s

Introduction to component writing

See also

Borland C++Builder is not just a development environment for visually assembling applications from existing components. It also includes everything necessary to create custom components. These components can be based on existing components or be entirely new. You create components within code and don't use the visual tools of the development environment.

If you have already developed components with Delphi, you can add them to the C++Builder Component palette. You can write C++Builder components in either C++ or Object Pascal.

The *Component Writer's Guide* and its accompanying Help file (BCBCWG.HLP) describe everything you need to know to write components for C++Builder applications in C++.

This material has two purposes:

- To teach you how to create working components
- To ensure that the components you write are well-behaved parts of the C++Builder environment

Whether you're writing components for your own applications or for commercial distribution, this book will guide you to writing components that fit in well with any C++Builder application.

What is a component?

See also

Components are the building blocks of C++Builder applications. Although most components represent visible parts of a user interface, components can also represent nonvisual elements in a program, such as timers and databases.

There are three different levels at which to think about components:

The functional definition of component

The technical definition of component

The component writer's definition of component

The functional definition of component

See also

From the component user's perspective, a component is something to choose from the Component palette and use in an application by manipulating it in the Forms Designer or in code. From the component writer's perspective, however, a component is an object in code. Although there are few real restrictions on what you can do when writing a component, it's good to keep in mind what the end user expects when using the components you write.

Before you attempt to write components, we strongly recommend that you become familiar with the existing components in C++Builder so you can make your components familiar to users. Your goal should be to make your components "feel" as much like other components as possible.

The technical definition of component

See also

At the simplest level, a component is any class descended from the class *TComponent*. *TComponent* defines the most basic behavior that all components must have, such as the ability to appear on the Component palette and operate in the Forms Designer.

But beyond that simple definition are several larger issues. For example, although *TComponent* defines the basic behavior needed to operate in the C++Builder environment, it can't know how to handle all the specific additions you make to your components. You'll have to specify those yourself.

Although it's not difficult to create well-behaved components, it does require that you pay close attention to the standards and conventions spelled out in this book.

The component writer's definition of component

See also

At a very practical level, a component is any element that can "plug into" the C++Builder development environment. It can represent almost any level of complexity, from a simple addition to one of the standard components, to a vast, complex interface to another hardware or software system. In short, a component can do or be anything you can create in code, as long as it fits into the component framework.

A component, then, is essentially an interface specification. This manual spells out the framework onto which you build your specialized code to make it work in C++Builder.

Defining the limits of "component" is therefore like defining the limits of programming. We can't tell you every kind of component you can create, any more than we can tell you all the programs you can write in a given language. What we can do is tell you how to write your code so that it fits well in the C++Builder environment.

What's different about writing components?

See also

There are three important differences between the task of creating a component for use in C++Builder and the more common task of creating an application that uses components:

- Component writing is nonvisual
- Component writing requires deeper knowledge of classes
- Component writing follows more conventions

Component writing is nonvisual

See also

The most obvious difference between writing components and building applications with C++Builder is that component writing is done strictly in code. Because the visual design of C++Builder applications requires completed components, creating those components requires writing C++ or Object Pascal code.

Although you can't use the same visual tools for creating components, you can use all the programming features of the C++Builder development environment, including the Code editor, and integrated debugger.

Component writing requires deeper knowledge of classes

See also

Other than the nonvisual programming, the biggest difference between creating components and using them is that when you create a new component, you derive a new class from an existing one, adding new properties and methods. Component users, on the other hand, use existing components and customize their behavior at design time by changing properties and specifying responses to events.

When deriving new classes, you have access to parts of the base classes unavailable to component users of those same classes. These parts, intended only for component writers, are collectively called the *protected interface* to the classes. Derived classes also need to call on their base classes for a lot of their implementation, so component writers need to be familiar with that aspect of object-oriented programming.

Component writing follows more conventions

See also

Writing a component is a more traditional programming task than visual application creation, and there are more conventions you need to follow than when you use existing components. Before you start writing components of your own it is important to really use the components that come with C++Builder. You'll become familiar with such things as naming conventions, and you'll also understand what type of capabilities component users will come to expect when they use your components.

Component users expect that they can do almost anything to your components at any time. Writing components that fulfill that expectation is not difficult, but it requires some forethought and adherence to conventions.

Creating a component

See also

Briefly, the process of creating your own component consists of these steps:

1 Create a unit (a .CPP file and header combination) for the new component.

2 Derive a component type from an existing component type.

3 Add properties, methods, and events as needed.

4 Register your component with C++Builder.

5 Create a Help file for your component and its properties, methods, and events.

All these steps are covered in detail in this Help file. When you finish, the complete component includes these files:

- An .OBJ file, which is created automatically if you provide a .CPP (C++ source) or .PAS (Object Pascal source) file.
- A header file
- If your source code is C++ or C, the header is an .H file.
- If your source code is Object Pascal, the compiler generates an .HPP interface file.
- A palette bitmap (.RES file or .DCR file)
- If the component uses a form, a .DFM file

Overview of component creation

See also

This set of topics provides a broad overview of component architecture, the philosophy of component design, and the process of writing components for C++Builder applications.

The main topics discussed are

- The Visual Component Library
- Components and classes
- How do you create components?
- What goes in a component?
- Creating a new component
- Testing uninstalled components
- Installing a component on the Component palette

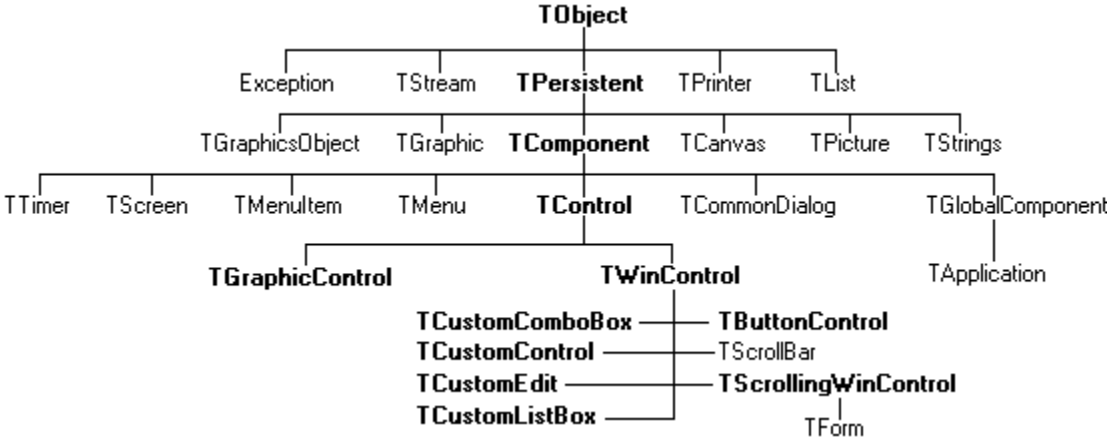
All this material assumes you have some familiarity with using C++Builder and its standard components.

The Visual Component Library

See also

C++Builder's components are all part of a class hierarchy called the Visual Component Library (VCL). The following figure shows the relationship of the classes that make up VCL.

Note that the class *TComponent* is the shared ancestor of every component in the VCL. *TComponent* provides the minimal properties and events necessary for a component to work in C++Builder. The various branches of the library provide other, more specialized capabilities.



When you create a component, you add to the VCL by deriving a new class from one of the existing class types in the hierarchy.

Components and classes

See also

Because components are classes, component writers work with classes at a different level than component users do. Creating new components requires that you derive new classes. OOP for component writers describes in detail the kinds of object-oriented tasks component writers need to use.

Briefly, there are two main differences between creating components and using components. When creating components,

- You have access to parts of the class that are inaccessible to end users.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions, and you need to think in terms of how users will use the components you write.

How do you create components?

See also

A component can be almost any program element you want to manipulate at design time. Creating a new component means deriving a new component class from an existing class. You can derive a new component from any existing component, but the following are the most common ways to create new components:

- Modifying existing controls
- Creating original controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

The following table summarizes the different kinds of components and the classes you use as starting points for each.

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TListBox*, or an abstract component type, such as *TCustomListBox*.
Create an original control	*TWinControl*
Create a graphic control	*TGraphicControl*
Create a nonvisual component	*TComponent*

You can also derive other classes that are not components, but you cannot manipulate them in a form. C++Builder includes many of these classes, such as *TINIFile* or *TFont*.

Modifying existing controls

See also

The simplest way to create a component is to start from an existing, working component and customize it. You can derive a new component from any of the components provided with C++Builder. For instance, you might want to change the default property values of one of the standard controls.

There are certain controls, such as list boxes and grids, that have a number of variations on a basic theme. In those cases, C++Builder provides an abstract control class (with the word "custom" in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special kind of list box that does not have some of the properties of the standard *TListBox* class. You can't remove a property from an ancestor class, so you need to derive your component from something higher in the hierarchy than *TListBox*. Rather than forcing you to go clear back to an abstract control class and reinvent all the list box functions, the Visual Component Library (VCL) provides *TCustomListBox*, which implements all the properties needed for a list box, but does not publish all of them.

When deriving a component from one of the abstract classes such as *TCustomListBox*, you publish those properties you want to make available in your component and leave the rest protected.

Creating original controls

See also

A windowed control is an item that's visible at runtime, usually one the user can interact with. These windowed controls all descend from the class *TWinControl*. The key aspect of a standard control is that it has a window handle, embodied in a property called *Handle*. The window handle means that Windows "knows about" the control, so that

- The control can receive the input focus.
- You can pass the handle to Windows API functions. (Windows needs a handle to identify which window to operate on.)

While you could create an original control (one that's not related to any existing control) using *TWinControl* as your starting point, C++Builder provides the *TCustomControl* component for just this purpose. A *TCustomControl* component is a specialized windowed control that makes it easier to draw complex visual images.

If your control doesn't need to receive input focus, you can make it a graphic control, which saves system resources.

All the components that represent standard windows controls, such as push buttons, list boxes, and edit boxes, descend from *TWinControl* except *TLabel*, as label controls never receive the input focus.

Creating graphic controls

See also

Graphic controls are very similar to custom controls, but they don't carry the overhead of being Windows controls. That is, Windows doesn't know about graphic controls. They have no window handles, and therefore consume no system resources. The main restriction on graphic controls is that they can't receive the input focus.

C++Builder supports the creation of custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract component derived from *TControl*. Although you can derive controls from *TControl*, you should derive them from *TGraphicControl*, because *TGraphicControl* provides a canvas to paint on and handles *WM_PAINT* messages. All you need to do is override the *Paint* method.

Subclassing Windows controls

See also

Windows has a concept called a *window class* that is somewhat similar to the object-oriented programming concept of object or class. A window class is a set of information shared between different instances of the same sort of window or control in Windows.

When you create a new kind of control (usually called a *custom control*) in traditional Windows programming, you define a new window class and register it with Windows. You can also base a new window class on an existing class, which is called *subclassing*.

In traditional Windows programming, if you wanted to create a custom control, you had to write it in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

Using C++Builder, you can create a component "wrapper" around any existing Windows class. So if you already have a library of custom controls that you want to use in your C++Builder applications, you can create C++Builder components that let you use your existing controls and derive new controls from them just as you would any other component.

Although this text does not include an example of subclassing a Windows control, you can see the techniques used in the components in the *StdCtls* header file that represent the standard Windows controls, such as *TEdit*.

Creating nonvisual components

See also

The abstract *TComponent* class is the base class for all components. The only components you create directly from *TComponent* are nonvisual components. Most of the components you write will probably be various kinds of visual controls.

TComponent defines all the properties and methods essential for a component to participate in the Form Designer. Any component you derive from *TComponent*, therefore, already has design capability built into it.

Nonvisual components are fairly rare. You mostly use them as an interface for nonvisual program elements (much as C++Builder uses them for database elements) and as place holders for dialog boxes (such as the file dialog boxes).

What goes in a component?

See also

There are few restrictions on what you can put in the components you write. There are certain conventions you should follow, however, if you want to make your components easy and reliable for the people who use them.

