
Creating and editing forms

In brief, to create a C++Builder application you do the following:

- 1 Start with a form.
- 2 Put components on the form.
- 3 Set the properties of the components.
- 4 Create an empty event handler.
- 5 Put code in to make the component do something.
- 6 Save the project.

In this chapter you will learn the following:

- What are forms and components
- How to use forms and components to create C++Builder applications
- How to customize forms and set properties
- How to save your project and manage runtime behaviors

Designing the user interface

A C++Builder application usually contains multiple forms: a main form, which is the primary user interface, and other forms such as dialog boxes, secondary windows, and so on.

You develop your application by customizing the main form and adding and customizing other forms. You customize forms by adding components, setting their properties, and creating menus to provide user control over the application at runtime. You can also add components such as the menu component or popup menu component to create a main menu and right click menus for your application.

What are forms?

Forms are specialized objects that contain components. Forms generally appear as windows and dialog boxes in a running application. Additional specialized objects called data modules can be placed on

forms but they are invisible when the application runs, since they hold nonvisual components, such as databases, which the application may not interact with directly.

You design the application user interface using forms, so forms are very important. But it's also important to understand that a form is just another component. So as with other components, you interact with a form by reading and setting its properties, calling its methods, and responding to its events.

Associated with each form is a .CPP and .H file pair, referred to collectively as the form unit.

What are components?

Components in C++Builder are the building blocks of an application. Each component represents some single application element, such as a user interface object, a database, or a system function. By choosing and connecting these elements, you build the interface of your application. You'll learn more about components later in this chapter. Components may also have an event handler which allows it to receive and respond to specific types of input. For example, a button has an OnButtonClick event handler which instructs the button how to respond to a click, right click, or double click.

Creating forms

C++Builder provides several ways to create new forms, reuse existing forms, and customize existing forms.

You can create a new form in many ways:

- Start C++Builder.

C++Builder generates a blank form, its associated unit, and a project file.

Note:

If you chose a different form or project as the default from the Object Repository dialog box, the default form opens, instead of a blank form.

- Create a new application by choosing File|New Application.

C++Builder generates a new form and unit file whenever it generates a new application.

- Add a new form to an existing project by choosing File|New Form.

Sharing forms

Before you begin designing and building the forms for your applications, think about whether you want these forms to be available for other developers to use. C++Builder is designed with the principle of reusable components in mind, and this encompasses larger elements such as forms or even entire projects.

The easiest way to share a form is to add an existing form to a project. You can easily use any of the predesigned forms in the Object Repository in your applications. You can also save any form you've designed in the Object Repository.

Adding a form to a project

When you start C++Builder for the first time, it opens with an empty project consisting of a single, blank form that contains no controls or other components. You can use that as a starting point, or you can choose File|New to start a new project, and choose a different form.

When you choose File|New, C++Builder displays the New Items dialog box, as shown in Figure 2.1.

Figure 6.1 The New Items dialog box

The New Items dialog box shows what is in the Object Repository. The Object Repository contains forms, projects, data modules, and experts you can either use directly, copy into your projects, or derive new items from.

Adding an existing form

You can use any of the forms from the Object Repository in your application.

To add a form from the Object Repository to your project,

- 1 With a project open, choose File|New.

The New Items dialog box appears.

- 2 Choose the Forms page.

The Forms page opens with the default new form highlighted.

Note

Depending on which templates have been installed, modified, or deleted in your C++Builder installation, your Object Repository window for forms might look different from the one shown in Figure 2.2.

Figure 6.2 Standard C++Builder forms in the Object Repository

- 3 Select the form you want to add.
- 4 Choose one of the sharing options: Copy, Inherit, Use.

Copy creates an exact copy of the form and places it in your project. Changes made to the template form in the Object Repository will not affect the form in your project. Note that Copy is the only option available for Form Experts (such as the Database Form Expert), because running the Expert generates a new form for you.

Inherit derives a new form object from the one in the Object Repository and adds it to your project. Changes to the template form in the Object Repository will show up in your form, unless you have already modified that part of the new form.

Use means you want to use and modify the template form. Any changes you make to the form in your project are reflected back into the template in the Object Repository.

5 Choose OK.

C++Builder adds the form and its associated unit to the project you have open. You can now use this form the way you would any form in a project.

Modifications you make might affect the original item in the Object Repository, depending on the sharing option you chose.

Note

With no project open, it is still possible to choose File|New and select a form from the Object Repository. The form's unit file then opens as a reference file. If you subsequently open a project or create a new project, the open template form *is not part of that project*. To save it as part of the project, you must explicitly add it to the project. forms:adding to projectsforms:adding to projectstemplates:adding to projectstemplates:adding to projectspredesigned forms:adding to projectspredesigned forms:adding to projects

Creating form templates

You can create form templates for others to use by saving a form or an object in the Object Repository.

To add your current form to the Object Repository,

1 Right-click the form and choose the Add To Repository command.

The Add To Repository dialog box appears.

2 In the Title edit box, specify a name for the form.

3 In the Description edit box, type a brief description of this form.

4 Choose the Page on which the form should appear in the New Items dialog box.

5 You can specify an Author of the form, which shows only in the detailed view of the Object Repository.

6 To specify an icon for the object, choose the Browse button.

The Select Icon dialog box appears.

