Welcome to Borland C++Builder Help

This topic automatically appears temporarily when you get Help (press F1) on a component from the Object Inspector or a term in the Code Editor.

This topic may also appear if the topic you are looking for is not found. This situation may occur if one of the Help files is missing from your installation. If you are certain that your installation includes all the shipping Help files and this topic appears by itself without bringing you to a relevant topic, please report it as a documentation bug.

You may be able to find the topic you want by using the Contents, Index, or Find tab from the Help system.

To search for a topic,

1. Click the Contents tab to browse through topics by category.
2. Click the Index tab to see a list of index entries.

   Either type the word you're looking for or scroll through the list.
3. Click the Find tab to search for words or phrases

Jumping in

See also

The best way to introduce yourself to the property-method-event (PME) model of programming is to write a quick application. This chapter guides you through the creation of a C++Builder program that looks up fields in a database. After you set up access to a database table, you'll write an event handler that brings up the standard File Save dialog box. This allows you to write information from the database table to a file.

To read these topics in sequence, press the >> button.

Starting a new application

See also

Before beginning any new application, you should create a unique folder to hold the application's source files. This way, you don't mix your application source files with other types of files and a unique folder lets you easily track and maintain all the files contained in your application.

1. Create the folder MySource off the root directory of your working drive to hold the project files you will create with this sample application.
2. Open a new project.

   Each application is represented in C++Builder with a *project*. When you start C++Builder, it opens with a blank project by default. If you have another project already open, choose File|New Application to create a fresh project.

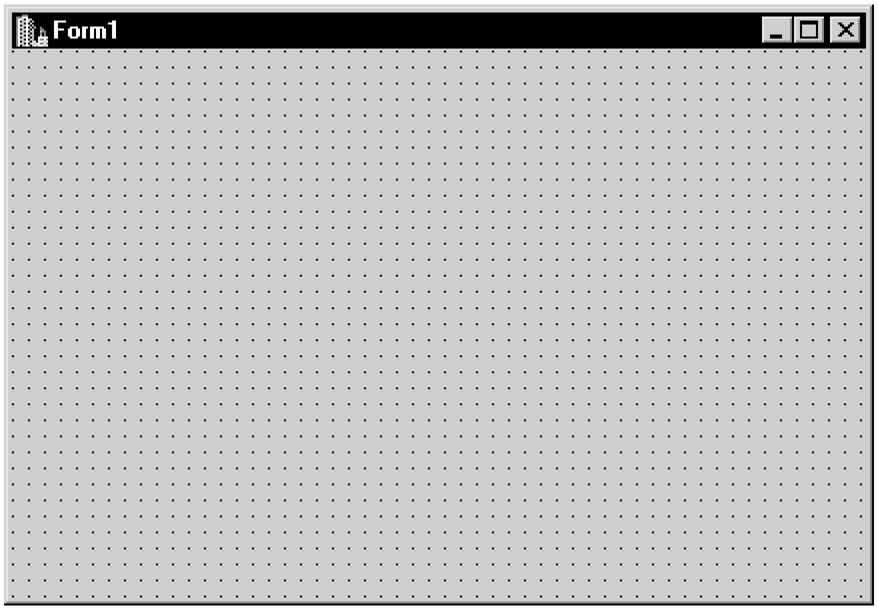
   Whenever you open a new project, C++Builder automatically creates the following two C++ files:

   - Unit1.CPP; the source file associated with the main project form.
   - Project1.CPP; the source file that keeps track of the forms in your project.
3. Choose File|Save All to save your project files to disk. When the Save dialog appears, navigate to your MySource folder, and save the two C++ files under their default file names. (Note that you can also use the C++Builder Save As dialog box to create the new MySource directory.)

   In addition to the two C++ files, C++Builder also creates other files associated with your project, as you can see by looking at your MySource directory. For more details on the files automatically created by C++Builder, see Chapter 4, "Creating and managing projects."

When you open a new project, C++Builder displays a graphical representation of the project form, named *Form1* by default.

The default project form



In C++Builder, you design the user interface for your applications using forms. Forms can contain menus and context menus, they can be put together to make application dialog boxes, and they can be parent or child windows. Essentially, forms are the canvases with which you create applications.

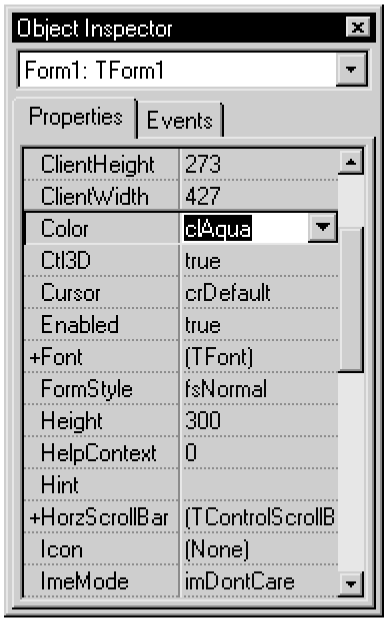
You place *objects* on forms to create user interfaces. Objects, for example, can be standard interface controls (such as checkboxes and drop-down lists), or they can be fully contained components (such as data grids, bar charts, and editors).

In addition to the form, C++Builder also displays the Object Inspector. While not an actual part of your application's interface, the Object Inspector shows you the design-time settings of the all objects you've placed in your project.

When the project form has focus, the Object Inspector shows the properties of the form, as shown in

Figure 3.2. The drop-down list at the top of the Object Inspector shows the currently displayed object; in this case, the object is *Form1:TForm1*.

The Objector Inspector



ClientHeight	273
ClientWidth	427
Color	clAqua
Ctl3D	true
Cursor	crDefault
Enabled	true
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0
Hint	
+HorzScrollBar	(TControlScrollB
Icon	(None)
ImeMode	imDontCare

To read these topics in sequence, press the >> button.

Setting a property value at design time

See also

To design the look and feel of your application interface, you set the values of object properties using the Objector Inspector. Setting property values in the Object Inspector while putting together your application interface is known as making *design time* settings.

- Set the *Color* property of *Form1* to *clAqua*.

  To set the *Color* property, find the form's *Color* property in the Object Inspector and click the drop-down list displayed to the right of the property. To change the background color of the form to aqua, choose *clAqua* from the list of predefined colors (see Figure 3.2).

Note: Using your programming environment to initialize the values of program objects is a notion that's new to many C++ programmers. When programming with C++Builder, you should make every effort to initialize object property values using the Object Inspector and its many property editors; resist the urge to set initial property values directly in the source code of the application. This way, you let C++Builder set up and maintain the overall structure of your program objects and source code.

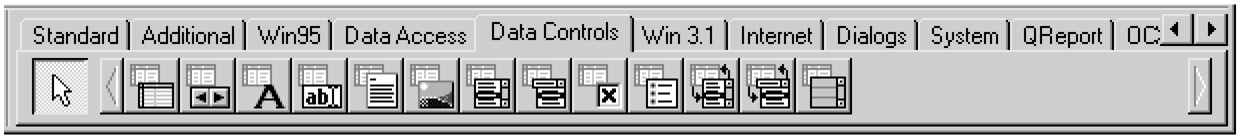
To read these topics in sequence, press the >> button.

Adding objects to the form

See also

The C++Builder Component palette (Figure 3.3) lists all the available components that you can use to create your application. Components are grouped onto different palette pages for easy access. You can scroll through the different pages of the palette by clicking the corresponding Component palette tabs.

The Data Controls page of the Component palette

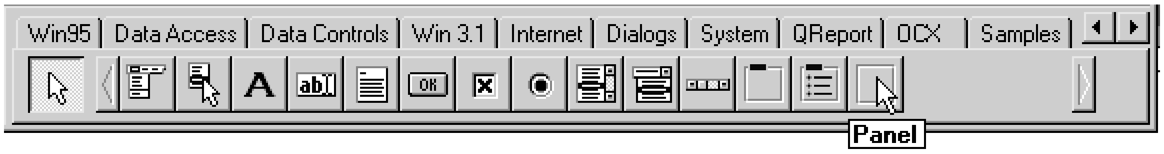


Using the components on the Component palette, you can quickly create an interface for your application.

1. Add a *Panel* component to your form by double-clicking the *Panel* component on the Standard page of the Component palette.

   To find the *Panel* component, point at a component in the Component palette for a moment; C++Builder displays a Help Hint showing the name of the respective component, as shown in Figure 3.4. When you find the *Panel* component, double-click it; C++Builder adds that component to the middle of the current form.

Finding the *Panel* component



   Placing a component on a form creates an *object instance* of that component. Although it might seem like a technicality, it helps to understand the difference between a C++Builder component and an object that you are using in your application. Once you place a component on a form, C++Builder generates all the C++ code necessary to create that object in your application. Here lies the real beauty of C++Builder; you don't have to worry about the code that creates or maintains the objects you use in your application, C++Builder does all that work for you.

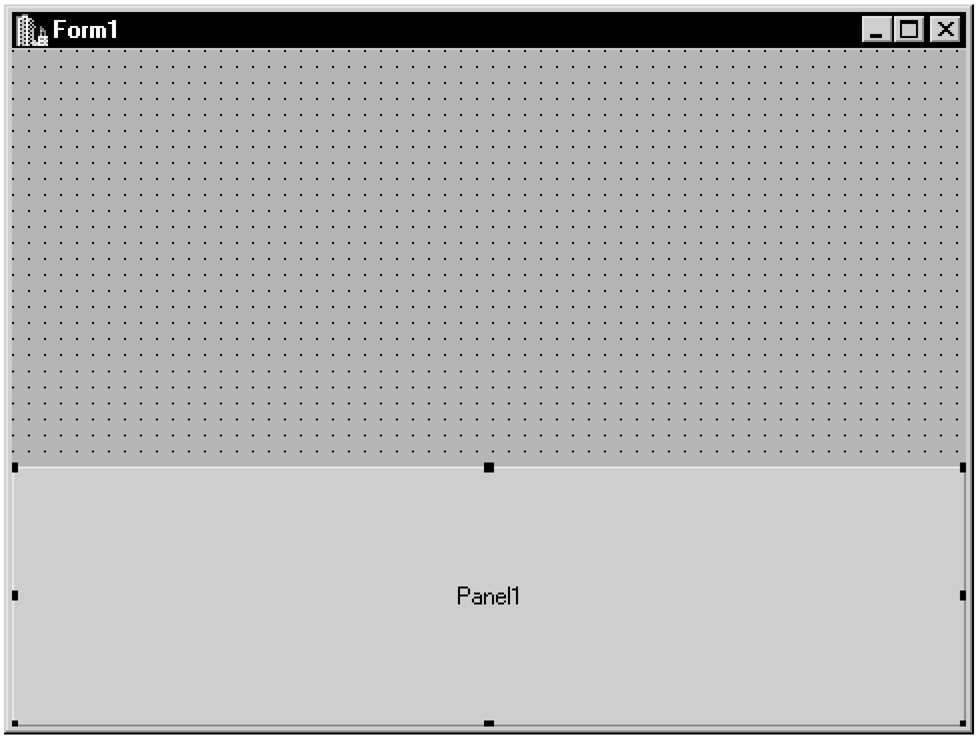
2. Set the *Panel* object's *Align* property to *alBottom*.

   Giving the *Panel* object focus on the form gives it focus in the Object Inspector. Setting the *Align* property to *alBottom* in the Object Inspector makes the panel lie flat across the bottom of the form.

3. Enlarge the size of you application's window by dragging the lower-right corner of the form.

4. Enlarge the size of the panel until the it fills the bottom third of the form (click the panel to select it, then drag its top border up) . Your form should now resemble the one shown in Figure 3.5.

Form with a Panel object



5. Drop a *Table* object on the form.

   Click the Data Access tab on the Component palette to access the components on that palette page. You can find the *Table* component on the left side of the page. When you find the *Table* component, click it once to select it, then click on the form to *drop* the component onto the form.

   When you drop the table on the form, C++Builder names the object *Table1* by default.

6. Set the *DatabaseName* property of *Table1* to *BCDEMOS*.

   Setting this property value sets up access to a database table.

To read these topics in sequence, press the >> button.

Accessing a database

See also

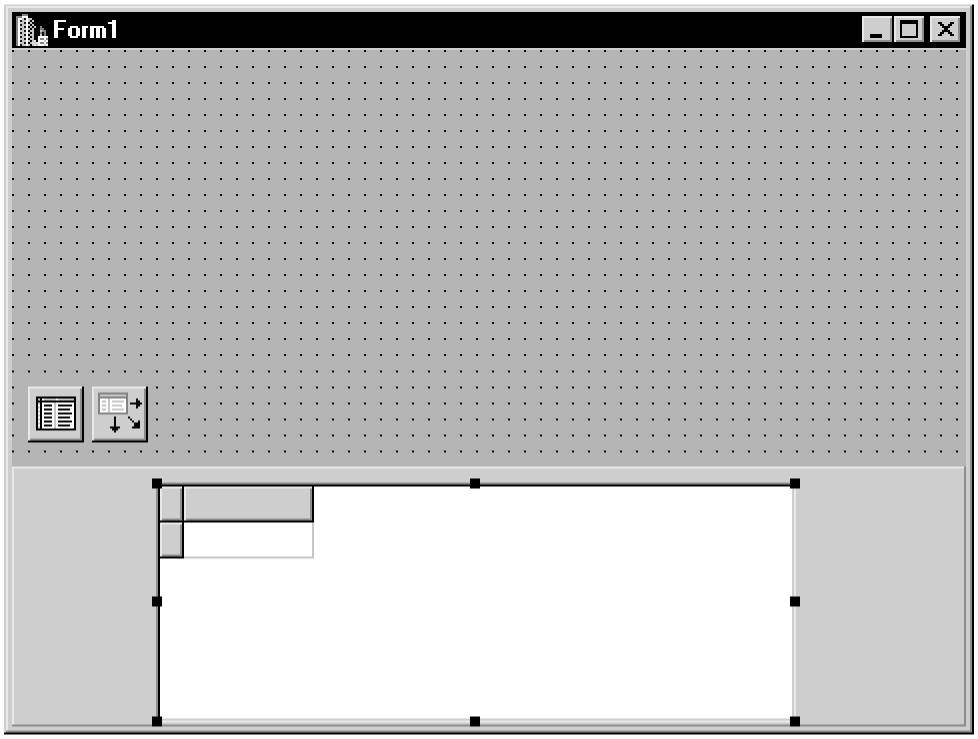
You're now ready to set up the data source for your database controls.

1. From the Data Access page of the Component palette, drop a *DataSource* component on the form. Set its *DataSet* property to *Table1*.

   On the Component palette, the Data Controls page holds the components that let you control how you view database data in your applications. To display all the fields contained in a database table, use a *DataGrid* component.

2. From the Data Controls page, choose the *DBGrid* component and drop it into *Panel1* (click the *DBGrid* component to select it, then click inside *Panel1* to drop the component there). Your form should now resemble Figure 3.6.

The form with a DBGrid object



Set the following *DataGrid* properties to align the grid and to access the database you've set up with the *Table* and *DataSource* objects:

- Set the *Align* property to *alLeft*, then drag the right edge of the grid so it fills three-quarters of the panel.
- Set the *DataSource* property to *DataSource1*.

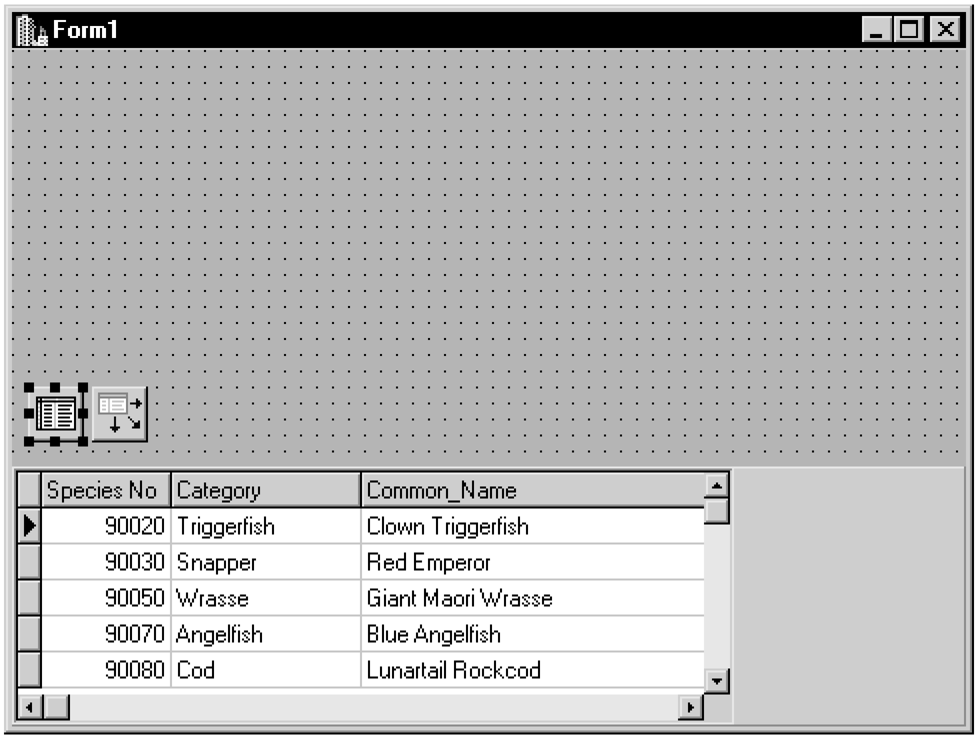
Now you can finish setting up the *Table1* object you previously dropped on the form.

3. Give the *Table1* object focus by clicking on it in the form, then set its properties as follows:

- Set *TableName* to *BIOLIFE.DB.*
- Set the *ReadOnly* property to *true*.
- Set *Active* to *true*.

When you set the *Active* property to *true*, the grid fills with the data contained in the BIOLIFE.DB database table. (If the grid doesn't fill as shown in Figure 3.7, make sure you've correctly set the properties for all of the application objects as explained in the previous instructions.)

The form with an Active Table1



	Species No	Category	Common_Name
▶	90020	Triggerfish	Clown Triggerfish
	90030	Snapper	Red Emperor
	90050	Wrasse	Giant Maori Wrasse
	90070	Angelfish	Blue Angelfish
	90080	Cod	Lunartail Rockcod

Since the *DBGrid* control is *data aware*, it displays the data in the table while you are designing your application. The data display gives you a visual check, showing that you're correctly hooked up to the database. However, take note that being data aware doesn't mean you can scroll through or edit the data at design time. To view all the data in the table, you'll have to run your application.

4. Press F9 to compile and run the project.

To read these topics in sequence, press the >> button.

Adding some spice

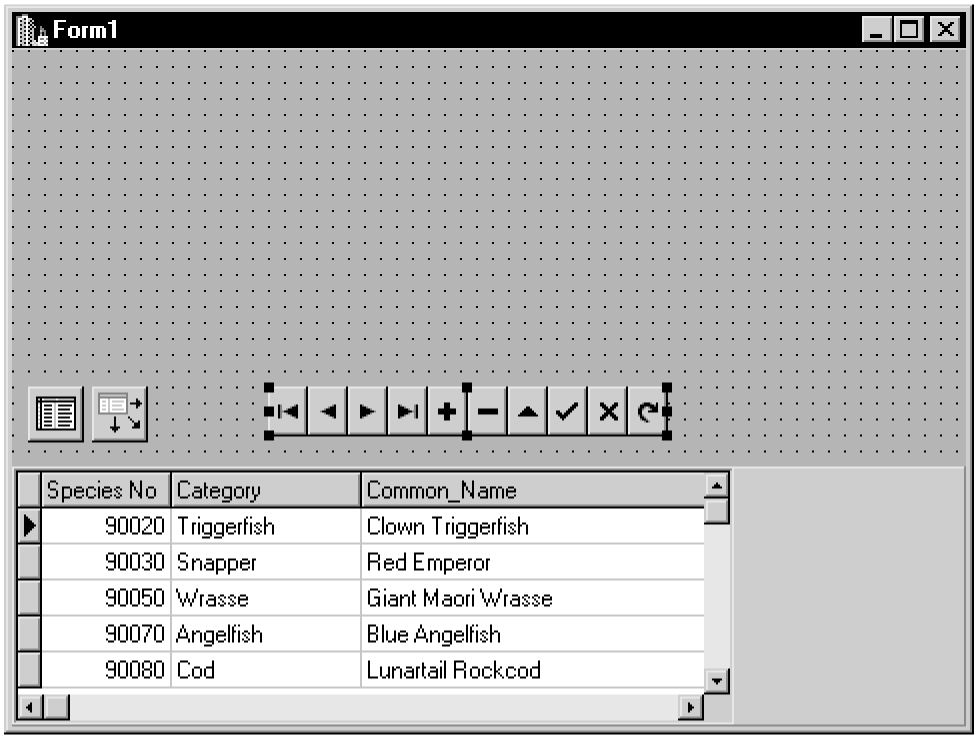
See also

When you run the application, C++Builder opens the program in a window just like the one you designed on the form. Your program is a full-fledged Windows program, complete with Minimize and Maximize buttons, and a Control menu that you can access from the upper-left corner of the window. You'll see the data grid fill with the values from the table and you can scroll the data grid to see all the values in the BIOLIFE.DB table. Since you set the ReadOnly property of the grid to *true*, you cannot change the values of the items in the database table.

Even though your program contains all this, it's still lacking a few details. Double-click the Control menu to close the application and return to the design-time view of the form.

1. From the Data Controls page of the Component palette, drop a *DBNavigator* object on the form (as shown in Figure 3.8) and set it's *DataSource* property to *DataSource1*.

Form1 with a *DBNavigator* object



	Species No	Category	Common_Name
▶	90020	Triggerfish	Clown Triggerfish
	90030	Snapper	Red Emperor
	90050	Wrasse	Giant Maori Wrasse
	90070	Angelfish	Blue Angelfish
	90080	Cod	Lunartail Rockcod

By default, the *DBNavigator* contains more controls than you'll need for this small application; for now, you will only need the *First*, *Prior*, *Next*, and *Last* navigator buttons. The *VisibleButtons* property of the *DBNavigator* controls which navigator buttons display.

2. With the *DBNavigator* in focus, double-click its *VisibleButtons* property label in the Objector Inspector.

   The enumerated list of *VisibleButtons* expands in the Objector Inspector. Each of the expanded *VisibleButtons* properties corresponds to a button on the *DBNavigator*. These properties contain Boolean values that can be set to either *true* of *false*, where a *false* value turns off the display of the button.

3. Set to false the following *VisibleButton* properties:

nbInsert	*nbDelete*	*nbEdit*
nbPost	*nbCancel*	*nbRefresh*

   Double-clicking the button property toggles the value from *true* to *false*.

4. From the Standard page of the Component palette, drop another *Panel* component onto the top of the form. C++Builder names this *Panel2* by default. To remove this caption from your running application,

clear the *Panel2* string from the panel's *Caption* property. After clearing the *Panel2* value, make sure you press Enter so the new property value takes effect.

5. Align *Panel2* to the top of the form by setting its *Align* property to *alTop*. Next drag the bottom of the panel down so it fills the top portion of the form.
6. Set the color in both panels to *clBlue*.

   C++Builder lets you set property values in several components at the same time. First click on *Panel1*, then Shift+Click *Panel2*. The Object Inspector now shows all the properties that are common to both components. From here, you can set the *Color* property of both panels to *clBlue*.
7. From the Data Controls palette page, drop a *DBImage* component on top of *Panel2* and size it so your form resembles the one shown in Figure 3.9 by setting its *Align* property to *alRight* and dragging out the left side of the image.

Form1 with a DBImage object

▪

8. Set the following *DBImage* properties:

- Set *DataSource* to *DataSource1*
- Set the *DataField* property to *Graphic*

   Again, because the *DBImage* component is data aware, the component displays the image of the fish in the first record of the table. This shows that you are indeed correctly hooked up to the database.
9. Click the green arrow speedbutton on the toolbar to compile and run your application (the Hint for this speedbutton shows Run).

   With the aid of the incremental linker in C++Builder, the compile/link time of your project should be drastically reduced from when you first ran your application.

To read these topics in sequence, press the >> button.

Final touches

See also

Now when you run the application, you can easily move through your database table using the buttons on the *DBNavigator*. However, there's a couple of other touches you can add to make the application a bit more useful. Close the running application to return to the design-mode of C++Builder.

1. From the Data Controls page of the Component palette, drop a *DBMemo* component onto *Panel2* and position it so it occupies the top left corner of the panel. Next, set the following property values:

- Set *DataSource* to *DataSource1*.
- Set *DataField* to *Notes*.
- Set *ScrollBars* to *ssVertical*.

2. Drop a *DBText* object on *Panel2* under the *DBMemo* object. Enlarge the *DBText* object so it fills the area under the *DBMemo*, then set its properties as follows:

- Set DataSource to *DataSource1*.
- Set DataField to *Common_Name*.

3. Customize the *Font* property of the *DBText* object using the Font editor.

   You can access several different types of property editors through the Object Inspector. For example, you can use the Menu editor, the Font editor, and the Picture editor to edit form menus, label fonts, and bitmap pictures and glyphs.

   When you click the Font property of the *DBText* object, C++Builder displays an ellipse button on the right side of the property setting, indicating that you can use a property editor to set this property. Clicking the ellipse displays the Font editor, a dialog that lets you edit character sizes and fonts.

   Modify the following DBText settings using the Font editor, then click OK when you're done:

- Set the *Font Style* to *Bold*.
- Set the *Color* to *Silver*.
- Set the *Size* to *12*.

To read these topics in sequence, press the >> button.

Adding an Exit button to your application

See also

Your application is now beginning to look like something you can use. To make it more user-friendly, add an Exit button so you can easily close the program.

1. Drop a *BitButton* (BitBtn) component onto *Panel1* from the Additional page of the Component palette, then set its *Kind* property to *bkClose*.

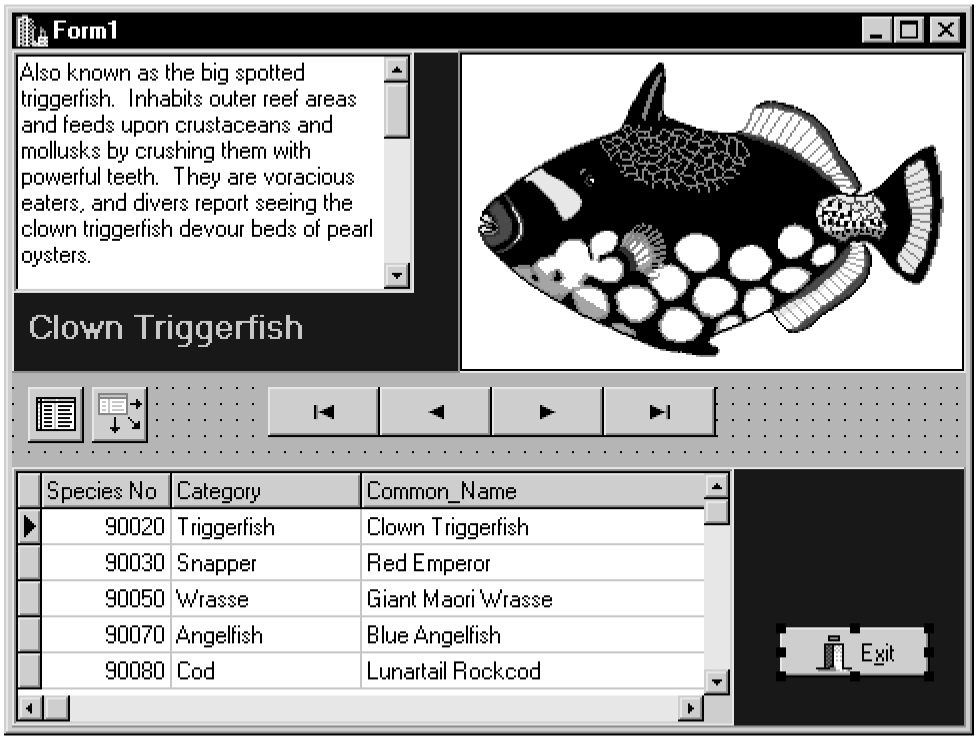
   This sets default values of the button so that when you click in the running application, the application closes.

2. Change the *Caption* property of the *BitButton* so that it reads E&xit.

   The ampersand before the *x* turns this character into a shortcut key, which lets you exit the program by pressing Alt+X.

   Your program should now look like the one shown in Figure 3.10.

Form1 with the Exit button



Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them with powerful teeth. They are voracious eaters, and divers report seeing the clown triggerfish devour beds of pearl oysters.

Clown Triggerfish

Species No	Category	Common_Name
90020	Triggerfish	Clown Triggerfish
90030	Snapper	Red Emperor
90050	Wrasse	Giant Maori Wrasse
90070	Angelfish	Blue Angelfish
90080	Cod	Lunartail Rockcod

3. Again, compile and run your application by pressing F9.

To read these topics in sequence, press the >> button.

Hooking up an event handler

See also

You now have a fully functioning Windows application that accesses a database table and displays images, memos, and individual data fields from the database. There's even an Exit button so you can gracefully shut down the application.

Up until now, however, you have not typed a single line of program code. And this is how it should be if you want to take complete advantage of the C++Builder environment. By using the Object Inspector to set the design-time values of your object properties, you let C++Builder maintain the code that it generates by itself. In other words, let C++Builder do the grunt work and save your coding efforts for the event handlers, the code that makes your program perform the tasks in your application.

Your next programming feat will be to hook up an event handler to a button, a task you'll encounter often when designing user interfaces with C++Builder. You will program the button so that when clicked, an event handler will call the standard Windows File Save dialog box, which lets you save information from the database out to a file.

1. From the Dialogs page of the Component palette, drop a *SaveDialog* object onto the form next to the *DataSource1* object.
2. From the Additional Component palette page, drop a *BitBtn* (BitButton) object above the *Exit* button, and set its following properties:

- Set *Caption* to *&Save*.
- Set *Glyph* to the predefined FileSave.BMP bitmap using the Picture editor.

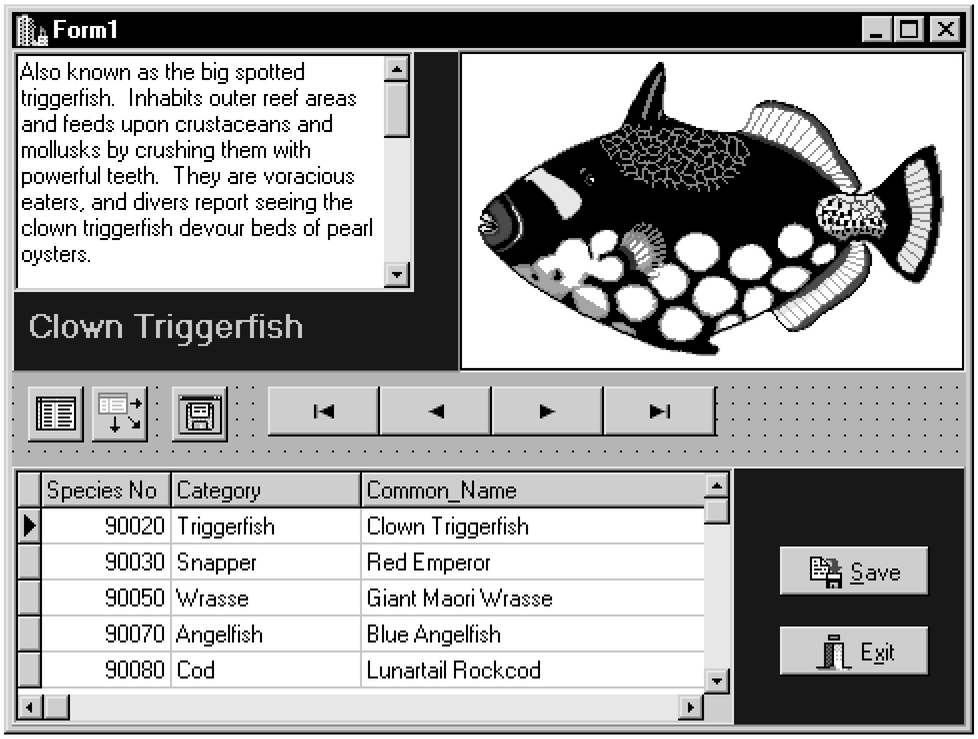
When you click the *Glyph* property, C++Builder displays an ellipse button on the right side of the property setting, indicating that you can use a property editor to set this property. Click the ellipse to access the Picture editor. In the Picture editor, click the Load button, and navigate to the following folder in the C++Builder directory hierarchy:

<C++Builder main directory>\Images\Buttons

In this directory, find and load the FileSave.BMP glyph. When you choose OK, you should see this glyph appear on the left side of your Save button.

The user interface of your application is now complete. If you followed all the steps in this chapter, your final creation should resemble the form shown in Figure 3.11.

The final user interface



Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them with powerful teeth. They are voracious eaters, and divers report seeing the clown triggerfish devour beds of pearl oysters.

Clown Triggerfish

Species No	Category	Common_Name
90020	Triggerfish	Clown Triggerfish
90030	Snapper	Red Emperor
90050	Wrasse	Giant Maori Wrasse
90070	Angelfish	Blue Angelfish
90080	Cod	Lunartail Rockcod

Now comes the meat and potatoes part of your C++Builder programming, creating and writing an event handler. The easy part is creating the event handler, while the technical programming comes to play as you write the actual C++ code for the event.

3. Double-click the Save button to create the default event handler for the button.

   Whenever you double click an object on a C++Builder form, C++Builder generates the code for that object's *default event handler*. C++Builder also opens the Edit window and places the cursor after the opening brace of the event handler. In this case, C++Builder generates the following event-handler code:

   ```
   void __fastcall TForm1::BitBtn2Click(Tobject *Sender)
   {

   }
   ```

   Up until now, you've used the Object Inspector to set object properties. However, you can also use the Object Inspector to access and create object event handlers. By clicking the Events tab on the Object Inspector, you can see the available events for the object currently in focus on the C++Builder form.

   Most components on the Component palette have a default event. When you double click an object on a form, C++Builder creates the handler for the default event. For most components, the *OnClick* event is default; this is the event that gets called whenever you click an object in a running application.

4. Complete the event handler with the code that is shown between the opening and closing braces of the event handler:

   ```
   void __fastcall TForm1::BitBtn2Click(Tobject *Sender)
   {
      FILE *outfile;
      char buff[100];
      sprintf(buff, "Save Info For: %s", DBText1->Field->AsString.c_str());
      SaveDialog1->Title = buff;
      if (SaveDialog1->Execute())
      {
          outfile = fopen(SaveDialog1->FileName.c_str(), "wt");
          if (outfile)
          {
              fprintf(outfile, "Facts on the %s\n\n",
   ```

```
            DBText11->Field->AsString.c_str());
          for (int i=0; i < DBGrid1->FieldCount; i++)
            fprintf(outfile, "%s: %s\n",
              DBGrid1->Fields[i]->FieldName.c_str(),
              DBGrid1->Fields[i]->AsString.c_str());
          fprintf(outfile, "\n%s\n", DBMemo1->Text.c_str());
      }
      fclose(outfile);
    }
 }
```

Detailing the workings of this code is beyond the scope of this introductory chapter. However, it should be suffice to say that this code calls the File Save dialog box (the *SaveDialog1* object) when you click the Save button. When you specify a file name, your application writes out data on the currently selected fish to that file.

5. To run the finished application, press F9.

Congratulations, you have now completed your first C++Builder application. After you run the program, don't forget to choose File|Save All to write your source files to disk!

First Look

See also

Borland C++Builder is an object-oriented, visual programming environment for Rapid Application Development of general-purpose and client/server applications for Microsoft Windows 95 and Windows NT. Using C++Builder, you can create highly efficient Windows applications with a minimum of manual coding.

C++Builder provides a comprehensive library of reusable components and a suite of RAD design tools, including application and form templates, and programming experts.

When you start C++Builder, you are immediately placed within the visual programming environment. It is within this environment that C++Builder provides all the tools you need to design, develop, test, and debug applications. This chapter briefly describes the development environment and touches on many of the tools that are available to you through the C++Builder environment. The rest of this manual, the other documents in the C++Builder documentation set, and online Help files provide details on how to use the tools.

To read these topics in sequence, press the >> button.

Starting C++Builder

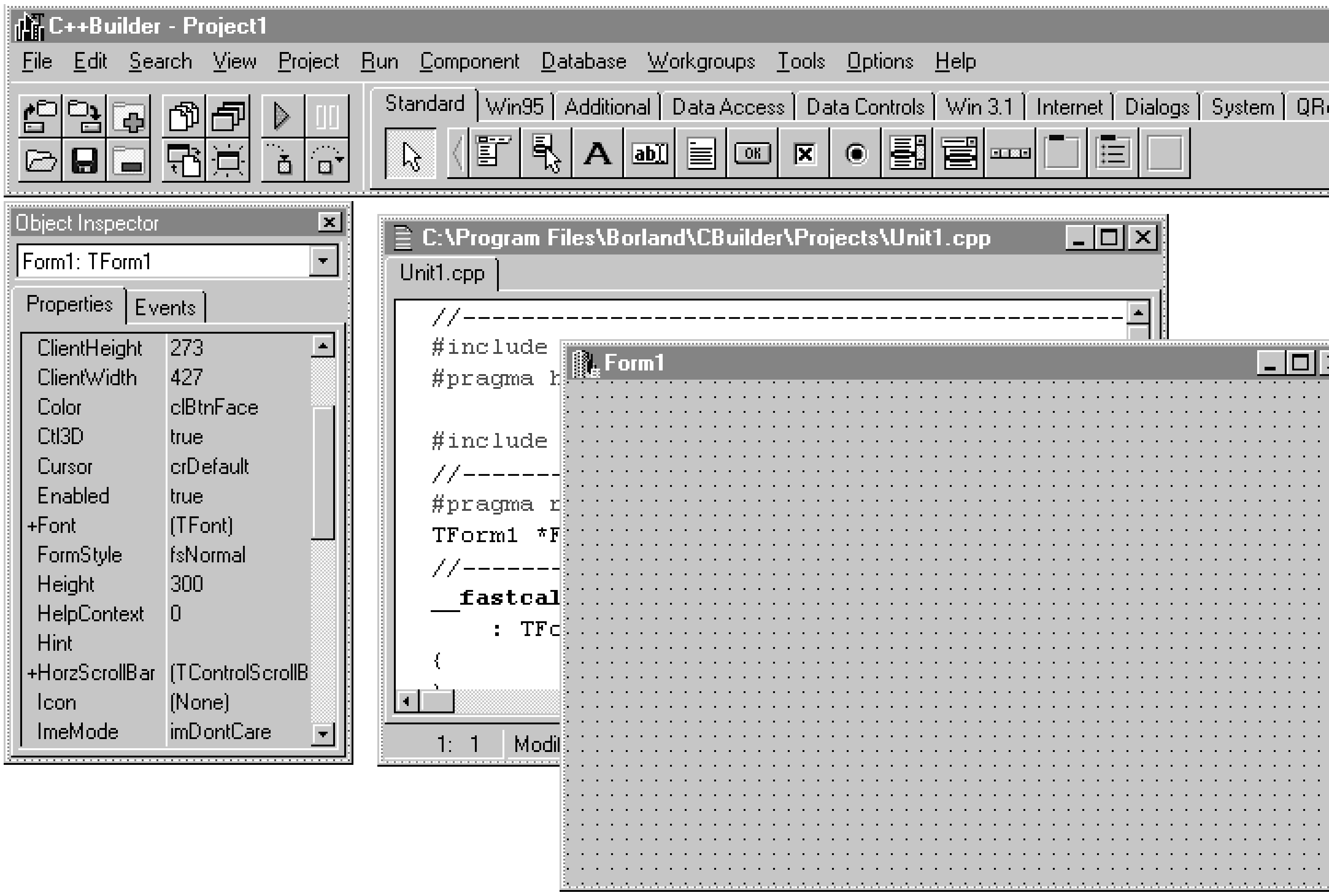
See also

The best way to get familiar with C++Builder is to start it up. You start C++Builder the same way you start any Windows-based application. Here are some of the common ways:

- Double-click on the C++Builder icon.
- Use the Explorer or the File Manager to navigate the file system. Locate and double-click the BCB.EXE file (if you performed a default installation, this file is located in \Program Files\Borland\CBuilder\Bin).
- Choose Run from the Windows Start menu, and specify the path to BCB.EXE.

The following figure shows what C++Builder looks like when you first start it up.

C++Builder development environment



ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	true
Cursor	crDefault
Enabled	true
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0
Hint	
+HorzScrollBar	(TControlScrollB
Icon	(None)
ImeMode	imDontCare

```
//---------------------------------------------
#include
#pragma h

#include
//-------
#pragma r
TForm1 *F
//-------
__fastcal
    : TFc
{
```

The C++Builder development environment (also called the IDE) has several flexible parts that you can locate anywhere on the screen. The Main window contains the main menu, toolbar, and Component palette. The Object Inspector, Code editor, and a form are automatically displayed. As you are working, you can resize each part and display additional tools as needed.

The environment is customizable so you can set it up to meet your needs. For example, you can make any tools you frequently use accessible through buttons on the toolbar.

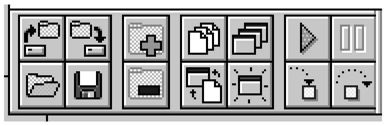
To read these topics in sequence, press the >> button.

Accessing commands fast

See also

The main menu at the top of the Main window provides access to many commands and tools in the C++Builder development environment. You can also see a *toolbar* in the upper left corner under the main menu. The toolbar has buttons that can save you time while you're working. Each button performs a common operation or command (Open File, Save Project, Run, and so on). The following figure illustrates the toolbar and shows the function of each of its speed buttons.

C++Builder toolbar



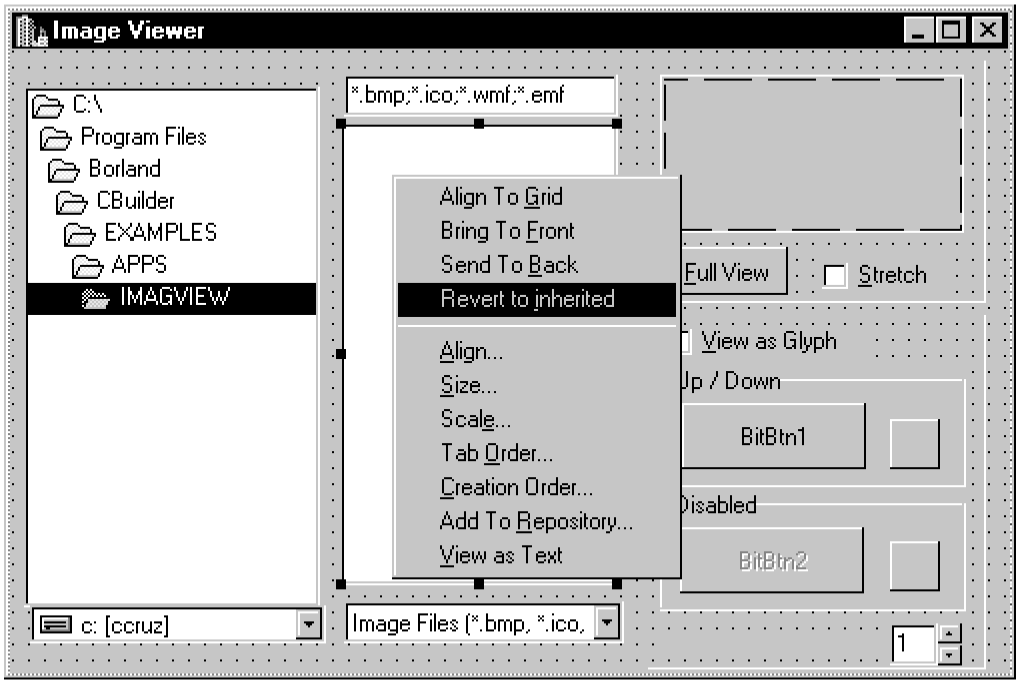
Right-click the toolbar and choose Properties to customize it for your needs.
To read these topics in sequence, press the >> button.

Displaying commands in context

See also

You can display commands that relate to the context in which you are working: point at the area and right-click. A menu containing commands relevant to your work is displayed. This menu is sometimes referred to as a *context menu* or SpeedMenu. You can customize context menus so they provide the specific functions you need.

Context menu on a C++Builder form



Context menus are available when you right-click on many elements in the C++Builder development environment, including

- Forms
- Component palette
- Object Inspector
- Toolbar
- Project Manager
- Debugger
- Code editor

To read these topics in sequence, press the >> button.

Designing applications

See also

C++Builder immediately presents you with the tools necessary to start designing applications:

- Blank window, known as a form, on which you design the UI for your application
- Extensive library of UI elements, called components, which reside on the Component palette
- An easy way to change object traits by using the Object Inspector
- Direct access to the underlying program logic through the Code editor
- Many other tools such as an image editor on the toolbar and an integrated debugger on menus to support application development

You can use C++Builder to design any kind of 32-bit application--from general-purpose utilities to sophisticated data access programs. C++Builder's database tools and data-aware components let you quickly develop powerful desktop database and client/server applications. Using C++Builder's data-aware controls, you can view live data while you design your application, letting you immediately see the results of database queries and changes to the application's interface.

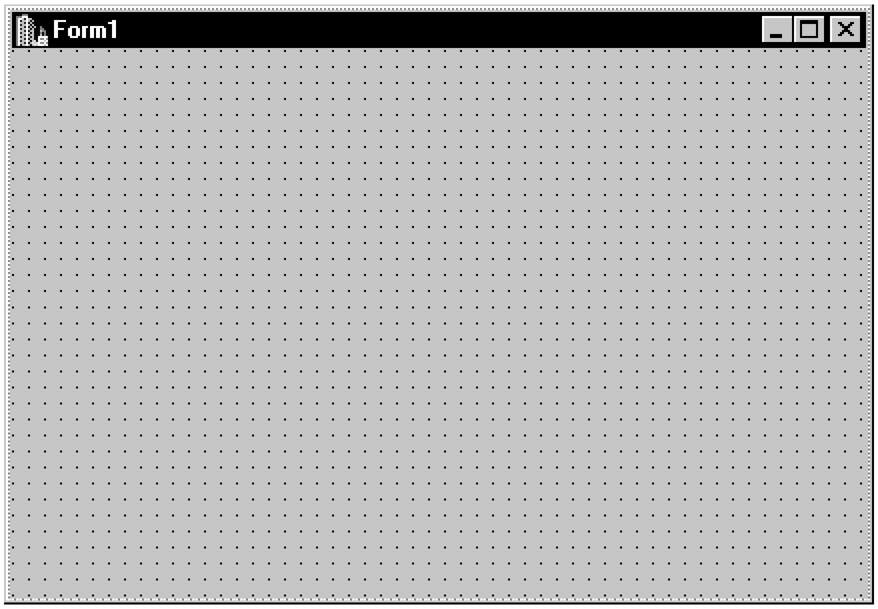
To read these topics in sequence, press the >> button.

Creating the application interface

See also

All visual design work in C++Builder takes place on *forms*. When you open C++Builder or create a new project, a blank form is displayed on the screen. You can use it to start building your application interfaces and dialog boxes. You design the look and feel of the graphical user interface for the application by placing and arranging visual *components such as buttons and list boxes on the form*. You can also place invisible components on forms to capture information from databases, perform calculations, and manage other interactions.

Form1 example



Designing a UI this way allows you to prototype your application very quickly and see how it will look right away. Read Creating and editing forms. for more information.
To read these topics in sequence, press the >> button.

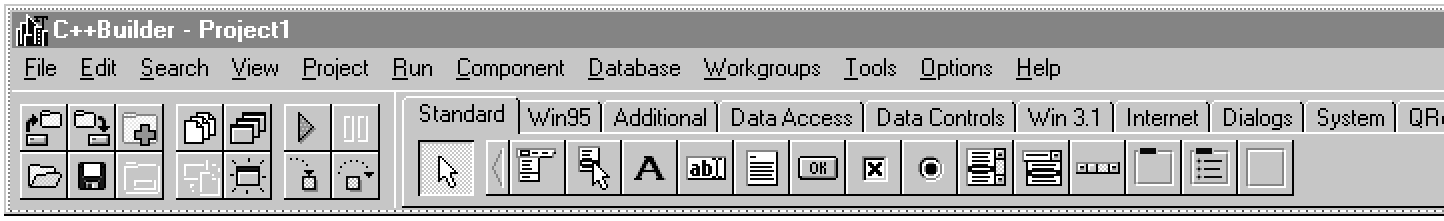
Adding components

See also

Components are the elements you use to build your C++Builder applications. They include all the visible parts of an application interface, such as dialog boxes and buttons, as well as those that aren't visible while the application is running, such as system timers or Dynamic Data Exchange (DDE) servers.

Many visual components are provided in the development environment itself on the Component palette. You select components from the Component palette and drop them onto the form to design the application user interface. Once a visual component is on the form, you can adjust its position and size.

Component palette


C++Builder - Project1
File Edit Search View Project Run Component Database Workgroups Tools Options Help
Standard Win95 Additional Data Access Data Controls Win 3.1 Internet Dialogs System QR

C++Builder components are grouped functionally on the different pages of the Component palette. For example, commonly used components such as those to create menus, edit boxes, or buttons are located on the Standard page of the Component palette. Handy controls such as a timer, paintbox, media player, and OLE container are on the System page.
Forms and components have many features in common; you can think of a form as an object that can contain other components.

C++Builder includes many components that you can use in the Visual Component Library (VCL). Refer to the *VCL Reference* in print or online for details on all of the available components.You can customize the component library by adding or deleting components or you can write new components. Refer to the *Component Writer's Guide* for details on writing components.

To read these topics in sequence, press the >> button.

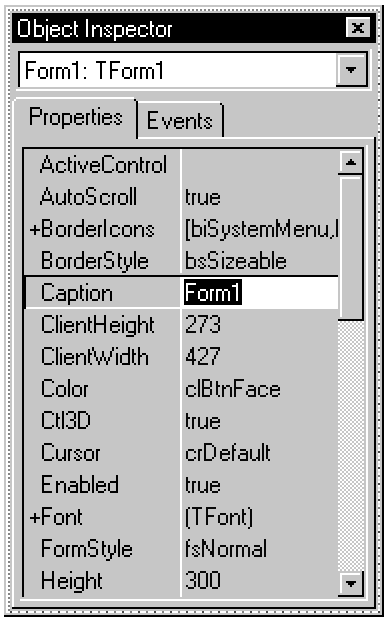
Changing component behavior

See also

You can easily customize the way a component appears and behaves in your application by using the Object Inspector. When a component has focus on the form, its properties and events are displayed in the Object Inspector.

The following figure shows what the Object Inspector looks like when the form itself is selected.

Object Inspector



ActiveControl	
AutoScroll	true
+BorderIcons	[biSystemMenu,I
BorderStyle	bsSizeable
Caption	Form1
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	true
Cursor	crDefault
Enabled	true
+Font	(TFont)
FormStyle	fsNormal
Height	300

You use the Properties page of the Object Inspector to set the initial program start-up values for the components you've placed on the form. You use the Events page of the Object Inspector to quickly navigate among events that each component can handle. By clicking on a particular event, C++Builder generates the event handler code for that specific component event. In C++Builder, you will spend most of your programming time writing in the event handlers for the objects that you place in your application forms.

To keep the Object Inspector visible at all times, right-click it and choose Stay On Top from the context menu. For more information about the Object Inspector see Creating and editing forms.

To read these topics in sequence, press the >> button.

Designing menus

See also

After you add a menu component to a form, you can use the Menu Designer to create and edit menu bars and pop-up menus. You need to add a menu component to your form for every menu you want to include in your application. C++Builder provides predesigned menu templates that you can use to design menus, or you can build the menu structure of your program from scratch.

The menus you design are immediately visible in the form without having to run the application to see the results. You can also change menus at runtime to provide additional options for the application user. Read Designing menus to learn about the Menu Designer.

To read these topics in sequence, press the >> button.

Developing applications

See also

As you visually design the user interface for your application, C++Builder generates the underlying C++ code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment as well.

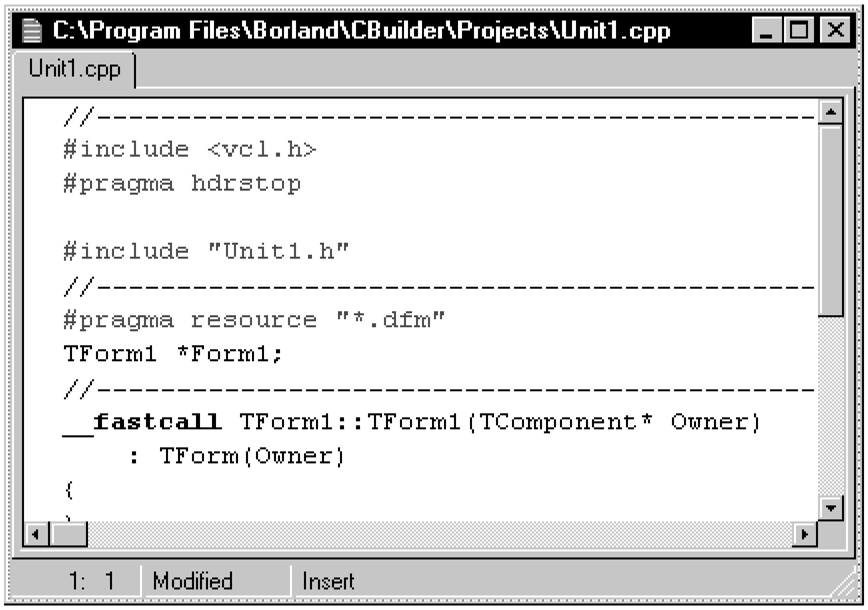
To read these topics in sequence, press the >> button.

Editing code

See also

The C++Builder Code editor is a full-featured ASCII editor. With the Code editor, you can view and edit all the code contained in your project source files. The following figure displays a file called UNIT1.CPP; the default unit file created with each new C++Builder project. This file is one of several files that make up a C++Builder project. For more on project files, see Creating and Managing Projects.

Sample code



```
//---------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
```

See Working with the Code editor for information on the Code editor.

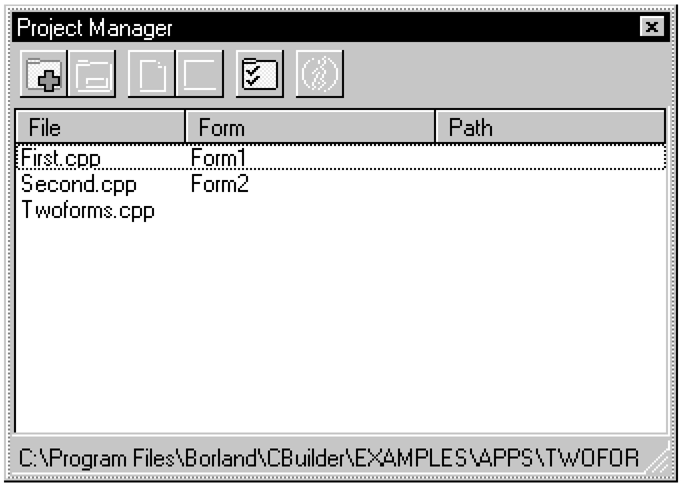
To read these topics in sequence, press the >> button.

Managing project files

See also

Using the Project Manager, you can easily keep track of and access the files that make up a C++Builder application. When you access the Project Manager from the View menu, you can see a list of each form file in your application and easily navigate between them.

C++Builder Project Manager



File	Form	Path
First.cpp	Form1	
Second.cpp	Form2	
Twoforms.cpp		

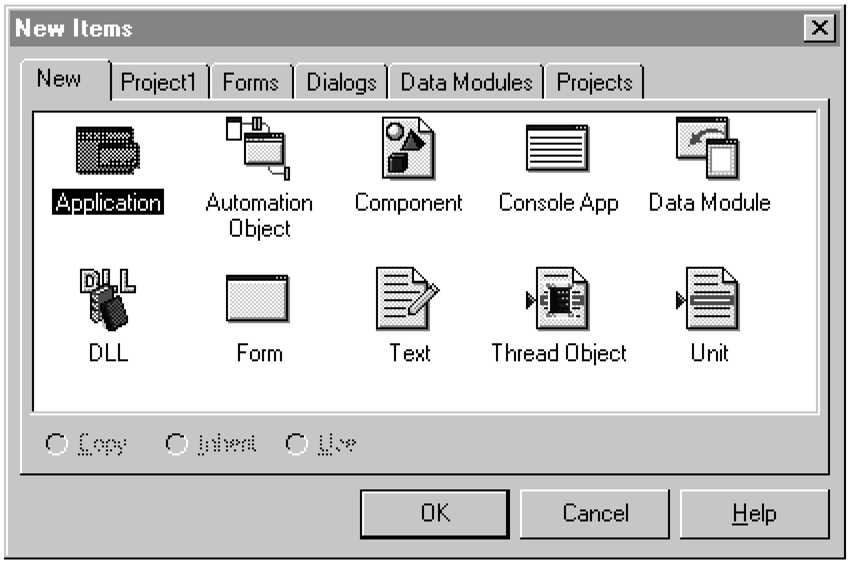
C:\Program Files\Borland\CBuilder\EXAMPLES\APPS\TWOFOR

Refer to Creating and managing projects for details on managing C++Builder projects.
To read these topics in sequence, press the >> button.

Storing application objects

See also

C++Builder uses the Object Repository to store application objects such as forms, data modules, experts, and DLLs. The Object Repository lets you easily reuse the objects that you build. Reusing objects lets you build families of applications with common user interfaces and functionality. It also provides a central location for application development tools that all members of a development team can access over a network. Therefore, you can develop forms that you can use in more than one application and save them in the Object Repository. Or, in a larger environment, you can develop a standard set of forms for other developers to use within their applications.



See Creating and managing projects for information on the Object Repository. Chapter 13 also provides a tutorial that shows how to reuse forms with the Object Repository.
To read these topics in sequence, press the >> button.

Using the online Help system

See also

The Help system provides online access to detailed information about C++Builder. Using Help is a convenient way to learn about the extensive language features, programming tasks, libraries, compiler options, and the C++Builder programming environment.

C++Builder provides Help on numerous libraries of reusable components such as the Visual Component Library, the Standard C++ Library, the Runtime Library, and Win32 APIs. You will also find Help on additional products (such as PVCS version control software and InstallShield Express) that are supplied with some versions of C++Builder.

Getting Help

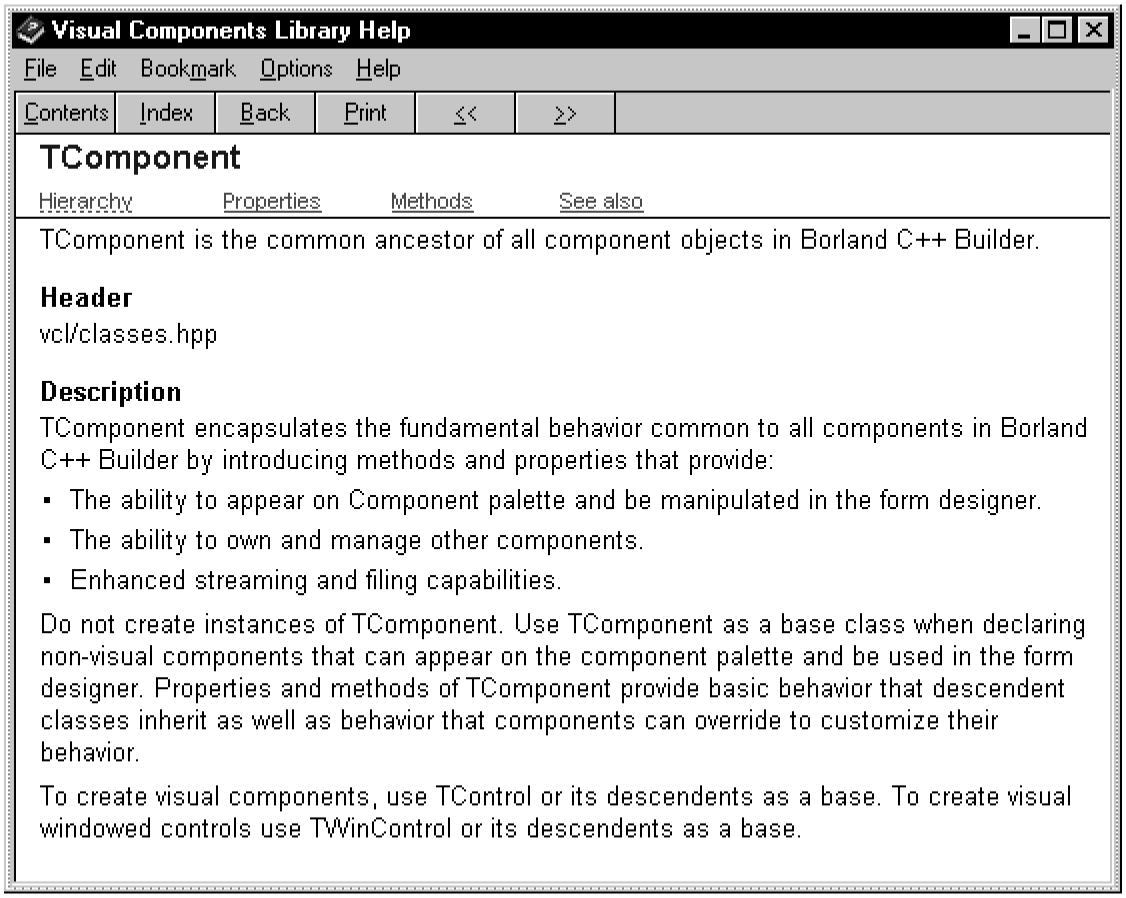
If you have questions while using C++Builder, try using the online Help system to find the information you need. You can display online Help while using C++Builder in any of the following ways:

- Choose Help|Contents from the main menu
- Press F1
- In a dialog box, click Help

You can also press F1 while the Code editor has focus to get help on the currently highlighted program token. C++Builder will find the closest match in the Help system.

The Help Contents displays a Help Topics dialog box showing the contents of the *User's Guide*. You can also go straight to the *VCL Reference*, runtime *Library Reference*, or *Programmer's Guide* contents by choosing Help (from the main menu) then picking the appropriate menu item. The following figure shows a sample Help topic and Contents screen.

C++Builder Help Topics Contents and Help file


Visual Components Library Help
File Edit Bookmark Options Help
Contents Index Back Print << >>
Hierarchy Properties Methods See also

TComponent

TComponent is the common ancestor of all component objects in Borland C++ Builder.

Header

vcl/classes.hpp

Description

TComponent encapsulates the fundamental behavior common to all components in Borland C++ Builder by introducing methods and properties that provide:

- The ability to appear on Component palette and be manipulated in the form designer.
- The ability to own and manage other components.
- Enhanced streaming and filing capabilities.

Do not create instances of TComponent. Use TComponent as a base class when declaring non-visual components that can appear on the component palette and be used in the form designer. Properties and methods of TComponent provide basic behavior that descendent classes inherit as well as behavior that components can override to customize their behavior.

To create visual components, use TControl or its descendents as a base. To create visual windowed controls use TWinControl or its descendents as a base.

Displaying online information

From the contents, you can view a topic by clicking on it. You can also click the Index or Find tabs on the Help Topics dialog box to access information in a different way. The index provides entries that either display an associated topic or, if there are several topics, displays a list of relevant topics to choose from. Find provides a full text search so you can search for specific terms throughout the Help system.

Displaying context-sensitive information

Pressing F1 within the C++Builder programming environment displays context-sensitive Help. If a dialog box is displayed, F1 provides information on using the dialog box and its options. If you're using the Code editor, you can place the insertion point next to or select an item for which you want to get Help and press F1. You can select any term (for example, a property or event in the Object Inspector) and press F1 to see a description of it. You can also use F1 to display information about selected menu items or error messages.

Customizing Help

With so much online information available, you may want to consider customizing the Help system to suit your needs. OpenHelp is a C++Builder tool that lets you specify the exact scope of the material you routinely access when using Help. OpenHelp.exe is provided with C++Builder.

To use OpenHelp, click on the OpenHelp icon.

OpenHelp lets you customize which Help files are searched when using Help. You can set up several search ranges so you can view the kind of information that is relevant to your work at a particular time. Whichever search range is best suited to your work is set as the default.

The Help files that are in a search range control what you will see in the Contents screen, and you can customize the order of topics in the Contents. The search range also determines which Help files have entries included in the Index and which Help files are searched in a full text search.

If you use other tools with C++Builder, you can add their Help files to the search range so you can integrate their Help information as well.

For more information on customizing Help, refer to OPENHELP.HLP in online Help.

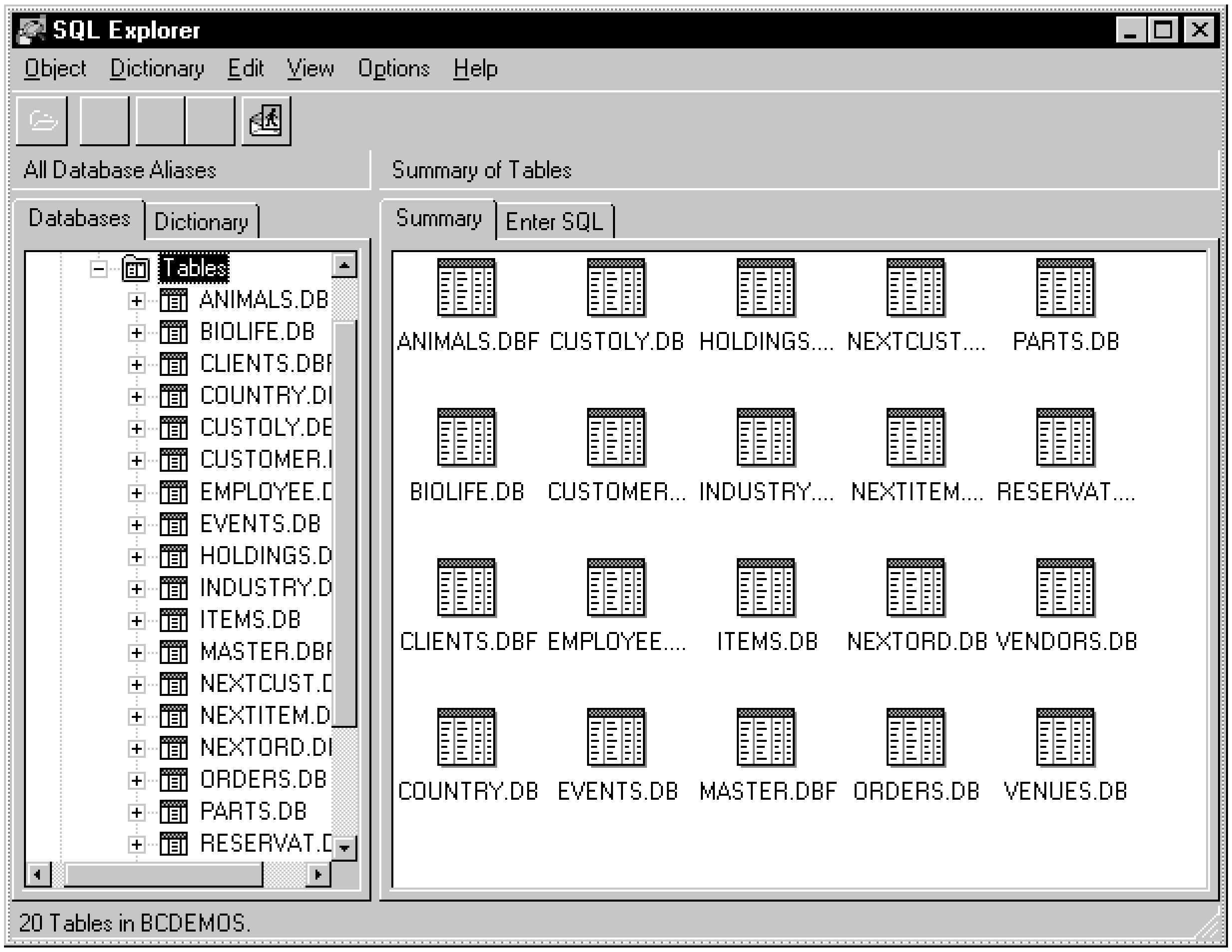
Writing database applications

See also

One of C++Builder's strengths is its support for creating advanced database applications. C++Builder includes built-in tools that allow you to connect to Oracle, Sybase, Informix, dBASE, Paradox, or other servers while providing transparent data sharing between applications. The Borland Database Engine (BDE) supports scaling from desktop to client/server applications.

Tools, such as the Database Explorer, simplify the task of writing database applications. The Database Explorer is a hierarchical browser for inspecting and modifying database server-specific schema objects including tables, fields, stored procedure definitions, triggers, references, and index descriptions. The following figure shows the Database Explorer.

C++Builder Database Explorer



Through a persistent connection to a database, Database Explorer enables you to

- Create and maintain database aliases
- View schema data in a database, such as tables, stored procedures, and triggers
- View table objects, such as fields and indexes
- Create, view, and modify data in tables
- Enter SQL statements to directly query any database
- Create and maintain data dictionaries to store attribute sets

See the *Database Application Developer's Guide* for details on how to use C++Builder to create database client applications that use the Borland Database Engine to retrieve data from and send data to local and remote database servers.

To read these topics in sequence, press the >> button.

Compiling and running applications

See also

All C++Builder projects target a single executable file, either an EXE or a DLL file. You can view or test your application at various stages of development by compiling, building, or running it. See Setting project options and compiling for a discussion on compiling C++Builder applications.

You can customize the project through Project Options dialog boxes or through the command line. C++Builder comes with a full set of command-line tools. Command-line tools include compilers, linkers, a project builder (MAKE), and other utilities. Refer to the BCBTOOLS online Help file for information on the command-line tools.

To read these topics in sequence, press the >> button.

Debugging applications

See also

C++Builder provides an integrated debugger that helps you find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging. See About the Integrated Debugger for details.

You prepare for debugging by compiling and linking your application with debug information. Then, you can begin debugging by running your program under the control of the debugger. You can use the features of the debugger to examine the current state of the program. You can debug a specific area of the program, execute it one line or instruction at a time, set breakpoints, or pause execution. By viewing the values of variables, the functions on the call stack, and the program output, you can check that the area of code you are examining is performing as designed.

To read these topics in sequence, press the >> button.

Deploying applications

See also

Some versions of C++Builder include add-on tools to help with application deployment. For example, InstallShield Express helps you to create an installation package for your application that includes all of the files needed for running a distributed application. PVCS Version Manager software is also available for tracking application updates.

To read these topics in sequence, press the >> button.

Creating forms

See also

In this chapter you will learn the following:

- What are forms and components
- How to set component properties
- How to create and customize dialog boxes
- How to manage runtime behaviors of forms
- How to customize forms and save them to the Object Repository
- How to use predesigned forms

Creating a C++Builder application

See also

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 Write event handler code to make the component do something.

6 Save the project.

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?

See also

Forms are specialized objects on which you place VCL components. Forms generally appear as windows and dialog boxes in a running application.

Even though you design the application user interface using forms, it's important to understand that a form is just another component. So as with other components, you interact with forms by reading and setting their properties, calling their methods, and responding to their events.

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

What are components?

See also

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. 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 a form

See also

C++Builder provides several ways to create a new form, 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.

You can then add components to the form.

To reuse existing forms, see "Using predesigned forms".

Placing components on a form

See also

The easiest way to place a component on a form is to

1 Double click a component on the Component palette.

2 Click the form to place the component. The upper left corner of the component is placed where you click on the form.

3 Click the resizing handles on the component and drag to size the component, if applicable.

You've just started creating your first form. Before you place other components on the form, see "Understanding components" to learn how to select appropriate components for your application.

To learn more about using components, see "Manipulating components in forms".

Understanding components

See also

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.

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*, whereas the functional view divides components according to *what you use them for*.

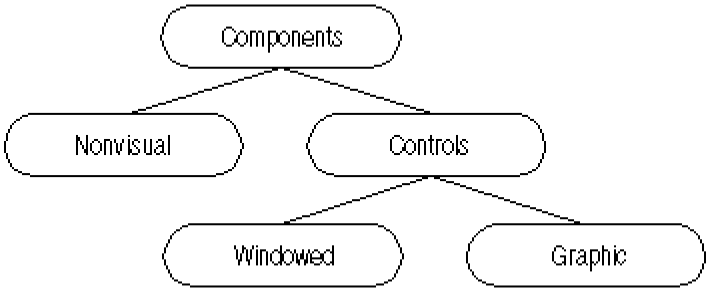
The component hierarchy

See also

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*.

The figure that follows shows a simplified view of the major kinds of components.

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 components, 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

See also

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.

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, choosing options	button, check box, radio button, bitmap button, 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

C++Builder's standard components

See also

The components on the palette are grouped into pages according to similar functions. You can add, remove, and rearrange components as you choose. The following table lists the default component palette pages and the kinds of components each contains.

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
QReport	Quick Reports components for easily creating embedded reports
ActiveX	Sample OLE controls
Samples	Sample custom components, including gauge, color grid, spin buttons, directory outline, and calendar grid
Internet	ActiveX controls for Internet programming

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

Common component elements

See also

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 used in your application

See also

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.

Note: As long as you use the Object Inspector to change a component's Name property, C++Builder maintains your changes in the underlying code that it generates. 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; unless you change all occurances of the object's name.

Changing the name of a component

See also

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, following 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

See also

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

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

Adding components to a form

See also

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

If there are already components in the center of the form, new ones will be placed on top. You can move them to the desired position.

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, 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

See also

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.

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.

Once a component is on the form, you must add it to a container component by cutting and then pasting it. Simply moving it will not add it to the container.

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

See also

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.

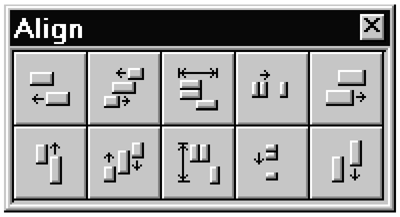
To align components using the Alignment palette

1 Select the components.

2 Choose View|Alignment palette.

The Alignment palette appears.

3 Select an alignment icon from the palette.

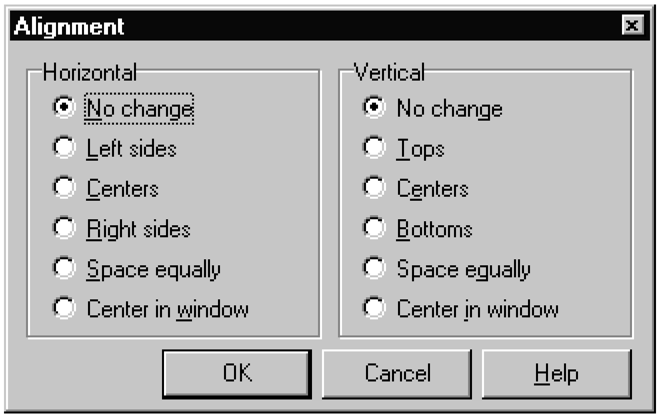


To align components using the Alignment dialog box

1 Select the components.

2 Choose Edit|Align.

The Alignment dialog box appears.


Alignment

Horizontal
- (●) No change
- ( ) Left sides
- ( ) Centers
- ( ) Right sides
- ( ) Space equally
- ( ) Center in window

Vertical
- (●) No change
- ( ) Tops
- ( ) Centers
- ( ) Bottoms
- ( ) Space equally
- ( ) Center in window

3 Select the alignment options you want in the dialog box.

4 Choose OK to put your alignment options into effect.

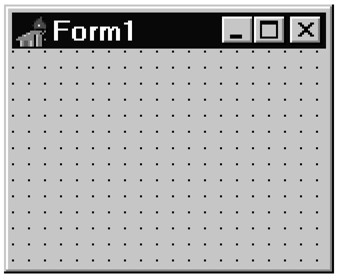
You can continue to choose or modify alignment options as long as the components remain selected.