This section discusses the philosophies underlying the design of components, including the following topics:

- Removing dependencies
- Properties, events, and methods
- Graphics encapsulation
- Registration

Removing dependencies

See also Example

Perhaps the most important philosophy behind the creation of C++Builder's components is the necessity of removing dependencies. One of the things that makes components so easy for end users to incorporate into their applications is the fact that there are generally no restrictions on what they can do at any given point in their code.

The very nature of components suggests that different users will incorporate them into applications in varying combinations, orders, and environments. You should design your components so that they function in any context, without requiring any preconditions.

An example of removing dependencies

An excellent example of removing dependencies in components is the *Handle* property of windowed controls. If you've written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you don't access a window or control until you've created it by calling the *CreateWindow* API function. Calling API functions with invalid handles causes a multitude of problems.

C++Builder components protect users from worrying about window handles and whether they are valid by ensuring that a valid handle is always available when needed. That is, by using a property for the window handle, the component can check whether the window has been created, and therefore whether there is a valid window handle. If the handle isn't already valid, the property creates the window and returns the handle. Thus, any time a user's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing the background tasks such as creating the window, components allow developers to focus on what they really want to do. If a developer needs to pass a window handle to an API function, it shouldn't be necessary to first check to make sure there's a valid handle and, if necessary, create the window. With component-based programming, the programmer can write assuming that things will work, instead of constantly checking for things that might go wrong.

Although it might take a little more time to create components that don't have dependencies, it's generally time well spent. Not only does it keep users of your components from having to repeatedly perform the same tasks, but it also reduces your documentation and support burdens, since you don't have to provide and explain numerous warnings or resolve the problems users might have with your components.

Properties, events, and methods

See also

Outside of the visible image the component user manipulates in the form at design time, the most obvious attributes of a component are its properties, events, and methods. Each of these is sufficiently important that it has its own section in this file, but this section explains a little of the philosophy of implementing them.

Properties

Properties give the component user the illusion of setting or reading the value of a variable in the component while allowing the component writer to hide the underlying data structure or to implement side effects of accessing the value.

There are several advantages to the component writer in using properties:

- Properties are available at design time.
  This allows the component user to set and change initial values of properties without having to write code.
- Properties can check values or formats as the user assigns them.
  Validating user input prevents errors caused by invalid values.
- The component can construct appropriate values on demand.
  Perhaps the most common type of error programmers make is to reference a variable that hasn't had an initial value assigned. By making the value a property, you can ensure that the value read from the property is always valid.

Creating properties explains how to add properties to your components.

Events

Events are connections between occurrences determined by the component writer (such as mouse actions and keystrokes) and code written by component users (event handlers). In essence, an event is the component writer's way of providing a hook for the component user to specify what code to execute when a particular occurrence happens.

It is events, therefore, that allow component users to *be* component users instead of component writers. The most common reason for subclassing in traditional Windows applications is that users want to specify a different response to, for example, a Windows message. But in C++Builder, component users can specify handlers for predefined events without subclassing, so they don't need to derive their own components.

Creating events explains how to add events for standard Windows occurrences or events you define yourself.

Methods

Methods are functions built into a component. Component users use methods to direct a component to perform a specific action or return a certain value not covered by a property. Methods are also useful for updating several related properties with a single call.

Because they require execution of code, methods are only available at runtime.

Creating methods explains how to add methods to your components.

Graphics encapsulation

See also

C++Builder takes most of the drudgery out of Windows graphics by encapsulating the various graphic tools into a canvas. The canvas represents the drawing surface of a window or control, and contains other classes, such as a pen, a brush, and a font. A canvas is much like a Windows device context, but it takes care of all the bookkeeping for you.

If you've ever written a graphic Windows application, you're familiar with the kinds of requirements Windows' graphics device interface (GDI) imposes on you, such as limits on the number of device contexts available, and restoring graphic objects to their initial state before destroying them.

When working with graphics in C++Builder, you don't have to worry about any of those things. To draw on a form or component, you access the *Canvas* property. If you want to customize a pen or brush, you set the color or style. When you finish, C++Builder takes care of disposing of the resources. In fact, it caches resources, so if your application frequently uses the same kinds of resources, the caching will probably prevent a lot of creating and recreating.

Of course, you still have full access to the Windows GDI, but you'll often find that your code is much simpler and runs faster if you use the canvas built into C++Builder components. Graphics features are detailed in Using graphics in components.

Registration

See also

Before your components can operate in C++Builder at design time, you have to register them with C++Builder. Registration tells C++Builder where you want your component to appear on the Component palette. There are also some customizations you can make to the way C++Builder stores your components in the form file. Registration is explained in Registering components.

Creating a new component

See also

There are several steps you perform whenever you create a new component. All the examples given that create new components assume you know how to perform these steps.

You can create a new component two ways:

- Using the Component wizard
- Creating a component manually

Once you do either of those, you have at least a minimally functional component ready to install on the Component palette.After installing, you can add your new component to a form and test it in both design time and runtime. You can then add more features to the component, update the Component palette, and continue testing.

Using the Component wizard

See also

You can use the Component wizard to create a new component. Using the Component wizard simplifies the initial stages of creating a new component, as you must specify only these things:

- The class name for the new component
- The class from which it is derived
- The Component palette page you want it to appear on

The Component wizard performs the same tasks you would do when creating a component manually, namely

- Creating a unit (a .CPP file and its associated header)
- Deriving the component
- Declaring a new constructor
- Registering the component

The Component wizard can't add new components to an existing unit (consisting of a .CPP file and an associated header file). If you want to add new components, you must add them to the unit manually.

To open the Component wizard, choose one of these two methods:

- Choose Component|New.
- Choose File|New, select the New page, and select Component.

After you fill in the fields in the Component wizard, choose OK. C++Builder creates a new unit consisting of a .CPP file and an associated header file.

The .CPP file appears in the Code editor. It contains a constructor for the component and the *Register* function that registers the component, informing C++Builder which component to add to the component library and on which page of the Component palette it should appear. The file also contains an include statement that specifies the header file that was created. For example,

```
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
__fastcall TNewComponent::TNewComponent(TComponent* AOwner): TComponent(AOwner)
{
}
namespace Unit1
{
  void __fastcall Register()
  {
    TComponentClass classes[1] = {__classid(TNewComponent)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

To open the header file in the Code editor, place your cursor on the header file name and click your right mouse button to display the context menu. Choose Open File at Cursor on the menu.

The header file contains the new class declaration, including a constructor declaration, and several include statements to support the new class. For example,

```
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
//---------------------------------------------------------------------------
class TNewComponent : public TComponent
{
private:
protected:
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
__published:
};
```

```
//---------------------------------------------------------------------------
#endif
```

You should save the .CPP file right away, giving it a meaningful name. When you do, the header file name changes to the same name as the .CPP file except it has an .H file extension instead of a .CPP file extension.

Creating a component manually

See also

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

To create a component manually, follow these steps:

1 Create a unit
2 Derive the component
3 Declare a new constructor
4 Register the component

Creating a unit

See also

A C++Builder unit is comprised of a .CPP file and an .H file combination that is compiled into an .OBJ file. C++Builder uses units for a number of purposes. Every form has its own unit, and most components (or logical groups of components) have their own units as well.

When you create a component, you either create a new unit for the component, or add the new component to an existing unit.

To create a unit for a component, choose one of these methods:

- Choose File|New to display the New Items dialog box, select the New tab, select Unit, and choose OK.
- Choose File|New Unit.

C++Builder creates a .CPP file and a header file and displays the .CPP file in the Code editor. Save the file with a meaningful name.

To open the header file, place your cursor on the name of the header file in the Code editor, right-click the name to display a menu, and choose Open File at Cursor.

Once you have either a new or existing unit for your component, you can derive the component class.

To add a component to an existing unit, choose File|Open to choose the source code for an existing unit.

Note:

When adding a component to an existing unit, make sure that unit already contains only component code. Adding component code to a unit that contains, for example, a form, causes errors in the Component palette.

Once you have either a new or existing unit for your component, you can derive the component class.

Deriving the component

See also Example

Every component is a class descended from *TComponent*, from one of its more specialized descendants, such as *TControl* or *TGraphicControl*, or from an existing component class. How do you create components? describes which class to derive from for different kinds of components.

Deriving new classes is explained in more detail in Creating new classes.

To derive a component class, add a class declaration to the header file.

An example of deriving a component

To create, for example, the simplest component class, a nonvisual component descended directly from TComponent, add the following class declaration to your header fileto the interface part of your component unit:

```
class TNewComponent : public TComponent
{
};
```

You should also add the necessary include statements that specify the .HPP files needed for the new component. These are the most common include statements you need:

```
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
```

Declaring a new constructor

See also

Each new component must have a constructor that overrides the constructor of the class from which it was derived. When you write the constructor for your new component, it must *always* call the inherited constructor.

Within the class declaration, declare a virtual constructor in the public section of the class. You can learn more about the public section in Controlling access.

For example,

```
class TNewComponent : public TComponent
{
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
};
```

In the .CPP file, implement the constructor:

```
__fastcall TNewComponent::TNewComponent(TComponent* AOwner): TComponent(AOwner)
{
}
```

Within the constructor, you add the code you want to execute when the component is created.

Registering the component

See also Example

Registering a component is a simple process that tells C++Builder which components to add to its component library, and on which pages of the Component palette the components should appear. Registering components describes the registration process and its nuances in much more detail.

To register a component,

1 Add a function named *Register* to the unit's .CPP file, placing it within a namespace. The namespace is the name of the file the component is in, minus the file extension, with all lowercase letters except the first letter.

   For example, this code exists within a *Newcomp* namespace, whereas *Newcomp* is the name of the .CPP file:

```
namespace Newcomp
{
  void __fastcall Register()
  {
  }
}
```

2 Within the *Register* function, declare an open array of type *TComponentClass* that holds the array of components you are registering. The syntax should look like this:

```
TComponentClass classes[1] = {__classid(TNewComponent)};
```

   In this case, the array of classes contains just one component, but you can add all the components you want to register to the array.

3 Within the *Register* function, call *RegisterComponents* for each component you want to register.

   RegisterComponents is a function that takes three parameters: the name of a Component palette page, the array of component classes, and the size - 1 of the component classes. If you're adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

   You can register multiple components with just one *RegisterComponents* call if all components go on the same page on the Component palette.

An example of registering a component

To register a component named *TNewComponent* and place it on the Samples page of the Component palette, add the following *Register* function to the .CPP file of the unit that contains *TNewComponent*'s declaration:

```
namespace Newcomp
{
  void __fastcall Register()
  {
     TComponentClass classes[1] = {__classid(TNewComponent)};
     RegisterComponents("Samples", classes, 0);
  }
}
```

This *Register* call places *TNewComponent* on the Samples page of the Component palette.

Once you register a component, you can test the component, and finally install the component onto the Component palette.

Testing uninstalled components

Example

You can test the runtime behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly-created components, but you can use the same technique for testing any component, regardless of whether the component appears on the Component palette.

Testing your components without installing has the added benefit of generating compile-time errors that are seen only when the class is instantiated. For example, trying to create an instance of an abstract class yields an error directing you to the pure virtual that must be overloaded.

In essence, you can test an uninstalled component by emulating the actions performed by C++Builder when a user places a component from the Component palette on a form.

To test an uninstalled component, do the following:

1 Create a new application or open an existing one.

2 Choose Project|Add to Project to add the component unit to your project.

3 Include the .H file of the component unit in the header file of a form unit.

4 Add a data member to the form to represent the component.

   This is one of the main differences between the way you add components and the way C++Builder does it. You add the data member to the public part at the bottom of the form's class declaration. C++Builder would add it above, in the published part of the class declaration that it manages.

   You should never add data members to the C++Builder-managed part of the form's class declaration. The items in that part of the class declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.

5 Construct the component in the form's constructor.

   When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You nearly always pass **this** as the owner. In a method, **this** is a reference to the class that contains the method. In this case, in the form's *OnCreate* handler, **this** refers to the form.

6 Assign the *Parent* property.

   Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that visually contains the control, which is most often the form, but might be a group box or panel. Normally, you'll set *Parent* to **this**, that is, the form. Always set *Parent* before setting other properties of the control.

Warning:

   If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property to **this** instead of the component's, you can cause Windows to crash.

7 Set any other component properties as desired.

An example of testing uninstalled components

Suppose you want to test a new component of class *TNewControl* in a unit named *NewCtrl*. Create a new project, then follow the steps to end up with a form unit that looks like this:

```
#ifndef TestFormH
#define TestFormH
//---------------------------------------------------------------------------
#include <classes.hpp>
#include <controls.hpp>
#include <stdCtrls.hpp>
#include <forms.hpp>
#include "NewCtrl.h"                            // 2. Add NewCtrl to the form header file
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:
private:
public:
    TNewControl* NewControl1;                          // 3. Add a data member
    virtual __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

The include statement that includes the NEWCOMP.H file assumes that the component resides in the directory of the current project or in a directory that is on the include path of the project.