7 Locate and select the icon (if any) you want to use, and choose OK to exit the Select Icon dialog box.

8 Choose OK to accept your specifications, and exit the Add To Repository dialog box.

The next time you choose File|New and click the Forms page tab, your form appears in the templates list, with the icon and title you chose. forms:sharingforms:sharingsharing formssharing formsforms:saving:as templatesforms:saving:as templatestemplates:saving forms astemplates:saving forms assaving:forms:as template saving:forms:as templatesObject RepositoryObject Repository

Inheriting from forms in the project

If you create an application that has several similar forms, you can create one version of the form, then create the others by inheriting. This allows you to change the standard form and have those changes reflected in all the inherited forms.

When browsing items in the Object Repository, you'll find the option to copy, inherit, or use at the bottom of the dialog box.

- When you inherit from the template form, you create a reference to the ancestor form, and only have additional code for added components and event handlers. If you inherit several forms in the same project, they share the inherited code. All the ancestors of the chosen form are also added to the project.

Inheriting forms is a good way to reduce the size of projects that use a number of similar forms. It also provides a way to create and maintain a set of standard form templates that work in a number of projects, even when each project requires customization.

- Inherit is not enabled when a Form expert is selected, since these cannot be ancestor forms. Choosing Copy on a non-inherited form works the same as before: creating a new form that is a copy of the form chosen.

Linking forms

Adding a form or component to a project adds a reference to it in the project file, but not to other parts of the project. You need to add a reference to it in the other files that need access to it. This is called form linking.

The most common kind of component that uses such links are data-access components. For example, you can have a table component on one form in an application and allow several different forms to provide different views into the same data set, such as a grid view and a form view.

C++Builder links forms by linking their associated units. Given two forms, Form1 and Form2, and their associated units, Unit1 and Unit2, respectively, components on Form1 can refer to components on Form2 if Unit1 contains Unit2 in one of its `#include` directives.

To link one form to another,

- 1 Select the form that needs to refer to the other.
- 2 Choose File|Include Unit.
- 3 Select the name of the form unit for the form to be referenced.
- 4 Choose OK.

Linking a form to another just means that the `#include` directives of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

Viewing forms and units

To switch among forms in a project,

- 1 Choose View|Forms.

The View Form dialog box appears.

- 2 Select the form you want to view, then choose OK.

To switch among units in a project,

- 1 Choose View|Units.

The View Unit dialog box appears.

- 2 Select the unit you want to view, then choose OK.

You can also use the Project Manager to navigate to different units and forms in your project.

About components

Components are objects in the true object-oriented programming (OOP) sense. Because they are objects, C++Builder components

- encapsulate some set of data and data-access functions
- inherit data and behavior from the objects they are derived from (called their ancestors)
- operate interchangeably with other objects derived from a common ancestor, through a concept called polymorphism

Each component therefore encapsulates some element of an application, such as a window or dialog box, a field in a database, or a system timer. Although each has its own unique aspects, all components share certain qualities they inherit from a common ancestor object called TComponent. TComponent defines the minimum set of attributes necessary for a component to operate in the C++Builder environment.

Components contain three kinds of information:

- State information. Information about the present condition of a component is called a property of the component. Properties are named attributes of a component that a user or application can read or set. Examples of component properties are Height and Color.
- Action information. Components generally have certain actions they can perform on demand. These actions are defined by methods, which are functions you call in code to tell the component what to do. Examples of component methods are Hide and Refresh.
- Feedback information. Components provide opportunities for application developers to attach code to certain occurrences, or events, making components fully interactive. By responding to events, you bring your application to life. Examples of component events are OnClick and OnKeyDown.

C++Builder's standard components

The components on the palette are grouped into pages according to similar functions. You can add, remove, and rearrange components as you choose. Table 2.1 lists the default pages and the kinds of components they contain.

Table 6.1 The standard C++Builder Component palette pages

Page name	Contents
Standard	Standard Windows controls, menus
Additional	Customized controls
Win95	Windows 95 common controls
Data Access	Non-visual components that access databases, tables, queries, and reports
Data Controls	Visual, data-aware controls
Win 3.1	Windows 3.1 style controls
Dialogs	Windows common dialog boxes
System	Components and controls for system-level access, including timers, file system, multimedia, and DDE
QReportt	Quick Reports components for easily creating embedded reports
OCX	Sample OLE controls
Samples	Sample custom components, including gauge, color grid, spin buttons, directory outline, and calendar grid
Internet	OCX controls for Internet programming

To learn more about any component, select the component and press F1.

Understanding components

Components are the building blocks of C++Builder applications. They encapsulate elements of applications in a standardized, reusable way. Understanding this component model is the most important step in understanding C++Builder.

There are a number of ways to categorize the components that come with C++Builder. Each provides a different set of information that can help you use the components more effectively.

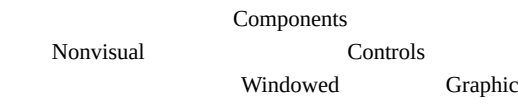
This section examines two such views of C++Builder's components: a hierarchical view and a functional view. The hierarchical view divides components according to what they are, where the functional view divides components according to what you use them for.