Using the grid to align components

See also

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.

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 Options|Environment to display the Environment Options dialog box.

2 Click the Preferences tab to display the Preferences page if it is not already visible.

3 In the Form designer options, check Display grid to view the grid, or uncheck it to hide it.

To enable the Snap To Grid option

1 Choose Options|Environment 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 Options|Environment 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

See also

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.

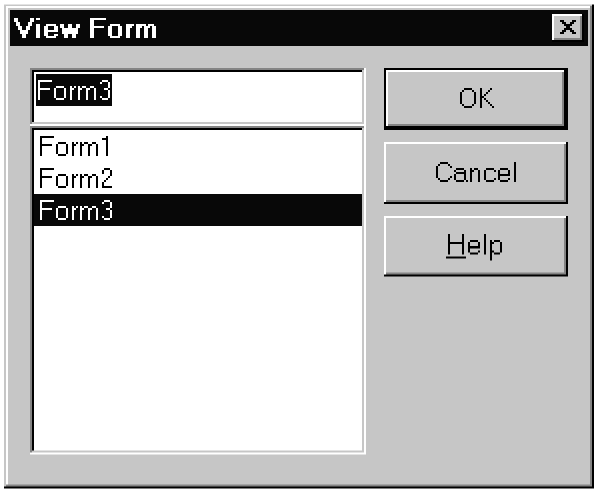
Viewing forms and units

See also

To switch among forms in a project

1 Choose View|Forms.

The View Form dialog box appears.

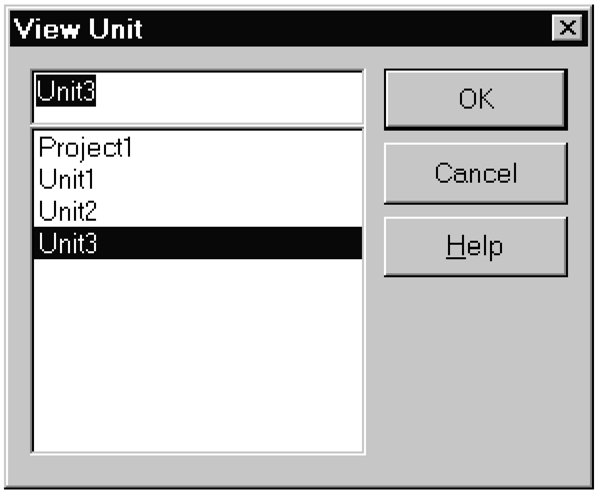


2 Select the form you want to view, then choose OK.

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. Choose View|Project Manager.

Setting component properties

See also

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 ActiveX 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

See also

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 the object type of the selected component. 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

See also

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

See also

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 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

See also

The Object Inspector dynamically changes the set of properties it displays, based on the component selected. Only the shared properties are displayed. 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. 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

See also

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.

Changing component properties

See also

You can change component properties at design time or dynamically when the application runs. 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 using the Code editor. (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

See also

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, shift+click to 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 because each component must have a unique name.)

2 With the components still selected, use the Object Inspector to set the property.

Building dialog boxes

See also

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

See also

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
- Specifying a caption 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

Making a dialog box modal or modeless

See also

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.

Setting form properties for a dialog box

See also

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.

Property	Setting	Effect
BorderIcons		
biSystemMenu	False	Removes Control (System) menu
biMinimize	False	Removes Minimize button
biMazimize	False	Removes Maximize button
BorderStyle		
	bsSizeable	Lets the user resize the form border
	bsSingle	Provides a single, non-resizable border
	bsNone	No distinguishable border; not resizable
	bsDialog	Window has a border, but not resizable
	bsToolWindow	Makes the title bar small; window is not resizable
	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

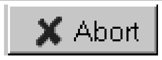
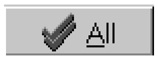
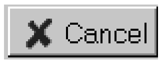
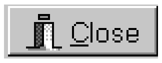
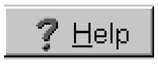
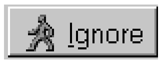
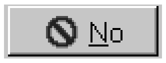
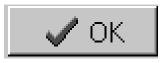
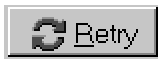
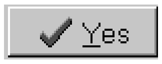
See also

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

See also

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.)

Kind value	Effect	Appearance	Equivalent property setting(s)	Comments
bkAbort	Makes a Cancel button with Abort as caption		Caption := 'Abort'ModalResult := mrAbort	Red X appears next to caption.
bkAll	Creates an OK button (with All caption)		Caption := '&All'ModalResult := 8	Green double check mark appears next to caption.
bkCancel	Makes a Cancel button		Caption := 'Cancel'Cancel := trueModalResult := mrCancel	Red X appears next to caption.
bkClose	Creates a Close button; closes the window		Caption := '&Close'	A lavender "exit" door appears as the glyph for this button.
bkCustom	None	N/A	N/A	Use this setting to create a custom command button, including specifying a custom Glyph bitmap.
bkHelp	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.)
bkIgnore	Creates a button to ignore changes and proceed with specified action		Caption := '&Ignore'ModalResult := mrIgnore	Use to continue an operation after an error condition has occurred.
bkNo	Makes a Cancel button (with No as the caption)		Caption := '&No'Cancel := trueModalResult := mrNo	Red circle with slash appears next to caption.
bkOK	Creates an OK button		Caption := 'OK'Default := trueModalResult := mrOK	Green check mark appears next to caption.
bkRetry	Creates a button to retry specified action		Caption := '&Retry'ModalResult := mrRetry	Green circular arrow appears next to the caption.
bkYes	Creates an OK button (with Yes caption)		Caption := '&Yes'Default := trueModalResult := mrYes	Green check mark appears next to caption.

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

See also

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 *bkCance*l. This sets the button's *Cancel* property to *true*, and the *ModalResult* property to *mrCancel*.

Executing button code on Enter

See also

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 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

See also

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

See also

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*.

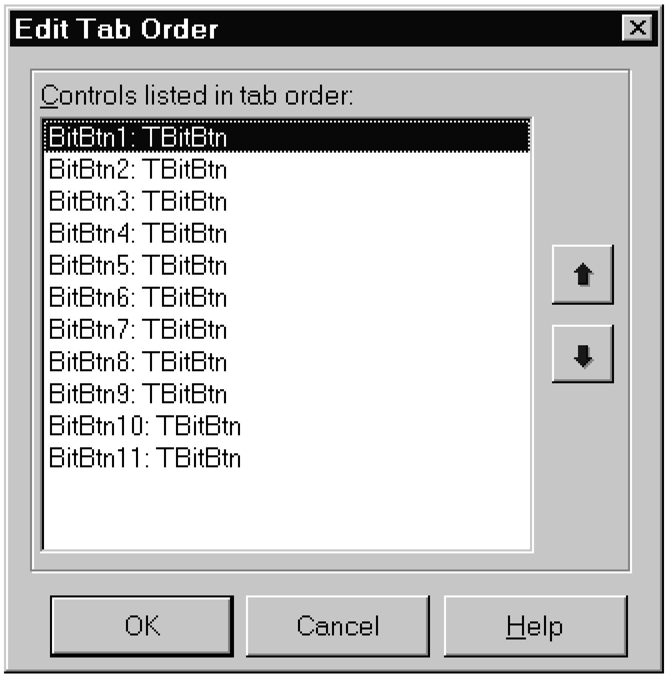
The tab order is initially set by C++Builder, corresponding to the order in which you add components to the form. You can change this by using the Edit Tab Order dialog box, or by changing the *TabOrder* property of each component.

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.



Edit Tab Order

Controls listed in tab order:

BitBtn1: TBitBtn
BitBtn2: TBitBtn
BitBtn3: TBitBtn
BitBtn4: TBitBtn
BitBtn5: TBitBtn
BitBtn6: TBitBtn
BitBtn7: TBitBtn
BitBtn8: TBitBtn
BitBtn9: TBitBtn
BitBtn10: TBitBtn
BitBtn11: TBitBtn

3 In the Controls list, select a component, and press the appropriate arrow button (Up or Down), or move 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 renumbers 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

See also

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

See also

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.

Disabling components

See also

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:

```
<component>->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;
}
```

Managing runtime behaviors of forms

See also

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

See also

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

See also

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 Options|Project 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's 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

See also

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 Options|Project, 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 forms move to the Available Forms list. Forms displayed in this list are not automatically created 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

See also

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.

Instantiating forms at runtime

See also

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 at 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.

Using predesigned forms

See also

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 then place components on the form. You can also choose to add a predesigned form to the project and use it or modifiy it.

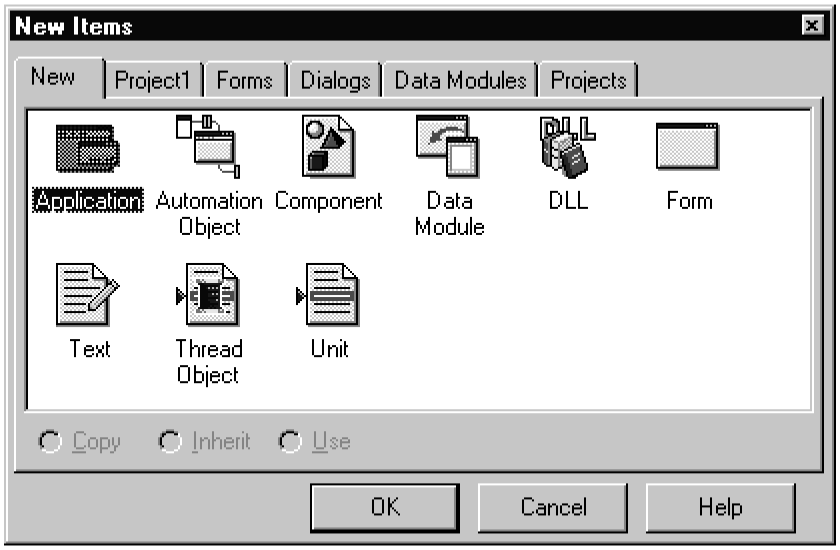
Adding a form to a project

See also

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. To add a predesigned form to your project, choose File|New.

When you choose File|New, C++Builder displays the New Items dialog box.

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

See also

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.

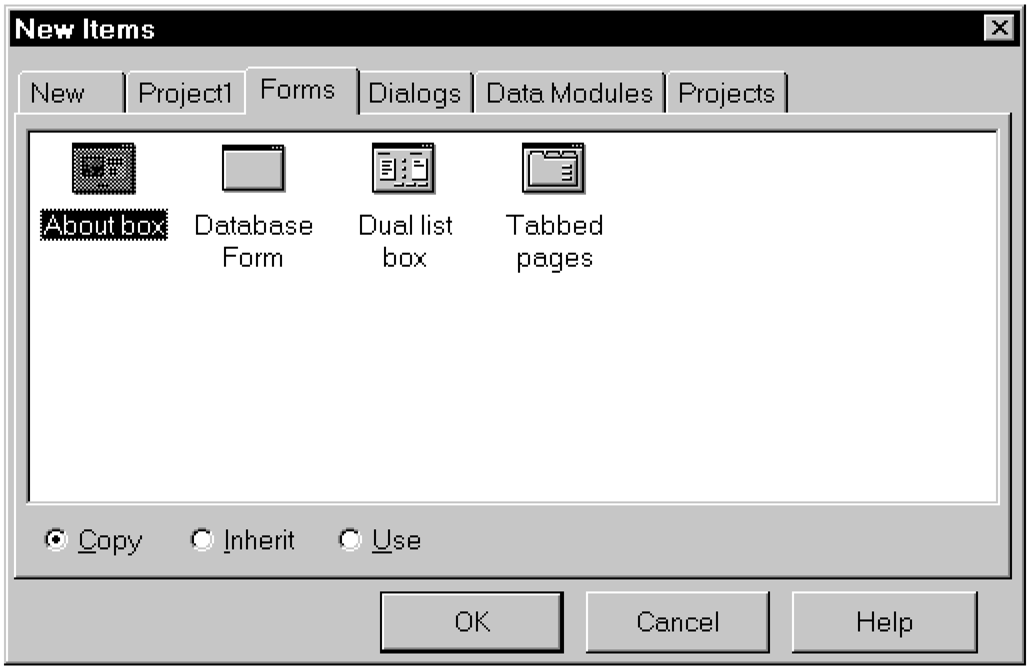
The New Items dialog box appears.

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.

Standard C++Builder forms in the Object Repository



1 Select the form you want to add.

2 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.

3 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.

Creating form templates

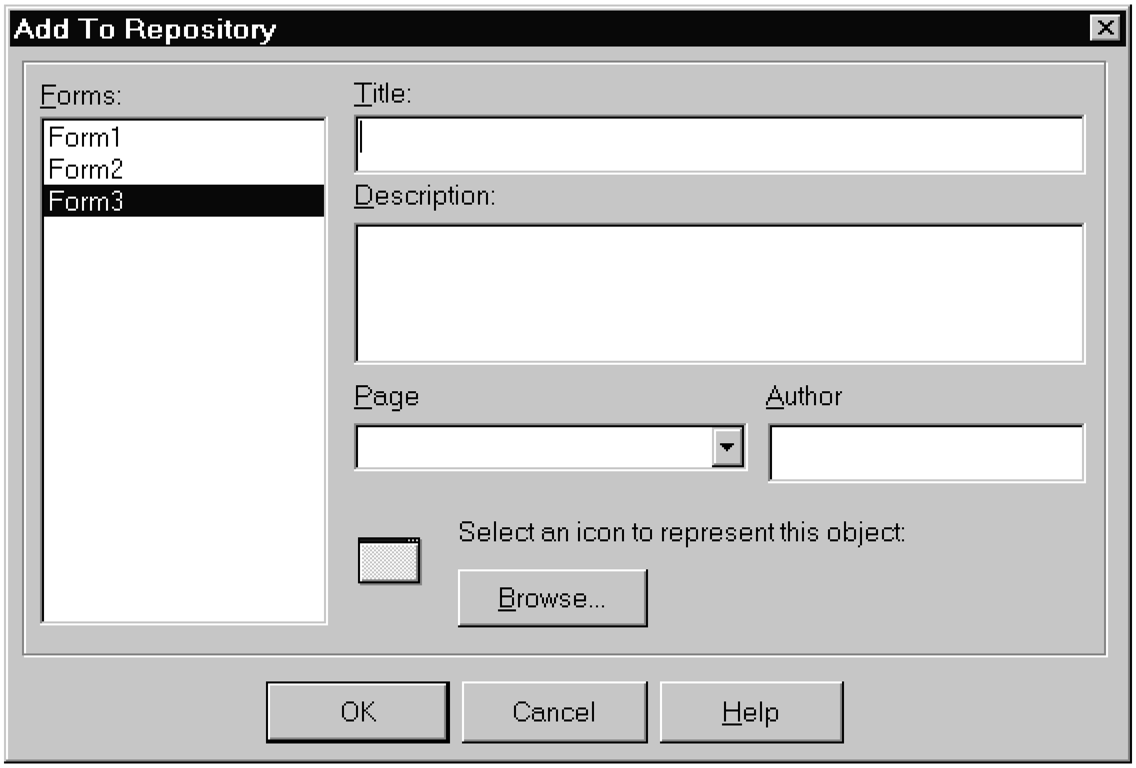
See also

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 page tab you selected above, your form appears in the templates list, with the icon and title you chose.

Inheriting from forms in the Object Repository

See also

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.

Sharing forms

See also

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. Place forms that you want to reuse into the Object Repository.

Linking forms

See also

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 Hdr.

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.

Creating and managing projects

See also

Before you begin programming with C++Builder, it helps to understand the files and projects that you will use to create your C++Builder applications.

This chapter introduces you to the files that make up a C++Builder project and how to manage the projects you create with C++Builder. In addition, the chapter talks about navigating in the C++Builder environment, and how to save and use objects stored in the Object Repository. The main topics in this chapter are

- C++Builder projects
- Saving and naming C++Builder files
- Using the Project Manager
- Navigating among project components
- Using the Object Repository

To read these topics in sequence, press the >> button.

C++Builder projects

See also

A C++Builder *project* is a collection of all the files that together make up the executable application or .DLL you are creating. In C++Builder, project files are organized in the project .MAK file.

As your application grows in size, you'll find it becomes more and more dependent on different intermediate files as its complexity increases. A Windows program can be composed of resource scripts, import libraries, object libraries, and source code, with each file type requiring a special setup to be compiled and linked into the final executable image. Although C++Builder takes care of the complexities of creating Windows applications, having a firm grasp of the files that go into your C++Builder projects will help you better understand the applications you create.

As the number of files in your project increases, the need increases for a way to manage the different project components. By studying the files that make up a project, you can see how a project combines one or more *source files* to produce a *target file.* Target files, for example, can be .OBJ, .DLL, or .EXE files. Each target file is dependent on all the sources files that are used to create it. Source files consist of files like .C, .CPP, and .H files. *Project management* is the organization and management of the sources and targets that make up your project.

To read these topics in sequence, press the >> button.

C++Builder project files

See also

To manage projects effectively, you need to understand the different file types that can constitute a C++Builder project. By default, C++Builder generates several sets of files with each new project you create. Project files can be broken down into the file groups shown in the following table:

File group	Files types in group
Project files	.mak, .cpp, and .res files
Form files	.cpp, .h, and .dfm files
Unit files	.cpp and .h files
The desktop file	a .dsk file

In C++Builder, you don't need to directly deal with most of the files in your project; C++Builder generates and maintains many of the files for you. Although you can edit most files directly in the C++Builder Code editor, it's usually easier and more reliable to edit these files using the visual editors and tools integrated with C++Builder.

To read these topics in sequence, press the >> button.

Project files

When you create a new project, C++Builder automatically creates the following three files:

- Project1.CPP
- Project1.MAK
- Project1.RES

These files are termed *project files* since their names are derived from the name you give your project. When you name the project .MAK file, C++Builder updates the three files, giving them all the same filename. For more information on naming C++Builder files, see "Saving and naming C++Builder files" on page 450.

Project1.CPP

The central point for each project's source code is the project .CPP file, which C++Builder names *Project1.CPP* by default. This file contains the entry point for your executable image (the *WinMain()* function for applications), which is where your executable begins program execution. In addition to the program entry point, the project .CPP file coordinates the other forms and units contained in your application. It is the place where you should declare your global variables and constants.

Although a very simple project could contain nothing besides the project .CPP file, a useful C++Builder project will contain references to the forms and units that your project uses. When you load, save, or compile a project, C++Builder knows which other files to act on by looking at the project .CPP file.

Project1.MAK

Project1.MAK is the project *makefile*, a text file that contains the project option settings and the build rules for the project. C++Builder uses this file to determine how the different source and target files are combined to make your final executable images.

For most projects, you can let C++Builder generate and maintain this file. To view the project makefile, choose View|Project Makefile. For more information on setting advanced project options and the MAKE utility, refer to the C++Builder online Help system.

Project1.RES

C++Builder uses standard Windows-format resource files to include items such as the application icon in a project. You can edit the project resource file to add other resources, such as bitmaps, cursors, icons, or strings to the application.

By default, each C++Builder application you create will have a resource file with the same name as the project file, but with the extension .RES. This file contains a binary image of the program icon.

Form files

C++Builder represents each form in your application with three different files, given the following names by default:

- Unit1.DFM
- Unit1.CPP and Unit1.h

When you first save a new form, C++Builder prompts you to enter a name for the form .CPP file. C++Builder uses the filename you supply to name all three form files.

Unit1.DFM

Forms are the most visible part of most C++Builder applications. Normally, you design forms using the form image and the C++Builder environment. When you create a form at design-time, C++Builder stores the image of the form in a binary file that describes the form. C++Builder gives binary form files the file extension of .DFM.

If needed, you can open a .DFM file in the Code editor to modify a text version of the data in the binary file. To switch back and forth between the form and text views of a form, use the View As Text and View As Form commands on the context menus of the Form image and the Code editor, respectively. Outside of C++Builder, you can use CONVERT.EXE to do similar conversions.

You can also translate your form files to text versions for maintenance and version control.

Unit1.CPP and Unit1.h

C++Builder generates a C++ source file and header file pair for each form you create. Together, the three .DFM, .CPP, and .h files make up a C++Builder form. A form .CPP file contains the event handlers that you write to handle the events of the components you place on the associated form--it is within the form .CPP files that you do most of your C++Builder programming. A .CPP file that is associated with a form is sometimes called the *form unit*.

Unit files

There is a small distinction between "Form files" and "Unit files" as they are talked about here. *Unit files* are not associated with a form; they simply a .CPP and .h file pair. Unit files (among other things) can be used to house global functions and functions that you plan to share across projects. See the next section, "About form and unit files," for more information on these file types.

The desktop file

C++Builder also generates an optional desktop file that it maintains in accordance with your project. The desktop settings file stores information about the state of your project, such as which windows are open and in what positions. This allows you to restore your workspace on a project-by-project basis.

The desktop-settings file has the same name as the project file, but with the extension .DSK. To generate and automatically save a desktop file,

1. Choose Options Environment.
2. On the Preferences page, check the Desktop box in the Autosave options section.

   C++Builder will generate and save a project .DSK file whenever you close the project. The file is stored in your main project directory.

When you create a desktop file for your projects, C++Builder will open with the same project and window setup that you had when you last closed C++Builder.

To read these topics in sequence, press the >> button.

About form and unit files

See also

C++Builder supports separately compiled modules of code called *units*. Using units promotes structured, reusable code across projects. The most common units in C++Builder projects are form units, which contain the event handlers and other code for the forms you create in your C++Builder projects.

Unit files are the building blocks of your C++Builder applications--they contain the C++ source code to all your application forms. For every form you create, C++Builder generates a unit source code file and header file pair.

Unit files may initially be created as part of a project, but this is not required. You can also create and save units as stand-alone files that any project can use. In this case, the unit files don't have a form associated with them. Stand-alone unit files can contain any kind of C or C++ code you want to write, such as global functions and functions that are not directly associated with a form. For example, you can write your own procedures, functions, .DLLs, and components, and put their source code in a separate unit file that has no associated form.

Whenever you add a new or existing unit to a project, you must include that unit in the project by including the unit's header file in the project .CPP file. To do so, use the File|Include Unit Hdr command to make sure that C++Builder correctly updates all files.

When you compile a unit, the C++Builder compiler produces a binary object file with the same filename as the unit source file, but with an .OBJ file extension. You should never need to open these binary files, and you do not need to distribute them with your completed application.

Unit files for forms

Whenever you create a new form, C++Builder creates the following default form unit code:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
```

The statement TForm1 *Form1; creates the Form1 object from the *TForm1* class, meaning *From1* contains all the behaviors and characteristics of a *TForm* object. To this form unit file, you add event handlers and support code for each component event that you need to handle for this particular application form.

Stand-alone unit files

You can write custom functions within any unit, including those associated with a form. You might, however, want to reuse the routines that you write. In this case, it's better to create a separate unit to contain those routines. By creating stand-alone units that have no associated forms, you can easily make your functions available to other units in your project, and to other projects that can use the functions.

To create a unit file not associated with a form,

1. Choose File|New Unit.

You can also use the New Items dialog box:

1. Choose File|New to access the New Items dialog box.
2. Choose Unit on the New page of the dialog box, then click OK.

Using either method, C++Builder opens a new unit .CPP file without appending the default unit form code.

It is not necessary to have a project open unless you want the new unit to be part of a project.

If you have a project open, you can also choose New Unit from the Project Manager context menu. When you create a unit in an open project with the New Unit command, the new unit is registered in the project file.

To read these topics in sequence, press the >> button.

Saving and naming C++Builder files

See also

When you first create a new project in C++Builder, it automatically creates several files and gives them all default names: Project1.MAK, Unit1.CPP, and so on. To keep track of the individual files in your project, you should give each file a meaningful name as you first create it in your project. Whenever you begin a new project, or any time you create a new form or unit file for your project, your first task should be to name the new files.

Along with custom file names, you should store each project in its own directory. Projects can share forms, files, and resources located in almost any directory, but it's best to keep the central project file and any other files specific to the project in a dedicated directory.

To read these topics in sequence, press the >> button.

Saving all open project files

See also

As you create new units and forms in a project, C++Builder supplies all associated files with the name of Unit1, Unit2, Unit3, and so on. C++Builder also supplies a default name for all the files associated with the project, Project1.MAK, Project1.CPP, and Project1.RES.

Saving a new project

When saving a new project, you must give unique names to the project files and the unit files, the two most visible file sets in a C++Builder project.

To save all open project files, use one of the following methods:

- Choose File|Save All.
- Choose the Save All button on the C++Builder toolbar.
- Choose the Save Project command on the Project Manager context menu.

When you are saving a new project, C++Builder displays the Save As dialog box, which first prompts you to save any new unit files, then it prompts you for the name of your new project files.

Naming unit files

Before naming your first unit file, be sure to supply a unique directory name for your project, otherwise, C++Builder by default saves the project to the current working directory. You can give a new directory name here, C++Builder will create the directory you specify.

When you name a unit file, C++Builder actually names the three files associated with the unit: Unit.CPP, Unit.H, and Unit.DFM.

Note: If you created the project from an existing project template, you will name the new project directory during the project expert process.

Naming project files

After you name your application unit file(s), C++Builder prompts you to name the project file. This processing order ensures that all unit and form file names are correctly registered in the project file, and that the unit and project file names do not match. When you name the project file, C++Builder names the three associated files: Project.MAK, Project.CPP, and Project. RES.

The executable file you are creating depends on the name you give to the application's project file; when you build your project, C++Builder produces an executable file on disk with the same name as the project file, but with the extension .EXE or .DLL appended as appropriate.

All unit and project file names must be legal C++ identifiers. When the compiler looks for a unit or project file, it first searches for a file with the full name of the unit or project identifier. If it does not find that file, it will then search for a version of the identifier name, truncated to eight characters. This is for backward compatibility and for compatibility with file servers that only store short file names. You should not manually truncate your file names.

Saving an existing project

If you have previously saved a project, the Save All command saves to disk all open project files that you have modified since the last save.

If you have opened any new forms or units in the project since the last save, the Save <Filename> As dialog box appears and prompts you to give unique names those unit files before saving them. The project file is then updated to reflect the new unit names and any newly shared files that you have specified for the project to use.

To read these topics in sequence, press the >> button.

Saving individual project files

See also

You are not limited to saving a project as a whole. C++Builder enables you to save individual constituent files of a project, including saving a copy of a file to a different directory or under a different file name.

You can also save your project as a project template which adds it to the Object Repository so that you or others can reuse it. For more information, see "Adding projects to the Object Repository".

To save individual project files (or non-project files such as text files) that you have open in the Code editor:

1. Bring the file to the front of the Code editor by selecting its tab.
2. Press Ctrl+S, or choose File|Save, then choose OK.

   If this is the first time you've saved the file, you're prompted to name it before saving it.

Changing a file name

To save a file using a different file name or location,

1. Bring the file to the topmost level of the Code editor by selecting its tab.
2. Choose File|Save As.

   The Save <Filename> As dialog box appears.
3. Specify the new file name, or location, or both, and choose OK.

   C++Builder saves a copy of the file under the name and location you specify.

Note: When you modify the name of an existing project file, C++Builder uses the new name to include the file in your project. The older file still exists, but it will no longer be included in the project.

To read these topics in sequence, press the >> button.

Removing files from a project

See also

You can remove forms and units from a project at any point during project development, but you should only do so using the C++Builder environment--the removal process deletes the reference to the file from the project file and any references to associated form files.

To remove a unit from a project, open the project and choose any of these methods:

- In the Project Manager window, select the unit or units you want to remove then choose the Remove button on the toolbar or choose Remove File from the context menu.
- Choose the Remove File From Project button on the C++Builder toolbar.
- Choose Project|Remove From Project from the C++Builder main menu.

Although removing a file from the project ends its relationship with the project, C++Builder does not delete the file from disk.

Note: C++Builder will not let you remove the project (.CPP) file for a project.

Caution: Do not use Windows file management programs to delete C++Builder project files from disk until you have performed the preceding removal process in every C++Builder project that uses the files. Otherwise, the project file of each project using the deleted files retains references to them. When you open the project again, C++Builder will attempt to find the deleted files and display error messages for each file it cannot find. In addition, the Project Manager will display inaccurate information about the project's constituent files.

To read these topics in sequence, press the >> button.

Saving a copy of the project

See also

C++Builder enables you to save a separate version of the currently open project in a different disk directory. The File|Save Project As command initiates the process. However, because the open project might use shared files in addition to its own files, the Save Project As command saves only a copy of the project source file, the project makefile, and the project resource file to the new location.

Important:

No unit files are saved to the new location. When you open a project that has been copied, the Project Manager displays all units in the copied project as shared files; that is, none of the unit files reside in the project directory of the currently open project.

To save a copy of the project in a new disk location,

1. Choose File|Save Project As from the C++Builder menu bar to display the Save <projectname> As dialog box.
2. C++Builder prompts you for the new name/location for your project .MAK file. Select the directory where you want to copy the project files.
3. If you want to save the project file under a different name, enter the new name in the File Name edit box. If a project file with the same name exists in the directory you specify, you're prompted as to whether you want to overwrite the existing file.
4. Choose OK to complete the task.

   The project open in C++Builder is now the new copied project you just saved.

When you use the Save Project As command, C++Builder saves the project file, the project options file, and the project resource file under the name and/or new location you specify. C++Builder also saves any modified unit files (in their current location), so you won't be prompted to save these changes again when you close the project.

If you check the file list in the Project Manager, you will see that all the unit files in the currently open version of the project reside in a directory other than the current project directory (the files are shared). If you want separate copies of any of those files in the new project directory, you need to save them individually to the new location using File|Save As. (Also, see the next section, "Creating a backup of an entire project.")

Make sure you understand the relationship of your files when you copy projects. If you don't understand how the new project is using its files, you might run into problems later.

Caution: Do not use file management tools other than those in C++Builder to save a copy of a project to a new location since C++Builder maintains relationships between files.

To read these topics in sequence, press the >> button.

Creating a backup of an entire project

See also

Backing up a project can be a simple matter of copying directories, or it can involve some additional steps. This depends upon how your project directories are structured and whether the project shares files from outside its own directory tree.

C++Builder does not encode the actual project directory into the project file. Instead, it records the *relative* location of all the files in the project. If these files reside in subdirectories within the main project directory, all path information is relative, which makes backup easy. You could back up such a project by copying the entire directory tree to another location. If you open the project at the backup location, all the project files that reside within that structure are present, and the project will compile.

However, if your project uses files that reside outside the project directory tree (such as shared files), the project might or might not compile at the backup location. Check the Project Manager's file list to ensure the shared files are accessible from the backup location. If they are, the project will compile. If other backup processes already preserve these outside files, then there is probably no need to make separate backup copies of these files in the backup project directory.

To read these topics in sequence, press the >> button.

Using the Project Manager

See also

The C++Builder Project Manager gives you a high-level view of the form, unit files, resource, object, and library files listed in the project file. You can use the Project Manager to open, add, save, and remove project files. You can also use the Project Manager to access the Project Options dialog box, which lets you configure your default project settings.

The Project Manager is an invaluable tool if you share files among different projects since it lets you quickly find the location of each file in the project. This is also helpful when you are creating backups of all the files in your project.

To open the Project Manager window, choose View|Project Manager (you must have a project open in order to view the Project Manager).

To read these topics in sequence, press the >> button.

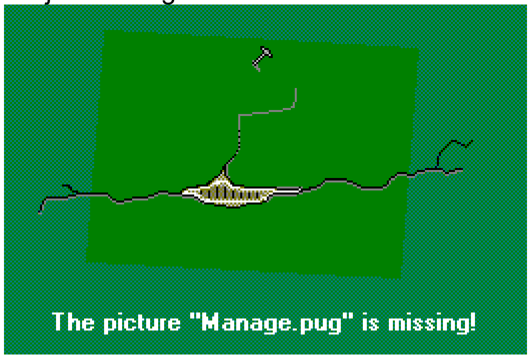
The Project Manager window

See also

The Project Manager window displays information about the status and file content of the currently open project. It also provides quick access to project management functions, lets you easily navigate among the project files, and gives you access to the project options through the toolbar and context menu.

It is highly recommended that you use the Project Manager to perform project related-tasks because it properly tracks and updates the effected files your project.

Project Manager window



The main elements of the Project Manager window are the

- The Project Manager file list
- The Project Manager toolbar
- The Project Manager status bar
- The Project Manager context menu

The Project Manager file list

The main area of the Project Manager details the file composition of the currently open project. The file list displays all the source files (which end with a .C, .CPP, .PAS, or .RC file extension) and binary object files (which can have a .RES, .LIB, or .OBJ file extension) in your project. You can also add other file types to your C++Builder project, however, C++Builder will not handle these files in any special way. C++Builder gets the file information from the project .CPP file.

The Project Manager lists in **bold** the files that you have modified in the Code editor, but have not yet saved.

If you have manually modified the project .CPP source file, the information in the file list might be inaccurate. When you open the Project Manager, C++Builder compares the information in the project file to the last saved information for the Project Manager. If these are not synchronized, the contents of the Project Manager file list becomes dimmed, and the Update button on the toolbar becomes enabled so that you can synchronize the information.

Caution: C++Builder has mechanisms for automatically tracking the files that make up a project and for keeping the project file updated. Avoid editing project files manually unless you have a thorough understanding of this process and its ramifications. By editing a project file, you circumvent C++Builder's automated project management and risk maintaining inaccurate information about project components. Compilation failures and other problems can result.

The Project Manager toolbar

The toolbar has six buttons that provide quick access to common project tasks. The following table describes each button and its use.

Button	**context menu command**	**C++Builder menu command**	**Function**	**Comment**
Add	Add File	File|Add To Project	Adds a shared or non-shared file	Non-shared files reside in the

			to the project. The Path column of the Project Manager's file list reflects the location of any shared file.	project directory. Shared files still reside outside the project directory; they are not copied to the current project directory. Any changes to the shared file, in any project, are reflected in all projects using the file.
Remove	Remove File	File|Remove From Project	Removes the selected file(s) from the current project.	Only the relationship of the selected file(s) with the current project is removed. The file still exists in its current location. The project file is updated to reflect the change.
View Unit	View Unit	View|Units (Ctrl + F12)	Opens and displays the selected file in the Code editor. If multiple files are selected in the Project Manager window, displays the most recently selected file.	Using the C++Builder menu command opens the View Unit dialog box, where you can select which unit to open.
View Form	View Form	View|Forms (Shift + F12)	Displays the visual image of the currently selected form unit. If the selected file is not a form unit, this button and the context menu command are disabled.	Using the C++Builder menu command opens the View Form dialog box, where you can select which form to display.
Options	Options	Project|Options	Displays the Project Options dialog box.	
Update	Update		Synchronizes the Project Manager window display with listings in the project .CPP source code file.	This button normally remains disabled, and its parallel context menu command dimmed, unless you have manually modified the project file (which is not recommended).

The Project Manager status bar

The area at the bottom of the Project Manager window displays the full path name of the project .MAK file, and indicates the number of forms and units in the project.

The project file path name can be a useful reference if you are bringing many forms and units that reside in locations other than the main project directory into the current project. The form and unit summary information can be helpful in evaluating the scope of proposed project modifications.

The Project Manager context menu

The Project Manager has a context menu that you access by right-clicking anywhere inside the Project Manager window. In addition to menu commands for all the toolbar commands, the context menu has commands for adding objects to the Object Repository, navigating through the project, and saving the project.

To read these topics in sequence, press the >> button.

Integrating forms and units into a project

See also

You can use the Project Manager (or commands on the File menu) to add new forms or unit files to a project. You can also add existing files from locations outside the project directory, a process known as *sharing* files.

Before you can add units to a project, you must have that project open. This way, C++Builder can update the project file with the new or shared files you add.

Adding new unit files

Whether you're adding a new form unit or a stand-alone unit file, the process is roughly the same:

- From the Project Manager context menu (or the File menu), choose New Form or New Unit, depending on the type of unit files you want to add to your project.

  C++Builder adds the new unit header files to the currently open project.

Sharing files from other projects or directories

A project can share any existing form and unit file that resides outside the project directory tree.

If you add a shared file to a project, bear in mind that the file is not copied to the current project directory; it remains in its current disk location. When you add the shared file to your project, C++Builder registers the file name and the path in the project file.

Note: The path that C++Builder uses for the shared file is either absolute or relative, depending on where the file is located. If the shared file is located on the same disk drive as your project, the C++Builder uses a relative path for the file; otherwise, it uses an absolute path.

When you compile your project, it does not matter whether the files that make up the project reside in the project directory, a subdirectory of that, or any other directory on disk; the compiler treats shared files the same as those created by the project itself.

To add a shared file to the current project, do one of the following:

- Choose the Add File button on the C++Builder toolbar.
- Choose the Add button on the Project Manager toolbar.
- Choose Add File from the Project Manager context menu.

Any of these actions displays the Add To Project dialog box, in which you can select the file you want the current project to use. The Path column of the Project Manager's file list displays the path to the shared file.

Using Borland C++, C or Pascal source code units

If you have existing source code units for custom procedures or functions written in Borland C++, C or Pascal you can use these units in a C++Builder project. You add these files in the same way as files created in C++Builder.

Note: Because C++Builder cannot set file-specific options (the project options you set affect all the source files in your project), you might need to take an extra step to include routines that require specific compiler or linker settings. To do so, you must compile the desired module to a binary format (as an .OBJ file, a .DLL file, or an .LIB file) outside of the project, then link or call the routines from there.

To read these topics in sequence, press the >> button.

Using the Project Manager to view forms and units

Perhaps the most useful features of the Project Manager are the commands that let you quickly navigate to the source code and form images contained in your projects.

To view a specific form, double-click on the form listing in the Project Manager; C++Builder gives focus to that form image. Double-clicking a unit listing in the Project Manager opens the Code editor and displays the selected unit source file. If the file is not currently open, C++Builder opens it for you.

In addition, you can also use the Project Manager to navigate in the following ways:

- In the Project Manager, select the unit or form you want to view or edit, then,
- Press the Enter key (press Shift+Enter to view a form image)
- On the toolbar, choose the View Unit or View Form button.
- Choose View Unit or View Form from the Project Manager context menu.

To read these topics in sequence, press the >> button.

Navigating among project components

See also

As you work on a project, you will find that you frequently need to navigate back and forth among forms, units, and the various open windows such as the Project Manager, Alignment palette, and so on.

You can easily toggle between viewing the currently displayed form and its associated unit source code using commands from the View menu, a keyboard shortcut, or the Project Manager.

You can also open or edit any open unit or form by choosing commands from the View menu, buttons on the Project Manager toolbar, or commands from the Project Manager context menu.

To read these topics in sequence, press the >> button.

Toggling between form image and unit source code

To switch between viewing the current form and its unit source code, use any of the following methods:

- Press F12.
- Choose View|Toggle Form/Unit.
- Click the Toggle Form/Unit speed button on the C++Builder toolbar.
- In the Project Manager, double-click either the File column to display a unit's source code, or the Form column to display the form image.

To read these topics in sequence, press the >> button.

Bringing a window to the front

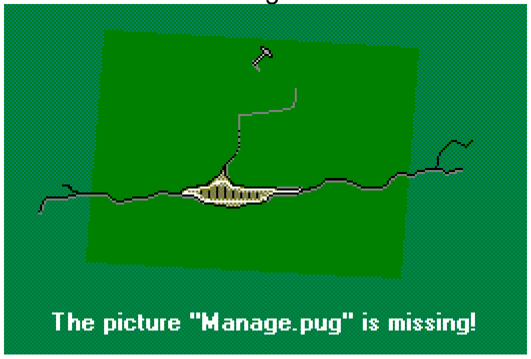
See also

If you have a number of windows open, such as the C++Builder Project Manager or Object Browser, you can easily get to the window you want by selecting it from the Window List dialog box. This dialog box displays a list of all open windows, and enables you to bring any of them to the front.

To bring a window to the front,

1. Press Alt+0 (zero) or choose View|Window List to access the Window List dialog box.
2. Double-click the name of the window you want to bring to the front.

The Window List dialog box



To read these topics in sequence, press the >> button.

Using the Object Repository

See also

C++Builder's Object Repository lets you share forms, dialog boxes, and data modules across projects. It can also help with reusing similar forms in a single project, and provides project templates as starting points for new projects.

This section focuses on how to use the Object Repository as a general project management tool and discusses the mechanics of using project templates. The main topics in this section are

- About the Object Repository
- Using Object Repository items
- Using project templates
- Customizing the Object Repository
- Using the Object Repository in a shared environment

To read these topics in sequence, press the >> button.

About the Object Repository

See also

C++Builder provides the Object Repository as a means for sharing and reusing objects within and across your projects. You access the items stored in the Object Repository through the New Items dialog box. Choose File|New to access the New Items dialog box:

The New Items dialog box

The New Items dialog box contains five tabs by default. It also contains a tab for the current open project.

Tab	Description
New	Offers experts and built-in objects from which you create new applications, components, forms, and so on. These objects are generated by C++Builder, and are not part of the Object Repository.
<Project>	Contains the unit and form objects contained in the currently open project
Forms	Contains a set of general-purpose forms.
Dialogs	Contains a set of general-purpose dialog boxes
Data modules	Contains a set of general-purpose data modules based on BCDEMOS, the sample database shipped with C++Builder
Projects	Contains a set of general-use project templates that you can use to begin your new applications

Sharing objects across projects

Not only can you use the New Items dialog to add forms, dialog boxes, and data modules to your project, you can also use it to add your own objects to the repository itself. By adding your own custom objects to the Object Repository, you make those objects available to other projects through the New Items dialog box.

For a simple case, you could have all your projects use the same About box, which they copy from the Object Repository. A more advanced use of the Object Repository would be to have a standard empty dialog box with the company or product logo and standard button placement, from which all your projects derive standard-looking dialog boxes.

Sharing forms within projects

The Object Repository can also help you share forms within a single project since it allows you to inherit from forms already in the project. When you open the New Items dialog box (by choosing File|New), you'll see a page tab with the name of your currently-open project. If you click that page tab, you'll see all the forms, dialog boxes, and data modules in that project. You can then derive a new item from any of the existing items, and customize it as needed.

For example, in a database application you might need several forms that display the same data, but which provide different command buttons. Instead of creating and maintaining several nearly-identical forms, you could lay out a generic form that contains all the data-display controls, then create separate forms that inherit the data-display layout, but have different command buttons.

By carefully planning your project forms, you can save tremendous amounts of time and effort by sharing forms within projects.

Sharing entire projects

You can also add an entire project to the Object Repository as a template for future projects. If you have a number of similar applications, you can base them all on a single, standardized model.

Using experts

The Object Repository contains references to experts. *Experts* are small applications that lead the user through a series of dialog boxes to create a form, project or other object in your application. C++Builder provides a number of experts (such as a new component expert, automation expert, and thread expert) and you can create and add experts of your own.

To read these topics in sequence, press the >> button.

Using Object Repository items

See also

There are three different ways which you can *share* items from the Object Repository. You can:

- Copying items from the Object Repository
- Inheriting items from the Object Repository
- Using items directly from the Object Repository

The sharing methods available to you are determined by the type of object you are sharing. Keep in mind that items in the Object Repository are there to be shared, and that you want to use them in ways that help rather than hinder their reuse.

Copying items from the Object Repository

The simplest sharing option is to copy an item from the Object Repository. Copying makes an exact duplicate of the item from the repository, and adds the copy to your project. Future changes to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original item in the Object Repository.

Copying is the only option available for using project templates. In addition, the Copy option is the only share option available for experts, whether form experts or project experts. With this, experts don't add shared code, but rather they run a process that generates a separate stand-alone copy of the code.

Inheriting items from the Object Repository

Inheriting is the most flexible and powerful Object Repository sharing option. Inheriting derives a new class from the repository item, and adds the new class to your project. When you recompile your project, any changes made to the item in the Object Repository will be reflected in your derived class, unless you have overridden a particular aspect of the object. In addition, changes made to your derived class are not reflected in the parent item in the Object Repository.

Inheriting is available for forms, dialog boxes, and data modules, but not for project templates. Inheriting is the *only* sharing option available for reusing items from within the same project.

Using items directly from the Object Repository

The least flexible sharing option is directly using an item from the Object Repository. When you add an item directly to a project, all design-time changes made to that object appear in *all* objects of that type, including all similar objects in other projects.

While sharing items directly is available for forms, dialog boxes, and data modules, you should use this type of sharing option sparingly. Items shared directly should generally be modified only at run time. This will avoid making changes that affect other projects.

To read these topics in sequence, press the >> button.

Using project templates

See also

C++Builder provides several project templates, which are predesigned projects that you can use as starting points for your own projects. Project templates are part of the Object Repository, and can be found on the Projects page of the New Items dialog box.

When you start a project from a project template, C++Builder prompts you for a project directory, a subdirectory in which to store the new project's files. If you specify a directory that doesn't currently exist, C++Builder creates the directory for you. C++Builder copies the template files to the project directory, from where you modify them to create a custom application. You can modify the project by adding new forms and units, or can use it unmodified by adding only your own event-handler code. In any case, the changes you made affect only the open project; the original project template remains unchanged and can be used again to start another new project.

To start a new project from a project template,

1. Choose File|New to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog box, specify a directory for the new project's files.

   A copy of the selected project opens in the specified directory.

Adding projects to the Object Repository

You can add your own projects and forms to those already available in the Object Repository. This is helpful in situations where you want to enforce a standard framework for programming projects throughout an organization.

For example, suppose you develop custom billing applications. You might have a generic billing application project that contains the forms and features common to all billing systems. Your business centers around adding and modifying features in this application to meet specific client requirements. In such a case, you might want to save the project containing your Generic Billing application as a project template and perhaps specify it as the default new project on your C++Builder development system. Likewise, you'll probably have a particular form within this project that you want to appear as the default main or new form.

To add a project to the Object Repository,

1. Open the project you want added to the Object Repository.
2. Choose Project | Add to Repository to invoke the "Add to Repository" dialog.
3. In the Title edit box, enter a project title.

   The title for the template will appear in the Object Repository window.
4. In the Description field, enter text that describes the template.

   This text will appear in the Object Repository window's status bar.
5. In the Page field, choose the name of the page in the New Items dialog box (probably Projects) you want the template to appear on.
6. In the Author field, enter text identifying the author of the application.

   Author information appears only when the user views the repository items with full details.
7. Choose Browse to select an icon to represent this template in the Object Repository.
8. Choose OK to save the current project as a project template.

Note: If you later make changes to a project template, those changes automatically appear in new projects created from that template. They will not, however, affect projects already created from that template.

Saving form templates

You can also save your own forms as form templates and add them to those already available in the Object Repository. This is helpful in situations where you want to develop standard forms for an organization's software, or if you create a form that you will use across several projects.

To add a form to the Object Repository as a template right click on the form and choose "Add to Repository, then follow the preceding steps for adding projects as templates, using the Forms page of the New Items dialog box instead of the Projects page.

To read these topics in sequence, press the >> button.

Customizing the Object Repository

See also

The settings in the Object Repository dialog box affect the behavior of C++Builder when you begin a new project or create a new form in an open project. You can also use the Object Repository to specify the following system defaults:

- Specifying the default new project
- Specifying the default new form
- Specifying the default main form

For example, by default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options.

Even though you can set the defaults for your projects and forms, you always have the option to override these defaults by choosing File|New and selecting any project or form from the New Items dialog box.

The Object Repository dialog box

Specifying the default new project

The default new project opens whenever you choose File|New Application. If you haven't specified a default project, C++Builder creates a blank project with an empty form.

You can specify a project template (including a project you have created and saved as a template) as the default new project. You can also designate a project expert to run by default when you start a new project, which enables you to build a project based on your responses to a series of dialog boxes.

To specify the default new project,

1. Choose Options|Repository to display the Object Repository dialog box.

   The repository itself is really just a text file that contains references to forms, projects, and experts. You can find details of the repository file format in online Help.
2. Choose Projects in the Pages list to view the currently stored project templates.
3. Select the project object you want as the default new project from the Objects list.
4. With the object you want selected, check New Project.
5. Choose OK to register the new default setting.

Specifying the default new form

The default new form opens whenever you choose File|New Form or use the Project Manager to add a new form to an open project. If you haven't specified a default form, C++Builder uses a blank form. You can specify any form template, including a form you have created and saved as a template, as the default new form. Or you can designate a form expert to run by default when a new form is added to a project.

To specify the default new form,

1. Choose Options|Repository to display the Object Repository dialog box.
2. Choose Forms in the Pages list to view the currently stored forms.
3. Select the form object you want as the default new form.
4. With the object you want selected, check New Form.
5. Choose OK to register the new default setting.

Specifying the default main form

Just as you can specify a form template or expert to be used whenever a new form is added to a project, you can also specify a form template or expert that should be used as the default main form whenever you begin a new project.

To specify the default main form for new projects,

1. Choose Options|Repository to display the Object Repository dialog box.
2. Choose Forms in the Pages list.

3. Select the form object you want as the default main form.
4. With the object you want selected, check Main Form.
5. Choose OK to register the new default setting.

To read these topics in sequence, press the >> button.

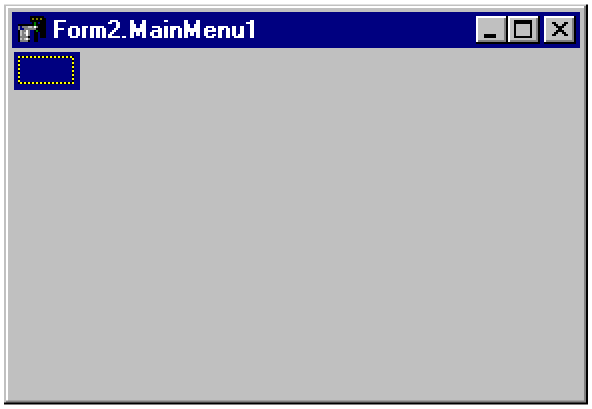
Using the Object Repository in a shared environment

See also

The Object Repository, by default, is stored in a file titled TEMPLATE.DRO. To change the location where C++Builder looks for the Object Repository file (when you want to share a repository between users), create a new String Value called "Base dir" in the "HKEY_CURRENT_USER|Software\Borland\CBuilder\1.0\Repository" key in the Windows Registry Editor and set its data Value to the directory where this file is to be located. We recommend that you use a UNC name for the location where the TEMPLATE.DRO file is to be shared. In addition, it is also suggested that Forms and Projects be saved using UNC names whenever they will be added to a shared Repository.

About the Menu designer

See Also



The Menu designer enables you to easily add menus to your form. You can simply add menu items directly into the Menu designer window. You can add, delete, and rearrange menu items at design time and you do not have to run the program to see the results. Your applications menus are always visible on the Form, as they will appear during runtime.

You can build each menu structure entirely from scratch, or you can start from one of the Menu templates (predesigned menus). You can also dynamically change menus, to provide more information or options to the user.

For more information about the Menu designer, click See Also at the top of this topic.

Opening the Menu designer

See Also

To open the Menu designer,

1. Place a MainMenu or a PopupMenu component on the form.
2. Leaving the component selected, choose from one of the following methods:

- Double-click the MainMenu or PopupMenu component.
- Click the ellipses button in the Values column for the Items property.
- Select Menu designer from the component's context menu.

Naming menus

See Also

When you add a menu component to the form, it has a default name, for example, MainMenu1. You can give the menu a more meaningful name that follows C++ naming conventions.

The menu name is added to the form's type declaration, and then it appears in the Component list.

Naming menu items

See Also

In contrast to the menu component itself, you need to explicitly name menu items while adding them to the form. You can do this in one of two ways:

- Directly type the value for the Name property.
- Type the value for the Caption property first, and let C++Builder derive the Name property from the caption.

For example, if you give a menu item a Caption property value of File, C++Builder assigns the menu item a Name property of File1. If you specify the Name property before the Caption property, C++Builder leaves the Caption property blank until you type a value.

Note: If you enter characters in the Caption property that are not valid for C++ identifiers, C++Builder modifies the Name property accordingly. For example, to start the caption with a number, C++Builder precedes the number with a character to derive the Name property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

Component name	**Derived caption**	**Explanation**
&File	File1	Removes ampersand
&File (second occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numeric order
1234 (second occurrence)	N12342	
$@@@#	N1	Removes all non-standard characters, adding preceding letter and numeric order
- (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, C++Builder adds any menu item names to the form's type declaration, and those names then appear in the Component list.

Adding menu items

See Also

To add menu items,

1. Open the Menu designer.
2. Select the position where you want to create the menu item.
3. Type the Name and press Enter. C++Builder automatically changes the caption of the menu item to reflect the name.
4. Press Enter.

   The next placeholder for a menu item is selected.
5. Enter values for the Name properties for each new item you want to create, or press Esc to return to the menu bar.

   Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press Enter to complete an action.

To add a separator bar to a menu, enter a hyphen as the caption of the menu item.

Inserting a menu item

See Also

To insert a menu item into a menu, place the cursor on a menu item, then press the Insert key.

Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

Deleting a menu item

See Also

1. Place the cursor on the menu item you want to delete.
2. Press the Delete key.

Note: You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in the menu at runtime.

Specifying accelerator keys

See Also

Add an ampersand (&) in front of the appropriate letter in the caption. The letter after the ampersand appears underlined in the menu.

Accelerator keys let the user access a menu command from the keyboard by pressing Alt+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Caution: C++Builder does not check for duplicate accelerators. You must track values you have entered in your application menus.

Specifying keyboard shortcuts

See Also

Enter a value for the ShortCut property, or select a key combination from the drop-down list. However, this list contains a subset of the valid combinations you can use.

Keyboard shortcuts enable the user to perform the action without accessing the menu directly by typing the shortcut key combination.

Caution: C++Builder does not check for duplicate shortcut keys, you must track values you have entered in your application menus.

Creating nested menus

See Also

1. Select the menu item under which you want to create a nested menu.
2. Press Ctrl+Right arrow to create the first placeholder, or choose Create Submenu from the context menu.
3. Enter a name for the nested menu item.
4. Press Enter to create the next placeholder.
5. Repeat steps 3 and 4 for each item you want to add to the nested menu.
6. Press Esc to return to the previous menu level.

Organizing your menu structure using nested menus can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one such nested level, if any.)

Shortcut: You can also create a nested menu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact nested menu. This pertains to nested menus as well; moving a menu item into an existing nested menu just creates one more level of nesting.

Moving menu items

See Also

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or onto a different menu entirely.

You cannot demote a menu item from the menu bar onto its own menu; nor can you move a menu item into its own nested menu. However, you can move any item onto a different menu.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.
2. Release the mouse button to drop the menu item at the new location.

   Note: When you move a menu item to a different place on the menu bar, all the sub-items beneath it move as well.

To move a menu item into a menu list,

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.

   Note: You cannot move a menu item down a level into its own menu.
2. Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

Moving menu items into submenus

When you move a menu item off the menu bar, its sub-items become a submenu. Similarly, if you move a menu item into an existing submenu, its sub-items then form another nested menu under the submenu.

You can move a menu item into an existing submenu, or you can create a placeholder at a nested level next to an existing item, and then drop the menu item into the placeholder to nest it.

Viewing the menu

See Also

You can view your menu at design time without running the application. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu designer to view a pop-up menu at design time.)

To view the menu,

1. If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.
2. If the form has more than one menu, select the menu you want to view from the form’s Menu property drop-down list.

The menu appears in the form exactly as it will when you run the program.

Editing menu items without opening the Menu designer

See Also

When you edit a menu item using the Menu designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list and edit its properties without ever opening the Menu designer.

To edit a menu item without opening the Menu designer, select the item from the Component list.

To close the Menu designer window and continue editing menu items,

1. Switch focus from the Menu designer window to the Object Inspector by clicking the properties page of the Object Inspector.
2. Close the Menu designer.

   The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item.

   To edit another menu item, select it from the Component list.

Switching among menus at design time

See Also

If you are designing several menus for your application, you can use the Menu designer context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch among menus in a form,

1. Right-click the Menu designer to display the context menu.
2. From the context menu, choose Select Menu.

   The Select Menu dialog box appears. This dialog box lists all the menus associated with the form whose menu is currently open in the Menu designer.
3. From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch among menus in a form,

1. Give focus to the form whose menus you want to choose from.
2. From the Component list, select the menu you want to edit.
3. On the Properties page of the Object Inspector, select the Items property for this menu, and then either click the ellipsis button (...), or double-click [Menu].

Using menu templates

See Also

C++Builder provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

Menu templates are stored in the file BCB.DMT. The menu templates shipped with C++Builder also reside in this file.

You can also save as a template any menu that you design using the Menu designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

To add a menu template to your application,

1. Right-click the Menu designer window.

   The Menu designer context menu appears.

2. From the context menu, choose Insert From Template.

   (If there are no templates, the Insert From Template option is dimmed.)

   The Insert Template dialog box opens, displaying a list of available menu templates.

3. Select the menu template you want to insert, then press Enter or choose OK.

   This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1. Right-click the Menu designer window.

   The Menu designer context menu appears.

2. From the context menu, choose Delete Templates.

   (If there are no templates, the Delete Templates option appears dimmed in the context menu.)

   The Delete Templates dialog box opens, displaying a list of available templates.

3. Select the menu template you want to delete, and press Del.

   C++Builder deletes the template from the templates list and from your hard disk.

Saving a menu as a template

See Also

Any menu you design can be saved as a template so you and others can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

Menu templates you save are stored in your BIN directory as .DMT files.

You edit the template file by using the template commands from the Menu designer context menu.

To save a menu as a template,

1. Choose Save As Template from the Menu designer context menu to open the Save Template dialog box.
2. In the Template Description edit box, enter a brief description of this menu.
3. Click OK.

   The Save Template dialog box closes, saving your menu design and returning you to the Menu designer window.

Note: The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the Name or Caption property for the menu.

When you save a menu as a template, C++Builder does not save its Name, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu designer, C++Builder then generates new names for it and all its items.

C++Builder also does not save any event handlers associated with a menu saved as a template, since C++Builder cannot test whether the code would be applicable in a new form. You can associate menu items in the template with existing event handlers in the form.

Naming conventions for template menu items and event handlers

See Also

When you save a menu as a template, C++Builder does not save its Name property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu designer, C++Builder then generates new names for it and all its items.

For example, suppose you save a File menu as a template. In the original menu, you name it MyFile. If you insert it as a template into a new menu, C++Builder names it File1. If you insert it into a menu with an existing menu item named File1, C++Builder names it File2.

C++Builder also does not save any OnClick event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, C++Builder still generates the event handler name.

You can easily associate items in the menu template with existing OnClick event handlers in the form.

Adding menu items dynamically

See Also

You can add menu items to an existing menu structure while the application is running to provide more information or options to the user.

- Insert a menu item by using the menu item's Add or Insert method.
- Alternately hide and show the items in a menu by changing their Visible property. The Visible property determines whether the menu item is displayed in the menu.
- Dim a menu item without hiding it by using the Enabled property.

In Multiple Document Interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. See Merging Menus.

To insert a menu item into an existing menu,

1. Create an event handler for the event you want to respond to.
2. Declare the menu item as a variable of type TMenuItem.
3. Write a procedure that adds the menu item to an existing menu.

Example

The following example adds a new menu item to the Window menu (WindowMenu) when Button1 is clicked.

```
void __fastcall TForm1::::Button1Click(TObject *Sender)
{
  TMenuItem *NewItem
  NewItem= new TMenuItem(WindowMenu);      //Creates the new menu item
  NewItem ->Caption = "My Menu Command";  //Caption for the new menu item
  WindowMenu->Insert(1, NewItem);          //Inserts the new menu item
}
```

Merging menus

See Also

For Multiple Document Interface (MDI) and Single Document Interface (SDI) applications, the application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is called merging menus.

You prepare menus for merging by specifying values for two properties:

- Menu, a property of the form
- GroupIndex, a property of menu items in the menu

Specifying the active menu: Menu property

The Menu property specifies the active menu for the form. Merge operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the Menu property in code. For example,

```
Form1->Menu = SecondMenu;
```

Determining the order of merged menu items: GroupIndex property

The GroupIndex property determines the order in which the merging menu items appear in the shared menu bar. Merged menu items can replace those on the main menu bar or can be inserted.

The default value for GroupIndex is 0. Several rules apply when specifying a value for GroupIndex:

- Lower numbers appear first (farther left) in the menu.
  For instance, set the GroupIndex property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.
- To replace items in the main menu, give items on the child menu the same GroupIndex value.
  This can apply to groups or to single items. For example, if your main form has an Edit menu item with a GroupIndex value of 1, you can replace it with one or more items from the child form's menu by giving them a GroupIndex value of 1 as well.
  Giving multiple items in the child menu the same GroupIndex value retains their order intact when they merge into the main menu.
- To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.
  For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

Additional rules apply for OLE client applications.

Importing menus from resource files

See Also

The Menu designer supports menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your project, saving you the time and effort of rebuilding menus that were created elsewhere.

To load an existing .RC menu file,

1. In the Menu designer, place the cursor where you want the menu to appear.

   The imported menu can be part of a menu you are designing, or an entire menu in itself.
2. Right click and choose Insert From Resource.

   The Insert Menu From Resource dialog box displays.
3. Select the resource file you want to load, and choose OK.

Note: If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

Working with the Code editor

See also

The Code editor lets you view and write the source code for your C++Builder applications. The Code editor is a full-featured, customizable editor that offers many powerful features, such as:

- Syntax highlighting
- Multiple and Group Undo
- Four predefined editor settings: Default, IDE Classic, Brief, and Epsilon
- Context-sensitive Help for language elements

This chapter introduces the major features of the Code editor. It describes how to navigate among multiple Code editor files and how to switch between your source code and the form editor. In addition, the following topics are discussed:

- Viewing files in the Code editor
- The Code editor context menu
- Searching in the Code editor
- Viewing components as code in the editor
- Customizing the editor

Viewing files in the Code editor

See also

The Code editor is a full-featured ASCII file editor. With it, you can edit any of your C++Builder source files and ASCII files, and you can open several files simultaneously. The Code editor represents each open file with a tab at the top of the editor window--you can easily switch between all the open files in the editor by clicking the appropriate file tab. When you click a tab, the editor displays the file in an editor *page*.

When you open a new project, C++Builder automatically generates a page in the Code editor for the initial form unit file. When you open a new form, unit, or other source file in your project, C++Builder adds a new page to the editor which contains the associated source code.

You can use any of the following methods to view a specific file in the Code editor:

- Choose File|Open to open an existing file.
- For already open files, click the tab for the file you want to view.
- Press Ctrl+Tab to move forward through the currently open files in the editor, press Shift+Ctrl+Tab to move backward through the editor pages.
- Choose View|Units to open the View Unit dialog box, and choose the unit you want to view.
- Use toolbar buttons to toggle between the current form and its associated unit, or to open the View Unit dialog box.
- Place the cursor on a file name within the editor, right click and choose Open file at Cursor.
- Choose View|Project manager and double click on a unit in the project window.

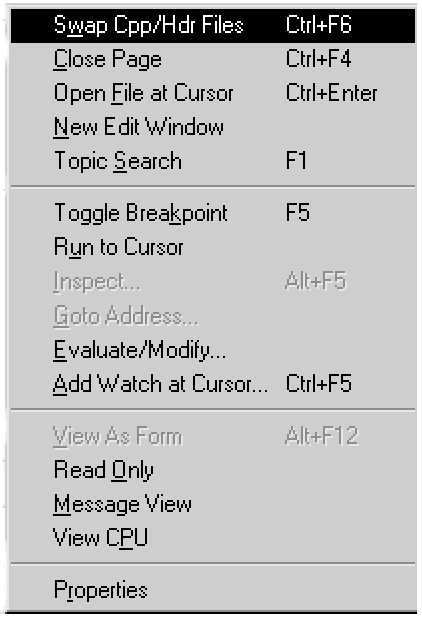
Note: Closing the Code editor by double-clicking the Control-menu box of the edit window closes all open files in the project. To close a single page, choose Close Page from the Code editor context menu.

The Code editor context menu

See also

Right-clicking in the Code editor gives you access to its context menu, which contains many commonly used edit commands, debugging commands, and commands that let you navigate through the C++Builder environment. For information on the commands contained on the context menu, select a menu item, and press F1 for context-sensitive Help.

The Code editor context menu.



Searching in the Code editor

See also

You can use the Search menu (located on the Main menu) to locate specific text in the Code editor. Use the Search menu commands to locate text, objects, units, variables, and symbols in the Code editor.

The commands in the Search menu are:

- *Find* searches for specific text.
- *Replace* searches for specific text and replaces it with new text.
- *Search Again* repeats a search.
- *Incremental Search* searches for text as you type.
- *Go to Line Number* moves the cursor to a specific line number.
- *Go to Address* becomes available when the CPU window is active; this command moves the Disassembly pane display to the address you specify.

To learn more about any of the commands on the Search menu, select the command and press F1.

Viewing components as code in the editor

See also

When you add a component to a form, C++Builder generates an instance variable for the component and adds it to the form's type declaration. Similarly, when you delete a component, C++Builder removes the corresponding type declaration from the source code. You can view similar code being added or removed from the Code editor.

To view code as C++Builder generates it

1. Move the form so you can view both the form and the Edit window (click the form's Title bar and drag it to a location that lets you see the entire Code editor).
2. Right click and choose Swap Cpp/Hdr files to switch to the .h file of the unit, and scroll in the Code editor until the form's type declaration section is visible.
3. Add a component to the form while watching what happens in the Code editor.
4. Delete the component, again while viewing the Code editor.

Note: C++Builder does not remove event handlers (or methods) associated with components you delete because those event handlers might be called by other components in the form. You can still run your program so long as the method declaration and the method itself both remain in the unit file. If you delete the method without deleting its declaration, C++Builder generates an error message.

Viewing the associated form

See also

When viewing and editing source code, you can easily switch between the code and the associated form to see how your design changes effect your code, and vice versa. Because C++Builder is a two-way tool, object declarations and definitions that you add to your source code are immediately reflected on the form that you add them to.

Use any of the following methods to view a project form:

- Click any part of the form that is visible under the Code editor to view the form associated with the currently displayed source file.
- Choose View|Form to open the View Form dialog box, then choose any form in the project that you want to view.
- Use toolbar buttons to toggle between the unit and form, or to open the View Form dialog box.

Note: If you view a form that is different from the one associated with the currently displayed code in the editor, the editor changes code pages to keep in sync with the selected form. Because of this, when you return to the Code editor, it might not display the code you were previously viewing.

Customizing the editor

See also

You can choose one of the four predefined editor settings. If you wish to, you can then customize the behaviors and appearance of that editor by:

- Selecting a default editor and customizing its keymapping
- Choosing colors for the editor
- Setting display and file saving options

Selecting a default editor

See also

Use the Editor SpeedSetting on the Editor page of the Options|Environment Options dialog box to select the default keymappings of your editor. These options are described in Table 7.1.

Option	Automatically sets
Default Keymapping	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Group Undo, Overwrite Blocks, Use Syntax Highlight
IDE Classic	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Cursor Through Tabs, Group Undo, Persistent Blocks, Use Syntax Highlight
BRIEF Emulation	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Cursor Through Tabs, Cursor Beyond EOF, Keep Trailing Blanks, BRIEF Regular Expressions, Force Cut And Copy Enabled, Use Syntax Highlight
Epsilon Emulation	Auto Indent Mode, Insert Mode, Smart Tab, Backspace Unindents, Cursor Through Tabs, Group Undo, Overwrite Blocks, Use Syntax Highlight

Once you choose a default keymapping, you can customize the behavior of the C++Builder editor using the other options on the Editor page. To learn more about the options on this page, press F1 or click Help while in the dialog box.

Choosing color settings for the editor

See also

Use the Colors page of the Options|Environment Options dialog box to specify how you want the different elements of your code to appear in the Code editor. The sample Code editor in this dialog box shows how your settings will appear in the C++Builder Code editor.

The Color Speed Settings configure the Code editor display according to predefined color combinations. After picking a predefined set of colors, you can further customize the display of your code; you can specify foreground and background colors and text attributes for anything listed in the Element list box.

Specifying display and file options

See also

Use the Display page of the Options|Environment Options dialog box to customize the display and font options for your Code editor. The sample window displays the selected font. The new settings take effect when you click OK.

Display and File check boxes allow you to configure the editor's display and choose whether to create backup files. Click the Help button in the dialog box to learn more about the available options.

Working with event handlers

See also

In C++Builder, almost all the code you write is executed, indirectly or directly, in response to events. Such code is called an *event handler*. Event handlers are actually specialized procedures, a form method attached to an event. The event handler executes when the particular event occurs.

In C++Builder you usually use the Object Inspector to generate event handlers. The Object Inspector not only generates the event handler name for you, but if you change that name later using the Object Inspector, C++Builder changes it everywhere that it occurs in your source code. This is not the case for event handlers you write without using the Object Inspector. You can write source code in the Code editor without using the Object Inspector. For example, you might write a general-purpose routine that's not associated directly with a component event, but that is called by an event handler.

This section discusses ways to use the Object Inspector and the Code editor to generate event handlers in your source code. The following topics are explained:

- Generating the default handler
- Writing an event handler
- Locating an existing event handler
- Associating an event with an existing event handler
- Coding menu events

Generating the default event handler

See also

The *default event* is the one the component most commonly needs to handle at runtime. For example, a button's default event is the *OnClick* event.

To generate a default event handler, double-click the component in the form.

Note: Certain components, don't need to handle user events. For these components, double-clicking them in the form doesn't generate an event handler. Double-clicking certain other components, such as the Image component or either of the menu components, opens a dialog box where you perform design-time property edits.

Writing an event handler

See also

Once you've placed a component on the form, you can easily write an event handler for it.

To write an event handler for a component

1. Select the component on the form.
2. Click the Events tab of the Object Inspector.
3. Choose from the list of available events for that component and double click the event you want to program.

   The Code editor displays the generated code for the event handler with the cursor positioned so you can start typing an event handler.
4. Type the source code for the event handler.

The example that follows demonstrates an event handler for a button's OnClick event. When the button is clicked the contents of a Memo component (Memo1) on the form (TViewerForm) is printed on the default printer.

```
#include <vcl/printer.hpp>
void __fastcall TViewerForm::PrintButtonClick(TObject *Sender)
{
   if (MessageBox (NULL, "Are you sure you want to print?", "Print",
        MB_YESNO) == IDYES)
   {
     int FontHeight, LinesOnPage;
     TCanvas *pcanvas;
      Printer()->BeginDoc();
      pcanvas = Printer()->Canvas;
      FontHeight = pcanvas->Font->Height;
      LinesOnPage = Printer()->PageHeight / -FontHeight;
      for (int i = 0, j = 0; i < Memo1->Lines->Count; i++, j++)
      {
         if ((i > 1) && (i % LinesOnPage == 0))
         {
            j = 0;
            Printer()->NewPage();
         }
           pcanvas->TextOut (10, ((j * -FontHeight) + 10),
           Memo1->Lines->Strings[i]);

      Printer()->EndDoc();
   }
}
```

To learn more about programming event handlers, search for event handlers in the Help index.

Locating an existing event handler

See also

If you want to view the code for an event handler that you have previously written, you can quickly locate it in the Code editor.

To locate an existing event handler in the Code editor

1. In the form, select the component whose event handler(s) you want to locate.
2. In the Object Inspector, display the Events page.
3. In the Events column, double-click the event whose code you want to view.

   C++Builder displays the Code editor, placing your cursor inside the code block of the event handler. You can now add your source code to this event handler.

   If you generated a default event handler for a component by double-clicking it in the form, you can locate that event handler in like fashion.

To locate an existing default event handler, double-click the component in the form. The Code editor opens, with your cursor at the start of the first line of code for the component's default event handler.

Associating an event with an existing event handler

See also

C++Builder emphasizes reuse at all levels of application development, from the project, to the form, to components and code. You can reuse code by writing event handlers that handle more than one component event.

You can do this by creating generic event handlers that, for example, are called by both a menu command and an equivalent button on a toolbar.

Once you write one such event handler, you can easily associate other compatible events with the original handler.

Writing an event handler for multiple component events

See also

The Sender parameter in an event handler informs C++Builder which component generated the event, and called the handler. You can write a single event handler that responds to multiple component events by using the Sender parameter in an if..else statement. However, you don't have to use Sender to share event code.

To associate a new component event with an existing event handler

1. In the form, select the component whose event you want to code.
2. Display the Events page of the Object Inspector, and select the event to which you want to attach handler code.
3. Click the down arrow next to the event to open a list of existing event handlers.

   The list shows only the event handlers in this form that can be assigned to the selected event.
4. Select from the list by clicking an event handler name.

   The code written for this event handler is now associated with the selected component event.

Note: C++Builder doesn't duplicate the event handler code for every component event associated with a shared handler. The same code is called whenever any of the associated component events occurs.

Displaying and coding shared events

See also

There is another way to share event handlers. When components have events in common, you can select the components to display their shared events, select a shared event, and then create a new event handler for it, or select an existing one. All the selected components will now use this event handler.

To display shared events

1. In the form, select all the components whose common events you want to view.
2. Display the Events page of the Object Inspector.

   The Object Inspector displays only those events that pertain to all the selected components. (Note also that only events in the current form are displayed.)

To associate a shared component event with an existing event handler

1. Select the components with which you want to associate a shared event handler.
2. Display the Events page of the Object Inspector, and select an event.

   The Object Inspector displays only those events which the selected components have in common.
3. From the drop-down list next to the event, select an existing event handler, and press Enter.

   Whenever any of the components you selected receives the specified event, the event handler you selected is called.

To create an event handler for a shared event

1. Select the components for which you want to create a shared event handler.
2. Display the Events page of the Object Inspector, and select an event.
3. Type a name for the new event handler and press Enter, or double-click the Handler column if you want C++Builder to generate a name.

   C++Builder creates an event handler in the Code editor, positioning the cursor on the first line of code.

   If you choose not to name the event handler, C++Builder names it for you based on the order in which you selected the components. For example, if you create a shared event handler for several buttons, C++Builder names the event handler Button<n>Click, where Button<n> is the first button you selected for sharing an event handler.
4. Type the code you want executed when the selected event occurs for any of the components.

Modifying a shared event handler

See also

Modifying an event handler that is shared by more than one component is virtually the same as modifying any existing event handler. Just remember that whenever you modify a shared event handler, you are modifying it for all the component events that call it.

To modify the shared event handler

1. Select any of the components whose event calls the handler you want to modify.
2. In the Events page of the Object Inspector, double-click the event handler name.
3. In the Code editor, modify the handler.

   Now when any of the components that share this handler receive the shared event, the modified code gets called.

Deleting event handlers

See also

When you delete a component, C++Builder removes its reference in the form's type declaration. However, deleting a component does *not* delete any associated methods from the unit, because those methods might be called by other components in the form.

You can still run your application so long as the method declaration and the method itself both remain in the unit. If you delete the method without deleting its declaration, C++Builder generates an "Undefined forward" error message, indicating that you need to either replace the method itself (if you intend to use it), or delete its declaration as well as the method code (if you do not intend to use it).

You can still explicitly delete an event handler, if you choose, or you can have C++Builder do it for you.

To manually delete an event handler, remove all event handler code *and* the handler's declaration.

You can also have C++Builder delete an event handler, simply remove all the code (including comments) inside the event handler code block. C++Builder removes all empty event handlers when you compile or save your projects.

Coding menu events

See also

While you use the C++Builder Menu designer to visually design your application menus, the underlying event code is what makes the menus useful. Each menu command needs to be able to respond to an OnClick event, and there are many times when you want to change menus dynamically in response to program conditions.

The following topics describe how to associate code with menu events. For specific code examples that demonstrate ways to dynamically modify menus in a running application, refer to Chapter 6, "Designing menus." For an example of how to write an event handler that displays a File Save dialog box, see "Hooking up an event handler" on page 337.

Menu component events

See also

The MainMenu component is different from most other components in that it doesn't have any associated events. You access the events for items in a main menu by means of the Menu designer.

In contrast, the PopupMenu component has one event: the OnPopup event. This is necessary because pop-up menus have no menu bar, therefore no *OnClick* event is available prior to users opening the menu.

Handling menu item events

See also

There is only one event for menu items: the OnClick event. Code that you associate with a menu item's OnClick event is executed whenever the user chooses the menu item in the running application, either by clicking the menu command or by using its accelerator or shortcut keys.

To generate an event handler for any menu item

1. From the Menu designer window, double-click the menu item.
2. Inside the code block, type the code you want to execute when the user clicks this menu command.

You can also easily generate event handlers for menu items displayed in a form. This procedure does not apply to pop-up menu items, as they are not displayed in the form at design time.

To generate an event handler for a menu item displayed in the form, simply click the menu item (not the menu component). For example, if your form contains a File menu with an Open menu item below it, you can click the Open menu item to generate or open the associated event handler.

This procedure applies only to menu items in a list, not those on a menu bar. Clicking a menu item on a menu bar opens that menu, displaying the subordinate menu items.

Associating a menu item with an existing event handler

See also

You can associate a menu item with an existing event handler, so that you don't have to rewrite the same code in order to reuse it.

To associate a menu item with an existing OnClick event handler

1. In the Menu designer window, select the menu item.
2. Display the Properties page of the Object Inspector, and ensure that a value is assigned to the menu item's Name property.
3. Display the Events page of the Object Inspector.
4. Click the down arrow next to OnClick to open a list of previously written event handlers.

   Only those event handlers written for OnClick events in this form appear in the list.
5. Select from the list by clicking an event handler name.

   The code written for this event handler is now associated with the selected menu item.

Writing an event handler for the Help|Memory Info menu

See also

The following example illustrates a menu handler for an item on the Help menu called "Memory Info." When the user selects this menu item, this handler sets several edit fields on a second form to show information on the current state of system memory.   After setting the edit fields, the handler then calls the second form's ShowModal function to display the form with the current memory state.