This is the .CPP file of the form unit:

```
#include <vcl.h>
#pragma hdrstop
#include "TestForm.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  NewControl1 = new TNewControl(this);                    // 4. Construct the component
  NewControl1->Parent = this;         // 5. Set Parent property if component is a control
  NewControl1->Left = 12;                           // 6. Set other properties as needed
}
```

Installing a component on the Component palette

See also

When you install a component on the Component palette, the component library is rebuilt. Any time the component library is rebuilt, either by installing a component, or by executing the Component|Rebuild Library command, C++Builder creates a temporary library source file. The name of the library source file is the name of the library as a .CPP file. For VCL, the name of the file is CMPLIB32.CPP.

To save the library source file that is generated, choose Options|Environment|Library|Save Library Source Code.

Now when you rebuild the component library, a CMPLIB32.CPP file is created and remains so that you can browse through it.

Where to place your component

See also

Before you install your new component, move all the component's files to the CBuilder\Lib\Obj directory. This includes:

- All binary files (.DFM, .RES or .RC, .DCR)
- All source files (.CPP, .PAS)
- All .OBJ and .LIB files
- All header files (.H and .HPP)

Adding a component

See also

You can add either a C++ or Pascal component to the component library and have it appear on the Component palette.

To add components to the component library,

1 Choose Component|Install.

   The Install Components dialog box appears.

2 Choose Add to open the Add Module dialog box.

3 In the Add Module dialog box, type the name of the unit you want to add, or choose Browse to specify a search path.

   Note that you can't specify a literal path in the Add Module dialog box. The path you choose in the Add Module Browse dialog box adds the directory path to the search path listed in the Install Components dialog box.

4 Choose OK to close the Add Module dialog box.

   The name(s) of the component unit(s) you have specified appears at the bottom of the Installed Components list. If you select the unit name, the class names for components already residing in the library are displayed in the Component classes list. Class names for newly added components are not shown.

5 Choose OK to close the Install Components dialog box and rebuild the library.

   Note that even if you've made no changes to the Installed Components list, C++Builder rebuilds the library. If you have unsaved forms open, you're prompted to save them. Once the library is rebuilt, the components you've installed are reflected in the Component palette.

Note:

Newly installed components initially appear on the page of the Component palette that was specified by the component writer in the component source code. You can move the components to a different page after they've been installed on the palette with the Component|Configure Palette dialog box.

Modifying how the component library is built

See also

When you install one or more components on the Component palette, the component library is rebuilt. To rebuild, C++Builder follows the build process specified in the default CMPLIB32.MAK file, which is a make file.

If you want to customize how the Component palette is built, choose Options|Environment|Library and make the changes you want.

OOP for component writers

See also

Working with C++Builder, you've encountered the idea that a class contains both data and code, and that you can manipulate classes both at design time and runtime. In that sense, you've become a component user.

When you create new kinds of components, you deal with classes in ways that end users never need to. Before you start creating components, you need to be familiar with these topics, which are related to object-oriented programming (OOP):

- Creating new classes
- Ancestors and descendants
- Controlling access
- Dispatching methods
- Classes and pointers

Creating new classes

See also

The primary difference between component users and component writers is that writers create new types of classes and users manipulate instances of classes. This concept is fundamental to object-oriented programming, and an understanding of the distinction is extremely important if you plan to create your own components.

The concept of types and instances is not unique to classes. Programmers continually work with types and instances, but they don't generally use that terminology. As a programmer you generally create variables of a type. Those variables are instances of the type.

Classes are generally more complex than simple types such as **int**, but by assigning different values to instances of the same type, a user can perform quite different tasks.

For example, it's quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of class *TButton*, but by assigning different values to the *Text*, *Default*, and *Cancel* properties and assigning different handlers to the *OnClick* events, the user makes the two instances do very different things.

Deriving new classes

See also

The purpose of defining classes is to provide a basis for useful instances. That is, the goal is to create a class that you or other users can use in different applications in different circumstances, or at least in different parts of the same application.

There are two reasons to derive new classes:

- Changing class defaults to avoid repetition
- Adding new capabilities to a class

In either case, the goal is to create reusable classes. If you plan ahead and design your classes with future reuse in mind, you can save a lot of later work. Give your classes usable default values, but make them customizable.

Changing class defaults to avoid repetition

See also

In all programming tasks, needless repetition is something to avoid. If you find yourself rewriting the same lines of code over and over, you should either place the code in a function, or build a library of routines you'll use in many programs.

The same reasoning holds for components. If you frequently find yourself changing the same properties or making the same method calls, you should probably create a new component class that does those things by default.

For example, it's possible that each time you create an application, you find yourself adding a dialog box form to perform a particular function. Although it's not difficult to recreate the dialog box each time, it's also not necessary. You can design the dialog box once, set its properties, and then install a wrapper component associated with the dialog box onto the Component palette, making the dialog box a reusable component. Not only can this reduce the repetitive nature of the task, it also encourages standardization and reduces the chance of error in recreating the dialog box.

Adding new capabilities to a class

See also

The other reason for creating a new kind of component is that you want to add capabilities not already found in the existing components. When you do that, you can either derive from an existing component class (for example, creating a specialized kind of list box) or from an abstract, base class, such as *TComponent* or *TControl*.

As a general rule, derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you can't take them away, so if an existing component class contains properties that you *don't* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add some capability to a list box, you would derive your new component from *TListBox*. However, if you want to add some new capability but exclude some existing capabilities of the standard list box, you need to derive your new list box from *TCustomListBox*, the ancestor of *TListBox*. Next, recreate or make visible the list box capabilities you want to include. Finally, add your new features.

Declaring a new component class

See also Example

When you decide that you need to derive a new class of component, you then need to decide what class to derive your new class *from*. As with adding new capabilities to an existing class, the essential rule to follow is this: Derive from the class that contains as much as possible that you want in your component, but which contains nothing that you don't want in your component.

C++Builder provides a number of abstract component classes specifically designed for component writers to use as bases for deriving new component classes. The Component creation starting points topic shows the different classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's header file.

An example of declaring a new component class

Here is the declaration of a simple graphical component:

```
class TSampleShape : public TGraphicControl
{
public:
  virtual __fastcall TSampleShape(TComponent *Owner);
};
```

A finished component declaration includes property, data member, and method declarations before the final }, but an empty declaration is also valid, and provides a starting point for the addition of component features.

Ancestors and descendants

See also

From a component user's standpoint, a class is a self-contained entity consisting of properties, methods, and events. Component users don't need to know or care about such issues as which class a given component is derived from. But these issues are extremely important to you as a component writer.

Component users can take for granted that every control has properties named *Top* and *Left* that determine where the control appears on the form that owns it. To them, it does not matter that all controls inherit those properties from a common ancestor, *TControl*. When you create a control, however, you must know which class to derive from so as to inherit the appropriate parts. You also must know everything your control inherits, so you can take advantage of inherited features without recreating them.

From the definition of component classes, you know that when you define a component, you derive it from an existing class. The class you derive from is called the *immediate ancestor* of your new class. The immediate ancestor of the class is called an *ancestor* of the new class, as are all of *its* ancestors. The new class is called a *descendant* of its ancestors.

If you do not specify an ancestor class, C++Builder derives your class from the default ancestor class, *TObject*. Ultimately, the standard class *TObject* is an ancestor of all classes in the Visual Component Library.

Class hierarchies

All the ancestor-descendant relationships in an application result in a hierarchy of classes. The most important thing to remember about class hierarchies is that each "generation" of descendant classes contains more than its ancestors. That is, a class inherits everything that its ancestor contains, then adds new data and methods or redefines existing methods.

A class cannot remove anything it inherits, however. For example, if a class has a particular property, all descendants of that class, direct or indirect, will also have that property.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling access

See also

There are five levels of *access control* on the parts of classes. Access control lets you specify which code can access which parts of the class. By specifying levels of access, you define the *interface* to your components. If you plan the interface carefully, you improve both the usability and reusability of your components.

Unless you specify otherwise, the data members, methods, and properties you add to your classes are private.

The following table shows the levels of access, in order, from most restrictive to most accessible:

Protection	Used for
private	Hiding implementation details
protected	Defining the developer's interface
public	Defining the runtime interface
__published	Defining the design-time interface
__automated	For OLE automation

Hiding implementation details

See also Example

Declaring part of a class as **private** makes that part invisible to code outside the class unless the functions are friends of the class. Private parts of a class are mostly useful for hiding details of implementation from users of the class. Because users of the class can't access the private parts, you can change the internal implementation of the class without affecting user code.

If you don't specify any access control (**private**, **protected**, **public, __published**, or **__automated**) on a data member, method, or property, that part is **private**.

An example of hiding implementation details

Here is an example shown in two parts that illustrates how declaring a data member as **private** prevents users from accessing information.

The first part is a form unit made up of a header file and a .CPP file that assigns a value to a private data member in the form's *OnCreate* event handler. Because the event handler is declared within the *TSecretForm* class, the unit compiles without error.

```
#ifndef HideInfoH
#define HideInfoH
//---------------------------------------------------------------------------
#include <classes.hpp>
#include <controls.hpp>
#include <stdCtrls.hpp>
#include <forms.hpp>
//---------------------------------------------------------------------------
class TSecretForm : public TForm
{
__published:    // IDE-managed Components
    void __fastcall FormCreate(TObject *Sender);
private:
    int FSecretCode;                                    // declare a private data member
public:         // User declarations
    virtual __fastcall TSecretForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TSecretForm *SecretForm;
//---------------------------------------------------------------------------
#endif
```

This is the accompanying .CPP file:

```
#include <vcl.h>
#pragma hdrstop
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TSecretForm *SecretForm;
//---------------------------------------------------------------------------
__fastcall TSecretForm::TSecretForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TSecretForm::FormCreate(TObject *Sender)
{
    FSecretCode = 42;                           // this compiles correctly
}
```

The second part of this example is another form unit that attempts to assign a value to the *FSecretCode* data member in the *SecretForm* form. This is the header file for the unit:

```
#ifndef TestHideH
#define TestHideH
//---------------------------------------------------------------------------
#include <classes.hpp>
#include <controls.hpp>
#include <stdCtrls.hpp>
#include <forms.hpp>
//---------------------------------------------------------------------------
class TTestForm : public TForm
{
__published:    // IDE-managed Components
    void __fastcall FormCreate(TObject *Sender);
public:         // User declarations
    virtual __fastcall TTestForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TTestForm *TestForm;
//---------------------------------------------------------------------------
#endif
```

This is the accompanying .CPP file. Because the *OnCreate* event handler attempts to assign a value to a data member private to the *SecretForm* form, the compilation fails with the error message 'TSecretForm::FSecretCode' is not accessible.

```
#include <vcl.h>
#pragma hdrstop
#include "testHide.h"
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TTestForm *TestForm;
//---------------------------------------------------------------------------
__fastcall TTestForm::TTestForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TTestForm::FormCreate(TObject *Sender)
{
  SecretForm->FSecretCode = 13;                //compiler stops here with error message
}
```

Although a program using the *HideInfo* unit can use classes of type *TSecretForm*, it can't access the *FSecretCode* data member in any of those classes.