The component hierarchy

One way to understand components is to look at the inheritance relationships that different components share. By noting the common properties, methods, and events that various groups of components inherit, you can understand some of their similarities and differences. For details on the Visual Component Library (VCL) hierarchy, see the Component Writer's Guide and the Library Reference.

Figure 2.4 shows a simplified view of the major kinds of components.

Figure 6.4 A simple component hierarchy



In general, there are two broad categories of components: nonvisual components and controls.

- Nonvisual components are components that represent program elements that the user cannot interact with directly, such as system timers and database connections. Nonvisual components appear at design time as small icons on a form or data module, which enables you to select them for setting properties and attaching event handlers.

Some nonvisual components actually represent visual elements when the application runs, such as a main menu bar or a Windows common dialog box. They store a description of items the application will create at the appropriate time when running.

- Controls are visible elements the user can interact with at runtime. In general, they look the same at design time as they will at runtime, which facilitates form layout. Whenever possible, C++Builder controls look and act “live” at design time: a list box displays its list of items, a data grid connected to an active data set displays actual data.

There are two subcategories of controls: windowed and graphic controls.

Windowed controls are controls that can get input focus. Most of the standard Windows controls are windowed controls. The term “windowed” comes from the fact that such controls have a window handle (which you can access through a Handle property).

Graphic controls (sometimes called “nonwindowed controls”) are controls that cannot receive focus, and do not have window handles. Graphic controls consume fewer system resources, so they are useful for complex forms that need to display numerous controls.

The distinction between windowed and graphic controls is important when you design your user interfaces. For example, if you create a tool bar that contains a large number of buttons, you could use standard Windows button component, but each one would consume a window handle. You could instead choose a nonwindowed speed-button component that would greatly reduce the drain on system resources.

What components do

Another way to understand components is to look at what they do. In many cases, several components can all perform the desired action, but you might choose one because it has greater capacity, better performance, or lesser impact on system resources.

The following table lists categories of common user interface tasks and suggests the components you would consider for those tasks. To learn more about a specific component, select it and press F1.

Table 6.2 Common component task categories	
Task category	Components
Text input and manipulation	edit box, memo, mask edit, rich edit, DBText, DBEdit, DBMemo
Specialized input	scroll bar, track bar, up-down, hot key
Button input,	button, check box, radio button, bitmap button,

choosing options	speed button, DB check box, DB radio button, tab set
List display and manipulation	list box, combo box, tree view, list view, DB list box, DB check box, DB lookup list box, DB lookup combo box
Grouping components	group box, radio group, panel, scroll box, tab control, page control, header control, tabbed notebook, notebook
Visual feedback	label, progress bar, status bar, panel
Tabular display	string grid, draw grid, DB grid, DB control grid
Graphic display	image, shape, bevel, image list, paint box
Windows common dialog boxes	open dialog, save dialog, font dialog, color dialog, print dialog, printer setup dialog, search dialog, replace dialog

Common component elements

All components have properties, methods, and events built into them. Some of these they inherit from their ancestor component types, which means they share these elements with other components. Such elements are called common elements. For example, all controls inherit a property called Height that represents the vertical size of the control. Height is therefore a common property of all controls.

Naming components

Several properties are common to all components, but the most important is the Name property. Name is different from other properties in several ways. Every component in an application must have a unique name, so assigning meaningful names at the outset makes your code more readable, and prevents possible name conflicts later. This differs from the Caption property which is used to display information to the user and has no other meaning to the underlying program.

Component names must follow the standard rules for naming C++ identifiers. If you enter a value that is not consistent with C++ naming requirements, the name reverts to its previous value, and C++Builder displays an error message.

C++Builder assigns default names to forms and components which may not make your code easy to read. It is good practice to change the Name property so that it is descriptive of the component's function.

Caution

As long as you use the Object Inspector to change a component's *Name* property, C++Builder maintains your changes in the underlying code. This is not the case, however, if you edit the component name yourself by making the change directly in the Code editor. If you manually edit a component name, C++Builder will be unable to load your form.

Changing the name of a component

To change the name of a component,

- 1 Select the component.

2 In the Object Inspector, select the Name property and enter a new name.

Component names must follow the standard rules for naming C++ identifiers.

Note:

Changing the Name property also changes the Caption property, unless you have already changed the Caption property. The value of the Caption property has precedence over the Name property.

Manipulating components in forms

You can select, cut, copy, paste, move, delete, and restore components the same way you do in other Windows applications. Some skills may be new to you including

- Adding components to a form
- Grouping components
- Aligning components

Note

Keyboard support is available for many of the tasks documented here. For more information about keyboard techniques, see the online Help topic Form Keyboard Shortcuts.

Adding components to a form

To add a component to the center of a form, double-click the component in the Component palette.

If the form already has components, additional components are offset (lower and to the right) as you add them, so all the components remain at least partially visible.

To add a component to a specific location in the form,

- 1 Click the component on the Component palette.
- 2 Move the cursor to where you want the upper left corner of the component to appear in the form, then click the form.

The component appears in its default size, in the position you clicked on the form.