```
void __fastcall TForm1::Memory1Click(TObject *Sender)
{
  char buf[16];
  MEMORYSTATUS ms;
     ms.dwLength = sizeof(MEMORYSTATUS);
     GlobalMemoryStatus(&ms);
  wsprintf (buf, "%d", ms.dwTotalPhys);
  MemoryForm->TotalPhysicalEdit->Text = buf;
  wsprintf (buf, "%d", ms.dwAvailPhys);
  MemoryForm->FreePhysicalEdit->Text = buf;
  wsprintf (buf, "%d", ms.dwTotalPageFile);
  MemoryForm->TotalPagingEdit->Text = buf;
  wsprintf (buf, "%d", ms.dwAvailPageFile);
  MemoryForm->FreePagingEdit->Text = buf;
  wsprintf (buf, "%d", ms.dwTotalVirtual);
  MemoryForm->TotalVirtualEdit->Text = buf;
  wsprintf (buf, "%d", ms.dwAvailVirtual);
  MemoryForm->FreeVirtualEdit->Text = buf;
  MemoryForm->ShowModal();
}
```

To learn more about programming event handlers, search for event handlers in the Help index.

Programming with VCL objects

See also

Object-oriented programming (OOP) is a natural extension of structured programming. OOP requires that you use good programming practices and makes it very easy for you to do so. The result is clean code that is easy to extend and simple to maintain.

Once you create an object for an application, you and other programmers can then use that same object in other applications. Reusing objects can greatly cut your development time and increase productivity for yourself and others.

This chapter explains what you need to know about objects to use C++Builder effectively. If you want to create new components and put them on the C++Builder Component palette, see the *Component Writer's Guide.*

These are the primary topics discussed in this chapter:

- What is an object?
- Inheriting data and code from an object
- Object scope
- Assigning values to object variables
- Creating nonvisual objects

Before reading this chapter, you should know how to design a user interface for your application using the C++Builder tools, how a C++Builder project is structured, and how to handle events. To review topics in any of these areas, refer to the appropriate chapters in this book, or use C++Builder's online Help.

To read these topics in sequence, press the >> button.

What is an object?

See also

An object is an instance of a class that wraps up data and code all into one bundle. Before OOP, code and data were treated as separate elements.

Think of the work involved to assemble a bicycle if you have all the bicycle parts and a list of instructions for the assembly process. This is analogous to writing a Windows program from scratch without using objects. C++Builder gives you a head start on "building your bicycle" because it already gives you many of the "preassembled bicycle parts"--the C++Builder forms and components.

An object is a specific instance of a class. A class is a collection of related information that includes both data and functions (the instructions for working with that data). The properties of C++Builder objects have default values. You can change these values at design time to modify an object to suit the needs of your project without writing code. If you want a property value to change at run time, you need to write only a very small amount of code.

To read these topics in sequence, press the >> button.

Examining a C++Builder object

See also

When you choose to create a new project, C++Builder displays a new form for you to customize. In the Code editor, C++Builder declares a new class for the form and produces the code in a .cpp and .h unit file pair that creates the new form object. A later section discusses why a new class is declared for each new form. For now, examine the following example to see what the code in the Code editor looks like:

```
Unit1.h file:
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:       // IDE-managed Components
private:// User declarations
public:     // User declarations
  virtual __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
Unit1.cpp file
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
```

The new class is *TForm1*, and it is derived from *TForm*, which is also a class. A later section of this chapter presents more about *TForm* and objects derived from other objects.

So far, type *TForm1* appears to contain no data members or methods, because you haven't added to the form any components (the data members of the new object), and you haven't created any event handlers (the methods of the new object). As discussed later, *TForm1* does contain data members and methods, even though you don't see them in the class declaration.

This code declaration points to a variable named *Form1* of the new class *TForm1*.

```
TForm1 *Form1;
```

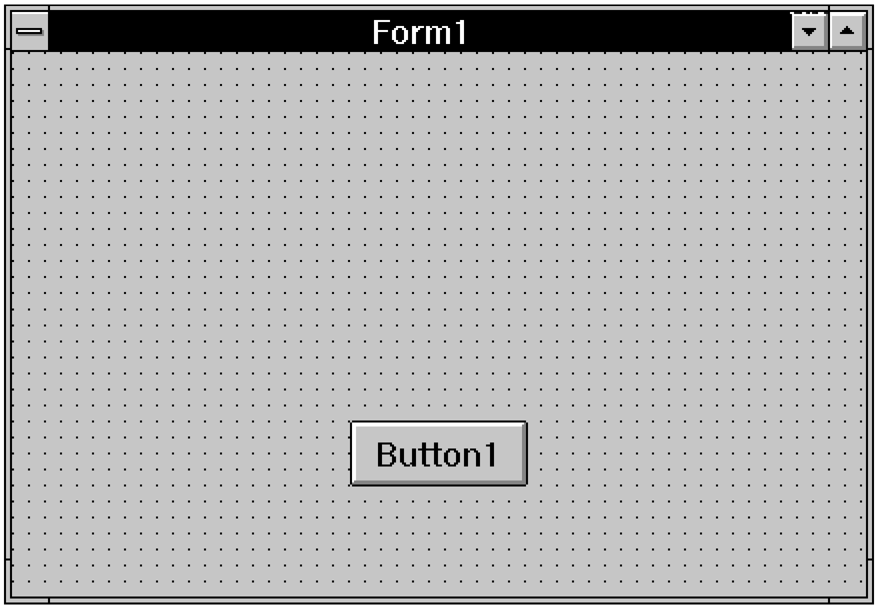
Form1 is called an instance of the type *TForm1*. The *Form1* variable refers to the form itself to which you add components to design your user interface.

You can declare more than one instance of a class. You might want to do this to manage multiple child windows in a Multiple Document Interface (MDI) application, for example. Each instance carries its own data in its own package, and all the instances of an object use the same code.

Although you haven't added any components to the form or written any code, you already have a complete C++Builder application that you can compile and run. All it does is display a blank form because the form object doesn't yet contain the data members or methods to do more.

Suppose, though, that you add a button component to this form and an *OnClick* event handler for the button, that changes the color of the form when the user clicks the button. You then have an application that's about as simple as it can be and still actually do something. Here is the form for the application:

A simple form



When the user clicks the button, the form changes color to green. This is the event-handler code for the button *OnClick* event:

```
void__fastcall TForm1::Button1Click(TObject *Sender)
{
 Form1->Color = clGreen;
}
```

If you create this application and then look at the code in the Code editor, there is both a Unit1.cpp and Unit1.h file. You can right-click and choose Swap Cpp/Hdr files to display the header file. You will see files like the following:

```
Unit1.h file:
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
  TButton *Button1;
  void__fastcall Button1Click(TObject *Sender);
private:// User declarations
public:    // User declarations
  virtual __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
Unit1.cpp file
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
}
```

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Form1->Color = clGreen;
}
//---------------------------------------------------------------------------
```

The new *TForm1* object now has a *Button1* data member--that's the button you added to the form. *TButton* is a class, so *Button1* is also an object. Classes, such as *TForm1*, can contain other objects, such as *Button1*, as data members. Each time you put a new component on a form, a new data member with the component's name appears in the form's class declaration in the unit header file (Unit1.h).

All the event handlers you write in C++Builder are methods of the form object. Each time you create an event handler, a method is declared in the form class. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared within the *TForm1* class declaration.

The actual code for the *Button1Click* method appears in the .cpp part of the unit file pair. The method is the same as the empty event handler you created with C++Builder and then filled in to respond when the user clicks the button as the application runs.

To read these topics in sequence, press the >> button.

Changing the name of a component

See also

You should always use the Object Inspector to change the name of a component. For example, when you change the default name of a form from *Form1* to something else by changing the value of the *Name* property using the Object Inspector, the name change is reflected throughout the code C++Builder produces. If you wrote the previous application, but named the form *ColorBox*, this is how your code would appear:

```
Unit1.h file:
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TColorBox : public TForm
{
__published:       // IDE-managed Components
  TButton *Button1;
  void__fastcall Button1Click(TObject *Sender);
private:// User declarations
public:    // User declarations
  virtual __fastcall TColorBox(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TColorBox *ColorBox;
//---------------------------------------------------------------------------
#endif
Unit1.cpp file
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TColorBox *ColorBox;
//---------------------------------------------------------------------------
__fastcall TColorBox::TColorBox(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void__fastcall TColorBox::Button1Click(TObject *Sender)
{
  Form1->Color = clGreen;
}
//---------------------------------------------------------------------------
```

Notice that the name of the form class changed from *TForm1* to *TColorBox*. Also note that the form variable is now *ColorBox*, the name you gave to the form. References to the *ColorBox* variable and the *TColorBox* type also appear in the final block of code, the code that creates the form object and starts the application running. If C++Builder originally generated the code, it updates it automatically when you use the Object Inspector to change the name of the form or any other component.

Note that the code you wrote in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form if you change the form's name. If you don't, your code won't be compiled. In this case, change the code to this:

```
__fastcall TColorBox::Button1Click(TObject *Sender);
{
 ColorBox->Color = clGreen;
}
```

You should change the name of a component only with the Object Inspector. While nothing prevents you from altering the code C++Builder produced and changing the name of component variables, you won't be able to compile your program.

To read these topics in sequence, press the >> button.

Inheriting data and code from an object

See also

The TForm1 object might seem quite simple. If you create the application, *TForm1* appears to contain one data member, *Button1*, one method, *Button1Click*, and no properties. Yet you can change the size of the form, add or delete minimize and maximize buttons, or set up the form to become part of a Multiple Document Interface (MDI) application. How is it possible to do all these things with a form that contains only one data member and one method?

The answer lies in the notion of inheritance. Recalling the bicycle analogy, once you have put together all the objects that make up a complete "bicycle object," you can ride it because it has all the essentials to make the bicycle useful--it has pedals you push to make the wheels go around, it has a seat for you to sit on, it has handlebars so you can steer, and so on. Similarly, when you add a new form to your project, it has all the capabilities of any form. For example, all forms provide a space to put other components on them, all forms have the methods to open, show, and hide themselves, and so on.

Suppose, though, that you want to customize that bicycle, just as you would customize a form object in C++Builder. You might adjust a few gear settings, add a headlight, and provide a horn with a selection of sounds--just as you might customize a form by adding or rearranging buttons, changing a few property values, and adding a new method that allows the form to appear with a plaid background.

To change the bicycle to make it exactly as you want it, you start with the basic model and then customize it. You do the same thing with C++Builder forms. When you add a new form to your project, you've added the "basic model" form. By adding components to the form, changing property values, and writing event handlers, you are customizing the new form.

To customize any object, whether it be a blank form, a form with multiple controls that is used as a dialog box, or a new version of the C++Builder bitmap button, you start by deriving a new class from an existing class. When you add a new form to your project, C++Builder automatically derives a new form object for you from the *TForm* class.

At the moment a new form is added to a project, the new form class is identical to the *TForm* class. Once you add components to it, change properties, and write event handlers, the new form object and the *TForm* class are no longer the same.

No matter how you customize your bicycle, it can still do all the things you expect a bicycle to do. Likewise, a customized form class still exhibits all the built-in capabilities of a class, and it can still do all the things you expect a form to do, such as change color, resize, close, and so on. That's because the new form object *inherits* all the data members, properties, methods, and events from the *TForm* class.

When you add a new form to a C++Builder project, C++Builder creates a new class, *TForm1*, deriving it from the more generic class, *TForm.* The first line of the *TForm1* class declaration specifies that *TForm1* is derived from *TForm*:

```
   class TForm1: public TForm
```

Because *TForm1* is derived from *TForm*, all the elements that belong to a *TForm* class automatically become part of the *TForm1* class. If you look up *TForm* in online Help (you can click the form itself and press F1), you see lists of all the properties, methods, and events in the *TForm* class. You can use all the elements inherited from *TForm* in your application. Only the elements you specifically add to your *TForm1* class, such as components you place on the form or event handlers (methods) you write to respond to events, appear in the *TForm1* class declaration. These are the things that make *TForm1* different from *TForm*.

The more broad-based or generic class from which another more customized object inherits data and code is called the *ancestor* of the customized class. The customized class itself is a *descendant* of its ancestor. A class can have only one immediate ancestor, but it can have many descendants. For example, *TForm* is the ancestor type of *TForm1*, and *TForm1* is a descendant of *TForm*. All form classes are descendants of *TForm*, and you can derive many form classes from *TForm*.

Inheriting from TForm

To read these topics in sequence, press the >> button.

Objects, components, and controls

See also

When you look up *TForm* in online Help, you'll notice that *TForm* is called a component. Don't let this confuse you. All components and controls are also classes. The terminology used in the documentation comes from the inheritance hierarchy of the C++Builder Visual Component Library. Figure 9.3 is a greatly simplified diagram of the hierarchy, omitting some intermediary objects. To see a more complete diagram, refer to the *Component Writer's Guide*.

A simplified hierarchy diagram

Everything in this hierarchy is a class. Components, which inherit data and code from a *TObject*, are classes with additional properties, methods, and events that make them suitable for specialized purposes, such as the ability to save their state to a file. Controls, which inherit data and code from a *TComponent* type (which in turn inherits elements from *TObject*) have additional specialized capabilities, such as the ability to display something. So controls are components and classes, components are classes but not necessarily controls, and classes are simply classes. And an object is an instance of a class. This chapter refers to all components and controls as classes.

Even though *TCheckBox* isn't an immediate descendant of *TObject*, it still has all the attributes of any class because *TCheckBox* is ultimately derived from *TObject* in the VCL hierarchy. *TCheckBox* is a very specialized kind of class that inherits all the functionality of *TObject*, *TComponent*, and *TControl*, and defines some unique capabilities of its own.

To read these topics in sequence, press the >> button.

Object scope

See also

An object's scope determines the availability and accessibility of the data members, properties, and methods within that object. Using the earlier bicycle analogy, if you were to add a headlight only to your customized "bicycle object," the headlight would belong to that bicycle and to no other. If, however, the "basic model bicycle object" included a headlight, then all bicycle objects would inherit the presence of a headlight. The headlight could lie either within the scope of the ancestor bicycle object--in which case, a headlight would be a part of all descendant bicycle objects--or within the scope only of the customized bicycle object, and available only to that bicycle.

Likewise, all data members, properties, and methods declared within an object declaration are within the scope of the object, and are available to that object and its descendants.

When you write code in an event handler of an object that refers to properties, methods, or data members of the object itself, you don't need to preface these identifiers with the name of the object variable. For example, if you put a button and an edit box on a new form, you could write this event handler for the *OnClick* event of the button:

```
__fastcall TForm1::Button1Click(TObject *Sender);
{
 Color = clFuchsia;
 Edit1->Color = clLime;
}
```

The first statement colors the form. You could have written the statement like this:

```
Form1->Color = clFuchsia
```

It's not necessary, however, to put the *Form1* qualifier on the *Color* property because the *Button1Click* method is within the scope of the *TForm1* object. Any time you are within an object's scope, you can omit the qualifier on all properties, methods, and data members that are part of the object.

The second statement refers to the *Color* property of a *TEdit* object. Because you want to access the *Color* property of the *TEdit1* type, not of the *TForm1* type, you need to specify the scope of the Color property by including the name of the edit box, so the compiler can determine which *Color* property you are referring to. If you omit it, the second statement is like the first; the form ends up lime green, and the edit box control remains unchanged when the handler runs.

Because it's necessary to specify the name of the edit box whose *Color* property you are changing, you might wonder why it's not necessary to specify the name of the form as well. This is unnecessary because the control *Edit1* is within the scope of the *TForm1* object; it's declared to be a data member of *TForm1*.

To read these topics in sequence, press the >> button.

Accessing components on another form

See also

If *Edit1* were on some other form, you would need to preface the name of the edit box with the name of the form class. For example, if *Edit1* were on *Form2*, it would be a data member in the *TForm2* class declaration, and would lie with the scope of *Form2*. You would write the statement to change the color of the edit box in *Form2* from the *TForm1::ButtonClick* method like this:

```
Form2->Edit1->Color = clLime;
```

In the same way, you can also access methods of a component on another form. For example,

```
Form2->Edit1->Clear();
```

To give the code of *Form1* access to properties, methods, and events of *Form2*, you need to add *Unit2* with an **#include** statement to *Unit1* (assuming the units associated with *Form1* and *Form2* are named *Unit1* and *Unit2*, respectively).

To read these topics in sequence, press the >> button.

Scope and descendants of a class

See also

The scope of a class extends to all the classes' descendants. That means all the data members, properties, methods, and events that are part of *TForm* are within the scope of *TForm1* also, because *TForm1* is a descendant of *TForm* .

To read these topics in sequence, press the >> button.

Overriding a method

See also

You *can,* however*,* use the name of a method within an ancestor class to declare a method within a descendant class. This is how you *override* a method. You would most likely want to override an existing method if you want the method in the descendant class to do the same thing as the method in the ancestor class, but the task is accomplished in another way. In other words, the code that implements the two methods differs.

It's not often that you would want to override a method unless you are creating new components. You should be aware that you can do so, though, and that you won't receive any warning or error message from the compiler. You can read more about overriding methods in the *Component Writer's Guide.*

To read these topics in sequence, press the >> button.

Public and private declarations

See also

When you build an application using the C++Builder environment, you are adding data members and methods to a descendant of *TForm.* You can also add data members and methods to a class without putting components on a form or filling in event handlers, but by modifying the class declaration directly.

There are **private**, **protected**, **public**, **published**, and **automated** declarations in a class. You can add new data members and methods to either the **public** or **private** part of a class. **Public** and **private** are keywords. When you add a new form to the project, C++Builder begins constructing the new form class. Each new class contains the **public** and **private** keywords that mark locations for data members and methods you want to add to the code directly. For example, note the **private** and **public** parts in this new form class declaration that so far contains no data members or methods:

```
class TForm1 : public TForm
{
__published:       // IDE-managed Components
private:// User declarations
public:    // User declarations
   virtual __fastcall TForm1(TComponent* Owner);
};
```

Use the **public** part to

- Declare data members you want methods in other classes to access
- Declare methods you want other classes to access

Declarations in the **private** part are restricted in their access. If you declare data members or methods to be **private**, they are unknown and inaccessible outside the unit the class is defined in. Use the **private** part to

- Declare data members you want only methods in the same class to access
- Declare methods you want only methods in the same class to access

To add data members or methods to a **public** or **private** section, put the data members or method declarations after the appropriate comment, or erase the comments before you add the code. Here is an example:

```
class TForm1 : public TForm
{
__published:       // IDE-managed Components
   TEdit *Edit1;
   TButton *Button1;
   void __fastcall Button1Click(TObject *Sender);
private:// User declarations
   int Number;
      int Calculate (int X, int Y);
public:    // User declarations
   virtual __fastcall TForm1(TComponent* Owner);
      void ChangeColor();
};
```

Place the code that implements the *Calculate* and *ChangeColor* methods in the .cpp part of the unit file pair.

The *Number* data member and *Calculate* function are declared to be **private**. Only objects within the unit can use *Number* and *Calculate*. Usually, this restriction means that only the form class can use them, because each C++Builder unit file pair contains just one class declaration of type *TForm*.

Because the *ChangeColor* method is declared to be **public**, code in other units can use it. Such a method call from another unit must include the object name in the call:

```
Form1->ChangeColor;
```

The unit making this method call must include the header that declares *Form1*.

Note: When adding data members or methods to an object, always put the data members *before* the method declarations within each **public** or **private** part.

The other declarations listed at the beginning of this section are discussed in further detail in the Programmer's Guide. In brief however, the **published** declaration is used for the IDE-managed components and is similar to **public**. **Published** properties display in the Object Inspector. **Automated** declarations cause automation type information to be generated for the method property. This automation information makes it possible to create OLE Automation Servers. **Protected** declarations are similar to **private** (in that the implementation details are hidden) however the data member or member function can be inherited.

To read these topics in sequence, press the >> button.

Accessing object data members and methods

See also

When you want to change the value of a property of an object that is a data member in the form object, or you want to call a method of an object that is a data member in the form object, you must include the name of that object in the property name or method call. For example, if your form has an edit box control on it and you want to change the value of its *Text* property, you should write the assignment statement something like this, making sure to specify the name of the edit box (in this example, *Edit1*):

```
   Edit1->Text = 'Frank Borland was here';
```

Likewise, if you want to clear the text selected in the edit box control, you would write the call to the *ClearSelection* method like this:

```
   Edit1->ClearSelection();
```

To read these topics in sequence, press the >> button.

Assigning values to object variables

See also

You can assign one object variable to another object variable if the variables are of the same type or assignment compatible, just as you can assign variables of any type other than objects to variables of the same type or assignment-compatible types. For example, if the variables *Form1* and *Form2* have been declared as objects of type TForm, you could assign *Form1* to *Form2*:

```
Form2 = Form1;
```

You can also assign an object variable to another object variable if the type of the variable you are assigning a value to is an ancestor of the type of the variable being assigned. For example, here is a *TDataForm* class declaration:

```
class TTDataForm : public TForm
{
__published:      // IDE-managed Components
  TButton *Button1;
  TEdit *Edit1;
  TDBGrid *DBGrid1;
  TDatabase *Database1;
private:// User declarations
public:    // User declarations
  virtual __fastcall TTDataForm(TComponent* Owner);
};
```

Here are the declarations of two objects; one of type *TForm*, and one of type *TDataForm*:

```
TDataForm *DataForm;
TForm *AForm;
```

AForm is of type *TForm*, and *DataForm* is of type *TDataForm*. Because *TDataForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm = DataForm;
```

You might wonder why this is important. Taking a look at what happens behind the scenes when your application calls an event handler shows you why.

Suppose you fill in an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called.

Each event handler has a *Sender* parameter of type *TObject*. For example, note the *Sender* parameter in this empty *Button1Click* handler:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
```

If you look back at "A simplified hierarchy diagram", you'll recall that *TObject* is at the top of the C++Builder Visual Component Library. That means that all C++Builder objects are descendants of *TObject*. Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. Although you don't see the code that makes the assignment, the component or control to which the event happened is assigned to *Sender*. This means the value of *Sender* is always the control or component that responds to the event that occurred.

How is this information useful? You can test *Sender* to find the type of component or control that called the event handler. For example:

```
if (typeid(*Sender) == typeid(TEdit))
  DoSomething;
else
  DoSomethingElse;
```

To read these topics in sequence, press the >> button.

Creating nonvisual objects

See also

Most of the objects you use in C++Builder are components you can see at both design time and run time, such as edit boxes and string grids. A few, such as common dialog boxes, are components you don't see at design time, but they appear at run time. Still others, such as timers and data source components, never have any visual representation in your application at run time, but they are there for your application to use.

Occasionally you might want to create your own nonvisual objects for your application. For example, you might want to create a *TEmployee* object that contains *Name*, *Title*, and *HourlyPayRate* data members. You could then add a *CalculatePay* method that uses the data in the *HourlyPayRate* data member to compute a paycheck amount for a given period.

The *TEmployee* class declaration could look like this:

```
class TEmployee : public TObject
{
  private:
     char Name[25];
       char Title[25];
       double HourlyRate;
   public:
     double CalculatePayAmount();
};
```

In this case, *TEmployee* is derived from *TObject*. *TEmployee* contains three data members and one method. Because it is derived from *TObject*, *TEmployee* also contains all methods of *TObject* (*TObject* has no data members).

Place class declarations you create without the help of C++Builder in the class declaration part of your unit(.h header file) along with the form class declaration.

In the variable declaration part of the unit (the .cpp file), you need to declare a variable of the new type:

```
 TEmployee *Employee;
```

To read these topics in sequence, press the >> button.

Creating an instance of an object

See also

TEmployee is just a class. An actual object doesn't exist in memory until the object is *instantiated*, or created, by a *constructor* call. A constructor is a method that allocates memory for the new object and points to the new object, which is called an *instance* of the class. Calling the constructor, assigns this instance to a variable.

If you want to declare an instance of the *TEmployee* type, your code must call *TEmployee::TEmployee* before you can access any of the data members of the object:

```
Employee = new TEmployee;
```

In the *TEmployee* class declaration, there isn't a constructor method, but because *TEmployee* is derived from *TObject*, and *TObject* has a constructor method, *TEmployee* can call its default constructor to create a *TEmployee* instance, which is assigned to the *Employee* variable.

Now you are ready to access the data members of an *Employee* object, just as you would any other C++Builder object.

To read these topics in sequence, press the >> button.

Destroying your object

See also

When you are through using your object, you should destroy it, which releases the memory the object used. You destroy an object by calling a *destructor*, a method that releases the memory allocated to the object.

C++Builder has two destructors you can use, *the default class destructor* and *Free*. You should almost always use *Free*. In its implementation, the *Free* method calls the destructor, but only when the instance pointer isn't null; in other words, the pointer still points to the instance. For this reason, it is safer to call *Free* than the destructor. Also, calling *Free* is slightly more efficient in the resulting code size.

To destroy the *Employee* object when you have finished using it, you would make this call:

```
delete Employee;
```

Just as it inherits the constructor, *TEmployee* inherits a destructor from *TObject*.

To follow good programming practices, you should place your destructor call in the **finally** part of a **try..finally** block, while placing the code that uses the object in the **try** part. This assures that the memory allocated for your object is released even if an exception occurs while your code is attempting to use the object.

To read these topics in sequence, press the >> button.

Summary

See also

This chapter presented these topics:

- What is an object?

  An object is an instance of a class. A class combines data and code and also contains methods or functions that act on the data, and properties.

- Inheriting data and code from an object

  An object inherits all the data members, methods, and properties of its ancestor class. All the controls and components in the C++Builder Visual Component Library are objects, as they all descend from *TObject*.

- Object scope

  All data members, properties, and methods of an object are within the scope of the object. Any time you refer to a data member, property, or method within the scope of the object, you can omit the object qualifier.

  An object's scope extends to all descendants of the object.

  You can use the **public** part of an object to declare data members you want to be able to access from methods in objects in other units and to declare methods you want other objects in other units to be able to use.

  You can use the **private** part of an object to restrict access to an object's data members and methods.

  You can access the data members and methods of an object using a **with** statement.

- Assigning values to object variables

  You can assign one object variable to another object variable if the variables are of the same type or are assignment compatible. Using the reserved word **is**, you can determine which specific object is assigned to a variable. Also, instead of specifying a particular object, you can use the **as** reserved word and specify any object of a particular type in your code.

- Creating nonvisual objects

  You can create your own nonvisual objects. Write the class declaration in the .h file of the unit file pair. Before you can use the object, you need to instantiate it with a call to the constructor. When you have finished using the object, you must destroy it and release its allocated memory with a call to the destructor.

If you are interested in creating your own C++Builder components, you can learn more about objects by reading the *Component Writer's Guide.*

To read these topics in sequence, press the >> button.

Setting project options and compiling

See also

This chapter shows how to set project options using the Project Options dialog box, and it describes how to compile and run your program.

Setting project options

You can customize the way C++Builder appears and works for a particular project, including customizing the integrated development environment (IDE) itself. The appearance and behavior of the IDE when you start C++Builder or begin a new project are governed by the settings of several groups of options:

- *Environment settings* affect all C++Builder projects.

To set these options, choose Options|Environment.

- *Project settings* affect the current project only.

To set these options, choose Options|Project.

- *Repository settings* determine the default new form, new data module, and new project.

To set these options, choose Options|Repository.

▶ If you share your installation of C++Builder with other users, it's possible that another user has modified the option settings. In this case, displays or behaviors might differ from those described in the examples and illustrations in this chapter. In a shared-installation situation, it's a good idea to check and modify environment options as described in the following sections, before creating a new project.

If the Default box on the Project Options dialog is cleared, the current settings affect only the current open project. C++Builder creates an *options file* with a file extension .MAK in the project directory each time you save the project. When you reopen the project in future sessions, the project options saved in the options file will be in effect.

If you check the Default box and click OK, C++Builder also uses the current settings as the default for any new projects you create.

▶ The .MAK file can be used with the Borland MAKE utility.

Changing the defaults for new projects

See also

The Project Options dialog box contains a check box labeled Default. This control enables you to modify some of C++Builder's default project configuration properties. When you check this control, C++Builder saves your current settings in a file called DEFAULT.MAK (located in the BIN directory). C++Builder then uses the project options settings stored in this file as the default for any new projects you create. If the Default checkbox is cleared, your selections affect only the current Project.

▶ Project options you set for an open project override the current C++Builder defaults, whether those defaults are as originally shipped or as modified by you or another user. If you open an existing project or create a project from a template in the Object Repository that has its own options file, those settings will override the default settings in DEFAULT.MAK. DEFAULT.MAK is required for making new projects.

.MAK file

See also

The .MAK file stores the compiler and linker settings that C++Builder uses to build your project. When you save a project for the first time, C++Builder prompts you to save a .MAK file that defines the name of the project.

Because C++Builder maintains the .MAK file, you will not normally need to modify it manually, and it is generally recommended that you not do so. Most changes you could make to the .MAK file can be made using the Project|Options in C++Builder, and doing so ensures that C++Builder keeps all the project's files synchronized. Additional build options, however, are available as command line switches. For more information, see *Advanced Compiler Options* in online Help

If you need to manually edit the .MAK file, choose Project|View Project Makefile from the main menu to open it in the Code editor.

C++Builder treats each section of the .MAK file differently. For example, you can modify some parts only from the IDE and some parts you can modify only by editing the .MAK file manually. Additionally, some .MAK file options are intended only for the command line make utility. The following examples show the sections of the .MAK file generated by C++Builder for a new application and how they are used by the IDE.

You can modify the options in the following section of the .MAK file manually, but you cannot set them in the IDE:

```
VERSION = BCB.01
ifndef BCB
BCB = $(MAKEDIR)\..
!endif
```

Use the IDE to set the options in the following section of the .MAK file. Do not change them manually because C++Builder will overwrite them to match the current settings in the IDE:

```
ROJECT = myproject.exe
OBJFILES = myproject.obj
RESFILES = myproject.res
RESDEPEN = $(RESFILES)
LIBFILES =
```

You can set the following options either in the IDE or by editing the .MAK file manually. C++Builder uses the most recent settings in the IDE if you have changed them since they were last saved.

```
FLAG1 = -w -Od -Hc -k -r- -y -v -vi- -c -a4 -b- -w-par -w-inl -Vx -Ve -x   \
  -WE
CFLAG2 = -I$(BCB)\include;$(BCB)\include\vcl -H=$(BCB)\lib\vcld.csm
PFLAGS = -AWinTypes=Windows;WinProcs=Windows;DbiTypes=BDE;DbiProcs=BDE;DbiErrs=BDE \
   -v -$Y -$W -$O- -JPHNV -M -U$(BCB)\lib\obj;$(BCB)\lib   \
  -I$(BCB)\include;$(BCB)include\vcl
RFLAGS = -i$(BCB)\include;$(BCB)\include\vcl
LFLAGS = -L$(BCB)\lib\obj;$(BCB)\lib -aa -Tpe -x -v -V4.0   \
  -j$(BCB)\lib\obj;$(BCB)\lib
IFLAGS =
LINKER = ilink32
```

You can modify the following options by editing the .MAK file manually, but you cannot set them in the IDE:

```
ALLOBJ = c0w32.obj $(OBJFILES)
ALLRES = $(RESFILES)
ALLLIB = $(LIBFILES) vcl.lib import32.lib cp32mt.lib
```

The IDE ignores the options in the following section of the .MAK file. They are available, however, for use with the command line Make utility. You cannot set them in the IDE, but you can change them by editing the .MAK file manually:

```
#autodepend
$(PROJECT): $(OBJFILES) $(RESDEPEN)
   $(BCB)\BIN\$(LINKER) @&&!
   $(LFLAGS) +
   $(ALLOBJ), +
   $(PROJECT),, +
   $(ALLLIB),, +
   $(ALLRES)
```

```
!
.pas.hpp:
   $(BCB)\BIN\dcc32 $(PFLAGS) { $** }
.pas.obj:
   $(BCB)\BIN\dcc32 $(PFLAGS) { $** }
.cpp.obj:
   $(BCB)\BIN\bcc32 $(CFLAG1) $(CFLAG2) -o$* $*
.c.obj:
   $(BCB)\BIN\bcc32 $(CFLAG1) $(CFLAG2) -o$* $**
.rc.res:
   $(BCB)\BIN\brcc32 $(RFLAGS) $<
```

Project source file

See also

The project source file is the main program file C++Builder uses to compile and link all other units and files. C++Builder updates this file throughout the development of the project. As with unit source code files, you can open the project file in the Code editor. The main reason to do this is so you can see the units and forms that make up the project, and which form is specified as the application's main form. As you add forms and units to the project, you can see C++Builder's updates of the project source code.

To open the project source file, use either of these methods:

- Right-click the Project Manager and choose View Project Source.
- Choose View|Project Source from the main menu.

The contents of the project source file appears in a Code editor window. To hide the project file again, close the window in the Code editor.

C++Builder generates the following source code for a default, blank project:

```
#include <vcl\vcl.h>
#pragma hdrstop
//---------------------------------------------------------------------------
USEFORM("Unit1.cpp", Form1);
USERES("Project1.res");
//---------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
Application->Initialize();
       Application->CreateForm(__classid(TForm1), &Form1);
       Application->Run();
       return 0;
}
```

The default project code performs the following:

- The vcl.h header file contains definitions that are needed for the Visual Component Library. This file is included in each C++Builder project.
- The #pragma hrdstop statement instructs the compiler to stop adding header files to the collection of precompiled headers. By default, vcl.h is included in the precompiled headers.
- The USEFORM and USERES statements are two of several macros used to add items such as files, forms, and resources to your project. These macros are defined in sysdefs.h (located in the INCLUDE\VCL directory) that also defines USEDATAMODULE, USEDATAMODULENS, USEFORMNS, USEUNIT, USEOBJ, USERC, USELIB, and USEFILE. C++Builder creates these macros automatically and you will not normally need to change them.
- The *WinMain* call is the program entry point.
- Application->Initialize(); initializes the VCL Application object for this process.
- The *Application->CreateForm* statement creates the form specified in the function parameters. The statement creates the form object specified in the second parameter using the form class specified in the first parameter.

  C++Builder adds an *Application->CreateForm* statement to the project file for each form you add to the project. The statements are listed in the order the forms are added to the project. This is the order that the forms will be created in memory at runtime. To change this order, edit the project source file.

  The *Application->Run();* statement causes the process to enter the program's message loop. From there, the process gets Windows messages and dispatches them to the proper window message procedure. This is essentially where your program begins running in the Windows environment.

Compiling, building, and running projects

All C++Builder projects have as a target a single distributable executable file, either an .EXE or a .DLL file. You can view or test your application at various stages of development the folowing ways:

- compiling
- building
- running

Compiling a single file

See also

To compile a single unit in your project,

1. Make the unit you want to compile the current selected form unit (in the editor) or file (in Project Manager window).
2. Choose the Project|Compile Unit from the main menu.

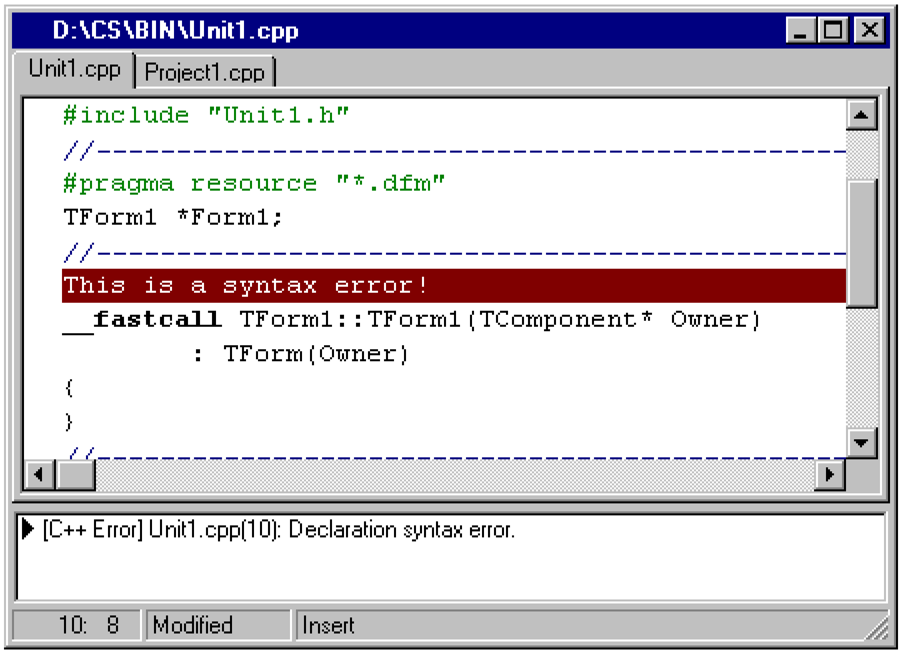
The hourglass cursor appears or the "progress" dialog is displayed, if that option has been set, to indicate that a compile is in progress and returns to normal if the compiler finds no errors.

If an error is found when the Code editor is not open, the Message window displays an error message. If you double-click the message, the Code editor displays the source file showing the line where the error occurred.

If an error is found when the Code editor is open,

- The Code editor window comes to the front.
- The unit source file page containing the error comes to the top of the Code editor.
- The line containing the error is highlighted in the Code editor.
- The Code editor message pane displays an error message.
- Context-sensitive help about the error message is available by pressing F1.

Compile error display



```
#include "Unit1.h"
//---------------------------------------------------
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------
This is a syntax error!
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------
```

```
[C++ Error] Unit1.cpp(10): Declaration syntax error.
```

Making a project

See also

To compile and link all the source files that changed since you last compiled them, choose Project|Make from the main menu.

When you choose this command, this is what happens:

- The compiler compiles source code for each unit if the source code has changed since the last time the unit was compiled.
- If a unit contains an include (.h) file, and the include file is newer than the unit's .OBJ file, the unit is recompiled.

Once all the units that make up the project have been compiled, C++Builder links them into an executable file (or dynamic link library). This file is given an .EXE (or .DLL) file extension and the same file name as the project source code file. This file now contains all the compiled code and forms found in the individual units, and the program is ready to run.

Obtaining compile status information

You can get information about the compile status of your project by displaying the Information dialog box (Choose Information from the Project menu). This dialog box displays information about the number of lines of source code compiled, the byte size of your code and data, the stack size, and the compile status of the project.

To get status information from the compiler as a project compiles,

1. Choose Options|Environment.
2. On the Preferences tab, check Show Compiler Progress.

For more information, see Environment Options in online Help.

Building a project

See also

To compile and link all the source files in your project, regardless of when they were last compiled, choose Project|Build All from the main menu.

The result of this command is similar to that of the Project|Make command, except that all units in the project are compiled, regardless of whether or not they have changed since the last compile. When you choose Build All, precompiled headers and files generated by the incremental linker are deleted and regenerated.

This technique is useful if you are unsure of exactly which files have changed or if you want to ensure that all files are current and synchronized. It's also important to Build All when you've changed global compiler directives or compiler options to ensure that all code compiles in the proper state.

Running a project

See also

You can test run a project from within C++Builder, or you can run the compiled .EXE file from the Windows operating environment without having to run C++Builder.

You can compile and then run your application from within C++Builder the following ways:

- Choose Run|Run from the main menu.
- Choose the Run button on the C++Builder toolbar.

These actions are the same as choosing the Project|Make command, except that C++Builder runs your application immediately if the compile operation succeeds.

Executing a project from Windows

Unless you select the In Memory-EXE option on the Linker page of the Project Options dialog box, the compiler creates a fully compiled standalone executable file (.EXE) that you can run from the Windows operating environment using the same techniques as you would for any other Windows application.

If you specified an icon for your project, it will appear in the Program Manager if you create a Windows program item for your application, or if you minimize the application while it is running.

Managing multiple project versions and team development

See also

When you are developing a complex programming project in a team setting, or managing several development projects, you might soon develop the need for a version control system (VCS). A version control system can archive files, control access to project files, and track multiple versions of your projects.

C++Builder has a built-in interface for Intersolv's PVCS Version Manager. It supports PVCS version 5.1 and later. C++Builder Client/Server Suite includes both the PVCS interface and PVCS itself.

Enabling team development support

Before using C++Builder's built-in team development support, you need to have PVCS installed on your system, and several files must reside in your \WINDOWS\SYSTEM directory.

For information on using the team development support features of C++Builder, refer to the Version Control online Help (PVCS.HLP).

Setting advanced options in C++Builder

See also

In C++Builder, you set project options through the Project Options dialog box, reached with the Options|Project menu command. The Project Options dialog box lets you set general-use options for the C++Builder compilers and linkers.

If you need to set more advanced compiler and linker options, you can do so by setting options in the project .MAK file.

When you open the Project makefile (by choosing View|Project Makefile), you will see a section that contains the following makefile code:

```
CFLAG1 = …
CFLAG2 = …
PFLAGS = …
RFLAGS - …
LFLAGS = …
IFLAGS = …
```

These sections contain the option settings for the C++ compiler, the Object Pascal compiler, the resource compiler, and the linkers.

To set advanced C++ compiler options, add the command-line switch of the desired option to the `CFLAG1` line (if you are adding or modifying a switch that uses a string, add the command-line switch to the `CFLAF2` line). Use the `PFLAGS` line for Object Pascal compiler options. The `RFLAGS` line contains resource compiler options, and the `LFLAGS` line sets the linker options. You can use the `IFLAGS` line to set general options that affect the behavior of C++Builder, for example you can use this line to set which compiler warning display during a compilation.

Compiler and linker options

The C++Builder compiler options are grouped into the following areas:

Compiler-specific options
C++-specific options
Optimization options
Warning message options

For a complete listing of the compiler and linker options, refer to the following charts:

Compiler options (alphabetical listing)
Compiler options (functional listing)

Linker options (alphabetical listing)
Linker options (functional listing)

Compiler options (alphabetical listing)

See also

The following table is an alphabetical listing of the C++Builder compiler options:

Option	Description
`@<filename>`	Read compiler options from the response file *filename*
`+<filename>`	Use alternate compiler configuration file *filename*
`-3`	Generate 80386 protected-mode compatible instructions (Default)
`-4`	Generate 80386/80486 protected-mode compatible instructions
`-5`	Generate Pentium instructions
`-6`	Generate Pentium Pro instructions
`-A`	Use ANSI keywords and extensions
`-AK`	Use Kernighan and Ritchie keywords and extensions
`-AT`	Use Borland C++ keywords and extensions (also `-A-`)
`-AU`	Use UNIX V keywords and extensions
`-a`	Align byte (Default: -a- use byte-aligning)
`-an`	Align data on "n" boundaries, where 1=byte, 2=word (2 bytes), 4=Double word (4 bytes), 8=Quad word (8 bytes), 16=Paragraph (16 bytes)
`-B`	Compile to .ASM (`-S`), the assemble to .OBJ
`-b`	Make enums always integer-sized (Default: -b- make enums byte-sized when possible)
`-C`	Turn nested comments on (Default: -C- turn nested comments off)
`-c`	Compile to .OBJ, no link
`-D<name>`	Define "name" to the null string
`-Dname=string`	Define "name" to "string"
`-d`	Merge duplicate strings (Default)
`-E<filename>`	Specify assembler
`-e<filename>`	Specify executable file name
`-f`	Emulate floating point
`-f-`	No floating point
`-ff`	Fast floating point
`-fp`	Correct Pentium FDIV flaw
`-gn`	Warnings: stop after *n* messages (Default = 255)
`-H`	Generate and use precompiled headers (Default)
`-H=<filename>`	Set the name of the file for precompiled headers
`-H”xxx”`	Stop precompiling after header file *xxx*
`-Hc`	Cache precompiled header
`-Hu`	Use but do not generate precompiled headers
`-in`	Make significant identifier length to be *n* (Default = 250)
`-Jg`	Generate definitions for all template instances and merge duplicates (Default)

-Jgd	Generate public definitions for all template instances; duplicates result in redefinition errors
-Jgx	Generate external references for all template instances
-jn	Errors: stop after *n* messages (Default = 25)
-K	Default character type unsigned (Default: -K- default character type signed)
-k	Turn on standard stack frame (Default)
-lx	Pass option *x* to linker
-M	Create a Map file
-O	Optimize jumps
-O1	Generate smallest possible code
-O2	Generate fastest possible code
-Oc	Eliminate duplicate expressions within basic blocks and functions
-Od	Disable all optimizations
-Oi	Expand common intrinsic functions
-OS	Pentium instruction scheduling
-Ov	Enable loop induction variable and strength reduction
-o<filename>	Compile .OBJ to *filename*
-P	Perform C++ compile regardless of source extension
-P<ext>	Perform C++ compile, set output to extension to *.ext*
-p	Use Pascal calling convention
-pc	Use C calling convention (Default: -pc, -p-)
-po	Use fastthis calling convention for passing **this** parameter in registers
-pr	Use fastcall calling convention for passing parameters in registers
-ps	Use stdcall calling convention
-R	Include browser information in generated .OBJ files
-RT	Enable runtime type information (Default)
-r	Use register variables (Default)
-S	Compile to assembler
-Tx	Specify assembler option *x*
-tWM	Generate a 32-bit multi-threaded target
-U<name>	Undefine any previous definitions of *name*
-u	Generate underscores (Default)
-V	Use smart C++ virtual tables (Default)
-V0	External C++ virtual tables
-V1	Public C++ virtual tables
-VC	Calling convention mangling compatibility
-Vd	for loop variable scoping
-Ve	Zero-length empty base classes

`-VF`	MFC compatibility
`-v`	Turn on source debugging
`-vi`	Control expansion of inline functions
`-w`	Display warnings on
`-wxxx`	Enable *xxx* warning message
`-w-xxx`	Disable *xxx* warning message
`-wmsg`	User-defined warnings
`-X`	Disable compiler autodependency output (Default: -X- use compiler autodependency output)
`-x`	Enable exception handling (Default)
`-xc`	Enable compatible exception handling
`-xd`	Enable destructor cleanup (Default)
`-xf`	Enable fast exception prologs
`-xp`	Enable exception location information
`-y`	Debug line numbers on

Message options (alphabetical listing)

`-wamb`	Ambiguous operators need parentheses
`-wamp`	Superfluous & with function
`-wasm`	Unknown assembler instruction
`-w-aus`	'identifier' is assigned a value that is never used (Default ON)
`-wbbf`	Bit fields must be signed or unsigned int
`-w-bei`	Initializing 'identifier' with 'identifier' (Default ON)
`-w-big`	Hexadecimal value contains more than three digits (Default ON)
`-w-ccc`	Condition is always true OR Condition is always false (Default ON)
`-wcln`	Constant is long
`-w-cpt`	Nonportable pointer comparison (Default ON)
`-wdef`	Possible use of 'identifier' before definition
`-w-dpu`	Declare type 'type' prior to use in prototype (Default ON)
`-w-dup`	Redefinition of 'macro' is not identical (Default ON)
`-w-dsz`	Array size for 'delete' ignored (Default ON)
`-weas`	Assigning ‘type’ to ‘enum’
`-w-eff`	Code has no effect (Default ON)
`-w-ias`	Array variable 'identifier' is near (Default ON)
`-w-ext`	'identifier' is declared as both external and static (Default ON)
`-whch`	Handler for '<type1>' Hidden by Previous Handler for '<type2>'
`-w-hid`	'function1' hides virtual function 'function2' (Default ON)
`-w-ibc`	Base class 'base1' is inaccessible because also in 'base2' (Default ON)
`-w-ill`	Ill-formed pragma (Default ON)

`-w-inl`	Functions containing reserved words are not expanded inline (Default ON)
`-w-lin`	Temporary used to initialize 'identifier' (Default ON)
`-w-lvc`	Temporary used for parameter 'parameter' in call to 'function' (Default ON)
`-wmsg`	User-defined warnings
`-w-mpc`	Conversion to type fails for members of virtual base class base (Default ON)
`-w-mpd`	Maximum precision used for member pointer type <type> (Default ON)
`-wnak`	Non-ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement for ANSI conformance)
`-w-nci`	The constant member 'identifier' is not initialized (Default ON)
`-wnfc`	Non-constant function ‘ident’ called for const object
`-wnod`	No declaration for function 'function'
`-w-nst`	Use qualified name to access nested type 'type' (Default ON)
`-w-ntd`	Use '> >' for nested templates instead of '>>' (Default ON)
`-w-nvf`	Non-volatile function <function> called for volatile object (Default ON)
`-w-obi`	Base initialization without a class name is now obsolete (Default ON)
`-wobs`	'ident' is obsolete
`-w-ofp`	Style of function definition is now obsolete (Default ON)
`-w-ovl`	Overload is now unnecessary and obsolete (Default ON)
`-w-par`	Parameter 'parameter' is never used (Default ON)
`-w-pch`	Cannot create precompiled header: header (Default ON)
`-w-pia`	Possibly incorrect assignment (Default ON)
`-wpin`	Initialization is only partially bracketed
`-wpre`	Overloaded prefix operator 'operator' used as a postfix operator
`-w-pro`	Call to function with no prototype (Default ON)
`-w-rch`	Unreachable code (Default ON)
`-w-ret`	Both return and return of a value used (Default ON)
`-w-rng`	Constant out of range in comparison (Default ON)
`-w-rpt`	Nonportable pointer conversion (Default ON)
`-w-rvl`	Function should return a value (Default ON)
`-wsig`	Conversion may lose significant digits
`-wstu`	Undefined structure 'structure'
`-wstv`	Structure passed by value
`-w-sus`	Suspicious pointer conversion (Default ON)
`-wucp`	Mixing pointers to different 'char' types
`-wuse`	'identifier' declared but never used
`-w-voi`	Void functions may not return a value (Default ON)
`-w-zdi`	Division by zero (Default ON)

Compiler options (functional listing)

See also

For the ease of access, the compiler options are grouped into the following functional categories:

Compiler-specific options

- Configuration and Response files
- Defines
- Code generation
- Floating point
- Compiler output
- Source
- Debugging
- Precompiled headers
- Processor
- Calling convention

C++-specific options

- C++ compatibiltiy
- Virtual tables
- Templates
- Exception handling

Optimization options

Warning message options

- Portability
- ANSI violations
- Obsolete C++
- Potential C++ errors
- Inefficient C++ coding
- Potential errors
- Inefficient coding
- General

Linker options

- Input options
- Map file options
- Output options
- File addressing options
- File addressing options
- Application target options
- Linker messages and warnings

Compiler configuration files

`@<filename>`	Read compiler options from the response file " *filename*

Compiler response files

`+<filename>`	Use alternate configuration file *filename*

Compiler options | Defines

`-D<name>`	Define *name* to the null string
`-Dname=string`	Define "name" to "string"
`-U<name>`	Undefine any previous definitions of *name*

Compiler options | Code generation

`-an`	Align data on "n" boundaries, where 1=byte, 2=word (2 bytes), 4=Double word (4 bytes), 8=Quad word (8 bytes), 16=Paragraph (16 bytes)

`-b`	Make enums always integer-sized (Default: -b- make enums byte-sized when possible)
`-K`	Default character type unsigned (Default: -K- default character type signed)
`-d`	Merge duplicate strings (Default)
`-po`	Use fastthis calling convention for passing **this** parameter in registers
`-r`	Use register variables (Default)

Compiler options | Floating point

`-f-`	No floating point
`-f`	Emulate floating point
`-ff`	Fast floating point
`-fp`	Correct Pentium **fdiv** flaw

Compiler options | Compiler output

`-c`	Compile to .OBJ, no link
`-e<filename>`	Specify executable file name
`-lx`	Pass option *x* to linker
`-M`	Create a Map file
`-o<filename>`	Compile .OBJ to *filename*
`-P`	Perform C++ compile regardless of source extension
`-P<ext>`	Perform C++ compile, set output to extension to *.ext*
`-tWM`	Generate a 32-bit multi-threaded target
`-X`	Disable compiler autodependency output (Default: -X- use compiler autodependency output)
`-u`	Generate underscores (Default)

Compiler options | Source

`-C`	Turn nested comments on (Default: -C- turn nested comments off)
`-in`	Make significant identifier length to be "n" (Default = 250)
`-AT`	Use Borland C++ keywords (also -A-)
`-A`	Use only ANSI keywords
`-AU`	Use only UNIX V keywords
`-AK`	Use only Kernighan and Ritchie keywords
`-VF`	MFC compatibility

Compiler options | Debugging

`-k`	Turn on standard stack frame (Default)
`-vi`	Control expansion of inline functions
`-y`	Line numbers on
`-v`	Turn on source debugging
`-R`	Include browser information in generated .OBJ files

Compiler options | Precompiled headers

-H	Generate and use precompiled headers (Default)
-Hu	Use but do not generate precompiled headers
-Hc	Cache precompiled header
-H=filename	Set the name of the file for precompiled headers
-H"xxx"	Stop precompiling after header file xxxx

Compiler options | Processor

-3	Generate 80386 instructions. (Default)
-4	Generate 80486 instructions
-5	Generate Pentium instructions
-6	Generate Pentium Pro instructions

Compiler options | Calling convention

-pc	Use C calling convention (Default: -pc, -p-)
-p	Use Pascal calling convention
-pr	Use fastcall calling convention for passing parameters in registers
-ps	Use stdcall calling convention

Compiler options | Assembler-code options

-B	Compile to .ASM (-S), the assemble to .OBJ
-E<filename>	Specify assembler
-S	Compile to assembler
-Tx	Specify assembler option *x*

C++ options | C++ compatibility

-VC	Calling convention mangling compatibility
-Vd	for loop variable scoping
-Ve	Zero-length empty base classes

C++ options | Virtual tables

-V	Use smart C++ virtual tables (Default)
-V0	External C++ virtual tables
-V1	Public C++ virtual tables

C++ options | Templates

-Jg	Generate definitions for all template instances and merge duplicates (Default)
-Jgd	Generate public definitions for all template instances; duplicates result in redefinition errors
-Jgx	Generate external references for all template instances

C++ options | Exception handling

-x	Enable exception handling (Default)
-xp	Enable exception location information
-xd	Enable destructor cleanup (Default)
-xf	Enable fast exception prologs

`-xc`	Enable compatible exception handling
`-RT`	Enable runtime type information (Default)

Optimization options

`-O`	Optimize jumps
`-O1`	Generate smallest possible code
`-O2`	Generate fastest possible code
`-Oc`	Eliminate duplicate expressions within basic blocks and functions
`-Od`	Disable all optimizations
`-Oi`	Expand common intrinsic functions
`-OS`	Pentium instruction scheduling
`-Ov`	Enable loop induction variable and strength reduction

Warning message options

`-w`	Display warnings on
`-wxxx`	Enable *xxx* warning message
`-w-xxx`	Disable *xxx* warning message
`-gn`	Warnings: stop after *n* messages (Default = 100)
`-jn`	Errors: stop after *n* messages (Default = 25)

Message options | Portability

`-w-rpt`	Nonportable pointer conversion (Default ON)
`-w-cpt`	Nonportable pointer comparison (Default ON)
`-w-rng`	Constant out of range in comparison (Default ON)
`-wcln`	Constant is long
`-wsig`	Conversion may lose significant digits
`-wucp`	Mixing pointers to different 'char' types

Message options | ANSI violations

`-w-voi`	Void functions may not return a value (Default ON)
`-w-ret`	Both return and return of a value used (Default ON)
`-w-sus`	Suspicious pointer conversion (Default ON)
`-wstu`	Undefined structure 'structure'
`-w-dup`	Redefinition of 'macro' is not identical (Default ON)
`-w-big`	Hexadecimal value contains more than three digits (Default ON)
`-wbbf`	Bit fields must be signed or unsigned int
`-w-ext`	'identifier' is declared as both external and static (Default ON)
`-w-dpu`	Declare type 'type' prior to use in prototype (Default ON)
`-w-zdi`	Division by zero (Default ON)
`-w-bei`	Initializing 'identifier' with 'identifier' (Default ON)
`-wpin`	Initialization is only partially bracketed
`-wnak`	Non-ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement

	for ANSI conformance)

Message options | Obsolete C++

`-w-obi`	Base initialization without a class name is now obsolete (Default ON)
`-w-ofp`	Style of function definition is now obsolete (Default ON)
`-wpre`	Overloaded prefix operator 'operator' used as a postfix operator
`-w-ovl`	Overload is now unnecessary and obsolete (Default ON)

Message options | Potential C++ errors

`-w-nci`	The constant member 'identifier' is not initialized (Default ON)
`-weas`	Assigning 'type' to 'enum'
`-w-hid`	'function1' hides virtual function 'function2' (Default ON)
`-wnfc`	Non-constant function 'ident' called for const object
`-wibc`	Base class 'base1' is inaccessible because also in 'base2' (Default ON)
`-w-dsz`	Array size for 'delete' ignored (Default ON)
`-w-nst`	Use qualified name to access nested type 'type' (Default ON)
`-whch`	Handler for '<type1>' Hidden by Previous Handler for '<type2>'
`-w-mpc`	Conversion to type fails for members of virtual base class base (Default ON)
`-w-mpd`	Maximum precision used for member pointer type <type> (Default ON)
`-w-ntd`	Use '> >' for nested templates instead of '>>' (Default ON)
`-w-nvf`	Non-volatile function <function> called for volatile object (Default ON)

Message options | Inefficient C++ coding

`-w-inl`	Functions containing reserved words are not expanded inline (Default ON)
`-w-lin`	Temporary used to initialize 'identifier' (Default ON)
`-w-lvc`	Temporary used for parameter 'parameter' in call to 'function' (Default ON)

Message options | Potential errors

`-w-pia`	Possibly incorrect assignment (Default ON)
`-wdef`	Possible use of 'identifier' before definition
`-wnod`	No declaration for function 'function'
`-w-pro`	Call to function with no prototype (Default ON)
`-w-rvl`	Function should return a value (Default ON)
`-wamb`	Ambiguous operators need parentheses
`-w-ccc`	Condition is always true OR Condition is always false (Default ON)

Message options | Inefficient coding

`-w-aus`	'identifier' is assigned a value that is never used (Default ON)
`-w-par`	Parameter 'parameter' is never used (Default ON)
`-wuse`	'identifier' declared but never used
`-wstv`	Structure passed by value
`-w-rch`	Unreachable code (Default ON)
`-w-eff`	Code has no effect (Default ON)

Message options | General

-wasm	Unknown assembler instruction
-w-ill	Ill-formed pragma (Default ON)
-w-ias	Array variable 'identifier' is near (Default ON)
-wamp	Superfluous & with function
-wobs	'ident' is obsolete
-w-pch	Cannot create precompiled header: header (Default ON)
-wmsg	User-defined warnings

Linker options (alphabetical listing)

The following list shows the TLINK and ILINK linker options:

Compiler-specific options

Compiler-specific options can be used with all C and C++ programs. They directly affect how the compiler generates code.

The subtopics are

Compiler define options

The macro definition capability of C++Builder lets you define and undefine macros (also called *manifest* or *symbolic* constants). The macros you define override those defined in your source files.

Defining macros

`(Command-line switch: `**`-D`*`name`***` and `**`-D`*`name`*`=`*`string`***`)`

The **`-D`*`name`*** option defines the identifier *name* to the null string. **`-D`*`name`*`=`*`string`*** defines *name* to *string*. In this assignment, *string* cannot contain spaces or tabs. You can also define multiple **#define** options on the command line using either of the following methods:

- Include multiple definitions after a single **`-D`** option by separating each define with a semicolon (;) and assigning values with an equal sign (=). For example:
  ```
  BCC.EXE -Dxxx;yyy=1;zzz=NO MYFILE.C
  ```
- Include multiple **`-D`** options, separating each with a space. For example:
  ```
  BCC.EXE -Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C
  ```

Undefining macros

`(Command-line switch = `**`-U`*`name`***`)`

This command-line option undefines the previous definition of the identifier *name*.

Compiler code generation options

Compiler Code Generation options affect how code is generated.

The options are

Instruction set
Calling conventions
Data alignment
Allocate enums as ints
Unsigned characters
Duplicate strings merged
fastthis
Register variables

Instruction set options

The Instruction Set options specify for which CPU instruction set the compiler should generate code.

80386
`(Command-line switch: -3)`

Choose the 80386 option if you want the compiler to generate 80386 protected-mode compatible instructions running on Windows 95 or Windows NT.

i486
`(Command-line switch: -4)`

Choose the i486 option if you want the compiler to generate i486 protected-mode compatible instructions running on Windows 95 or Windows NT.

Pentium
`(Command-line switch: -5)`

Choose the Pentium option if you want the compiler to generate Pentium instructions on Windows 95 or Windows NT.

While this option increases the speed at which the application runs on Pentium machines, expect the program to be a bit larger than when compiled with the **80386** or **i486** options. In addition, **Pentium**-compiled code will sustain a performance hit on non-Pentium systems.

Pentium Pro
`(Command-line switch: -6)`

Choose the Pentium Pro option if you want the compiler to generate Pentium Pro instructions running on Windows 95 or Windows NT.

Default = 80386 (`-3`)

Calling convention options

Calling Convention options tell the compiler which calling sequences to generate for function calls. The C, Pascal, and Register calling conventions differ in the way each handles stack cleanup, order of parameters, case, and prefix of global identifiers.

You can use the **_ _cdecl**, **_ _pascal**, **_ _fastcall,** or **_ _stdcall** keywords to override the default calling convention on specific functions.

▶ These options should be used by expert programmers only.

C

`(Command-line switch: -pc, -p-)`

This option tells the compiler to generate a C calling sequence for function calls (generate underbars, case sensitive, push parameters right to left). This is the same as declaring all subroutines and functions with the **_ _cdecl** keyword. Functions declared using the C calling convention can take a variable parameter list (the number of parameters does not need to be fixed).

You can use the **_ _pascal**, **_ _fastcall**, or **_ _stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Pascal

`(Command-line switch: -p)`

This option tells the compiler to generate a Pascal calling sequence for function calls (do not generate underbars, all uppercase, calling function cleans stack, pushes parameters left to right). This is the same as declaring all subroutines and functions with the **_ _pascal** keyword. The resulting function calls are usually smaller and faster than those made with the C (**`-pc`**) calling convention. Functions must pass the correct number and type of arguments.

You can use the **_ _cdecl**, **_ _fastcall**, or **_ _stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Register

`(Command-line switch: -pr)`

This option forces the compiler to generate all subroutines and all functions using the **Register** parameter-passing convention, which is equivalent to declaring all subroutine and functions with the **_ _fastcall** keyword. With this option enabled, functions or routines expect parameters to be passed in registers.

You can use the **_ _pascal**, **_ _cdecl**, or **_ _stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Standard Call

`(Command-line switch: -ps)`

This option tells the compiler to generate a Stdcall calling sequence for function calls (does not generate underscores, preserve case, called function pops the stack, and pushes parameters right to left). This is the same as declaring all subroutines and functions with the **_ _stdcall** keyword. Functions must pass the correct number and type of arguments.

You can use the **_ _cdecl**, **_ _pascal**, **_ _fastcall** keywords to specifically declare a function or subroutine using another calling convention.

Default = C (**`-pc`**)

Data alignment options

The Data Alignment options let you choose the compiler aligns data in stored memory. Word, double-word, and quad-word alignment forces integer-size and larger items to be aligned on memory addresses that are a multiple of the type chosen. Extra bytes are inserted in structures to ensure that members align correctly.

Byte alignment

`(Command-line switch: -a1 or -a-)`

When Byte Alignment is turned on, the compiler does not force alignment of variables or data fields to any specific memory boundaries; the compiler aligns data at either even or odd addresses, depending on which is the next available address.

While byte-wise alignment produces more compact programs, the programs tend to run a bit slower. The other data alignment options increase the speed that 80x86 processors fetch and store data.

Word alignment (2-byte)

`(Command-line switch: -a2)`

When Word Alignment is on, the compiler aligns non-character data at even addresses. Automatic and global variables are aligned properly. **char** and **unsigned char** variables and fields can be placed at any address; all others are placed at an even-numbered address.

Double word (4-byte)

`(Command-line switch: -a4)`

Double Word alignment aligns non-character data at 32-bit word (4-byte) boundaries.

Quad word (8-byte)

`(Command-line switch: -a8)`

Quad Word alignment aligns non-character data at 64-bit word (8-byte) boundaries.

Paragraph (16-byte)

`(Command-line switch: -a16)`

Paragraph alignment aligns non-character data at 128-bit word (16-byte) boundaries.

Default = Byte Alignment (**`-a1`**)

Allocate enums as ints option

`(Command-line switch: -b)`

When the Allocate Enums As Ints option is set, the compiler always allocates a whole word (a four-byte **int** for 32-bit programs) for enumeration types (variables of type **enum**).

When this option is off (`-b-`), the compiler allocates the smallest integer that can hold the enumeration values: the compiler allocates an **unsigned** or **signed char** if the values of the enumeration are within the range of 0 to 255 (minimum) or -128 to 127 (maximum), or an **unsigned** or **signed short** if the values of the enumeration are within the following ranges:

- 0 to 4,294,967,295 or -2,147,483,648 to 2,147,483,647

The compiler allocates a four-byte **int** (32-bit) to represent the enumeration values if any value is out of range.

Default = ON

Unsigned characters option

`(Command-line switch: -K)`

When Unsigned Characters is set, the compiler treats all **char** declarations as if they were **unsigned char** type, which provides compatibility with other compilers.

Default = OFF (**char** declarations default to **signed**; **`-K-`**)

Duplicate strings merged option

(Command-line switch: **-d**)

When you set this option, the compiler merges two literal strings when one matches another. This produces smaller programs (at the expense of a slightly longer compile time), but can introduce errors if you modify one string.

Default = OFF (**-d-**)

fastthis option

(Command-line switch: **-po**)

This option causes the compiler to use the **_ _fastthis** calling convention when passing the **this** pointer to member functions. The **this** pointer is passed in a register. Likewise, calls to member functions load the register with **this**. Note that you can use **_ _fastthis** to compile specific functions in this manner.

The names of member functions compiled with **_ _fastthis** are mangled differently from non-fastthis member functions, to prevent mixing the two. It is easiest to compile all classes with **_ _fastthis**, but you can compile some classes with **_ _fastthis** and some without, as in the following example:

```
// no -po on the command-line
class X;
#pragma option -po
class Y      //Y will use fastthis
{
...
};
class X      //X will not use fastthis,
{            //since its class declaration
             //appeared before fastthis was turned on
...
};
#pragma option -po-
```

- If you use a makefile to build a version of the class library that has **_ _fastthis** enabled, you must define `_CLASSLIB_ALLOW_po` and use the **-po** option. The `_CLASSLIB_ALLOW_po` macro can be defined in <Your_BCB_dir>\INCLUDE\SERVICES\borlandc.h
- If you use a makefile to build a **_ _fastthis** version of the runtime library, you must define `_RTL_ALLOW_po` and use the **-po** option.
- If you rebuild the libraries and use **-po** without defining the appropriate macro, the linker emits undefined symbol errors.

Default = OFF

Register variable options

These options suppress or enable the use of register variables.

None
`(Command-line switch: -r-)`

None tells the compiler not to use register variables, even if you have used the **register** keyword.

Register keyword
`(Command-line switch: -rd)`

Register Keyword tells the compiler to use register variables only if you use the **register** keyword and a register is available. Use this option or the Automatic option (`-r`) to optimize the use of registers.

- You can use **-rd** in **#pragma** options.

Automatic
`(Command-line switch: -r)`

Automatic tells the compiler to automatically assign register variables if possible, even when you do not specify a register variable by using the **register** type specifier.

Generally, you can use **Automatic**, unless you are interfacing with preexisting assembly code that does not support register variables.

Default = Automatic (`-r`)

Compiler floating point options

Floating Point options tell the compiler how to handle floating-point code and floating-point optimization.

No floating point

`(Command-line switch: -f-)`

Use No Floating Point if you are not using floating point. No floating-point libraries are linked when this option is set (`-f-`). If you enable this option and use floating-point calculations in your program, you will get link errors. When unchecked (`-f`), the compiler emulates 80x87 calls at runtime.

Default = OFF (`-f`)

Fast floating point

`(Command-line switch: -ff)`

When Fast Floating Point is on, floating-point operations are optimized without regard to explicit or implicit type conversions. Calculations can be faster than under ANSI operating mode.

When this option is unchecked (`-ff-`), the compiler follows strict ANSI rules regarding floating-point conversions.

Default = OFF

Correct Pentium FDIV flaw

`(Command-line switch: -fp)`

Some early Pentium chips do not perform specific floating-point division calculations with full precision. Although your chances of encountering this problem are slim, this switch inserts code that emulates floating-point division so that you are assured of the correct result. This option decreases your program's FDIV instruction performance.

▪ Use of this option only corrects FDIV instructions in modules that you compile. The run-time library also contains FDIV instructions which are not modified by the use of this switch. To correct the run-time libraries, you must recompile them using this switch.

The following functions use FDIV instructions in assembly language which are not corrected if you use this option:

acos	**cosh**	**pow10l**
acosl	**coshl**	**powl**
asin	**cosl**	**sin**
asinl	**exp**	**sinh**
atan	**expl**	**sinhl**
atan2	**fmod**	**sinl**
atan2l	**fmodl**	**tan**
atanl	**pow**	**tanh**
cos	**pow10**	**tanhl**
tanl		

In addition, this switch does not correct functions that convert a floating-point number to or from a string (such as *printf* or *scanf*).

Default = OFF

Compiler output options

The following options control the compiler outputs

General compiler output options

Compile to .OBJ, no link
(Command-line switch = **-c**)

Compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link on the resulting .OBJ files.

Specify executable file name
(Command-line switch = ***-efilename***)

Link file using *filename* as the name of the executable file. If you do not specify an executable name with this option, the linker creates an executable file based on the name of the first source file or object file listed in the command.

Pass option to linker
(Command-line switch = **-l*x***)

Use this command-line option to pass option(s) *x* to the linker from a compile command. Use the command-line option **-l-*x*** to disable a specific linker option.

Create a MAP file
(Command-line switch = **-M**)

Use this compiler option to instruct the linker to create a map file.

Compile .OBJ to ***filename***
(Command-line switch = **-o*filename***)

Use this option to compile the specified source file to *filename*.OBJ.

C++ compile
(Command-line switch = **-P**)

The -P command-line option causes the compiler to compile all source files as C++ files, regardless of their extension. Use **-P-** to compile all .CPP files as C++ source files and all other files as C source files.

The command-line option **-P*ext*** causes the compiler to compile all source files as C++ files and it changes the default extension to whatever you specify with *ext*. This option is provided because some programmers use different extensions as their default extension for C++ code.

The option **-P-*ext*** compiles files based on their extension (.CPP compiles to C++, all other extensions compile to C) and sets the default extension (other than .CPP).

Generate a multi-threaded target
(Command-line switch = **-tWM**)

The compiler creates a multi-threaded .EXE or .DLL. (The command-line option **-WM** is supported for backward compatibility only; it has the same functionality as **-tWM**.)

- This option is not needed if you include a module definition file in your compile and link commands which specifies the type of 32-bit application you intend to build.

Autodependency information option

(Command-line switch: **-X-**)

When the Autodependency option is set (**-X-**), the compiler generates autodependency information for all project files with a .C or .CPP extension.

The Project Manager can use autodependency information to speed up compilation times. The Project Manager opens the .OBJ file and looks for information about files included in the source code. This information is always placed in the .OBJ file when the source module is compiled. After that, the time and date of every file that was used to build the .OBJ file is checked against the time and date information in the .OBJ file. The source file is recompiled if the dates are different. This is called an autodependency check.

If the project file contains valid dependency information, the Project Manager does the autodependency check using that information. This is much faster than reading each .OBJ file.

When this option is unchecked (**-X**), the compiler does not generate the autodependency information.

Modules compiled with autodependency information can use MAKE's autodependency feature.

Default = ON (**-X-**)

Generate underscores option

`(Command-line switch: -u)`

When Generate Underscores option is set, the compiler automatically adds an underscore character (_) in front of every global identifier (functions and global variables) before saving them in the object module. Pascal identifiers (those modified by the **_ _pascal** keyword) are converted to uppercase and are not prefixed with an underscore.

Underscores for C and C++ are optional, but you should turn this option on to avoid errors if you are linking with the Borland C++ libraries.

Default = ON

Compiler source options

Compiler source options tell the compiler how to interpret source code.

The options are

Source
Nested comments

Identifier Length

Language compliance
Borland extensions

ANSI

UNIX V

Kernighan and Ritchie

Compatibility
MFC compatibility

Nested comments option

`(Command-line switch: -C)`

When Nested Comments is on, you can nest comments in your C and C++ source files.

Nested comments are not allowed in standard C implementations, and they are not portable.

Default = OFF

Identifier length option

`(Command-line switch: `**`-i`*`n`***`,`*` where `***`n`***` = significant characters)`

Use Identifier Length to specify the number of significant characters (those which will be recognized by the compiler) in an identifier.

Except in C++, which recognizes identifiers of unlimited length, all identifiers are treated as distinct only if their significant characters are distinct. This includes variables, preprocessor macro names, and structure member names.

Valid numbers for *n* are 0, and 8 to 250, where 0 means use the maximum identifier length of 250.

By default, C++Builder uses 250 characters per identifier. Other systems (including some UNIX compilers) ignore characters beyond the first eight. If you are porting to other environments, you might want to compile your code with a smaller number of significant characters, which helps you locate name conflicts in long identifiers that have been truncated.

Default = 250

Language compliance options

The Language Compliance options tell the compiler how to recognize keywords in your programs.