Defining the developer's interface

See also

Declaring part of a class as **protected** makes that part invisible to code outside the class like parts declared private. With **protected** parts, however, units that contain classes derived from the class can access the **protected** parts.

You can use **protected** declarations to define a *developer's interface* to the class. That is, users of the class don't have access to the **protected** parts, but derived classes do. In general, that means you can make interfaces available that allow component writers to change the way a class works without making those details visible to end users.

Defining the runtime interface

See also Example

Declaring part of a class as public makes that part visible to any code that has access to the class as a whole. That is, the **public** part has no special restrictions on it.

Public parts of classes are available at runtime to all code, so the **public** parts of a class define that class's *runtime interface*. The runtime interface is useful for items that aren't meaningful or appropriate at design time, such as properties that depend on actual runtime information or which are read-only. Methods that you intend for users of your components to call should also be declared as part of the runtime interface.

Note that read-only properties can't operate at design time, so they should appear in the **public** declaration section.

An example of defining the runtime interface

Here is an example that shows two read-only properties declared as part of a component's runtime interface:

```
class TSampleComponent : public TComponent
{
private:
  int FTempCelsius;                                  // implementation details are private
  int GetTempFahrenheit();
public:
  ...
  __property int TempCelsius = {read=FTempCelsius};          // properties are public
  __property int TempFahrenheit = {read=GetTempFahrenheit};
};
```

This is the *GetTempFahrenheit* method in the .CPP file:

```
int TSampleComponent::GetTempFahrenheit()
{
   return FTempCelsius * (9 / 5) + 32;
}
```

Because the user can't change the value of the properties, the properties should not appear in the Object Inspector, and, therefore, they should not be part of the design-time interface.

Defining the design-time interface

See also Example

Declaring part of a class as **__published** makes that part public and also generates runtime type information for the part. Among other things, runtime type information ensures that the Object Inspector can access properties and events.

Because only published parts show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that a user might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Note:

Read-only properties cannot be part of the design-time interface because the user cannot alter them. Read-only properties should be **public**.

An example of defining the design-time interface

Here is an example of a published property. Because it is published, it appears in the Object Inspector at design time.

```
class TSampleComponent : public TComponent
{
private:
  int FTemperature;
  ...
__published:
  __property int Temperature = {read=FTemperature, write=FTemperature};
};
```

Temperature, the property in this example, is available at design time, so users of the component can adjust the value.

Dispatching methods

Dispatch is the term used to describe how your application determines which class method should be invoked when it encounters a class method call. When you write code that calls a class method, it looks like any other function call. Classes, however, have two different ways of dispatching methods.

The two types of method dispatch are

- Regular (not virtual) methods
- Virtual methods

Regular methods

See also

Class methods are regular (or nonvirtual) unless you specifically declare them as virtual, or unless they override a virtual method in a base class. The compiler can determine the exact address of a regular class member at compile time. This is known as compile-time binding.

A base class regular method is inherited by derived classes. In the following example, an object of type *Derived* can call the method *Regular()* as it were it's own method. Declaring a method in a derived class with the same name and parameters as a regular method in the class's ancestor *replaces* the ancestor's method. In the following example, when *d->AnotherRegular()* is called, it is being dispatched to the *Derived* class replacement for *AnotherRegular()*.

```
class Base
{
public:
    void Regular();
    void AnotherRegular();
    virtual void Virtual();
};
class Derived : public Base
{
public:
    void AnotherRegular();          // replaces Base::AnotherRegular()
    void Virtual();                 // overrides Base::Virtual()
};
void FunctionOne()
{
    Derived *d;
    d = new Derived;
    d->Regular();                   // Calling Regular() as it were a member of Derived
                                    // The same as calling d->Base::Regular()
    d->AnotherRegular();            // Calling the redefined AnotherRegular(), ...
                                    // ... the replacemment for Base::AnotherRegular()
    delete d;
}
void FunctionTwo(Base *b)
{
    b->Virtual();
    b->AnotherRegular();
}
```

Virtual methods

See also

Unlike regular methods, which are bound at compile time, virtual methods are bound at *runtime*. The virtual mechanism of C++ allows a method to be called depending on the class type that is being used to invoke the method.

In the previous example, if you were to call *FunctionTwo()* with a pointer to a *Derived* object, the function *Derived::Virtual()* would be called. The virtual mechanism dynamically inspects the class type of the object you passed at runtime and dispatches the appropriate method. But the call to the regular function *b>AnotherRegular()* will always call *Base::AnotherRegular()* because the address of *AnotherRegular()* was determined at compile time.

To declare a new virtual method, preface the method declaration with the **virtual** keyword.

When the compiler encounters the **virtual** keyword, it creates an entry in the class's virtual method table (VMT). The VMT holds the address of all the virtual methods in a class. This lookup table is used at runtime to determine that *b>Virtual* should call *Derrived::Virtual()*, and not *Base::Virtual()*.

When you derive a new class from an existing class, the new class receives its own VMT, which includes the entries from its ancestor's VMT, plus any additional virtual methods declared in the new class. In addition, the descendant class can *override* any of its inherited virtual methods.

Overriding methods

See also Example

Overriding methods means extending or refining an ancestor's method, rather than replacing it. To override a method in a descendant class, redeclare the method in the derived class, ensuring that the number and type of arguments are the same.

An example of overriding methods

The following code shows the declaration of two simple components. The first declares two methods, each with a different kind of dispatching. The other, derived from the first, replaces the nonvirtual method and overrides the virtual method.

```
class TFirstComponent : public TComponent
{
public:
  void Move();                              // regular method
  virtual void Flash();                     // virtual method
};
class TSecondComponent : public TFirstComponent
{
public:
  void Move();                // declares new method "hiding" TFirstComponent::Move()
  void Flash();               // overrides virtual TFirstComponent::Flash in TFirstComponent
};
```

Classes and pointers

One thing to be aware of when writing components that you don't need to consider when using existing components is that every class (and therefore every component) is really a pointer.

This becomes important when you pass classes as parameters. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references. Passing a class by reference, then, would be passing a reference to the reference.

Creating properties

See also

Properties are the most distinctive parts of components, largely because component users can see and manipulate them at design time and get immediate feedback as the components react in real time. Properties are also important because, if you design them well, they make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining component properties
- Creating array properties
- Writing property editors

Why create properties?

See also

Properties provide significant advantages, both for you as a component writer and for the users of your components. The most obvious advantage is that properties can appear in the Object Inspector at design time. That simplifies your programming job, because instead of handling several parameters to construct a class, you just read the values assigned by the user.

From the component user's standpoint, properties look like variables. Users can set or read the values of properties much as if those properties were class data members. About the only thing they cannot do with a property that they would with a variable is pass it as an argument to a method by reference.

From the component writer's standpoint, however, properties provide much more power than simple class data members because

- Users can set properties at design time.
  This is very important, because unlike methods, which are only available at runtime, properties let users customize components before running an application. In general, your components should not contain a lot of methods; most of them can probably be encapsulated into properties.
- Unlike a data member, a property can hide implementation details from users.
  For example, the data might be stored internally in an encrypted form, but when setting or reading the value of the property, the data would appear unencrypted. Although the value of a property might be a simple number, the component might look up the value from a database or perform complex calculations to arrive at that value.
- Properties allow side effects to outwardly simple assignments.
  What appears to be a simple assignment involving a data member can be a call to a method, and that method could do almost anything.

  A simple example is the *Top* property of all components. Assigning a new value to *Top* doesn't just change some stored value; it causes the component to relocate and repaint itself. The effects of property setting need not be limited to an individual component. For example, setting the *Down* property of a speed-button component to true causes the speed button to set the *Down* properties of all other speed buttons in its group to false.
- The implementation methods for a property can be virtual, meaning that what looks like a single property to a component user might do different things in different components.

Types of properties

See also

A property can be of any type. The most important aspect of choosing types for your properties is that different types appear differently in the Object Inspector. The Object Inspector uses the type of the property to determine what choices appear to the user. You can specify a different property editor when you register your components, as explained in "Writing property editors" in this chapter Writing property editors.

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 is also a drop-down list that shows all possible values of the enumerated type.
Set	Properties of set types appear in the Object Inspector looking like 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 or false if it's not included.
Object	Properties that are themselves classes often have their own property editors. However, if the class that is a property also has published properties, the Object Inspector allows the user to expand the list of class 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.

Publishing inherited properties

See also Example

All components inherit properties from their ancestor types. When you derive a new component from an existing component type, your new component inherits all the properties in the ancestor type. If you derive instead from one of the abstract types, many of the inherited properties are either **protected** or **public**, but not **__published**.

If you need more information about levels of protection such as **protected**, **private**, and **__published**, see Controlling access.

To make a **protected** or **public** property appear in the Object Inspector so the user can access it at design time, you must *redeclare* the property as **__published**.

Redeclaring means adding the declaration of an inherited property to the declaration of a descendant class.

An example of publishing an inherited property

If you derive a component from TWinControl, for example, it inherits a Ctl3D property, but that property is **protected**, so users of the component cannot access Ctl3D at design time or runtime. By redeclaring Ctl3D in your new component, you can change the level of protection to either **public** or **__published**.

The following code shows a redeclaration of Ctl3D as **__published**, making it available at design time:

```
class TSampleComponent : public TWinControl
{
__published:
  __property Ctl3D;
};
```

Note that redeclarations can only make a property less restricted, not more restricted. Thus, you can make a protected property **public**, but you cannot hide a **public** property by redeclaring it as **protected**.

When you redeclare a property, you specify only the property name, not the type and other information described in Defining component properties. You can also declare new default values when redeclaring a property, or specify whether to store the property.

Defining component properties

See also

This section focuses on how to declare properties in C++Builder components and the conventions used by the standard components.

Specific topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

The property declaration

See also Example

Declaring a property and its implementation is easy. You add the property declaration to the declaration of your component class.

To declare a property, you specify three things:

- The name of the property
- The type of the property
- Methods to read and/or set the value of the property

At a minimum, a component's properties should be declared in a **public** part of the component's class declaration, making it easy to set and read the properties from outside the component at runtime.

To make the property editable at design time, declare the property in a **__published** part of the component's class declaration. Published properties automatically appear in the Object Inspector. Public properties that aren't published are available only at runtime.

An example of a property declaration

Here is a typical property declaration:

```
class TYourComponent : public TComponent
{
private:
   int FCount;                                          // data member for storage
   int __fastcall GetCount();                           // read method
   void __fastcall SetCount( int ACount );              // write method
public:
   __property int Count = {read=GetCount, write=SetCount};   // property declaration
  ...
};
```

Internal data storage

See also

There are no restrictions on how you store the data for a property. In general, however, C++Builder's components follow these conventions:

- Property data is stored in data members.
- Identifiers for properties' data members start with the letter *F*, and incorporate the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a class data member called *FWidth*.
- Data members for property data should be declared as **private**. This ensures that the component that declares the property has access to them, but component users and descendant components don't. Derived components should use the inherited property itself, not direct access to the internal data storage, to manipulate a property.

The underlying principle behind these conventions is that only the implementation methods for a property should access the data behind that property. If a method or another property needs to change that data, it should do so through the property, *not* by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

Direct access

See also Example

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal storage data member without calling an access method. Direct access is useful when the property has no side effects, but you want to make it available in the Object Inspector.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part, usually to update the status of the component based on the new property value.

An example of a property that uses direct access

The following component-type declaration shows a property that uses direct access for both the **read** and **write** parts:

```
class TSampleComponent : public TComponent
{
private:                                        // internal storage is private
  bool FReadOnly;                               // declare data member to hold value
  ...
__published:                                    // make property available at design time
  __property bool ReadOnly = {read=FReadOnly, write=FReadOnly};
};
```

Access methods

See also Example

The syntax for property declarations allows the **read** and **write** parts of a property declaration to specify access methods instead of a data member. Regardless of how a particular property implements its read and write parts, however, that implementation should be protected, and usually declared as **virtual**. Users can then create descendant components that override the property implementation, bringing polymorphic behavior to the property.