When you add a component to a form, C++Builder generates an instance variable, or field, for the component and adds it to the form's type declaration. C++Builder is a two-way tool, so adding a component changes the form's type declaration:

```
class TAboutBox : public TForm
{
__published:
    TButton *Button1;
    TButton *Button2;
    void __fastcall Button1Click(TObject *Sender);
private:    //private user declarations
```

```
public:    //public user declarations
    virtual __fastcall TAboutBox(TComponent* Owner);
};
```

Similarly, when you delete a component, C++Builder deletes the corresponding type declaration.

To add multiple copies of the same component,

- 1 Press and hold the Shift key.
- 2 Click the component on the palette, then click the form once for each copy you want of the component. (You don't need to hold down the Shift key after the component is selected.)

Clicking in the form continues to add the component to the form, as long as the component remains selected in the palette.

- 3 Click the pointer icon to clear the selected component.

Grouping components

C++Builder provides several components—the group box, panel, notebook, page control, and scroll box—that can contain other components. These are called *container components*. You can use these container components to group other components so that they behave as a unit at design time. You often use container components such as the panel component to create customized tool bars, backdrops, status bars, and so on.

For more information about creating tool bars and status bars, search online Help under “tool bars” or “status bars.”

When you place components within container components, you create a new *parent-child* relationship between the container and the components it contains. Design-time operations you perform on these “container” (or parent) components, such as moving, copying, or deleting, also affect any components grouped within them.

Note

The form remains the *owner* for all components, regardless of whether they are *parented* within another component. For more information about the difference between a component's parent and owner, see online Help topics Owner Property and Parent Property.

Once a component is on the form, you can add it to a container component by cutting and then pasting it.

To group components,

- 1 Add a group box or panel component to a form.
- 2 Making sure that the container component is selected, add components as you normally would.

As you add components, they appear inside the container component.

To add multiple copies of a component to a container,

- 1 Press Shift and then select a component from the palette.
- 2 Click anywhere in the container component.

Each subsequent click continues to place the component in whatever eligible receiving component (including the form) is clicked.

- 3 Select the pointer icon when you finish adding components.

Aligning components

Once you select the components you want to align, you can use the Alignment palette or the Alignment dialog box to set the alignment. When aligning a group of components, the first component you select is used as a guide to which the other components are aligned.

To align components using the Alignment palette,

- 1 Select the components.
- 2 Choose View|Alignment Palette.

The Alignment palette appears.

- 3 Select an alignment icon from the palette.

To align components using the Alignment dialog box,

- 1 Select the components.
 - 2 Choose Edit|Align.
- The Alignment dialog box appears.
- 3 Select the alignment options you want in the dialog box.
 - 4 Choose OK to put your alignment options into effect.

You can continue to choose or modify alignment options as long as the components remain selected.

For more information about the Alignment palette and the Alignment dialog box, see online Help.

Using the grid to align components

The evenly spaced dots that appear in the form at design time are the form *grid*. The grid makes it easier to align components visually.

Figure 6.7 Form grid

By default, both the grid and its Snap To Grid option, which causes the left and top sides of each component to always align with the nearest grid markings, are enabled at design time. You can, however, choose to disable the Snap To Grid option, or disable the grid altogether.

You can also modify the granularity of the grid—that is, how far apart the grid dots appear.

To set form grid options,

- 1 Choose Tools|Options to display the Environment Options dialog box.
- 2 Click the Preferences tab to display the Preference page if it is not already visible.
- 3 In the Form Designer options, check the Display Grid to view the grid, or uncheck it to hide it.

To enable the Snap To Grid option,

- 1 Choose Tools|Options to display the Environment Options dialog box.
- 2 In the Form Designer options, check the Snap To Grid to enable the option, or uncheck it to disable it.

To modify the granularity of the grid,

- 1 Choose Tools|Options to display the Environment Options dialog box.
- 2 In the Form Designer options, enter new values in the Grid Size boxes.

Grid Size X controls the horizontal spacing, and Grid Size Y controls the vertical spacing. The default values are 8 and 8. To space the dots further apart, choose larger numbers. To create a finer grid, choose smaller numbers.

Locking the position of components

Once you have aligned the components on a form, you can prevent components from being moved accidentally.

To lock the position of the components on a form, choose Edit|Lock Controls from the menu bar.

Building dialog boxes

C++Builder provides a number of predesigned dialog boxes as components for your user interface. They appear on the Dialogs page of the Component palette. You can specify different options for these dialog boxes, such as whether a Help button appears in the dialog box, and then add any custom changes your dialog requires. You can also develop custom dialog boxes.

Developing custom dialog boxes

This section discusses the most common considerations when designing any dialog box, including

- Making a dialog box modal or modeless
- Setting form properties for a dialog box
- Creating standard-command buttons
- Setting the tab order
- Testing the tab order
- Removing a component from the tab order
- Enabling and disabling components
- Setting the focus in a dialog box

Making a dialog box modal or modeless

At design time, dialog boxes are simply customized forms. At runtime, they can be either *modal* or *modeless*. When a form runs modally, the user must explicitly close it before working in another running form. Most dialog boxes are modal.

When a running form is modeless, it can remain onscreen while the user works in another form (for example, the application main form). You might create a modeless form to display status information, such as the number of records searched during a query, or information the user might want to refer to while working.

Note

If you want a modeless dialog box to remain on top of other open windows at runtime, set its *FormStyle* property to *fsStayOnTop*.