Borland extensions

`(Command-line switches: -A-, -AT)`

Borland Extensions tells the compiler to recognize Borland's extensions to the C language keywords, including **near**, **far**, **huge**, **asm**, **cdecl**, **pascal**, **interrupt**, **_export**, **_ds**, **_cs**, **_ss**, **_es**, and the register pseudovariables (**_AX**, **_BX**, and so on). For a complete list of keywords, see the keyword index.

ANSI

`(Command-line switch: -A)`

The ANSI option compiles C and C++ ANSI-compatible code, allowing for maximum portability. Non-ANSI keywords are ignored as keywords.

UNIX V

`(Command-line switch: -AU)`

The UNIX V option tells the compiler to recognize only UNIX V keywords and treat any of Borland's C++ extension keywords as normal identifiers.

Kernighan and Ritchie

`(Command-line switch: -AK)`

The Kernighan and Ritchie option tells the compiler to recognize only the K&R extension keywords and treat any of Borland's C++ extension keywords as normal identifiers.

Hint: If you get declaration syntax errors from your source code, check that this option is set to Borland Extensions.

Default = Borland Extensions (**-A-**)

MFC compatibility option

See also

`(Command-line switches: -VF)`

Turn this option on to compile code that is compatible with the Microsoft foundation classes (MFC). Among other things, the compiler makes the following adjustments to be compatible with MFC:

- Accepts spurious semicolons in a class scope
- Allows anonymous structs
- Uses the old-style scoping resolution in **for** loops
- Allows methods to be declared with a calling convention, but leaves off the calling convention in the definition
- Tries the operator **new** if it cannot resolve a call to the operator **new[ ]**
- Lets you omit the operator & on member functions
- Allows a const class that is passed by value to be treated as a trivial conversion, not as a user conversion
- Allows you to use a cast to a member pointer as a selector for overload resolution, even if the qualifying type of the member pointer is not derived from the class in which the member function is declared
- Accepts declarations with duplicate storage in a class, as in
  `extern "C" typedef`
- Accepts and ignores `#pragma comment(linker, "...")` directives

Default = OFF

Compiler debugging options

See also

Compiler Debugging options affect the generation of debug information during compilation. When linking larger .OBJ files, you may need to turn these options off to increase the available system resources.

The options are

Standard stack frame

Out-of-line inline functions

Line numbers

Debug information in OBJs

Standard stack frame option

See also

`(Command-line switch: -k)`

When the Standard Stack Frame option is on, the compiler generates a standard stack frame (standard function entry and exit code). This is helpful when debugging, since it simplifies the process of stepping through the stack of called subroutines.

When this option is off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code smaller and faster.

The Standard Stack Frame option should always be on when you compile a source file for debugging.

Default = ON

Out-of-line inline functions option

See also

`(Command-line switch: -vi)`

When the Out-of-Line Inline Functions option is on, the compiler expands C++ inline functions inline.

To control the expansion of inline functions, the Debug Information In OBJs option (**`-v`**) acts slightly different for C++ code: when inline function expansion is disabled, inline functions are generated and called like any other function.

Because debugging with inline expansion can be difficult, the following command-line options are provided:

- **`-v`** turns debugging on and inline expansion off
- **`-v-`** turns debugging off and inline expansion on
- **`-vi`** turns inline function expansion on
- **`-vi-`** turns inline expansion off (inline functions are expanded out of line)

For example, if you want to turn both debugging and inline expansion on, use the **`-v`** and **`-vi`** options.

Default = OFF

Line numbers option

See also

`(Command-line switch: -y)`

When the Line Numbers option is on, the compiler automatically includes line numbers in the object and object map files. Line numbers are used by both the integrated debugger and Turbo Debugger.

Although the Debug Info in OBJs option (`-v`) automatically generates line number information, you can turn that option off (`-v-`) and turn on Line Numbers (`-y`) to reduce the size of the debug information generated. With this setup, you can still step, but you will not be able to watch or inspect data items.

Including line numbers increases the size of the object and map files but does not affect the speed of the executable program.

- When Line Numbers is on, make sure you turn off Pentium scheduling in the Compiler options. When this option is set, the source code will not exactly match the generated machine instructions, which can make stepping through code confusing.

Default = OFF

Debug information in OBJs option

See also

`(Command-line switch: -v)`

When the Debug Info In OBJs option is on, debugging information is included in your .OBJ files. The compiler passes this option to the linker so it can include the debugging information in the .EXE file. For debugging, this option treats C++ inline functions as normal functions.

You need debugging information to use either the integrated debugger or the standalone Turbo Debugger.

When this option is off (**`-v-`**), you can link and create larger object files. While this option does not affect execution speed, it does affect compilation and link time.

- When Line Numbers is on, make sure you turn off Pentium scheduling in the Compiler options. When this option is set, the source code will not exactly match the generated machine instructions, which can make stepping through code confusing.

Default = ON

Precompiled header options

Precompiled header files can dramatically increase compilation speed by storing an image of the symbol table on disk in a file, then later reloading that file from disk instead of parsing all the header files again. Directly loading the symbol table from disk is much faster than parsing the text of header files, especially if several source files include the same header file.

The options are

General usage

Cache precompiled header

Precompiled header name

Stop precompiling after header file

General precompiled header options

See also

Precompiled headers can dramatically increase compilation speeds, although they require a considerable amount of disk space.

Generate and use

`(Command-line switch: -H)`

When you set this option, the IDE generates and uses precompiled headers. The default file name is <*projectname*>.CSM for IDE projects, and is BC32DEF.CSM for command-line compiles.

Use but do not generate

(Command-line switch: **`-Hu`**)

When the Use But Do Not Generate option is set, the compilers use preexisting precompiled header files; new precompiled header files are not generated.

Do not generate or use

(Command-line switch: **`-H-`**)

When the Do Not Generate Or Use option is on, the compilers do not generate or use precompiled headers.

Default = Do not generate or use (**`-H-`**)

Cache precompiled header option

`(Command-line switch: -Hc)`

When you enable this option, the compiler caches the precompiled headers it generates. This is useful when you are precompiling more than one header file.

- To use this option, you must also enable the Generate and Use (**`-H`**) precompiled header option.

Default = OFF

Precompiled header name option

`(Command-line switch: `**`-H=`*`filename`***`)`

This option lets you specify the name of your precompiled header file. The compilers set the name of the precompiled header to *filename*.

When this option is set, the compilers generate and use the precompiled header file that you specify.

Stop precompiling after header file option

`(Command-line switch: `**`-H”`*`xxx`*`”`**`; for example -H”owl/owlpch.h”)`

This option terminates compiling the precompiled header after the compiler compiles the file specified as *xxx*. You can use this option to reduce the amount of disk space used by precompiled headers.

When you use this option, the file you specify must be included from a source file for the compiler to generate a .CSM file.

- You cannot specify a header file that is included from another header file. For example, you cannot list a header included by windows.h because this would cause the precompiled header file to be closed before the compilation of windows.h was competed.

Assembler-code options

Compile to .ASM, then assemble

(Command-line switch = **-B**)

This command-line option causes the compiler to first generate an .ASM file from your C++ (or C) source code (same as the **-S** command-line option). The compiler then calls TASM32 (or the assembler specified with the **-E** option) to create an .OBJ file from the .ASM file. The .ASM file is then deleted. To use this 32-bit compiler option, you must install a 32-bit assembler, such as TASM32.EXE, and then specify this assembler with the **-E** option.

▪ Your program will fail to compile with the **-B** option if your C or C++ source code declares static global variables that are keywords in assembly. This is because the compiler does not precede static global variables with an underscore (as it does other variables), and the assembly keywords will generate errors when the code is assembled.

Specify assembler

(Command-line switch = **-E*filename***)

Assemble instructions using *filename* as the assembler. The 32-bit compiler uses TASM32 as the default assembler.

Compile to assembler

(Command-line switch = **-S**)

This option causes the compiler to generate an .ASM file from your C++ (or C) source code. The generated .ASM file includes the original C or C++ source lines as comments in the file.

Specify assembler option

(Command-line switch = **-T*x***)

Use this command-line option to pass the option(s) *x* to the assembler you specify with the **-E** option. To disable all previously enabled assembler options, use the **-T-** command-line option.

Search directories

See also

The Source Directories options let you specify the directories that contain your standard include files, library and .OBJ files, and program source files.

Click the down-arrow icon or press Alt+Down arrow to display the history list of previously entered directory names.

Include

`(Command-line equivalent: -Ipath, where path = directory path)`

Use the Include list box to specify the drive and/or directories that contain program include files. Standard include files are those given in angle brackets (<>) in an #include statement (for example, `#include <myfile>`).

- The Borland compilers and linkers use specific file search algorithms to locate the files needed to complete the compilation and link cycles.

Library

`(Command-line equivalent: -Lpath, where path = directory path)`

Use the Library list box to specify the directories that contain the Borland C++ IDE startup object files (C0*x*.OBJ), run-time library files (.LIB files), and all other .LIB files. By default, the linker looks for them in the directory containing the project file (or in the current directory if you're using the command-line compiler).

- You can also use the linker option ***/Lpath*** to specify the library search directories when you link files from the command line.

Source

The Source list box specifies the directories where the compiler and the integrated debugger should look for your project source files.

Specifying multiple directories

Multiple directory names are allowed in each of the list boxes; use a semicolon (;) to separate the specified drives and directories. To display a history list of previously entered directory names, click the down-arrow icon or press Alt+Down arrow.

From the command line, you can enter multiple include and library directories in the following ways:

- You can stack multiple entries with a single **-L** or **-I** option by separating directories with a semicolon:

```
BCC.EXE -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
BCC.EXE -Ldirname1 -Ldirname2 -Iinc1 -Iinc2 -Iinc3 myfile.c
```

- You can mix listings:

```
BCC.EXE -Ldirname1;dirname2 -Iinc1 -Ld:dirname3 -Iinc2;inc3 myfile.c
```

If you list multiple **-L** or **-I** options on the command line, the result is cumulative; the compiler searches all the directories listed in order from left to right.

Guidelines for entering directory names

See Also

Use the following guidelines when entering directories in the Directories options page.

- You must separate multiple directory path names (if allowed) with a semicolon (;).
- You can use up to a maximum of 127 characters (including whitespace).
- Whitespace before and after the semicolon is allowed but not required.
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one.

For example,

```
C:\;C:..\BC5;D:\myprog\source
```

File search algorithms

See also

#include-file search algorithms

Borland C++ searches for files included in your source code with the **#include** directive in the following ways:

- If you specify a path and/or directory with your include statement, Borland C++ searches only the location specified. For example, if you have the following statement in your code:

  ```
  #include "c:\bc\include\owl\owl.h"
  ```

  the header file owl.h must reside in the directory C:\BC\INCLUDE\OWL. In addition, if you use the statement:

  ```
  #include <owl\owl.h>
  ```

  and you set the Include option (**-I**) to specify the path `c:\bc\include`, the file owl.h must reside in C:\BC\INCLUDE\OWL, and not in C:\BC\INCLUDE or C:\OWL.
- If you put an `#include <somefile>` statement in your source code, Borland C++ searches for "somefile" only in the directories specified with the Include (**-I**) option.
- If you put an `#include "somefile"` statement in your code, Borland C++ first searches for "somefile" in the current directory; if it does not find the file there, it then searches in the directories specified with the Include (**-I**) option.

Library file search algorithms

The library file search algorithms are similar to those for include files:

- Implicit libraries: Borland C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for `#include <somefile>`.

  Implicit library files are the ones Borland C++ automatically links in and the start-up object file (C0x.OBJ). To see these files in the Project Manager, turn on run-time nodes (choose Options|Environment|Project View, then check Show Runtime Nodes).
- Explicit libraries: Where Borland C++ searches for explicit (user-specified) libraries depends in part on how you list the library file name. Explicit library files are ones you list on the command line or in a project file; these are file names with a .LIB extension.
- If you list an explicit library file name with no drive or directory (like this: `mylib.lib`), Borland C++ first searches for that library in the current directory. If the first search is unsuccessful, Borland C++ looks in the directories specified with the Library (**-L**) option. This is similar to the search algorithm for `#include "somefile"`.
- If you list a user-specified library with drive and/or directory information (like this: `c:\mystuff\mylib1.lib`), Borland C++ searches only in the location you explicitly listed as part of the library path name and not in any specified library directories.

C++-specific options

The C++ options affect compilation of all C and C++ programs. For most of the C++ options, you'll usually want to use the default settings.

The subtopics are

C++ compatibility options

Virtual table options

Template options

Exception handling/RTTI options

C++ compatibility options

Use the C++ Compatibility options to handle C++ compatibility issues, such as handling **char** types, specifying options about hidden pointers, passing class arguments, adding hidden members and code to a derived class, passing the **this** pointer to **Pascal** member functions, changing the layout of classes, or insuring compatibility when class instances are shared with non-C++ code or code compiled with previous versions of Borland C++.

Don't restrict scope of 'for' loop expression variables

Calling convention mangling compatibility

Zero-length empty base classes

Virtual base pointers

Pass class values via reference to temporary

Disable constructor displacements

Push 'this' first for pascal member functions

'deep' virtual bases

Vtable pointer follows data members

Treat 'far' classes as 'huge'

Don't restrict scope of 'for' loop expression variables option

(Command-line switch: **-Vd**)

This option lets you specify the scope of variables declared in **for** loop expressions. The output of the following code segment changes, depending on the setting of this option.

```
int main(void)
{

  for(int i=0; i<10; i++)
  {
        cout << "Inside for loop, i = " << i << endl;
  }     //end of for-loop block

  cout << "Outside for loop, i = " << i << endl;  //error without -Vd

}       //end of block containing for loop
```

If this option is disabled (the default), the variable *i* goes out of scope when processing reaches the end of the **for** loop. Because of this, you'll get an Undefined Symbol compilation error if you compile this code with this option disabled.

If this option is set (**-Vd**), the variable *i* goes out of scope when processing reaches the end of the block containing the **for** loop. In this case, the code output would be:

```
Inside for loop, i = 0
...
Outside for loop, i = 10
```

Default = OFF

Calling convention mangling compatibility option

`(Command-line switch: -VC)`

When this option is set, the compiler disables the distinction of function names where the only possible difference is incompatible code generation options. For example, with this option set, the linker will not detect if a call is made to a **_ _fastcall** member function with the **cdecl** calling convention.

This option is provided for backward compatibility only; it lets you link old library files that you cannot recompile.

Default = OFF

Zero-length empty base classes

(Command-line switch: **-Ve**)

Usually the size of a class is at least one byte, even if the class does not define any data members. When you set this option, the compiler ignores this unused byte for the memory layout and the total size of any derived classes.

Default = OFF

Virtual base pointer options

When a class inherits virtually from a base class, the compiler stores a hidden pointer in the class object to access the virtual base class subobject.

The Virtual Base Pointers options specify options about the hidden pointer.

Always near

`(Command-line switch: `**`-Vb-`**`)`

When the Always Near option is set, the hidden pointer will always be a near pointer. (When a class inherits virtually from a base class, the compiler stores a *hidden pointer* in the class object to access the virtual base class subobject.)

This option allows for the smallest and most efficient code.

Same size as 'this' pointer

`(Command-line switch: `**`-Vb`**`)`

When the Same Size as **this** Pointer option is set, the compiler matches the size of the hidden pointer to the size of the **this** pointer in the instance class.

This allows for compatibility with previous versions of the compiler.

Default = Always Near (**`-Vb-`**)

Pass class values via reference to temporary option

(Command-line switch: **-Va**)

When this option is set, the compiler passes class arguments using the "reference to temporary" approach. When an argument of type class with constructors is passed by value to a function, this option instructs the compiler to create a temporary variable at the calling site, initialize this temporary variable with the argument value, and pass a reference from this temporary to the function.

This option insures compatibility with previous versions of the compiler.

Default = OFF

Disable constructor displacements option

`(Command-line switch: -Vc)`

When the Disable Constructor Displacements option is set, the compiler does not add hidden members and code to a derived class (the default).

This option insures compatibility with previous versions of the compiler.

Default = OFF

Push 'this' first for Pascal member functions option

`(Command-line switch: -Vp)`

When this option is set, the compiler passes the **this** pointer to Pascal member functions as the first parameter on the stack.

By default, the compiler passes the **this** parameter as the last parameter on the stack, which permits smaller and faster member function calls.

Default = OFF

'deep' virtual bases option

(Command-line switch: **-Vv**)

When a derived class overrides a virtual function which it inherits from a virtual base class, and a constructor or destructor for the derived class calls that virtual function using a pointer to the virtual base class, the compiler can sometimes add hidden members to the derived class. These “hidden members” add code to the constructors and destructors.

This option directs the compiler *not* to add the hidden members and code so that the class instance layout is the same as with previous versions of Borland C++; the compiler does not change the layout of any classes to relax the restrictions on pointers.

Default = OFF

Vtable pointer follows data members option

`(Command-line switch` **`-Vt`**`)`

When this option is set, the compiler places the virtual table pointer after any nonstatic data members of the specified class.

This option insures compatibility when class instances are shared with non-C++ code and when sharing classes with code compiled with previous versions of Borland C++.

Default = OFF

Treat 'far' classes as 'huge' option

(Command-line switch **-Vh**)

When this option is set, the compiler treats all classes declared **_ _far** as if they were declared as **_ _huge**. For example, the following code normally fails to compile. Checking this option allows the following code fragment to compile:

```
struct __huge A
{
virtual void f();  // A vtable is required to see the error.
};
struct __far B  :  public A
{
};
// Error: Attempting to derive a far class from the huge base 'A'.
```

Default = OFF

Virtual table options

The virtual tables options control C++ virtual tables and the expansion of inline functions when debugging.

Smart

(Command-line switch: **-V**)

This option generates common C++ virtual tables and out-of-line inline functions across the modules in your application. As a result, only one instance of a given virtual table or out-of-line inline function is included in the program.

The Smart option generates the smallest and most efficient executables, but produces .OBJ and .ASM files compatible only with TLINK and TASM.

Default = ON

Local

(Command-line switch: **-Vs**)

You use the Local option to generate local virtual tables (and out-of-line inline functions) so that each module gets its own private copy of each virtual table or inline function it uses.

The Local option uses only standard .OBJ and .ASM constructs, but produces larger executables.

Default = OFF

External

(Command-line switch: **-V0**)

You use the External option to generate external references to virtual tables. If you don't want to use the Smart or Local options, use the External and Public options to produce and reference global virtual tables.

- When you use this option, one or more of the modules comprising the program must be compiled with the Public option to supply the definitions for the virtual tables.

Default = OFF

Public

(Command-line switch: **-V1**)

Public produces public definitions for virtual tables. When using the External option (**-V0**), at least one of the modules in the program must be compiled with the Public option to supply the definitions for the virtual tables. All other modules should be compiled with the External option to refer to that Public copy of the virtual tables.

Default = OFF

Template options

See also

The Template options specify how the compiler generates template instances in C++.

Smart

`(Command-line switch: -Jg)`

When the Smart option is set, the compiler generates public (global) definitions for all template instances. If more than one module generates the same template instance, the linker automatically merges duplicates to produce a single copy of the instance.

To generate the instances, the compiler must have available the function body (in the case of a template function) or the bodies of member functions and definitions for static data members (in the case of a template class), typically in a header file.

This is a convenient way of generating template instances.

Default = ON

Global

`(Command-line switch: -Jgd)`

When the Global option is set, the compiler generates public (global) definitions for all template instances.

The Global option does not merge duplicates. If the same template instance is generated more than once, the linker reports public symbol re-definition errors.

Default = OFF

External

`(Command-line switch: -Jgx)`

When the External option is set, the compiler generates external references to all template instances.

When you use this option, all template instances in your code must be publicly defined in another module with the external option (**`-Jgd`**) so that external references are properly resolved.

Default = OFF

Exception handling/RTTI options

Use the Exceptions Handling options to enable or disable exception handling and to tell the compiler how to handle the generation of run-time type information.

If you use exception handling constructs in your code and compile with exceptions disabled, you'll get an error.

The options are

Enable exceptions

Exception handling options

Enable run-time type information

Enable exceptions option

`(Command-line switch: -x)`

When this option is set, C++ exception handling is set. If this option is disabled (**`-x-`**) and you attempt to use exception handling routines in your code, the compiler generates error messages during compilation.

Disabling this option makes it easier for you to remove exception handling information from programs; this might be useful if you are porting your code to other platforms or compilers.

- Disabling this option turns off only the compilation of exception handling code; your application can still include exception code if you link .OBJ and library files that were built with exceptions enabled (such as the Borland runtime libraries).

Default = ON

Exception handling options

Enable exception location information

`(Command-line switch: -xp)`

When this option is set, run-time identification of exceptions is available because the compiler provides the file name and source-code line number where the exception occurred. This enables the program to query file and line number from where a C++ exception was thrown.

Default = OFF

Enable destructor cleanup

`(Command-line switch: -xd)`

When this option is setand an exception is thrown, destructors are called for all automatically declared objects between the scope of the catch and throw statements.

In general, when you set this option, you should also set Enable Runtime Type Information (**`-RT`**) as well.

- Destructors are not automatically called for dynamic objects allocated with **new**, and dynamic objects are not automatically freed.

Default = ON

Enable fast exception prologs

`(Command-line switch: -xf)`

When this option is set, inline code is expanded for every exception handling function. This option improves performance at the cost of larger executable file sizes.

- If you set both Fast Exception Prologs and Enable Compatible Exceptions (**`-xc`**), fast prologs will be generated but Enable Compatible Exceptions will be disabled (the two options are not compatible).

Default = OFF

Enable run-time type information option

`(Command-line switch: -RT)`

This option causes the compiler to generate code that allows run-time type identification.

In general, if you set Enable Destructor Cleanup (**`-xd`**), you will need to set this option as well.

Default = ON

Linker options

Linker options let you control how the linkers combine intermediate files (.OBJ, .LIB, and .RES) into executable (.EXE) and dynamic-link library (.DLL) files. You will usually want to keep the default settings for most options in this section.

The linker options can be broken into the following subtopics

Linker input options

The following options are available to control the input to your link cycles:

Case-sensitive link

Default libraries

Allow import by ordinal

Object search paths

Case-sensitive link

`(Command-line switch = /c)`

When the Case-Sensitive Link option is set, the linker differentiates between upper and lower-case characters in public and external symbols. Normally, this option should be checked, since C and C++ are both case-sensitive languages.

Default = ON

Default libraries

`(Command-line switch = /n)`

Some compilers, other than C++Builder, place a list of libriaries needed by their modules in their compiled .OBJ modules.

When this option is set, the linker searches for the default libraries specified by C++Builder and ignores any default librearies specified in the .OBJ files.

When this option is disabled, the linker tries to find any undefined routines in the linked .OBJ modules in addition to the default libraries supplied by C++Builder. You might need to disable this option when linking modules written in another language.

Default = ON

Allow import by ordinal

`(Command-line switch = `**`/o`**`)`

This option lets you import by ordinal value instead of by the import name. When you specify this option, the linker emits only the ordinal numbers (and not the import names) to the resident or nonresident name table for those imports that have an ordinal number specified. If you do not specify this option, the linker ignores all ordinal numbers contained in import libraries or the .DEF file, and emits the import names to the resident and nonresident tables.

Object search paths

See also

`(Command-line switch = **/j**)`

This option lets you specify the directories the linker will search if there is no explicit path given for an .OBJ module in the compile/link statement.

The Specify Object Search Path uses the following command-line syntax:

```
  /j<PathSpec>[;<PathSpec>][...]
```

The linker uses the specified object search path(s) if there is no explicit path given for the .OBJ file and the linker cannot find the object file in the current directory. For example, the command

```
  TLINK32 /jc:\myobjs;.\objs splash .\common\logo,,,utils logolib
```

directs the linker to first search the current directory for SPLASH.OBJ. If it is not found in he current directory, the linker then searches for the file in the C:\MYOBJS directory, and then in the .\OBJs directory. However, notice that the linker does not use the object search paths to find the file LOGO.OBJ because an explicit path was given for this file.

Map file options

Linker map file options tell the linker what type of map file to produce. These options control the information generated on segment ordering, segment sizes, and public symbols.

By default, the linker creates a map file that contains general segment information, which includes a list of segments, the program start address, and any warning or error messages produced during the link.

For settings other than Off, the map file is placed in the directory where the project makefile is stored.

The map file options are:

Off (default Segments map file not created)

Publics map

Detailed segments map

Mangled names

Off option (map file)

`(Command-line switch = `**`/x`**`)`

By default, the linker generates a map file with that contains general segment information that includes a list of segments, the program start address, and any warning or error messages produced during the link. There is no switch for this setting. Use the **`/x`** option to suppress the creation of the default map file.

Default = OFF (Default map file is created)

Publics map option

`(Command-line switch = /m)`

This option causes the linker to produce a map file that contains an overview of the application segments and two listings of the public symbols. The segments listing has a line for each segment, showing the segment starting address, segment length, segment name, and the segment class. The public symbols are broken down into two lists, the first showing the symbols in sorted alphabetically, the second showing the symbols in increasing address order. Symbols with absolute addresses are tagged `Abs`.

A list of public symbols is useful when debugging: many debuggers use public symbols, which lets you refer to symbolic addresses while debugging.

Detailed segments map option

(Command-line switch = **/s**)

The Detailed Segments option creates the most comprehensive map file by adding a detailed map of segments to the map file created with the Publics option (**/m**). The detailed list of segments contains the segment class, the segment name, the segment group, the segment module, and the segment ACBP information. If the same segment appears in more than one module, each module appears as a separate line.

The ACBP field encodes the A (alignment), C (combination), and B (big) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields: A, C, and B. The ACBP value in the map is printed in hexadecimal. The following field values must be ORed together to arrive at the ACBP value printed.

Field	Value	Description
A (alignment)	00	An absolute segment
	20	A byte-aligned segment
	40	A word-aligned segment
	60	A paragraph-aligned segment
	80	A page-aligned segment
	A0	An unnamed absolute portion of storage
C (combination)	00	Cannot be combined
	08	A public combining segment
B (big)	00	Segment less than 64K
	02	Segment exactly 64K

With the Segments options set, public symbols with no references are flagged `idle`. An idle symbol is a publicly defined symbol in a module that was not referenced by an EXTDEF record or by any other module included in the link. For example, this fragment from the public symbol section of a map file indicates that symbols `Symbol1` and `Symbol3` are not referenced by the image being linked:

```
0002:00000874      Idle      Symbol1
0002:00000CE4                Symbol2
0002:000000E7      Idle      Symbol3
```

Mangled names option (map file)

`(Command-line switch = `**`/M`**`)`

Prints the mangled C++ identifiers in the map file, not the full name. This can help you identify how names are mangled (mangled names are needed as input by some utilities).

Default = OFF

Linker input options

The following linker output options are available:

Pack code segments

Include debug information

Subsystem version (major.minor)

Pack code segments

`(Command-line switch = /P)`

When this option is set., the linker pack all code into one “segment.” The command-line option to turn this off is **`/P-`**.

Default = ON

Include debug information

(Command-line switch = **/v**)

When the Include Debug Information option is on, the linker includes information in the output file needed to debug your application with the C++Builder integrated debugger or Turbo Debugger.

On the command line, this option causes the linker to include debugging information in the executable file for all object modules that contain debugging information. You can use the **/v+** and **/v-** options to selectively enable or disable debugging information on a module-by-module basis (but not on the same command-line where you use **/v**). For example, the following command includes debugging information for modules `mod2` and `mod3`, but not for `mod1` and `mod4`:

```
  TLINK32 mod1 /v+ mod2 mod3 /v- mod4
```

Default = ON in IDE; OFF on the command line

Subsystem version (major.minor)

(`Command-line switch = `***`/Vd.d`***)

This option lets you specify the Windows version ID on which you expect your application will be run. The linker sets the Subsystem version field in the .EXE header to the number you specify in the input box.

You can also set the Windows version ID in the SUBSYSTEM portion of the module definition file (.DEF file) However, any version setting you specify in the IDE or on the command line overrides the setting in the .DEF file.

Command-line usage

When you use the ***`/Vd.d`*** command-line option, the linker sets the Windows version ID to the number specified by ***`d.d`***. For example, if you specify `/V4.0`, the linker sets the Subsystem version field in the .EXE header to 4.0, which indicates a Windows 95 application.

Default = 3.1

Linker file addressing options

The following linker output file addressing options are available:

Image base address (in hexadecimal)

File alignment (in hexadecimal)

Object alignment (in hexadecimal)

Reserved stack size (in hexadecimal)

Commit stack size (in hexadecimal)

Reserved heap size (in hexadecimal)

Commit heap size (in hexadecimal)

Image base address (in hexadecimal)

(Command-line switch = **/B:*xxxx***)

The Image Base Address option specifies an image base address for an application, and is used in conjunction with the Image is based option. If this setting is turned on, internal fixes are removed from the image and the requested load address of the first object in the application is set to the hexadecimal number specified. All successive objects are aligned on 64K linear address boundaries. This option makes applications smaller on disk and improves both load-time and run-time performance (the operating system no longer has to apply internal fixes).

The command-line version of this option (**/B:*xxxx***) accepts either decimal or hexadecimal numbers as the image base address.

▪ It is not recommended that you enable this option when producing a DLL. In addition, do not use the default setting of 0x400000 if you intend to run your application of Win32s systems.

Default = 0x400000 (recommended for true Win32 system applications)

File alignment (in hexadecimal)

(Command-line switch = **/Af:*xxxx***)

The File Alignment option specifies page alignment for code and data within the executable file. The linker uses the file alignment value when it writes the various objects and sections (such as code and data) to the file. For example, if you use the default value of 0x200, the linker stores the section of the image on 512-byte boundaries within the executable file.

When using this option, you must specify a file alignment value that is a power of 2, with the smallest value being 16.

- The old style of this option (**/A:*dd***) is still supported for backward compatibility. With this option, the decimal number ***dd*** is multiplied by the power of 2 to calculate the file alignment value.

The command-line version of this option (**/Af:*xxxx***) accepts either decimal or hexadecimal numbers as the file alignment value.

Default = 512 (0x200)

Object alignment (in hexadecimal)

`(Command-line switch =` **`/Ao:`*****`xxxx`***`)`

The linker uses the object alignment value to determine the virtual addresses of the various objects and sections (such as code and data) in your application. For example, if you specify an object alignment value of 8192, the linker aligns the virtual addresses of the sections in the image on 8192-byte (0x2000) boundaries.

When using this option, you must specify an object alignment value that is a power of 2, with the smallest value being 4096 (the default).

The command-line version of this option (**`/Ao:`*****`xxxx`***) accepts either decimal or hexadecimal numbers as the object alignment value.

Default = 4096 (0x1000)

Reserved stack size (in hexadecimal)

(Command-line switch = **/S:*xxxx***)

Specifies the size of the reserved stack in hexadecimal. The minimum allowable value for this field is 4K (0x1000).

▪ Specifying the reserved stack size here overrides any STACKSIZE setting in a module definition file.

The command-line version of this option (**/S:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 1Mb (0x1000000)

Committed stack size (in hexadecimal)

(Command-line switch = **/Sc:*xxxx***)

Specifies the size of the committed stack in hexadecimal. The minimum allowable value for this field is 4K (0x1000) and any value specified must be equal to or less than the Reserved Stack Size setting (`/S`).

- Specifying the committed stack size here overrides any STACKSIZE setting in a module definition file.

The command-line version of this option (**/Sc:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 8K (0x2000)

Reserved heap size (in hexadecimal)

`(Command-line switch = `**`/H:`*`xxxx`***`)`

Specifies the size of the reserved heap in hexadecimal. The minimum allowable value for this field is 0.

- Specifying the reserved heap size here overrides any HEAPSIZE setting in a module definition file.

The command-line version of this option (**`/H:`*`xxxx`***) accepts hexadecimal numbers as the stack reserve value.

Default = 1Mb (0x1000000)

Committed stack size (in hexadecimal)

(Command-line switch = **/Hc:*xxxx***)

Specifies the size of the committed heap in hexadecimal. The minimum allowable value for this field is 0 and any value specified must be equal to or less than the Reserved Heap Size setting (**/H**).

- Specifying the committed heap size here overrides any HEAPSIZE setting in a module definition file.

The command-line version of this option (**/Hc:*xxxx***) accepts hexadecimal numbers as the stack reserve value.

Default = 4K (0x1000)

Application target options

See also

The following options tell the linkers what type of application image to build.

▪ The following options are not needed if you include a module definition file in your compile and link commands that specifies the type of 32-bit application you intend to build.

Link using 32-bit Windows API

(Command-line switch = **/aa**)

The linker generates a protected-mode executable that runs using the 32-bit Windows API.

Link for 32-bit console application

(Command-line switch = **/ap**)

The linker generates a protected-mode executable file that runs in console mode.

Link 32-bit .DLL file

(Command-line switch = **/Tpd**)

The linker generates a 32-bit protected-mode Windows .DLL file.

Link 32-bit .EXE file

(Command-line switch = **/Tpe**)

The linker generates a 32-bit protected-mode Windows .EXE file.

Linker messages and warnings

The following options control the messages and warnings that the linker generates while it links:

Verbose

Maximum linker errors

Linker warnings

Verbose

(Command-line switch = **/r**)

This option causes the linker to emit messages that indicate what part of the link cycle is currently being executed by the linker. With this option turned on, the linker emits some or all of the following messages:

```
  Starting pass 1
  Generating map file
  Starting pass 2
  Reading resource files
  Linking resources
```

Maximum linker errors

`(Command-line switch = `**`/E`*`nn`***`)`

Specifies maximum errors the linker reports before terminating. **`/E0`** (default) reports an infinite number of errors (that is, as many as possible).

Linker warnings

Warnings options enable or disable the display of linker warnings. The warnings are

Warn duplicate symbol in .LIB

"No Stack" warning

No entry point

Duplicate symbol

No def file

Import does not match previous definition

Extern not qualified with _import

Using based linking in DLL

Self-relative fixup overflowed

.EXE module built with a .DLL extension

Multiple stack segments found

Warn duplicate symbol in .LIB

See Also

(`Command-line switch = `**`/wdpl`**)

When the Warn Duplicate Symbols option is on, the linker warns you if a symbol appears in more than one object or library files.

If the symbol must be included in the program, the linker uses the symbol definition from the first file it encounters with the symbol definition.

Use the command-line option **`/w-dpl`** to turn this warning off.

Default = OFF

"No stack" warning

(Command-line switch = **`/wstk`**)

This option lets you control whether or not the linker emits the "No stack" warning. The warning is generated if no stack segment is defined in any of the object files or in any of the libraries included in the link. Except for .DLLs, this indicates an error. If a C++Builder program produces this error, make sure you are using the correct C0*x* startup object file.

Use the command-line option **`/w-stk`** to turn this warning off.

Default = OFF

Image is based

Image is based

The Image is Based option affects whether an application has an image base address. If this setting is turned on, internal fixes are removed from the image and the requested load address of the first object in the application is set to the number specified in the Image Base Address input box. Using this option can greatly reduce the size of your final application module; however, it is not recommended for use when producing a DLL.

Default = OFF

Warning message options

The Warning Message Options let you control the messages generated by the compiler. Messages are indicators of potential trouble spots in your program. These messages can warn you of many problems that may be waiting to happen, such as variables and parameters that are declared but never used, type mismatches, and many others.

Setting a message option causes the compiler to generate the associated message or warning when the specific condition arises. Note that some of the messages are on by default.

The different compiler messages can be broken in to the following subtopics

Portability

ANSI violations

Obsolete C++

Potential C++ errors

Inefficient C++ coding

Potential errors

Inefficient coding

General

User-defined warnings

Displaing warnings

Selecting warnings

Stop after n warnings

Stop after n errors

Portability message options

Portability options enable or disable individual warning messages about statements that might not operate correctly in all computer environments.

Option	Command-line switch	Default
Non-portable pointer conversion	`-w-rpt`	ON
Non-portable pointer comparison	`-w-cpt`	ON
Constant out of range in comparison	`-w-rng`	ON
Constant is long	`-wcln`	OFF
Conversion may lose significant digits	`-wsig`	OFF
Mixing pointers to signed and unsigned char	`-wucp`	OFF

ANSI violation message options

ANSI violations message options enable or disable individual warning messages about statements that violate the ANSI standard for the C language.

Option	Command-line switch	Default
Void functions may not return a value	`-w-voi`	ON
Both return and return of a value used	`-w-ret`	ON
Suspicious pointer conversion	`-w-sus`	ON
Undefined structure 'ident'	`-wstu`	OFF
Redefinition of 'ident' is not identical	`-w-dup`	ON
Hexadecimal value more than three digits	`-w-big`	ON
Bit fields must be signed or unsigned int	`-wbbf`	OFF
'ident' declared as both external and static	`-w-ext`	ON
Declare 'ident' prior to use in prototype	`-w-dpu`	ON
Division by zero	`-w-zdi`	ON
Initializing 'ident' with 'ident'	`-w-bei`	ON
Initialization is only partially bracketed	`-wpin`	OFF
Non-ANSI keyword used	`-wnak`	OFF

Obsolete C++ message options

Obsolete C++ message options let you choose the obsolete items or incorrect syntax C++ warnings display.

Option	Command-line switch	Default
Base initialization without class name is obsolete	**`-w-obi`**	ON
This style of function definition is obsolete	**`-w-ofp`**	ON
Overloaded prefix operator used as a postfix operator	**`-w-pre`**	ON

Potential C++ error message options

Potential C++ errors message options enable or disable individual warning messages about statements that violate C++ language implementation.

Option	Command-line switch	Default
Constant member 'ident' is not initialized	`-w-nci`	ON
Assigning 'type' to 'enumeration'	`-w-eas`	ON
'function' hides virtual function 'function2'	`-w-hid`	ON
Non-const function <function> called for const object	`-w-ncf`	ON
Base class 'ident' inaccessible because also in 'ident'	`-w-ibc`	ON
Array size for 'delete' ignored	`-w-dsz`	ON
Use qualified name to access nested type 'ident'	`-w-nst`	ON
Handler for '<type1>' Hidden by Previous Handler for '<type2>'	`-w-hch`	ON
Conversion to 'type' will fail for virtual base members	`-w-mpc`	ON
Maximum precision used for member pointer type <type>	`-w-mpd`	ON
Use '> >' for nested templates instead of '>>'	`-w-ntd`	ON
Non-volatile function <function> called for volatile object	`-w-nvf`	ON

Inefficient C++ coding message options

Inefficient C++ coding message options enable or disable individual warning messages about inefficient C++ coding.

Option	Command-line switch	Default
Functions containing 'ident' not expanded inline	`-w-inl`	ON
Temporary used to initialize 'ident'	`-w-lin`	ON
Temporary used for parameter 'ident'	`-w-lvc`	ON

Potential error message options

Potential error message options enable or disable individual warning messages about potential coding errors.

Option	Command-line switch	Default
Possibly incorrect assignment	`-w-pia`	ON
Possible use of 'ident' before definition	`-wdef`	OFF
No declaration for function 'ident'	`-wnod`	OFF
Call to function with no prototype	`-w-pro`	ON
Function should return a value	`-w-rvl`	ON
Ambiguous operators need parentheses	`-wamb`	OFF
Condition is always (true/false)	`-w-ccc`	ON
Continuation character \ found in //	`-w-com`	ON

Inefficient coding message options

Inefficient coding message options are used to enable or disable individual warning messages about inefficient coding.

Option	Command-line switch	Default
'ident' assigned a value which is never used	**`-w-aus`**	ON
Parameter 'ident' is never used	**`-w-par`**	ON
'ident' declared but never used	**`-wuse`**	OFF
Structure passed by value	**`-wstv`**	OFF
Unreachable code	**`-w-rch`**	ON
Code has no effect	**`-w-eff`**	ON

▪ The warnings `Unreachable Code` and `Code Has No Effect` can indicate serious coding problems. If the compiler generates these warnings, be sure to examine the lines of code that cause these warnings.

General message options

General message options enable or disable a few general warning messages.

Option	Command-line switch	Default
Unknown assembler instruction	`-wasm`	OFF
Ill-formed pragma	`-w-ill`	ON
Array variable 'ident' is near	`-w-ias`	ON
Superfluous & with function	`-wamp`	OFF
'ident' is obsolete	`-w-obs`	ON
Cannot create precompiled header	`-w-pch`	ON
User-defined warnings	`-w-msg`	ON

User-defined warnings

(Command-line switch: -**wmsg**)

The User-defined Warnings option allows user-defined messages to appear in the IDE's Message window. User-defined messages are introduced with the #pragma message compiler syntax.

- In addition to messages that you introduce with the `#pragma` message compiler syntax, User-defined warnings displays warnings introduced by third-party libraries. If you need Help concerning a message generated by third-party libraries, please contact the vendor of the header file that issued the warning.

Default = ON

Display warnings options

Use the Display Warnings options to choose which warnings are displayed by the compiler.

All
(Command-line switch: `-w`)

Display all warning and error messages.

Default = OFF

Selected
(Command-line switch: -***waaa***)

Choose which warnings are displayed. Using `pragma warn` in your source code overrides messages options set either at the command line or in the IDE.

To disable a message from the command line, use the command-line option **-w-*aaa***, where ***aaa*** is the 3-letter message identifier used by the command-line option.

Default = Refer to the individual messages for their default status

Stop after *n* warnings

(Command-line switch: **`-g`*`n`***)

Warnings: Stop After causes compilation to stop after the specified number of warnings has been detected. You can enter any number from 0 to 255.

Entering 0 causes compilation to continue until either the end of the file or the error limit set in Stop after n errors has been reached, whichever comes first.

Default = 100

Stop after *n* errors

(Command-line switch: **`-j`*`n`***)

Errors: Stop After causes compilation to stop after the specified number of errors has been detected. You can enter any number from 0 to 255.

Entering 0 causes compilation to continue until the end of the file or the warning limit set in Stop after n warnings has been reached, whichever comes first.

Default = 25

Optimization options

See also

Optimization options are the software equivalent of performance tuning. There are two general types of compiler optimizations:

- Those that make your code smaller
- Those that make your code faster

Although you can compile with optimizations at any point in your product development cycle, be aware when debugging at the assembly level that some assembly instructions might be "optimized away" by certain compiler optimizations.

General settings

The main Optimizations page in the Project Options dialog box contains four radio buttons that let you select the overall type of optimizations you want to use. Because of the complexities of setting compiler optimizations, it is recommended that you use either the Optimize for Size or the Optimize for Speed radio buttons. The general optimization settings are:

Disable all optimizations

Optimize for size

Optimize for speed

General

Inline intrinsic functions

Induction variables

Aggregate optimization options

Disable all optimizations
(Command-line switch: **`-Od`**)

Disables all optimization settings, including ones which you may have specifically set and those which would normally be performed as part of the speed/size tradeoff.

Because this disables code compaction (tail merging) and cross-jump optimizations, using this option can keep the debugger from jumping around or returning from a function without warning, which makes stepping through code easier to follow.

Optimize for size
(Command-line switches: **`-O1`**)

This option sets an aggregate of optimization options that tells the compiler to optimize your code for size. For example, the compiler scans the generated code for duplicate sequences. When such sequences warrant, the optimizer replaces one sequence of code with a jump to the other and eliminates the first piece of code. This occurs most often with **switch** statements. The compiler optimizes for size by choosing the smallest code sequence possible.

This option (**`-O1`**) sets the following optimizations:

- Jump optimizations (**`-O`**)
- Duplicate expressions (**`-Oc`**)
- Instruction scheduling (**`-OS`**)
- The compiler options **`-Ot`** and **`-G`** are supported for backward compatibility only, and are equivalent to the **`-O1`** compiler option.

Optimize for speed
(Command-line switch: **`-O2`**)

This radio button sets an aggregate of optimization options that tells the compiler to optimize your code for speed. This switch (**`-O2`**) sets the following optimizations:

- Duplicate expression within functions (**`-Oc`**)
- Intrinsic functions (**`-Oi`**)
- Instruction scheduling (**`-OS`**)
- Induction variables (**`-Ov`**)

If you are creating Windows applications, you'll probably want to optimize for speed.

- The compiler options **`-Os`** and **`-G-`** are supported for backward compatibility only, and are equivalent to the **`-O2`** compiler option. The **`-Ox`** option is also supported for backward compatibility and for compatibility with Microsoft make files.

Jump optimization option

`(Command-line switch: -O)`

When Jump Optimization option is set, the compiler reduces the code size by eliminating redundant jumps and reorganizing loops and switch statements.

When this option is set, the sequences of stepping in the debugger can be confusing because of the reordering and elimination of instructions. If you are debugging at the assembly level, you might want to disable this option.

Default = ON

Eliminate duplicate expressions option

(Command-line switch: **-Oc**)

When this option is set, the compiler eliminates common subexpressions within groups of statements unbroken by jumps (basic blocks) and functions. This option globally eliminates duplicate expressions within the target scope and stores the calculated value of those expressions once (instead of recalculating the expression).

Although this optimization could theoretically reduce code size, it optimizes for speed and rarely results in size reductions. Use this option if you prefer to reuse expressions rather than create explicit stack locations for them.

- The **-Og** compiler option is supported for backward compatibility only, and is equivalent to the **-Oc** compiler option.

Inline intrinsic functions option

(`Command-line switch:` **`-Oi`**)

When Inline Intrinsic Functions is set, the compiler generates the code for common memory functions like strcpy() within your function's scope. This eliminates the need for a function call. The resulting code executes faster, but it is larger.

The following functions are inlined with this option:

alloca	*fabs*	*memchr*	*memcmp*
memcpy	*memset*	*rotl*	*rotr*
stpcpy	*strcat*	*strchr*	*strcmp*
strcpy	*strlen*	*strncat*	*strncmp*
strncpy	*strnset*	*strrchr*	

You can control the inlining of these functions with the pragma **intrinsic**. For example, `#pragma intrinsic strcpy` causes the compiler to generate inline code for all subsequent calls to **strcpy** in your function, and `#pragma intrinsic -strcpy` prevents the compiler from inlining **strcpy**. Using these pragmas in a file overrides any compiler option settings.

When inlining any intrinsic function, you must include a prototype for that function before you use it; the compiler creates a macro that renames the inlined function to a function that the compiler recognizes internally. In the previous example, the compiler would create a macro `#define strcpy _ _strcpy_ _`.

The compiler recognizes calls to functions with two leading and two trailing underscores and tries to match the prototype of that function against its own internally stored prototype. If you don't supply a prototype, or if the prototype you supply doesn't match the compiler's prototype, the compiler rejects the attempt to inline that function and generates an error.

Induction variables option

(Command-line switch: **-Ov**)

When this option is set, the compiler creates induction variables and it performs strength reduction, which optimizes for loops speed.

Use this option when you're compiling for speed and your code contains loops. The optimizer uses induction to create new variables (induction variables) from expressions used in loops. The optimizer assures that the operations performed on these new variables are computationally less expensive (reduced in strength) than those used by the original variables.

Optimizations are common if you use array indexing inside loops, because a multiplication operation is required to calculate the position in the array that is indicated by the index. For example, the optimizer creates an induction variable out of the operation `v[i]` in the following code because the `v[i]` operation requires multiplication. This optimization also eliminates the need to preserve the value of `i`:

```
int v[10];
void f(int x, int y, int z)
{
  int i;
  for (i = 0; i < 10; i++)
     v[i] = x * y * z;
}
```

With Induction variables set, the code changes:

```
int v[10];
void f(int x, int y, int z)
{
  int i, *p;
  for (p = v; p < &v[9]; p++)
    *p = x * y * z;
}
```

Pentium instruction scheduling option

(Command-line switch: **-OS**)

When set, this switch rearranges instructions to minimize delays that can be caused by Address Generation Interlocks (AGI) which occur on the i486 and Pentium processors. This option also optimizes the code so that it takes advantage of the Pentium parallel pipelines. Best results for Pentium systems are obtained when you use this switch in conjunction with the Generate Pentium Instructions option (**-5**).

- Scheduled code is more difficult to debug at the source level because instructions from a particular source line may be mixed with instructions from other source lines. Stepping through the source code is still possible, although the execution point might make unexpected jumps between source lines as you step. Also, setting a breakpoint on a source line may result in several breakpoints being set in the code. This is especially important to note when inspecting variables, since a variable may be undefined even though the execution point is positioned after the variable assignment.

Stepping through the following function when this switch is set demonstrates the stepping behavior:

```
int v[10];
void f(int i, int j)
{
  int a,b;

  a = v[i+j];
  b = v[i-j];
  v[i] = a + b;
  v[j] = a - b;
}
```

Execution starts by computing the index `i-j` in the assignment to `b` (note that `a` is still undefined although the execution point is positioned after the assignment to `a`). The index `i+j` is computed, `v[i-j]` is assigned to `b`, and `v[i+j]` is assigned to `a`. If a breakpoint is set on the assignment to `b`, execution will stop twice: once when computing the index and again when performing the assignment.

Default = OFF (**-O-S**)

Glossary

* A B C D E F G H I J K L
M N O P Q R S T U V W X Y Z

A

abstract
accelerator key
actual parameter
actual variable
alias
ancestor
application
array
ASCII

B

base type
batch operation
BDE
BDE Configuration Utility
BLOB
block
Boolean
Borland Database Engine
breakpoints
byte

C

callback routines
call stack
canvas
case variant
char
child
class
class method
client
client area
column
compile
compiler directive
compile time
compile-time error
complete evaluation
component
conditional symbol
const parameter

constant
constant address expression
container application
container component
control

D
data
data access component
data-aware
data control component
data type
database
database server
dataset
DDE client
DDE conversation
DDE server
default ancestor
default event
default new form
default new project
derive
descend
descendant
design time
detail table
dispatch
drag
dynamic
dynamic data exchange (DDE)
dynamic-link library (DLL)

E
embedding
encapsulate
end user
enumerated data type
exception
exception handler
execution point
expressions
event
event handler

F
feature

field
file buffer
file type
filter
filter program
focus
form
formal parameter
function
function header

G
global heap
global variable
glyph
grandchild
grandparent
grid

H
handling exceptions
header
heap
heap suballocator
help context
Hint
host type

IJK
IDAPI
identifier
implementation
include file
index
index type
inheritance
instance
integer
integrated debugger
interface
key

L
label
language driver
late binding
linking
literal value

local heap
local symbol information
local variable
lock
logic error
Longint
lookup table
loop

M
main form
master table
method
method identifier
method pointer
MDI application
modal
modeless
module

N
nil
nonvisual component
nonwindowed control

O
object file
object instance variable
object type
OLE
OLE container
OLE object
OLE server
ordinal
override
owner

P
parameter
parent
pixel
pointer
power set
primary index
private
private part
procedure
procedure header

program
project
project directory
project file
property
protected
protected block
public
published

Q
qualified identifier
qualified method identifier
qualifier
query

R
raise
real
record
record type
recursion
relational database
report
root class
routine
row
run time
run-time error
run-time library
run-time only

S
scalar type
scope
separator
separator bar
set
short-circuit evaluation
Shortint
sizing handles
source code
splash screen
SQL
SQL table
stack
statement

static
step over
string
string list
subrange
switch directive
symbol

T
table
tag field
template
trace into
typecasting
type
type definition
typed constant

U
unit
untyped file
untyped pointer
unqualified identifier
use count
user-defined

V
value parameter
variable parameter
variable
virtual
visual component

W
warning
watches
window handle
windowed control
word
wrapper

XYZ
z-order

abstract
A method that is declared but not implemented. Descendant types must override the abstract method.

accelerator key
Accelerator keys enable the user to access a menu command or component from the keyboard, by pressing Alt+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu or component caption.

actual parameter
A variable, expression, or constant that is substituted for a formal parameter in a procedure or function call.

actual variable
A variable that a program can use at run time, as distinguished from the definition of that variable within the program. A location in memory used for storage purposes, as distinguished from an identifier.

alias
A name that specifies the location of database tables. If the database is on a server, an alias also specifies connection parameters for the server.

ancestor
An object from which another object is derived. An ancestor class can be a parent or a grandparent. See default ancestor.

application
An application is the executable file and all related files that a program needs to function which serve a common purpose or purposes, as distinguished from the design and source code of the project. Often used synonymously with 'program'. Compare with program and project.

array
A group of data elements identical in type that are arranged in a single data structure and are randomly accessible through an index.

ASCII

An acronym for "American Standard Code for Information Interchange" and used to describe the byte values assigned to specific characters. Examples: The capital letter A has an ASCII value of 65. The ASCII code for a space is 32.

In Pascal, you can reference a character by its ASCII code prefixed with a number sign (#). Example: To put the symbol for American cents into a character C, for example, you could code "c := #155;".

base type
The type referred to in a pointer declaration, an array declaration, or the enumeration type used in a set declaration. A type declaration builds a new type by combining or referencing one or more other base types, which could themselves be arbitrarily complex.

batch operation
Operations that you perform with the TBatchMove component on groups of records, or on datasets, to add, delete, or copy groups of records in a single operation.

BDE
Borland Database Engine; also referred to in some documentation as IDAPI. C++Builder uses this database engine to access and deliver data. BDE maintains information about your PC's environment in the BDE configuration file (usually called IDAPI.CFG). Use the BDE Configuration Utility to change the settings in this configuration file.

BDE Configuration Utility

A program that enables you to change the settings in the BDE configuration file, usually called IDAPI.CFG. The executable file is named BDECFG.EXE.

BLOB
Binary large object. Many database tables use specific field types to contain BLOB data. C++Builder lets you access BLOB data that exists as plain text with the TDBMemo component, and BLOB data that exists as a graphic with the TDBImage component.

block
The associated declaration and statement parts of a program or subprogram. Examples: In the var block of the routine declare an integer variable. Follow the **then** of your **if..then** statement with a **begin** to start a block of code that will be executed only if the condition is met.

Boolean
A data type that can have a value of either True or False. Data size = byte.

Borland Database Engine
See BDE.

breakpoints

A place in your code where you want the program to pause or perform an action so that you can examine the current values of program variables and data structures, or the call stack.

Breakpoint behavior falls into two categories:

- Unconditional (or simple). The breakpoint is activated whenever the debugger reaches the line in your source or the machine instruction where you set the breakpoint.
- Conditional. The breakpoint is activated only when it satisfies the conditions that you specify.

byte
A data type capable of holding a value from 0 to 255. A sequence of 8 bits.

callback routines
Routines in your application that are called by Windows and not by your application. For example, EnumFonts is a Windows routine that will call the given callback function for every font installed in the system.

call stack
The list of calls that were made to reach the present location, and which consequently show the path by which the program must return. Available during debugging.

canvas
The graphical drawing surface of an object. The canvas has a brush, a pen, a font, and an array of pixels. The canvas encapsulates the Windows device context.

case variant

1. The element of a case statement that is examined to determine what code will be executed. In a case statement beginning "Case I of", I is the case variant.
2. In record type definitions, case variants allow instances of that record to treat the same area of memory as different fields.

char
A Pascal type that represents a single character.

child

1. A child class is any class that is descended from another. For example, in "type B = class(A)", B is a child of A. Compare with grandchild.
2. The child of a window appears inside that window and cannot draw outside of its bounds. This is called a child or child window.

class
A list of features representing data and associated code assembled into single entity. A class includes not only features listed in its definition but also features inherited from ancestors. The term "class" is interchangeable with the term "object type."

class method
Class methods provide behavior for a class that is global in nature, or otherwise does not require instance data. A class method is called by using the class name followed by the method (TClass.SomeMethod) and can be called with an instance or without. As such, a class method cannot rely on any properties, fields or instance methods in its executions.

client

Generically, any thing that requests the services of something else. In Object Pascal, a client is any code that uses one or more features of an object or unit. In Windows, a client is code that makes use of the Windows API.

In database systems, a workstation connected to an intelligent "back-end" server from which it can request data. The client workstation can process the data locally and write it back to the server.

client area
In Windows, the area of a control which a program (that is, a client of Windows' services) is allowed to draw on. On a window, for example, that would usually exclude the frame and title bar.

column

The vertical component of a table, sometimes called a field. A column contains one value for each row in a table. See also row.

compile

The act of translating a block of source code into machine instructions. (As opposed to "interpret" which is the line-by-line translation of source code to machine instructions.)

Also see linking.

compiler directive
An instruction to the compiler that is embedded within the program; for example, {$R+} turns on range checking.

compile time
The period of time when the compiler is actively compiling source code.

compile-time error
An error detected by the compiler during compilation, such as a syntax error or unknown identifier.

complete evaluation

Every operand in a Boolean expression built from the **and** and **or** operators evaluates, even if the expression result can be determined before the entire expression is evaluated. This is useful when operands are routines that can alter the meaning of a program. Opposite of short-circuit Boolean evaluation.

component

1. The elements of a C++Builder application, iconized on the Component palette. Components, including forms, are objects you can manipulate. Forms are components that can contain other components (forms are not iconized on the Component palette).
2. In C++Builder, any class descended from TComponent is, itself, a component. In the broader sense, a component is any class that can be interacted with directly through the C++Builder Form Designer. A component should be self-contained and provide access to its features through properties.
3. In traditional Pascal, the word "component" is also sometimes used synonymously with feature, as in "The record consists of several components: three string fields and two byte fields."

conditional symbol

Used with conditional compiler directives to specify a condition that is either true or false. You define (set to true) or undefine (set to false) conditional symbols with the $DEFINE and $UNDEF directives.

constant
An identifier with a fixed value in a program. At compile time, all instances of a constant in source code are replaced by the fixed value. Contrast with typed constant.

constant address expression

An expression that takes the address, the offset, or the segment of a global variable, a typed constant, a procedure, or a function.

Constant address expressions cannot reference local variables (stack-based) or dynamic (heap-based) variables, because their addresses cannot be computed at compile time.

const parameter

A const (constant) parameter is one that is passed by reference but that cannot be changed by the procedure. Const is more efficient in performance and memory usage a than a value parameter. See value parameter and variable parameter.

container application
An application that contains an embedded OLE object. (See OLE.)

container component
Any of several component classes that have the inherent ability to contain other components. Examples include TForm, TPanel, TNotebook, and TGroupbox. A container component is the parent of the components it contains.

control
A visual component. Specifically, any descendant of TControl.

data

1. Information stored in a database. Data may be a single item in a field, a record that consists of a series of fields, or a set of records. C++Builder applications can retrieve, add, modify, or delete data in a database.
2. Generally, any information that has intrinsic value regardless of the means used to access it.

data access component
A C++Builder component that enables you to connect to a database and access its data. Data access components are visible on a form only at design time, not at run time.

data-aware
Able to display and update data stored in an underlying table. All C++Builder data control components are data-aware.

data control component

A C++Builder component that enables you to create the interface of a database application. Many data controls are data-aware versions of component classes available on the Standard page of the Component palette.

data type
A fundamental unit of data definition that defines what kind of data can be stored in memory or in data tables, and what operations can be performed on that data.

database
A collection of data in tables.

database server
A system that manages relational databases. For example, SQL Server is a type of database server.

dataset
A collection of data determined by a TTable, TQuery, or TStoredProc component. A dataset defined by TTable includes every row in a table, and a dataset defined by a TQuery contains a selection of rows and columns from a table.

DDE client
In a DDE conversation, the client is the application that requests data. The DDE client is often called the destination.

DDE conversation
A link between a DDE client application and a DDE server application which provides a means for both applications to continuously and automatically send data back and forth.

DDE server
In a DDE conversation, the server is the application that updates the DDE client. The DDE server is often called the source.

default ancestor
The ancestor of any class that does not specify an ancestor: TObject.

default event
For a given component, the event whose event handler is automatically generated or displayed in the unit source code when you double-click the component at design time. For example, the OnClick event is the default event for a Button component.

default new form

The Form Template that is used to create a new form in the IDE at design time when you choose File | New Form. In a new installation, the Blank Form template is used. You can change the specified Form Template in the Environment Options dialog box (Options | Environment | Gallery).

default new project

The Project Template that is used to open a new C++Builder project in the IDE at design time when you choose File | New Project. In a new installation, the Blank Project template is used. You can change the specified Project Template in the Environment Options dialog box (Options | Environment | Gallery).

derive
To create a new class based on an existing class. The new class inherits all of the features of the existing class, which is called its parent or, more generically, an ancestor.

descend

To acquire, in the process of being created, all the characteristics of another class. A class that descends from another is a descendant of the parent class. The process of creating a descendant class is deriving.

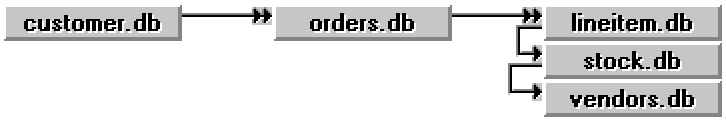
See also ancestor, derive, descendant, inheritance, and parent.

descendant
An object derived from another object. A descendant is type compatible with all of its ancestors.

design time
Period when you can use C++Builder to design your application, using the form, the Object Inspector, Component palette, Code Editor, and so forth; as opposed to run time, when the application you design is actually running.

detail table

In multi-table relationships, the table whose records are subordinate to those of the master table. In a data model, the detail table is the one being pointed to by another table. For example, in the following data model, all of the tables except CUSTOMER.DB are detail tables.



dispatch
The means of resolving calls to object methods. Dispatching can be either static, virtual, or dynamic.

Do not confuse with TObject.Dispatch which dispatches message procedure calls, not virtuals.

drag
To move an object from one location to another by using your mouse.

To drag an object, click it and continue to hold down the left mouse button while you move the mouse pointer to a new location on your screen. When you are satisfied with the new location, release the mouse button.

dynamic
A form of virtual method which is more space efficient (but less speed efficient) than simple virtual.

dynamic data exchange (DDE)

The process of sending data to and receiving data from other applications through a predefined message protocol. You can use this to exchange data with other applications, or you can control other applications through the use of commands and macros.

dynamic-link library (DLL)
An executable module (extension .DLL) that contains code or resources that can be accessed by other DLLs or applications. In the Windows environment, DLLs permit multiple applications to share code and resources.

embedding
The act of placing one thing within another. In Windows, specifically the capability of one application to provide some or all of the services of another application integrated with its own services. For example, a word processor might allow a spreadsheet to be embedded into a document, allowing the user to write text around the spreadsheet and perhaps even change the spreadsheet while still working in the word processor. See OLE Container.

encapsulate

To provide access to one or more features through an interface that protects clients from relying upon or having to know the inner details of the implementation.

enumerated data type
A user-defined ordinal type that consists of an ordered list of identifiers.

end user

A member of an application's intended audience and, by extension, everyone in that audience. Synonymous with user, but emphasizes the fact that the programmer is not the user.

In C++Builder documentation end user refers to a user of an application you develop using C++Builder unless otherwise noted.

exception
An event or condition that, if it occurs, breaks the normal flow of execution. Also, an exception is an object that contains information about what error occurred and where it happened.

exception handler

Code designed to resolve the situation in the run-time environment that raised the exception and/or to restore the environment to a stable state afterwards.

execution point

The execution point indicates the next executable line in your source code or machine instruction that will be executed when you run your program through the integrated debugger. The execution point is indicated by the highlighted line of code in the Code Editor or address location in the CPU window Disassembly Pane.

expressions
Part of a statement that represents a value or can be used to calculate a value.

event

A user action, such as a button click, or a system occurrence such as a preset time interval, recognized by a component.

Each component has a list of specific events to which it can respond. Code that is executed when a particular event occurs is called an event handler.

event handler
A form method attached to an event. The event handler executes when that particular event occurs.

When you use the Object Inspector to attach code to a component event, C++Builder generates a procedure header and a **begin..end** block for you. For example, this is the code C++Builder generates for a button click event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

The code you write inside the **begin..end** block executes whenever Button1 is clicked.

features
A generic term used to refer the fields of a record, the types, constants, variables and routines of a unit, and the fields, properties, and methods of a class.

field

1. One possible element of a structured data type (that is, a record or object), a field is an instance of a specific data type. (Compare with property.)
2. In database terminology, a column of information in a table. A collection of related fields makes up one record. See also record.

file buffer
An area of memory set aside to expedite the transfer of data to and from a file.

file type
A file type refers to the specific data type that a file holds.

filter
Anything used to check or alter data. For example, the file filter in the Save dialog box can be set to show only Pascal files.

filter program
A program that takes output from another program as input and produces an altered, reduced, or verified version of that output.

focus
The component or window that is active in a running application is said to have "focus." Any keyboard input the user enters is directed to that component or window.

formal parameter

An identifier in a procedure or function declaration heading that represents the arguments that will be passed to the subprogram when it is called.

See parameter name for information on a given parameter.

form

To an end user, a form is merely another window. In C++Builder, a form is a window that receives components (placed by the programmer at design time, or created dynamically with code at run time), regardless of the intended run-time functionality of the window.

function
A subroutine that computes and returns a value.

function header

Text that gives the name of a routine followed by a list of formal parameters, followed by the function's return type. In a unit, a routine may have a header entered into the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.

global heap

Memory available to all applications.

Although global memory blocks of any size can be allocated, the global heap is intended only for large memory blocks (256 bytes or more). Each global memory block carries an overhead of at least 20 bytes, and under the Windows standard and 386 enhanced modes, there is a system-wide limit of 8192 global memory blocks, only some of which are available to any given application.

- C++Builder suballocates small allocations from large global memory blocks to reduce the likelihood of hitting the system limit. (See HeapLimit, HeapBlock.)

global variable
A variable used by a routine (or the main body of a program) that was not declared by that routine (or a **var** part of the main body) is considered a global variable by that code. A variable global to one part of a program may be inaccessible to another part of the same program, and hence considered local in that context.

glyph
A bitmap that displays on a BitBtn or SpeedButton component with the component's Glyph property.

grandchild
A class descended from another through one or more intermediate classes. Example: In the following type definition "type E = class(D)", E is the child of D. If D is descended from class C, then E is a grandchild of class C, as well as C's parent, C's parent's parent, and so on, until the root class is reached. C and its ancestors are E's grandparents.

grandparent
A class from which others are descended through one or more intermediate classes. See grandchild.

grid

1. The evenly spaced dots on the form that aid in placing components during design time (not visible at run time). Control through Options | Environment | Preferences.
2. In database terminology, an object on a form that enables you to view and edit all the records in a dataset in a spreadsheet-like format. You create a grid with a TDBGrid component.

handling exceptions
Making a specific response to an exception, which then clears the error condition and destroys the exception object.

header
Text that gives the name of a routine followed by a list of formal parameters, followed in the case of a function by the function's return type. In a unit, a routine may have a header entered into the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.

heap
An area of memory reserved for the dynamic allocation of variables.

heap suballocator

When allocating a memory large block, the heap manager simply allocates a global memory block using the Windows GlobalAlloc routine.

When allocating a small block, the Object Pascal heap manager allocates a larger global memory block and then divides (suballocates) that block into smaller blocks as required. Allocations of small blocks reuse all available suballocation space before the heap manager allocates a new global memory block, which, in turn, is further suballocated.

help context
A number assigned individually to the controls in a program so that when the user activates Help, the Help system can query the focused control and use the help context as a reference to supply information appropriate to what the user is doing.

hint
Pop-up text that appears when the mouse pointer passes over an object in the user interface at run time. Specified in the Hint property of many visual components.

host type
The particular server being used for a process or series of processes, hence "hosting" the activities.

IDAPI
See BDE.

identifier
A programmer-defined name for a specific item (a constant, type, variable, procedure, function, unit, program, or field).

implementation
The second, private part of a unit that defines how the elements in the **interface** part (the public portion) of the unit work.

include file (.INC)
An include file (.INC) is a source-code file that is included in a compilation using the {$I filename} compiler directive.

Include files are seldom part of a C++Builder project, but can optionally be used.

index

1. A position within a list of elements.
2. In database terminology, a sort order for a table associated with a specific field or fields, used to locate records quickly. An index performs the following tasks:

- Determines the location of records.
- Keeps records in sorted order.
- Speeds up search operations.

index type
Specifies the type of elements in an array. Valid index types are all the ordinal types except Longint and subranges of Longint.

inheritance
The assumption of the features of one class by another.

instance
A variable of an object type. More generally, a variable of any type. Actual memory is allocated.

integer
A numeric variable type that is a whole number in the range -32,768 to +32,767.

integrated debugger
The integrated debugger is contained within the Integrated Development Environment. This debugger lets you debug your source code without leaving C++Builder. The functionality of this debugger can be reached through the Run and View menus.

interface
The first, public part of a unit that describes the constants, types, variables, procedures, and functions that are available within it.

key

A field or group of fields in a table, used to order records. A key has three effects:

- The table is prevented from containing duplicate records.
- The records are maintained in sorted order based on the key fields.
- A primary index is created for the table.

label
An identifier that marks the target for a **goto** statement.

language driver
Determines a table's sort order and available character set. The BDE Configuration Utility enables you to specify the default language driver for tables.

late binding
When the address used to call virtual methods or dynamic methods is determined at run time.

linking
The process of turning compiled source code into an executable file. At the linking stage resources are bound into the executable.

literal value
A value that appears in the actual source code, such as the string "Hello, World" or the numeral 1 (as opposed to a calculated value or a declared constant).

local heap

Memory available only to your application or library.

It exists in the upper part of an application's or library's data segment.

The total size of local memory blocks that can be allocated on the local heap is 64K minus the size of the application's stack and static data. For this reason, the local heap is best suited for small memory blocks (256 bytes or less). The default size of the local heap is 8K, but you can change this with the **$M** compiler directive.

local symbol information

Information used by the IDE to debug a routine. Local symbol information must be enabled in the Project Options dialog box (Options **|** Project). Enabled by default in new C++Builder installations.

local variable
A variable declared within a procedure or function.

lock
A device that prevents other users from viewing, changing, or locking a table while one user is working with it.

logic error
Logic errors occur when your program statements are valid, but the actions they perform are not the actions you intended. For example, logic errors occur when variables contain incorrect values, when graphic images don’t look right, or when the output of your program is incorrect.

Longint (type)
A 4-byte integer, able to store integers in the range -2,147,483,648 to +2,147,483,647.

lookup table

A secondary table that enables database systems to use a small code field to enable many records in a primary table to refer to information stored in the lookup table.

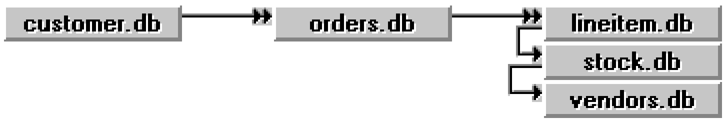
This can be used as a means of ensuring that values entered in a primary table are legitimate values, thus safeguarding data integrity.

loop
A statement or group of statements that repeat until a specific condition is met.

main form
At design time, the first form created in or added to a project. The form designated as the main form can be changed in the Project Options dialog box (Options | Project | Forms). The main form is usually the first displayed at run time, and usually the principal form displayed throughout the execution of the program.

master table

In a multi-table relationship, the primary table of your data model. If you have only one table in your data model, that table is the master table. In a multi-table data model, the master table is the one pointing to other tables. For example, in the following data model, all of the tables except VENDORS.DB are master tables.



method
Procedure or function associated with a particular object.

method identifier
The identifying string or dynamic index of a method.

method pointer
A pointer to a specific method in a specific object.

multiple document interface (MDI) application
An application whose interface consists of a main application window, called the frame window, that can contain multiple child windows, or documents. The child window's document title merges with the parent window's title bar when the child window is maximized.

modal

The run-time state of a form designed as a dialog box in which the user must close the form before continuing with the application. A modal dialog box restricts access to all other areas of the application. See Help for the ShowModal method for more information.

modeless
The run-time state of a form designed as a dialog box in which the user can switch focus away from the dialog box without first closing it. See Help for the Show method for more information.

module
A self-contained routine or group of routines. A unit is an example of a module.

nil

A pointer value referencing nothing. A pointer must be assigned a memory address in order to be meaningfully and safely used.

- Referencing a pointer having a nil value causes a General Protection Fault exception.

nonvisual component

A component that appears at design time as a small picture on the form, but either has no appearance at run time until it is called (like TSaveDialog) or simply has no appearance at all at run time (like TTimer).

nonwindowed control

A nonwindowed control is a control that cannot receive the focus, that cannot be the parent of other controls, and which does not have a window handle.

object files (.OBJ)

An intermediate machine-code file usually produced with an assembler. It is linked with a project or unit using the $L filename compiler directive.

Functions residing in .OBJ files are declared EXTERNAL in Pascal declarations. Object files (.OBJ) are seldom a part of a C++Builder project.

object instance variable
The identifier created internally for an instance of an object.

object type
A class.

OLE

Object Linking and Embedding is a method for sharing complex data among applications. With OLE, data from a server application is stored in a container application. The data is stored in an OLE Object.

OLE container
An application that can contain an OLE object. In C++Builder, an OLE container application has a TOLEContainer component.

OLE object
The data shared by an OLE server and OLE container. An OLE object can be linked or embedded in the container application. The data for linked objects are stored in an external file; embedded objects are stored in the container application.

Examples of OLE objects are documents, spreadsheets, pictures, and sounds.

OLE server
An application that can create and edit an OLE object.

ordinal (type)
Any Object Pascal type consisting of a closed set of ordered elements.

override
Redefine an object method in a descendant object type.

owner
An object responsible for freeing the resources used by other (owned) objects.

parameter
A variable or value that is passed to a function or procedure.

parent

1. The immediate ancestor of a class, as seen in its declaration. Example: In "type B = class(A)", class A is the parent of class B.
2. Parent property: the component that provides the context within which a component is displayed.

pixel
Any of the individual colored dots that make up an image on the screen. Derived from the words "picture element."

pointer
A variable that contains the address of a specific memory location.

power set
The set of all possible subsets of values of the base type including the empty set.

primary index

An index on the key fields of a table. An index performs the following tasks:

- Determines the location of records.
- Keeps records in sorted order.
- Speeds up search operations.

A primary index typically has a requirement of uniqueness--that is, no duplicate keys can exist.

private
The keyword indicating the beginning of a class declaration.

private part
Elements declared in this part of a class declaration can be used exclusively within the module that contains the class declaration. Outside that module they are unknown and inaccessible.

procedure
A subprogram that can be called from various parts of a larger program. Unlike a function, a procedure returns no value.

procedure declaration
The procedure declaration is the first occurrence of the procedure header that appears in a unit or project.

procedure header

Text that gives the name of a routine followed by a list of formal parameters. In a unit, a routine may have a header declared in the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.

program
An executable file. Less formally, a program and all the files it needs to run. Contrast with application.

project
The complete catalogue of files and resources used in building an application or DLL. More specifically, the main source code file of the programming effort, which lists the units that the application or DLL depends on.

project directory
The directory in which the project file resides.

project file
The file that contains the source code for a C++Builder project. This file has a .DPR extension. It lists all the unit files used by the project and contains the code to launch the application.

property

A feature that provides controlled access to methods or fields of an object. A published property may also be stored to a file.

protected
Used in class type definitions to make features visible only to the defining class and its descendants.

protected block
The **try** block of a **try...except** or **try...finally** statement.

public
Used in class type definitions to make features visible to clients of that class.

published
Used to make features in class type definitions streamable. Streamable features are visible at design time.

qualified identifier
An identifier that contains a period, that is, includes a qualifier. A qualified identifier forces a particular feature (of an object, record or unit) to be used regardless of other features of the same name that may also be visible within the current scope.

qualified method identifier
An object-type identifier, followed by a period (.), followed by a method identifier. Like any other identifier, you can prefix a qualified method identifier with a unit identifier and a period.

qualifier
An identifier, followed by a period (.), that precedes a method or other identifier to specify a particular symbol reference.

query
A way to retrieve data from your tables. A query can examine the data in a single table or in several tables.

raise
Raising an exception means constructing an exception object to signal an error or other exception condition. The application then must handle the exception.

real
A number represented by floating-point or scientific notation.

record
1. An instance of a record type.
2. In database terminology, a horizontal row in a table that contains a group of related fields of data.

record type
A structured data type that consists of one or more fields.

recursion
A programming technique in which a subroutine calls itself. Use care to ensure that a recursion eventually exits. Otherwise, an infinite recursion will cause a stack fault.

relational database

A database management model in which data is stored as rows (records) and columns (fields), and in which the data in one table can access the data in other tables by means of a common data field. The database structure can be used to create one-to-many and many-to-one relationships with data elements.

report
Organized summary or detail information that is presented to the end user either as a printed document or an online display.

root class
A class that itself has no ancestors, and from which all other classes are descended. In C++Builder, the root class is TObject.

routine
A procedure or function.

row
The horizontal component of a table, sometimes called a record. A row contains one value for each column in a related group of columns in a table. See also column.

run time
Period when the application you design is running.

run-time error
An error that occurs when the application runs, as opposed to a compile-time error.

run-time library
The standard procedures and functions available to all Object Pascal programs.

run-time only

Routines, properties, events, or components that can be modified, called, or seen only while your application is running (as opposed to design time).

scalar type
Any Object Pascal type consisting of ordered components.

scope
The visibility of an identifier to code within a program or unit.

separator
A blank (space) or a comment. Object Pascal treats a comment as a space.

separator bar

A line inserted between menu items. A dash character (-) entered in the Caption property of a new item in the menu designer creates a separator bar at the current position.

set
A collection of zero or more elements of a certain scalar or subrange base type

short-circuit evaluation
Strict left-to-right evaluation of a Boolean expression where evaluation stops as soon as the result of the entire expression is evident. This model guarantees minimum code execution time, and usually minimum code size.