You should avoid making access methods public, however. Keeping access methods **protected** ensures that component users don't accidentally call those methods, inadvertently modifying a property.

The read method

The **read** method for a property is a function that takes no parameters, and returns a value of the same type as the property. By convention, the function's name is "Get" followed by the name of the property. For example, the **read** method for a property named *Count* would be named *GetCount*.

The only exceptions to the no parameters rule are for array properties, which pass their indexes as parameters, and for properties that use the index specifier, which also pass index values.

The **read** method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

If you don't declare a **read** method, the property is write-only. Write-only properties are very rare, and generally not very useful.

The write method

The **write** method for a property is always a member function that takes a parameter, of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the method's name is "Set" followed by the name of the property. For example, the **write** method for a property named *Count* would be named *SetCount*.

The value passed in the parameter is used to set the new value of the property, so the **write** method needs to perform any manipulation needed to put the appropriate values in the internal storage.

If you don't declare a **write** method, the property is read-only.

Properties can share read and write methods. By using the index specifier, you can write one **read** or **write** method that can be used by multiple properties. Properties that use the index specifier pass an index value to their access methods. Click Example on this screen to see an example of properties that use the index specifier.

It's common to test whether the new value actually differs from the current value before setting the value. For example, here's a simple **write** method for an integer property called *Count* that stores its current value in a field called *FCount*:

```
void __fastcall TMyComponent::SetCount( int Value )
{
  if ( Value != FCount ) {
    FCount = Value;
    Update();
    }
}
```

An example of property access methods

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same **read** and **write** access methods:

```
class TSampleCalendar : public TCustomGrid
{
private:
  int __fastcall GetDateElement(int Index);    // note Index parameter
  void __fastcall SetDateElement(int Index, int Value);
public:
__property int Day = {read=GetDateElement, write=SetDateElement, index=3,
nodefault};
__property int Month = {read=GetDateElement, write=SetDateElement, index=2,
nodefault};
__property int Year = {read=GetDateElement, write=SetDateElement, index=1,
nodefault};
};
```

Because each element of the date (day, month, and year) is an **int**, and because setting each requires encoding the date when set, the code avoids duplication by sharing the **read** and **write** methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the **read** method that obtains the date element:

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &Aday); // break date into elements
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
```

This is the **write** method that sets the appropriate date element:

```
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0)                  // all elements must be positive
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay); // get date elements
    switch (Index)
    {
      case 1: AYear = Value;  break;
      case 2: AMonth = Value; break;
      case 3: ADay = Value;   break;
      default: return;
    }
 }
 FDate = TDateTime(AYear, AMonth, ADay);  // encode the modified date
 Refresh();                              // update the visible calendar
}
```

Default property values

See also Example

When you declare a property, you can optionally declare a *default value* for the property. The default value for a component's property is the value set for that property in the component's constructor. For example, when you place a component from the Component palette on a form, C++Builder creates the component by calling the component's constructor, which determines the initial values of the component's properties.

C++Builder uses the declared default value to determine whether to store a property in a form file. For more information on storing properties and using the stored specifier, and the importance of default values, see Storing and loading properties. If you do not specify a default value for a property, C++Builder *always* stores the property.

To declare a default value for a property, append an equal sign after the property name and a set of braces that holds the **default** keyword and the default value. For example,

```
__property bool IsTrue = {default=true};
```

Note!:

Declaring a default value in the property declaration **does not** actually set the property to that value. It is your responsibility as the component writer to ensure that the component's constructor actually sets the property to that value.

Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append an equal sign after the property name and a set of braces that holds the **nodefault** keyword. For example,

```
__property int NewInteger = {nodefault};
```

When you declare a property for the first time, there is no need to specify **nodefault** because the absence of a declared default value means the same thing.

An example of a property with a default value

Here is the declaration of a component that includes a single Boolean property named *IsTrue* with a default value of true, including the constructor that sets the default value.

```
class TSampleComponent : public TComponent
{
private:
  bool FIsTrue;
public:
  virtual __fastcall TSampleComponent( TComponent* Owner );
__published:
  __property bool IsTrue = {read=FIsTrue, write=FIsTrue, default=true};
};
__fastcall TSampleComponent::TSampleComponent ( TComponent* Owner )
  : TComponent ( Owner )
{
   FIsTrue = true;
}
```

Note that if the default value for *IsTrue* had been false, you would not need to set it explicitly in the constructor, because all classes (and therefore, components) always initialize all their data members to zero, and a "zeroed" Boolean value is false.

Creating array properties

See also Example

Some properties lend themselves to being indexed, much like arrays. That is, they have multiple values that correspond to some kind of index value. An example in the standard components is the *Lines* property of the *TMemo* component. *Lines* is an indexed list of the strings that make up the text of the memo, which you can treat as an array of strings. In this case, the array property gives the user natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties work just like other properties, and you declare them in largely the same way. The only differences in declaring array properties are as follows:

- The declaration for the property includes one or more indexes with specified types. Indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, *must* be methods. They cannot be data members.
- The access methods for reading and writing the property values take additional parameters that correspond to the index or indexes. The parameters must be in the same order and of the same type as the indexes specified in the property declaration.

Although they seem quite similar, there are a few important distinctions between array properties and arrays. Unlike the index of an array, the index type for an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can only reference individual elements of an array property, not the entire range of the property.

An example of an array property

Here's the declaration of a property that returns a string based on an integer index:

```
class TDemoComponent : public TComponent
{
private:
    System::AnsiString __fastcall GetNumberSize(int Index);
public:
    __property System::AnsiString NumberSize[int Index] = {read=GetNumberSize};
  ...
};
```

This is the *GetNumberSize* method in the .CPP file:

```
System::AnsiString __fastcall TDemoComponent::GetNumberSize(int Index)
{
  System::AnsiString Result;
  switch (Index)
  {
    case 0:
      Result = "Zero";
      break;
    case 100:
      Result = "Medium";
      break;
    case 1000:
      Result = "Large";
      break;
    default:
      Result = "Unknown size";
    }
    return Result;
 }
```

Writing property editors

See also

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires five steps:

- Deriving a property-editor class
- Editing the property as text
- Editing the property as a whole
- Specifying editor attributes
- Registering the property editor

Deriving a property-editor class

See also Example

The DSGNINTF.HPP file defines several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor types described inTable 4.2 the table below.

To create a property-editor class, derive a new class from one of the existing property editor types.

The DSGNINTF.HPP file also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings.
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop-down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

An example of a property-editor class

One of the simplest property editors is *TFloatPropertyEditor*, the editor for properties that are floating-point numbers. Here is its declaration:

```
class TFloatProperty : public TPropertyEditor
{
    typedef TFloatProperty ThisClass;
    typedef TPropertyEditor inherited;
public:
    virtual bool __fastcall AllEqual(void);
    virtual System::AnsiString __fastcall GetValue(void);
    virtual void __fastcall SetValue(const AnsiString Value);
};
```

Editing the property as text

See also

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor classes provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits a set of methods used for assigning and reading different sorts of values, as shown inTable 4.3 the following table

Property type	"Get" method	"Set" method
Floating point	*GetFloatValue*	*SetFloatValue*
Closure (event)	*GetMethodValue*	*SetMethodValue*
Ordinal type	*GetOrdValue*	*SetOrdValue*
String	*GetStrValue*	*SetStrValue*.

When you override a *GetValue* method, you will call one of the "Get" methods, and when you override *SetValue*, you will call one of the "Set" methods.

Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns "unknown".

To provide a string representation of your property, override the property editor's *GetValue* method.

If the property isn't a string value, your *GetValue* must convert the value into a string representation.

Setting the property value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method.

SetValue should convert the string and validate the value before calling one of the "Set" methods.

Editing the property as a whole

See also

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

Edit methods use the same "Get" and "Set" methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a "Get" method and a "Set" method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

Within your implementation of the *Edit* method, follow these steps:

1 Construct the editor you are using for the property.

2 Read the current value and assign it to the property using a "Get" method.

3 When the user selects a new value, assign that value to the property using a "Set" method.

4 Destroy the editor.

Specifying editor attributes

See also Example

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Flag	Related method	Meaning if included
paValueList	*GetValues*	The editor can give a list of enumerated values.
paSubProperties	*GetProperties*	The property has subproperties that can display.
paDialog	*Edit*	The editor can display a dialog box for editing the entire property.
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	*SetValue*	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.

An example of specifying editor attributes

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList];
end;
```

If the *GetAttributes* method were written in C++, it would look like this:

```
virtual __fastcall TPropertyAttributes TColorProperty::GetAttributes()
{
  return TPropertyAttributes() << paMultiSelect << paDialog << paValueList;
}
```

Registering the property editor

See also Example

Once you create a property editor, you need to register it with C++Builder. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* method.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.
  Specify the type information like this:
  ```
  __typeinfo(TMyComponent)
  ```
- The type of the component to which this editor applies. If this parameter is NULL, the editor applies to all properties of the given type.
- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

An example of registering a property editor

Here is an excerpt from the function that registers the editors for the standard components on the Component palette:

```
namespace Newcomp
{
  void __fastcall Register()
  {
    RegisterPropertyEditor(__typeinfo(TComponent), 0L, "", TComponentProperty);
    RegisterPropertyEditor(__typeinfo(TComponentName), TComponent, "Name",
TComponentNameProperty);
    RegisterPropertyEditor(__typeinfo(TMenuItem), TMenu, "", TMenuItemProperty);
    ...
  }
}
```

The three statements in this function cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you've created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are NULL and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

Creating events

See also

Events are very important parts of components, although the component writer usually doesn't need to do much with them. An event is a link between an occurrence in the system (such as a user action or a change in focus) that a component might need to respond to and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the component user.

By using events, application developers can customize the behavior of components without having to change the classes themselves. As a component writer, you use events to enable application developers to customize the behavior of your components.

Events for the most common user actions (such as mouse actions) are built into all the standard C++Builder components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

Events are implemented as properties, so you should already be familiar with Creating properties before you attempt to create or change a component's events.

What are events?

See also

Loosely defined, an event is a mechanism that links an occurrence to some code. More specifically, an event is a closure, a pointer to a specific method in a specific class instance.

From the component user's perspective, an event is just a name related to a system event, such as *OnClick*, that the user can assign a specific method to call. For example, a push button called *Button1* has an *OnClick* method. By default, C++Builder generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs on the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.The component user sees the event as a way of specifying what user-written code the application should call when a specific event occurs.

From the component writer's perspective, the most important thing to remember is that you're providing a link, a place where the component's user can attach code in response to certain kinds of occurrences. Your components provide outlets where the user can "plug in" specific code.

To write an event, you need to understand the following:

- Events are closures
- Events are properties
- Event types are closure types
- Event handlers are optional

Events are closures

See also Example

C++Builder uses closures to implement events. A closure is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the closure as a place holder: your code detects that an event occurs, so you call the method (if any) specified by the user for that event.

Closures maintain a hidden pointer to a class instance. When the user assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a specific method of a specific class instance. That instance is usually the form that contains the component, but it need not be.

Calling the click-event handler

All controls, for example, inherit a virtual method called *Click* for handling click events:

```
virtual void __fastcall Click(void);
```

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

Events are properties

See also

Components use properties to implement their events. Unlike most other properties, events don't use methods to implement their read and write parts. Instead, event properties use a private data member of the same type as the property.

By convention, the data member's name is the same as the name of the property, but preceded by the letter *F*. For example, the *OnClick* closure is stored in a data member called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
class TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
  ...
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
  ...
};
```

To learn about *TNotifyEvent* and other event types, see Event types are closure types.

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

Event types are closure types

See also

Because an event is a pointer to an event handler, the type of the event property must be a closure type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class.

To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

C++Builder defines closures for all its standard events. When you create your own events, you can use an existing closure if that's appropriate, or define one of your own.

Event handlers have a return type of void

Event handlers must have a return type of void only. Even though the handler can return only void, you can still get information back from the user's code by passing arguments by reference. When you do this, make sure you assign a valid value to the argument before calling the handler so you don't require the user's code to change the value.

An example of passing arguments by reference to an event handler is the key-pressed event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two arguments, one to indicate which object generated the event, and one to indicate which key was pressed:

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender, Char &Key);
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component might want to change the character. One example might be to force all characters to uppercase in an edit control. In that case, the user could define the following handler for keystrokes:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
  Key = UpCase(Key);
}
```

You can also use arguments passed by reference to let the user override the default handling.