Forms have two methods that govern their runtime modality. To display a form in a modeless state, you call its *Show* method; to display a form modally, you call its *ShowModal* method.
 dialog boxes:modal
 dialog boxes:modalmodal
 dialog boxesmodal
 dialog boxesdialog
 boxes:modeless
 dialog boxes:modelessmodeless
 dialog boxesmodeless
 dialog boxes

Setting form properties for a dialog box

By default, C++Builder forms have Maximize and Minimize buttons, a resizable border, and a Control menu that provides additional commands to resize the form. While these features are useful at runtime for modeless forms, modal dialog boxes seldom need them.

C++Builder provides a *BorderStyle* property for the form that includes several useful values. Setting the form's *BorderStyle* to *bsDialog* implements the most common settings for a modal dialog box:

- Removing the Minimize and Maximize buttons
- Providing a Control menu with only the Move and Close options
- Making the form border non-resizable, and giving it a “beveled” appearance

The following table shows other form property settings that can be used, individually or in concert, to create different form styles.

Table 6.3 Alternate form settings

Property	Setting	Effect
XE "BorderIcons property" BorderIcons		
XE "biSystemMenu constant"XE "Control menu:removing from forms"XE	False	Removes Control (System) menu

"System menu:removing from forms" <i>biSystem</i> Menu		
XE "biMinimize constant"XE "Minimize button:removing from forms" <i>biMinimi</i> ze	False	Removes Minimize button
XE "biMazimize constant"XE "Maximize button:removing from forms" <i>biMazim</i> ize	False	Removes Maximize button
BorderStyle		
XE "BorderStyle property:forms"X E "bsSizeable constant"XE "forms:borders:re sizable"XE "resizable borders"	bsSizeable	Lets the user resize the form border
XE "bsSingle constant"XE "non-resizeable borders"	bsSingle	Provides a single, non-resizable border
XE "bsNone constant"	bsNone	No distinguishable border; not resizable
XE "bsDialog constant"	bsDialog	Window has a border, but not resizable
XE "bsToolWindow constant"	bsToolWindow	Makes the title bar small; window is not resizable
XE "bsSizeToolWind ow constant"	bsSizeToolWindow	Makes the title bar small; window is resizable

Note

Changing these settings doesn’t change the design-time appearance of the form; these property settings become visible at runtime.

Specifying a caption for a dialog box

By default, C++Builder displays the Name property value for each form in the form's Title bar. If you change the Name property of the form prior to changing the Caption property, the Title bar caption

changes to the new name. Once you change the Caption property, the form's title bar always reflects the current value of Caption.

Creating standard-command buttons

You can quickly create buttons for many standard commands by adding BitBtn components to the form and setting the *Kind* property for each button. The possible *Kind* property settings and their effect are shown in the following table. (In addition to the property settings shown, the bitmap button displays graphics, such as a green check mark for the OK button, or a red X for the Cancel button.)

Table 6.4 C++Builder’s predefined bitmap button types				
Kind value	Effect	Appearance	Equivalent property setting(s)	Comments
XE "bkAbort constant"XE "Cancel buttons"bkA bort	Makes a Cancel button with Abort as caption		Caption := ‘Abort’ ModalResult := mrAbort	Red X appears next to caption.
XE "bkAll constant"XE "OK buttons"bkA ll	Creates an OK button (with <u>A</u> ll caption)		Caption := ‘&All’ ModalResult := 8	Green double check mark appears next to caption.
XE "bkCancel constant"bk Cancel	Makes a Cancel button		Caption := ‘Cancel’ Cancel := True ModalResult := mrCancel	Red X appears next to caption.
XE "bkClose constant"XE "Close buttons"XE "lavender “exit” doors"bkClo se	Creates a Close button; closes the window		Caption := ‘&Close’	A lavender “exit” door appears as the glyph for this button.
XE "bkCustom constant"XE "buttons:cus tomizing"X E "bitmap buttons:cust om"bkCusto m	None	N/A	N/A	Use this setting to create a custom command button, including specifying a custom <i>Glyph</i> bitmap.
XE "bkHelp constant"XE "Help buttons"bkH elp	Creates a button with Help as the caption		Caption := ‘&Help’	A blue ? appears next to the caption. Use the event handler of this button to call your program Help file. (If the dialog box has a Help context, C++Builder does this automatically.)
XE	Creates a button		Caption := ‘&Ignore’	Use to continue an

"bkIgnore constant"XE "Ignore buttons"bkIg nore	to ignore changes and proceed with specified action	ModalResult := mrIgnore	operation after an error condition has occurred.
XE "bkNo constant"bk No	Makes a Cancel button (with No as the caption)	Caption := '&No' Cancel := True ModalResult := mrNo	Red circle with slash appears next to caption.
XE "bkOK constant"bk OK	Creates an OK button	Caption := 'OK' Default := True ModalResult := mrOK	Green check mark appears next to caption.
XE "bkRetry constant"XE "Retry buttons"bkR etry	Creates a button to retry specified action	Caption := '&Retry' ModalResult := mrRetry	Green circular arrow appears next to the caption.
XE "bkYes constant"XE "OK buttons"XE "buttons:add ing;to dialog boxes"XE "adding;butt ons;to dialog boxes"XE "dialog boxes:addin g buttons"XE "bitmap buttons:addi ng to dialog boxes"bkYes	Creates an OK button (with <u>Y</u> es caption)	Caption := '&Yes' Default := True ModalResult := mrYes	Green check mark appears next to caption. buttons:addin g;to dialog boxesbuttons:addin g;to dialog boxesadding;button s;to dialog boxesadding;button s;to dialog boxesdialog boxes:adding buttonsdialog boxes:adding buttonsbitmap buttons:adding to dialog boxesbitmap buttons:adding to dialog boxes