Shortint
A one byte type capable of holding any whole number value from -128 to +127.

sizing handles
The small black rectangles that appear on the perimeter of a component, form or window when selected. You drag them to resize the object.

source code
The line-by-line statements written by the developer of a computer program using an appropriate editing tool and following the syntax rules for a particular programming language.

splash screen
A form you design to "introduce" your application, and which appears immediately at run time while the application main form and secondary forms are being loaded in memory, or while a database server connection is being established. See also main form.

SpeedMenu
A local menu on an object which you can access by right-clicking with a pointing device.

SQL
Structured Query Language, abbreviated SQL and commonly pronounced "sequel." A relational database language used to define, manipulate, search, and retrieve data in databases.

SQL table
A table on a database server defined by SQL.

stack
An area of memory reserved for storing local variables. Also keeps track of program execution and subroutine calls.

statement
The simplest unit in a program; statements are separated by semicolons.

static
Resolved at compile time, as are calls to procedures and methods.

step over
A debugger command that executes a program one line at a time, stepping over procedures while executing them as a single unit. Contrast with trace into.

string
A sequence of characters that can be treated as a single unit of data.

string list
A flexible collection of strings and (potentially) the objects associated with them.

subrange
Any specified contiguous portion of a scalar type.

switch directive
A compiler directive that turns compiler features on or off depending on the state (+ or -) of the switch. For example, {F+} turns the Force Far calls directive on; {F-} turns it off.

symbol
Any identifier. Symbols include reserved words.

symbol table

A list of the identifiers used by the debugger to track all variables, constants, types, and function names used in a program. Each executable program and DLL has its own symbol table.

- The IDE automatically generates symbolic debug information. To manually choose to turn on debug information for your project, choose Options|Project, click the Compiler tab, and check Debug Information.

table
A structure made up of rows (records) and columns (fields) that contains data.

tag field
A Longint storage for a specific instance of a component to be used as wanted by the programmer.

TDBDataset

A descendant of TDataset that includes the functionality needed to connect to a database, handle passwords, and perform other tasks associated with database connectivity.

You cannot instantiate an object of TDataset directly; you instantiate TTable, TQuery, or another TDataset descendant.

template
A predesigned project or form that serves as a starting point for your application design.

trace into

A debugger command that executes a program one line at a time, tracing into procedures which were compiled with debug information and following the execution of each line. Contrast with step over.

typecasting
The forcing of the compiler to treat an expression of type X as though it were an expression of type Y.

Using AS to typecast object instances causes generation of code to validate the compatibility of the typecast at run time. Normal typecasts are evaluate at compile time and are not validated at run time.

type
A description of how data should be stored and accessed. Contrast with variable--the actual storage of the data.

type compatibility

An instance may be used in place of or assigned to another type it is said to be compatible with.

Integer types are all cross-compatible. A descendant class instance is type-compatible with a variable of an ancestor type. Sibling classes are not type-compatible, nor are ancestors type-compatible with their descendants.

typed constant
A variable that is given a default value upon startup of the application. All global variables occupy a constant space in memory.

type definition
The specification of a non-predefined type. Defines the set of values a variable can have and the operations that can be performed on it.

unit

A independently compileable code module consisting of a public part (the interface part) and a private part (the implementation part).

Every form in C++Builder has an associated unit.

The source code of a unit is stored in a .PAS file. A unit is compiled into a binary symbol file with a .DCU extension. The link process combines .DCU files into a single .EXE or .DLL file.

untyped file
Low-level I/O (input/output) channels that let you directly access any disk file regardless of its internal format.

untyped pointer
A pointer that does not point to any specific type. An untyped pointer cannot be referenced without a typecast. (Also see typecast.)

unqualified identifier

An identifier that contains no periods, that is, an identifier with no qualifier. The semantics of an unqualified identifier depend on the current scope. Example: "Create" is an unqualified identifier that will call any routine called "Create" within the current scope (or cause a compile error if no such routine is visible) but "TForm.Create" will call the specific "Create" method which is a feature of TForm. See qualifier.

use count

An internal variable that Windows uses to determine whether or not a DLL should stay in memory. A DLL stays in memory while its use count is greater than zero.

Windows increments Use Count every time an application loads a DLL and decrements whenever an application frees the DLL.

user-defined
Said of a type that is defined by a programmer and not inherently part of the Pascal language. This includes any type definitions you may code or definitions provided by Borland, or any other source.

value parameter

A procedure or function parameter that is passed by value; that is, the value of a parameter is copied to the local memory used by the routine and therefore, changes made to that parameter are local.

See variable parameter, const parameter.

variable parameter

A subroutine parameter that is passed by reference. Changes made to a variable parameter remain in effect after the subroutine has ended. See value parameter, const parameter.

variable
An identifier that represents an address in memory, the contents of which can change at run time.

virtual
Calls are resolved at run time by name.

visual component
A component that is visible, or can be made visible on a form at run time.

Warning

A message that appears in the Message window that does not stop your code from compiling, but indicates areas you might want to examine for problems. For example, a warning can alert you to code that

- is inefficient
- is not portable
- violates the ANSI standard

watches

A watch expression lets you track the values of program variables or expressions as you step over or trace into your code. Use the Watch List window to view the currently set watches.

As you step through your program, the value of the watch expression will change if your program updates any of the variables contained in the watch expression.

window handle
A number assigned by Windows to a control that must be used to request services for that control from the Windows API.

windowed control
A control that can receive the focus, that can own other controls, and which does have a window handle.

word
In C++Builder, A location in memory occupying 2 adjacent bytes; the storage required for a variable of type integer or word. Also, a predefined data type with a range of 0 to 65535.

wrapper
An object, routine, group of objects, or group of routines designed to encapsulate some functionality for the programmer usually for some perceived benefit. VCL is an object-oriented wrapper for the Windows API.

z-order
The conceptual distance of an object from the surface of the screen. Whether or not a control is covered by other controls depends on its z-order relative to those controls.

<Library Name>is already loaded, probably as a result of an incorrect program termination. Your system may be unstable and you should exit and restart Windows now.

An error occurred while atempting to initialize C++Builder's component library.   One or more DLLs are already in memory, probably as a result of an incorrect program termination in a previous C++Builder or BDE session.

You should exit and then restart Windows.

<IDname> is not a valid identifier

The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there cannot be any spaces in the identifier.

A field or method named <Name> already exists

The name you have specified is already being used by an existing method or field.

For a complete list of all fields and methods defined, check the form declaration at the top of the unit source file.

A component class named <Name> already exists

The component library already contains a component with the same class name you have specified.

Breakpoint is set on line that contains no code or debug information. Run anyway?

A breakpoint is set on a line that does not generate code or in a module which is not part of the project. If you choose to run anyway, invalid breakpoints will be disabled (ignored).

Could not stop due to hard mode

The integrated debugger has detected that Windows is in a modal state and will not allow the debugger to stop your application. Windows enters "hard mode" whenever processing an inter-task SendMessage, when there is no task queue, or when the menu system is active. You will not generally encounter hard mode unless you are debugging DDE or OLE processes within C++Builder.

A standalone debugger such as the Turbo Debugger for Windows can be used to debug applications even when Windows is in hard mode.

Another file named <FileName> is already on the search path

A file with the same name as the one you just specified is already in another directory on the search path.

Cannot find <FileName.PAS> or <FileName.DCU> on the current search path

The .PAS or .DCU file you just specified cannot be found on the search path.

You can modify the search path, copy the file to a directory along the path, or remove the file from the list of installed units.

Cannot find implementation of method <MethodName>

The indicated method is declared in the form's class declaration but cannot be located in the implementation section of the unit. It probably has been deleted, commented out, renamed, or incorrectly modified.

Use UNDO to reverse your changes, or correct the procedure declaration manually. Be sure the declaration in the class is identical to the one in the implementation section. (This is done automatically if you use the Object Inspector to create and rename event handlers.)

For more information about the syntax of procedure declarations, see Handling events.

Debug session in progress. Terminate?

Your application is running and will be terminated if you proceed. When possible, you should cancel this dialog and terminate your application normally (for example, by selecting Close on the System Menu).

Declaration of class <ClassName> is missing or incorrect

C++Builder is unable to locate the form's class declaration in the interface section of the unit.   This is probably because the type declaration containing the class has been deleted, commented out, or incorrectly modified.   This error will occur if C++Builder cannot locate a class declaration equivalent to the following:

```
type
  ...
  TForm1 = class(TForm)
  ...
```

Use UNDO to reverse your edits, or correct the declaration manually.   For more information about class declaration syntax, see Object Types.

Error address not found

The address you have specified cannot be mapped to a source code position.   This error usually occurs for one of the following reasons:

- The address you entered is invalid or is not an address in your application.
- The module containing the specified address was not compiled with debug information.
- The address specified does not correspond to a program statement.

Note that the runtime and visual component libraries are compiled without debug information.

Error creating process: <Process> (<ErrorCode>)

C++Builder was unable to start your application for the reason specified.

For more information about "Insufficient memory to run" errors, see README.TXT.

Field <Field Name> does not have a corresponding component. Remove the declaration?

The first section of your form's class declaration defines a field for which there is no corresponding component on the form.   Note that this section is reserved for use by the form designer.

To declare your own fields and methods, place them in a separate public, private, or porotected section.

This error will also occur if you load the binary form file (.DFM) into the Code Editor and delete or rename one or more components.

Field <Field Name> should be of type <Type1> but is declared as <Type2>. Correct the declaration?

The type of specified field does not match its corresponding component on the form.   This error will occur if you change the field declaration in the Code Editor or load the binary form file (.DFM) into the Code Editor and modify the type of a component.

If you select No and run your application, an error will occur when the form is loaded.

IMPLEMENTATION part is missing or incorrect

In order to keep your form and source code synchronized, C++Builder must be able to find the unit's implementation section.   This reserved word has been deleted, commented out, or misspelled.

Use UNDO to reverse your changes or correct the reserved word manually.   For more information about unit syntax, see the reserved word Unit.

Incorrect field declaration in class <ClassName>

In order to keep your form and source code synchronized, C++Builder must be able to find and maintain the declaration of each field in the first section of the form's class definition.   Though the compiler allows more complex syntax, the form designer will report an error unless each field that is declared in this section is equivalent to the following:

```
type
  ...
  TForm1 = class(TForm)
  Field1:FieldType;
  Field2:FieldType;
  ...
```

This error has occurred because one or more declarations in this section have been deleted, commented out, or incorrectly modified.   Use UNDO to reverse your changes or correct the declaration manually.

Note that this first section of the form's class declaration is reserved for use by the form designer.   To declare your own fields and methods, place them in a separate public, private, or protected section.

Incorrect method declaration in class <ClassName>

In order to keep your form and source code synchronized, C++Builder must be able to find and maintain the declaration of each method in the first section of the form's class definition. The form designer will report an error unless the field and method declarations in this section are equivalent to the following:

```
type
  ...
  TForm1 = class(TForm)
  Field1:FieldType;
  Field2:FieldType;
  ...
  <Method1 Declaration>;
  <Method2 Declaration>;
  ...
...
```

This error has occurred because one or more method declarations in this section have been deleted, commented out, or incorrectly modified. Use UNDO to reserve your changes or correct the declaration manually.

Note that this first section of the form's class declaration is reserved for use by the form designer. To declare your own fields and methods, place them in a separate public, private, or protected section.

Insufficient memory to run

C++Builder was unable to run your application due to insufficient memory or Windows resources. Close other Windows applications and try again.

This error sometimes occurs because of insufficient low (conventional) memory. For further information, see README.TXT.

Invalid event profile <Name>

The VBX control you are installing is invalid.

Module header is missing or incorrect

The module header has been deleted, commented out, or otherwise incorrectly modified.   Use UNDO to reverse your changes, or correct the declaration manually.

In order to keep your form and source code synchronized, C++Builder must be able to find a valid module header at the beginning of the source file.   A valid module header consists of the reserved word unit, program or library, followed by an identifier (for example, Unit1, Project1), followed by a semi-colon. The file name must match the identifier.

For example, C++Builder will look for a unit named Unit1 in UNIT1.PAS, a project named Project1 in PROJECT1.DPR, and a library (.DLL) named MyDLL in MYDLL.DPR.

Note that module identifiers cannot exceed eight characters in length.

No code was generated for the current line

You are attempting to run to the cursor position, but you have specified a line that did not generate code, or is in a module which is not part of the project.

Specify another line and try again.

Note that the smart linker will remove procedures that are declared but not called by the program (unless they are virtual method of an object that is linked in).

Property and method <MethodName> are not compatible

You are assigning a method to an event property even though they have incompatible parameter lists. Parameter lists are incompatible if the number of types of parameters are not identical. For a list of compatible methods in this form, see the dropdown list on the Object Inspector Events page.

Source has been modified. Rebuild?

You have made changes to one or more source or form modules while your application is running. When possible, you should terminate your application normally (select No, switch to your running application, and select Close on the System Menu), and then run or compile again.

If you select Yes, your application will be terminated and then recompiled.

Symbol <BrowseSymbol> not found.

The browser cannot find the specified symbol. This error will occur if you enter an invalid symbol name or if debug information is not available for the module that contains the specified symbol.

The <Method Name> method referenced by <Form Name> does not exist. Remove the reference?

The indicated method is no longer present in the class declaration of the form. This error occurs when you manually delete or rename a method in the form's class declaration that is assigned to an event property.

If you select No and run this application, an error will occur when the form is loaded.

The <Method Name> method referenced by <Form Name>.<Event Name> has an incompatible parameter list. Remove the reference?

A form has been loaded that contains an event property mapped to a method with an incompatible parameter list.   Parameter lists are incompatible if the number or types of parameters are not identical.

For a list of methods declared in this form which are compatible for this event property, use the dropdown list on the Object Inspector's Events page.

This error occurs when hyou manually modify a method declaration that is referenced by an event property.

Note that it is unsafe to run this program without removing the reference or correcting the error.

The project already contains a form or module named <Name>

Every module name (program or library, form and unit) in a project must be unique.

USES clause is missing or incorrect

In order to keep your forms and source code synchronized, C++Builder must be able to find and maintain the USES clause of each module.

In a unit, a valid USES clause must be present immediately following the interface reserved word.   In a program or library, a valid USES clause must be present immediately following the program or library header.

This error occurs because the USES clause has been deleted, commented out, or incorrectly modified. Use UNDO to reverse your changes or correct thte declaration manually.   For more information about the USES clause syntax, see the reserved word USES.

File menu

Use the File menu to open, save, close, and print new or existing projects and files.

The commands on the File menu are:

New	Opens the New Items dialog box which contains items that can be created from the Object Repository.
New Application	Creates a new project containing an empty form.
New Form	Creates and adds a blank form to the current project.
New Data Module	Creates and adds a new, blank data module form to the project.
New Unit	Creates and adds a new file to the project.
Open	Use the Open dialog box to load an existing project, form, unit, or text file into the Code editor.
Open Project	Use the Open Project dialog box to load an existing project (.MAK file).
Reopen	Lists the most recently closed projects and modules for you to select and open.
Save	Saves the current file using its current name.
Save As	Saves the current file using a new name.
Save Project As	Saves the current project using a new name.
Save All	Saves all open files, both current project and modules.
Close	Closes the current project and all associated units and forms.
Close All	Closes all open files.
Include Unit Hdr	Adds a file with an **#include** command to the active module.
Print	Sends the active file to the printer.
Exit	Closes the open project and exits C++Builder.

New Items dialog box (File|New)

See Also

Use the New Items dialog box to select a form, or project template, or wizard that you can use as a starting point for your application. The New Items dialog box provides a view into the Object Repository. The Object Repository contains forms, projects, and wizards. You can use the objects directly, copy them into your projects, or inherit items from existing objects.

The New Items dialog box pages

When you first install C++Builder, there are default pages in the New Items dialog box. They are described in the following paragraphs.

The New page

The new page contains pre-built items that you can include in your project. These items are as follows:

Application	Creates a new project containing a form and a unit pair that consists of a .cpp and .h file.
Automation Object	Creates a unit file which contains an Automation Object template.
Component	Creates a new component unit using the Component Wizard.
Console App	Creates a new console application project.
Data Module	Creates a new Data Module unit.
DLL	Creates a new DLL project.
Form	Creates and adds a blank default form to the current project.
Text	Creates a new ASCII text file.
Thread Object	Creates a new Thread Object unit.
Unit	Creates and adds a new unit to the current project.

The Project page

If a project is open, the second page in the New Items dialog box is the current project page. The current project title is displayed on the tab. The current project page contains all the forms of the project. You can create an inherited form from any existing project forms.

User-defined pages

The remaining pages, if any, are user-defined pages containing Forms, Projects, Data Modules, or Wizards from the Object Repository. By default, the New Items dialog box contains the following pages:

- Forms
- Dialogs
- Data Modules
- Projects

Usage options

Three options on the New Items dialog box let you specify how to use a Repository Object in your project:

- Copy the item
- Inherit from the item
- Use the item directly

To learn about these options, see Object Repository usage options.

File | New Application

Choose File|New Application to create a new C++Builder project. A new blank project is displayed unless you modified C++Builder to use a custom template as the default.

You can have only one project open at any time. If a project is open when you choose File|New Project, C++Builder prompts you to save any changes made to the current project.

A new project consists of

- a new project file (PROJECT1.MAK)
- a new form file (FORM1.DFM), and its associated header (UNIT1.H) and cpp file (UNIT1.CPP)
- a project resource file (PROJECT1.RES)
- a project make file (PROJECT1.MAK)

You can change the names of the project, header, and cpp files when you save them.

File | New Form

Choose File|New Form to create a blank form and a new unit cpp file and add them to the project.

In addition to the standard blank form, you can specify a custom form as the default form to be added to new projects.

When you create a new form, C++Builder automatically adds the new form and an associated unit file to the list of files included in the open project. If no project is open, a blank form is created.

If you selected a blank form from the New Items dialog box, or you did not specify a default form, the new form is titled FormXX and the new unit is UnitXX.CPP. (XX represents the form/unit number. For example, the first form is Form1, the second Form2, and so on.)

You can change the name of the form by editing the Name property using the Object Inspector.

You can change the unit name by saving the file with File|Save As, or by saving the entire project using File|Save Project As.

Changes made to any form or unit name are reflected throughout the source code anywhere that name appears within that unit.

File | New Data Module

Choose File|New Data Module to add a new data module to your project.

At design time, a data module looks like a standard C++Builder form with a white background and no alignment grid. As with forms, you can place nonvisual components on a module from the Component palette, and you can resize a data module to accommodate the components you add to it.

File | New Unit

Choose File|New Unit to add a new unit header file to your project.

The new unit file displays in the Code editor with its own tab and it becomes the active page.

Select Directory dialog box

Use the Select Directory dialog box to choose a working directory for your new project.

To open the Select Directory dialog box,

- Select a non-blank project template from the New Items dialog box.

Directory Name

Displays the current directory. If you enter a directory that does not exist, C++Builder will create it.

Directories

Lists the current directory.

Files: (*.*)

List all the files in the current directory. You cannot select any of these files. C++Builder displays this file list so you know the contents of the current directory.

Drives

Lists all the available drives. You can select one of the available drives.

File | Open

Choose File|Open to display the Open dialog box.

Open dialog box

Use the Open dialog box to load an existing project, form, unit, or text file into the Code editor.

Simply opening a file does not automatically add it to your current project. To add a file to a project, choose Project|Add to Project.

You can open multiple forms, units, or text files but you can have only one project open at any time. If a project is already open when you try to open a project, C++Builder prompts you to save any changes made to the current project before opening the new one.

Look In

Lists the current directory. Use the drop down list to select a different drive or directory.

Files

Displays the files in the current directory that match the wildcards in File Name or the file type in Files Of Type. You can display a list of files (default) or you can show details for each file.

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files of Type

Choose the type of file you want to open; the default file type is unit file (.CPP). All files in the current directory of the selected type appear in the Files list box.

Directories

(Under Windows 95, this looks like a file folder) Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

File | Open Project

Choose File|Open Project to open an existing project.

If there is a project currently open, you are prompted to save your changes and the currently open project is closed.

File | Reopen

Choose File|Reopen to reopen a recently closed project or module.

When you close a project or a module, it is added to the Reopen list.

To reopen a project or module,

1 Choose Reopen from the File menu.

2 Click the project or module that you want to reopen.

Note: Only projects or modules that have been closed with the File|Close command appear in the Reopen list. Saved Items will not appear in the list.

File | Save

Choose File|Save to store changes made to all files included in the open project using the current name for each file.

If you try to save a project that has an unsaved project file or unit file, C++Builder opens the Save As dialog box, where you enter the new file name.

Note: Open files that are not included in the project file will not be saved. To save these files, select each file in the Code editor and choose File|Save.

File | Save As

Choose File|Save As to save the active file with a different name or in a different location.

Save project as dialog box

Use the Save As dialog box to change the project file name or to save the project in a new location. If the file name already exists, C++Builder asks if you want to replace the existing file.

File Name

Enter a name for the file you are saving.

Files

Displays the files in the current directory that match the file type in the Save File as Type combo box.

Save File As Type

Choose a file extension; the default is .CPP. All files in the current directory of the selected type appear in the Files list box. Note that saving a project file with a different extension does not change the format of the file.

Directories

Select the directory where you want to store the file. In the current directory, files that match the file type in the Save Files As Type combo box appear in the Files list box.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

File | Save Project As

Choose File|Save Project As to save the .MAK file to a different name or location. Besides copying and/or renaming the .MAK file, this command saves the project source file to the same name as the .MAK with a .CPP extension. If you have modified forms or units that are used by other projects, and you do not want the current modifications reflected in those other projects, you should first use File|Save As to copy/rename each unit file before choosing this command to save the project.

Save Project As dialog box

Use the Save Project As dialog box to change the project file name or to save the project in a new location. If the file name already exists, C++Builder asks if you want to replace the existing file.

File name
Enter a name for the project file you are saving.

Files
Displays the files in the current directory that match the file type in the Save File as Type combo box.

Save File As Type

Choose a file extension; the default is .MAK. All files in the current directory of the selected type appear in the Files list box. Note that saving a project file with a different extension does not change the format of the file.

Directories

Select the directory where you want to store the file. In the current directory, files that match the file type in the Save Files As Type combo box appear in the Files list box.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

File | Save All

Choose File|Save All to save all open files, including the current project and modules.

To save all files,

1 From the File menu, choose Save All.

2 The Save All dialog box appears with a default name for the item to be saved.

3 Type in a new file name if you do not want to use the default name.

4 Click Save.
The Save As dialog appears again with a default name for the next item to be saved.

5 Repeat steps 3-4 until all modules are saved.

File | Close

Choose File|Close to close the active window.

Closing a form also closes the associated unit file. Before closing the file, C++Builder prompts you to save any changes. If you have not previously saved the project, or any file, C++Builder opens the Save As dialog box, where you can enter the new file name.

If you close the project file in the Code editor, you will close the entire project. You can also close the entire project by choosing File|Close when the Project Manager is the active window.

File | Close All

Choose File|Close all to close all open files.

To close all open files,

- From the File menu, choose Close All.

  The project file and all modules are closed.

File | Include Unit Hdr

Choose File**|**Include Unit Hdr to add an **#include** command for the unit file associated with the active cpp file.

Use this command after adding a form to a project to add the related unit header file for that form.

Include Unit dialog box

Choose File|Include File Header to open this dialog box.

You use this dialog box to insert an **#include** command for a unit into the current unit.

To add a unit,

1 From the File menu, select Include Unit Hdr.

  The Include Unit dialog box appears.

2 In the Include Unit list, click the unit name you want to add.

3 Click OK to include the unit in the current unit.

Include Unit list

This list displays a list of all units in the project that are not being included by the current unit. You can only include unit headers when they are part of the current project. If all units are already included, a message box displays instead.

Component wizard

Choose Component|New or File|New| and Component from the Object Repository to open the Component wizard dialog box.

Component wizard dialog box

Use this dialog box to create the basic unit for a new component.

Class Name

Enter the name of the new class you are creating. A general rule is that all visual component classes are prefaced with a T. For example, the name of your new button component could be TMYBUTTON.

Ancestor Class

Use the drop-down list to select a base class, or enter the name of a base class for your new component. Unless you override them in the component declaration, your new component will inherit all the properties, methods, and events from its ancestor class.

Palette page

Use the drop-down list to select a page, or enter the name of the page on which you want your new component to appear, when you add it to the library.

File | Print

Choose File|Print to print the active page in the Code editor or the active form. When you choose File|Print, C++Builder displays one of two dialog boxes depending on whether the Code editor or the form is the active window.

- When the Code editor is active, C++Builder displays the Print Selection dialog box.
- When the form is active, C++Builder displays the Print Form dialog box.

Print Form dialog box

Use this dialog box to specify any scaling options when printing a form. The scaling options depend on the size of the printer paper. You can change the size of the paper using the Paper Size option in the Printer Setup dialog box.

To display this dialog box, select File|Print when a form is active.

There are three available scaling options:

- Proportional - Scales the form using value of the PixelsPerInch property. Depending on the value of PixelsPerInch, your form may print on more than one page.
- Print To Fit Page - Scales the form so that it will fit onto one page.
- No Scaling - Prints the form using its current onscreen size. If you choose this option, your form might print on more than one page.

Setup

Click the Setup button to display the Printer Setup dialog box.

Print Selection dialog box

Use this dialog box to print the active file from the Code editor.

File To Print
Lists the file that you are going to print. The file listed is the active page in the Code editor when you chose File|Print.

Print Selected Block
Sends only the selected block of text to the printer. This option is available only when you have text selected in the file.

If this option is not checked, the entire file will print.

Header/Page Number
Includes the name of the file, current date, and page number at the top of each page.

Line Numbers
Places line numbers in the left margin.

Syntax Print
Uses bold, italic, and underline characters to indicate elements with syntax highlighting.

Use Color
Prints colors that match colors onscreen (requires a color printer).

Wrap Lines
Uses multiple lines to print characters beyond the page width. If not selected, code lines are truncated and characters beyond the page width do not print.

Left Margin
Specifies the number of character spaces used as a margin between the left edge of the page and the beginning of each line.

Setup
Click the Setup to display the Printer Setup dialog box.

Printer Setup

Changes printer options and selects a printer from a list. To display this dialog box, click Setup from the Print Selection or Print From dialog box.

For more information about setting printer options, see your Windows documentation.

File | Exit

Choose File|Exit to close the open project and then close C++Builder.

If you exit C++Builder before saving your changes, C++Builder asks you if you want to save them.

Edit menu

Use commands from the Edit menu to manipulate text and components at design time.

The commands on the Edit menu are:

Undo/Undelete	Undoes your last action or last deletion
Redo	Reverses an undo
Cut	Removes a selected item and places it on the Clipboard
Copy	Places a copy of the selected item on the Clipboard, leaving the original in place
Paste	Copies the contents of the Clipboard into the Code editor window or form
Delete	Removes the selected item
Select All	Selects all the components on the form
Align to Grid	Aligns the selected components to the closest grid point
Bring to Front	Moves the selected component to the front
Send to Back	Moves the selected component to the back
Align	Aligns components
Size	Resizes components
Scale	Resizes all the components on the form
Tab Order	Modifies the tab order of the components on the active form
Creation Order	Modifies the order in which nonvisual components are created
Lock Controls	Secures all components on the form in their current position
Object	Edits or converts an OLE object which you have inserted onto the form

Edit | Undo/Undelete

See Also

Choose Edit|Undo in the Code editor to undo your most recent keystrokes or mouse actions. Choose Edit|Undelete when working with a form to replace an item you just deleted.

Using Undo in the Code editor

Undo can reinsert any characters you delete, delete any characters you insert, replace any characters you overwrite, or move your cursor back to its prior position.

You can undo multiple successive actions by choosing Undo repeatedly. This undoes your changes by "stepping back" through your actions and reverting them to their previous state. You can specify an undo limit on the Editor Options page of the Project|Environment dialog box.

If you undo a block operation, your file appears as it was before you executed the block operation.

The Undo command does not change an option setting that affects more than one window.

Check Group Undo on the Editor Options page of the Options|Environment dialog box to undo a group of actions.

Edit | Redo

Choose Edit|Redo to reverse the effects of your most recent undo.

Redo has an effect only immediately after an Undo command.

Edit | Cut

Choose Edit|Cut to remove the following items from their current position and place them on the Clipboard:

- Selected text from the Code editor.
- Components from the form.
- Menus from the Menu designer.

Cut replaces the current Clipboard contents with the selected item.

To insert the contents of the Clipboard elsewhere,

- Choose Edit|Paste.

Edit | Copy

Choose Edit|Copy to place an exact copy of the selected text, component, or menu on the Clipboard and leave the original untouched. Copy replaces the current Clipboard contents with the selected items.

To paste the contents on the Clipboard elsewhere,

- Choose Edit|Paste.

Edit | Paste

Choose Edit|Paste to insert the contents of the Clipboard into the active Code editor page, the active form, or active menu in the Menu designer.

Note: You can paste only text into the Code editor window, components onto the form, and menu items into the Menu designer.

When pasting into the Code editor window, the text is inserted at the current cursor position.

When pasting onto the form, nonvisual components are pasted into the upper left corner of the form, and visual components are pasted into the exact position from which they were cut or copied.

When pasting into the Menu designer, menu items are inserted at the cursor position.

You can paste the current contents of the Clipboard as many times as you like until you cut or copy a new item onto the Clipboard.

Edit | Delete

Choose Edit|Delete to remove the selected text or component without placing a copy on the Clipboard.

Even though you cannot paste the deleted text, you can restore it by immediately choosing Edit|Undo or Edit|Undelete.

Delete is useful if you want to remove an item but you do not want to overwrite the contents of the Clipboard.

Edit | Select All

Choose Edit|Select All to select every component on the active form. When you select all the components, only those properties which the components have in common will appear in the Object Inspector.

Edit | Align To Grid

Choose Edit|Align To Grid to align the selected components to the closest grid point.

You can specify the grid size on the Preferences page of the Options|Environment dialog box.

Edit | Bring To Front

Choose Edit|Bring To Front to move a selected component in front of all other components on the form. This is called changing the component's z-order.

Note: The Bring To Front and Send To Back commands do not work if you are combining windowed and non-windowed components. For example, you cannot change the z-order of a label in relation to a button.

Edit | Send To Back

Choose Edit|Send To Back to move a selected component behind all other components on the form. This is called changing the component's z-order.

Note: The Send To Back and Bring To Front commands do not work if you are combining windowed and non-windowed components. For example, you cannot change the z-order of a label in relation to a button.

Edit | Align

Choose Edit|Align to open the Alignment dialog box.

Alignment dialog box

Use this dialog box to line up selected components in relation to each other or to the form.

- The Horizontal alignment options align components along their right edges, left edges, or midline.
- The Vertical alignment options align components along their top edges, bottom edges, or midline.

The options for horizontal or vertical aligment are:

Option	**Description**
No Change	Does not change the alignment of the component
Left Sides	Lines up the left edges of the selected components (horizontal only)
Centers	Lines up the centers of the selected components
Right Sides	Lines up the right edges of the selected components (horizontal only)
Tops	Lines up the top edges of the selected components (vertical only)
Bottoms	Lines up the bottom edges of the selected components (vertical only)
Space Equal	Lines up the selected components equidistant from each other
Center In Window	Lines up the selected components with the center of the window

Edit | Size

Choose Edit|Size to open the Size dialog box.

Size dialog box

Use this dialog box to resize multiple components to be exactly the same height or width.

- The Width options change the horizontal size of the selected components.
- The Height options align the vertical size of the selected components.

The options for horizontal or vertical sizing are:

Option	**Description**
No Change	Does not change the size of the components.
Shrink To Smallest	Resizes the group of components to the height or width of the smallest selected component.
Grow To Largest	Resizes the group of components to the height or width of the largest selected component.
Width	Sets a custom width for the selected components.
Height	Sets a custom height for the selected components.

Edit | Scale

Choose Edit|Scale to open the Scale dialog box.

Scale dialog box

Use this dialog box to proportionally resize the form and all the components on that form.

Dialog box options

Scaling Factor, In Percent

Enter a percentage to which you want to resize the form.

Percentages over 100 grow the form.

Percentages under 100 shrink the form.

Edit | Tab Order

See Also

Choose Edit|Tab Order to open the Edit Tab Order dialog box.

Edit Tab Order dialog box

Use this dialog box to modify the tab order of the components on the form or within the selected component if that component contains other components.

The list box displays those components on the active form that can be a tab stop and their type in their current tab order. The default tab order is determined by the order in which you placed the components on the form.

To change the tab order of a component,

1. Select the component name.
2. Click the up button to move the component up in the tab order, or click the down arrrow to move its down in the tab order.

   You can also drag the selected component to its new position in the tab order.
3. To save your changes, click OK.

Edit | Creation Order

Choose Edit|Creation Order to open the Creation Order dialog box.

Creation Order dialog box

Use this dialog box to specify the order in which your application will create nonvisual components. when you load the form at design time or run time.

The list box displays only those nonvisual components on the active form, their type, and their current creation order. The default creation order is determined by the order in which you placed the nonvisual components on the form.

To change the creation order,

1. Select a component name.
2. Click the up button to move the component creation order up, or click the down arrrow to move its creation order down.

   You can also drag the selected component to its new position in the creation order.
3. To save your changes, click OK.

Edit | Lock Controls

Choose Edit|Lock Controls to secure all components on the active form in their current position. When this command is checked, you cannot move or resize a component. However, you can use the Object Inspector to edit the Height, Left, Top, and Width properties for a selected component.

When this command is checked, components are locked. When components are locked, you can choose Lock Controls to unlock them.

Note: Lock Controls has no effect on the form itself. When you select Lock Controls, you can still resize or move the form.

Edit | Object

Choose Edit|Object to edit or convert an OLE object which you have inserted onto the form. This menu item varies depending on the selected OLE object.

If you choose Convert from the submenu, the Convert dialog box opens.

Convert dialog box

Use this dialog box to specify a different source application for an embedded object.

Dialog box options

Current Type

Displays the type of object that you are converting or activating.

Object Type

Select the type of object to which you want to convert the file.

Convert To

Converts the selected embedded object to the type of information selected in the Object Type box.

Activate As

Opens the embedded object in the type selected in the Object Type box, but returns the object to Current Type after editing.

Display As Icon

Displays the selected embedded object as an icon in a Word document.

Change Icon

Changes the icon that represents an embedded object. This button appears only if you select the Display As Icon check box.

Result

Describes the result of the selected options.

Search menu

See Also

Use the Search menu to locate text, errors, objects, units, variables and symbols in the Code editor.

The commands on the Search menu are:

Find	Searches for specific text
Replace	Searches for specific text and replaces it with new text
Search Again	Repeats search
Incremental Search	Searches for text as you type
Go to Line Number	Moves cursor to specific line number
Go to Address	Goes to the specified address

Search | Find

See Also

Choose Search|Find to display the Find Text dialog box.

Find Text dialog box

Use this dialog box to specify text you want to locate and to set options that affect the search.

Dialog box options

Text to Find

Enter a search string; or, click the down arrow next to the input box to select from a list of previously entered search strings.

Options

Specifies attributes for the search string.

Case sensitive	Differentiates uppercase from lowercase when performing a search.
Whole words only	Searches for words only. (With this option off, the search string might be found within a longer word.)
Regular expressions	Recognizes regular expressions in the search string.

Direction

Specifies which direction you want to search, starting from the current cursor position.

Forward	From the current position to the end of the file. Forward is the default.
Backward	From the current position to the beginning of the file.

Scope

Determines how much of the file is searched.

Global	Searches the entire file in the direction specified by the Direction setting. Global is the default scope.
Selected text	Searches only the selected text in the direction specified by the Direction setting. You can use the mouse or block commands to select a block of text.

Origin

Specifies where the search should start.

From cursor	The search starts at the cursor's current position, and then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the Direction setting. From cursor is the default.
Entire scope	The search covers either the entire block of selected text or the entire file (no matter where the cursor is in the file), depending upon the Scope options.

Search | Replace

Choose Search|Replace to display the Replace Text dialog box.

Replace Text

Use this dialog box to specify text you want to search for and replace with other text (or with nothing).

Most components of the Replace Text dialog box are identical to those in the Find Text dialog box.

Text to find

Enter a search string; or, click the down arrow next to the input box to select from a list of previously entered search strings.

Replace with

Enter the replacement string; or, click the down arrow next to the input box to select from a list of previously entered search strings. To replace the text with nothing, leave this input box blank.

Options

Specifies attributes for the search strings.

Case sensitive	Differentiates uppercase from lowercase when performing a search.
Whole words only	Searches for words only. (With this option off, the search string might be found within longer words.)
Regular expressions	Recognizes specific regular expressions in the search string.
Prompt on replace	Prompts you before replacing each occurrence of the search string. When Prompt on replace is off, the editor automatically replaces the search string.

Direction

Specifies which direction to search the file, starting from the current cursor position.

Forward	From the current position to the end of the file. Forward is the default.
Backward	From the current position to the beginning of the file.

Scope

Determines how much of the file is searched.

Global	The entire file in the direction specified by the Direction setting. Global is the default scope.
Selected text	Only the selected text in the direction specified by the Direction setting. To select a block of text, use the mouse or block commands.

Origin

Specifies where the search should start.

From cursor	The search starts at the cursor's current position, and then proceeds either forward to the end of the scope, or backward to the beginning of the scope depending on the Direction setting. From cursor is the default.
Entire scope	The search covers either the entire block of selected text or the entire file (no matter where the cursor is in the file), depending upon the Scope options.

Replace all

Click Replace all to replace every occurrence of the search string. If you check Prompt on replace, the Confirm dialog box appears on each occurrence of the search string.

Search | Search Again

See Also

Choose Search|Search Again to repeat the last Find or Replace command.

The settings last made in the Find Text or Replace Text dialog box remain in effect when you choose Search Again. For instance, if you have not cleared the Replace Text settings, the Search Again command searches for the string you last specified and replaces it with the text specified in the Replace Text dialog box.

Search | Incremental Search

See Also

Choose Search|Incremental Search to bypass the Find Text dialog box by moving the cursor directly to the next occurrence of text that you type.

When you are performing an incremental search, the Code editor status line reads "Searching For:" and displays each letter you have typed.

For example, if you type "class" the cursor moves to the next occurrence of the word, highlighting each letter as you type it. This behavior continues until the editor loses focus or you press Enter or Escape.

Here are some Incremental Search keystroke options.

Option	**Effect**	
Backspace	Remove the last character from the search string and move to the previous match.	
F3	Repeat search	(Default keybinding)
Ctrl+L	Repeat search	(Classic keybinding)
Ctrl+S	Repeat search	(Epsilon keybinding)
Shift+F5	Repeat search	(Brief keybinding)

Search | Go to Line Number

Choose Search|Go to Line Number to display the Go To Line Number dialog box.

Go to Line Number dialog box

This dialog box prompts you for the line number you want to find. (The current line number and column number are displayed in the Line and Column Indicator on the status bar of the Code editor.)

When this dialog box first appears, the current line number is in the input box.

Enter New Line Number

Specify the line number of the code you want to go to; or click the down arrow next to the input box to select from a list of previously entered line numbers.

Search | Go to Address

Choose Search|Go to Address to display the Goto Address dialog box.

Note that this option is only available while debugging your program.

Goto Address dialog box

Use this dialog box to specify the address of the most recent runtime error during an integrated debugging session or an address that you want to jump to.

Address

Enter the address you want to jump to and click OK. You can enter the address in decimal or hexadecimal, or you can enter a function name that will evaluate to an address (for example, WinMain).

When you click OK, C++Builder displays the address location in the source file or it displays the CPU window with the address highlighted in various panes. You can view low-level information about the program in the CPU window.

Regular expressions

See Also

Regular expressions are characters that customize a search string.

The regular expressions that C++Builder recognizes are:

Character	Description
^	A circumflex at the start of the string matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
.	A period matches any character.
*	An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, bo* matches bot, bo and boo but not b.
+	A plus sign after a string matches any number of occurrences of that string followed by any characters except zero characters. For example, bo+ matches `boo`, and booo, but not bo or be.
[ ]	Characters in brackets match any one character that appears in the brackets, but no others. For example [bot] matches b, o, or t.
[^]	A circumflex at the start of the string in brackets means NOT. Hence, [^bot] matches any characters except b, o, or t.
[-]	A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o.
{ }	Braces group characters or expressions. Groups can be nested, with a maximum number of 10 groups in a single pattern.
\	A backslash before a wildcard character tells the Code editor to treat that character literally, not as a wildcard. For example, **\\^** matches **^** and does not look for the start of a line.

Note: C++Builder also supports Brief regular expressions if you are using Brief keystoke mappings.

Brief regular expressions

See Also

You can use the following symbols in Brief regular expressions:

<	A less than at the start of the string matches the start of a line.
%	A percent sign at the start of the string matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
>	A greater than at the end of the expression matches the end of a line.
?	A question mark matches any single character.
@	An at sign after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, bo@ matches bot, boo, and bo.
+	A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, bo+ matches bot and boo, but not b or bo.
|	A vertical bar matches either expression on either side of the vertical bar. For example, `b`ar|car will match either bar or car.
~	A tilde matches any single character that is **not** a member of a set.
[ ]	Characters in brackets match any one character that appears in the brackets, but no others. For example [bot] matches b, o, or t.
[^]	A circumflex at the start of the string in brackets means NOT. Hence, [^bot] matches any characters except b, o, or t.
[-]	A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o.
{ }	Braces group characters or expressions. Groups can be nested, with the maximum number of 10 groups in a single pattern.
\	A backslash before a wildcard character tells C++Builder to treat that character literally, not as a wildcard. For example, **\\^** matches ^ and does not look for the start of a line.

View menu

Use the commands on the View menu to display or hide different elements of the C++Builder environment and open windows that belong to the integrated debugger.

The commands on the View menu are:

Project Manager	Displays the Project Manager
Project Source	Opens the project source file in the Code editor
Project Makefile	Opens the project .MAK file in the Code editor
Object Inspector	Displays the Object Inspector
Alignment Palette	Displays the Alignment Palette
Component List	Displays the Components dialog box
Window List	Displays a list of open windows
Call Stack	Displays the Call Stack dialog box
Threads	Displays the threads status box
CPU	Displays the CPU window
Breakpoints	Displays the Breakpoints List dialog box
Watches	Displays the Watch List dialog box
Toggle Form/Unit	Toggles the inactive form or unit window active
Units	Displays the View Unit dialog box
Forms	Dislpays the View Form dialog box
New Edit Window	Opens a new page in the Code editor
ToolBar	Hides or shows the ToolBar
Component Palette	Hides or shows the Component Palette

View | Project Manager

See Also!ALink(ProjMan, 2)

Choose View|Project Manager to open the Project Manager. If the Project Manager is already open, it becomes the active window.

Use the Project Manager window to add, delete, save, or copy a file to the current project. The Project Manager also contains a listing of all the units and their associated forms used in the current project.

Note: Units are listed in the Project Manager even if they are not currently open.

You can position the Project Manager anywhere on your desktop.

View | Project Source

See also

Choose View|Project Source to display the project file for the current project and make it the active page in the Code editor. If the project source file is not currently open when you choose this command, C++Builder opens it for you.

An alternative way to perform this command is

- Choose View Project Source from the Project Manager context menu.

View | Project Makefile

See also

Choose View| Project Makefile to display the .MAK file for the current project and make it the active page in the Code editor. If the .MAK file is not currently open when you choose this command, C++Builder opens it for you.

View | Object Inspector

See Also

Choose View|Object Inspector to toggle between the Object Inspector and the last active form or Code editor file. If you have closed the Object Inspector, choose this command to reopen it.

Use the Object Inspector to edit property values and event-handler links.

View | Alignment Palette

See Also

Choose View|Alignment Palette to display the Alignment Palette, which you can use to align components to the form, or to each other.

Note: You can also align components by using the Alignment dialog box.

View | Watches

Choose View|Watches to open the Watch List window.

The Watch List window displays all the currently set watch expressions.

If you keep this window open during your debugging sessions, you can monitor how your program updates the values of important variables as the program runs.

View | Threads

Choose View|Threads to view the Threads status box.

Use this status box to view the status of all the threads currently executing in the application being debugged.

View | CPU

Choose View|CPU to display the CPU window.

View | Breakpoints

Choose View|Breakpoints to open the Breakpoint List window.

The Breakpoint List window lists all currently set breakpoints.

Each breakpoint listing shows the following:

- The file in which the breakpoint is set
- The line number of the breakpoint
- Any condition or pass count associated with the breakpoint

View | Call Stack

Choose View|Call Stack to open the Call Stack window.

The Call Stack window lists the current sequence of routines called by your program. In this listing, the most recently called routine is at the top of the window, with each preceding routine call listed beneath.

Each entry in the Call Stack window displays the procedure name and the values of any parameters passed to it.

View | Component List

See Also

Choose View|Component List to display the Components window.

Components window

Use this window to add components to your forms using the keyboard.

Dialog box options

Search By Name

Enter the name of the component you want to add. This list box performs an incremental search so that the cursor moves to the first component containing the letters you type.

Component

Select the component you want to add. Components are listed in alphabetical order, and their button representation is on the left.

When you select a component, its name appears in the Search edit box.

Add To Form

Click Add To Form to place an instance of the selected component in the center of the form. You can select Add To Form by pressing Enter.

To add the component you selected in the Component list box, do one of the following,

- Press Enter.
- Double-click the component name.
- Click the Add to Form button.

Note: When you add a component to a form using the keyboard, C++Builder uses the default component size and adds the component to the center of the form unless a container component (such as a group box or   panel) is selected.

If a container component is selected, C++Builder places the component you are adding in the center of that container. To add a component into a container you must select the container before selecting Add To Form.

View | Toggle Form/Unit

Choose View|Toggle Form/Unit to display either the inactive form or the unit associated with the active form or unit.

Alternative ways to perform this command are:

- Choose the Toggle Form/Unit button from the ToolBar.
- Choose View Unit or View Form from the Project Manager context menu.

View | Units

Choose View|Units to display the View Unit dialog box.

View Unit dialog box

Use this dialog box to open the project file or any unit in the current project. When you choose a unit, it becomes the active page in the Code editor.

If the unit you want to open is not currently open, C++Builder will open it.

Alternative ways to perform this command are:

- Choose the Select Unit From List button from the ToolBar.
- Choose View Unit from the Project Manager context menu.

View | Forms

Choose View|Forms to display the View Form dialog box.

View Form dialog box

Use this dialog box to quickly open any form in the current project. When you select a form, it becomes the active form, and its associated unit becomes the active module in the Code editor.

If the associated unit is not open when you select a form, C++Builder opens it.

Alternative ways to perform this command are:

- Choose the Select Form From List button from the ToolBar.
- Choose View Form from the Project Manager context menu.

View | Window List

Choose View|Window List to display the Window List dialog box.

Window List dialog box

Use this dialog box to make an inactive C++Builder window active. If you have a lot of windows open, this is the easiest way to locate a specific window. The Windows List dialog box displays all the open C++Builder windows.

To select a window, do one of the following:

- Double-click the window name.
- Select the window name and click OK.

View | New Edit Window

Choose View|New Edit Window to open a new Code editor that contains a copy of the active page from the original Code editor.

Any changes you make to either the original or the copy are reflected in both files.

So that you can distinguish between the windows, the caption in the original window is postfixed with a 1, the first copy with a 2, the second copy with a 3, and so on.

An alternate way to perform this command is:

- Right click the Code editor and choose New Edit Window.

View | ToolBar

Choose View|ToolBar to show or hide the ToolBar.

When this command is checked, the ToolBar is visible.

An alternative way to hide the ToolBar is,

- Choose Hide from the ToolBar context menu.

View | Component Palette

See Also

Choose View|Component Palette to show or hide the Component palette.

When this command is checked, the Component palette is visible.

An alternative way to hide the Component palette is,

- Choose Hide from the Component palette context menu.

Project menu

Use the Project menu to compile or build your application. You need to have a project open.

The commands on the Project menu are:

Add to Project	Enables you to add a file to a project
Remove from Project	Enables you to remove a file from a project
Add To Repository	Enables you to easily add a project to the Object Repository
Compile Unit	Compiles any source code that has changed since the last compile
Makel	Compiles everything in the project, regardless of whether source has changed
Build All	Compiles everything in the project, regardless of whether source has changed
Information	Displays all the build information and build status for your project

Project | Add to Project

See also

Choose this command to open the Add To Project dialog box.

Add To Project dialog box

Use the Add To Project dialog box to add an existing unit and its associated form to the C++Builder project. When you add a unit to a project, C++Builder automatically adds that unit to the **uses** clause of the project file.

File Name

Enter the name of the file you want to load or wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to open; the default file type is Source file (.CPP). All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

Project | Remove From Project

See also

Choose this command to open the Remove From Project dialog box.

Remove From Project dialog box

Use this dialog box to select a module to remove from the current project. When you click OK, C++Builder removes the selected module from the **uses** clause of the current project file but does not delete the file from your disk. Any references to the unit that you added must be removed manually.

If you have modified the file you are removing during this editing session, C++Builder prompts you to save your changes, just in case you want to use the form or unit in another project. If you have not modified the file, C++Builder removes that file from the project without prompting you.

▪ Do not delete unit files by using other file management programs, or directly from the DOS prompt. Doing so will cause errors.

Add To Repository

See also

Choose this command to open the Save As Template dialog box. Use this command to add a project to the Object Repository.

You can add your own projects and forms to those already available in the Object Repository. This is helpful in situations where you want to enforce a standard framework for programming projects throughout an organization.

Project | Compile Unit

See also

Use this command to compile a single unit in your project. To do so:

1. Make the unit you want to compile the active selection in the Code editor or Project Manager window.
2. Choose the Project|Compile Unit.

The hourglass cursor appears (or the Progress dialog box displays if you checked Show Compiler Progress on the Preferences page on the Options|Environment dialog box) and indicates that a compile is in progress. The cursor returns to normal if the compiler finds no errors.

- If an error is found when the Code editor is not open, the Message window displays an error message. If you double-click the message, the Code editor displays the source file positioned at the line where the error ocurred.
- If an error is found when the Code editor is open, then

  the Code editor window comes to the front.

  the unit source file page containing the error comes to the top of the Code editor.

  the line containing the error is highlighted in the Code editor.

  the Code editor message pane displays an error message.
- Context-sensitive help regarding the error message is available by pressing F1.

Project | Make

See also

Choose this command to compile all files in the current project that have changed since it was last built into a new executable file (.EXE). This command is simiilar to Project|Build All, except that Project|Make builds only those files that have changed whereas Project|Build All rebuilds all files whether they have changed or not.

When you choose this command

- The compiler compiles source code for each unit if the source code has changed since the last time the unit was compiled.
- If a unit contains an include (.H) file, and the include file is newer than the unit's .OBJ file, the unit is recompiled.

Once all the units that make up the project have been compiled, C++Builder links them into an executable file (or dynamic link library). This file is given an .EXE (or .DLL) file extension and the same file name as the project source code file. This file now contains all the compiled code and forms found in the individual units, and the program is ready to run.

If you checked Show Compiler Progress from the Preferences page on the Options|Environment dialog box, the Progress dialog box displays information about the compilation progress and results. When your application successfully compiles, choose OK to close the Progress dialog box.

Project | Build All

See also

Choose this command to rebuild all the components of your application regardless of whether they have changed.

This command is similar to Project|Make except that Project|Build All rebuilds everything whereas Project|Make rebuilds only those files that have been changed since the last build. When you choose Build All, precompiled headers and files generated by the incremental linker are deleted and regenerated.

- This command is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It’s also important to Build All when you’ve changed global compiler directives or compiler options, to ensure that all code compiles in the proper state.

Project | Information

Choose this command to open the Information dialog box.

Information dialog box
Use this dialog box to view the program compilation information and compilation status for your project.

Program Information
The Program Information options provide you with information about your project.

Options	What it lists
Source Compiled	Total number of lines compiled
Code Size	Total size of the executable or DLL without debug information
Data Size	Memory needed to store the global variables
Initial Stack Size	Memory needed to store the local variables

Status Information
The Status Information line displays whether or not your last compile succeeded or failed.

Run menu

The Run menu contains commands that provide a way for you to debug your program from within C++Builder. The following commands form the core functionality of the integrated debugger:

Run	Compiles and executes your application
Parameters	Specifies startup parameters for your application
Step Over	Executes a program one line at a time, stepping over procedures while executing them as a single unit
Trace Into	Executes a program one line at a time, tracing into procedures and following the execution of each line
Trace To Next Source Line	Executes the program, stopping at the next executable source line in your code
Run To Cursor	Runs the loaded program up to the location of the cursor in the Code Editor window
Show Execution Point	Positions the cursor at the execution point in an edit window
Program Pause	Temporarily pauses the execution of a running program
Program Reset	Ends the current program run and releases it from memory
Evaluate/Modify	Opens the Evaluate/Modify dialog box, where you can evaluate or change the value of an existing expression
Inspect	Opens the Inspect dialog where you can examine the value of a a variable or expression
Add Watch	Opens the Watch Properties dialog box, where you can create and modify watches
Add Breakpoint	Opens the Edit Breakpoint dialog box, where you can create and modify breakpoints

- The integrated debugger commands become accessible when you generate symbolic debug information for the project you are working on.

Run | Run

Choose Run|Run to compile and execute your application, using any startup parameters you specified in the Parameters dialog box.

If you have modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application.

If the compiler encounters an error, it displays an Error dialog box. When you choose OK to dismiss the dialog box, the Code Editor places the cursor on the line of code containing the error.

The compiler builds .EXE files according to the following rules:

- The project (.MAK) file is always recompiled.
- If the source code of a unit has changed since the last time the unit was compiled, the unit is compiled. When a unit is compiled, C++Builder creates a file with a .DCU extension for that unit.

  If C++Builder cannot locate the source code for a unit, that unit is not recompiled.
- If the interface section of a unit has changed, all the other units that depend on the changed unit are recompiled.
- If a unit links in an .OBJ file (external routines), and the .OBJ file has changed, the unit is recompiled.
- If a unit contains an Include file and the Include file has changed, the unit is recompiled.

Run | Parameters

Choose Run|Parameters to open the Parameters dialog box.

Parameters dialog box

Use this dialog box to pass command-line parameters to your application when you run it, just as if you were running the application from the Program Manager File|Run menu.

Run Parameters

Enter the parameters you want to pass to your application when it starts, or use the drop-down button to choose from a history of previously specified parameters. Parameters take effect only when your application is started. Do not enter the application name in this edit box.

Run | Step Over

Choose Run|Step Over to execute a program one line at a time, stepping over procedures while executing them as a single unit.

The Step Over command executes the program statement highlighted by the execution point and advances the execution point to the next statement.

- If you issue the Step Over command when the execution point is located on a function call, the debugger runs that function at full speed, then positions the execution point on the statement that follows the function call.
- If you issue Step Over when the execution point is positioned on the **end** statement of a routine, the routine returns from its call, and the execution point is placed on the statement following the routine call.

The debugger considers multiple program statements on one line of text as a single line of code; you cannot individually debug multiple statements contained on a single line of text. The debugger also considers a single statement that spans several lines of text as a single line of code.

By default, when you initiate a debugging session with Run|Step Over, C++Builder moves the execution point to the first line of code that contains debugging information (this is normally a location that contains user-written code). To step over start-up code that C++Builder automatically generates, see Debugging Start-up Code.

In addition to stepping over procedures, you can trace into them, following the execution of each line. Use Run|Trace Into to execute each line of a procedure.

An alternative way to perform this command is:

Choose the Step Over button.

Run | Trace Into

Choose Run|Trace Into to execute a program one line at a time, tracing into procedures and following the execution of each line.

The Trace Into command executes the program statement highlighted by the execution point and advances the execution point to the next statement.

- If you issue the Trace Into command when the execution point is located on a function call, the debugger traces into the function, positioning the execution point on the function’s first statement.
- If you issue Step Over when the execution point is positioned on the **end** statement of a routine, the routine returns from its call, and the execution point is placed on the statement following the routine call.
- If the execution point is located on a function call that does not have debugging information, such as a library function, the debugger runs that function at full speed, then positions the execution point on the statement following the function call.

By default, when you initiate a debugging session with Run|Trace Into, C++Builder moves the execution point to the first line of code that contains debugging information (this is normally a location that contains user-written code). To trace into start-up code that C++Builder automatically generates, see Debugging Start-up Code.

In addition to tracing into procedures, you can step over them, executing each procedure as a single unit. Use Run|Step Over to execute procedures as a single unit.

An alternative way to perform this command is:

Choose the Trace Into button.

Run | Trace To Next Source Line

Use this command to stop on the next source line in your application, regardless of the control flow. For example, if you select this command when stopped at a Windows API call that takes a callback function, control will return to the next source line, which in this case is the callback function.

Run | Run To Cursor

Choose Run|Run To Cursor to run the loaded program up to the location of the cursor in the Code Editor window.

When you run to the cursor, your program is executed at full speed, then pauses and places the execution point on the line of code containing the cursor.

You can use Run To Cursor to run your program and pause before the location of a suspected problem. You can then use Run|Step Over or Run|Trace Into to control the execution of individual lines of code.

An alternative way to perform this command is:

Right click the Code editor and choose Run To Cursor.

Run | Show Execution Point

Choose Run|Show Execution Point to position the cursor at the execution point in an edit window. If you closed the edit window containing the execution point, C++Builder opens an edit window displaying the source code at the execution point.

Run | Program Pause

Choose Run|Program Pause to temporarily pause the execution of a running program.

The debugger pauses program execution and positions the execution point on the next line of code to execute. You can examine the state of your program in this location, then continue debugging by running, stepping, or tracing.

If your program assumes control and does not return to the debugger--for example, if it is running in an infinite loop--you can press Ctrl+Alt+Sys Req to stop your program. You might need to press this key combination several times before your program actually stops, because the command will not work if Windows kernel code is executing.

In addition to temporarily pausing a program running in the debugger, you can also stop a program and release it from memory. Use Run|Program Reset to stop a running program and release it from memory.

Run | Program Reset

Choose Run|Program Reset to end the current program run and release it from memory.

Use Program Reset to restart a program from the beginning, such as when you step past the location of a bug, or if variables or data structures become corrupted with unwanted values.

When you reset a program, C++Builder performs the following actions:

- Closes all open program files
- Clears all variable settings

Resetting a program does not delete any breakpoints or watches you have set, which makes it easy to resume a debugging session.

Windows resources

Resetting a program does not necessarily release all Windows resources allocated by your program. In most cases, all resources allocated by VCL routines are released. Windows resources allocated by code which you have written, however, might not be properly released.

If your system becomes unstable, through either multiple hardware or language exceptions or through a loss of system resources as a result of resetting your program, you should exit C++Builder before restarting your debugging session.

Run | Inspect

See also

Choose Inspect to open an Inspector window for the term highlighted (or at the insertion point) in the Code editor. If the insertion point is on a blank space when you choose this command, an empty Inspector window displays where you can enter an item you want to inspect.

This command is only available when the integrated debugger is paused in a program you are debugging, such as when

- you are stepping through code.
- your program is stopped at a breakpoint.
- you first choose Run|Run and then choose Run|Pause.

An alternate way to perform this command is:

Right click the Code editor and choose Inspect.

Run | Evaluate/Modify

This command opens the Evaluate/Modify dialog box, where you can evaluate or change the value of an expression.

An alternate way to perform this command is:

Right click the Code editor and choose Evaluate/Modify.

Run | Add Watch

The Add Watch command opens the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List window to display and manage the current list of watches.

Alternate ways to perform this command are:

- Right-click the Code editor and choose Add Watch at Cursor.
- Right-click the Watch List and choose Add Watch.
- Select a watch in the Watch List, then right-click and choose Edit Watch.

Run | Add Breakpoint

The Add Breakpoint command opens the Edit Breakpoint dialog box, where you can create and modify breakpoints.

Alternate ways to perform this command are:

- Right-click the Breakpoint List and choose Add Breakpoint.
- Select a breakpoint in the Breakpoint List, then right-click and choose Edit Breakpoint.

Component menu

Use the Component menu to build a component, install a new component, rebuild the component library or configure the Component palette.

The options on the Component menu are:

New	Opens the Component Wizard
Install	Installs new components
Open Library	Opens component library file
Rebuild Library	Recompiles the component library
Configure Palette	Opens the Palette dialog box

Component | New

See Also

Choose Component|New to display the Component Wizard. You can also display this Wizard by choosing File|New and selecting Component from the Object Repository.

Component Wizard

Use this Wizard to create the basic unit for a new component.

Class Name

Enter the name of the new class you are creating. A general rule is that all visual component classes are prefaced with a T. For example, the name of your new button component could be TMYBUTTON.

Ancestor type

Use the drop-down list to select a base class or enter the name of a base class for your new component. Unless you override them in the component declaration, your new component will inherit all the properties, methods, and events from its ancestor class.

Palette Page

Use the drop-down list to select a page, or enter the name of the page on which you want your new component to appear when you add it to the library.

Component | Install

See Also

Use Component|Install to display the Install Components dialog box.

Install Components

Use this dialog box to install new components. If a project is open when you choose Component|Install, C++Builder prompts you to save any changes made to the current project when you choose the OK button in this dialog box.

Library Filename

Enter the name of the library file you want to build.

Search Path

Enter the path you want the compiler and linker to search when the component library is rebuilt.

Installed Components

Displays the components that are already installed. When you select a component, its classes are displayed in the Component Classes list box.

Component Classes

Displays the classes that are defined in the unit selected in the Installed Components list box.

Add

Click Add to add a new unit. This button displays the Add Module dialog box, where you enter the new unit or module name.

OCX

Click OCX to add an OLE control to the Component palette. This button displays the Import OLE Control dialog box, where you can select an OCX control to install.

Remove

Click Remove to remove the selected unit and its components from the Component palette.

Revert

Click Revert to discard your changes if you get an error while building the library. The file will revert to its previous state.

Add Module dialog box

Use the Add Module dialog box to add a new unit to the Component Library Unit list.

To open this dialog box,
Choose Component|Install, and click the Add button in the Install Component dialog box.

Module Name
Enter the unit name in the Module Name edit box. If you don't include a file extension, the following extensions will be searched: .CPP, .PAS, .OBJ.

Browse
Click Browse to use the Add Module Browse dialog box to search for the name of the unit you want to install.

Add Module Browse dialog box

Use the Add Module Browse dialog box to search drives and directories to find the file you want to add.

To open this dialog box,
Choose Component|Install, and click the Add button in the Install Component dialog box. Then click Browse to see the Add Module dialog box, and click the Browse button.

Look in
The drop down list box displays the current directory. Use the list box or the buttons next to it to change directories or drives.

Displays the files in the current directory that match the file type in the Files of Type combo box.

File Name
Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files of Type
Choose the type of file you want to open. The default file type is .CPP. All files in the current directory of the selected type appear in the Files list box. (You may add .CPP, .PAS, or .OBJ files.)

Import OLE Control dialog box

Use the Import OLE Control dialog box to specify the OCX control you want to install.

To open this dialog box,
Click the ActiveX button on the Install Components dialog box.

Registered controls
Displays the names of the OLE control libraries that are registered on your system. To select an OLE control library for import, click on the entry in the listbox. When you select a control library, the name of the corresponding library is shown below the listbox.

Register button
Click on this button to open the Register OLE control dialog box where you can add OLE controls that aren't currently registered. An OLE control must be registered before you can import it into C++Builder.

Unregister button
Unregisters the currently selected OLE control by removing it from the system registry. Unregistering an OLE control does not actually delete the control; it merely removes its registration information from the system registry.

Unit File name
The name of the import unit generated by C++Builder. When you import an OLE control library, C++Builder generates an interface unit that contains a class declaration for each OLE control in the library. By default, the import unit is placed in the first directory listed in the library search path.

Browse button
Allows you to browse for a directory in which to place the import unit.

Palette Page
Specifies on which page of the Component palette C++Builder will install the OLE control. The ActiveX page of the Component palette is the default page for OLE controls. If you enter the name of a page that does not exist, C++Builder creates a new page with that name on the Component palette.

Class names
Displays the suggested class names of the OLE controls found in the selected OLE control library. Unless another OLE control library uses the same class names, you should have no reason to change these. If you do make changes, it is strongly suggested that you start each class name with a “T” as is the C++Builder convention.

OK and Cancel buttons
When you press OK, C++Builder generates an import unit by the specified name and returns to the “Install Components” dialog box, adding the newly generated unit to the list of installed units.

Unit File Name dialog box

Use the Unit File Name dialog box to change the directory where C++Builder stores the import unit generated for the OLE control.

To open this dialog box,
Click the Browse button in the Import OLE Control dialog box.

File Name
Enter the name for the generated import unit.

Files
Displays the files in the current directory that match the file type in the Save File As Type combo box.

Save As Type
Choose the type of file you want to save.

Save in
Select the directory where you want the unit stored. Use the buttons to change directories and drives.

Component | Open Library

Choose Component|Open Library to display the Open Library dialog box.

Open Library dialog box
You use this dialog box to install a dynamic component library (.CCL). If a project is open when you choose Component|Open Library, C++Builder prompts you to save any changes made to the current project.

Look in
Select the directory whose contents you want to view. Use the buttons to change directories or drives.

File Name
Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files Of Type
Choose the type of file you want to open. The default file type is Library file (.CCL). All files in the current directory of the selected type appear in the Files list box.

Component | Rebuild Library

Choose Component|Rebuild Library to recompile the existing component library without leaving the C++Builder environment.

In order to recompile the library, C++Builder must close the open project. Therefore, before C++Builder recompiles the library, C++Builder prompts you to save any changes you might have made to the open project.

Component | Configure Palette

Displays the Environment Options dialog box with the Palette tab selected.

Database menu

The Database menu contains commands that enable you to create, modify, track, and view your databases.

Here is a list of the commands on the Database menu:

Explore

SQL Monitor

Database Form wizard

Database | Explore

Choose Database|Explore to open the SQL/Database Explorer. The SQL/Datbase Explorer enables you to maintain a persistent connection to a remote database server during application development and to create and edit BDE aliases and metadata objects.

Database | SQL Monitor

Choose Database|SQL Monitor to open the SQL Monitor. The SQL Monitor enables you to see the actual statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source.

Database | Database Form wizard

Choose Database|Form wizard to use the Database Form wizard to create a form that displays data from a local or remote database.

Tools menu

See Also

Use the Tools menu to view and change environment settings, to modify the list of programs on the Tools menu, and to modify templates and wizards.

These are the commands on the Tools menu:

Configure Tools	Opens the Configure Tools dialog box. Use this dialog box to add, delete, or edit the programs displayed on the Tools menu.
Database Desktop	Runs the Database Desktop applicaton, a database maintenance and data definition tool.

Tools| Configure Tools

See Also

Choose Tools|Configure Tools to display the Configure Tools dialog box that lets you add, delete, or edit programs displayed on the Tools menu.

Configure Tools dialog box

The Configure Tools dialog box provides the following options:

Tools

Lists the programs currently installed on the Tools menu.

Add

Click Add to display the Tool Properties dialog box, where you can specify a menu name, a path, and startup parameters for the program.

Delete

Click Delete to remove the currently selected program from the Tools menu.

Edit

Click Edit to display the Tool Properties dialog box, where you can edit the menu name, the path, or the startup parameters for the currently selected program.

Arrow

Use the arrow buttons to rearrange the programs in the list. The programs appear on the Tools menu in the same order they are listed in the Tool Options dialog box.

To add a program to the tools menu,

- Choose Add. C++Builder displays the Tool Properties dialog box, where you specify information about the application you are adding.

To delete a program from the tools menu,

- Select the program to delete, and choose Delete. C++Builder prompts you to confirm the deletion.

To change a program on the tools menu,

- Select the program to change, and choose Edit. C++Builder displays the Tool Properties dialog box with information for the selected program.

Tool Properties dialog box

See Also

Use the Tool Properties dialog box to enter or edit the properties for a program listed on the Tools menu.

To display the Tool Properties dialog box,

- Click Add or Edit in the Tool Options dialog box.

Dialog box options

Title

Enter a name for the program you are adding. This name will appear on the Tools menu.

You can add an accelerator to the menu command by preceding that letter with an ampersand (&). If you specify a duplicate accelerator, C++Builder displays a red asterisk (*) next to the program names in the Tool Options dialog box.

Program

Enter the location of the program you are adding. You can include the full path to the program. Click the Browse button to search your drives and directories to locate the path and file name for the program.

Working Dir

Specify the working directory for the program. C++Builder specifies a default working directory when you select the program name in the Program Edit Box. You can change the directory path if needed.

Parameters

Enter parameters to pass to the program at startup. For example, you might want to pass a file name when the program launches. You can type the parameters or use the Macros button to supply startup parameters. You can specify multiple parameters and macros.

Browse

Click Browse to select the program name for the Program edit box. When you click Browse, the Select Transfer Item dialog box opens.

Macros

Click Macros to expand the Tool Properties dialog box to display a list of available macros. You can use these macros to supply startup parameters for your application.

To add a macro to the list of parameters,

- Select a macro from the list and click Insert.

Transfer macros

See Also

Use transfer macros to supply startup parameters to a program on the Tools menu.

To display the macros,

- Click the Macros button on the Tool Properties dialog box.

The transfer macros are:

Macro	Description
$COL	Expands to the column number of the cursor in the active Code editor window. For example, if the cursor is in column 50, at startup C++Builder passes "50" to the program.
$ROW	Expands to the row number of the cursor in the active Code editor window. For example, if the cursor is in row 8, at startup C++Builder passes "8" to the program.
$CURTOKEN	Expands to the word at the cursor in the active Code editor window. For example, if the cursor is on the word Token, at startup C++Builder passes "Token" to the program.
$PATH	Expands to the directory portion of a parameter you specify. When you insert the $PATH macro, C++Builder inserts $PATH() and you specify a parameter within the parentheses. For example, if you specify $PATH($EDNAME), at startup C++Builder passes the path for the file in the active Code editor window to the program.
$NAME	Expands to the file name portion of a parameter you specify. When you insert the $NAME macro, C++Builder inserts $NAME() and you specify a parameter within the parentheses. For example, if you specify $NAME($EDNAME), at startup C++Builder passes the file name for the file in the active Code editor window to the program.
$EXT	Expands to the file extension portion of a parameter you specify. When you insert the $EXT macro, C++Builder inserts $EXT() and you specify a parameter within the parentheses. For example, if you specify $EXT($EDNAME), at startup C++Builder passes the file extension for the file in the active Code editor window to the program.
$EDNAME	Expands to the full file name of the active Code editor window. For example, if you are editing the file C:\PROJ1\UNIT1.PAS, at startup C++Builder passes "C:\PROJ1\UNIT1.PAS" to the program.
$EXENAME	Expands to the full file name of the current project executable. For example, if you are working on the project PROJECT1 in C:\PROJ1, at startup C++Builder passes "C:\PROJ1\PROJECT1.EXE" to the program.
$PARAMS	Expands to the command-line parameters specified in the Run Parameters dialog box.
$PROMPT	Prompts you for parameters at startup. When you insert the $PROMPT macro, C++Builder inserts $PROMPT() and you specify a default parameter within the parentheses.
$SAVE	Saves the active file in the Code editor.
$SAVEALL	Saves the current project.
$TDW	Sets up your environment for running Turbo Debugger. For example, this macro saves your project, ensures that your project is compiled with debug info turned on, and recompiles your project if it is not compiled with debug info turned on. Be sure to use

this macro if you add Turbo Debugger to the Tools menu.

Select Transfer Item dialog box

See Also

Use the Select Transfer Item dialog box to search drives and directories for a program to add to the Tools menu.

To locate a transfer item,

- Click the Browse button on the Tool Properties dialog box.

File Name

Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files

Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type

Choose the type of file you want to open. The default file types are .EXE, .COM, and .PIF files. All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. Files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Database Desktop

See Also

The Database Desktop (DBD) is a database maintenance and data definition tool. It enables you to query, restructure, index, modify, and copy database tables, including Paradox, dBASE, and SQL tables. It also enables you to create new Paradox and dBASE tables. You do not have to own Paradox or dBASE to use the DBD with desktop files in these formats.

The DBD can copy data and data dictionary information from one format to another. For example, you can copy a Paradox table to an existing database on a remote SQL server. For a complete description of the DBD, see Database Desktop Contents.

To open the Database Desktop,

- Choose Tools|Database Desktop.

Adding programs to the Tools menu

See Also

The commands on the Tools menu transfer execution to an external application. You can add programs to, delete programs from, or edit programs on the Tools menu.

To add a program to the tools menu,

1. Choose Tools|Tools.

   C++Builder displays the Tool Options dialog box, which lists the programs currently on the Tools menu.
2. Choose Add.

   C++Builder displays the Tool Properties dialog box.
3. Specify a title for the program. The title you specify will be listed on the Tools menu.
4. Specify the program file or choose Browse to select it from a list.
5. Specify the working directory for the program, if necessary.
6. Specify startup parameters for the program, if necessary. You can type the parameters or use the Macros button to supply startup parameters. You can specify multiple parameters and macros.
7. Choose OK.

   C++Builder closes the Tool Properties dialog box. The new program is on the Tools list in the Tool Options dialog box.
8. Choose Close.

   C++Builder closes the Tool Options dialog box. The new program is on the Tools menu.

C++Builder version control interface

See also

After you have created more than one application or source code file with C++Builder, you may want a way to organize and keep track of the programs. You may want to archive old versions of an application before you start to work on the new version. If you work with a team of programmers, you will need a way to coordinate which programmers have access to certain files, so that one programmer does not accidentally lose, change, or overwrite another programmer's code.

A version control system is useful when managing multiple or complex programming projects. Version control is used to archive files, control access to files by locking, and manage multiple versions of your projects. C++Builder includes an interface to Intersolv PVCS Version Manager 5.1 or later.

- For more information, see the PVCS Version Control online help (PVCS.HLP) located in the Borland C++Builder Help directory.

Enabling PVCS support

C++Builder's version control system support should be enabled when you run the setup program.

The Workgroups menu group appears in the C++Builder menu bar. Choose commands from the Workgroups menu to use the PVCS Version Manager. Here is a list of the commands on the Workgroups menu:

- Browse PVCS Project
- Manage Archive Directories
- Add Project to Version Control
- Set Data Directories

Workgroups | Manage Archive Directories

See also

Choose Workgroups|Manage Archive Directories to specify the locations of archive directories.