Event handlers are optional

See also

The most important thing to remember when creating events for your components is that users of your components might not attach handlers to the events. That means that your component shouldn't fail or generate errors simply because a user of the component failed to attach a handler to a particular event.

The mechanics of calling handlers and dealing with events that have no attached handler are explained in Calling the event, but the principle has important implications for the design of your components and their events.

The optional nature of event handlers has two aspects:

- Component users are not required to handle events.
  Events happen almost constantly in a Windows application. Just by moving the mouse pointer across a component makes Windows send numerous mouse-move messages to the component, which the component translates into *OnMouseMove* events. In most cases, users of components don't care to handle the mouse move events, and this does not cause a problem. The component does not depend on the mouse events being handled.

  Similarly, the components you create should not be dependent on users handling the events they generate.
- Component users can write any code they want in an event handler.
  In general, there are no restrictions on the code a user can write in an event handler. The components in the C++Builder component library all have events written in such a way that they minimize the chances of user-written code generating unexpected errors. Obviously, you can't protect against logic errors in user code, but you can ensure that all data structures are initialized before calling events so that users don't try to access invalid information.

Implementing the standard events

See also

All the controls that come with C++Builder inherit events for all of the most common Windows events. Collectively, these are called the *standard events*. Although all these events are built into the standard controls, by default they are **protected**, meaning users can't attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events
- Making events visible
- Changing the standard event handling

Identifying standard events

See also

There are two categories of standard events: those defined for all controls and those defined only for the standard windowed controls.

Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether windowed or graphical or custom, inherit these events. The following table lists all the events available in all controls:

OnClick	*OnDragDrop*	*OnEndDrag*	*OnMouseMove*
OnDblClick	*OnDragOver*	*OnMouseDown*	*OnMouseUp*

All the standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names, but without the preceding "On." For example, *OnClick* events call a method named *Click*.

Standard events for standard controls

In addition to the events common to all controls, standard controls (those that descend from *TWinControl*) have the following events:

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

As with the standard events in *TControl*, the windowed-control events have corresponding methods.

Making events visible

See also Example

The declarations of the standard events are **protected** as are the methods that correspond to them. If you want to make those events accessible to users either at runtime or design time, you need to redeclare the event property as either **public** or **__published**.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that's defined in *TControl* but not made visible to users, and promote it to a level so the user can see and use it.

An example of making an event visbile

For example, if you create a component that needs to surface the *OnClick* event at design time, you add the following to the component's class declaration:

```
class TMyControl : public TCustomControl
{
  ...
__published:
    __property OnClick;             // Makes OnClick available in the Object Inspector
};
```

Changing the standard event handling

See also Example

If you want to change the way your custom component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As a component user, that's exactly what you would do. When you're creating components, you can't do that because you must keep the event available for the users of the component.

This is precisely the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the user's code.

The order in which you call the inherited method is significant. As a general rule, you call the inherited method first, allowing the user's event-handler code to execute before your customizations (and in some cases, to keep from executing the customizations). There might be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to the changed status.

An example of changing the standard event handling

Suppose you're writing a component and you want to modify the way your new component responds to clicks. Instead of assigning a handler to the *OnClick* event as a component user would do, you override the protected method *Click*:

```
void __fastcall TMyControl::Click()
{
     TWinControl::Click();        // perform standard handling, including calling handler
    // your customizations go here
}
```

Defining your own events

See also

Defining entirely new events is a relatively rare thing. Usually you refine the handling of existing events. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you'll need to define an event for it.

There are the issues involved in defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

Triggering the event

See also

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a *WM_LBUTTONDOWN* message to the application. Upon receiving that message, a component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

But some events are less clearly tied to specific external events. For example, a scroll bar has an *OnChange* event, which is triggered by numerous kinds of occurrences, including keystrokes, mouse clicks, or changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

Two kinds of events

There are two kinds of occurrences you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component might need to respond to. State-change events might also be related to messages from Windows (focus changes or being enabled, for example), but they can also occur through changes in properties or other code. You have total control over the triggering of events you define. You should be consistent and complete so that users of your components know how to use the events.

Defining the handler type

See also

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for the events you define yourself are either simple notifications or event-specific types. It's also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. All a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it's not enough to know only which event happened and what component it happened to. For example, if the event is a key-press event, it's likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters with any necessary information about the event.

If your event was generated in response to a message, it's likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers return void only, the only way to pass information back from a handler is through a parameter passed by reference. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

Declaring the event

See also

Once you've determined the type of your event handler, you're ready to declare the closure and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of most events in C++Builder begin with "On." This is just a convention; the compiler doesn't enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all closure properties are assumed to be events and appear on the Events page.

Component users expect to find events in the alphabetical list of names starting with "On." Using other kinds of names is likely to confuse them.

Calling the event

See also

You should centralize calls to an event. That is, create a virtual method in your component that calls the user's event handler (if the user assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from your component can customize event handling by overriding that one method, rather than searching through your code looking for places you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid
- Users can override default handling

Empty handlers must be valid

See also Example

You should never create a situation in which an empty event handler causes an error. The proper functioning of your component should not depend on a particular response from the user's event-handler code. In fact, an empty handler should produce the same result as no handler at all.

Components should never require the user to use them in a particular way. An important aspect of that principle is that component users expect no restrictions on what they can do in an event handler.

An example of calling an event handler

Because an empty handler should behave the same as no handler, the code for calling the user's handler should look like this:

```
if (OnClick)
  OnClick(this);
// perform default handling }
```

Warning:

You should *never* have something like this:

```
if (OnClick)
  OnClick(this);
else
  // perform default handling
```

Users can override default handling

See also Example

For some kinds of events, the user might want to replace the default handling or even suppress all responses. To enable users to do that, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

Note that this is in keeping with the notion that empty handlers should have the same effect as no handler at all. Because an empty handler won't change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

An example of overriding default handling

When handling key-press events, for example, the user can suppress the component's default handling of the keystroke by setting the *Key* parameter to a null character. The logic for supporting that looks like this:

```
if (OnKeyPress)
  OnKeyPress(this, &Key);
if (Key != NULL)
  //perform default handling
```

The actual code is a little different from this because it's dealing with Windows messages, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets *Key* to a null character, the component skips the default handling.

Creating methods

See also

Component methods are no different from any other class's methods. That is, they are member functions built into the structure of a component class. Although there are essentially no restrictions on what you can do with the methods of a component, C++Builder does use some standards you should follow.

The guidelines to follow when writing methods for your components include

- Avoiding interdependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

As a general rule, minimize the number of methods users need to call to use your components. A lot of the features you might be inclined to implement as methods are probably better encapsulated into properties. Doing so provides an interface that suits the C++Builder environment, and also lets users access them at design time.

Avoiding interdependencies

See also

At all times when writing components, minimize the preconditions imposed on the component user. To the greatest extent possible, component users should be able to do anything they want to a component, whenever they want to do it. There will be times when you can't accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of interdependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if the user calls it when the component is in a bad state, the method corrects that state before executing its main code. At a minimum, you should throw an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause the user problems. A warning message, for example, is preferable to crashing if the user doesn't accommodate your interdependencies.

Naming methods

See also

C++Builder imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for users of your components, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people might use your components.

If you're accustomed to writing code that only you or a small group of programmers uses, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive.
  A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.
- Function names should reflect the nature of what they return.
  Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.
- If a function return type is void, the function name should be active.
  Use active verbs in your function names. For example, *ReadFileNames* is much more helpful than *DoFiles*.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

Protecting methods

See also

All parts of classes, including data members, methods, and properties, can have various levels of protection, as explained in Controlling access. Choosing the appropriate level of protection for methods is quite simple.

As a general rule, methods you write in your components are either **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that particular type of component, to the point that even components derived from it should not have access to it.

Note:

There is generally no reason for declaring a method (other than an event handler) as **__published**. Doing so looks to the end user exactly as if the method were **public**.

Methods that should be protected

Methods that should be public

Methods that should be public

See also

Any method that users of your components need to be able to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid unduly tying up system resources or putting Windows in a state where it can't respond to the user.

Note:
Constructors and destructors should *always* be **public**.

Methods that should be protected

See also

Any methods that are implementation methods for the component should be **protected** so that users can't call them at the wrong time. If you have methods that a user's code should not call, but the methods are called in derived classes, declare the methods as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there's a chance a user will call it before setting up the data. On the other hand, by making it **protected**, you ensure that the user can't call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the users of the component to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that component users don't accidentally call those methods, inadvertently modifying a property.

Making methods virtual

See also

Virtual methods in C++Builder components are no different from virtual methods in other classes. You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by end users, you can probably make all your methods nonvirtual. On the other hand, if you create components of a more abstract nature, which other component writers will use as the starting point for their own components, consider making the added methods virtual. That way, components derived from your components can override the inherited virtual methods.

Declaring methods

See also Example

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, you do these things:

- Add the declaration to the component's class declaration in the component's header file
- Write the code that implements the method in the .CPP file of the unit

An example of declaring methods

The following code shows a component that defines two new methods, one **protected** method and one **public** virtual method. This is the interface definition in the .H file:

```
class TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
    ...
};
```

This is the code in the .CPP file of the unit that implements the methods:

```
void __fastcall TSampleComponent::MakeBigger()
{
  Height = Height + 5;
  Width = Width + 5;
}
int __fastcall TSampleComponent::CalculateArea()
{
   return Width * Height;
}
```

Using graphics in components

See also

Windows provides a powerful Graphics Device Interface (GDI) for drawing device-independent graphics. The GDI imposes a lot of extra requirements on the programmer, such as managing graphic resources. C++Builder takes care of all the GDI drudgery for you, allowing you to spend your time doing productive work instead of searching for lost handles or unreleased resources. C++Builder tackles the tedious tasks so you can focus on the productive ones.

Note that, as with any part of the Windows API, you can call GDI functions directly from your C++Builder application if you want to. However, you will probably find that using C++Builder's encapsulation of the graphic functions is a much more productive way to create graphics.

There are several important topics dealing with graphics in C++Builder:

- Overview of graphics
- Using the canvas
- Working with pictures
- Offscreen bitmaps
- Responding to changes

Overview of graphics

See also Example

C++Builder encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens and brushes and fonts. After rendering your graphic images, you must then restore the device context to its original state before disposing of it.

Instead of forcing you to deal with graphics at a detailed level, C++Builder provides a simple yet complete interface: your component's *Canvas* property. The canvas ensures that it has a valid device context, and releases the context when you're not using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all those resources for you, so you need not concern yourself with creating, selecting, and releasing things such as pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting C++Builder manage graphic resources is that it can cache resources for later use, which can greatly speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because C++Builder caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, C++Builder reuses an existing one.

An example of simplified graphics code

As an example of how much simpler C++Builder's graphics code can be, here are two samples of code. The first uses standard GDI functions to draw a yellow ellipse outlined in blue on a window in an application written with ObjectWindows. The second uses a canvas to draw the same ellipse in an application written with C++Builder.