Note that setting the *Kind* property also sets the *ModalResult* property, discussed previously, in every case except *bkCustom*, *bkHelp*, and *bkClose*. In these cases, *ModalResult* remains *mrNone*, and choosing the button doesn't automatically close the dialog box.

Executing button code on Esc

C++Builder provides a *Cancel* property for Button components. When your form contains a button whose *Cancel* property is set to *True*, pressing the Esc key at runtime executes any code contained in the button's *OnClick* event handler.

To designate a button as the Cancel button, set its *Cancel* property to *True*.

- To specify that the modal dialog box close when the user chooses a Cancel button, set the button's *ModalResult* property to *mrCancel*.

Setting a button's *ModalResult* property to a nonzero value means the modal dialog box closes automatically when the user chooses the button.

You can also use the *BitBtn* component to create a Cancel button.

To use the bitmap button to create a Cancel button,

- Add a *BitBtn* component to your form, and set its *Kind* property to *bkCancel*. This sets the button's *Cancel* property to *True*, and the *ModalResult* property to *mrCancel*.

Executing button code on Enter

When your form contains a button whose *Default* property is set to *True*, pressing Enter at runtime executes any code contained in the button's *OnClick* event handler—unless another button has focus when the Enter key is pressed.

Even if your form contains a default button, another button can take focus away at runtime. Pressing the Enter key calls the *OnClick* event handler code of the button with focus, overriding any other button's *Default* property setting. (The button with focus is indicated by a darker, thicker border than that of other buttons in the dialog box.)

For example, in the File|Open dialog box, the Open button is the default button. If you select a file name and press Enter, the file code attached to the Open button will execute. If you tab to the Cancel button and press Enter, the code attached to that button will execute.

Note

Although other components in a form can have focus, usually button components respond when the user presses Enter. The default button takes the *OnClick* event when another non-button component in the form has focus.

To specify a button as the default button, set its *Default* property to *True*.

To change focus at runtime, call the button's *SetFocus* method.

To specify that the modal dialog box close when the user chooses a default button, set the button's *ModalResult* property to *mrOK*.

Setting a button's *ModalResult* property to a nonzero value means the modal dialog box closes automatically when the user chooses the button.

You can also use the *BitBtn* component to create a Default button.

To use the bitmap button to create a default button,

Add a *BitBtn* component to your form, and set its *Kind* property to *bkOK*. This automatically sets the button's *Default* property to *True* and the *ModalResult* property to *mrOK*.

Closing a dialog box when the user chooses a button

You can specify that a modal dialog box closes automatically when the user chooses any button whose *ModalResult* property has a nonzero value. By setting *ModalResult* to match a button's caption, you can also determine which button the user chose. For example, if you have a Cancel button, set its *ModalResult* property to *mrCancel*; if your form contains an OK button, set its *ModalResult* to *mrOK*. Both buttons will close the form when chosen, because *ModalResult* returns a nonzero value to the *ShowModal* function. However, because *ModalResult* returns *mrCancel* in one case, and *mrOK* in the other, your code can determine which button was pressed and branch accordingly.

To automatically close the dialog box when the user chooses a Cancel button or presses Esc, set the Cancel button's *ModalResult* property to *mrCancel*.

To automatically close the dialog box when the user presses Enter when an OK button has focus, set the button's *ModalResult* property to *mrOK*.

Setting the tab order

Tab order is the order in which focus moves from component to component on a form that is displayed in a running application when the user presses the Tab key. To let the Tab key shift focus to a component on a form, the *TabStop* property of the component must be set to *True*.

The tab order is initially set by C++Builder, corresponding to the order in which you add components to the form. You can change this by using the Edit Tab Order dialog box, or by changing the *TabOrder* property of each component.

To change the tab order using the Edit Tab Order dialog box,

- 1 Select the form, or a container component in the form, that contains the components whose tab order you want to set.
- 2 Choose Edit|Tab Order, or right click and choose Tab Order.

The Edit Tab Order dialog box appears, displaying a list of components ordered (first to last) in their current Tab order.

- 3 In the Controls list, select a component, and press the appropriate arrow button (Up or Down), or drag the component to its new location in the tab order list.
- 4 When the components are ordered the way you want, choose OK.

Using the Edit Tab Order dialog box changes the value of the components' *TabOrder* property. You can also do this manually, if you want.

Keep in mind the following points when manually setting your tab order (you needn't be concerned with these points if using the Edit Tab Order dialog box):

- Each *TabOrder* property value must be unique. If you assign two components the same *TabOrder* value, C++Builder rennumbers the *TabOrder* value for all other components accordingly.