To use the Manage Archive Directories dialog box,

1. Specify an archive directory in the Directory and Drives lists.
2. To designate the specified directory an archive location, choose Add.
3. To remove an archive location from the Archive directories list, choose Remove.
4. Choose OK to save the archive directory specification.

▪ For more information, see the PVCS Version Control online help (PVCS.HLP) located in the Borland C++Builder Help directory.

Workgroups | Add Project to Version Control

Choose Workgroups|Add Project to Version Control to create a new PVCS project for the current C+
+Builder Project.

Workgroups | Browse PVCS Project

Choose Workgroups| Browse PVCS project to display an explorer type view of the current project.   The project must fist be created using the Add Project to Version Control menu item.

Set Data Directories

Set Data Directories to specify the location of public and private project root directories. PVCS will use these directories to store subdirectories and files which contain information specific to the projects under PVCS version control.

Custom Expert menu item

You have pressed F1 on a menu item for an Expert that was installed externally. For more information regarding this item, select it to open the Expert. Help is available for externally installed Experts if provided by the developer of the Expert.

Custom Version Control menu item

You have pressed F1 on a menu item for a version control system that was installed externally. For more information regarding this item, select it to gain access to the version control system. Help may be available for your version control system depending on the particular installation.

Ð“Options menu

Use the Options menu commands to control the behavior of the C++Builder environment and to set options used to build the current project. The Options menu also gives you access to the Object Repository dialog box, from where you can customize the Object Repository.

The commands on the Compile menu are:

Project

Environment

Repository

Options | Project

The Options|Project command displays the Project Options dialog box. Use this dialog box to set your compiler, linker, and application options. The Project Options dialog box contains the following pages:

Forms

Application

C++

Pascal

Linker

Directories/Conditionals

You can change the page displayed by clicking the tabs at the top of the dialog box.

▪ The options listed in this dialog box are the options that you need to set for most C++Builder applications. If you need to set more advanced compiler and linker options, refer to the section Setting Advanced Options.

Default check box

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Forms (Options | Project)

Use the Forms page of the Project Options dialog box to select the main form for your current project, and to specify which forms will be automatically created when your application begins. You also set the order in which child windows are created as your application starts.

Main Form

Displays the form users will see when they start your application. Use the drop-down list to select which form is the main form for the project.

The main form is always the first form listed in the Auto-Create Forms list box.

Auto-Create Forms

Lists forms that are automatically added to the startup code of the project file. These forms are automatically created and displayed when you first run your application. You can rearrange the create order of forms by dragging and dropping forms to a new location.

To select multiple forms, hold down the Shift key while selecting the form names.

Available Forms

Lists the forms that are used by your application but are not automatically created. To create an instance of a form that is listed in this column, you must call the form's CreateForm method.

Arrow buttons

Use the arrow buttons to move one or more files from one list box to the other.

To move all the files from one list box into the other,

- Click the double arrow buttons (>> or <<).
- Drag and drop the files from one list box into the other.

To move only the selected file or files from one list box into the other,

- Click the single arrow buttons (> or <).
- Drag and drop the file from one list box into the other.

Default

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Application (Options | Project)

Use the Application page of the Project Options dialog box to specify a title, a Help file, and an icon for your application.

Title

Specify a title that will appear under the application's icon when the application is minimized. The character limit is 255 characters.

Help File

Specify the name of the Help (.HLP) file your application automatically calls whenever the user calls Help (usually by pressing F1). The Help file you specify is passed to the WinHelp function call.

If you are unsure of the Help file name, you can click the Browse button to display the Application Help File dialog box.

Icon

Displays the icon (.ICO) file that will represent the application in the Program Manager. The icon you select will also display when the application is minimized.

To select an icon, click Load Icon; C++Builder displays the Application Icon dialog box, from where you can select an icon.

Default check box

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Application Icon dialog box

Use the Application Icon dialog box to select an icon that will represent your application in the Program Manager or when your application is minimized.

To display this dialog box, click Load Icon on the Application page of the Project Options dialog box.

File Name
Enter the name of the file you want to use. You can also use wildcards in the Files list box to filter the files that C++Builder displays.

Files
Displays all files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

Files of Type
Choose the type of file you want to use; the default file type is an icon (.ICO) file. All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the Files of Type combo box appear in the Files list box.

Note: If you are using Windows 95 or Windows NT, use the toolbar to specify the directory and drive.

- You can enter a path relative to your C++Builder root directory using the **$(BCB)** environment macro.

Drives
Select the drive that contains the icon you want to use. The directory structure for the current drive appears in the Directories list box.

Application Help File dialog box

Use the Application Help File dialog box to select the Help (.HLP) file you want to use for your project. The Help file you specify is entered into the Help File edit box on the Application page of the Project Options dialog box.

To display this dialog box, click Browse on the Application page of the Project Options dialog box.

File Name

Enter the name of the file you want to use. You can also enter wildcards to filter the files that are displayed in the Files list box.

Files

Displays all files in the current directory that match the wildcards in the File Name edit box or the file type in the Files of Type combo box.

Files of Type

Choose the type of file you want to use; the default file type is a Help file (.HLP). All files in the current directory of the selected type appear in the Files list box.

Directories

Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the Files of Type combo box appear in the Files list box.

- If you are using Windows 95 or Windows NT, use the toolbar to specify the directory and drive.

Drives

Select the current active drive. The directory structure for the current drive appears in the Directories list box.

C++ (Options | Project)

The C++ page of the Project Options dialog box specifies the option settings for the C++ compiler.

Speed settings

The Speed Setting button enables an aggregate set of compiler and liner options. If you use these speed setting options, you shouldn't need to use the individual option settings on the C++, Pascal, and Linker pages of the Project Options dialog box.

- **Full debug** lets you set the recommended set of options to use while you are developing your application. The Full Debug button sets the following options:
- C++ compiler options
  - Enables Code Optimizations | None
  - Enables Debugging | Debug Information
  - Enables Debugging | Line Number Information
  - Disables Debugging | Automatic Register Variables
  - Enables Debugging | Disable Inline Expansions
  - Enables Compiling | Stack Frames
- Object Pascal compiler options
  - Disables Code Generation | Optimization
  - Enables Code Generation | Stack Frames
  - Enables Debugging | Debug Information

  Enables Debugging | Local Symbols

  Enables Debugging | Symbol Information
- Linker options
  - Enables Linking | Include Debug Information
- **Release** sets the compiler and linker options that you should use when you have finished developing and debugging your application. The Release button sets the following options (this button reverses the settings made by the Full Debug button):
- C++ compiler options
  - Enables Code Optimizations | Speed
  - Disables Debugging | Debug Information
  - Disables Debugging | Line Number Information
  - Enables Debugging | Automatic Register Variables
  - Disables Debugging | Disable Inline Expansions
  - Disables Compiling | Stack Frames
- Object Pascal compiler options
  - Enables Code Generation | Optimization
  - Disables Code Generation | Stack Frames
  - Disables Debugging | Debug Information

  Disables Debugging | Local Symbols

  Disables Debugging | Symbol Information
- Linker options
  - Disables Linking | Include Debug Information

Code Optimizations

The Code Optimization settings enable an aggregate set of compiler options.

- **None** turns off all code optimizations. This is normally the best optimization to use when you are developing and debugging your application-the debugger will not skip around due to code that has been "optimized away" by the compiler.
- **Speed** sets the following compiler switches designed to optimize your code for speed:
- Duplicate expression within functions
- Intrinsic functions
- Induction variables
- **Speed with scheduling** sets an aggregate of compiler switches designed to optimize your code for speed, along with code that is optimized for Pentium processors (do not use this setting if you plan to

use your application on systems with processors less than a Pentium).

Debugging

Use the various debugging options to help you debug your code. Remember to review these options before your final build of the project.

- **Debug information** adds debug information to your compiled .OBJ files.
- **Line number information** adds line numbers to your .OBJ files.
- **Automatic register variables** tells the compiler to automatically assign register variables if possible, even when you do not specify a register variable, by using the **register** type specifier.

  You can usually turn this option on, unless you are interfacing with preexisting assembly code that does not support register variables.
- **Disable inline expansions** causes the compiler to expand C++ inline functions inline. See Out-of-line Inline Functions for more information.
- To have debug information added to your final executable file, you must also check Include Debug Information on the Linker page of the Project Options dialog.

Precompiled headers

This section contains the general-use pre-compiled heading options. Pre-compiled headers can dramatically increase compilation speeds, although they require a considerable amount of disk space.

- **None** does not generate pre-compiled headers.
- **Use pre-compiled headers** causes the compiler to generate and use pre-compiled headers. The default file name is VCL.CSM for C++Builder projects.
- **Cache pre-compiled headers** causes the compiler to cache the pre-compiled headers it generates. This is useful when you are pre-compiling more than one header file.

Compiling

The Compiling options control general C++Builder compiler behavior.

- **Merge duplicate strings** causes the compiler to merge two literal strings when one matches another. This produces smaller programs (at the expense of a slightly longer compile time), but can introduce errors if you modify one string.
- **Stack frames** causes the compiler to generate a standard stack frame (standard function entry and exit code). This is helpful when debugging, since it simplifies the process of stepping through the stack of called subroutines.

  When this option is off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code smaller and faster.

  The Standard Stack Frame option should always be on when you compile a source file for debugging.
- **Show warnings** turns on all general compiler warnings.
- **Show general msgs** shows general compiler and linker messages (these messages are not warnings or error messages).
- The options listed in here are the options that you need to set for most C++Builder applications. If you need to set more advanced compiler and linker options, refer to the section Setting Advanced Options.

Default check box

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Pascal (Options | Project)

Use the Pascal page of the Project Options dialog box to set Object Pascal compiler options.

These options correspond to switch directives that you can set directly in your Object Pascal program code. Selecting an Object Pascal option is equivalent to setting the switch directive to its positive (+) state.

Code generation

The Code Generation options effect the code output by the Object Pascal compiler.

- **Optimization** enables compiler optimizations.
This setting corresponds to the {$O} Object Pascal compiler directive.
- **Aligned record fields** aligns elements in structures to 32-bit boundaries.
Corresponds to {$A}.
- **Stack frames** forces compiler to generate stack frames on all procedures and functions.
Corresponds to {$W}.
- **Pentium-safe FDIV** generates floating-point code that is safe for all Pentium processors.
Corresponds to {$U}.

Runtime errors

These options specify how Object Pascal handles errors generated at run time.

- **Range checking** checks that array and string subscripts are within bounds.
Corresponds to {$R}.
- **Stack checking** checks that space is available for local variables on the stack.
Corresponds to {$S}.
- **I/O checking** checks for I/O errors after every I/O call.
Corresponds to {$I}.
- **Overflow checking** checks overflow for integer operations.
Corresponds to {$Q}.

Syntax options

These options control how the Object Pascal compiler handles language syntax.

- **Strict var-strings** sets up string parameter error checking. (If the Open parameters option is selected, this option is not applicable.)
Corresponds to {$V}.
- **Complete boolean eval** evaluates every piece of an expression in Boolean terms, regardless of whether the result of an operand evaluates as false.
Corresponds to {$B}.
- **Extended syntax** enables you to define a function call as a procedure and to ignore the function result. Also enables Pchar support.
Corresponds to {$X}.
- **Typed @ operator** controls the type of pointer returned by the @ operator.
Corresponds to {$T}.
- **Open parameters** enables open string parameters in procedure and function declarations. Open parameters are generally safer, and more efficient.
Corresponds to {$P}.
- **Huge strings** enables new garbage collected strings. The **string** keyword corresponds to the new AnsiString type when this option is enabled. Otherwise the **string** keyword corresponds to the ShortString type.
Corresponds to {$H}.
- **Assignable typed constants** is used for backward compatibility with Delphi 1.0. When enabled, the compiler allows assignments to typed constants.
Corresponds to {$J}.

Debugging

Set these options to specify what type of debug information you want included in your .OBJ files.

- **Debug information** puts debug information into the unit (.DCU) file.

Corresponds to {$D}.

- **Local symbols** generates local symbol information.

Corresponds to {$L}.

- **Symbol information** generates symbol information.

Corresponds to {$Y}.

Messages

These options specify the level of messages generated by the Object Pascal compiler.

- **Show hints** causes the compiler to generate hint messages.
- **Show warnings** causes the compiler to generate warning messages.

Default check box

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Linker (Options | Project)

Use the Linker page of the Project Options dialog box to specify how your program files are to be linked.

Application target

The Application Target radio buttons specify the type of executable file you want to create from your project. You can generate either an .EXE or a .DLL executable file through the C++Builder environment.

Map file

Select the type of map file you want produced during your project compilation, if any. C++Builder gives the map file an .MAP file extension, and places it in the directory where your project .MAK file is stored.

Map option	Effect
Off	Does not produce map file.
Segments	Linker produces a map file that includes a list of segments, the program start address, and any warning or error messages produced during the link.
Publics	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, and a list of alphabetically sorted public symbols.
Detailed	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, a list of alphabetically sorted public symbols, and an additional detailed segment map. The detailed segment map includes the segment address, length in bytes, segment name, group, and module information.

- For details on the map files produced, see Map File Options.

Linking

The options in the Linking section specify general linker options.

- **Use incremental linker** causes C++Builder to link your projects using the ILINK32, the incremental linker.
- **In-memory .EXE** causes C++Builder to compile and link an executable image of your .EXE in RAM; the .EXE is *not* written to disk. Use this option for faster compile/link cycles on systems with smaller amounts of RAM. Be sure to disable this option when you want to create an .EXE image on disk. (This option is disabled if you are running on Windows 95.)
- **Show warnings** controls whether or not C++Builder displays compiler warning messages. You can control individual warnings through the Advanced Compiler Options.
- **Include debug information** includes debug information in the final executable image. Debug information is used by both the integrated debugger and by Turbo Debugger for Windows (TDW32.EXE).
- **Link debug version of VCL** causes C++Builder to link the VCLD.LIB version of the VCL. Use this option only when you want to step into the Object Pascal code of the Visual Component Library; it will greatly increase the size of your executable files (but it will not effect application performance).

Application type

The Application Types settings specify the type of application you are building with your project.

- **Windows GUI** causes C++Builder to generate a Graphical User Interface (GUI) application that can be run on 32-bit Windows systems.
- **Generate Console Application** causes linker to set a flag in the application’s .EXE file indicating a console mode application.

Stack sizes

Use these edit boxes to specify the minimum and maximum stack size and heap image base for the compiled executable.

Default check box

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Directories/Conditionals (Options | Project)

You use the Directories/Conditionals page of the Project Options dialog box to specify the location of files needed to compile and link your program. In addition, you can specify compiler defines on this page. Click the down arrow next to any edit box to choose from a list of previously entered directories or symbols.

Directories settings

The Directories settings tell C++Builder where to find C++ header and library files. Specify the Include Path for the header files used by your project and the Library Path for your project object files, resource files, and the Delphi unit files.

You can specify multiple directories by separating them with a semicolon. See Search Directories for more information on setting these options in C++Builder.

- You can enter a path relative to your C++Builder root directory using the **$(BCB)** environment macro.

Conditionals

Specify symbolic defines for the C++, Object Pascal, and Resource compilers using this list box. You can include multiple definitions by separating each define with a semicolon (;). Assign values with an equal sign (=). For example:

```
xxx;yyy=1;zzz=NO MYFILE.C
```

For more information on C++ compiler defines, see Compiler define options.

Pascal Unit Aliases

Useful for backwards compatibility. Specify alias names for Object Pascal units that may have changed names or were merged into a single unit. The syntax format is `<oldunit>=<newunit>`. You can separate multiple aliases with semicolons. The default value is `WinTypes=Windows;WinProcs=Windows`.

Default check box

Check Default to save the current project options as the default options. C++Builder will use the default options for each new project you create.

Options | Environment

The Options|Environment command displays the Environment Options dialog box. Use this dialog box to set your editor and configuration preferences, and to customize the way components and pages are arranged on the Component palette. The Environment Options dialog box contains the following pages:

Preferences

Library

Editor Options

Display

Colors

Palette

You can change the page displayed by clicking the tabs at the top of the dialog box.

Preferences (Options | Environment)

See Also

Use the Preferences page of the Environment Options dialog box to specify your C++Builder configuration preferences.

Compiling

- **Show compiler progress** displays a progress report dialog box while your program compiles.
- **Beep on completion** causes C++Builder to beep when your compilation is finished.
- **Cache hdrs on startup** places the compiled header files in memory upon startup. These pre-compiled header files are then available for all C++Builder compiles you make in that session.

Form designer

Set grid preferences that make it easier to design forms.

Grid options	Effect
Display Grid	Displays dots on the form to make the grid visible.
Snap To Grid	Automatically aligns components on the form with the nearest gridline. You cannot place a component "in between" gridlines.
Grid Size X	Sets grid spacing in pixels along the x-axis. Specify a higher number (between 2 and 128) to increase grid spacing.
Grid Size Y	Sets grid spacing in pixels along the y-axis. Specify a higher number (between 2 and 128) to increase grid spacing.
Show Component Captions	Select this option to display component captions.

Debugging

Use the Debugging check boxes to select the debugger you want to use and to enable stepping.

- **Integrated debugging** uses the C++Builder Integrated Debugger
- **Hide designers on run** hides designer windows, such as the Object Inspector and Form window, while the application is running. The windows reappear when the application closes.
- **Break on exception** stops the application when it reaches an exception and displays the following information:
- The exception class
- The exception message
- The location of the exception

  When this option is unchecked, exceptions do not stop the running application. If you are stepping your application, you can step through the exception handlers as if going through the code sequentially.

- **Minimize on run** Minimizes C++Builder when you run your application by choosing Run|Run. When you close your application C++Builder is restored.
- **Path for source** Enter the paths where your source files are located. You can specify multiple paths by separating them with a semicolon (;).
- You can enter a path relative to your C++Builder root directory using the **$(BCB)** environment macro.

Autosave options

Specify which files and options are saved automatically by the environment or when you run your program. A check mark means it is enabled.

- **Editor Files** saves all modified files in the Code editor when you choose Run|Run, Run|Trace Into, Run|Step Over, Run|Run To Cursor, or when you exit C++Builder.
- **Desktop** saves the arrangement of your desktop in a project .DSK file when you close a project or exit C++Builder. When you later open the same project, all files opened when the project was last closed are opened again regardless of whether or not they are used by the project.

Library (Options | Environment)

Use the Library page of the Environment Options dialog box to control how the component library is built. The component library is used by the Component palette.

The options on this page take effect whenever you choose Component|Rebuild Library.

Map file

Select the type of map file produced, if any, when you rebuild the library. The map file is placed in the same directory as the library, and it has a .MAP extension.

Map option	Effect
Off	Does not produce map file.
Segments	Linker produces a map file that includes a list of segments, the program start address, and any warning or error messages produced during the link.
Publics	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, and a list of alphabetically sorted public symbols.
Detailed	Linker produces a map file that includes a list of segments, the program start address, any warning or error messages produced during the link, a list of alphabetically sorted public symbols, and an additional detailed segment map. The detailed segment map includes the address, length in bytes, segment name, group, and module information.

Options

- **Build with debug info**, when enabled, specifies that the C++Builder compiles and links the component library with debug information. This makes the resulting CMPLIB32.LIB file larger, but it does not affect performance.

  When you compile a library using debug information, you can use the integrated debugger or Turbo Debugger for Windows (TDW32.EXE) to debug the library file.
- **Save library source code** saves the source code for the library project and C++ files, using the file names CMPLIB32.MAK and CMPLIB32.CPP, respectively.

  Although these files are generated by C++Builder, it does not use these files directly to build a new version of the component library. Instead, C++Builder uses the files CMPLIB32.CLL and CMPLIB32.OPT to build the VCL (it also uses these files to create the .MAK and .CPP component library source files). C++Builder provides this option so you can view the component library source code, and so you can build the library from the command line.
- **Use incremental linker** causes C++Builder to link your projects using the ILINK32, the incremental linker.
- **Link debug version of VCL** causes C++Builder to link the VCLD.LIB version of the VCL. Use this option only when you want to step into the Object Pascal source code of the Visual Component Library; as it will greatly increase the size of your executable files.

Path

The Path settings tell C++Builder where to find C++ header and library files used to build the component library. Specify the Include Path for the header files used by the VCL and the Library Path for the VCL object files, resource files, and the Delphi unit files.

You can specify multiple directories by separating them with a semicolon. See Search Directories for more information on setting these options in C++Builder.

- You can enter a path relative to your C++Builder root directory using the **$(BCB)** environment macro.

Messages

- **Show Hints** causes the compiler to generate hint messages.
- **Show Warnings** causes the compiler to generate warning messages.

Editor (Options | Environment)

See Also

Use the Editor page of the Environment Options dialog box to customize the behavior of the C++Builder editor.

Editor Speed Setting

Use the Editor Speed Settings to configure the editor. They are pre-configured default settings that can be customized.

Option	Automatically sets
Default Keymapping	Auto Indent Mode, Insert Mode, Use tab character, Backspace Unindents, Group Undo, Overwrite Blocks, Use Syntax Highlight
IDE Classic	Auto Indent Mode, Insert Mode, Use tab character, Backspace Unindents, Cursor Through Tabs, Group Undo, Persistent Blocks, Use Syntax Highlight
BRIEF Emulation	Auto Indent Mode, Insert Mode, Use tab character, Backspace Unindents, Cursor Through Tabs, Cursor Beyond EOF, Keep Trailing Blanks, BRIEF Regular Expressions, Force Cut And Copy Enabled, Use Syntax Highlight
Epsilon Emulation	Auto Indent Mode, Insert Mode, Use tab character, Backspace Unindents, Cursor Through Tabs, Group Undo, Overwrite Blocks, Use Syntax Highlight

Editor Options check boxes

Use the following editor options to control text handling in the Code editor. Check the option to enable it.

Check box	When selected
Auto Indent Mode	Positions the cursor under the first nonblank character of the preceding nonblank line when you press Enter.
Insert Mode	Inserts text at the cursor without overwriting existing text. If Insert Mode is disabled, text at the cursor is overwritten. (Use the Ins key to toggle Insert Mode in the Code editor without changing this default setting.)
Use Tab Character	Inserts tab character. If disabled, inserts space characters. If Smart Tab is enabled, this option is off.
Smart Tab	Tabs to the first non-whitespace character in the preceding line. If Use Tab Character is enabled, this option is off.
Optimal Fill	Begins every autoindented line with the minimum number of characters possible, using tabs and spaces as necessary.
Backspace Unindents	Aligns the insertion point to the previous indentation level (outdents it) when you press Backspace, if the cursor is on the first nonblank character of a line.
Cursor Through Tabs	Enables the arrow keys to move the cursor to the beginning of each tab.
Group Undo	Undoes your last editing command as well as any subsequent editing commands of the same type, if you press Alt+Backspace or choose Edit| Undo.
Cursor Beyond EOF	Positions the cursor beyond the end-of-file character.
Undo After Save	Allows you to retrieve changes after a save.
Keep Trailing Blanks	Keeps any blanks you might have at the end of a line.
BRIEF Regular Expressions	Uses BRIEF regular expressions.

Persistent Blocks	Keeps marked blocks selected, even when the cursor is moved, until a new block is selected.
Overwrite Blocks	Replaces a marked block of text with whatever is typed next. If Persistent Blocks is also selected, text you enter is added to the currently selected block.
Double Click Line	Highlights the line when you double-click any character in the line. If disabled, only the selected word is highlighted.
Find Text At Cursor	Places the text at the cursor into the Text To Find list box in the Find Text dialog box when you choose Search|Find. When this option is disabled you must type in the search text, unless the Text To Find list box is blank, in which case the editor still inserts the text at the cursor.
Force Cut And Copy Enabled	Enables Edit|Cut and Edit|Copy, even when there is no text selected.
Use Syntax Highlighting	Enables syntax highlighting. To set syntax highlighting preferences, use the options from the Editor Display page.

Block Indent

Specify the number of spaces to indent a marked block. The default is 1; the upper limit is 16. If you enter a value greater than 16, you will receive an error.

Undo Limit

Specify the number of keystrokes that can be undone. The default value is 32,767 (32K).

Note: The undo buffer is cleared each time C++Builder generates code.

Tab Stops

Set the character columns that the cursor will move to each time you press Tab. If each successive tab stop is not larger than its predecessor, you will receive an error. The default tab stop is 8.

Syntax Extensions

Specify, by extension, which files will display syntax highlighting information. The default extensions are .CPP, .C, .H, and .HPP.

Display (Options | Environment)

Use the Display page of the Environment Options dialog box to select display and font options for the Code editor. The sample window displays the selected font.

The new settings take effect when you click OK.

Display and File check boxes

Configure the editor's display and choose how it saves files.

Check box	Effect
BRIEF Cursor Shapes	Uses BRIEF cursor shapes.
Create Backup File	Creates a backup file that replaces the first letter of the extension with a tilde (~) when you choose File|Save.
Preserve Line Ends	Preserves end-of-line position.
Zoom To Full Screen	Maximizes the Code editor to fill the entire screen. When this option is off, the Code editor does not cover the C++Builder main window when maximized.

Keystroke Mapping

Enables you to quickly switch key bindings.

Mapping	Effect
Default	Uses key bindings that match CUA mappings (default)
Classic	Uses key bindings that match Borland Classic editor keystrokes
Brief	Uses key bindings that emulate most of the standard BRIEF keystrokes
Epsilon	Uses key bindings that emulate a large part of the Epsilon editor

Visible Right Margin

Check to display a line at the right margin of the Code editor.

Right Margin

Set the right margin of the Code editor. The default is 80 characters. The valid range is 0 to 1024. If you enter a value larger than 1024, you will receive an error.

Editor Font

Select a font type from the available screen fonts installed on your system (shown in the list). The Code editor displays and uses only monospaced screen fonts, such as `Courier`. Sample text is displayed below the combo box.

Size

Select a font size from the predefined font sizes associated with the font you selected in the Font list box. Sample text is displayed below the combo box.

Sample text

Displays a sample of the select editor font and size.

Colors (Options | Environment)

See Also

Use the Colors page of the Environment Options dialog box to specify how the different elements of your code appear in the Code editor.

You can specify foreground and background colors for anything listed in the Element list box. The sample Code editor shows how your settings will appear in the C++Builder Code editor.

Color Speed Settings

Enables you to quickly configure the Code editor display using predefined color combinations. The sample Code editor shows how your settings will appear in the C++Builder Code editor.

Option	Effect
Defaults	Displays reserved words in bold. Background is white.
Classic	Displays reserved words in light blue and code in yellow. Background is dark blue.
Twilight	Displays reserved words and code in light blue. Background is black.
Ocean	Displays reserved words in black and code in dark blue. Background is light blue.

Element

Specifies syntax highlighting for a particular code element. You can choose from the Element list or click the element in the sample Code editor.

The element options are:

Whitespace	Marked block
Comment	Search match
Reserved Word	Execution point (for debugging)
Identifier	Enabled break (for debugging)
Symbol	Disabled break (for debugging)
String	Invalid break (for debugging)
Integer	Error line
Float	Preprocessor
Ocatal	Illegal char
Hex	Plain text
Character	Right margin

Color Grid

Sets the foreground (FG) and background (BG) colors for the selected code element.

To select a color using the mouse, choose one of the following methods:

- Click a color to select it as the foreground color.
- Right-click a color to select it as the background color.

If you choose the same color for the foreground and the background, it is marked as FB (this is not recommended, as you will be unable to read any text).

Text Attributes check boxes

Specify format attributes for the code element. The attribute options C++Builder supports are:

- Bold
- Italic
- Underline

Use defaults for check boxes

Display the code element using default Windows system colors (foreground, background, or both).

Unchecking either option restores the previously selected color or, if no color has been previously selected, sets the code element to the Windows system color.

Note: To change the Windows system colors, use the Windows Control Panel.

Using syntax highlighting

See Also

Syntax highlighting changes the colors and attributes of your code in the editor, making it easier to quickly identify parts of your code.

To enable syntax highlighting,

- On the Editor Options page of the Options|Environment dialog box, check the Use Syntax Highlight option.

To change the syntax highlighting colors for elements of your code,

- Use the Editor Colors page of the Options|Environment dialog box.

Palette (Options | Environment)

Use the Palette page of the Environment Options dialog box to customize the way the component palette appears. You can rename, add or remove, or reorder pages and components.

Pages
Lists the pages in the Component palette. Pages are listed in the order they currently appear. You can rearrange these pages or view their content so that you can rearrange their components.

The Library page contains a listing of all the components installed in the library. You can move any component onto this page, however, that will not create a second instance of the component within the library.

Components
Lists the components in their current order for each page of the Component palette. The components displayed correspond to the currently selected page in the Pages list box. You can rearrange the order in which components appear on a page, or move them to a different page.

Up arrow and down arrow
Click the up arrow or the down arrow to change the position of the selected page or component. You can also drag pages or components to a new position.

Add
Click Add to display the Add Page dialog box, where you can create new pages on the Component palette.

Once you have created a new Component palette page, you can then move components from other pages onto it, or you can add new components.

Delete
Click Delete to remove the selected page or component from the palette. Before you can delete a page, it must be empty of components.

If you accidentally delete a component, you restore that component using the Library page.

Rename
Click Rename to display the Rename Page dialog box, where you can rename the pages on the Component palette.

Reset Defaults
Click Reset Defaults to reset the Component palette to the layout that is specified in the component library.

Rename Page dialog box

See Also

Use this dialog box to specify a new name for a page on the Component palette.

(You first need to select the page you want to rename from Pages list box on the Palette page of the Options|Environment dialog box.)

To open this dialog box,

- Click the Rename button on the Palette page of the Options|Environment dialog box.

Page Name

Enter the new name for the page in the Page Name edit box.

When you click OK, the new name is reflected in the Pages list box, but the new name is not reflected in the Component palette until you click OK in the Environment Options dialog box.

To exit this dialog box without changing the page name, choose Cancel.

Add Page dialog box

See Also

Use this dialog box to add a new page to the Component palette.

The new page is added to the end of the Pages list. You can change the position of the page using the Palette page of the Options|Environment Options dialog box.

To open this dialog box, click the Add button on the Palette page of the Options|Environment dialog box.

Page Name

Enter the new name for the page in the Page Name edit box.

When you click OK, the new name is added in the Pages list box, but the new page is not added to the Component palette until you click OK in the Environment Options dialog box.

To exit this dialog box without changing the page name, choose Cancel.

Options | Repository

Choose Options|Repository to display the Object Repository dialog box. You use the Object Repository dialog box to add, delete, and rename the objects contained in the Object Repository, and to add and delete entire Object Repository pages. You also use this dialog to specify template and wizard options for forms and projects.

Object Repository dialog box

The settings in the Object Repository Options dialog box affect the behavior of C++Builder when you begin a new project or create a new form in an open project. This is where you specify

- Default project
- Default new form
- Default main form

You always have the option to override these defaults by choosing File|New and selecting from the New Items dialog box.

By default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options.

Pages

This list box displays the pages in the Object Repository. When you select a page, the items on that page appear in the Objects list box. Select [Object Repository] to view all items in the Object Repository.

Objects

The Objects list box displays the items on the currently selected page of the Object Repository.

Add Page button

Use the Add Page button to add a new blank page to the Object Repository. Click here to add a page.

To add a page,

1 Click the Add Page button.

The Add Page dialog box appears.

2 Type the name of the page you want to add.

3 Click OK.

Delete Page button

Use the Delete Page to remove an item from the Object Repository.

To delete a page,

1 In the Pages list box, select the name of the page you want to delete.

2 Click the Delete Page button.

The selected page is removed from the Object Repository.

Rename Page button

Use the Rename Page button to rename an item in the Object Repository.

To rename a page,

1 In the Pages list box, select the name of the page you want to rename.

1 Click the Rename Page button.

The Rename Page dialog box appears.

2 Type the name of the page you want to rename.

3 Click OK.

The renamed page appears in the Pages list box.

Edit Object

Use the Edit Object button to edit the properties of items in the Object Repository.

To edit an object,

1 From the Objects list box, select the item you want to edit.

2 Click the Edit Object button.

  The Edit Object Info dialog box appears.

3 Edit the information as desired.

4 Click OK.

Delete Object

Use the Delete Object button to remove items from the Object Repository.

Object Repository Options

The settings in the Object Repository Options dialog box affect the behavior of C++Builder when you begin a new project or create a new form in an open project. When you select an item in the Objects list box, the appropriate options become available at the bottom of the Objects list box. Depending on the item you select, one or more of the options listed below become available.

- Default project
- Default new form
- Default main form

You always have the option to override these defaults by choosing File|New and selecting from the New Items dialog box.

By default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options. For more information see Customizing the Object Repository.

Up arrow and down arrow

Click the up arrow or the down arrow to change the position of the selected page. You can also move pages using a drag-and-drop operation.

Help menu

Use the Help menu to access the online Help system, which displays in a special Help window.

The Help system provides information on virtually all aspects of the C++Builder environment, Object Pascal language, libraries, and so on.

Choose…	If you want …
Contents	to display the Help Topics dialog box
Keyword Search	to use the Help index
Programmer's Guide	information on Borland C/C++ language elements
VCL Reference	information on the Visual Component Library
RTL Reference	information on the Borland C/C++ runtime library
About	Displays a copyright and version number for C++Builder

Help | Contents

Choose Help|Contents to display the Help Topics dialog box.

To find a topic in Help,

- Click the Contents tab to browse through topics by category.
- Click the Index tab to see a list of index entries: either type the word you're looking for or scroll through the list.
- Click the Find tab to search for words or phrases that may be contained in a Help topic.

Help | Keyword Search

In the Code editor, place the insertion point on or next to a term (such as a class, function, member, or property) or highlight one or more terms and press F1 (or choose Help|Keyword Search) to get

- help about the selected term.
- a list of available topics when there are multiple topics to choose from.
- a dialog displaying the closest match if no related topics are found.

You can get Help this way for most C and C++ language elements such as

- reserved words.
- global variables.
- functions in the Borland run-time, class, and Visual Components libraries.
- Windows API.

Help | Programmer's Guide

Choose Help|Programmer's Guide to display Borland C++Builder Programmers' Guide Help (BCPG.HLP).

Help | VCL Reference

Choose Help|VCL Reference to display the Visual Component Library Help (VCL.HLP).

Help | RTL Reference

Choose Help|RTL Reference to display the Borland C++Builder Runtime Library Help (BCRTL.HLP).

Help | About

Choose Help|About to display the About C++Builder dialog box showing copyright and version information.

Menu designer context menu

See Also

The Menu designer context menu provides quick access to the most common Menu designer commands and to the menu template options.

To display the Menu designer context menu,

Choose one of the following methods:

- Right-click anywhere on the Menu designer.
- Press Alt+F10 when the cursor is in the Menu designer window.

The first three commands on the Menu designer context menu directly perform an action.

Command	Action
Insert	Inserts a placeholder above or to the left of the cursor
Delete	Deletes the selected menu item (and all its sub-items, if any)
Create SubMenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item
Select Menu	Displays a list of the menus in the current form so you can select which menu to edit
Save as Template	Displays the Save Template dialog box where you can name and save a menu as a template
Insert from Template	Displays a list of predefined menus that can be inserted into the current menu
Delete Templates	Displays a list of predefined menu templates that can be deleted
Insert from Resource	Allows you to insert a .mnu file into the Menu designer

The rest of the commands on the Menu designer context menu open dialog boxes. Right click and choose the command and press F1 for more information.

Insert (Menu designer context menu)

Right click and choose Insert from the Menu designer context menu to add a menu item placeholder above or to the left of the selected menu item. This option depends on what is selected in the menu designer.

Delete (Menu designer context menu)

Right click and choose Delete from the Menu designer context menu to remove the selected menu item. All sub items, if any, are also deleted.

Create Submenu (Menu designer context menu)

Right click and choose Create Submenu from the Menu designer context menu to insert a menu item placeholder to the right of the selected menu item and add an arrow to the selected item to indicate a nested level.

Select Menu (Menu designer context menu)

Right click and choose Select Menu from the Menu designer context menu to open the Select Menu dialog box.

Select Menu dialog box

Use this dialog box to quickly select from among the existing form menus.

Save As Template (Menu designer context menu)

Right click and choose Save As Template from the Menu designer context menu to open the Save Template dialog box, which enables you to save a menu for later reuse.

Save Template dialog box

Use this dialog box to save a menu for reuse.

Insert From Template (Menu designer context menu)

Right click and choose Insert From Template from the Menu designer context menu to open the Insert Template dialog box.

Insert Template Dialog Box

Use this dialog box to add a predesigned menu to the active menu component.

Delete Templates (Menu designer context menu)

Right click and choose Delete Templates from the Menu designer context menu to open the Delete Templates dialog box.

Delete Templates dialog box

Use this dialog box to select and remove a predesigned menu.

Note: After you delete a template, you cannot retrieve it.

Insert from Resource (Menu designer context menu)

Right click and choose Insert From Resource from the Menu designer context menu to open the Insert Menu From Resource dialog box.

Insert Menu From Resource dialog box

Use this dialog box to import a menu from a Windows resource (.RC) file. You first need to save each individual menu as a separate resource file.

Dialog box options

Look in

Select the directory whose contents you want to view. In the current directory, the file type in the List Files of Type combo box appear in the Files list box. Use the buttons to change directories, or drives.

Displays the files in the current directory that match the file type in the List Files Of Type combo box.

File Name

Enter the name of the file you want to use, or enter wildcards to use as filters in the Files list box.

Files Of Type

Choose the type of file you want to open; the default file type is a menu file (.MNU). All files in the current directory of the selected type appear in the Files list box.

Form context menu

Use the Form context menu to manipulate components in the form at design time.

To display the Form context menu,

1. On the form, select the component or components you want to manipulate.

   (Select a single component by clicking it. Select more than one by dragging across them or by holding down Shift while clicking each one.)

2. Right-click anywhere on the form, or press Alt-F10.

The following commands always appear on the Form context menu:

Align to Grid

Bring to Front

Send to Back

Align

Size

Scale

Tab Order

Creation Order

Add to Repository

View as Text

Other commands appear on the Form context menu when you select certain components on the form. The additional commands are:

Query Builder

Execute

Edit Report

Next Page

Previous Page

Your installation of C++Builder may have additional commands installed by a third party. To get help on a command not listed above, display the Form context menu and select the command. This activates the module named in the command. Then, press F1 to get help on that module.

Query Builder (Form context menu)

To open the Visual Query Builder, choose Query Builder from the Form context menu when the Query component is selected. If a database is not already open, this command opens the Databases dialog box which enables you to select a database.

Execute (Form context menu)

At design time, to perform the process specified in the Mode property, choose Execute from the Form context menu when you have the BatchMove component selected.

The Mode property enables you to perform any of the following tasks:

- Copy a dataset to a table.
- Append a dataset to a table.
- Update a table with data from a dataset.
- Displaying data using data from a control.

To run this process at run time, you must call the Execute method for BatchMove.

Edit Report (Form context menu)

To load QuickReports and open the report you specified using the ReportName and ReportDir properties, choose Edit Report from the Form context menu while you have the Report component selected, .

If you have not previously specified a report, C++Builder allows you to select a report.

You can also perform this command by double-clicking the Report component on the form.

Next Page (Form context menu)

To set the ActivePage property to the next page in the notebook, choose Next Page from the Form context menu when you have either the Notebook component or TabbedNotebook component selected, .

The next page is determined by the page order which you can specify using the Notebook editor.

Previous Page (Form context menu)

To set the ActivePage property to the previous page of the notebook, choose Previous Page from the Form context menu when you have either the Notebook component or the TabbedNotebook component selected, .

The previous page is determined by the page order which you can specify using Notebook editor.

Align To Grid (Form context menu)

Choose Align To Grid from the Form context menu to align the selected components to the closest grid point.

You can specify the size of the grid on the Preferences page of the Options|Environment dialog box.

This context menu command works the same as Edit|Align To Grid.

Bring To Front (Form context menu)

Choose Bring To Front from the Form context menu to move a selected component in front of all other components on the form.

This is called changing the component's z-order.

This context menu command works the same as Edit|Bring To Front.

Send To Back (Form context menu)

Choose Send To Back from the Form context menu to move a selected component behind all other components on the form.

This is called changing the component's z-order.

This context menu command works the same as Edit|Send To Back.

Align (Form context menu)

Choose Align from the Form context menu to open the Alignment dialog box.

Alignment dialog box

Use this dialog box to line up selected components in relation to each other or to the form.

- The Horizontal alignment options align components along their right edges, left edges, or center.
- The Vertical alignment options align components along their top edges, bottom edges, or center.

The options for Horizontal or Vertical alignment are

Option	Description
No Change	Does not change the alignment of the component
Left Sides	Lines up the left edges of the selected components (horizontal only)
Centers	Lines up the centers of the selected components
Right Sides	Lines up the right edges of the selected components (horizontal only)
Tops	Lines up the top edges of the selected components (vertical only)
Bottoms	Lines up the bottom edges of the selected components (vertical only)
Space Equally	Lines up the selected components equidistant from each other
Center in Window	Lines up the selected components with the center of the window.

This context menu command works the same as Edit|Align.

Size (Form context menu)

Choose Size from the Form context menu to open the Size dialog box.

Size dialog box

Use this dialog box to resize multiple components to be exactly the same height or width.

- The Horizontal options align the width of the selected components.
- The Vertical options align the height of the selected components.

The options for Horizontal or Vertical sizing are

Option	Description
No Change	Does not change the size of the components.
Shrink To Smallest	Resizes the group of components to the height or width of the smallest selected component.
Grow To Largest	Resizes the group of components to the height or width of the largest selected component.
Width	Sets a custom width for the selected components. To use this option, you must set Horizontal to Enter Value.
Height	Sets a custom height for the selected components. To use this option, you must set Vertical to Enter Value.

This context menu command works the same as Edit|Size.

Scale (Form context menu)

Choose Scale from the Form context menu to open the Scale dialog box.

Scale dialog box
Use this dialog box to proportionally resize the form and all of its components.

Scaling Factor In Percent
Enter a percentage to which you want to resize the form.

Percentages over 100 grow the form.

Percentages under 100 shrink the form.

This context menu command works the same as Edit|Scale.

Tab Order (Form context menu)

Choose Tab Order from the Form context menu to open the Edit Tab Order dialog box.

Edit Tab Order

Use this dialog box to modify the current tab order of the components on the active form or within the selected component if that component can contain other components.

Dialog box options

Controls

Lists the components on the active form in their current tab order. The first component listed is the first component in the tab order.

Up

Click Up to move the component selected in the Controls list box higher in the tab order.

Down

Click Down to move the component selected in the Controls list box lower in the tab order.

Creation Order (Form context menu)

Choose Creation Order from the Form context menu to open the Creation Order dialog box.

Creation Order dialog box

Use this dialog box to specify the order in which your application will create nonvisual components.

The list box displays only those nonvisual components on the active form, their type, and their current creation order. The default creation order is determined by the order in which you placed the nonvisual components on the form.

To change the creation order

1. Select the component name.
2. Click the up button to move the component creation order up, or click the down arrrow to move its creation order down.
3. To save your changes, click OK.

This context menu command works the same as Edit|Creation Order.

Add To Repository (Form context menu)

Choose Add To Repository from the Form context menu to open the Add To Repository dialog box. Use this command to easily add any form to the Object Repository.

Once you've designed a custom dialog box, you might want to reuse it in other projects. The best way to do this is to add the form to the Object Repository.

Saving a form as an object is similar to saving a copy of the form under a different name. When you save a form as a object, however, it then appears in the Object Repository.

View as Text (Form context menu)

Use this command to view a text description of the form’s attributes.

Note: This command changes to View as Form when you view the form as text.

View as Form (Form context menu)

Use this command to view the form as a visual object.

Note: This commad changes to View as Text when you view the form as a visual object.

Select Icon dialog box

Use the Select Icon dialog box to choose a bitmap to represent your template in the New Items dialog box.

You can use a icon of any size, but it will be cropped to 60 x 40 pixels.

To open this dialog box,

- Click the Browse button in the Add To Repository dialog box.

Dialog box options

Look in

Displays the current directory. Use the icons to change or add directories if needed.

File Name

Enter the name of the file you want to use, or enter wildcards to use as filters in the Files list box.

Files Of Type

Choose the type of file you want to open; the default file type is a bitmap file (.BMP). All files in the current directory of the selected type appear in the Files list box.

Project Manager context menu

The Project Manager context menu contains commands that enable you to manage your project.

The commands on the Project Manager context menu are:

Save Project

Add To Repository

New Unit

New Form

Add File

Remove File

View Unit

View Form

View Project Source

Options

Update

To open the Project Manager context menu, do one of the following:

- Right-click anywhere on the SpeedBar.
- Press Alt+F10 when this is the active window.

Save Project (Project Manager context menu)

Choose Save Project from the Project Manager context menu to store changes made to all files in the open project using each file's current name.

If you try to save a project that has an unsaved project file or unit file, C++Builder opens the Save As dialog box, where you enter the new file name.

This context menu command works the same as File|Save Project.

Add To Repository (Project Manager context menu)

Choose Add To Repository from the Project Manager context menu to open the Save Project Template dialog box.

Use the Save Project Template dialog box to add a project template to the Object Repository.

Save Project Template dialog box

Use this dialog box to save a project template to the Object Repository.

After saving an application as a template, use the Edit Object Info dialog box to edit the description, delete the template, or change the bitmap.

Dialog box options

Title

Enter the name of the template.

The maximum length for a title is 40 characters.

Description

Enter a description of the template. The description appears under the template name on the Select Template dialog box. The maximum length for a description is 255 characters.

Page

From the drop down list box, choose the name of the page (probably Projects) you want the template to appear on.

Author

Enter text indentifying the author of the application.

Author information only appears when you select the View Details option from the context menu.

Template bitmap

Click the Browse button to open the Select Bitmap dialog box.

You can use a bitmap of any size, but it will be cropped to 60 x 40 pixels.

New Unit (Project Manager context menu)

Choose New Unit from the Project Manager context menu to create a new unit and add it to the project. When you add a new unit the Code editor becomes the active window and the new unit is the active page.

The new unit is titled UnitXX.CPP. XX represents the unit number. For example, the first form is Unit1, the second Unit2, and so on.

You can change the unit name by saving the file with File|Save As, or by saving the entire project using Save Project from the Project Manager context menu.   Changing a unit name by any other means might cause an error.

This context menu command works the same as the File|New Unit.

New Form (Project Manager context menu)

Choose New Form from the Project Manager context menu to create a blank form and a new unit and add them to the project.

The new form is titled FormXX and the new unit is UnitXX.CPP. (XX represents the form/unit number. For example the first form is Form1, the second Form2, and so on.)

You can change the name of the form by editing the Name property from the Object Inspector.

You can change the unit name by saving the file with File|Save As or by saving the entire project using Save Project on the Project Manager context menu.

This context menu command works the same as File|New Form.

Add File (Project Manager context menu)

Choose Add File from the Project Manager context menu to open the Add To Project dialog box.

Add To Project
Use the Add To Project dialog box to add an existing unit and its associated form to the C++Builder project. When you add a unit to a project, C++Builder automatically adds that unit to the project file.

Dialog box options

File Name
Enter the name of the file you want to load, or enter wildcards to use as filters in the Files list box.

Files
Displays the files in the current directory that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box.

List Files Of Type
Choose the type of file you want to open; the default file type is Source file (.PAS). All files in the current directory of the selected type appear in the Files list box.

Directories
Select the directory whose contents you want to view. In the current directory, files that match the wildcards in the File Name edit box or the file type in the List Files Of Type combo box appear in the Files list box.

Drives
Select the current active drive. The directory structure for the current drive appears in the Directories list box.

This context menu command works the same as File|Add File.

Remove File (Project Manager context menu)

Choose Remove File from the Project Manager context menu to remove the module selected in the Project Manager from the current project file.

If you have modified the file you are removing during this editing session, C++Builder prompts you to save your changes, just in case you want to use the form or unit in another project. If you have not modified the file, C++Builder removes that file from the project without prompting you.

If a file has been modified during the current editing session, it is bold in the Project Manager.

Warning: Do not delete unit files by using other file management programs, or directly from the DOS prompt. Doing so causes errors.

This context menu command works the same as File|Remove File.

View Unit (Project Manager context menu)

Choose View Unit from the Project Manager context menu to make the selected module in the Project Manager the active page of the Code Editor. It also makes the Code Editor the active window.

If the module you want to view is not currently open, C++Builder opens it.

This context menu command works the same as View|Unit.

View Form (Project Manager context menu)

Choose View Form from the Project Manager context menu to make the form associated with the module selected in the Project Manager the active window.

If the form you want to view is not currently open, C++Builder opens it.

This context menu command works the same as View|View Form.

View Project Source **(Project Manager context menu)**

Choose View Project from the Project Manager context menu to make the project file the active page in the Code editor.

If the project file is not currently open, C++Builder opens it.

Options (Project Manager context menu)

Choose Options from the Project Manager context menu to open the Options|Project dialog box.

You can use this dialog box to set compiler directives.

This context menu command works the same as the Options|Project.

Update (Project Manager context menu)

Choose Update from the Project Manager context menu to synchronize any changes you have made directly in the project file itself.

Note: This command remains disabled (dimmed) unless you have *manually* edited the project file. Since C++Builder maintains the project file for you automatically, manually modifying this file is not recommended.

SpeedBar context menu

The SpeedBar context menu contains commands that enable you to edit or rearrange the buttons on the SpeedBar. You can also use the SpeedBar context menu to hide the SpeedBar.

The commands on the SpeedBar context menu are:

Show Hints

Hide

Help

Properties

To display the SpeedBar context menu,

- Right-click anywhere on the SpeedBar.

Properties (SpeedBar context menu)

Right click and choose Properties from the SpeedBar context menu to open the SpeedBar Editor dialog box.

SpeedBar editor

Use this dialog box to add buttons that represent menu commands to the SpeedBar, remove buttons from the SpeedBar, or rearrange buttons on the SpeedBar.

Dialog box options

Categories

Select a menu whose commands you want to add as buttons to the SpeedBar. The commands on the selected menu appear in the Command list box.

Commands

Drag and drop a command from this list box onto the SpeedBar. The Commands list box displays all the commands available on the menu selected in the Categories list box.

The icon to the left of the menu command shows how the button will appear on the SpeedBar.

Reset

Click Reset to reset the SpeedBar to the default configuration.

When the SpeedBar editor is open, you can delete or rearrange any of the buttons currently on the SpeedBar; however, none of the buttons on the SpeedBar are active.

Hide

Right click and choose Hide from the following context menus to close that interface element.

Alignment Palette context menu

Component Palette context menu

Object Inspector context menu

SpeedBar context menu

If you close an interface element, you can display it again using the View menu.

Show Hints

Right click and choose Show Hints from the following context menus to toggle the display of Help Hints.

Alignment Palette context menu

Component Palette context menu

SpeedBar context menu

When this command is checked, Help Hints are enabled.

Help

Right click and choose Help from any of the following context menus to get Help on using that element.

Alignment Palette context menu

Component Palette context menu

Object Inspector context menu

SpeedBar context menu

Component palette context menu

The Component palette context menu enables you to edit or rearrange the components on the Component palette. You can also use the Component palette context menu to hide the Component palette.

Properties

Show Hints

Hide

Help

To display the Component palette context menu,

- Right-click anywhere on the SpeedBar.

Properties (Component palette context menu)

Choose Properties from the Component palette context menu to open the Palette page of the Options| Environment dialog box.

You can use this dialog box to rearrange the Components on the Component palette.

Alignment palette context menu

The Alignment palette context menu contains the following commands:

Stay On Top

Show Hints

Hide

Help

Stay On Top (Alignment palette context menu)

Choose Stay On Top from the Alignment palette context menu to keep the Alignment palette in front of all other windows and dialog boxes.

Object Inspector context menu

The Object Inspector context menu provides you with commands for closing the Object Inspector, displaying Help, and for keeping the Object Inspector the topmost window.

The commands on the Object Inspector context menu are:

Revert to Inherited

Expand

Collapse

Stay On Top

Hide

Help

Revert to Inherited (Object Inspector and Form context menu)

Right click and choose Revert to Inherited from the Object Inspector when you want to change an object that has had its properties overwritten back to the original inherited behavior. This option is only available when the object has properties.

Stay On Top (Object Inspector context menu)

Right click and choose Stay On Top from the Object Inspector context menu to keep the Object Inspector in front of all other C++Builder windows and dialog boxes.

Expand (Object Inspector context menu)

Right click and choose Expand from the Object Inspector context menu to view the nested properties of the selected property.

Properties with nested properties show a plus (+) sign on their left side in the Object Inspector. You need to view these nested properties to set them.

Collapse (Object Inspector context menu)

Right click and choose Collapse from the Object Inspector context menu to hide the nested properties of the selected property.

Properties with nested properties show a plus (+) sign on their left side in the Object Inspector. You need to view these nested properties to set them.

Code editor context menu

See Also

The Code editor context menu contains commands for navigating, modifying, and debugging your source code. This menu is unique to the Code editor, and the commands contained in the menu pertain only to the Code editor.

The context menu commands are listed below. To view detailed information on a Code editor context menu command, click that command in the list below:

Open Source/Header file

Close Page

Open File At Cursor

New Edit Window

Topic Search

Toggle Breakpoint

Run to Cursor

Inspect

Goto Address

Evaluate/Modify

Add Watch at Cursor

View as Form

Read only

Message View

View CPU

Properties

To open the Code editor context menu, do one of the following:

- Right-click anywhere in the Code editor window.
- Press Alt+F10 when the Code editor window is active.

Open Source/Header file (Code editor context menu)

See Also

Choose Open Source/Header file from the Code editor context menu to open or display the header or source file related to the currently viewed file.

For example, if you have Unit1.cpp open and choose Open Source/Header file, Unit1.h will be opened if it is not already, or will be switched to become the active file in the edit window.

Close Page (Code editor context menu)

Choose Close Page from the Code editor context menu to close the current page in the Code editor window.

If you have modified code, not saved the changes, and this is the last page open in a file, C++Builder opens the Save As dialog box, where you can enter a new file name.

If you are closing the last page in the project and have not saved it yet, C++Builder opens the Save As dialog box, where you can enter a name for the project.

Open File At Cursor (Code editor context menu)

See Also

Choose Open File At Cursor from the Code editor context menu to open the file at the current cursor position.

C++Builder searches for files with the default extension of .CPP, unless another file extension is explicitly specified. Similarly, C++Builder uses the directory settings for unit and include files specified in the Directories/Conditionals page of the Options|Project dialog box.

To change directory settings for unit and include files,

1. Choose Options|Project from the main C++Builder menu.
2. Click the Directories/Conditionals page.
3. Set unit and include directories as you want.
4. Choose OK to put your choices into effect.

New Edit Window (Code editor context menu)

Choose New Edit Window to open a new Code Editor that contains a copy of the active page from the original Code eitor.

Any changes you make to either the original or the copy are reflected in both files.

So that you can distinguish between the windows, the caption in the original window is postfixed with a 1, the first copy with a 2, the second copy with a 3, and so on.

Topic Search (Code editor context menu)

Choose Topic Search from the Code editor context menu to display a Help window for the word or token at the cursor in the Code editor.

If no Help topic exists, a dialog box is displayed, with the closest match highlighted.

Toggle Breakpoint (Code editor context menu)

Choose Toggle Breakpoint from the Code editor context menu to toggle a breakpoint on and off at the current cursor position.

If no breakpoint is set when you choose this command, C++Builder sets one and turns it on. If a breakpoint is already set, choosing this command toggles the breakpoint off.

To modify breakpoint properties,

- Right-click an existing breakpoint in the Breakpoint List window and choose Edit Breakpoint.

Run To Cursor (Code editor context menu)

Choose Run To Cursor to run the loaded program up to the location of the cursor in the Module window.

When you run to the cursor, your program is executed at full speed, then pauses and places the execution point on the line of code containing the cursor.

You can use Run To Cursor to run your program and pause before the location of a suspected problem. You can then use Run|Step Over or Run|Trace Into to control the execution of individual lines of code.

An alternative way to perform this command is:

- Choose Run|Run To Cursor.

Inspect (Code editor context menu)

See Also

Choose Inspect to to open an Inspector window for the term at the curent insertion point. If the insertion point is on a blank space when you choose this command, an empty Inspector window displays where you can enter an item you want to inspect.

- This command is available only while you run your program from the IDE.

Goto Address (Code editor context menu)

The Go to Address command prompts you for a new area of memory to display in the Disassembly pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with 0x. If the CPU view is already open, it becomes the active view.

- This command is available only while you run your program from the IDE.

Evaluate/Modify (Code editor context menu)

The Evaluate/Modify command opens the Evaluate/Modify dialog box, which lets you evaluate or change the value of an existing expression.

An alternate way to perform this command is:

- Choose Run|Evaluate/Modify from the main menu.

Add Watch at Cursor (Code editor context menu)

See also

The Add Watch at Cursor command opens the Watch Properties dialog box, where you can create and modify watches. After you create a watch, use the Watch List window to display and manage the current list of watches.

Alternate ways to perform this command are:

- Choose Run|Add Watch from the Code editor context menu.
- Choose Add Watch from the Watch List context menu.
- Right-click an existing watch in the Watch List window and choose Edit Watch from the Watch List context menu.

View as Form (Code editor context menu)

Choose View as form (or View from Text command) from the Code editor context menu to toggle viewing a form file as a text file or a form.

Caution	If you make changes to the text file that are not supported by C++Builder, you will not be able to view the file as a form.

Read Only (Code editor context menu)

Choose Read Only from the Code editor context menu to make the current open file read only. When a file is read only, you cannot make any changes to the file.

When you mark a file as read only this command is checked on the Code editor context menu and "Read only" is displayed in the Code editor status line.

Message View (Code editor context menu)

Choose Message View to display or hide the message pane at the bottom of the Code editor that shows messages, warnings, and errors generated as you build your program.

View CPU (Code editor context menu)

See also

Choose this command to display the CPU view. If the CPU view is already open, it becomes the active view.

Properties (Code editor context menu)

Choose Properties from the Code editor context menu to set editor environment options.

View Source
Positions the Code editor at the source location of the highlighted error, but does not make the Code editor the active window. If the source file is not already open in the Code editor, it appears in a new Code editor page.

Edit Source
Positions the Code editor at the source location of the highlighted error and makes the Code editor the active window. If the source file is not already open in the Code editor, it appears in a new Code editor page.

New Items context menu

See Also

The context menu provides the following options for items in the New Items dialog box.

- View Large Icons
- View Small Icons
- View List
- View Details
- Arrange by Name
- Arrange by description
- Arrange by Date
- Arrange by Author
- Properties

Note: Selecting Properties from the context menu opens the Object Repository dialog box. You can use this dialog box to edit, add pages to, and rename items in the Object Repository. You can also access this dialog from the Tools|Repository menu.

View large icons (Object Repository context menu)

Choose this option to view large icons in the Object Repository.

View small icons (Object Repository context menu)

Choose this option to view small icons in the Object Repository.

View list (Object Repository context menu)

Choose this option to view a list of names and icons in the Object Repository.

View details (Object Repository context menu)

Choose this option to view details about the items in the Object Repository. The details include name, description, modified, and author.

Arrange by name (Object Repository context menu)

Choose this option to view by name, the items in the Object Repository.

Arrange by description (Object Repository context menu)

Choose this option to view by description, the items in the Object Repository. If descriptions are blank, the previous sort order is used.

Arrange by date (Object Repository context menu)

Choose this option to view items in the Object Repository ordered by date.

Arrange by author (Object Repository context menu)

Choose this option to view items in the Object Repository ordered by author.

Properties (Object Repository context menu)

Choose this option to display the Object Repository dialog box. You can use this dialog box to edit, add pages to, and rename items in the Object Repository. You can also access this dialog from the Tools| Repository menu.

About the integrated debugger

See Also

No matter how careful you are when writing code, your programs are likely to contain errors, or bugs, that prevent them from running the way you intended. Debugging is the process of locating and fixing errors in your programs. C++Builder provides debugging features, collectively referred to as the integrated debugger, that let you find and fix errors in your programs. The integrated debugger is a full-featured debugger that enables you to

- Control the execution of your program
- Monitor the values of variables and items in data structures
- Modify the values of data items while debugging

Types of errors

There are three basic types of program errors:

- Compile-time
- Logical errors
- Runtime errors

The integrated debugger can help you track down both runtime errors and logic errors. By running to specific program locations and viewing the state of your program at those places, you can monitor how your program behaves and find the areas where it is not behaving as you intended.

Compile-time errors

Errors that violate a rule of language syntax. You cannot compile your program unless it contains valid statements.

The most common causes of compile-time (syntax) errors are

- Typographical mistakes
- Missing semicolons
- References to undeclared variables
- Wrong number or type of arguments passed to a function
- Wrong type of values assigned to a variable

Runtime errors

Runtime errors occur when your program contains valid statements, but the statements cause errors when they are executed. For example, your program might try to open a nonexistent file, or it might try to divide a number by zero. The operating system detects runtime errors and stops program execution when they occur.

Using the debugger, you can run to a specific program location. From there, you can execute your program one statement at a time, watching the behavior of your program with each step. When you execute the statement that causes your program to fail, you can fix the source code, recompile the program, and resume testing.

Logic errors

Logic errors occur when your program statements are valid, but the actions they perform are not the actions you intended. For example, logic errors occur when variables contain incorrect values, when graphic images do not look right, or when the output of your program is incorrect.

Logic errors are often the difficult to find because they can show up in unexpected places. You need to thoroughly test your program to ensure that it works as designed. The debugger helps you locate logic errors by monitoring the values of variables and data objects as your program executes.

Fixing syntax errors

If your code has compile-time (syntax) errors and you try to compile it, the Message View of the Code editor opens and displays the errors and warnings generated.

To correct syntax errors,

1. In the Message View, double-click the error or warning that you want to fix. (If the Message View is not open, right-click the Code editor and choose Message View.)

   The IDE positions your cursor on the line in your source code that caused the problem.
2. Make your correction.
3. If your code has more than one problem, double-click another error or warning in the Message window.
4. Choose Project|Build All or Project|Make to recompile your program.
5. Choose Run|Run to verify that your program is operating correctly.

Planning a debugging strategy

After program design, program development consists of a continuous cycle of coding and debugging. Only after you thoroughly test your program should you distribute it to your end users. To ensure that you test all aspects of your program, it is best to have a thorough plan for your debugging cycles.

One good debugging method involves dividing your program into different sections that you can debug systematically. By closely monitoring the statements in each section, you can verify that each area is performing as designed. If you do find a programming error, you can correct the problem in your source code, recompile the program, and resume testing.

Using the integrated debugger

Although there are many ways to debug code, you will typically use one or more of the following steps:

1. Preparing your project for debugging by compiling and linking your program with debug information.
2. Control Program Execution by running to a program location you would like to examine.
3. Examine the state of the program data values and view the program output.
4. Modify program data values to test bug fixes.
5. Reset or pause the debugging session.
6. Fix the error.

Preparing your project for debugging

If you find a runtime or logic error in your program, you can begin a debugging session by running your program under the control of the debugger:

1. Compile and link your program with debug information.
2. Run your program from the IDE.

Generating debug information for your project

The IDE automatically generates debug information. To manually choose to turn on debug information for your project,

1. Choose Options|Project.
2. Click the C++ tab.
3. Click Full Debug in the Speed Settings section to simultaneously set the following debugger options:

- Debug information – to include symbolic debug information.
- Line number information – to include line numbers in the debug information in the object files.
- Disable Inline Expansions – to ensure that the debugger can trace into inline functions

4. Click the Linker tab and, if it is not already checked, click Include Debug Information. This option is also turned on automatically when you click Full Debug on the C++ tab.

Enabling the debugger

The debugger is enabled automatically. To manually choose to enable the debugger,

1. Choose Options|Environment and click the Preferences tab.
2. Check Integrated Debugging. This option is on by default.
3. Check Minimize On Run if you want to minimize the C++Builder environment when you run your program. This option is on by default.
4. Click Hide Designers On Run to close the Object Inspector and Form Designer when you run your program. This option is on by default.

Debugging VCL source

To reduce the amount of time it takes to debug your program, you generally will not want to examine the VCL source code included in your project. In cases when you need to examine VCL source (when you develop your own components, for example), use the following steps

1. Choose Options|Project.
2. Click the Linker tab to access the linker options.
3. Check Link Debug Version of VCL. This option is off by default.

- Choosing this option typically increases the file size of your program and decreases performance.

Turning debugging information off

Adding debug information increases the file size of your program. When you have fully debugged your program, be sure to build the final executable files with debugging information turned off to reduce the final size of your program files.

To turn off debugging information

1. Choose Options|Project
2. Click the C++ tab then and click Release under Speed Settings.

Running your program in the IDE

After you compile your program with debug information, you can begin a debugging session by running your program from the IDE. Doing so lets you control when your program runs and when it pauses. Whenever your program is paused in the IDE, the debugger takes control.

When you run your program under the control of the debugger, it behaves as it normally would; your program creates windows, accepts user input, calculates values, and displays output. When your program is not running, the debugger has control, and you can use its features to examine the current state of the program. By viewing the values of variables, the functions on the call stack, and the program output, you can ensure that the area of code you are examining is performing as it was designed to.

As you run your program through the debugger, you can watch the behavior of your application in the windows it creates.

- For best results, arrange your screen so you can see both the Code editor and your application window as you debug.

Debugging with program arguments

To pass runtime arguments to the program you want to debug,

1. Choose Run|Parameters.
2. In the Run Parameters dialog box, type the arguments to pass to your program when you run it under debugger control and click OK.

Controlling program execution

The most important aspect of a debugger is that it lets you control the execution of your program. You can control whether your program will execute a single line of code, an entire function, or an entire program block. By specifying when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

The debugger treats multiple program statements on one line as a single line of code; you cannot individually debug multiple statements contained on a single line of text. In addition, the debugger treats a single statement that spans several lines of text as a single line of code.

The debugger lets you control program execution in the following ways:

- Running to the cursor
- Stepping through code
- Running to a breakpoint location
- Pausing your program

Execution point

The execution point indicates the next line of source code or machine instruction in your program that will be executed when you run your program through the integrated debugger. Whenever you pause program execution, the debugger highlights a line of source code or machine instruction, marking the location of the execution point.

Running to the cursor

See also

When beginning a debugging session, you often run your program to a spot just before the suspected location of the problem. At that point, use the debugger to ensure that all data values are as they should be. If everything appears to be correct, you can run your program to another location, and again check to ensure things are functioning correctly.

You can tell the debugger you want to execute your program normally (not step-by-step) until a certain spot in your code is reached:

1. In the Code editor or CPU window, position the cursor on the line where you want to begin (or resume) debugging.
2. Right-click the Code editor or the Disassembly pane of the CPU window and choose Run To Current.

Stepping overview

Stepping is the simplest way to move through your code one statement or machine instruction at a time. Stepping lets you run your program one line (or instruction) at a time – the next line of code (or instruction) will not execute until you tell the debugger to continue. After each step, you can examine the state of the program, view the program output, and modify program data values. Then, when you are ready, you can continue executing the next program statement.

You can step through code in two basic ways:

Trace Into	The Trace Into command causes the debugger to walk through your code one statement or instruction at a time. If the execution point is located on a function call, the debugger moves to the first line of code or instruction that defines that function. From here, you can execute that function, one statement or instruction at a time. When you step past the return of the function, the debugger resumes stepping from the point where the function was called. (Stepping through your program one statement at a time is known as single stepping.)
Step Over	The Step Over command is the same as Trace Into, except that when the execution point is on a function call, the debugger executes the function at full speed and then pauses on the line of code or instruction following the function call.

Statement stepping and instruction stepping

The debugger lets you step through either

- statements in your source code viewed in the Code editor.
- machine instructions viewed in the CPU window.

The debugger automatically steps through your code at the instruction level and displays the CPU window in the following situations:

- If you start a debugging session by choosing the Trace Into command.
- If the CPU window has focus when you choose the Trace Into or Step Over command.
- When program execution stops at a location for which source code is unavailable. For example, the debugger cannot open the source file if you link a DLL built with debug information but do not include its source file in your project, or if you place the source file in a directory not specified in your project.

Statement Stepping granularity

The debugger steps over single lines of lines of code based on the following rules:

- If you string several statements together on one line, you cannot debug those statements individually; the debugger treats all statements as a single line of code.
- If you spread a single statement over multiple lines in your source file, the debugger executes all the lines as a single statement.

Stepping over code

See also Overview of Stepping

To Step Over, choose the Run|Step Over or press F8 (default key mapping).

When you choose the Step Over command, the debugger executes the code highlighted by the execution point. If the execution point is highlighting a function call, the debugger executes that function at full speed, including any function calls within the function highlighted by the execution point. The execution point then moves to the next complete line of code or executable instruction.

Example

The following code fragment shows how Step Over works. Suppose these two functions are in a program compiled with debug information:

```
func_1() {
  statement_a;
  func_2();
  statement_b;
}

func_2() {
  statement_m;
  func_3();
}
```

If you choose Step Over when the execution point is on `statement_a` in `func_1`, the execution point moves to highlight the call to `func_2`. Choosing Step Over again runs `func_2` at full speed, moving the execution point to `statement_b`. Notice that when you step over `func_2`, `func_3` is also run at full speed.

As you debug, you can choose to Trace Into some functions and Step Over others. Step Over is good to use when you have fully tested a function, and you do not need to single step through its code.

Tracing into code

See also Overview of stepping

To Trace Into code, choose either of the following commands:

- Run|Trace Into or press F7 (default key mapping)
- Run|Trace To Next Source Line or press Shift+F7 (default key mapping).

When you choose Run|Trace Into, the debugger executes the code highlighted by the execution point. If the execution point is highlighting a function call, the debugger moves the execution point to the first line of code or instruction that defines the function being called. If the executing statement calls a function that does not contain debug information, the debugger runs the function at full speed (as if you had chosen the Step over command).

When you choose Run|Trace To Next Source Line, the debugger moves to the next source line in your application, regardless of the control flow. For example, if you select this command when stopped at a Windows API call that takes a callback function, control will return to the next source line, which in this case is the callback function.

- To trace into start-up code that C++Builder generates automatically, see Debugging Start-up Code.

Example

The following code fragment shows how Trace Into works. Suppose these two functions are in a program compiled with debug information:

```
func_1() {
  statement_a;
  func_2();
  statement_b;
}

func_2() {
  int customers;
  statement_m;
}
```

If you choose Trace Into when the execution point is on `statement_a` in `func_1`, the execution point moves to highlight the call to `func_2`. Choosing Trace Into again positions the execution point at the first line in the definition of `func_2`. Another Trace Into command moves the execution point to `statement_m` which is the first executable line of code in `func_2`.

When you step past a function return statement (in this case, the closing function brace), the debugger positions the execution point on the line following the original function call. Here, the debugger would highlight `statement_b` with the execution point.

As you debug, you can choose to Trace Into some functions and Step Over others. Use Trace Into when you need to fully test the function highlighted by the execution point.

Running to a breakpoint

See also

You set breakpoints on lines of source code or address locations (machine instructions) where you want program execution to pause during a run. Using a breakpoint is similar to using the Run to Cursor command in that the program runs at full speed until it reaches a certain point. Unlike Run to Cursor, however, you can have multiple breakpoints and you can choose to stop at a breakpoint only under certain conditions. Once your program's execution is paused, you can use the debugger to examine the state of your program.

Interrupting program execution

See also

Sometimes while debugging, you will find it best to stop program execution or to start the debugging session from the beginning of the program.

Choose…	**To…**
Run|Program Pause	temporarily pause the execution of a running program.
Run|Program Reset	terminate the current debugging session, and start with a fresh slate.

Pausing your program

See also

Instead of stepping through code, you can use a simpler technique to pause your program:

Choose Run|Program Pause and your program will stop executing.

You can then examine the value of variables and inspect data at this state of the program. When you are done, choose Run|Run to continue the execution of your program.

- In most cases, the CPU window will display when you resume debugging after pausing your program, such as when the current instruction does not have corresponding source code.

Restarting a program

See also

Sometimes while debugging, you might need to start over from the beginning of your program. For example, it might be best to restart the debugging session if you have executed past the point where you believe there is a bug, or if variables or data structures become corrupted with unwanted values.

To restart your program, choose Run|Program Reset.

When you terminate the process, the IDE

- resets the integrated debugger so that running or stepping, begins at the start of the program.
- does not change the location of the source code displayed in the Code editor so that you can easily position the cursor to run your program to the line you were on when you reset it.

Fixing program errors

See also

Once you have found the location of the error in your program, you can type the correction directly into the Code editor and the change takes effect immediately. Once you change a line of code in the Code editor, however, the IDE prompts you to rebuild your program before you resume program execution and continue debugging.

Instead of fixing an error while debugging, you might want to test your fix by modifying data values using the debugger. This way, you do not have to recompile your program to see if your fix works.

Using breakpoints

See also

Breakpoints pause program execution during a debugging session at source code or address locations that you specify. You can set breakpoints before potential problem areas, then run your program at full speed. Your program pauses when it encounters a breakpoint, and the Code editor or CPU view Disassembly pane displays the line or address location containing the breakpoint. You can then use the debugger to view the state of your program, or to step over or trace into your code one line or machine instruction at a time.

The IDE keeps track of all your breakpoints during a debugging session and associates them with your current project. You can maintain all your breakpoints from a single Breakpoints List window and not have to search through your source code files to look for them.

Debugging with breakpoints

When you run your program from the IDE, it will stop whenever the debugger reaches the location in your program where the breakpoint is set, but before it executes the line or machine instruction.

- If you set a breakpoint on a line in your source code, the line that contains the breakpoint appears in the Code editor highlighted by the execution point.
- If you set a breakpoint on an address location, the instruction that contains the breakpoint appears in the CPU window Disassembly pane (or in the Code editor on the line that most closely corresponds to the address location) highlighted by the execution point.

At this point, you can perform any other debugging actions.

Setting breakpoints after program execution begins

While your program is running, you can switch to the debugger (just like you switch to any Windows application) and set a breakpoint. When you return to your application, the new breakpoint is set, and your application will pause or perform a specified action when it reaches the breakpoint.

- You must set a breakpoint on an executable line of code or machine instruction. For example, breakpoints set on comment lines, blank lines, declarations, or other non-executable lines of code are displayed as invalid breakpoints in the Code editor, and are disabled when you run your program.

Setting breakpoints

See also

You can set breakpoints before you begin debugging or while your program is running using the Code editor or the CPU window Disassembly pane. Your application will halt when it reaches a breakpoint.

- For a breakpoint to be valid, it must be set on an executable line of code. Breakpoints set on comment lines, blank lines, declarations, or other non-executable lines of code are invalid and become disabled when you run your program.

To set a breakpoint on a line of source code,

Select the line in the Code editor where you want to set the breakpoint, then choose one of the following methods:

- Click the left margin of the line.
- Right-click anywhere on the line and choose Toggle Breakpoint.
- Place the insertion point anywhere in the line and press F5 (default key mapping).
- Right-click the Breakpoint List window and choose Add Breakpoint.

If you know the line of code where you want the breakpoint set,

1. Choose Run|Add Breakpoint and type the source-code line number in the Line Number box.
2. Complete the settings in the Edit Breakpoint dialog and choose New to create the breakpoint.

When you set a breakpoint, the line on which the breakpoint is set becomes highlighted, and a stop-sign appears in the left margin of the breakpoint line.

To set a breakpoint at a specific address location,

1. Open the CPU window and highlight a machine instruction in the Disassembly pane.
2. Right-click the Disassembly pane and choose Toggle Breakpoint or press F5.

- You can also set a breakpoint on either a source or address location from the Breakpoint List window.