This is the ObjectWindows code:

```
void TMyWindow::Paint(TDC& PaintDC, bool erase, TRect& rect)
{
    HPEN PenHandle, OlPenHandle;
    HBRUSH BrushHandle, OldBrushHandle;
    PenHandle = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    OldPenHandle = SelectObject(PaintDC, PenHandle);
    BrushHandle = CreateSolidBrush(RGB(255, 255, 0));
    OldBrushHandle = SelectObject(PaintDC, BrushHandle);
    Ellipse(10, 20, 50, 50);
    SelectObject(OldBrushHandle);
    DeleteObject(BrushHandle);
    SelectObject(OldPenHandle);
    DeleteObject(PenHandle);
)
```

This C++Builder code accomplishes the same thing:

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Pen->Color = clBlue;
  Canvas->Brush->Color = clYellow;
  Canvas->Ellipse(10, 20, 50, 50);
}
```

Using the canvas

See also

The canvas class encapsulates Windows graphics at several levels, ranging from high-level functions for drawing individual lines, shapes, and text to intermediate-level properties for manipulating the drawing capabilities of the canvas to low-level access to the Windows GDI.

The following table summarizes the capabilities of the canvas.

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as MoveTo, LineTo, Rectangle, and Ellipse
	Displaying and measuring text	TextOut, TextHeight, TextWidth, and TextRect methods
	Filling areas	FillRect and FloodFill methods
Intermediate	Customizing text and graphics	Pen, Brush, and Font properties
	Manipulating pixels	Pixels property
	Copying and merging images	Draw, StretchDraw, BrushCopy, and methods; CopyMode property
Low	Calling Windows GDI functions	Handle property

Working with pictures

See also

Most of the graphics work you do in C++Builder is limited to drawing directly on the canvases of components and forms. C++Builder also provides for handling standalone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in C++Builder:

- Using a picture, graphic, or canvas
- Loading and storing graphics
- Handling palettes

Using a picture, graphic, or canvas

See also

There are three kinds of classes in C++Builder that deal with graphics:

- A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a standalone class.
- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. C++Builder defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a metafile, or a user-defined graphic type, and an application can access them all in the same way through the picture class. For example, the image control has a property called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas (the only standard graphic that has a canvas is *TBitmap*). Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

Loading and storing graphics

See also Example

All pictures and graphics in C++Builder can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

To load an image into a picture from a file, call the picture's *LoadFromFile* method.

To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

An example of loading a bitmap

To load a bitmap into an image control's picture pass the name of a bitmap file to the picture's *LoadFromFile* method:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   Image1->Picture->LoadFromFile("c:\\windows\\athena.bmp");
}
```

The picture recognizes .BMP as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

Handling palettes

See also

When running on a palette-based device, C++Builder controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control
- Responding to palette changes

Most controls have no need for a palette, but controls that contain graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as *realizing* palettes.

Realizing palettes is the process of ensuring that the frontmost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the "real" palette. As windows move in front of one another, Windows continually realizes the palettes.

Note:

C++Builder itself provides no specific support for creating or maintaining palettes, other than in bitmaps. If you have a palette handle, however, C++Builder controls can manage it for you.

Specifying a palette for a control

See also

To specify a palette for a control, override the control's *GetPalette* method to return the handle of the palette.

Specifying the palette for a control does these things for your application:

- It tells the application that your control's palette needs to be realized.
- It designates the palette to use for realization.

Responding to palette changes

See also

If your control specifies a palette by overriding *GetPalette*, C++Builder automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*.

The primary role of *PaletteChanged* is to determine whether to realize the control's palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. C++Builder goes one step farther, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.

Offscreen bitmaps

See also

When drawing complex graphic images, a common technique in Windows programming is to create an offscreen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an offscreen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in C++Builder, which represents bitmapped images in resources and files, can also work as an offscreen image.

There are two main aspects to working with offscreen bitmaps:

- Creating and managing offscreen bitmaps
- Copying bitmapped images

Creating and managing offscreen bitmaps

See also

When creating complex graphic images, avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas. The most common use of an offscreen bitmap is in the *Paint* method of a graphic control.

For an example of painting a complex image on an offscreen bitmap, see the source code for the Gauge control from the Samples page of the Component palette. The gauge draws its different shapes and text on an offscreen bitmap before copying them to the screen. Source code for the gauge is in the file GAUGES.PAS in the SOURCE\SAMPLES subdirectory.

Copying bitmapped images

See also

C++Builder provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

The following table summarizes the image-copying methods in canvas objects.

To create this effect	Call this method
Copy an entire graphic	Draw
Copy and resize a graphic	StretchDraw
Copy part of a canvas	CopyRect
Copy a bitmap with raster operations	BrushCopy

Responding to changes

See also Example

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

An example of responding to changes

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes. Although the shape component is written in Object Pascal, the following is a C++ translation of the shape component with a new name, *TMyShape*.

This is the class declaration in the header file:

```
class TMyShape : public TGraphicControl
{
private:
protected:
public:
  virtual __fastcall TMyShape(TComponent* Owner);
__published:
  TPen *FPen;
  TBrush *FBrush;
  void __fastcall StyleChanged(TObject *Sender);
};
```

This is the code in the .CPP file:

```
__fastcall TMyShape::TMyShape(TComponent* Owner)
  : TGraphicControl(Owner)
{
   Width = 65;
   Height = 65;
   FPen = new TPen;
   FPen->OnChange = StyleChanged;
   FBrush = new TBrush;
   FBrush->OnChange = StyleChanged;
}
void __fastcall TMyShape::StyleChanged(TObject *Sender)
{
   Invalidate();
}
```

Handling messages

See also

One of the keys to traditional Windows programming is handling the *messages* sent by Windows to applications. C++Builder handles most of the common ones for you. It's possible, however, that you will need to handle messages that C++Builder doesn't already handle or that you will create your own messages and need to handle them.

There are three aspects to working with messages:

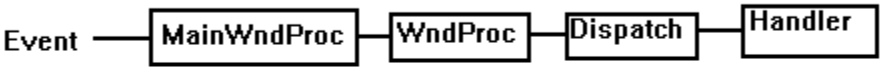
- Understanding the message-handling system
- Changing message handling
- Creating new message handlers

Understanding the message-handling system

See also

All C++Builder classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. You should never need to alter this message-dispatch mechanism. All you'll need to do is create message-handling methods.

What's in a Windows message?

See also

A Windows message can be thought of as a data structure that contains several useful data members. The most important of these is an integer-size value that identifies the message. Windows defines a lot of messages, and the MESSAGES.HPP file declares identifiers for all of them.

Windows programmers are used to working with the Windows definitions that identify a message, such as *WM_COMMAND* or *WM_PAINT*. A traditional Windows program contains a window procedure that serves as a callback for system generated messages. In this window procedure there is usually a large switch statement with case labels for each message this window intends to handle.

Additional useful information is passed to this window procedure in two parameters, *wParam* and *lParam*, for "word parameter" and "long parameter". Often, each parameter contains more than one piece of information and it is necessary to pull out the relevant portions with Windows macros such as *LOWORD* and *HIWORD*. For example, calling HIWORD(lParam) yields the high word of this parameter.

Originally, Windows programmers had to remember or look up in the Windows API what information each parameter contained. Windows has recently implemented "message crackers" to simplify the syntax associated with handling a Windows message and its associated parameters. With "message crackers", instead of using a large switch statement that unpacks all of the information into the parameters, you can simply associate a handler function with the message. If you include WINDOWSX.H into a standard Windows program, the *HANDLE_MSG* macro is available to your program so you can write code like this:

```
void MyKeyDownHandler(  HWND hwnd. UINT nVirtKey, BOOL fDown, int CRepeat, UINT flags )
{
   ...
}

LRESULT MyWndProc( HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam )
{
  switch(  Message )
  {
     HANDLE_MSG( hwnd, WM_KEYDOWN, MyKeyDownHandler );
     ...
}
```

Using this style of message cracking makes it clearer that messages are being dispatched to a particular handler. Also, you can give significant names to the parameter list for your handler function. It is easier to understand a function that takes a parameter called *nVirtKey*, which is the value for *wParam* in a *WM_KEYDOWN* message.

Dispatching messages

See also

C++Builder simplifies message dispatching in several ways:

- Each component inherits a complete message-dispatching system.
- The dispatch system has default handling. You define handlers only for messages you need to respond to specially.
- You can modify small parts of the message-handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component doesn't have a handler defined for the message, the default handling takes care of it, usually by ignoring the message.

Tracing the flow of messages

C++Builder registers a method called *MainWndProc* as the window procedure for each type of component in an application. *MainWndProc* contains an exception-handling block, passing the message structure from Windows to a virtual method called *WndProc* and handling any exceptions by calling the application class's *HandleException* method.

MainWndProc is a nonvirtual method that contains no special handling for any particular messages. Customizations take place in *WndProc*, since each component type can override the method to suit its particular needs.

WndProc methods check for any special conditions that affect their processing so they can "trap" unwanted messages. For example, while being dragged, components ignore keyboard events, so the *WndProc* method of *TWinControl* passes along keyboard events only if the component is not being dragged. Ultimately, *WndProc* calls *Dispatch*, a nonvirtual method inherited from *TObject*, which determines which method to call to handle the message.

Dispatch uses the *Msg* data member of the message structure to determine how to dispatch a particular message. If the component defines a handler for that particular message, *Dispatch* calls the method. If the component doesn't define a handler for that message, *Dispatch* calls *DefaultHandler*.

Changing message handling

See also

Before changing the message-handling of your components, make sure that's what you really want to do. C++Builder translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change the message handling, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

Overriding the handler method

See also Example

To change the way a component handles a particular message, you override the message-handling method for that message. If the component doesn't already handle the particular message, you need to declare a new message-handling method.

To override a message-handling method,

1 Declare a new method in your component with the same name as the method it overrides in the protected part of the component declaration.

2 Map the method to the message it overrides by using three macros.

   The macros take this form:

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(parameter1, parameter2, parameter3)
END_MESSAGE_MAP
```

   Parameter1 is the message index as Windows defines it, *parameter2* is the message structure type, and *parameter3* is the name of the message method.

   You can include as many *MESSAGE_HANDLER* macros as you want between the *BEGIN_MESSAGE_MAP* and *END_MESSAGE_MAP* macros.

An example of overriding a message handler

For example, to override a component's handling of the *WM_PAINT* message, you redeclare the *WMPaint* method, and with three macros, map the method to the *WM_PAINT* message:

```
class TMyComponent : public TComponent
{
protected:
   void __fastcall WMPaint(TWMPaint* Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint)
END_MESSAGE_MAP(TComponent)
};
```

Using message parameters

See also

Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a pointer, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the return value for the message: the value returned by the *SendMessage* call that sends the message.

Because the type of the *Message* parameter in the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters. If for some reason you need to refer to the message parameters by their old-style names (*WParam*, *LParam*, and so on), you can typecast *Message* to the generic type *TMessage*, which uses those parameter names.

Trapping messages

See also Example

Under certain circumstances, you might want your components to ignore certain messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message that way, you override the virtual method *WndProc*.

The *WndProc* method screens messages before passing them to the *Dispatch* method, which in turn determines which method gets to handle the message. By overriding *WndProc*, your component gets a chance to filter out messages before dispatching them. An override of *WndProc* for a control derived from *TWinControl* looks like this:

```
void __fastcall TMyControl::WndProc(TMessage* Message)
{
  // tests to determine whether to continue processing
  TWinControl->WndProc(Message);
}
```

TControl defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding *WndProc* helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

TheWndProc method

Here is part of the *WndProc* method for *TControl* as it is implemented in VCL in Object Pascal:

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ...
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                    { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      ...                                                    { handle others
normally }
    end;
...                                                     { otherwise process
normally }
end;
```

Creating new message handlers

See also

Because C++Builder provides handlers for most common Windows messages, the time you will most likely need to create new message handlers is when you define your own messages. Working with user-defined messages has two aspects:

- Defining your own messages
- Declaring a new message-handling method

Defining your own messages

See also

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard Windows messages and notification of state changes.

Defining a message is a two-step process. The steps are

1 Declaring a message identifier

2 Declaring a message-structure type

Declaring a message identifier

See also Example

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant *WM_USER* represents the starting number for user-defined messages. When defining message identifiers, you should base them on *WM_USER*.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the MESSAGES.HPP file to see which messages Windows already defines for that control.