- If you attempt to give a component a *TabOrder* value equal to or greater than the number of components on the form (because numbering starts with 0), C++Builder changes the value, so it is last in the tab order.
- Components that are invisible or disabled are not recognized in the tab order, even if they have a valid *TabOrder* value. When the user presses Tab, the focus skips over such components and goes to the next one in the tab order.

To change tab order using the component's *TabOrder* property,

- 1 Select the component whose position in the tab order you want to change.
- 2 In the Object Inspector, select the *TabOrder* property.
- 3 Change the *TabOrder* property's integer value to the value you want the component to have in the tab order, starting with the number zero.

Testing the tab order

You can test the tab order you've set by running the application. At design time, focus always moves from component to component in the order that the components were placed on the form. Changes you make to the tab order are reflected only at runtime.

To test the tab order,

- 1 Compile and run the application.
- 2 Display the form whose tab order you want to test.
- 3 Use the Tab key to view the order in which focus moves from component to component.

Removing a component from the tab order

In some cases, you might want to prevent users from being able to tab to a component on a form, and have them skip to the next one instead.

To remove a component from the tab order,

- 1 Select the component.
- 2 Use the Object Inspector to set the value of the *TabStop* property to False.

Note

Removing a component from the tab order doesn't disable the component. dialog boxes:tab orderdialog boxes:tab ordertab ordertab ordercomponents:setting tab ordercomponents:setting tab ordercomponents:removing from tab ordercomponents:removing from tab ordertabbing through dialog boxestabbing through dialog boxes

Disabling components

When a component is disabled, it appears dimmed in the running application, and the user cannot tab to it, even if its *TabStop* property is set to *True*.

By disabling a component at design time, that component will be initially unavailable to the user when the dialog box first opens. You can also dynamically change whether a component is enabled at runtime.

To disable a component at design time, use the Object Inspector to set the component's *Enabled* property to *False*.

To disable a component at runtime, type the following code in an event handler for the component:

```
<componentn>->Enabled = false;
```

For example, the following event handler specifies that when *Button1* receives an *OnClick* event, *Button2* is disabled.

```
void __fastcall TAboutBox::Button1Click(TObject* Sender)
{
    Button2->Enabled = false;
}
```

Setting component properties

You can set component properties at design time or change values while an application is running. For more information about the properties for each component, search online Help for the keyword components, or see the topic Visual Component Library components, and select the component whose properties you want to view. You can also press F1 with the component selected in the form.

Note

The components on the OCX and Samples page of the component palette are provided as examples only, not formally part of the C++Builder VCL, and therefore are not documented as part of C++Builder.

About the Object Inspector

The Object Inspector enables you to

- Set design-time properties for components you have placed on a form (or for the form itself)
- Create and help you navigate through event handlers

The Object selector at the top of the Object Inspector is a drop-down list containing all the components on the active form and it also displays their object type. This lets you quickly select different components on the current form.

You can resize the columns of the Object Inspector by dragging the separator line to a new position.

The Object Inspector has two pages:

- Properties page
- Events page

Properties page

The Properties page of the Object Inspector enables you to set design-time properties for components on your form, and for the form itself. You can set runtime properties by writing source code inside event handlers.

The Properties page displays only the published properties of the component that is selected on the form.

Events page

The Events page of the Object Inspector enables you to connect forms and components to program events. When you double-click an event from the Events page, C++Builder creates an event-handler and switches focus to the Code Editor. In the Code Editor, you write the code inside event-handlers that specifies how a component or form responds to the a particular event.

The Events page displays only the events of the component that is selected in the form.

How the Object Inspector displays properties

The Object Inspector dynamically changes the set of properties it displays, based on the component selected. For example, if you select a Label and a GroupBox, you'll see the property Color along with other properties. If you select a Label and then a Button, the choice for Color goes away because Color is not a property for buttons. Only the shared properties are displayed. The Object Inspector has several other behaviors that make it easier to set component properties at design time.

- When you use the Object Inspector to select a property, the property remains selected in the Object Inspector while you add or switch focus to other components in the form, provided that those components also share the same property. This enables you to type a new value for the property without always having to reselect the property.

If a component does not share the selected property, C++Builder selects its *Caption* property. If the component does not have a *Caption* property, C++Builder selects its *Name* property.

- When more than one component is selected in the form, the Object Inspector displays all properties that are shared among the selected components. This is true even when the value for the shared property differs among the selected components. In this case, the property values displayed are either the default, or the value of the first component selected. When you change any of the shared properties in the Object Inspector, the property value changes those values in all the selected components.

There is one exception to this: when you select multiple components in a form, the *Name* property no longer appears in the Object Inspector, even though they all have a *Name* property. This is because you cannot assign the same name to more than one component in a form.

Tab-jumping to property names in the Object Inspector

You can jump directly to a property by pressing the Tab key followed by any alphabetic character. The cursor jumps to the Property column of the first property beginning with the typed letter. (The Object Inspector lists property names alphabetically.)

To tab to a specific property (in this case, Width),

- 1 Select the form.
- 2 In the Object Inspector, select the form's *AutoScroll* property.
- 3 Press Tab, W to jump directly to the *Width* property.
- 4 Press Tab again to place the cursor in the Value column, where you can begin entering your edits.