Invalid breakpoints

If a breakpoint is not placed on an executable line of code, the debugger considers it invalid. For example, a breakpoint set on a comment, a blank line, or declaration is invalid. If you set an invalid breakpoint, the debugger displays the Invalid Breakpoint error box when you attempt to run the program. To correct this situation, close the error box and delete the invalid breakpoint from the Breakpoint List window. You can then set the breakpoint in the intended location. You can, however, also ignore invalid breakpoints; the IDE disables any invalid breakpoints when you run your program.

- During the linking phase of compilation, lines of code that do not get called in your program are marked as dead code by the linker. In turn, the integrated debugger marks any breakpoints set on dead code as invalid.

The Breakpoint List window

See also

The Breakpoint List window shows all breakpoints currently set in the loaded project. (If no project is loaded, it shows all breakpoints set in the active Code editor page or in the CPU window.) The Breakpoint List shows the file name and line number location along with any condition and pass count associated with each breakpoint. The Breakpoint List window also lets you add, edit, delete, and enable or disable breakpoints. A breakpoint appears grayed if it is either disabled or invalid.

To display the Breakpoint List window, choose View|Breakpoints.

Breakpoint List window commands

The Breakpoint List window offers two sets of commands depending on whether or not you have highlighted a listed breakpoint.

Select a breakpoint, then right-click to access the following commands:

Enabled	Enables or a disables a breakpoint. Disabling a breakpoint hides it from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default. Disabling is useful when you temporarily do not need a breakpoint, but want to preserve its settings.
Delete	Removes a breakpoint. When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. This command is not reversible.
View Source	Locates a breakpoint in your source code quickly. Select a breakpoint in the Breakpoint List window, then use this command to scroll the Code editor to the line location where the breakpoint is set.
Edit Source	Works the same as the View Source command, but also gives the Code editor focus.
Properties	Opens the Edit Breakpoint dialog box, where you can create or modify a breakpoint.

Right-click the Breakpoint list window on an area other than a listed breakpoint to access the following commands:

Add	Provides two submenu options: Source Breakpoint. Opens the Edit Breakpoint dialog box where you can set a breakpoint on a specific line location in your source code. When you run your program, the execution point in the Code editor indicates the breakpoint location. Address Breakpoint. Opens the Edit Breakpoint dialog box where you can set a breakpoint on a specific machine instruction. When you run your program, the execution point in the CPU window Disassembly pane indicates the breakpoint location.
Delete All	Removes all breakpoints.
Disable All	Disables all enabled breakpoints.
Enable All	Enables all disabled breakpoints.

Modifying breakpoint properties

See also

You can specify breakpoint properties when you create a breakpoint, or you can edit the properties after creation. Use the Edit Breakpoint dialog box to modify breakpoint properties.

Edit Breakpoint dialog box

Use the Edit Breakpoint dialog box to add a breakpoint or to modify an existing breakpoint. You can open the Edit Breakpoint dialog box the following ways:

- Choose Run|Add Breakpoint (to add a breakpoint on a source line, but not an address location).
- Choose View|Breakpoint, then right-click the Breakpoint List window, choose Add, and then choose Source or Address.
- Right-click an existing breakpoint in the Breakpoint List window and choose Properties.

Use the following options to specify where and when you want a breakpoint to pause your program:

Filename

Sets or changes the program file for the breakpoint. Enter the name of the program file for the breakpoint. (This option appears only for a breakpoint set on a line of source code in the Code editor.)

Line Number

Sets or changes the line number for the breakpoint. Enter or change the line number for the breakpoint. (This option appears only for a breakpoint set in the Code editor on a line of source code.)

Address

Sets a breakpoint on a machine instruction. Enter a specific starting address or any symbol, such as a variable or a class data member or method, that evaluates to an address. (This setting appears only for a breakpoint set on a machine instruction in the Disassembly pane in the CPU window.)

Condition

Specifies a conditional expression that is evaluated each time the breakpoint is encountered. Program execution stops when the expression evaluates to true. You can enter any valid language expression. All symbols in the expression, however, must be accessible (within scope) from the breakpoint's location, and the expression cannot contain function calls.

- For more information, see Creating Boolean expressions.

Pass Count

Stops program execution at a certain line number or machine instruction after a specified number of passes. The integrated debugger decrements the pass count number each time the line containing the breakpoint is encountered. When the pass count equals 1, program execution pauses.

When you use pass counts with conditions, program execution pauses the *n*th time that the conditional expression is true. The debugger decrements the pass count only when the conditional expression is true.

- For more information, see Using Pass Counts.

Creating conditional breakpoints

See also

When a breakpoint is first set, by default, program execution pauses each time the breakpoint is encountered. The Edit Breakpoint dialog box lets you customize your breakpoints so that your program pauses only when a specified set of conditions is met.

To create a conditional breakpoint,

1. Choose Run|Add Breakpoint.
2. Right-click the Breakpoint List window, choose Add, and then choose Source or Address.
3. Enter the required information in the Edit Breakpoint dialog box.

The integrated debugger provides two types of breakpoint conditions:

- Boolean expressions
- Pass counts

Creating Boolean expressions

See also

The Condition edit box in the Edit Breakpoint dialog box lets you enter an expression that is evaluated each time the breakpoint is encountered during the program execution. If the expression evaluates to true (or not zero), the breakpoint pauses the program run. If the condition evaluates to false (or zero), the debugger does not stop at the breakpoint location.

Conditional breakpoints are useful when you want to see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

For example, suppose you want a breakpoint to pause on a line of code only when the variable *mediumCount* is greater than 10. To do so,

1. Place the insertion point on the line of code you want in the Code editor and press F5 to set the breakpoint.
2. Choose View|Breakpoints to open the Breakpoint List window.
3. In the Breakpoint List window, highlight the breakpoint you just created, then right-click and choose Edit Breakpoint.
4. On the Edit Breakpoint dialog box, enter the following expression into the Condition edit box:

   ```
   mediumCount > 10
   ```

5. Click Modify to confirm your settings.

▪ You can input any valid language expression into the Condition edit box, but all symbols in the expression must be accessible (within scope) from the breakpoint's location, and the expression cannot contain any function calls.

Using pass counts

See also

The Pass Count edit box enables you to specify a particular number of times that a breakpoint must be passed for the breakpoint to be activated. A pass count tells the debugger to pause program execution the *n*th time that the breakpoint is encountered during the program run (you supply the number *n* which is set to 1 by default).

The pass count number decrements each time the line containing the breakpoint is encountered during the program execution. If the pass count equals 1 when the breakpoint line is encountered, program execution pauses on that line of code. For example, if you enter `2,` your program does not stop until the second time the debugger reaches the line where the breakpoint is set.

When you use a pass count in conjunction with a Boolean condition, the breakpoint pauses program execution the *n*th time that the condition is true; the condition must be true for the pass count to decrement. For example, if you enter the expression `x>3` in Conditions and the number `2` in Pass Count, your program does not stop until the second time the debugger reaches the breakpoint when the value of *x* is greater than 3.

Locating breakpoints

See also

If a breakpoint is not visible in the Code editor or in the CPU view, you can use the Breakpoint List window to quickly locate the breakpoint.

To scroll the Code editor to the location of a breakpoint in your source code,

- Right-click the breakpoint in the Breakpoint List window and choose View Source.

To scroll the Code editor to the location of a breakpoint in your source code and make the Code editor active,

- Right-click the breakpoint in the Breakpoint List window and choose Edit Source.

If you choose View Source, the Breakpoint List window remains active so you can modify the breakpoint or go on to view another. If you choose Edit Source, the Code editor gains focus so you can modify the source code at that location.

Disabling and enabling breakpoints

See also

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

To disable a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Disable.

To disable all breakpoints in a source code file,

- Right-click the Breakpoint List window and choose Disable All.

To enable a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Enabled.

To enable all breakpoints in a source code file,

- Right-click the Breakpoint List window and choose Enable All.

Deleting breakpoints

See also

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. You can delete breakpoints using either the Code editor or the Breakpoints window:

To delete a single breakpoint,

- Right-click the breakpoint in the Breakpoint List window and choose Delete.
- Right-click the breakpoint in the Code editor or CPU window and choose Toggle breakpoint
- Place the insertion point anywhere in the line in the Code editor containing the breakpoint or highlight the breakpoint in the CPU window and press F5.
- Click the stop-sign glyph at the left end of the line containing the breakpoint in the Code editor.

To delete all breakpoints in a source code file,

- Right-click the Breakpoint List window and choose Delete all.

Examining program data values

See also

After you have paused your application using the integrated debugger, you can examine the different symbols and data structures with regard to the location of the current execution point. You frequently need to examine the values of variables and expressions to uncover bugs in your program. For example, it is helpful to know the value of the index variable as you step though a **for** loop, or the values of the parameters passed to a function call.

Data evaluation operates at the level of expressions. An expression consists of constants, variables, and values contained in data structures, combined with language operators.

- Almost anything you can use as the right side of an assignment operator can be used as a debugging expression, except for variables not accessible from the current execution point.

You can view the state of your program by

- Watching program values
- Evaluating and modifying expressions
- Inspecting data elements
- Viewing the low-level state of your program
- Viewing functions in the Call Stack window

Modifying program data values

See also

Sometimes you will find that a programming error is caused by an incorrect data value. Using the integrated debugger, you can test a "fix" by modifying the data value while your program is running. You can modify program data in the following ways:

- Modifying variables
- Changing the value of inspector items
- Using the CPU window's Memory Dump pane

Watch expressions

See also

If you want to monitor the value of a variable or expression while you debug your code, add a watch to the Watch List window. The Watch List window displays the current value of the watch expression based on the scope of the execution point.

Each time your program's execution pauses, the debugger evaluates all the items listed in the Watch List window and updates their displayed values.

You can set a watch expression in the following ways:

- The easiest way to set a watch is to place the insertion point on a term in the Code editor, then right-click and choose Add Watch at Cursor.
- You can also set a watch and specify its properties on the Watch Properties dialog box. For more information, see Setting watch properties.

The Watch List window

After you enter a watch expression, use the Watch List window to display the current value of the expression. Check the check box beside a watch to enable it, or clear the check box to disable it.

To display the Watch List window, choose View|Watch.

The left side of the Watch List window shows the expressions entered as watches. Corresponding data types and values appear on the right. Values of compound data objects (such as arrays and structures) appear between braces ({ }).

- The Watch List window will be blank if you have not added any watches.

If the execution point moves to a location where any of the variables in an expression is undefined (out of scope), the entire watch expression becomes undefined. If the execution point returns to a location where the watch expression can be evaluated (that is, if the execution point re-enters the scope of the expression), the Watch List window again displays the current value of the expression.

Watch List commands

Right-click the Watch List to access the following commands that enable you manipulate watch points:

Edit Watch	Opens the Watch Properties dialog box that lets you modify the properties of a watch
Add Watch	Opens the Watch Properties dialog box that lets you create a watch
Enable Watch	Enables a disabled watch expression
Disable Watch	Disables an enabled watch expression
Delete Watch	Removes a watch expression
Enable All Watches	Enables all disabled watch expressions
Disable All Watches	Disables all enabled watch expressions
Delete All Watches	Removes all watch expressions

Edit Watch

Right-click the Watch List and choose Edit Watch to open the Watch Properties dialog box, where you can create and modify a watch. After you create a watch, use the Watch List window to modify the watches currently set in your program.

Add Watch

Right-click the Watch List and choose Add Watch to open the Watch Properties dialog box, where you can create a watch. After you create a watch, use the Watch List window to modify the watches currently set in your program.

Alternate ways to perform this command are

- Choose Run|Add Watch.
- Right-click the Code editor and choose Add Watch At Cursor.

Enable Watch
Right-click the Watch List and choose Enable Watch to enable a disabled watch expression.

Disable Watch

Right-click the Watch List and choose Disable Watch to disable an enabled watch expression.

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but the debugger does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

Delete Watch
Right-click the Watch List and choose Delete Watch to remove a watch expression.

When you no longer need to examine the value of an expression, you can delete the watch from the debugging session. This command is not reversible.

Enable All Watches
Right-click the Watch List and choose Enable All Watches to enable all disabled watch expressions.

Disable All Watches

Right-click the Watch List and choose Disable All Watches to disable all enabled watch expressions.

Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but the debugger does not evaluate the watch.

Disabling watches improves performance of the debugger because it does not monitor the watch as you step through or run your program. When you set a watch, it is enabled by default.

Delete All Watches

Right-click the Watch List and choose Delete All Watches to remove all watch expressions.

When you no longer need to examine the value of an expression, you can delete the watch from the debugging session.

Setting watch properties

See also

Use the Watch properties dialog box to set the properties of a new watch expression or to change the properties of an existing one.

You can open the Watch Properties dialog box in the following ways:

- Choose Run|Add Watch from the main menu.
- Right-click the Watch List window and choose Add Watch.
- Select a watch in the Watch List window, then right-click and choose Edit Watch.

Watch Properties dialog box

You can set the following properties for a watch expression:

Expression

Specifies the expression to watch. Enter or edit the expression you want to watch. Use the drop-down button to choose from a history of previously selected expressions.

Repeat Count

Specifies the repeat count when the watch expression represents a data element, or specifies the number of elements in an array when the watch expression represents an array.

When you watch an array and specify the number of elements as a repeat count, the Watch List window displays the value of every element in the array.

Digits

Specifies the number of significant digits in a watch value that is a floating-point expression. Enter the number of digits.

- This option takes affect only when you select Floating Point as the Display format. For more information, see Formatting watch expressions.

Enabled

Enables or disables the watch. Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but the debugger does not evaluate the watch.

Display format radio buttons

To format the display of a watch expression, select a radio button.

- For more information, see Formatting watch expressions.

Formatting watch expressions

See also

By default, the debugger displays the result of a watch in the format that matches the data type of the expression. For example, by default, integer values are displayed in decimal form. If you select Hexadecimal in the Watch Properties dialog box for an integer type expression, the debugger changes the display format from decimal to hexadecimal.

If you are setting up a watch on an element in a data structure (such as an array), you can display the values of consecutive data elements. For example, suppose you have an array of five integers named *xarray*. Type the number `5` in Repeat Count on the Watch Properties dialog box to see all five values of the array. To use a repeat count, however, the watch expression must represent a single data element.

To format a floating-point expression, select Decimal at Display format and enter a number for Digits on the Watch Properties dialog to indicate the number of significant digits you want displayed in the Watch List window.

The following table describes the watch expression format options and their effects.

Option	Types affected	Description
Hexadecimal	integers/characters	Shows integer values in hexadecimal with the 0x prefix, including those in data structures.
Character	characters/strings	Shows special display characters for ASCII 0 to 31. By default, such characters are shown using the appropriate C escape sequences (\n, \t, and so forth).
Decimal	integers	Shows integer values in decimal form, including those in data structures.
Floating point	floating point	Shows the significant digits specified; from 2-18. The default is 7.
Memory dump	all	Shows the size in bytes starting at the address of the indicated expression. By default, each byte displays two hex digits. Use the memory dump with the character, decimal, hexadecimal, and string options to change the byte formatting. Use the Repeat Count setting to specify the number of bytes you want to display.
Pointer	pointers	Shows the address of the pointer.
Structure/Union	structures /unions	Shows field names and unions as well as values such as X:1;Y:10;Z:5.
String	char, strings	Shows ASCII 0 to 31 as C escape sequences. Use this option only to modify memory dumps.
Default	all	Shows the result in the display format that matches the data type of the expression.

Enabling and disabling watches

See also

Evaluating many watch expressions can slow down the process of debugging. Disable a watch expression when you prefer not to view it in the Watch List window, but want to save it for later use.

When you set a watch, it is enabled by default. Disabling a watch hides the watch from the current program run. When you disable a watch, its settings remain defined, but the debugger does not evaluate it.

To enable or disable a watch,

1. Choose View|Watch to open the Watch List window.
2. Select a watch, then right-click and choose Enable or Disable watch.

The flag <disabled> appears next to a watch that is disabled.

To disable or enable all watches,

Right-click the Watch List window and choose Enable All Watches or Disable All Watches.

Deleting watches

See also

When you no longer need to examine the value of an expression, you can delete the watch from the debugging session.

To delete a single watch,

1. Choose View|Watch to open the Watch List window.
2. Select a watch, then right-click and choose Delete Watch.

To delete all watches in a source code file,

- Right-click the Watch List window and choose Delete All Watches.

Evaluating and modifying expressions

See also

Use the Evaluate/Modify dialog box to evaluate or change the value of an existing expression or property. The Evaluate/Modify dialog box has the advantage over watches in that it enables you to change the values of variables and items in data structures during the course of your debugging session.

You can test different error hypotheses and see how a section of code behaves under different circumstances by modifying the value of data items during a debugging session. This technique can be useful if you think you have found the solution to a bug, and you want to try the correction without having to exit the debugger, changing the source code, and recompiling the program.

To evaluate an expression or property

1. Open the Evaluate/Modify dialog box one of the following ways.

- Choose Run|Evaluate/Modify
- Right-click the Code editor and choose Evaluate/Modify.

2. Type an expression in the Expression box.

   By default, the word at the cursor position in the current Code editor is placed in the Expression input box. You can accept this expression, enter another one, or choose an expression from the history list of expressions you have previously evaluated. If you want to evaluate a function call, enter the function name, parentheses, and arguments just as you would type it into your program, but leave out the statement-ending semicolon (;).

3. Choose Evaluate. The value of the item appears in the Result edit box.

Evaluating expressions

See also

You can evaluate any valid language expression, except those that contain variables that are not accessible from the current execution point.

Formatting values

To format the result that displays, add a comma and one or more format specifiers to the end of the expression entered in the Expression box. For example:

- To display a result in hexadecimal, type `,H` after the expression.
- To see a floating point number to 3 decimal places, type `,F3` after the expression.

For a complete list of format options, see Evaluate/modify format specifiers.

Evaluate/Modify dialog box

The Evaluate/Modify dialog box provides the following options:

Expression

Lets you specify the variable, array, or object to evaluate or modify.

Result

Displays the value of the item specified in the Expression text box after you choose Evaluate or Modify.

New value

Lets you assign a new value to the item specified in the Expression edit box.

Evaluate

Evaluates the expression in the Expression edit box and displays its value in the Result edit box.

Modify

Changes the value of the expression in the Expression edit box using the value in the New Value edit box.

Modifying variables

After you have evaluated a variable or data structure item, you can modify its value. When you modify a value through the debugger, the modification is effective for that specific program run only. Changes you make through the Evaluate/Modify dialog box do not affect your source code or the compiled program. To make your change permanent, you must modify your source code in the Code editor, then recompile your program.

To change the value of an expression

1. Open the Evaluate/Modify dialog box one of the following ways.

- Choose Run|Evaluate/Modify
- Right-click the Code editor and choose Evaluate/Modify.

2. Specify the expression in the Expression edit box. To modify a component property, explicitly specify the property name. For example, enter: `Button1->Caption`
3. Enter a value in the New Value edit box.
4. Choose Modify. The new value is displayed in the Result box.

- You cannot undo a change to a variable after you choose Modify. To restore a value, however, you can enter the previous value in the Expression box and modify the expression again.

Keep these points in mind when you modify program data values:

- You can change individual variables or elements of arrays and data structures, but you cannot change the contents of an entire array or data structure with a single expression.
- The expression in the New Value box must evaluate to a result that is assignment-compatible with the variable you want to assign it to. A good rule of thumb is that if the assignment would cause a compile-time or runtime error, it is not a legal modification value.
- Use caution when you modify variables while debugging an application – any side effects that occur will modify the data values of the program you are debugging. For example, if you modify a function that increments a variable, the new value of that variable will be reflected when you continue to step through your application. Modifying values (especially pointer values and array indexes) can have undesirable effects because you might overwrite other variables and data structures. Because these errors might not be immediately apparent, use caution whenever you modify program values from the debugger.

Inspecting data elements

See also

Inspector windows are the best way to view data items because the debugger automatically formats Inspector windows according to the type of data it is displaying. Inspector windows are especially useful when you want to examine compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in an Inspector window, you can “walk” through compound data objects by opening an Inspector window on a component of the compound object.

To display an Inspector window directly from the Code editor,

1. Place the insertion point in the Code editor on the data element you want to inspect.
2. Right-click and choose Inspect.

To inspect a data element from the menu bar,

1. Choose Run|Inspect from the menu bar to display the Inspect dialog box.
2. Type the expression you want to inspect, then choose OK.

Scope

Unlike watch expressions, the scope of a data element in an Inspector window is fixed at the time you evaluate it:

- If you use the Inspect command from the Code editor, the debugger uses the location of the insertion point to determine the scope of the expression you are inspecting. This makes it possible to inspect data elements that are not within the current scope of the execution point.
- If you use the Run|Inspect command from the menu bar, the data element is evaluated within the scope of the execution point.

If the execution point is in the scope of the expression you are inspecting, the value appears in the Inspector window. If the execution point is outside the scope of the expression, the value is undefined and the Inspector window becomes blank.

Inspecting local variables

You can inspect all the local variables defined in the current scope by inspecting the expression `$Locals`.

1. Choose Run|Inspect from the menu bar to display the Inspect Expression dialog box.
2. Type `$Locals` and choose OK.

Data types

The number of panes and the appearance of the data in the Inspector window depends on which of the following types of data you inspect:

- scalar variables
- functions
- constants
- arrays
- pointers
- classes
- objects
- structures and unions

For example, if you inspect an array, you will see a line for each member of the array with the array index of the member. The value of the member follows in its display format, followed by the value in hexadecimal.

The Inspector window

See also

The number of tabs and the appearance of the data in the Inspector window depends on the type of data you inspect. You can inspect the following types of data: arrays, classes, constants, functions, pointers, scalar variables (**int**, **float**, and so on), and structures and unions. The Inspector window contains three areas:

- The top of the Inspector window shows the name, type, and address or memory location of the inspected element, if available.
- The middle pane contains one or more of the following views depending on the type of data you inspect. To change the view, click its tab.

Data	Shows data names (or class data members) and current values.
Methods	This view appears only when you inspect a class, structure, or union and shows the class methods (member functions) and current address locations.
Properties	This view displays only when you inspect an Object class with properties (such as a VCL Object) and shows its property names and current values.

- The bottom of the Inspector window shows the data type of the item currently selected in the middle pane.

Inspector window commands

Right-click the Inspector window to access the following commands:

Range	Lets you specify how many data elements you want to view. It is available only when you inspect pointers and arrays.
Change	Lets you assign a new value to a data item. An ellipsis (…) appears next to an item that can be changed. You can click the ellipsis as alternative to choosing the change command.
Show Inherited	Switches the view in the Data, Methods, and Properties panes between two modes: one that shows all intrinsic and inherited data members or properties of a class, or one that shows only those declared in the class.
Inspect	Opens a new Inspector window on the data element you have selected. This is useful for seeing the details of data structures, classes, and arrays.
Descend	Same as the Inspect command, except the current Inspector window is replaced with the details that you are inspecting (a new Inspector window is not opened). To return to a higher level, use the history list.
Type Cast	Lets you specify a different data type for an item you want to inspect. Type casting is useful if the Inspector window contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers.
New Expression	Lets you inspect a new expression.

Inspecting scalar variables

See also

When you inspect a scalar variable, such as simple data items including C and C++ **char**, **int**, **long**, and Object Pascal **Integer**, **Real**, and so on, the top of the Inspector window shows the name, type, and address of the variable. The middle pane shows the name of the scalar on the left and its current value on the right. Integer values are displayed first in decimal, followed by the hexadecimal value enclosed in parentheses.

If the variable inspected is of C++ type **char**, the equivalent character appears to the left of the numeric values. If the present value does not have a printable character equivalent, the debugger displays a backslash (\) followed by the hexadecimal value that represents the character value.

Inspecting pointers and arrays

See also

When you inspect a pointer or an array, Inspector windows show the values of variables that point to other data items. The top of the Inspector window shows the name, type, count, address (or register if applicable), and pointer location of the variable. The middle pane shows the current values of the data pointed to. The bottom of the Inspector window shows the data type to which the pointer points.

If the value pointed to is a compound data object (such as a structure or record, or an array), the values are enclosed in braces ({}) and the Inspector window displays as much of the data as possible.

If the pointer appears to be pointing to a null-terminated character string, the debugger displays the value of each item in the character array. The left of each line displays the array index ([0], [1], [2], and so on), and the values appear on the right. When you inspect character strings, the entire string appears at the top of Inspector window, along with the address of the pointer variable and the address of the string that it points to.

Range

You can use the Range command to cause the Inspector window to display multiple lines of information. This is helpful for C++ programmers who use pointers to point to arrays of data structures as well as to single items. For example, suppose you have the following code:

```
// In C++
int array[10];
int *arrayp = array;


{ In Object Pascal }
Type
Tarray = array[0..9] of Integer;
arrayp = ^Tarray;
```

To see what *arrayp* points to;

1. Select *arrayp* in the Inspector window.
2. Right-click and choose Range.
3. Specify a start of 0 and a count of 5. Had you not done so, you would have seen only one item in the array.

Inspecting C++ structures and unions

See also

When you inspect a structure or union, the Inspector window shows the values of members contained in compound data objects.

The top of the window shows the name of the object. The middle pane lists the names and values of the data members of contained in the object, and contains as many lines as needed to show the entire data object.

The bottom of the window shows the data type of the member currently selected.

Inspecting functions

See also

When you inspect a function, the top of the Inspector window shows the function name, prototype, and its address in memory. The middle pane shows the function's arguments. To inspect a function, enter the function’s name without parentheses or arguments.

If the function is currently on the call stack, its parameters appear at the bottom of the inspector window.

Isolating the view in an Inspector window

See also

You can more closely inspect certain elements (such as classes, structures, and arrays) in the Inspector window to isolate the view to the member level:

1. Select an item in the Inspector window.
2. Right-click and choose Inspect to open a new Inspector window, or choose Descend to update the display of the current Inspector window.

The scope of the data element remains the same as it was when you opened it on the Inspector window. If you select a data member that is a pointer to a class, the Inspector window displays the class pointed to.

Changing the value of Inspector items

See also

An ellipsis (…) appears next to a data element that can modified.

To change the value of a inspected element,

1. Select an item in the Inspector window.
2. Click the ellipsis (…), or right-click the element and choose Change.
3. Type a new value, then choose OK.

Selecting a range of data items

See also

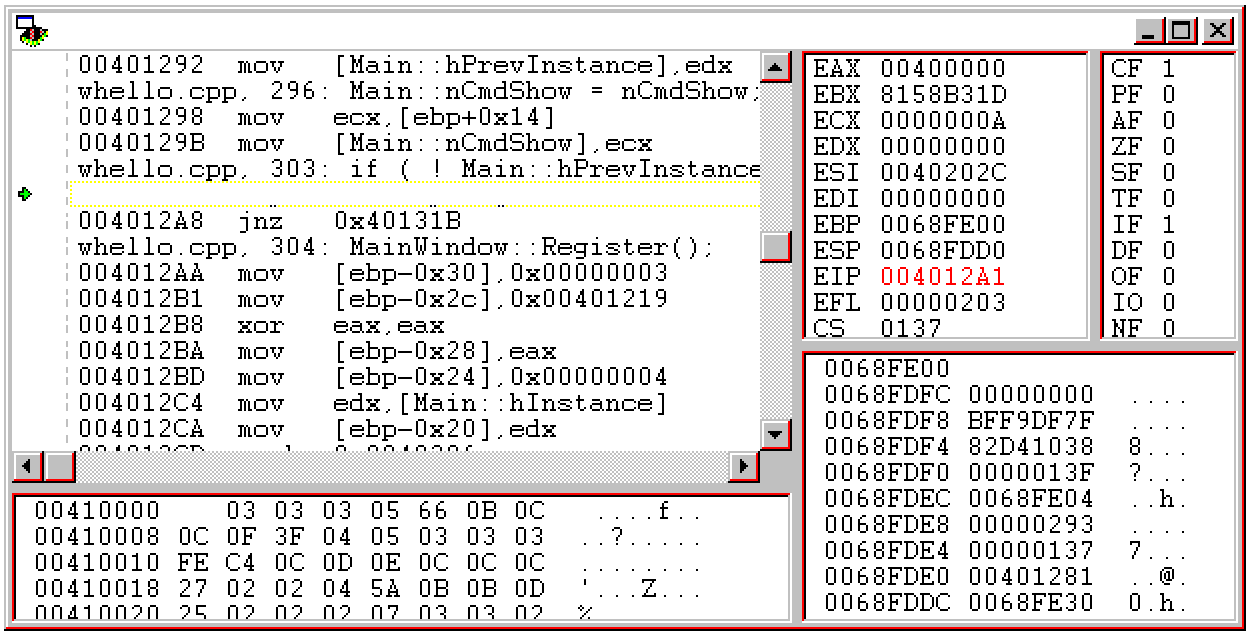
If you are inspecting an array, it is possible the number of items displayed might be so great that you will have to scroll the Inspector window to see data you want. For easier viewing, narrow the display to a range of data items.

1. Select an item in the Inspector window.
2. Right-click and choose Range.
3. In the Start box, enter a new value for the first element in the array.
4. In the Count box, enter the number of elements in the array you want to view.

The CPU window

See also

The CPU window consists of five separate panes. Each pane gives you a view into a specific low-level aspect of your running application.



```
00401292  mov   [Main::hPrevInstance],edx
whello.cpp, 296: Main::nCmdShow = nCmdShow;
00401298  mov   ecx,[ebp+0x14]
0040129B  mov   [Main::nCmdShow],ecx
whello.cpp, 303: if ( ! Main::hPrevInstance

004012A8  jnz   0x40131B
whello.cpp, 304: MainWindow::Register();
004012AA  mov   [ebp-0x30],0x00000003
004012B1  mov   [ebp-0x2c],0x00401219
004012B8  xor   eax,eax
004012BA  mov   [ebp-0x28],eax
004012BD  mov   [ebp-0x24],0x00000004
004012C4  mov   edx,[Main::hInstance]
004012CA  mov   [ebp-0x20],edx
```

```
EAX 00400000    CF 1
EBX 8158B31D    PF 0
ECX 0000000A    AF 0
EDX 00000000    ZF 0
ESI 0040202C    SF 0
EDI 00000000    TF 0
EBP 0068FE00    IF 1
ESP 0068FDD0    DF 0
EIP 004012A1    OF 0
EFL 00000203    IO 0
CS  0137        NF 0
```

```
00410000    03 03 03 05 66 0B 0C    ....f..
00410008 0C 0F 3F 04 05 03 03 03   ..?.....
00410010 FE C4 0C 0D 0E 0C 0C 0C   ........
00410018 27 02 02 04 5A 0B 0B 0D   '...Z...
```

```
0068FE00
0068FDFC 00000000  ....
0068FDF8 BFF9DF7F  ....
0068FDF4 82D41038  8...
0068FDF0 0000013F  ?...
0068FDEC 0068FE04  ..h.
0068FDE8 00000293  ....
0068FDE4 00000137  7...
0068FDE0 00401281  ..@.
0068FDDC 0068FE30  0.h.
```

- The Disassembly pane displays the assembly instructions that have been disassembled from your application's machine code. In addition, the Disassembly pane displays the original program source code above the associated assembly instructions.
- The Memory Dump pane displays a memory dump of any memory accessible to the currently loaded executable module. By default, memory is displayed as hexadecimal bytes.
- The Machine Stack pane displays the current contents of the program stack. By default, the stack is displayed as hexadecimal longs (32-bit values).
- The Registers pane displays the current values of the CPU registers.
- The Flags pane displays the current values of the CPU flags.

Right-click anywhere on the CPU window to access commands specific to the contents of the current pane.

Opening the CPU window

To open the CPU window anytime during a debugging session,

Choose View|CPU or right-click the Code editor and choose CPU View to open the Disassembly pane at the location of the execution point.

- The CPU window opens automatically whenever program execution stops at a location for which source code is unavailable. For example, the debugger cannot open the source file if you link a DLL built with debug information but do not include its source file in your project, or if you place the source file in a directory not specified in your project.

Resizing the CPU window panes

You can customize the layout of the CPU window by resizing the panes within the window. Drag the pane boarders within the window to enlarge or shrink the windows to your liking.

Disassembly pane

The left side of the Disassembly pane lists the address of each disassembled instruction. A green arrow to the left of the memory address indicates the location of the current execution point. To the right of the memory addresses, the Disassembly pane displays the assembly instructions that have been disassembled from the machine code produced by the compiler. If you make the Disassembly pane wide enough, the debugger displays the instruction opcodes following the listing of the instruction memory addresses.

When you click an address in the Disassembly pane,

- the upper left corner shows the effective address (when available) and the value it stores. For example, if you select an address containing an expression in brackets such as `[eax+edi*4-0x0F]`, the top of the Disassembly pane shows the location in memory being referenced and its current value.
- the upper right corner shows the current thread ID.

If you are viewing code that has been linked with a symbol table, the debugger displays the source code that is associated with the disassembled instructions.

- Press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte. Beware that changing the starting point of the display in the Disassembly pane changes where the debugger begins disassembling the machine code.

Disassembly pane commands

Right-click the Disassembly pane to access the following commands:

- Run to Current
- Toggle Breakpoint
- Go to Address
- Go to Current EIP
- Follow
- Caller
- Previous
- Search
- View Source
- Mixed
- Change Thread
- New EIP

Run to Current

The Run To Current command lets you run your program at full speed to the instruction that you have selected in the Disassembly pane. After your program is paused, you can use this command to resume debugging at a specific program instruction.

Toggle Breakpoint

This command adds or removes a breakpoint at the selected instruction in the Disassembly pane. When you choose Toggle Breakpoint, the debugger sets an unconditional (simple), breakpoint at the instruction that you have selected in the Disassembly pane. A simple breakpoint has no conditions, and the only action is that it will pause the program's execution.

If a breakpoint exists on the selected instruction, then Toggle Breakpoint will delete the breakpoint at that code location.

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Disassembly pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with 0x.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte.

Go to Current EIP

This command positions the Disassembly pane at the location of the current program counter (the location indicated by the EIP register). This location indicates the next instruction to be executed by your program.

This command is useful when you have navigated through the Disassembly pane, and you want to return to the next instruction to be executed.

Follow

This command positions the Disassembly pane at the destination address of the instruction currently highlighted.

Use the Follow command in conjunction with instructions that cause a transfer of control (such as **CALL**, **JMP**, and **INT**) and with conditional jump instructions (such as **JZ**, **JNE**, **LOOP**, and so forth). For conditional jumps, the address is shown as if the jump condition is TRUE. Use the Previous command to return to the origin of the jump.

Caller

This command positions the Disassembly pane at the instruction that called the current interrupt or subroutine.

- If the current interrupt routine has pushed data items onto the stack, the debugger might not be able to determine where the routine was called from.

Previous

This command restores the Disassembly pane to the display it had before you issued the last Follow command.

Search

This command searches forward in the Disassembly pane for an expression or byte list that you supply. Supply a byte list to search for two or more values located in a specific order. Be sure to precede hexadecimal values with 0x.

For example, if you enter

```
0x5D 0xC3
```

the debugger goes to the following location:

```
004013AB 5D
004013AC C3
```

You can also search for DWords, but you must reverse the order of the bytes.

For example, if you enter

```
0x1234
```

the debugger positions the pane at the following location in memory:

```
34 12
```

View source

This command activates the Code editor and positions the insertion point at the source code line that most closely corresponds to the disassembled instruction selected in the Disassembly pane. If there is no corresponding source code (for example, if you are examining Windows kernel code), this command has no effect.

Mixed

Switches the display format of the Disassembly pane:

When Mixed is….	The Disassembly pane displays….
checked	source code lines before the first disassembled instruction relating to that source line.
unchecked	disassembled instructions without source code.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Disassembly pane, all panes in the CPU window reflect the state of the CPU for that thread.

New EIP

This command changes the location of the instruction pointer (the value of EIP register) to the line currently highlighted in the Disassembly pane. Use this command when you want to skip certain machine instructions. When you resume program execution, execution starts at this address.

- This command is not the same as stepping through instructions; the debugger does not execute any instructions that you might skip.
- Use this command with extreme care; it is easy to place your system in an unstable state when you skip over program instructions.

Memory Dump pane

The Memory Dump pane displays the raw values contained in addressable areas of your program. The pane has three sections: the memory addresses, the current values in memory, and an ASCII representation of the values in memory.

The Memory Dump pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is an 8-byte hexadecimal listing of the values contained at that location in memory. Each byte in memory is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period.

The format of the memory display depends on the format selected with the Display As command. If you choose one of the floating-point display formats (Floats or Doubles), a single floating-point number is displayed on each line. The Bytes format (default) displays 8 bytes per line, Words displays 4 words per line, and DWords displays 2 long words per line.

- You can press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

Memory Dump pane commands

Right-click the Memory Dump pane to access the following commands:

- Go to Address
- Change Thread
- Search
- Next
- Change
- Follow
- Previous
- Display As

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Memory Dump pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with 0x.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Memory Dump pane, all panes in the CPU window reflect the state of the CPU for that thread.

Search

This command searches forward in the Memory Dump pane for an expression or byte list that you supply. Supply a byte list to search for two or more values located in a specific order. Be sure to precede hexadecimal values with 0x.

For example, if you enter

```
0x5D 0xC3
```

the debugger positions the pane at the following location:

```
004013AB 5D
004013AC C3
```

You can also search for DWords, but you must reverse the order of the bytes.

For example, if you enter

```
0x1234
```

the debugger positions the pane at the following location in memory:

```
34 12
```

Next
Finds the next occurrence of the item you last Searched for in the Memory Dump pane.

Change
Lets you modify the bytes located at the current cursor location and prompts you for an item of the current display type.

- You can invoke this command by typing directly in the Dump pane.

Follow

Lets you choose the following commands:

Near Code	Positions the Disassembly pane at the address currently selected in the Memory Dump pane.
Offset to Data	Lets you follow DWord-pointer chains (near and offset only) and positions the Memory Dump pane at the address specified by the DWord currently highlighted.

Previous

This command restores the Memory Dump pane of the CPU window to the location displayed before you issued the last Follow command.

Display as

Use the Display As command to format the data listed in the Memory Dump pane of the CPU window. You can choose any of the data formats listed in the following table:

Data type	Display format
Bytes	Hexadecimal bytes
Words	2-byte hexadecimal numbers
DWords	4-byte hexadecimal numbers
Floats	4-byte floating-point numbers using scientific notation
Doubles	8-byte floating-point numbers using scientific notation
Long Doubles	10-byte floating-point numbers using scientific notation

Machine Stack pane

The Machine Stack pane displays the raw values contained in the your program stack. The pane has three sections: the memory addresses, the current values on the stack, and an ASCII representation of the stack values.

➜ A green arrow indicates the value at the top of the call stack.

The Machine Stack pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is a 4-byte listing of the values contained at that memory location. Each byte is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period.

The format of the memory display depends on the format selected with the Display As command. If you choose one of the floating-point display formats (Floats or Doubles), a single floating-point number is displayed on each line. The Bytes format displays 4 bytes per line, Words displays 2 words per line, and DWords (the default) displays 1 long word per line.

- You can press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

Machine Stack pane commands

Right-click the Machine Stack pane to access the following commands:

- Go to Address
- Change Thread
- Top of Stack
- Follow
- Previous
- Change
- Display As

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Machine Stack pane of the CPU window. Enter any expression that evaluates to a program memory location. Be sure to precede hexadecimal values with 0x.

- The debugger displays dashes if you view a program memory location in which nothing is loaded.

You can also press Ctrl+Left Arrow and Ctrl+Right Arrow to shift the starting point of the display up or down one byte.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Machine Stack pane, all panes in the CPU window reflect the state of the CPU for that thread.

Top of stack
Positions the Machine Stack pane at the address of the stack pointer (the address held in the ESP register).

Follow

Lets you choose the following commands:

Offset to stack	Lets you follow DWord-pointer chains (near and offset only) on the call stack and positions the Machine Stack pane at the address location of the value currently selected in the Machine Stack pane.
Near Code	Positions the Disassembly pane at the address location of the value currently selected in the Machine Stack pane.
Offset to Data	Lets you follow DWord-pointer chains (near and offset only) and position the Memory Dump pane at the address location of the value currently selected in the Machine Stack pane.

Previous
This command restores the Machine Stack pane in CPU window to the location displayed before you issued the last Follow command.

Change

Lets you enter a new value for the stack word currently highlighted.

- You can invoke this command by typing directly in the Machine Stack pane.

Display as

Use the Display As command to format the data that’s listed in the Machine Stack pane of the CPU window. You can choose any of the data formats listed in the following table:

Data type	Display format
Bytes	Displays data in hexadecimal bytes
Words	Displays data in 2-byte hexadecimal numbers
DWords	Displays data in 4-byte hexadecimal numbers
Floats	Displays data in 4-byte floating-point numbers using scientific notation

Registers pane

The Registers pane displays the contents of the CPU registers of the 80386 and greater processors. These registers consist of eight 32-bit general purpose registers, six 16-bit segment registers, the 32-bit program counter (EIP), and the 32-bit flags register (EFL).

After you execute an instruction, the Registers pane highlights in red any registers that have changed value since the program was last paused.

Registers pane commands

Right-click the Registers pane to access the following commands:

- Increment Register
- Decrement Register
- Zero Register
- Change Register
- Change Thread

Increment register

Increment Register adds 1 to the value in the currently highlighted register. This option lets you test “off-by-one” bugs by making small adjustments to the register values.

Decrement register

Decrement Register subtracts 1 from the value in the currently highlighted register. This option lets you test “off-by-one” bugs by making small adjustments to the register values.

Zero register
The Zero Register command sets the value of the currently highlighted register to 0.

Change register
Lets you change the value of the currently highlighted register. This command opens the Change Register dialog box where you enter a new value. You can make full use of the expression evaluator to enter new values. Be sure to precede hexadecimal values with 0x.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Registers pane, all panes in the CPU window reflect the state of the CPU for that thread.

Flags pane

The Flags pane shows the current state of the flags and information bits contained in the 32-bit register EFL. After you execute an instruction, the Flags pane highlights in red any flags that have changed value since the program was last paused.

The processor uses the following 15 bits in this register to control certain operations and indicate the state of the processor after it executes certain instructions:

Letters in pane	Flag/bit name	EFL register bit number
CF	Carry flag	0
PF	Parity flag	2
AF	Auxiliary carry flag	4
ZF	Zero flag	6
SF	Sign flag	7
TF	Trap flag	8
IF	Interrupt flag	9
DF	Direction flag	10
OF	Overflow flag	11
IO	I/O privilege level	12 and 13
NF	Nested task flag	14
RF	Resume flag	16
VM	Virtual 8086 mode	17
AC	Alignment check	18

Flags pane commands

Right-click the Flags pane to access the following commands:

- Toggle Flag
- Change Thread

Toggle flag
The flag and information bits in the Flags pane can each hold a binary value of 0 or 1. This command toggles the selected flag or bit between these two binary values.

Change thread

This command opens the Select a Thread dialog box. Select the thread you want to debug from the threads listed. When you choose a new thread from the Flags pane, all panes in the CPU window reflect the state of the CPU for that thread.

Locating function calls

See also

While debugging, it is useful to know the order of function calls that brought you to your current location. The Call Stack window lets you view the current sequence of function calls. It also shows the values of the arguments passed to each function call (the arguments with which the call was made).

To open the Call Stack window,

- Choose View|Call Stack from the menu bar.

To scroll the Code editor to the location of a function call,

- Right-click the function call in the Call Stack window and choose View Source.

To scroll the Code editor to the location of a function call and make the Code editor active,

- Right-click the function call in the Call Stack window and choose Edit Source.

If you choose View Source, the Call Stack window remains active. If you choose Edit Source, the Code editor gains focus, enabling you to modify the source code at that location.

Stepping over function calls

The Call Stack window is useful if you accidentally trace into code you wanted to step over. Using the Call Stack window, you can return to the point from which the current function was called, then resume debugging.

To use the Call Stack window to step over function calls,

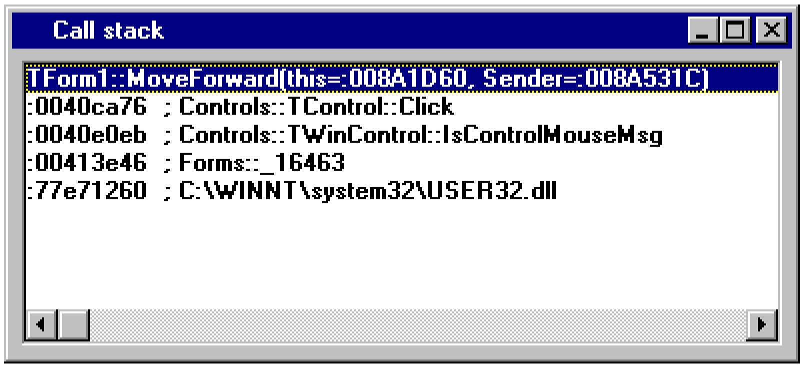
1. In the Call Stack window, right-click the calling function (the second function in the Call Stack window) and choose Edit Source. The Code editor becomes active with the cursor positioned at the location of the function call.
2. In the Code editor, move the cursor to the statement following the function call.
3. Choose Run|Run to Cursor.

The Call Stack window

See also

The Call Stack window displays the function calls that brought you to your current program location and the arguments passed to each function call.

The following illustration shows a typical Call Stack window:



```
TForm1::MoveForward(this=:008A1D60, Sender=:008A531C)
:0040ca76  ; Controls::TControl::Click
:0040e0eb  ; Controls::TWinControl::IsControlMouseMsg
:00413e46  ; Forms::_16463
:77e71260  ; C:\WINNT\system32\USER32.dll
```

The top of the Call Stack window lists the last function called by your program. Below this is the listing for the previously called function. The listing continues, with the first function called in your program located at the bottom of the list. Each function listing in the window is followed by the arguments that were passed when the call was made.

The Call Stack window also shows the names of member functions (or methods). Each class function is prefixed with the name of the class that defines the function.

Call Stack commands

Right-click the Call Stack to access the following commands:

View Source	Locates a function call in your source code quickly.
Edit Source	Locates a function call in your source code quickly, and gives focus to the Code editor.

Customizing the colors of the execution point and breakpoints

See also

You can customize the colors used to indicate the execution point and the enabled, disabled, and invalid breakpoint lines.

To set execution point and breakpoint colors,

1. Choose Options|Environment.
2. On the Environment Options dialog box, select the Colors tab.
3. From the Element list, select the following options that you want to change:

   Execution point

   Enabled Break

   Disabled Break

   Invalid Break
4. Select the background (BG) and foreground (FG) colors you want.

Logging debug messages

You can keep track of the significant events that occur during your debugging session using the Windows API function *OutputDebugString*. When a program you are debugging executes calls to *OutputDebugString*, debug messages are output to a temporary file in the Code editor titled OutDbg1.txt. If a file called OutDbg1.txt is already open, new debug messages are appended to the end of the file.

- If you want to save the contents of OutDbg1.txt in the Code editor, you must choose File|Save or File|Save As.

  If you close the OutDbg1.txt page before saving it, its contents are lost. The IDE does not automatically save its contents and does not prompt you to save when you close its window.

Turning debug messages on and off

You can enable or disable messages sent to the Code editor via *OutputDebugString* as follows:

1. In the Windows registry, locate the key:

```
HKEY_CURRENT_USER
  Software
    Borland
      C++Builder
        1.0
          Debugging
            ShowDebugStrings
```

2. Set the value to

   1 to display debug messages in the Code editor.

   0 to disable debug messages in the Code editor.

   The default is 1.

Example

The following example uses *OutputDebugString* to output the message "`in TForm1 Constructor`":

```
#include <vcl\vcl.h>
#pragma hdrstop

#include "Unit1.h"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    OutputDebugString ("in TForm1 Constructor");
}
```

Debugging start-up code

See also

The integrated debugger demonstrates the following behavior depending on the first action you take when you start a debugging session.

If the first action you take is to choose

- Run|Trace Into (or right-click the CPU window and choose Trace Into), the CPU window displays with the execution point positioned at your program's start-up code (typically address 0x401000).
- Run|Step Over, the Code editor displays with the execution point positioned at your program's entry point (such as main, WinMain, or OWLMain).
- Run|Trace To Next Source Line, the Code editor displays with the execution point positioned at your first executable line of code. This location may be other than your program's entry point, such as a constructor for a global object.
- If you build your program without debug information, when you choose
- Run|Trace Into or Run|Step Over, the CPU window will display.
- Run|Trace To Next Source Line, your program will run to completion.

Example

Suppose you choose File|New Application and then choose Run|Trace to Next Source Line. In this situation, the Code editor displays with the execution point positioned at the first executable line, the destructor for *TObject* in sysdefs.h:

```
virtual __fastcall ~TObject() {}
```

Suppose you create a console application that contains the following code. If you start the debugger by choosing Run|Trace to Next Source Line, the Code editor displays the execution point at the line containing the constructor `myclass:myclassfoo()` which initializes the global object *myglobal*.

```
#include <vcl\condefs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#pragma hdrstop
//--------------------------------------------------
USERES("step.res");
//--------------------------------------------------
class myclass {
   public:
      myclass();
   private:
      int x;
};

myclass::myclass(){
       x = 100;
}

myclass myglobal;

int main(int argc, char **argv)
{
         return 0;
}
```

Handling exceptions in the Debugger

See also

C++Builder lets you control the way exceptions are handled while you debug your program. In addition, it eases the debugging session by treating most hardware exceptions as language exceptions. C++Builder traps the hardware exceptions generated by your C++Builder application, and you can gracefully recover rather than having your program execution end with a system crash.

If a hardware or language exception occurs while you are debugging a C++Builder application, your program halts and the Exception dialog box displays. If you choose OK, you can continue to run your program if your program handles the exception.

To pause the program run when an exception occurs,

1. Choose Options|Environment, then click the Preferences tab.
2. Check Break On Exceptions.

If Break On Exception is checked, C++Builder displays the Exception dialog box when an exception is generated. When you choose OK to close the dialog box, C++Builder opens the Code editor with the execution point positioned on the location of the exception (if no corresponding source is available, the CPU window displays instead).

Debugging multi-threaded applications

See also

The integrated debugger supports debugging multi-thread programs in both Windows NT and Windows 95. Only a single thread, however, can be “active” at a given time. The active thread is the one that responds to debugger commands such as stepping and expression evaluation.

Both the Call Stack window and the CPU window are “thread aware,” meaning that they display information based on a particular thread.

You can specify the active thread in the following ways:

- Choose View|Threads from the menu bar, then select a Thread Id listed in the Thread Status window.
- Right-click the CPU window and choose Change Thread, then select a Thread Id listed in the Select a Thread dialog box

Debugging class member functions

See also

If you use classes in your programs, you can still use the integrated debugger to step through the member functions in your code. The debugger handles member functions the same way it would step through functions in a program that is not object-oriented.

If you define a member function inline, to facilitate debugging the inline function, be sure that Disable Inline Expansion is on (it is off by default):

1. Choose Options|Project.
2. Click the Compiler tab.
3. If the box is cleared, click Disable Inline Expansions. When Disable Inline Expansions is checked, the debugger accurately represents your C++ source code while stepping and tracing.

Debugging external code

See also

If you link external code into your program, you can Step over or Trace into that code if the .OBJ file you link in contains debugging information.

You can debug external code written in any language, including C, C++, Object Pascal, and assembly language. As long as the code meets all the requirements for external linking and contains full Borland symbolic debugging information, the integrated debugger can step through it and display the source in an Edit window. If the external code does not contain Borland debug information, you can still step through the code using the CPU window.

Debugging dynamic-link libraries

See also

The debugger automatically loads the symbol table for all .DLLs. This action occurs when the program starts or when the program explicitly loads a .DLL (in response to a LoadLibrary call). Because of this behavior, there is no special procedure you need to follow to set breakpoints or to trace into .DLLs.

You can debug DLLs written in any language, including C, C++, Object Pascal, and assembly language. As long as the DLL meets all the requirements for external linking and contains full Borland symbolic debugging information, the integrated debugger can step through it and display the source in an Edit window. If the DLL does not contain Borland debug information, you can still step through the code using the CPU window.

Enabled (Breakpoint)

Enables a disabled breakpoint.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint, but want to preserve its settings.

Delete (Breakpoint)

Removes a breakpoint.

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. This command is not reversible.

View Source

Locates a breakpoint in your source code quickly.

The View Source command scrolls the Code Editor to the location of the breakpoint that is selected in the Breakpnoint List window.

Edit Source

Locates a breakpoint in your source code quickly.

The Edit Source command scrolls the Code Editor to the location of the breakpoint that is selected in the Breakpoint List window, and makes the Code Editor active.

Properties
Opens the Edit Breakpoint dialog box, where you can create or modify a breakpoint.

Add (Breakpoint)

Provides two submenu options:

- Source Breakpoint. Opens the Edit Breakpoint dialog box where you can set a breakpoint on a specific line location in your source code. When you run your program, the execution point in the Code editor indicates the breakpoint location.
- Address Breakpoint. Opens the Edit Breakpoint dialog box where you can set a breakpoint on a specific machine instruction. When you run your program, the execution point in the CPU window Disassembly pane indicates the breakpoint location.

Delete All (Breakpoints)
Removes all breakpoints.

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. This command is not reversible.

Disable All (Breakpoints)

Disables all enabled breakpoints.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

Enable All (Breakpoints)

Enable all disabled breakpoints.

Disabling a breakpoint hides the breakpoint from the current program run. When you disable a breakpoint, its settings remain defined, but the breakpoint does not cause your program to stop. When you set a breakpoint, it is enabled by default.

Disabling is useful when you temporarily do not need a breakpoint but want to preserve its settings.

Evaluate/Modify format specifiers

By default, the debugger displays the result in the format that matches the data type of the expression. Integer values, for example, are normally displayed in decimal form. To change the display format, type a comma (,) followed by a format specifier after the expression.

Example

Suppose the Expression box contains the integer value z and you want to display the result in hexadecimal:

1. In the Expression box, type z,h.
2. Choose Evaluate.

Format specifiers

The following table describes the Evaluate/Modify format specifiers.

Specifier	Types affected	Description
,C	Char, strings	Character. Shows characters for ASCII 0 to 31 in the Pascal #nn notation.
,S	Char, strings	String. Shows ASCII 0 to 31 in Pascal #nn notation.
,D	Integers	Decimal. Shows integer values in decimal form, including those in data structures.
,H or ,X	Integers	Hexadecimal. Shows integer values in hexadecimal with the $ prefix, including those in data structures.
,Fn	Floating point	Floating point. Shows n significant digits where n can be from 2 to 18. For example, to display the first four digits of a floating point value, type ,F4. If n is not specified, the default is 11.
,P	Pointers	Pointer. Shows pointers as 32-bit addresses with additional information about the address pointed to. It tells you the region of memory in which the pointer is located and, if applicable, the name of the variable at the offset address.
,R	Records, classes, objects	Records/Classes/Objects. Shows both field names and values such as (X:1;Y:10;Z:5) instead of (1,10,5).
,nM	All	Memory dump. Shows n bytes, starting at the address of the indicated expression. For example, to display the first four bytes starting at the memory address, type ,4M. If n is not specified, it defaults to the size in bytes of the type of the variable. By default, each byte is displayed as two hex digits. Use memory dump with the C, D, H, and S format specifiers to change the byte formatting.

Inspect error

See also

The symbol you are trying to inspect is not recognized by the debugger. For example, it may be an invalid symbol or out of scope.

Change

Enter a new value for the data element displayed in an Inspector window.

Type Cast

Lets you specify a different data type for an item displayed in an Inspector window. Type casting is useful if the Inspector window contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers.

Range

See also

Lets you change the values of an array displayed in an Inspector window.

If you are inspecting a data structure, it is possible the number of items displayed might be so great that you will have to scroll in the Inspector window to see the data you are want. For easier viewing, narrow the display to a range of data items.

Start

Enter a new value for the first element in the array.

Count

Enter the number of elements in the array you want to view.

Threads status box

See also

Choose View|Threads to view the Threads status box.

Use this status box to view the status of all the threads currently executing in the application being debugged.

When a debug event occurs (breakpoint, exception, paused), the thread status view will indicate the status of each thread as it executes. Right-click the Threads status box to access commands to locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

State

The thread state is either Running or Stoped.

Status

The thread status displays one of the following:

Breakpoint	The thread stopped due to a breakpoint.
Faulted	The thread stopped due to a processor exeception.
Unknown	The thread is not the current thread so its status is unknown.
Stepped	The last step command was successfully completed.

Location

Displays the source position. Displays the address if there is no source location available.

Thread ID

Displays the OS assigned thread ID.

Threads status box commands

Right-click the Threads status box to access the following commands:

View Source	Displays the Code editor at the corresponding source location of the selected thread ID, but does not make the Code editor the active window.
Go to Source	Displays the Code editor at the corresponding source location of the the selected thread ID and makes the Code editor the active window.
Make Current	Makes the selected thread the active process if it is not so already.

Select a thread

Select the thread you want to debug from the threads listed.

View Source
Locates a function call in your source code quickly. The Code editor scrolls to the location of the function call selected in the Call Stack window, but does not make the Code editor the active window.

Edit Source
Locates a function call in your source code quickly. The Code editor scrolls to the location of the function call selected in the Call Stack window and makes the Code editor the active window.

Enter search bytes

Enter a byte list to search forward in the Disassembly or Memory Dump pane on the CPU window. You can supply a byte list to search for two or more values located in a specific order. Be sure to precede hexadecimal values with 0x.

For example, if you enter

```
0x5D 0xC3
```

the debugger goes to the following location:

```
004013AB 5D
004013AC C3
```

You can also search for DWords, but you must reverse the order of the bytes.

For example, if you enter

```
0x1234
```

the debugger positions the pane at the following location in memory:

```
34 12
```

Enter a new value

Enter a new value for the memory or register value selected in the CPU window. Be sure to precede hexadecimal values with 0x.

Disassembly pane

Memory Dump pane

Machine Stack pane

Registers pane

Flags pane

Using the Form editor

The C++Builder Form editor provides a convenient and reliable way to create forms that include visual and non-visual components. C++Builder generates a form in the Form editor in the following situations:

- Whenever you add a form to a project.
- Whenever you create a form using the Object Repository.

To add components to the form,

- Select the component and drag it onto the form.

To close a Form editor page do one of the following:

- Select the Form editor page you wish to close, the choose File|Close.
- Click the close button in the upper right corner of the Form editor window.

To close all Form editor pages and the project do one of the following:

- Choose File|Close All.
- Choose File|New Application.

If you have modified a form and have not saved the changes, C++Builder opens the Save File As dialog box, where you can enter a file name.

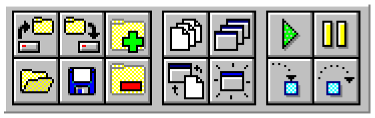
If you have modified the project and have not saved the changes, C++Builder opens the Save As dialog box, where you can enter a name for the project.

To get context-sensitive Help from the Form editor window, place the cursor on the component for which you want Help, then press F1.

About the SpeedBar

See Also

The C++Builder SpeedBar provides shortcuts for menu commands. The graphic below shows the default SpeedBar. However, you can customize the SpeedBar by choosing Properties from the SpeedBar context menu.



To find out more about a default SpeedBar button, click a button in the graphic above.

You can use the separator line that lies between the SpeedBar and Component palette to horizontally resize the SpeedBar.

The SpeedBar has Help Hints. To enable Help Hints, select Show Hints from the SpeedBar context menu.

Open Project button

Displays the Open Project dialog box so you can open a new project.

The menu equivalent is File|New.

Open File button

Displays the Open File dialog box so you can open a new text file.

The menu equivalent is File|Open.

Save button
Stores changes made to all files in the open project the current name of each file.

If you have not previously saved the project, C++Builder opens the Save As dialog box, where you can enter a file name.

Save button

Stores changes made to active file in the Code editor.

If you have not previously saved the file C++Builder opens the Save As dialog box where you can enter a filename.

The menu equivalent is File|Save.

Add File To Project button

Adds the active file in the Code editor to the currently open project.

The menu equivalents are

- File|Add to Project
- Add File on the Project Manager context menu

Remove File From Project button

Opens the Remove From Project dialog box, where you can select a file that you want to remove from the open project.

The menu equivalents are

- File|Remove from Project
- Remove File on the Project Manager context menu

Select Unit From List button

Displays the View Unit dialog box, where you can view any unit in the current project.

When you choose a unit, that unit becomes the active page in the Code editor.

The menu equivalent is View|Units.

Select Form From List button

Displays the View Form dialog box, where you to view any form in the current project.

When you choose a form, it becomes the active form.

The menu equivalent is View|Forms.

Toggle Form/Unit button

Makes the Code editor the active window when the form is selected and makes the form the active window when the Code editor is selected.

The menu equivalent is View|Toggle Form/Unit.

New Form button

Creates a blank form and a new unit and adds them to the project.

The menu equivalent is File|New.

Run button

Compiles and executes your application.

The menu equivalent is Run|Run.

Pause button
Pauses program execution and positions the execution point on the next line of code to execute.

Trace Into button

Executes the program statement highlighted by the execution point. Trace Into lets you execute routines, written by the programmer, one statement at a time. When the routine returns from the call the debugger positions the execution point on the statement following the routine call.

If the execution point highlights a routine whose code was generated by C++Builder, choosing Trace Into causes the debugger to position the execution point at the statement following the routine call.

Note: The Trace Into button is disabled if symbolic debug information is off.

The menu equivalent is Run|Trace Into.

Step Over button

Executes your program one statement at a time, without branching into subroutines. Stepping through your program is helpful if your program is about to execute a routine whose code you do not need to debug at this time.

If the execution point is located on a call to a routine, then issuing Step Over runs that routine at full speed and places the execution point on the statement following the routine call.

Note: The Step Over button is disabled if symbolic debug information is off.

The menu equivalent is Run|Step Over.

Opening a context menu

- Right-click in the window.
- Press Alt+F10 when the cursor is in the window.

Maximize button
Grows your window to encompass your entire screen.

Code editor
Enables you to view or modify any part of the source code contained in the active page.

Page tabs
Provides a way to move between the open files in the Code editor.

Title bar
Displays the name of the active file in the Code editor.

Line and column indicator

Displays the line and column position of the cursor in the Code editor. The first and second numbers show the line number and column number, respectively.

Modified indicator
Indicates whether the text in the active page of the Code editor has been modified since the last time the file was saved. (Blank if the file has not been modified.)

Mode indicator

Indicates whether the editor is in Insert or Overwrite mode.

- In Insert mode (the default mode), text you type is inserted at the cursor.
- In Overwrite mode, text you type overwrites previously entered text.

Use the Insert key on your keyboard to toggle between these two modes.

Using the Code editor

The C++Builder Code editor provides a convenient and reliable way to view and modify your source code. C++Builder generates a page in the Code editor in the following situations:

- Whenever you create a new project
- Whenever you add a form or unit to a project
- Whenever you open a file, even if you do not add it to the project file

You can also use the Code editor to open text files for viewing or modification.

To open a Code editor page,

- Add a new form, unit or other file to a project.

To select a Code editor page,

- In the Code editor window, click the tab corresponding to the page you want to view or modify.

To modify text in the Code editor,

1. Position the cursor at the position you want new text to begin.
2. Type in the new text, pressing Enter to end each line.

To close a Code editor page do one of the following:

- Right-click the Code editor window, then choose Close Page.

To close all Code editor pages and the project, do one of the following:

- Select the Code editor page you wish to close, then choose File|Close.
- Click the close button in upper right corner of the code editor window.

If you have modified code in a page and have not saved the changes, C++Builder opens the Save File As dialog box, where you can enter a file name.

If you have modified the project and have not saved the changes, C++Builder opens the Save As dialog box, where you can enter a name for the project.

Code editor window

The Code editor window contains one or more Code editor pages. The Code editor window cannot be empty - once you close the last page in the Code editor window, the window is closed.

You can open multiple files in one Code editor window. Each file opens on a new page of the Code editor, and each page is represented by a tab at the bottom of the window. For example, when you open a project, it becomes the first tab in the window. Any other files that you open, such as unit files, become subsequent tabs in the window.

You can open a copy of any editor page, which opens a separate window.

To open a Code editor window, you can do one of the following:

- Open a file.
- Choose View|New Edit Window.

The New Window command opens a copy of the current page in the Code editor.

If you have modified the code and not saved the changes, C++Builder opens the Save As dialog box, where you can enter a file name.

Behind the scenes in the Code editor

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. For example, the following code sample shows how to add a button component to a blank form.

Unit1.h

```
Class TForm1 : public TForm
{
__published: // IDE-managed Components
    TButton *Button1;
\\ other code follows
}
```

Adding the button changes the form's declaration (`Class Tform1 : public Tform`) by adding the field for the button itself (`TButton1 *Button1;`). You can view similar code being added to the Code editor, either in your current project or in a new project.

To view code being added in the Code editor,

1. Drag the form's title bar until you can see the entire Code editor.
2. Select the Unit.h tab and scroll in the Code editor until the `__published` declaration part is visible.
3. Add a component to the form while watching what happens in the Code editor.

Note: Do not edit any code that C++Builder generates. Edit only code that you create.

Getting Help in the Code editor

Context-sensitive Help is available from nearly every portion of the Code editor. The context is determined by the current position of the cursor.

To get context-sensitive Help from the Code editor window, place the cursor on the code for which you want Help, then press F1.

Viewing pages in the Code editor window

When a page of the Code editor is displayed, you can scroll through all the data it contains, not just particular sections of your code.

To view a page in the Code editor, do one of the following,

- If the Code editor is already the active window, click the tab corresponding to the page you want to view.
- Choose View|Units.
- Choose View Unit from the Project Manager SpeedMenu.

Write Block To File dialog box

See also

This dialog box enables you to specify the filename and location of an operating system file in which you want to write a block of text you have selected in the Code Editor Window.

When using default key mapping, access this dialog box with: Ctrl+K+W

Read File As Block dialog box

See also

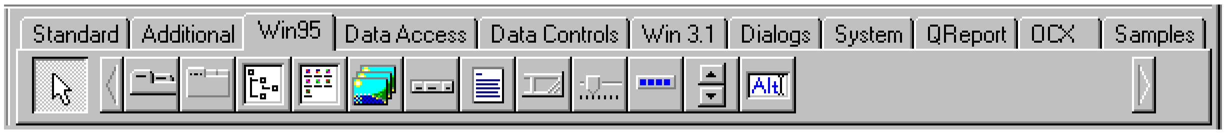
This dialog box enables you to specify the filename and location of an operating system file containing a block of Object Pascal source code which you want to insert in the Code Editor Window at the current cursor position.

When using default key mapping, access this dialog box with: Ctrl+K+R

About the Component palette

See Also

Components are the building blocks of every C++Builder application, and the basis of the C++Builder visual component library. Each page tab in the Component palette displays a group of icons representing the components used to design your application interface.



Components can be either visual or non-visual. Each component has specific attributes that enable you to control your application. These attributes are Properties, Events, and Methods.

You can horizontally resize the Component palette by dragging the separator line which lies between the Component palette and the SpeedBar.

The Component palette provides Help Hints. Help Hints display a small pop-up window containing the name or brief description of the button when your cursor is over the button for longer than one second. To enable Help Hints, select Show Hints from the Component palette context menu.

To get Help on a specific component, click the component and press F1.

The default page tabs divide components into the following functional groups:

Standard components	Standard Windows components
Additional components	Customized components
Windows 95 components	Windows 95 common components
Data Access components	Database access components
Data Controls components	Data-aware controls
Windows 3.1 components	Windows 3.1 components
Dialogs components	Dialog box components
System components	System components
QReport components	Report components
OCX components	OCX components
Sample components	Sample components
Internet components	Internet components

Note: The components on the VBX and Samples page are provided as samples only. Source code for the components on the Samples page can be found in the EBONY\SOURCE\SAMPLES directory of a default installation.

Customizing the Component palette

See Also

To customize the layout of the Component palette, choose the Palette page from the Environment Options dialog box.

Saving a customized Component palette

1. Open the Preferences page of the Environment Options dialog box.
2. Check Desktop from the Autosave options.
3. Click OK

Rearranging Component palette pages

1. Open the Palette page of the Environment Options dialog box.
2. Select a page from the Pages list box.
3. Click the up arrow or down arrow, or drag and drop the page to its new location.
4. Click OK for your changes to take effect.

Rearranging components on the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Select a component from the Components list box.
3. Click the up arrow or down arrow, or drag and drop the component into its new location.
4. Click OK for your changes to take effect.

Moving components to a different Component palette page

1. Open the Palette page of the Environment Options dialog box.
2. Drag and drop the component from the Components list box onto a page in the Pages list box.
3. Click OK for your changes to take effect.

Note: When you move a component to a new page, the component is added as the last item on the page.

Renaming Component palette pages

1. Open the Palette page of the Environment Options dialog box.
2. Select the page from the Pages list box.
3. Click Rename to open the Rename Page dialog box.
4. Enter a new name.
5. Click OK to close the Rename Page dialog box.
6. Click OK for your changes to take effect.

Renaming a component

1. Open the Palette page of the Environment Options dialog box.
2. Select the component from the Components list box.
3. Click the Rename button.
3. Click Rename to open the Rename Page dialog box.
4. Enter a new name.
5. Click OK to close the Rename Page dialog box.
6. Click OK for your changes to take effect.

Adding pages to the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Click the Add button to open the Add Page dialog box.
3. Enter a new page name.
4. Click OK to close the Add Page dialog box.
5. Click OK for your changes to take effect.

Removing pages from the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Select the page from the Pages list box
3. Press Delete.
4. Click OK for your changes to take effect.

Note: Before you can remove a page, it must be empty of components.

Removing components from the Component palette

1. Open the Palette page of the Environment Options dialog box.
2. Select the component you want to remove.
3. Press Delete.
4. Click OK for your changes to take effect.

Alignment palette

Use the Alignment palette to align components to the form, or to each other.

To open the Alignment palette, choose View|Alignment Palette.

The Alignment palette has Tool Help for each button.

The icons on the Alignment palette are:

Icon	Effect
	Aligns the selected components to the left edge of the component first selected. (Not applicable for single components.)
	Moves the selected components horizontally until their centers are aligned with the component first selected. (Not applicable for single components.)
	Aligns the selected component(s) to the center of the form along a horizontal line.
	Aligns the selected components to the right edge of the component first selected. (Not applicable for single components.)
	Aligns the selected components to the top edge of the component first selected. (Not applicable for single components.)
	Moves the selected components vertically until their centers are aligned with component first selected. (Not applicable for single components.)
	Aligns the selected component(s) to the center of the form along a vertical line.
	Aligns the selected components to the bottom edge of the component first selected. (Not applicable for single components.)

If you are unsure of how a particular button on the Alignment palette will act, you can click the button, and the icon on the button will change to show you how it will align the selected components.

Note: You can also use the Alignment dialog box to align components.

About the Object Inspector

See Also

The Object Inspector is the gateway between your application's visual appearance and the code that makes your application run.

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.

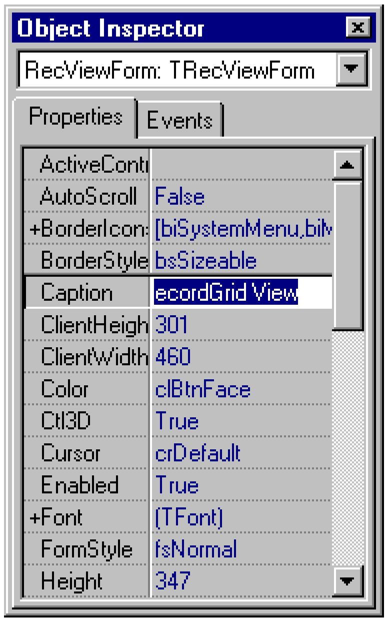
The Object selector at the top of the Object Inspector is a drop-down list containing all the components on the active form and it also displays their object type. This lets you quickly select different components on the current form.

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

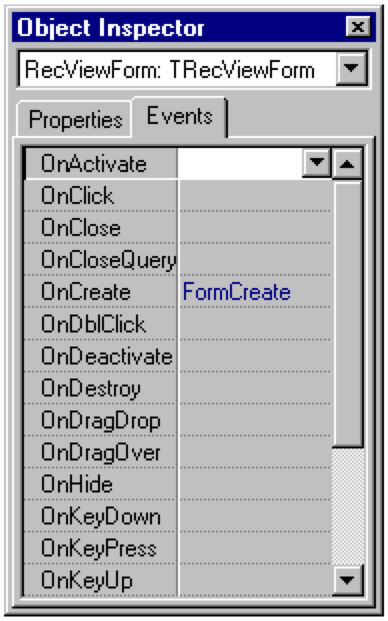
Properties page



ActiveContı	
AutoScroll	False
+BorderIcon:	[biSystemMenu,biM
BorderStyle	bsSizeable
Caption	ecordGrid View
ClientHeigh	301
ClientWidth	460
Color	clBtnFace
Ctl3D	True
Cursor	crDefault
Enabled	True
+Font	(TFont)
FormStyle	fsNormal
Height	347

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 run-time properties by writing source code inside event handlers.
The Properties page displays only the properties of the component that is selected on the form.
By setting properties at design time you are defining the initial state of a component.

Events page



OnActivate	
OnClick	
OnClose	
OnCloseQuery	
OnCreate	FormCreate
OnDblClick	
OnDeactivate	
OnDestroy	
OnDragDrop	
OnDragOver	
OnHide	
OnKeyDown	
OnKeyPress	
OnKeyUp	

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

Object selector
Displays the active component whose properties and events you are currently editing. You can use the drop-down list to select a component.

Property column
Lists the design-time properties for the component you have selected on your form.

Value column
Displays the current value for a property. Each value uses one of the C++Builder property editors to set values.

Object Inspector tabs

Provide you with a means to switch between the Property page and the Event page of the Object Inspector. To change pages, click a tab.

Minimize button
Shrinks the window down to an icon.

Scroll bars
Provides you with a means to scroll the current window to view objects that cannot fit into the window. To scroll a window, click the up scroll arrow or the down scroll arrow, or you can drag the scroll box.

Control menu
Click the Control-menu box to open the Control menu for the current window.

Handler column

Displays the event-handler link for an event. To generate a default event-handler link for an event, double-click the Handler column.

Event column
Lists the possible events for the component you have selected on your form.

About the Object Repository

See Also

C++Builder provides the Object Repository as a means for sharing and reusing forms and projects. The repository itself is really just a text file that contains references to forms, projects, and wizards.

Sharing across projects

By adding forms, dialog boxes, and data modules to the Object Repository, you make them available to other projects. For example, in a simple case, you could have all your projects use the same About box, copied from the Object Repository. A more advanced use of the Object Repository would be to have a standard empty dialog box with the company or product logo and standard button placement, from which all your projects derive standard-looking dialog boxes.

These sharing options are described in detail in Object Repository usage options.

Sharing within projects

The Object Repository can also help you to share items within a project by allowing you to inherit from forms already in the project. When you open the New Items dialog box (by choosing File|New), you'll see a page tab with the name of your project. If you click that page tab, you'll see all the forms, dialog boxes, and data modules in your project. You can then derive a new item from the existing item, and customize it as needed.

For example, in a database application you might need several forms that display the same data, but which provide different command buttons. Instead of creating and maintaining several nearly-identical forms, you could lay out a generic form that contains all the data-display controls, then create separate forms that inherit the data-display layout, but have different command buttons.

By carefully planning your project forms, you can save tremendous amounts of time and effort by sharing forms within projects.

Sharing entire projects

You can also add an entire project to the Object Repository as a template for future projects. If you have a number of similar applications, for example, you can base them all on a single, standardized model.

Using wizards

The Object Repository also contains references to wizards, which are small applications that lead the user through a series of dialog boxes to create a form or project. C++Builder provides a number of wizards, and you can also add your own.