An example of user-defined messages

The following code shows two user-defined messages:

```
#define WM_MYFIRSTMESSAGE  (WM_USER + 400)
#define WM_MYSECONDMESSAGE (WM_USER + 401)
```

Declaring a message-structure type

See also Example

If you want to give useful names to the parameters of your message, you need to declare a message-structure type for that message. The message structure is the type of the parameter passed to the message-handling method. If you don't use the message's parameters, or if you want to use the old-style parameter notation (*wParam*, *lParam*, and so on), you can use the default message structure, *TMessage*.

To declare a message-structure type, follow these conventions:

1 Name the structure type after the message, preceded by a *T*.

2 Call the first data member in the structure *Msg*, of type *Cardinal*.

3 Define the next two bytes to correspond to the *Word* parameter and the next two bytes as unused

OR

Define the next four bytes as a *Longint* parameter.

4 Define the next four bytes as a *Longint* parameter.

5 Add a final data member called *Result*, of type *Longint*.

An example of a message structure

For example, here is the message structure for all mouse messages, *TWMKey*:

```
struct TWMKey
{
  Cardinal Msg;                    // first parameter is the message ID
  Word CharCode;                   // this is the first wParam
  Word Unused;
  Longint KeyData;                 // this is the lParam
  Longint Result;                  // this is the result data member
};
```

Declaring a new message-handling method

See also Example

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that isn't already handled by the standard components.
- You have defined your own message for use by your components.

To declare a message-handling method, do the following:

1 Declare the method in a **protected** part of the component's class declaration.
2 Be sure that the method returns void.
3 Name the method after the message it handles, but without any underline characters.
4 Pass a pointer called *Message* of the type of the message structure.
5 Map the method to the message using macros.
6 Within the message method implementation, write code for any handling specific to the component.
7 Call the inherited message handler.

An example of a message handler

Here's the declaration of a message handler for a user-defined message called *CM_CHANGECOLOR*:

```
#define CM_CHANGECOLOR (WM_USER + 400)
class TMyControl : public TControl
{
protected:
  void __fastcall CMChangeColor(TMessage* Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_CHANGECOLOR, TMessage, CMChangeColor)
END_MESSAGE_MAP(TControl)
};
void __fastcall TMyControl::CMChangeColor(TMessage* Message)
{
  Color = Message->LParam;                  // set color from long parameter
  TControl::CMChangeColor(Message);         // call the inherited message handler
}
```

Registering components

See also

Registering a component installs the component on the Component palette so you can select it, place it on a form, and manipulate it at design time.

Making your components available at design time requires several steps:

- Registering components
- Adding palette bitmaps
- Providing Help for your component
- Storing and loading properties

These steps don't apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only step that is always necessary is registration.

Registering components

See also

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you register them all at once.

To register a component, add a *Register* function to the .CPP file of the unit. Within the *Register* function you will register the components and determine where the components are installed on the Component palette.

Once you've set up the registration, you can install the components onto the Component palette as described in Chapter 2.

Writing the Register function

See also Example

Registration involves writing a single function in the .CPP file of the unit, which must have the name *Register*. Within the *Register* method, you call the function *RegisterComponents* for each component you want to register.

Note:

The *Register* function itself must exist within a namespace. The namespace is the name of the file the component is in with all lowercase letters except the first letter.

For example, this code exists within a Newcomp namespace, whereas Newcomp is the name of the .CPP file:

```
namespace Newcomp
{
  void __fastcall Register()
  {
  }
}
```

Inside the *Register* function, you must register each component you want to add to the Component palette. If the header and .CPP file combination contain several components, you can register them all in one step.

1 Within the *Register* function, declare an open array of type *TComponentClass* that holds the array of components you are registering. The syntax should look like this:

```
TComponentClass classes[1] = {__classid(TNewComponent));
```

In this case, the array of *classes* contains just one component, but you can add all the components you want to register to the array. For example, this code places two components in the array:

```
TComponentClass classes[2] = {__classid(TNewComponent), __classid(TAnotherComponent)};
```

Another way to add a component to the array is to assign the component to the array in a separate statement. This statement adds a third component to the *classes* array:

```
classes[2] = __classid(TOneMoreComponent);
```

Because the *classes* array already contains two classes, *TOneMoreComponent* is assigned to the third position in the array, with the first position having an index value of 0.

2 Within the *Register* function, call *RegisterComponents* to register the components in the *classes* array.

RegisterComponents is a function that takes three parameters: the name of a Component palette page, the array of component classes, and the size of the *classes* array minus 1.

An example of a Register function

The following *Register* function found in the NEWCOMP.CPP file, registers a component named *TMyComponent* and places it on a Component palette page called "Miscellaneous":

```
namespace Newcomp
{
  void __fastcall Register()
  {
      TComponentClass classes[1] = {__classid(TMyComponent)};
      RegisterComponents("Miscellaneous", classes, 0);
  }
}
```

Note that the third argument in the *RegisterComponents* call is 0, which is the size of the *classes* array minus 1.

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
namespace Mycomps
{
  void __fastcall Register()
  {
      // declares an array that holds two components
      TComponentClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
      // registers the two components in the classes1 array
      RegisterComponents("Miscellaneous", classes1, 1);
      // declares a second array
      TComponentClass classes2[1];
      // assigns a component to be the first element in the array
      classes2[0]  = __classid(TThird);
      // registers the component in the classes2 array
      RegisterComponents("Assorted", classes2, 0);
  }
}
```

In the example two arrays, *classes1* and *classes2* are declared. In the first *RegisterComponents* call the *classes1* array size is 2, so the third argument is the size minus 1, which is 1. In the second *RegisterComponents* call, the size of the *classes2* array is 1, so the third argument is 0.

Adding palette bitmaps

See also Example

Every component needs a bitmap to represent the component on the Component palette. If you don't specify your own bitmap, C++Builder uses a default bitmap.

Because the palette bitmaps are only needed at design time, you don't compile them into the component compilation unit. Instead, you supply them in a Windows resource file with the same name as the .H and .CPP files, but with the extension .DCR (for "dynamic component resource"). You can create this resource file using the Image editor in C++Builder. Each bitmap should be 24 pixels square.

For each component you want to install, supply a palette bitmap file, and within each palette bitmap file, supply a bitmap for each component you register. The bitmap image has the same name as the component. Keep the palette bitmap file in the same directory with the compiled files, so C++Builder can find the bitmaps when it installs the components on the Component palette.

An example of adding a palette bitmap

For example, if you create a component named *TMyControl*, you need to create a .DCR or .RES resource file that contains a bitmap called TMYCONTROL. The resource names are not case-sensitive, but by convention, they are usually in uppercase letters.

Providing Help for your component

See also

When you select a component on a form, or a property or event in the Object Inspector, you can press F1 to get Help on that item. Users of your components can get the same kind of documentation for your components if you create the appropriate help files.

You can provide a small Help file with just the information on your components, and users will be able to find your documentation without having to take any special steps. Your help file becomes part of the user's overall C++Builder Help system.

Creating the Help file

See also

You can use any tools you want to create a Windows Help file. C++Builder includes the Microsoft Help Workshop, which compiles your Help files and presents an online help authoring guide. You can find complete information about creating Help files in the online guide.

To make your component's Help work with the Help for the rest of the components in the library, observe the following conventions:

1 Each component should have a screen.

   The component screen should show which unit it's declared in, give a brief description of the component's purpose, then list separately all the properties, events, and methods available to component users. Application developers access this screen by selecting the component on a form and pressing F1. For an example of a component screen, place any component on a form and press F1.

   The component screen should have a "K" footnote for keyword searching that includes the name of the component. For example, the keyword footnote for the *TMemo* component reads "TMemo."

2 Each property, event, and method that is declared within the component should have a screen.

   A property, event, or method screen should show the declaration of the item, and describe its use. Application developers see these screens either by highlighting the item in the Object Inspector and pressing F1 or by placing the test cursor in the Code editor on the name of the item and pressing F1. To see an example of a property screen, select any item in the Object Inspector and press F1.

Each component, property, event, or method screen should have

- A topic ID that is unique to the topic entered as a "#" footnote.
- A title entered as a "$" footnote.

  The title appears as specified in the Topics Found dialog box, the Bookmark dialog box, and the History window.
- A "K" footnote for keyword searching that includes the name of the item.

  For example, the keyword footnote for the *Top* property reads "Top". For *TMemo*, the keyword reads "TMemo".

Providing context sensitivity for your component

See also

Each component, property, and event screen must have an"A" footnote. The "A" footnote is used to display the screen when the user selects a component and presses F1, or when a property or event is selected in the Object Inspector and the user presses F1. The "A" footnotes must follow certain naming conventions:

If the Help screen is for a component, the "A" footnote consists of two entries separated by a semicolon using this syntax:

ComponentName_**Object**;*ComponentName*

ComponentName is the name of the component.

For example, for a component named *MyComponent*, the "A" footnote is

MyComponent_Object;MyComponent

If the Help screen is for a property or event, the "A" footnote consists of three entries separated by semicolons using this syntax:

ComponentName_Element;*Element_Type*;*Element*

ComponentName is the name of the component, *Element* is the name of the property or event, and *Type* is the either Property or Event

For example, for a property named *BackgroundColor* of a component named *MyGrid*, the "A" footnote is

MyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor

Adding Help files to C++Builder Help

See also

To add your Help file to the C++Builder Help file,

Use the OpenHelp utility.

Search for ..Borland\Common files\OpenHelp.exe.

You'll find information in the OpenHelp.hlp file about using OpenHelp, including adding your Help file to the Help system.

Storing and loading properties

See also

C++Builder stores forms and their components in form (.DFM) files. A form file is a binary representation of the properties of a form and its components. When C++Builder users add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into C++Builder or executed as part of an application, the components must restore themselves from the form file.

Most of the time you won't need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading

Using the store-and-load mechanism

See also

When an application developer designs forms, C++Builder saves descriptions of the forms in a form (.DFM) file, which it later attaches to the compiled application. When a user runs the application, those descriptions are read in.

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its public and published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

Specifying default values

See also Example

C++Builder components only save their property values if those values differ from the default values. If you don't specify otherwise, C++Builder assumes a property has no default value, meaning the component always stores the property, whatever its value.

A property whose value is not set by a component's constructor assumes a zero value. A zero value means whatever value the property assumes when its storage memory is set to zero. That is, numeric values default to zero, Boolean values to false, pointers to NULL, and so on. If there is any doubt, specify the default value explicitly.

To specify a default value for a property,

1 Add an equal sign (=) after the property name.

2 After the equal sign, add braces({}).

3 Within the braces, type the keyword **default**, followed by another equal sign.

4 Specify the new default value.

For example,

```
__property Alignment = {default=taCenter};
```

You can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

Note:

Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value.

An example of specifying a default value

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```
class TMyStatusBar : public TPanel
{
public:
  virtual __fastcall TMyStatusBar(TComponent* AOwner);
__published:
  __property Align = {default=alBottom};
};
```

The constructor of the *TMyStatusBar* component is in the .CPP file:

```
__fastcall TMyStatusBar::TMyStatusBar (TComponent* AOwner)
  : TPanel(AOwner)
{
   Align = alBottom;
}
```

Determining what to store

See also Example

You can control whether C++Builder stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose to not store a given property at all, or you can designate a function that determines at runtime whether to store the property.

To control whether C++Builder stores a property,

1 Add an equal sign (=) after the property name.

2 After the equal sign, add braces({}).

3 Within the braces, type the stored specifier, followed by true, false, or the name of a Boolean method.

An example of stored properties

The following code shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean method:

```
class TSampleComponent : public TComponent
{
protected:
  bool __fastcall StoreIt();
public:
  __property int Important = {stored=true};         // always stored
   ...
__published:
  __property int Unimportant = {stored=false};      // never stored
  __property int Sometimes = {stored=StoreIt};      // storage depends on function value
};
```

Initializing after loading

See also

After a component reads all its property values from its stored description, it calls a virtual method called *Loaded*, which provides a chance to perform any initializations that might be required. The call to *Loaded* occurs before the form and its controls are shown, so you don't need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

Note:

The first thing you do in any *Loaded* method you write is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you perform initializations on your own component.