Pressing Tab acts as a toggle between the Value column and the Property column. Whenever you are in the Property column, pressing an alphabetic character jumps you to the first property starting with that character. Object Inspector Object Inspector tabbing through forms and components tabbing through forms and components

Changing component properties

You can change component properties at design time or at runtime. You can also view a form as a text file and make changes that will be reflected in the Object Inspector.

To change a component property at design time,

- 1 Select the component in the form or with the Object selector.
- 2 Select the property that you want to change by selecting it from the Properties page.
- 3 Type a new value for that property.

To change a component property at runtime,

- 1 Select the component in your source code. (For example, Form1)
- 2 Select the property that you want to change (Color) and type a new value (clAqua).

See the following example:

```
Form1->Color = clAqua;
```

Displaying and setting shared properties

You can set shared properties to the same value without having to individually set them for each component.

To display and edit shared properties,

- 1 In the form, select the components whose shared property you want to set.
The Properties page of the Object Inspector displays only those properties that the selected components have in common. (Notice, however, that the *Name* property is no longer visible.)
- 2 With the components still selected, use the Object Inspector to set the property.

Saving your project

Once you've completed your form layout and property settings, you'll probably want to save your work. If you started with a new project, you can either save the entire project, or just the form.

To save just the form (and its associated unit),

- 1 Select the form.
- 2 Choose File|Save.

This will save the form file (.DFM) and the associated unit file pair (.CPP and .H)

To save the entire project, choose File|Save All.

You're prompted for a name for each unit (form) in the project, and for the project file itself. To learn more about saving projects, see the chapter on creating and managing projects.

Managing runtime behaviors of forms

You can specify two runtime aspects of forms at design time.

- Designating a form as the project main form
- Controlling the creation order of forms at runtime

Setting properties at runtime

Any property you can set at design time can also be set at runtime by using code. Other properties called runtime-only properties can be accessed only at runtime.

When you use the Object Inspector to set a component property at design time, you follow these steps:

- 1 Select the component.
- 2 Specify the property (by selecting it from the Properties page).
- 3 Enter a new value for that property.

Setting properties at runtime involves the same steps: in your source code, you specify the component, the property, and the new value, in that order. runtime property settings override any settings made at design time.

Specifying the project main form

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form.

To change the project main form,

- 1 Choose Project|Options to display the Project Options dialog box.
- 2 Select the Forms page of the dialog box.
- 3 In the Main Form combo box, select the form you want as the project main form and choose the OK button.

Now if you run the application, your new main form choice is displayed.

Specifying forms to auto-create

The form you specify as the project main form is always “created” (loaded in memory) when the application runs. As you create additional forms for the project, these are also auto-created at runtime.

However, there might be times when you decide you don’t want all the forms in an application created in memory when the application first starts running; you might prefer to control when the forms are created. For example, if there are several different forms that automatically connect to databases, you might prefer to create those forms only as necessary.

You can use the Project Options dialog box to specify which of your application’s forms will auto-create at runtime.

To specify whether forms are auto-created at runtime,

- 1 Choose Project|Options, and select the Forms page of the dialog box.
The names of all forms in the project are displayed in the Auto-Create Forms list.
- 2 In the Auto-Create Forms list, select any form(s) that you do not want created in memory at runtime, and choose the > button. To move all form names from one list to the other, choose the << or the >> button.
The selected files move to the Available Forms list. Forms displayed in this list do not auto-create at runtime.
- 3 Choose the OK button to save the information and close the dialog box.

Note

It’s usually best to have C++Builder create your application forms. If you decide not to auto-create a form, you must specifically create the forms at runtime by writing code. If you try to reference a form that hasn’t first been created, for example by calling its *Show* method, C++Builder raises an exception.

Controlling the form auto-create order

To change a form’s creation order, in the auto-create list, select the form name and drag it to the position you want.

Note

The main form and auto-create specifications on the Forms page of the Project Options dialog box are reflected in the *Application->CreateForm* statements in the project file. forms:auto-creating at run timeforms:auto-creating at run timeauto-creating forms at run timeauto-creating forms at run timecreating:forms:at run timecreating:forms:at run time

Instantiating forms at runtime

If you move a form into the Available Forms List Box in the Forms page of the Project Options dialog box, you must instantiate that form at runtime.

Instantiate a form a runtime when you cannot determine at design time how many instances of the form will be required when the application is running. Instantiating forms at runtime can also reduce the memory requirements of the application and reduce the amount of startup time when the application is run.

Note

When creating an application that instantiates forms at runtime, make sure that the code of the application does not try to access the instance of the form before it has been created.

To instantiate a form at runtime,

- 1 Add the name of the header file unit where the form is declared as a *#include* statement in the unit that will instantiate the form. This is necessary only if the form type is declared in a different unit.
- 2 Declare a pointer of the type of the form.
- 3 Assign the return value of the “new” *TFormName(Owner)* of the form type to the memory variable. The memory variable specifies the instance of the form type. For example:

```
TAboutBox *Box = new TAboutBox(this);  
Box->ShowModal();  
delete Box;
```

Note

The name of the pointer identifier that the instance of the form is assigned to should not be the name of an existing object or component. While unique instance names are not required at runtime, they are recommended so that the instance of the form is not confused with the Name of another object.