Object Repository usage options

See Also

When you use an item from the Object Repository in a project, you have as many as three options on how to include that item. Keep in mind that items in the Object Repository are there to be shared, and that you want to use them in ways that help, rather than hinder, reuse.

In general, these are the three options for using Object Repository items,

- Copy the item.
- Inherit from the item.
- Use the item directly.

Copying items from the Object Repository

The simplest sharing option is to copy an item from the Object Repository into your project. Copying makes an exact duplicate of the item as it stands and adds the copy to your project. Future changes to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Note: Copying is the only option available for using project templates.

Inherit from Object Repository items

The most flexible and powerful sharing option is to inherit from an item in the Object Repository. Inheriting derives a new class from the item and adds the new class to your project. When you recompile your project, any changes made to the item in the Object Repository will be reflected in your derived class, unless you have changed a particular aspect. Changes made to your derived class do not affect the shared item in the Object Repository.

Note: Inheriting is available as an option for forms, dialog boxes, and data modules, but not for project templates. It is the only option available for reusing items from within the same project.

Using Object Repository items directly

The least flexible sharing option is using an item from the Object Repository directly in your project. Using the item adds the item itself to your project, just as if you had created it as part of that project. Design-time changes made to the item therefore appear in all projects that directly use the item, as well as affecting any projects that inherit from the item.

Note: Using items directly is an available option for forms, dialog boxes, and data modules.

Items shared this way should generally be modified only at run time, to avoid making changes that affect other projects.

The Use option is the only option available for wizards, whether form wizards or project wizards. Using a wizard doesn't actually add shared code, but rather runs a process that generates its own code.

Using project templates

See Also

C++Builder provides project templates, pre-designed projects you can use as starting points for your own projects. Project templates are part of the Object Repository (located in the OBJREPOS subdirectory), which also provides form objects and wizards.

When you start a project from a project template (other than the blank project template), C++Builder prompts you for a project directory, a subdirectory in which to store the new project's files. If you specify a directory that doesn't currently exist, C++Builder creates it for you. C++Builder copies the template files to the project directory. You can then modify it, adding new forms and units, or use it unmodified, adding only your event-handler code. In any case, your changes affect only the open project. The original project template is unaffected and can be used again.

To start a new project from a project template,

1 Choose File|New to display the New Items dialog box.

2 Choose the Projects tab.

3 Select the project template you want and choose OK.

4 In the Select Directory dialog box, specify a directory for the new project's files.

A copy of the project template opens in the specified directory.

Adding projects to the Object Repository

You can add your own projects and forms to those already available in the Object Repository. This is helpful in situations where you want to enforce a standard framework for programming projects throughout an organization.

For example, suppose you develop custom billing applications. You might have a generic billing application project that contains the forms and features common to all billing systems. Your business centers around adding and modifying features in this application to meet specific client requirements. In such a case, you might want to save the project containing your Generic Billing application as a project template and perhaps specify it as the default new project on your C++Builder development system. Likewise, you'll probably have a particular form within this project that you want to appear as the default main or new form.

To add a project to the Object Repository,

1 If necessary, open the project you want added to the Object Repository.

2 Choose Project|Add to Repository.

3 In the Title edit box, enter a project title.

The title for the template will appear in the Object Repository window.

4 In the Description field, enter text that describes the template.

This text will appear in the Object Repository window's status bar.

5 In the Page field, choose the name of the page in the New Items dialog box (probably Projects) you want the template to appear on.

6 In the Author field, enter text identifying the author of the application.

Author information appears only when the user views the repository items with full details.

7 Choose Browse to select an icon to represent this template in the Object Repository.

9 Choose OK to save the current project as a project template.

Note: If you later make changes to a project template, those changes automatically appear in new projects created from that template. They will not, however, affect projects already created from that template.

You can also save your own forms as form templates and add them to those already available in the Object Repository. This is helpful in situations where you want to develop standard forms for an organization's software.

To add a form to the Object Repository as a template, right-click the form and choose Add to Repository.

Customizing the Object Repository

See Also

The settings in the Object Repository Options dialog box affect the behavior of C++Builder when you begin a new project or create a new form in an open project. This is where you specify,

- Default project
- Default new form
- Default main form

You always have to option to override these defaults by choosing File|New and selecting from the New Items dialog box.

By default, opening a new project displays a blank form. You can change this default behavior by changing Object Repository options.

Specifying the default new project

The default new project opens whenever you choose New Application from the File menu on the C++Builder menu bar. If you haven't specified a default project, C++Builder creates a blank project with an empty form. You can specify a project template (including a project you have created and saved as a template) as the default new project.

You can also designate a project wizard to run by default when you start a new project. A project wizard is a program that enables you to build a project based on your responses to a series of dialog boxes.

To specify the default new project,

1 Choose Options|Repository to display the Object Repository dialog box.
2 Choose Projects in the Pages list.
3 Select the project object you want as the default new project from the Objects list.
4 With the object you want selected, check New Project.
5 Choose OK to register the new default setting.

Specifying the default new form

The default new form opens whenever you choose File|New Form or use the Project Manager to add a new form to an open project. If you haven't specified a default form, C++Builder uses a blank form. You can specify any form template, including a form you have created and saved as a template, as the default new form. Or you can designate a form wizard to run by default when a new form is added to a project.

To specify the default new form for new projects,

1 Choose Options|Repository to display the Object Repository dialog box.
2 Choose Forms in the Pages list.
3 Select the form object you want as the default new form.
4 With the object you want selected, check New Form.
5 Choose OK to register the new default setting.

Specifying the default main form

Just as you can specify a form template or wizard to be used whenever a new form is added to a project, you can also specify a form template or wizard that should be used as the default main form whenever you begin a new project.

To specify the default main form for open projects,

1 Choose Options|Repository to display the Object Repository dialog box.
2 Choose Forms in the Pages list.
3 Select the form object you want as the default main form.
4 With the object you want selected, check Main Form.
5 Choose OK to register the new default setting.

Dialog wizard

See AlsoYou can use the Dialog wizard to design a dialog box for your application.

To display this wizard, choose File|New to display the Object Repository. Click the Dialogs tab and select Dialog wizard.

Follow the instructions in the wizard and click Next. Once you get to the last screen, click Finish. The wizard will then create the dialog box form. You can modify the form to customize it further if needed.

You will be prompted to select the type of dialog box (single or multipage) and the button placement for the standard OK, Cancel, and Help buttons.

Edit Object Info dialog box

See Also

Use this dialog box to edit information of Object Repository items. To display this dialog box,

1 Choose File|New to display the Object Repository.
2 Right-click on the Object Repository and select Properties.
3 Select a page from the Pages list and click one of the objects in the Objects list.
  The Edit Object button displays.
4 Click Edit Object to display the Edit Object Info dialog box.

Title
This text box displays the title of the selected item.

Description
This text box displays the description of the selected item.

Page
Displays the current page containing the selected item.

To change the page on which the item appears, select a different page from the Page list.

Author
Displays the name of the Author of the selected item.

Browse button
The icon of the selected item is displayed to the left of the Browse button. Use the Browse button to select a different icon.

Add page dialog box

Specify the name for a new page to add to the Object Repository in the Add page edit box. After you add a page to the Object Repository, it appears as a separate tab sheet when you choose File|New to invoke the New Items dialog box for the Object Inspector.

Rename page dialog box

Specify a new name for an existing page in the Object Repository in the Rename page edit box. After you rename a page to the Object Repository, it appears in place of the old name on the existing page.

Adding an automation object (Automation Object wizard)

See also

Once you have created a server, whether an application or a dynamic-link library, you can then add an OLE automation object. The process is largely automated, but you need to supply some information to the Automation Object wizard.

To add an automation object to your server,

1 Choose File|New, and choose Automation Object from the Object Repository.

C++Builder opens the Automation Object wizard.

2 Name the automation object.

Class Name is the name used internally by your server to identify the OLE object. It must be a valid C++ identifier, and by convention, its name should start with the letter T.

3 Name the OLE class.

This is the name used externally to create these objects. When your server registers the OLE object with Windows, it places this name in the system registry. Client applications use this name when calling CreateOLEObject (called CreateObject in Visual Basic).

4 Describe the object being exported. This string goes into the registry.

5 Specify the instancing for the object.

For in-process servers (DLLs), this is always Multiple Instance, applications are more typically Single Instance.

6 Choose OK to generate the automation object.

The Automation Object wizard produces the following:

- An empty automation object declaration, descending from TAutoObject.
- Registration code that calls C++Builder's OLE Automation manager, which in turn registers the object with Windows, placing the object and server in the system registry.
- Part of the process of generating registration code for your automation object includes generating a globally-unique ID (GUID) for the server. In general, you should never change this once generated. The pairing of the OLE class name and GUID is unique, and if you change one or the other, you will cause errors in applications using those automation objects. You can, however, safely edit the portions of the registration information containing the description and instancing.

Using the Database Form wizard

See also

The Database Form wizard enables you to easily generate a form that displays data from an external InterBase, Paradox, dBase, or ORACLE database.

The Database Form wizard helps you create two types of database forms,

- Simple database forms
- Master-detail forms

The tool automates such form building tasks as

- Connecting the form to Table and Query components.
- Writing SQL statements for Query components.
- Placing interactive and non-interactive components on a form.
- Defining a tab order.
- Connecting DataSource components to interactive components and Table/Query components.

Creating a form using the Database Form wizard

See also

You can use the Database Form wizard to create a simple information form.

To build a form using the Database Form wizard,

1. Open the Database Form wizard by choosing Database|Form wizard.
2. Select a Form Option. A simple form uses one table. A master/detail form creates a link from a detail table to a master, or summary, table and displays only those records in the detail table that are linked to the current record in the master table (in C++Builder's sample files, CUSTOMERS.DB would be a master table and ORDERS.DB would be an associated detail table).
3. Select a Dataset Option. A table will provide access to all records (rows) and all fields selected later in the Form wizard. A query will allow you to limit the records (rows) retrieved by specifying retrieval criteria.
4. Click Next.
5. Select an an alias from the Drive or Alias Name list. Select the DBDEMOS alias to create a sample form with C++Builder's sample database files. To create an alias, see the online help for the BDE Configuration Utility.

- If you have not created an alias, select the drive and directory containing the database in the Form wizard dialog box.

6. Select the fields to use on the generated form:

To use only some of the fields,

1. Press and hold Ctrl.
2. Select each field you want from the Available Fields list.
3. Choose the > button.

   To use all of the fields from the Available Fields list, click the button marked >>.

   To remove fields from the Selected Fields list, click the buttons marked < or <<.

To reorder the fields in the Selected Fields list,

1. Select a field to move.
2. Choose the Up or Down button to change the field's position in the list.

- For the purposes of this exercise, use all the fields from the Available Fields list.

3. Choose Next to proceed.

   The next Form wizard screen presents options for displaying the selected fields on the form. The wizard explains and illustrates each of your choices. For the purposes of this exercise, choose the Vertical option.

   The Form wizard generates text labels for each of the data entry components in the generated form when you opt for a vertical layout. You can choose the way these labels are displayed in relation to the data entry fields. The screen explains and illustrates your choices. For this exercise, choose the Left option, then choose Next to proceed.

   Select a method for generating the form. You may choose to generate only a form when your application will contain only one dataset or choose to generate both a form and a data module if your application will contain several datasets or data sources and you want to centralize their access or re-use standard designs.

4. Choose the Finish button to generate the form.

   The Form wizard generates a database form based on your choices.

Features chart

The following table summarizes the tools and features in C++Builder. Available tools and features differ depending on C++Builder version. All features are available in the C++Builder Client/Server Suite.

CS=Client/Server Suite DB=Developer DT=Desktop

Tool/Feature	**Description**	**C/S**	**DB**	**DT**
SQL Explorer	SQL-enabled integrated tool for browsing databases, managing BDE aliases, and creating data dictionaries.	3		
Database Explorer	Integrated tool for browsing databases, managing BDE aliases, and creating data dictionaries.		3	3
SQL Monitor	Integrated tool for tracing and examining SQL query performance.	3		
Visual Query Builder	Integrated tool for visual building of SQL queries.	3		
Data Pump Wizard	Standalone tool for moving metadata and data between databases	3		
Data Dictionary	Stores extended field attributes apart from application code; shares attributes across fields, datasets, and applications.	3	3	
Object Repository	Stores data modules and forms for sharing among applications.	3	3	3
Data modules	Non-visual component containers for centralized data-access across forms and applications.	3	3	3
Data-access components	Non-visual components for encapsulating database connections.	3	3	3
Data-aware controls	Visual components for providing a user interface to data.	3	3	3
Database Desktop (DBD)	Tool for browsing, creating, and changing desktop databases.	3	3	3
Borland Database Engine (BDE)	Borland's core database engine and connectivity software.	3	3	3
SQL Links	BDE SQL drivers for Sybase, Microsoft SQL-Server, Oracle, and InterBase	3		
ODBC socket	BDE support socket for third-party ODBC drivers.	3	3	
BDE API	BDE application programming interface and online help files.	3	3	
Quick Reports	C++Builder components for creating pre-defined reports in an application.	3	3	3
InterBase NT	Borland InterBase Workgroup Server for NT, two-user license.	3		
Local InterBase Server	32-bit Local InterBase Server.	3	3	

Using the object repository with data modules

The Object Repository stores links to data modules, forms, and projects for reuse and reference. When you create a new application, you can:

n ▪ *Copy* an existing data module, form, or project, ensuring that your copy is completely independent of the repository.

- *Inherit* an existing data module, form, or project, ensuring that changes to the linked module, form, or project in the repository are replicated to your application when you recompile.
- *Use* an existing data module, form, or project, ensuring that changes you make to the module, form, or project are available for use in other applications.

The Object Repository supports team development practices. It uses a referencing mechanism to data modules, forms, and projects where they exist on a network server or shared machine. Every developer in your organization can save objects to a shared location, and then set C++Builder's Object Repository reference to point to that location.

The Object Repository is also customizable through the Options|Repository menu in the IDE.

Using the data dictionary

The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with extended fields attribute set in the Data Dictionary. Using the Data Dictionary also ensures a consistent data appearance within and across the applications you create.

In a client/server environment the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see Using the Fields Editor. To learn more about creating a Data Dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

Using the SQL/Database Explorer

In C++Builder Client/Server Suite you browse databases and populate tables with data using the SQL Explorer from the IDE. In C++Builder Developer you use the Database Explorer. Both versions of the Explorer enable you to:

n ▪ Examine existing database tables and structures. The SQL Explorer enables you to examine remote SQL databases, and to query them.

n ▪ Populate tables with data.

n ▪ Create extended field attribute sets in a Data Dictionary for later retrieval and reuse. Extended field attributes describe how values in a field are formatted and displayed.

n ▪ Associate extended field attributes with fields in your application.

n ▪ Create and manage BDE aliases, used by your application to connect to databases.

To learn more about the SQL Explorer, and the Database Explorer see their respective online help files.

Using the SQL Monitor

In C++Builder Client/Server Suite you can use the SQL Monitor to trace and time calls between your client application and a remote SQL database server. Timing information is useful for optimizing SQL statements and transactions. A series of options enables you to trace:

- Prepared query statements
- Executed query statements
- Statement operations
- Connect/Disconnect
- Transactions
- Blob I/O
- Vendor errors
- Vendor API calls

You can use the SQL Monitor to examine how optimally SQL statements in an application are performed, to see the SQL statements generated by the Borland Database Engine (BDE), to see if the database client libraries are functioning properly, and to see if the database server is executing a run-away query. The SQL Monitor also enables you to save and print session logs for further reference and comparison.

Using the Visual Query Builder

In C++Builder Client/Server Suite you can enter SQL statements directly in SQL and Update SQL property editors, or you can invoke the Visual Query Builder to construct a query based on a visible representation of tables and fields in a database.

To use the Visual Query Builder, select a query component, right-click it to invoke the SpeedMenu, and choose Query Builder.

Data access component summary

C++Builder provides non-visual data-access components that encapsulate your client's communication with a database. Data-access components only deal with database connectivity, which enables you to focus your attention on your application's data needs without worrying about user interface. Typically these components are placed in a data module container in an application.

The following table summarizes the components that appear on the Data Access page of the Component palette, and where in the *Database Application Developer's Guide* you can get more information about them:

Component	**Purpose**
TDataSource	Acts as a conduit between other data access components and data-aware visual controls. For more information about TDataSource, see Chapter 6, "Working with data sources."
TTable	Represents a dataset that retrieves all columns and records from a database table. For more information about TTable, see Chapter 8, "Working with tables."
TQuery	Represents a dataset that retrieves a subset of columns and records from one or more local or remote database tables based on an SQL query. For more information about TQuery, see Chapter 9, "Working with queries."
TStoredProc	Represents a dataset that retrieves one or more records from a database table based on a stored procedure defined for a database server. For more information about TStoredProc, see Chapter 10, "Working with stored procedures."
TDatabase	Encapsulates a client/server connection to a single database in one session. For more information about TDatabase, see Chapter 3, "Connecting to databases."
TSession	Represents a single session in a multi-threaded database application. Each session can have multiple database connections as long as each thread associated with a particular database has its own session. For more information about TSession, see Chapter 3, "Connecting to databases."
TBatchMove	Encapsulates a dataset used to move data from one table to another en masse. For more information about TBatchMove, see Chapter 16, "Working with TBatchMove."
TReport	Encapsulates a ReportSmith report in an application. For more information about TReport, see Chapter 14, "Using reports."
TUpdateSQL	Represents SQL INSERT, UPDATE, and DELETE statements that can be used to update the read-only result sets of some queries. For more information about TUpdateSQL, see Chapter 15, "Working with cached updates."

Data controls summary

C++Builder provides data-aware visual components, called visual controls, that encapsulate user interaction with your application's data sources. You design a user interface with these controls. You place these controls on the forms in your application. Each control is linked to one or more fields or records, and controls how a user sees a field or record in your application.

You link visual controls to data-access components through a data source component. A data-source component acts as a conduit between an application's low-level data-access interactions and the high-level view of data its users are provided.

The following table summarizes the data-aware visual controls that appear on the Data Controls page:

Component	**Purpose**
TDBGrid	Display and edit dataset records in tabular format.

TDBNavigator	Cursor through dataset records; enable Edit and Insert states; post new or modified records; cancel edit mode; refresh data display.
TDBText	Display a field as a label.
TDBEdit	Display and edit a field in an edit box.
TDBMemo	Display and edit multi-line or blob text in a scrollable, multi-line edit box.
TDBImage	Display and edit a graphics image or binary blob data.
TDBListBox	Display a list of choices for entry in a field.
TDBComboBox	Display an edit box and drop-down list of choices for edit and entry in a field.
TDBCheckBox	Display and set a Boolean field condition in a check box.
TDBRadioGroup	Display and set exclusive choices for a field in a radio button group.
TDBLookupListBox	Display a list of choices derived from a field in another dataset for entry into a field.
TDBLookupComboBox	Display an edit box and drop-down list of choices derived from a field in another dataset for edit and entry in a field.
TDBCtrlGrid	(Client/Server Suite and Developer only). Display and edit records in a tabular grid, where each cell in the grid contains a repeating set of data-aware components and one record.

Using TTable

A table component is the most fundamental and flexible dataset component class in C++Builder. It gives you access to every row and column in an underlying database table, whether it is from Paradox, dBASE, an ODBC-compliant database such as Microsoft Access, or an SQL database on a remote server, such as InterBase, Sybase, or SQL Server.

You can view and edit data in every column and row of a table, you can work with a range of rows in a table, and you can filter records to retrieve a subset of all records in a table based on filter criteria you specify.

Creating a table component

To create a table component:

1. Place a table component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *DatabaseName* of the component to the name of the database to access.
3. Set the *TableName* property to the name of the table in the database. You can select tables from the drop-down list if the *DatabaseName* property is already specified.
4. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the table component. The data source component is used to pass a result set from the table to data-aware components for display.
5. To view data from a table, place a data-aware component from the Data Controls page of the Component palette on a form and set its *DataSource* property to the name of the table's data source. Set the table component's *Active* property to *true* to display data in the data-aware component.

Note: These are general steps for creating a table component. There are additional properties you may need to set because of application requirements.

Searching on a table

You can search a table for specific records using the generic search methods Locate and Lookup. These methods enable you to search on any type of columns in any table, whether or not they are indexed or keyed.

Table components also support Goto and Find. While these methods are documented here to allow you to work with legacy applications, you should usually use Lookup and Locate in your new applications. Moreover, you may see performance gains in existing applications if you convert them to use the new method.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass Locate the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. For example, the following code moves the cursor to the first row in the CustTable where the value in the Company column is "Professional Divers, Ltd.":

```
{
bool LocateSuccess;
TLocateOptions SearchOptions;

SearchOptions << loPartialKey;
LocateSuccess = CustTable->Locate("Company", "Professional Divers, Ltd.",
SearchOptions);
}
```

If Locate finds a match, the first record containing the match becomes the current record. Locate returns *true* if it finds a matching record, *false* if it does not. If a search fails, the current record does not change.

The real power of Locate comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are variants, which enables you to specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semi-colons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the *Lookup* method), or you must construct the variant array on the fly using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
{
  Variant tmp(OPENARRAY(int, (0,1)), vtInteger);
  tmp << (int)("Sight Diver");
  tmp << (int)("P");
VarArrayOf(&tmp,1);
}
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, Locate uses the index. Otherwise, Locate creates a BDE filter for the search.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. Lookup does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass Lookup the name of column to search, the field value to match, and the field or fields to return. For example, the following code looks for the first row in the CustTable where the value in the Company column is "Professional Divers, Ltd.", and returns the company name, a contact person, and a phone number for the company:

```
{
```

```
Variant LookupResults;
LookupResults = CustTable->Lookup("Company", "Professional Divers, Ltd.",
  "Company;
Contact; Phone");
}
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as variants. If more than one return value is requested, Lookup returns a variant array. If there are no matching records, Lookup returns a Null variant. For more information about variant arrays, see the C++ Language Reference.

The real power of Lookup comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual in the string items with semi-colons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the Lookup method), or you must construct the variant array on the fly using the *VarArrayOf* function. The following code illustrates a lookup search on multiple columns:

```
{
  Variant tmp(OPENARRAY(int, (0, 1)), vtInteger);
  tmp << (int)("Sight Diver", 0);
  tmp << ("Christiansted", 1);
  Variant LookupResults;
  LookupResults = CustTable->Lookup("Company;City", tmp,
  "Company;Addr1;Addr2;State;Zip");
}
```

Lookup uses the fastest possible method to locate matching records. If the columns to search are indexed Lookup uses the index. Otherwise, Lookup creates a BDE filter for the search.

Using TQuery

A query component encapsulates an SQL statement that returns a set of related rows and columns from one or more database tables. SQL (Structured Query Language) is an industry-standard relational database language that is used by most remote server-based database vendors, such as Sybase, Oracle, InterBase, and SQL Server.

C++Builder query components can be used with remote database servers (C++Builder Client/ Server Suite only), with Paradox and dBASE, and with OBDC-compliant databases such as Microsoft Access and FoxPro.

A query component differs from a table component in two significant ways. It can:

- Access more than one table at a time (called a "join" in SQL).
- Automatically access a subset of rows and columns in its underlying table(s), rather than always returning all rows and columns and requiring you to set ranges and filters to restrict row access.

The data returned by a query component depends on the selection criteria you specify in the component's SQL property at design time or at run time. The selection criteria is in the form of an SQL statement, and can be

- *Static*, where all parameters in the statement are specified at design time, or
- *Dynamic*, where some or all of a statement's parameters are set at run time.

In addition to encapsulating standard SQL features, query components also offer a powerful ability not found on most remote database servers: the ability to create "heterogeneous joins," queries against more than one server or table type.

Choosing between query and table components

Table components are full-featured, flexible, and easily used data access components that are sufficient for many database applications.

If you are a desktop client/server developer, you are familiar with the table component paradigm, and know its strengths. Query components offer different capabilities from table components, and are useful when you need to:

- Query data in multiple tables.
- Restrict data access to a subset of rows and columns across tables.
- Write applications that require SQL syntax for compatibility with other SQL applications.

If you are a client/server developer used to working with remote databases, you are familiar with SQL, and know the strength in query-based data access. On the other hand, table components may occasionally offer you data access alternatives, such as ranges, that make them attractive.

Creating a query component

To create a query component:

1. Place a query component from the Data Access page of the Component palette in a data module, and set its *Name* property appropriately for your application.
2. Set the *DatabaseName* property of the component to the name of the database to query. *DatabaseName* can be a Borland Database Engine (BDE) alias, or an explicit directory path and database name. If your application uses database components, *DatabaseName* can be set to a local alias defined for the database component instead.
3. Specify an SQL statement in the *SQL* property of the component, and optionally specify any parameters for the statement in the *Params* property. For more information, see "Setting the SQL property at design time" in the Database Applications Developers Guide.
4. Place a data source component in the data module, and set its *DataSet* property to the name of the query component. The data source module is used to return a result set from the query to data-aware components for display.

Note: These are the general steps for creating a query component. Depending on the type of query you create (static or dynamic), there may be additional properties you need to set before executing a query.

Using TStoredProc

A stored procedure component encapsulates a stored procedure in a database on a remote server. A stored procedure is a set of statements, stored as part of a remote server's database metadata (like tables, indexes, and domains). A stored procedure performs a frequently-repeated database-related task on the server and passes results to a client application, such as a C++Builder database application. The stored procedure component enables C++Builder applications to execute server stored procedures.

Often, operations that act upon large numbers of rows in database tables—or that use aggregate or mathematical functions—are candidates for stored procedures. By moving these repetitive and calculation-intensive tasks to the server, you improve the performance of your database application by:

n ▪ Taking advantage of the server's usually greater processing power and speed.

n ▪ Reducing the amount of network traffic since the processing takes place on the server where the data resides.

For example, consider an application that needs to compute a single value: the standard deviation of values over a large number of records. To perform this function in your C++Builder application, all the values used in the computation must be fetched from the server, resulting in increased network traffic. Then your application must perform the computation. Because all you want in your application is the end result—a single value representing the standard deviation—it would be far more efficient for a stored procedure on the server to read the data stored there, perform the calculation, and pass your application the single value it requires.

See your server's database documentation for more information about its support for stored procedures.

Creating a stored procedure component

To create a stored procedure component for a stored procedure on a database server:

1 Place a stored procedure component from the Data Access page of the Component palette in a data module.

2 Set the *DatabaseName* property of the stored procedure component to the name of the database in which the stored procedure is defined. *DatabaseName* must be a BDE alias.

3 Set the *StoredProcName* property to the name of the stored procedure to use, or select its name from the drop-down list for the property.

4 Specify input parameters, if necessary, in the *Params* property. You can use the Parameters editor to set input parameters.

Note: The Parameters editor also lets you see the output parameters used by the procedure to return results to your application.

Understanding input parameters

Use input parameters to pass values from an application to a stored procedure.

Many stored procedures require you to pass them a series of input arguments, or parameters, to specify what and how to process. In C++Builder, the Parameters editor retrieves information about input and output parameters from the server. For some servers, all of the information required to run the stored procedure may not be accessible. In this case, you may need to enter information about the type of parameter (input, output, result) and/or the data type of the parameter in order to use the procedure. You will also need to enter the value that you will be passing to the stored procedure as an input parameter. The order in which the input parameters are displayed is significant, and is determined by the stored procedure definition on the server. If you are not sure of the ordering of the input and output parameters for a stored procedure, use the Parameters editor.

Setting input parameters at design time

At design time, the easiest and safest way to view and edit input parameters is to invoke the Parameters editor. The Parameters editor lists input parameters in the correct order, and lets you assign values to them.

To invoke the Parameters editor, click the ellipsis in the Params property value box in the Object

Inspector or:

- Select the stored procedure component.
- Invoke the context menu.
- Choose Define Parameters.

The StoredProc Parameters editor

The Parameter name list box displays all input, output, and result parameters for the procedure. Information on input and output parameters is retrieved from the server. For some servers, not all parameter information is accessible. In this case, you must provide the missing parameter information. If you are sure that the stored procedure is valid, but no parameter names are displayed, use SQL/Database Explorer to look for the stored procedure and inspect its text.

The Parameter type combo box describes whether a parameter selected in the list box is an input, input/output, output, or result parameter. If a server's stored procedure allows it, a single parameter may be both an input and output. If the parameter type information is missing, you must set the parameter type for it.

Note: Sybase, MS-SQL, and Informix servers generally do not return information on parameter types.

The Data type combo box lists the data type for a parameter selected in the list box. The data type can be any standard SQL data type except BLOB and arrays of data types. If the data type has not been provided from the server, you must set the data type. Some servers will also allow you to override the default data type here.

Note: Informix servers generally do not return information on data types.

Input parameters are passed by value from your application to a stored procedure. The Value edit box enables you to enter a value for a selected input parameter, and the NULL Value check box enables you to set a NULL value for the selected input parameter if its data type supports NULL values. Values should be assigned based on the declared data type. Each input parameter must be assigned either a value or a NULL value.

The Add button enables you to add parameters to a stored procedure definition. The Delete button enables you to remove parameters, and the Clear button removes all parameters from the list. If the stored procedure that you have named in the StoredProcName property is valid, these buttons will be disabled because you cannot modify this data. These buttons will be enabled when the stored procedure is not named, is invalid, or if C++Builder is unable to retrieve this information from the server. If you are sure that the stored procedure is valid, but no parameter names are displayed, use SQL/Database Explorer to look for the stored procedure and inspect its text and then add parameters as appropriate.

You will not be able to add, delete, or clear parameters for servers that pass parameter information to C++Builder except if you are working with Oracle overloaded stored procedures. For more information on working with Oracle overloaded stored procedures, see "Working with Oracle overloaded stored procedures" in the Database Applications Developers Guide.

To signal the end of parameter definition, choose OK.

Important: Defining parameters at design time also prepares the stored procedure for execution. A stored procedure must be prepared before it can be executed at run time.

Setting parameters at run time

To assign values to parameters at run time, access the *Params* property directly. *Params* is an array of parameter strings. For example, the following code assigns the text of an edit box to the first string in the array:

```
StoredProc1->Params[0]->Items[0]->AsString = Edit1->Text;
```

You can also access parameters by name using the *ParamByName* method:

```
StoredProc1->ParamByName("Company")->AsString = Edit1->Text;
```

The *ParamBindMode* property determines how the elements of the *Params* array will be matched with stored procedure parameters. If the *ParamBindMode* property is set to *pbByName* (the default), parameters will be bound based on their names in the stored procedure. If *ParamBindMode* is set to *pbByNumber*, parameters will be bound based on the order in which they are defined in the stored procedure.

```
ParamBindMode = pbByName;
```

Preparing and executing a stored procedure

To use a stored procedure, you must prepare it and execute it. You can prepare a stored procedure at:

- Design time, by choosing OK in the Parameters editor.
- Run time, by calling the *Prepare* method of the stored procedure component.

  For example, the following code prepares a stored procedure for execution:

  ```
  StoredProc1->Prepare();
  ```

Note: You can prepare a stored procedure both at run time and at design time. In fact, if your application changes parameter information at run time, such as when using Oracle overloaded procedures, you should prepare the procedure again.

To execute a prepared stored procedure, call the ExecProc method for the stored procedure component. The following code illustrates code that prepares and executes a stored procedure:

```
StoredProc1->Params[0]->Items[0]->AsString = Edit1->Text;
StoredProc1->Prepare();
StoredProc1->ExecProc();
```

When you execute a stored procedure, it returns either output parameters or a result set. There are two possible return types: singleton returns, which return a single value or set of values, and result sets, which return many values, much like a query.

Understanding output parameters and result sets

A stored procedure returns values in an array of output parameters. Return values can be either asingleton result or a result set with a cursor, if the server supports it.

To access a stored procedure's output parameters at run time, you can index into the *Params* string list, or you can use the *ParamByName* method to access the values. The following statement both set the value of a edit box based on output parameters:

```
Edit1->Text = StoredProc1->Params[6]->Items[0]->AsString;
Edit1->Text = StoredProc1->ParamByName("Company")->AsString;
```

Note: If a stored procedure returns a result set, you may find it more useful to access and display return values in standard data-aware controls.

Working with result sets

On some servers, such as Sybase, stored procedures can return a result set similar to a query. Applications can use data aware controls to display the output of such stored procedures.

To display the results from a stored procedure in data-aware controls:

1. Place a datasource component on the data module.
2. Set the *DataSet* property of the datasource to the name of the stored procedure component from which to receive data.
3. Set the *DataSource* properties of the data-aware controls to the name of the datasource component.

The data-aware controls can now display the results from a stored procedure when the *Active* property for the stored procedure component is *true*.

Using TBatchMove

The TBatchMove component enables you to copy, append, delete, and update tables and data, and is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate. It inherits many of its fundamental properties and methods from TDataSet.

Creating a batch move component

To create a batch move component:

1. Place a *TBatchMove* component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *Source* property of the component to the name of the table to copy, append, or update from. You can select tables from the drop-down list of available table components.
3. Set the *Destination* property to the name of the table to create, append to, or update. If you are appending, updating, or deleting, *Destination* must be the name of an existing table. You can select tables from the drop-down list of available table components in these cases.
4. Set the *Mode* property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For more information about these modes, see "Specifying a batch move mode" in the Database Applications Developers Guide.
5. Optionally set column mappings using the *Mappings* property. You need not set this property if you want batch move to match column based on their position in the source and destination tables.
6. Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch move operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used.

Using Filters

An application is frequently interested in only a subset of records within a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. C++Builder supports filtering of a table or query to handle both of these requirements. Using filters you can temporarily restrict an application's view of data.

There are three ways to filter a dataset:

- Restricting record visibility at the time of record retrieval using an *OnFilterRecord* event handler.
- Setting the Filter property of the dataset.
- Finding a record in a dataset that matches search values using the *Locate* method for the dataset. If you use *Locate*, C++Builder automatically generates a filter for you at run time if it needs to.

Note: Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

Setting the Filter property of the dataset

The simplest and most common way to use filtering is to turn on filtering for the entire dataset. To turn on filtering a dataset, set the *Filtered* property to *true*. With filtering on, the dataset generates an event for each record in the dataset. In handling that event, you can determine whether to accept or "filter out" each record. You could also supply a filter condition as the value of the *Filter* property instead of writing an event handler.

Note: When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions which can make the current record disappear if it no longer passes the filter. When this occurs, the next record that passes the filter condition becomes the current record.

You can also filter records programmatically by creating a standard control (such as an edit box). The following code example illustrates this.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Table1->Filtered = true;
Table1->Filter = Edit1->Text;
Table1->Refresh;
}
```

Some other filter examples:

```
CustTable->Filter = "State = California";
Table1->Filter = "PatientAge >= 18";
```

Turning filters on and off at run time

At run time you can turn filtering off by setting the *Filtered* property to *false*, and you can turn it on by setting the *Filtered* property to *true*. If you turn off filtering, all records in a dataset are available to your application, but in order for the application to see all records, you must also call the dataset's *Refresh* method. For example:

```
{
CustTable->Filtered = false;
CustTable->Refresh();
}
```

If you turn on filtering at run time, you must also call *Refresh* again to make the filter take effect. The current record may no longer pass the filter condition, and may disappear. If that happens, the next record that passes the filter condition becomes the current record.

Filtering records with the OnFilterRecord event handler

To restrict visible records in a dataset to a subset of those that match a certain set of criteria, follow these steps:

1 Write an *OnFilterRecord* event handler for the dataset.

2 Set the *Filtered* property for the dataset to *true*.

When *Filtered* is *true*, a dataset generates an *OnFilterRecord* event for each record it fetches. The event handler for the *OnFilterRecord* tests each record, and only those that meet the Filter's conditions are visible in the application.

Note: When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

Switching filter event handlers at run time

You can change how records are filtered at run time by assigning a different handler to the OnFilterRecord event for a dataset. To apply the new filter to the dataset after changing it, call the dataset's Refresh method:

```
Table1.OnFilterRecord := Query2FilterRecord;
Refresh;
```

Filtering datasets with the Filter property

You can filter dataset components to restrict which records you want to see in the dataset by setting these properties:

- Set the value of the *Filter* property for the dataset.
- Set the *Filtered* property for the dataset to *true*.
- Fine-tune the filter by modifying the *FilterOptions* property.

This method of filtering datasets is most valuable when the filter will be set at run-time. The value in the *Filter* property cannot be compared to a value outside of the table, for example, it cannot be compared to a value in an edit box. To change the value of a filter "on the fly", see the section on "Filtering datasets with the OnFilterRecord event".

The *Filter* property is a string that lets you set conditions on one or more fields of your dataset. You can compare fields to literal values and constants using the comparison operators in the following table and the AND, NOT, and OR operators to combine comparisons. You can also compare a field to a field in an edit control (see the example in "Using the Filtered property" for an example). You must enclose field names that contain spaces within square brackets. :

The following are some examples that can be used to set filtering conditions:

- PatientAge >= 18
- State = 'FL'
- (PatientAge >= 18) AND (Balance > 0)
- (Temperature < 212) AND (NOT Windy)
- In the above example Windy is a logical field.
- (SalePrice < $10,000) OR (Terms > 30)
- [Sale Date] > '1/1/96'.

Writing an OnFilterRecord event handler

A filter for a dataset is an event handler that responds to *OnFilterRecord* events generated by the dataset for each record it retrieves. At the heart of every filter handler is a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your filter handler must set an *Accept* parameter to *true* to include a record, or *false* to exclude it. For example, the following filter displays only those records with the State field set to “CA”:

```
void __fastcall TForm1::CustTableFilterRecord(TDataSet *DataSet, Boolean
  &Accept)
{
Accept = (String(“CA”) == DataSet->FieldByName(“State”)->AsString);
}
```

Using TDatabase

A database component encapsulates the connection to a single database within an application. If you do not need to control database connections (such as specifying a transaction isolation level), do not create database components. Temporary database components are created automatically at run time when an application attempts to open a table for which there is not already a database component. But if you want to control the persistence of database connections, logins to a database server, property values of database aliases, or transactions, then you must create a database component for each desired connection.

Creating a database component

The Data Access page of the Component palette contains a database component you can place in a data module or form. The main advantages to creating a database component at design time are that you can set its initial properties and write *OnLogin* events for it. *OnLogin* offers you a chance to customize the handling of security on a database server when a database component first attaches the server.

Logging into a server

You can control server login with a database component's *LoginPrompt* property and the its *OnLogin* method. Controlling login is essential to server security, and may be required by your remote server.

LoginPrompt specifies whether your users are prompted to log in to a database server the first time your application attempts to connect to a database requiring a login. If *true* (the default), your application displays a standard Login dialog box. The Login dialog box prompts for a user name and password. A mask symbol (an asterisk by default) displays for each character entered in the Password edit box.

If you set *LoginPrompt* to *false*, the standard Login dialog box is not displayed at run time when an application attempts to connect to a database server that requires a login. Instead, your application must provide the user name and password, either through the *Params* property or the *OnLogin* event for the database component. At design time, the standard Login dialog box appears when you first connect to a remote database server if *LoginPrompt* is *false* and the *Params* property does not contain user name and password entries.

Note: Storing hard-coded user name and password entries in the *Params* property or in code for an *OnLogin* event can compromise server security.

The *Params* property is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

At design time you can create or edit connection parameters in three ways:

- Use the SQL/Database Explorer or BDE Configuration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the Object Inspector to invoke the String List editor. To learn more about the String List editor, see the C++Builder User’ s Guide.
- n ▪ Double-click a database component in a data module or form to invoke the Database Properties editor.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

Handling transactions

There are two ways to manage transactions in a C++Builder database application using implicit and explicit control. It especially focuses on:

- Using the methods and properties of the *TDatabase* component that enable explicit transaction control.
- Using passthrough SQL with *TQuery* components to control transactions.

C++Builder also supports local transactions, transactions made against local Paradox and dBASE tables.

Understanding transactions

When you create a database application using C++Builder, C++Builder provides transaction control for all database access, even against local Paradox and dBASE tables. A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If one of the actions in the group fails, then all the actions fail.

Transactions ensure database consistency even if there are hardware failures. It also maintains the integrity of data while permitting concurrent multiuser access.

For example, an application might update the ORDERS table to indicate that an order for a certain item was taken, and then update the INVENTORY table to reflect the reduction in inventory available. If there were a hardware failure after the first update but before the second, the database would be in an inconsistent state because the inventory would not reflect the order entered. Under transaction control, both statements would be committed at the same time. If one statement failed, then both would be undone (rolled back).

Using implicit transactions

By default, C++Builder provides implicit transaction control for your applications through the Borland Database Engine (BDE). When an application is under implicit transaction control, C++Builder uses a separate transaction for each record in a dataset that is written to the underlying database. It commits each individual write operation, such as *Post* and *AppendRecord*.

Using implicit transaction control is easy. It guarantees both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop client applications in a multi-user environment, particularly when your applications run against a remote SQL server, such as Sybase, Oracle, Microsoft SQL, or InterBase,® or remote ODBC-compliant databases such as Access and FoxPro, you should control transactions explicitly.

Note: If you used cached updating, you may be able to minimize the number of transactions you need to use in your applications. For more information about cached updates, see Chapter 15, "Working with cached updates," in the *Database Application Developer's Guide*.

Working with explicit transaction control

There are two mutually exclusive ways to control transactions explicitly in a C++Builder database application:

- Use the methods and properties of the *TDatabase* component.
- Use passthrough SQL in a *TQuery* component. Passthrough SQL is only meaningful in the C++Builder Client/Server Suite, where you use SQL Links to pass SQL statements directly to remote SQL or ODBC servers.

The main advantage to using the methods and properties of a database component to control transactions is that it provides a clean, portable application that is not dependent on a particular database or server.

The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation.

Using TDatabase methods and properties

The following table lists the methods and properties of TDatabase that are specific to transaction management, and describes how they are used:

Method or property	Purpose
Commit	Commits data changes and ends the transaction.
Rollback	Undoes data changes and ends the transaction.
StartTransaction	Starts a transaction.
TransIsolation	Specifies the transaction isolation level for a transaction.

StartTransaction, *Commit*, and *Rollback* are methods your application can call at run time to start transactions, control the duration of transactions, and save or discard changes made to the database.

TransIsolation is a database component property that enables you to control how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Starting a transaction

When you start a transaction, all subsequent statements that read from and write to the database occur in the context of that transaction. Each statement is considered part of a group. Write changes made by any statement in the group must be successfully committed to the database, or every change made by every write statement made in the group are undone.

Grouping statements is useful when statements are dependent upon one another. Consider a bank transaction at an Automated Teller Machine (ATM). When a customer decides to transfer money from a savings account to a checking account, two changes must take place in the bank's database records:

n ▪ The savings account must be debited.

n ▪ The checking account must be credited.

If, for any reason, one of these actions cannot be completed, then neither action should take place. Because these actions are so closely related, they should take place within a single transaction.

To start a transaction in a C++Builder application, call a database component's StartTransaction method:

```
DatabaseInterBase->StartTransaction();
```

All subsequent database actions take place in the context of the newly started transaction until the transaction is explicitly terminated by a subsequent call to *Commit* or *Rollback*.

How long should you keep a transaction going? Ideally, only as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit your changes.

Committing a transaction

To make database changes permanent, a transaction must be committed using a database component's *Commit* method. Executing a commit statement saves database changes and ends the transaction. For example, the following statement ends the transaction started in the previous code example:

```
DatabaseInterBase->Commit();
```

Note: *Commit* should always be attempted in a try {}catch(…) statement. If a transaction cannot commit successfully, you can attempt to handle the error, and perhaps retry the operation.

Rolling back a transaction

To discard database changes, a transaction must roll back its changes using *Rollback*. *Rollback* undoes a transaction's changes and ends the transaction. For example, the following statement rolls back a transaction:

```
DatabaseInterBase->Rollback();
```

Rollback usually occurs in:

- Exception handling code when you cannot recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

Using the TransIsolation property

TransIsolation specifies the transaction isolation level for a database component's transactions. Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transaction's changes to a table. Before changing or setting TransIsolation, you should be familiar with transactions and transactions management in C++Builder.

The default setting for *TransIsolation* is *tiReadCommitted*. The following table summarizes possible values for TransIsolation and describes what they mean:

Isolation level	**Meaning**
tiDirtyRead	Permit reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level your transaction is least isolated from the changes made by other transactions.
tiReadCommitted	Permit reading only of committed (permanent) changes made to the database by other simultaneous transactions. This is the default isolation level.
tiRepeatableRead	Permit a single, one time reading of the database. Your transaction cannot see any subsequent changes to data by other simultaneous transactions. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, then C++Builder will use the next highest isolation level. The actual isolation level used by each type of server is listed in the following table. For a detailed description of how each isolation level is implemented, see your server documentation.

***TransIsolation* setting**	**Paradox and dBASE**	**Oracle**	**Sybase and Microsoft SQL servers**	**InterBase**
Dirty read	Dirty read	Read committed	Read committed	Read committed
Read committed (Default)	Not supported	Read committed	Read committed	Read committed
Repeatable read	Not supported	Repeatable read (READ ONLY)	Not supported	Repeatable Read

Note: When using transactions with local Paradox and dBASE tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables

If an application is using ODBC to interface with a server, the ODBC driver must also support the isolation level. For more information, see your ODBC driver documentation.

Using passthrough SQL

To be able to use passthrough SQL to control a transaction you must:

- Use the C++Builder Client/Server Suite.

- Install the proper SQL Links drivers. If you chose the Standard installation when installing C++Builder, all SQL Links drivers are already properly installed.
- Configure your network protocol correctly. See your network administrator for more information.
- Have access to a database on a remote server.
- Use the BDE Configuration utility to set SQLPASSTHROUGHMODE to NOT SHARED.

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

Setting SQLPASSTHROUGHMODE

SQLPASSTHROUGHMODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHROUGHMODE is set to SHARED AUTOCOMMIT. If however, you want to pass SQL transaction control statements to your server, you must use the BDE Configuration utility to set the BDE SQLPASSTHROUGHMODE to NOT SHARED. For more information about SQLPASSTHROUGH modes, see the online help for the BDE Configuration utility.

Note: When SQLPASSTHROUGHMODE is NOT SHARED, you must use separate database components for *TQuery* components that pass SQL transaction statements to the server and those other dataset components that do not.

Using local transactions

The BDE supports local transactions against Paradox and dBASE tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Understanding the limitations of local transactions

The following limitations apply to local transactions

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Transactions cannot be run against temporary tables.
- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:

  Several tables are open.

  The cursor is closed on a table to which no changes were made.

Using TSession

Each time an application runs, C++Builder creates a default *TSession* component for it. A session component provides global control over all database connections in an application. If you do not create multi-threaded database applications, you need only concern yourself with the default session in your application.

A multi-threaded database application is a single application that attempts to run two or more simultaneous operations, such as SQL queries, against the same database. If you create a multi-threaded database application, then each additional thread after the first requires its own session component.

Creating a session component

If you are creating a multi-threaded database application, then you need to create additional session components. To create a session component, place a session component from the Data Access page onto a data module used in your project.

The following table lists the properties of *TSession*, what they are used for, their default values if any, and whether they are available at design time:

Property	Purpose	Default	Design time access
Active	*true*, starts the BDE session. *false*, disconnects datasets and stops the session.	*false*	Yes
Databases	Specifies an array of all active databases in the session.		No
DatabaseCount	Provides an integer value specifying the number of currently active databases in the session.		No
KeepConnections	*true*, maintain database connection(s) even if there are no open datasets. *false*, close database connection(s) when there are no open datasets.	*true*	Yes
Name	Names a session component (for example, Session1).		Yes
NetFileDir	Specifies the directory path of the Paradox network control file, which enables sharing of Paradox tables on network drives.		Yes
PrivateDir	Specifies the path in which to store temporary files (for example, files used to process local SQL statements).		Yes
SessionName	Specifies the name of the session that must be used by database components to link themselves to a particular session.		Yes

Using TSessionList

Multi-threaded applications need to manage sessions through a *TSessionList* component. You never need to create your own session list component, but you may certainly need to manipulate the properties and use the methods of the default session list.

A default session list component, called Sessions, is created for you whenever you start a database application.

Sessions is a component of type *TSessionList*. You use the properties and methods of Sessions to keep track of multiple sessions in a multi-threaded database application. The following table summarizes the properties and methods of the *TSessionList* component:

Property or Method	Purpose
Count	Returns the number of sessions, both active and inactive, in the sessions list.
FindSession	Searches the session list for a session with a specified name, and returns a pointer to the session component, or nil if there is no session with the specified name. If passed a blank session name, FindSession returns a pointer to the default session, Session.
GetSessionNames	Returns a string list containing the names of all currently instantiated session components. This procedure always returns at least one string, 'Default' for the default session (note that the default session's name is actually a blank string).
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value. Sessions is the default property, so you can call it like so Sessions[0], rather than Sessions.Sessions[0].

Specfiying a data source

A *TDataSource* component is a nonvisual database component that acts as a conduit between a dataset and data-aware components on a form that enable the display, navigation, and editing of the data underlying the dataset. All datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form. Similarly, each data-aware control needs to be associated with a data source component in order to have data to display and manipulate. You also use data source components to link datasets in master-detail relationships.

You place a data source component in a data module or form just as you place other nonvisual database components. You should place at least one data source component for each dataset component in a data module or form.

Using TDataSource properties

TDataSource has only a few published properties. The following sections discuss these key properties and how to set them at design time and run time.

Setting the DataSet property

The *DataSet* property specifies the name of the dataset from which the a data source component gets its data. At design time you can select a dataset from the drop down list in the Object Inspector. At run time you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the CustSource data source component between Customers and Orders:

```
if (CustSource->DataSet == "Customers")
CustSource->DataSet = "Orders";
else
CustSource->DataSet = "Customers";
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on the two forms. For example:

```
void __fastcall TForm2::FormCreate (TObject *Sender)
```

```
{
DataSource1->Dataset = Form1->Table1;
}
```

Setting the Name property

Name enables you to specify a meaningful name for a data source component that distinguishes it from all other data sources in your application. The name you supply for a data source component is displayed below the component’s icon in a data module.

Generally, you should provide a name for a data source component that indicates the dataset with which it is associated. For example, suppose you have a dataset called Customers, and that you link a data source component to it by setting the data source component’s *DataSet* property to "Customers." To make the connection between the dataset and data source obvious in a data module, you should set the *Name* property for the data source component to something like "CustomersSource".

Setting the Enabled property

The *Enabled* property determines if a data source component is connected to its dataset. When *Enabled* is *true*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *false*. When *Enabled* is *false*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *true*. It is recommended, however, to control access to a dataset through a dataset component’s *DisableControls* and *EnableControls* methods because they affect all attached data sources.

Setting the AutoEdit property

The *AutoEdit* property of *TDataSource* specifies whether datasets connected to the data source automatically enter Edit state when the user starts typing in data-aware controls linked to the dataset. If *AutoEdit* is *true* (the default), C++Builder automatically puts the dataset in Edit state when a user types in a linked data-aware control. Otherwise, a dataset enters Edit state only when the application explicitly calls its *Edit* method.

Using TDataSource methods

TDataSource has three event handlers associated with it:

- OnDataChange
- OnStateChange
- OnUpdateData

Using the OnDataChange event

OnDataChange is called whenever the cursor moves to a new record. When an application calls *Next*, *Previous*, *Insert*, or any method that leads to a change in the cursor position, then an *OnDataChange* is triggered.

This event is useful if an application is keeping components synchronized manually.

Using the OnUpdateData event

OnUpdateData is called whenever the data in the current record is about to be updated. For instance, an *OnUpdateData* event occurs after *Post* is called, but before the data is actually posted to the database.

This event is useful if an application uses a standard (non-data aware) control and needs to keep it synchronized with a dataset.

Using the OnStateChange event

OnStateChange is called whenever the state for a data source’s dataset changes. A dataset’s *State* property records its current state. *OnStateChange* is useful for performing actions as a *TDataSource’s* state changes.

For example, during the course of a normal database session, a dataset's state changes frequently. To track these changes, you could write an OnStateChange event handler that displays the current dataset state in a label on a form. The following code illustrates one way you might code such a routine. At run time, this code displays the current setting of the dataset's State property and updates it every time it changes:

```
void __fastcall TForm1::StateChange(TObject *Sender)
{
char S[10];

switch (CustTable->State)
  {
  case dsInactive: strcpy(S,"Inactive");
  case dsBrowse: strcpy(S, "Browse");
  case dsEdit: strcpy(S, "Edit");
  case dsInsert: strcpy(S, "Insert");
  case dsSetKey: strcpy(S, "SetKey");
  }
}
```

Similarly, *OnStateChange* can be used to enable or disable buttons or menu items based on the current state:

```
void __fastcall TForm1::StateChange( TObject *Sender)
{
if (CustTable->State == dsBrowse)
  CustTableEditBtn->Enabled;
if (CustTable->State == (dsInsert|dsEdit|dsSetKey))
  CustTableCancelBtn->Enabled;
...
}
```

Displaying data in a data control

To display data in a data control:

1. Place a data control from the Data Access page of the Component palette onto a form.
2. Set the *DataSource* property of the control to the name of a data source component from which to get data. A data source component acts as a conduit between the control and a dataset containing data.
3. Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property.

If the *Enabled* property of the data control's data source is *true* (the default), and the *Active* property of the dataset attached to the data source is also *true*, data is now displayed in the data control.

Note: Two data controls, TDBGrid and TDBNavigator, access all available field components within a dataset, and therefore do not have *DataField* properties. For these controls, omit Step 3.

Enabling mouse, keyboard, and timer events

The *Enabled* property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *true*.

To prevent mouse, keyboard, or timer events from accessing a data control set its *Enabled* property to *false*. When *Enabled* is *false*, a data source does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Enabling editing in controls on user entry

A dataset must be in dsEdit state to permit editing to its data. The *AutoEdit* property of the data source to which a control is attached determines if the underlying dataset enters dsEdit mode when data in a control is modified in response to keyboard or mouse events. When AutoEdit is *true* (the default), dsEdit mode is set as soon as editing commences. If *AutoEdit* is *false*, you must provide a TDBNavigator control with an Edit button (or some other method) to permit users to set dsEdit state at run time.

Updating fields

The *ReadOnly* property of a data control determines if a user can edit the data displayed by the control. If *false* (the default), users can edit data. To prevent users from editing data in a control, set *ReadOnly* to *true*.

Properties of the data source and dataset underlying a control also determine if the user can successfully edit data with a control and can post changes to the dataset.

The *Enabled* property of a data source determines if controls attached to a data source are able to display fields values from the dataset, and therefore also determine if a user can edit and post values. If *Enabled* is *true* (the default), controls can display field values.

The *ReadOnly* property of the dataset determines if user edits can be posted to the dataset. If *false* (the default), changes are posted to the dataset. If *false*, the dataset is read-only.

Note: Table components have an additional, read-only run-time property *CanModify* that determines if a dataset can be modified. *CanModify* is set to *true* if a database permits write access. If *CanModify* is *false*, a dataset is read-only. Query components that perform inserts and updates are, by definition, able to write to an underlying database, provided that your application and user have sufficient write privileges to the database itself.

The following table summarizes the factors that determine if a user can edit data in a control and post changes to the database:

Data control ReadOnly property	Data source Enabled property	Dataset ReadOnly property	Dataset CanModify property (tables only)	Database write access	Can write to database?
false	*true*	*false*	*true*	Read/Write	Yes

false	*true*	*false*	*false*	Read-only	No
false	*false*	—	—	—	No
true	—	—	—	—	No

In all data controls except TDBGrid, when you modify a field, the modification is copied to the underlying field component in a dataset when you Tab from the control. If you press Esc before you Tab from a field, C++Builder abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In TDBGrid, modifications are copied only when you move to a different record; you can press Esc in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, C++Builder checks all data-aware components associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, C++Builder raises an exception, and no modifications are made to the record.

Disabling and enabling data display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's *EnableControls* method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a try{} while statement so that you can re-enable controls even if an exception occurs during processing. The catch clause should call *EnableControls*. The following code illustrates how you might use *DisableControls* and *EnableControls* in this manner:

```
CustTable->DisableControls();
try
  {
  CustTable->First(); // Go to first record, which sets EOF False
  while(!CustTable->DB_EOF) //Cycle until EOF is True
     {
       // Process each record here
     }
  CustTable->Next(); // EOF False on success; EOF True when Next fails  on
  last record
  CustTable->EnableControls();
  }
catch(...)
{
CustTable->EnableControls();
}
```

Refreshing data

The *Refresh* method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application.

Important

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

Opening and closing datasets

To read or write data in a table or through a query, an application must first open a dataset. There are two ways to open a dataset:

- Set the *Active* property of the dataset to *true*, either at design time in the Object Inspector, or in code at run time:

  ```
  CustTable.Active := true;
  ```

- Call the *Open* method for the dataset at run time:

  ```
  CustQuery.Open;
  ```

There are also two ways to close a dataset:

- Set the *Active* property of the dataset to *false*, either at design time in the Object Inspector, or in code at run time:

  ```
  CustQuery.Active := false;
  ```

- Call the *Close* method for the dataset at run time:

  ```
  CustTable.Close;
  ```

You need to close a dataset when you want to change any of its properties that affect the query, such as the *DataSource* property. At run time you may also want to close a dataset for other reasons specific to your application.

Setting dataset states

The state—or mode—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is dsInactive, meaning that nothing can be done to its data. A dataset is always in one state or another. At run time you can examine a dataset's read-only State property to determine its current state. The following table summarizes possible values for the State property and what they mean:

Value	**State**	**Meaning**
dsInactive	Inactive	Dataset closed. Its data is unavailable.
DsBrowse	Browse	Dataset open. Its data can be viewed, but not changed. This is the default state of an open dataset.
DsEdit	Edit	Dataset open. The current row can be modified.
DsInsert	Insert	Dataset open. A new row can be inserted.
DsSetKey	SetKey	TTable only. Dataset open. Enables searching for rows based on indexed fields, or indicates that a SetRange operation is under way. A restricted set of data can be viewed, and no data can be changed.
DsCalcFields	CalcFields	Dataset open. Indicates that an OnCalcFields event is under way. Prevents changes to fields that are not calculated.
DsUpdateNew	UpdateNew	Internal use only.
DsUpdateOld	UpdateOld	Internal use only.
DsFilter	Filter	Dataset open. Indicates that a filter operation in under way. A restricted set of data can be viewed, and no data can be changed.

When an application opens a dataset, C++Builder automatically puts the dataset into dsBrowse mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- Code in your application, or
- C++Builder's built-in behavior.

To put a dataset into dsBrowse, dsEdit, dsInsert, or dsSetKey states, call the method corresponding to the name of the state. For example, the following code puts CustTable into dsInsert state, accepts user input for a new record, and writes the new record to the database:

```
CustTable->Insert(); // Your application explicitly sets dataset state to
  Insert
AddressPromptDialog->ShowModal();
if (AddressPromptDialog->ModalResult == IDOK)
  CustTable->Post(); // Ebony sets dataset state to Browse on successful
  completion
else
  CustTable->Cancel(); // Ebony sets dataset state to Browse on cancel
```

This example also illustrates that C++Builder automatically sets the state of a dataset to dsBrowse when

- The *Post* method successfully writes a record to the database. (If *Post* fails, the dataset state remains unchanged.)
- The *Cancel* method is called.

Some states cannot be set directly. For example, to put a dataset into dsInactive state, set its *Active* property to *false*, or call the *Close* method for the dataset. The following statements are equivalent:

```
CustTable->Active = false;
CustTable->Close();
```

The remaining states (dsCalcFields, dsUpdateNew, dsUpdateOld, and dsFilter) cannot be set by your application. Instead, C++Builder sets them as necessary For example, dsCalcFields is set when a dataset's *OnCalcFields* event is called. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

Note: Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset.

dsInactive

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time a dataset is closed until you set its *Active* property to *true*. At run time, a dataset is closed until it is opened either by calling the *Open* method, or by setting the *Active* property to *true*. When you open an inactive dataset, C++Builder automatically puts it into dsBrowse state.

To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in dsEdit or dsInsert modes when an application calls *Close*, you should prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
void __fastcall CustForm::CustTableBeforeClose(TDataSet *DataSet)
{
bool posted;
if ((CustTable->State == dsEdit) || (CustTable->State == dsInsert))
  if (::MessageBox(0, "Post changes before closing?", "", MB_YESNO) == IDYES)
     {
     CustTable->Post();
     posted = true;
     }
else
  {
  CustTable->Cancel();
  posted = false;
  }
}
```

To associate a procedure with the BeforeClose event for a dataset at design time:

1 Select the table in the data module (or form).

2 Click the Events page in the Object Inspector.

3 Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

dsBrowse

When an application opens a dataset, C++Builder automatically puts the dataset into dsBrowse state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use dsBrowse to scroll from record to record in a dataset.

From dsBrowse all other dataset states can be set. For example, calling the *Insert* or *Append* methods for a dataset changes its state from dsBrowse to dsInsert (note that other factors and dataset properties such as *CanModify*, may prevent this change). Calling *SetKey* to search for records puts a dataset in dsSetKey mode.

Two methods associated with all datasets can return a dataset to dsBrowse state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to dsBrowse state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to dsBrowse state. If *Post* fails, the current state remains unchanged.

dsEdit

A dataset must be in dsEdit mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into dsEdit mode if the read-only *CanModify* property for the dataset is *true*. *CanModify* is *true* if the database underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into dsEdit state if:

- The control's *ReadOnly* property is *false* (the default),
- The *AutoEdit* property of the data source for the control is *true*, and
- *CanModify* is *true* for the dataset.

Important

For TTable components only, if the *ReadOnly* property is *true*, *CanModify* is *false*, preventing editing of records.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from dsEdit state to dsBrowse state in code by calling the *Cancel*, *Post*, or *Delete* methods. Cancel discards edits to the current field or record. *Post* attempts to write a modified record to the dataset, and if it succeeds, returns the dataset to dsBrowse. If *Post* cannot write changes, the dataset remains in dsEdit state. *Delete* attempts to remove the current record from the dataset, and if it succeeds, returns the dataset to dsBrowse state. If *Delete* fails, the dataset remains in dsEdit state.

Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

dsInsert

A dataset must be in dsInsert mode before an application can add new records. In your code you can use the *Insert* or *Append* methods to put a dataset into dsInsert mode if the read-only *CanModify* property for the dataset is *true*. *CanModify* is *true* if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into dsInsert state if

- The control's *ReadOnly* property is *false* (the default),
- The *AutoEdit* property of the data source for the control is *true*, and
- *CanModify* is *true* for the dataset.

Important

For *TTable* components only, if the *ReadOnly* property is *true*, *CanModify* is *false*, preventing editing of records.

Note

Even if a dataset is in *dsInsert* state, inserting records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from dsInsert state to dsBrowse state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Delete* and *Cancel* discard the new record. *Post* attempts to write the new record to the dataset, and if it succeeds, returns the dataset to dsBrowse. If *Post* cannot write the record, the dataset remains in dsInsert state.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

dsSetKey

dsSetKey mode applies only to *TTable* components. To search for records in a *TTable* dataset, the dataset must be in dsSetKey mode. You put a dataset into dsSetKey mode with the SetKey method at run time. The *GotoKey* method, which carries out the actual search, returns the dataset to dsBrowse state upon completion of the search.

Note: Other search methods, including *FindKey*, and *FindNearest* automatically put a dataset into *dsSetKey* state during a search, and return the dataset to *dsBrowse* state upon completion of the search.

dsCalcFields

C++Builder puts a dataset into dsCalcFields mode whenever an application calls the dataset's *OnCalcFields* event handler. This state prevents modifications or additions to the records in a dataset except for the calculated fields the handler modifies itself. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the OnCalcFields handler finishes, the dataset is returned to dsBrowse state.

dsFilter

C++Builder puts a dataset into dsFilter mode whenever an application calls the dataset's *OnFilterRecord* event handler. This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not invalidated.

When the *OnFilterRecord* handler finishes, the dataset is returned to dsBrowse state.

Using the Fields editor

The Fields editor enables you to create, delete, arrange, and define persistent field components associated with a dataset. Persistent field components guarantee your application receives a consistent view of the data underlying a dataset.

To start the Fields editor:

- Double-click the dataset component, or
- Select the component, right-click to invoke the SpeedMenu, then choose Fields editor.

The Fields editor contains a title bar, navigator buttons, and a list box.

The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. So, if you open the Customers dataset in the CustomerData data module, the title bar displays 'CustomerData.Customers', or as much of the name as fits.

Below the title bar is a set of navigation buttons that enable you to scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty.

The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.

Creating persistent field components

To create a persistent field component for a dataset:

- Right-click the Fields editor list box.
- Choose Add fields from the context menu. The Add Fields dialog box appears.

The Available fields list box displays all fields in the dataset which do not have persistent field components. Select the fields for which to create persistent field components, and click OK.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, C++Builder no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, C++Builder verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, C++Builder raises an exception warning you that the field is not valid, and does not open the dataset.

Deleting persistent field components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table.To remove one or more persistent field components for a dataset:

1. Select the field(s) to remove in the Fields editor list box.
2. Press *Del*.

Note: You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always re-create persistent field components that you delete by accident, but any changes previously made to its properties or events is lost.

Note: If you remove all persistent field components for a dataset, then C++Builder again generates dynamic field components for every column in the database table underlying the dataset.

Arranging the order of persistent field components

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of a single field:

1. Select the field.
2. Drag it to a new location.

Alternatively, you can select the field, and use Ctrl-Up and Ctrl-Dn to move the field to a new location in the list.

To change the order for a block of fields:

1. Select the fields.
2. Drag them to their new location.

If you select a non-contiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block the order of fields to one another does not change.

Defining new persistent field components

Besides selecting which dynamic field components to make into persistent field components for a dataset, you can also create new persistent fields as additions to or replacements of the other persistent fields in a dataset. There are three types of persistent fields you can create:

- *Data fields*, which usually replace existing fields (for example to change the data type of a field), are based on columns in the table or query underlying a dataset.
- *Calculated fields*, which displays values calculated at run time by a dataset's OnCalcFields event handler.
- *Lookup fields*, which retrieve values from a specified dataset at run time based on search criteria you specify.

These types of persistent fields are only for display purposes. The data they contain at run time are not retained either because they already exist elsewhere in your database, or because they are temporary. The physical structure of the table and data underlying the dataset is not changed in any way.

To create a new persistent field component:

1. Right-click the Fields editor list box.
2. Choose New field from the SpeedMenu.

   The New Field dialog box appears.

Dialog box options

New Field dialog box

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

Field type radio group

The Field type radio group box enables you to specify the type of new field component to create. The default type the first time you open the New Field dialog box in a session is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. The Lookup definition group box is only used to create lookup fields.

Field properties group box

The Field properties group box enables you to enter general field component information. Enter the component's field name in the Name edit box. The name you enter here corresponds to the field component's FieldName property. C++Builder uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's Name property and is only provided for informational purposes (Name contains the identifier by which you refer to the field component in your source code). C++Builder discards anything you enter directly in the Component edit box.

Type combo box

The Type combo box in the Field properties group enables you to specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating point currency values in a field, select Currency from the drop-down list. The Size edit box enables you to specify the maximum number of characters that can be displayed or entered in a string-based field or the size of Bytes and VarBytes fields. For all other data types, Size is meaningless.

Defining a data field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field’s data type directly, you must define a new field to replace it.

Important

Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a data field in theNew Field dialog box:

1. Enter the name of an existing persistent field in the Name edit box. Do not enter a new field name.
2. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Data in the Field type radio group if it is not already selected.
5. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form’s **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Defining and programming a calculated field

A calculated field displays values calculated at run time by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Calculated in the Field type radio group.
5. Choose OK. The newly defined calculated field is automatically added to end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's class declaration in the source code.
6. Place code that calculates values for the field in the *OnCalcFields* event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field," below.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector.

Programming a calculated field

After you define a calculated field, you must write code to calculate its value. Otherwise it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

1. Select the dataset component from the Object Inspector drop-down list.
2. Choose the Object Inspector Events page.
3. Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a CityStateZip calculated field for the Customers table on the CustomerData data module. CityStateZip should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the Customers table, select the Customers table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The TCustomerData.CustomersCalcFields procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip->Value = CustomersCity->Value + ", " + CustomersState-
  >Value + "  " + CustomersZip->Value;
```

Defining a lookup field

A lookup field displays values it searches for at run time based on search criteria. In its simplest form, a lookup field is passed the name of a field to search, a field value to search for, and the field in the lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator use a lookup field to determine automatically the city and state that correspond to a zip code a customer provides. In that case, the column to search on might be called ZipTable.Zip, the value to search for is the customer's zip code as entered in Order.CustZip, and the values to return would be those in the ZipTable.City and ZipTable.State columns for the record where ZipTable.Zip matches the current value in the Order.CustZip field.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at run time. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. To specify more than one field, enter field names directly instead. Separate multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating. To return values from more than one field in the lookup dataset, enter field names directly instead. Separate multiple field names with semicolons.

Creating Data Dictionary attribute sets for field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

To create an attribute set based on a field component in a dataset:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the Object Inspector.
4. Right-click the Fields editor list box to invoke the context menu.
5. Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing **Save attributes as** instead of **Save attributes** from the context menu.

Note: You can also create attribute sets directly from the SQL/Database Explorer. When you create an attribute set from the data dictionary, the set is not applied to any fields, but you can specify two additional attributes in the set: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) that is automatically placed on a form when a field based on the attribute set is dragged onto the form.

Associating Data Dictionary attribute sets with field components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment, DisplayWidth, DisplayFormat, EditFormat, MaxValue, MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Right-click the Fields editor list box to invoke the context menu.
4. Choose Associate attributes.
5. Select or enter the attribute set to apply from the Attribute set name dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

Unassociating a Data Dictionary attribute set from a field component

If you change your mind about associating an attribute set with a field component, you can easily remove the attribute set from the field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to remove an attribute set.
3. Right-click the Fields editor list box to invoke the SpeedMenu.
4. Choose Unassociate attributes.

After you remove an attribute set from a field component, you can either use the Object Inspector to set its properties, or you can associate a different attribute set with the component.

Accessing field values with the default dataset method

The preferred method for accessing a field's value is to use variants with the default dataset method, *FieldValues*. For example, the following statement puts the value of an edit box into the CustNo field in the Customers table:

```
Customers->FieldByName("CustNo")->AsString = Edit2->Text;
```

Because FieldValues is the default method for a dataset, you do not need to specify its method name explicitly. The following statement, however, is identical to the previous one:

```
Customers->FieldValues["CustNo"] = Edit2->Text;
```

For more information about variants, see the *C++ Language Reference*.

Accessing field values with the Fields property

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. Accessing field values with a dataset's *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the **Fields** property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using the field component's conversion routine. For more information about field component conversion functions, see "Using field component conversion functions."

For example, the following statement assigns the current value of the seventh column (Country) in the Customers table to an edit control:

```
Edit1->Text = CustTable->Fields[6]->AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
{
Customers->Edit();
Customers.Fields[6]->AsString = Edit1->Text;
Customers->Post();
}
```

Accessing field values with the FieldByName method

You can also access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion function, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the CustNo field in the Customers dataset to an edit control:

```
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

Conversely, you can assign a value to a field:

```
{
  Customers->Edit();
  Customers->FieldByName("CustNo")->AsString = Edit2->Text;
  Customers->Post();
}
```

Using conversion functions

Conversion functions attempt to convert one data type to another. For example, the *AsString* function converts numeric and Boolean values to string representations. The following table lists field component conversion functions, and which functions are recommended for field components by field-component type:

Function	TStringField	TIntegerField	TSmallintField	TWordField	TFloatField	TCurrencyField	TBCDField	TDateTimeField	TDateField	TTimeField	TBooleanField	TBytesField	TVarBytesField	TBlobField	TMemoField	TGraphicField
AsVariant	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
AsString		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
AsInteger	3				3	3	3									
AsFloat	3	3	3	3				3	3	3						
AsCurrency	3	3	3	3				3	3	3						
AsDateTime	3															
AsBoolean	3															

Note that the *AsVariant* method is recommended to translate among all data types. When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. The following table lists permissible conversions that produce special results:

Conversion	Result
String to Boolean	Converts "*true*," "*false*," "Yes," and "No" to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to "*true*" or "*false*."

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

You use a conversion function as you would use any method belonging to a component: append the function name to the end of the component name wherever it occurs in an assignment statement.

Conversion always occurs before an actual assignment is made. For example, the following statement converts the value of CustomersCustNo to a string and assigns the string to the text of an edit control:

```
Edit1->Text = CustomersCustNo->AsString;
```

Conversely, the next statement assigns the text of an edit control to the CustomersCustNo field as an integer:

```
CustomersCustNo->AsInteger = StrToInt(Edit1->Text);
```

An exception occurs if an unsupported conversion is performed at run time.

Deploying an application

To make an application available to end users, you deploy it. Deployment involves preparing a package for your end users that enables them to install and run your application and access data with it. C++Builder ships with InstallShield Express, a product that helps you deploy your applications.

A deployment package usually includes your application executable file, and support files, such as DLLs, and data files. Particularly when you are upsizing an application to use a remote server, there are additional files and DLLs you need to deploy. For example, to access remote database servers, your application must use the appropriate Borland SQL Links drivers and support files.

Before you deploy any application you build with C++Builder, see the online text file DEPLOY.TXT for the latest information about deployment and licensing.

SQL/Database Explorer

The SQL/Database Explorer enables you to maintain a persistent connection to a remote database server during application development and to work with BDE aliases and metadata objects. With the SQL/Database Explorer, you can create, view, and modify:

- BDE aliases.
- Metadata objects such as tables, views, triggers, and stored procedures.
- Users and server security information.

The All Database Aliases pane of the SQL/Database Explorer displays all valid, defined aliases. Select an alias to display the its definition.

To connect to the database specified by an alias,

1. Select the alias in the All Database Aliases pane.
2. Do one of the following,

- Choose Object|Open.
- From the context menu, choose Open.

When you are connected to a database, the icon in the left pane will turn green.

To expand a database,

- In the All Database Aliases pane, click "+” next to the alias you want to view. The native server object types expand beneath the icon.

Once connected to a database, you can perform SQL operations on the database.

To perform SQL operations,

1. Select the Enter SQL tab.
2. Enter SQL statements in the statement area.
3. Click on the Execute button.

Your SQL statements will execute and the results will be displayed in the table grid.

SQL Monitor

The SQL Monitor enables you to see the actual statement calls made through SQL Links to a remote server or through the ODBC socket to an ODBC data source.

To open the SQL Monitor,

- Choose Database|SQL Monitor.

You can elect to monitor different types of activities. Choose Options|Trace Categories to select different categories of activities to monitor. You can monitor any number of the following categories:

Category	Displays
Prepared Query Statements	Prepared statements to be sent to the server.
Executed Query Statements	Statements to be executed by the server. Note that a single statement may be prepared once and executed several times with different parameter bindings.
Statement Operations	Each operation performed such as ALLOCATE, PREPARE, EXECUTE, and FETCH.
Connect / Disconnect	Operations associated with connecting and disconnecting to databases, including allocation of connection handles, freeing connection handles, if required by server.
Transactions	Transaction operations such as BEGIN, COMMIT, and ROLLBACK (ABORT).
Blob I/O	Operations on Blob datatypes, including GET BLOB HANDLE, STORE BLOB, and so on.
Miscellaneous	Operations not covered by other categories.
Vendor Errors	Error messages returned by the server. The error message may include an error code, depending on the server.
Vendor Calls	Actual API function calls to the server. For example, ORLON for Oracle, ISC_ATTACH for InterBase.

Menu title area
Menu titles display in this area. Click the highlighted block to add new items to the menu.

Menu command area
Menu commands display in this area. Click the highlighted block to add new menu commands.