two previous techniques. Listing 10.7 is the main form for the project `SetRTTI.dpr`, which you'll find on the CD-ROM in the directory for this chapter.

LISTING 10.7 Obtaining RTTI for Set Types

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Grids;

type
  TMainForm = class(TForm)
    lbSamps: TListBox;
    memInfo: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure lbSampsClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;
```

LISTING 10.7 Continued

```
implementation
uses TypInfo, Buttons;

{$R *.DFM}


procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Add some example enumerated types
  with lbSamps.Items do
  begin
    AddObject('TBorderIcons', TypeInfo(TBorderIcons));
    AddObject('TGridOptions', TypeInfo(TGridOptions));
  end;
end;

procedure GetTypeInfoForOrdinal(AOrdTypeInfo: PTypeInfo; AStrings: TStrings);
var
//  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;

  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
  i: integer;
begin

  // Get the TTypeData pointer
  OrdTypeData := GetTypeData(AOrdTypeInfo);

  // Get the type name string
  TypeNameStr := AOrdTypeInfo.Name;
  // Get the type kind string
  TypeKindStr := GetEnumName(TypeInfo(TTypeKind), Integer(AOrdTypeInfo^.Kind));

  // Get the minimum and maximum values for the type
  MinVal := OrdTypeData^.MinValue;
  MaxVal := OrdTypeData^.MaxValue;


  // Add the information to the memo
  with AStrings do
  begin
    Add('Type Name: '+TypeNameStr);
    Add('Type Kind: '+TypeKindStr);
```

LISTING 10.7 Continued

```
    // Call this function recursively to show the enumeration
    // values for this set type.
    if AOrdTypeInfo^.Kind = tkSet then
    begin
      Add('==========');
      Add('');
      GetTypeInfoForOrdinal(OrdTypeData^.CompType^, AStrings);
    end;

    // Show the values and names of the enumerated types belonging to the
    // set.
    if AOrdTypeInfo^.Kind = tkEnumeration then
    begin
      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));

      for i := MinVal to MaxVal do
        Add(Format('  Value: %d   Name: %s', [i,
         GetEnumName(AOrdTypeInfo, i)]));
    end;
  end;

end;

procedure TMainForm.lbSampsClick(Sender: TObject);
begin
  memInfo.Lines.Clear;
  with lbSamps do
    GetTypeInfoForOrdinal(PTypeInfo(Items.Objects[ItemIndex]), memInfo.Lines);
end;
end.
```

In this demo, we set up two set types in a list box. We add the pointer to the `TTypeInfo` structures for these two types to the `Objects` array of the list box by using the `TypeInfo()` function. When the user selects one of the items in the list box, the `GetTypeInfoForOrdinal()` procedure is called, passing both the `PTypeInfo` pointer and the `memInfo.Lines` property that's populated with the RTTI data.

The `GetTypeInfoForOrdinal()` procedure goes through the same steps you've already seen for getting the pointer to the type's `TTypeData` structure. This initial type information is stored to the `TStrings` parameter and then the `GetTypeInfoForOrdinal()` is called recursively, passing `OrdTypeData^.CompType^`, which refers to the enumerated data type for the set. This RTTI data is also added to the same `TStrings` property.

Assigning Values to Properties Through RTTI

Now that we've shown you how to find and determine which published properties exist for components, we ought to show you how to assign values to properties through RTTI. This task is simple. The `TypInfo.pas` unit contains many helper routines to allow you to interrogate and manipulate component-published properties. These are the same helper routines used by the Delphi IDE (Object Inspector). It would be a good idea to open `TypInfo.pas` and to familiarize yourself with these routines. We'll demonstrate a few of them here.

Suppose that you want to assign an integer value to a property for a given component. Also suppose that you don't know whether this property exists on that component. Here's a procedure that assigns an integer value to a property for a given component, only if that property exists:

```
procedure SetIntegerPropertyIfExists(AComp: TComponent; APropName: String;
  AValue: Integer);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComp.ClassInfo, APropName);
  if PropInfo <> nil then
  begin
    if PropInfo^.PropType^.Kind = tkInteger then
      SetOrdProp(AComp, PropInfo, Integer(AValue));
  end;
end;
```

This procedure takes three parameters. The first, `AComp`, is the component whose property you want to modify. The second parameter, `APropName`, is the name of the property to which you want to assign the value of the third parameter, `AValue`. This procedure uses the `GetPropInfo()` function to retrieve the `TPropInfo` pointer on the specified property. `GetPropInfo()` will return `nil` if the property doesn't exist. If the property does exist, the second `if` clause determines whether the property is of the correct type. The property type `tkInteger` is defined in the `TypInfo.pas` unit along with other possible property types, as shown here:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
    tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, tkWString,
    tkVariant, tkArray, tkRecord, tkInterface, tkInt64, tkDynArray);
```

Finally, the assignment is made to the property using the `SetOrdProp()` procedure, another helper routine from `TypInfo.pas` used to set values to ordinal-type properties. The call to this procedure might look something like the following:

```
SetIntegerPropertyIfExists(Button2, 'Width', 50);
```

`SetOrdProp()` is referred to as a *setter* method, a method used to set a value to a property. There is also a *getter* method, which retrieves the property value. Several of these

SetXXXProp() helper routines are in the TypInfo.pas unit for the possible property types, as shown in Table 10.7.

TABLE 10.7 Getter and Setter Methods

Property Type	*Setter Method*	*Getter Method*
Ordinal	SetOrdProp()	GetOrdProp()
Enumerated	SetEnumProp()	GetEnumProp()
Objects	SetObjectProp()	GetObjectProp()
String	SetStrProp()	GetStrProp()
Floating Point	SetFloatProp()	GetFloatProp()
Variant	SetVariantProp()	GetVariantProp()
Methods (Events)	SetMethodProp()	GetMethodProp()
Int64	SetInt64Prop()	GetInt64Prop()

Again, there are many other helper routines you'll find useful in TypInfo.pas.

The following code shows how to assign an object property:

```
procedure SetObjectPropertyIfExists(AComponent: TComponent; APropName: String;
  AValue: TObject);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComponent.ClassInfo, APropName);
  if PropInfo <> nil then
  begin
    if PropInfo^.PropType^.Kind = tkClass then
      SetObjectProp(AComponent, PropInfo, AValue);
  end;
end;
```

This method might be called as follows:

```
var
  F: TFont;
begin
  F := TFont.Create;
  F.Name   := 'Arial';
  F.Size   := 24;
  F.Color  := clRed;
  SetObjectPropertyIfExists(Panel1, 'Font', F);
end;
```

The following code shows how to assign a method property:

```
procedure SetMethodPropertyIfExists(AComp: TComponent; APropName: String;
  AMethod: TMethod);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComp.ClassInfo, APropName);
  if PropInfo <> nil then
  begin
    if PropInfo^.PropType^.Kind = tkMethod then
      SetMethodProp(AComp, PropInfo, AMethod);
  end;
end;
```

This method requires the use of the `TMethod` type, which is defined in the `System.pas` unit. To call this method to assign an event handler from one component to another, you can use `GetMethodProp` to retrieve the `TMethod` value from the source component, as shown here:

```
SetMethodPropertyIfExists(Button5, 'OnClick',
    GetMethodProp(Panel1, 'OnClick'));
```

The accompanying CD-ROM has a project, `SetProperties.dpr`, that demonstrates these routines.

Summary

This chapter introduced you to the Visual Component Library (VCL) and Component Library for Cross Platform (CLX). We discussed the hierarchies and the special characteristics of components at different levels in each hierarchy. We also covered Runtime Type Information in depth. This chapter prepared you for the following chapters, which cover component writing.