

Getting Started with Delphi® and C++Builder® 2009

Rapid Application Development Using the
Delphi® and C++Builder® Integrated Development Environment



EMBARCADERO
TECHNOLOGIES®

By the RAD Studio Team at Embarcadero Technologies, Inc.

2009 04 16



Getting Started with
Delphi® and C++Builder®
2009

© 2009 Embarcadero Technologies, Inc. Embarcadero, the Embarcadero Technologies logos, and all other Embarcadero Technologies product or service names are trademarks or registered trademarks of Embarcadero Technologies, Inc. All other trademarks are property of their respective owners.

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, CodeGear™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.

Contents

Chapter 1

Introduction	1-1
What is RAD Studio?	1-1
Finding information	1-2

Chapter 2

Tour of the IDE	2-1
First Look.	2-1
Welcome Page	2-2
Toolbars	2-4
Tools.	2-5
Accessibility options	2-6
Form Designer.	2-6
Tool Palette.	2-8
Object Inspector.	2-10
Project Manager.	2-11
File Browser.	2-12
Structure View.	2-13
The Code Editor.	2-15
Code Navigation.	2-16
Method Hopping.	2-16
Bookmarks.	2-16
Finding Classes.	2-17
Finding Units.	2-17
Code Folding.	2-17
Change Bars.	2-18
Block Comments.	2-18
Live Templates.	2-18
SyncEdit.	2-19
Code Insight.	2-20
Code Completion.	2-20
Help Insight.	2-20
Class Completion.	2-21
Block Completion.	2-21
Code Parameter Hints.	2-21
Code Hints.	2-22
Error Insight	2-22
Code Browsing.	2-22
Refactoring.	2-23
Keystroke Macros.	2-23
To-Do Lists.	2-24
Custom Template Libraries.	2-24
History Manager.	2-24
Data Explorer.	2-26

Chapter 3

Starting your first RAD Studio application	3-1
Using project templates from the Object Repository	3-2
Basic customization of the main form.	3-4
Adding the components using the Form Designer.	3-6
Adding an Action Manager.	3-6
Adding the main menu.	3-6
Adding a status bar	3-6
Adding a text box.	3-7
Adding the main menu commands.	3-8
Defining Action properties.	3-9
Adding word wrap and font capabilities.	3-10
Customizing the components.	3-11
Coding responses to user actions in the Code Editor.	3-13
Creating an event handler for the New command.	3-14
Creating the event handlers for the Open command.	3-17
Creating the event handlers for the SaveAs command.	3-18
Creating the event handlers for the Save command.	3-19
Creating the event handler for the Font command.	3-20
Creating the event handler for the Word Wrap command	3-20
Creating event handlers for the status bar.	3-21
Compiling and running the application.	3-22
Debugging the application	3-23

Chapter 4

More advanced topics	4-1
VCL and RTL	4-1
Third party add-ins.	4-4

Chapter 5

Other resources	5-1
EDN.	5-1
Partners.	5-2

List of figures

- Figure 1-1. The **Trace Into** menu item in **Run** 1-3
- Figure 1-2. **F1** Help for the **Trace Into** menu item 1-3
- Figure 2-1. The RAD Studio Welcome Page 2-1
- Figure 2-2. The default layout for creating a RAD Studio application 2-3
- Figure 2-3. The main menu bar 2-3
- Figure 2-4. The main toolbars 2-4
- Figure 2-5. Shortcut keys in the **File** menu 2-4
- Figure 2-6. Customizing the toolbars 2-5
- Figure 2-7. Creating a basic RAD Studio application using the Form Designer 2-6
- Figure 2-8. The Tool Palette showing the **Standard** components category 2-8
- Figure 2-9. The Tool Palette in Code Editor mode, showing the standard **Delphi Projects** templates 2-9
- Figure 2-10. **Properties** tab in the Object Inspector 2-10
- Figure 2-11. **Events** tab in the Object Inspector 2-10
- Figure 2-12. Customizing the Object Inspector 2-10
- Figure 2-13. Hierarchical file list of the project, displayed by the Project Manager 2-11
- Figure 2-14. Browsing files and folders using File Browser 2-12
- Figure 2-15. Setting up the file filter used in the File Browser 2-12
- Figure 2-16. Structure View in Form Designer mode 2-13
- Figure 2-17. Structure View in Code Editor mode 2-13
- Figure 2-18. Structure View explorer options 2-14
- Figure 2-19. Setting Bookmarks in the source code 2-17
- Figure 2-20. Collapsed blocks of code 2-17
- Figure 2-21. Expanding the list of Live Templates for Delphi 2-18
- Figure 2-22. Highlighting all the occurrences of an identifier in a section of code 2-19
- Figure 2-23. Code Completion popup window showing the list of available options 2-20
- Figure 2-24. Using Code Parameter Hints to show the required types for the parameters 2-21

List of Figures

- Figure 2-25.** Displaying in-place Code Hints 2-22
- Figure 2-26.** Automatic marking of errors in the code 2-22
- Figure 2-27.** Comparing two versions of a file using the **Diff** page 2-25
- Figure 2-28.** Exploring the list of available database connections 2-27
- Figure 3-1.** Description of all options in the **File** menu 3-2
- Figure 3-2.** Expanding the **New** option in the **File** menu 3-2
- Figure 3-3.** The default layout for creating a RAD Studio application (Delphi view) 3-3
- Figure 3-4.** The default layout for creating a RAD Studio application (C++Builder view) 3-3
- Figure 3-5.** Basic customization of the main form (Delphi view) 3-5
- Figure 3-6.** Basic customization of the main form (C++Builder view) 3-5
- Figure 3-7.** Using the action filter in the Tool Palette to select *TActionManager* 3-6
- Figure 3-8.** Using the status filter in the Tool Palette to select *TStatusBar* 3-7
- Figure 3-9.** Using the memo filter in the Tool Palette to select *TMemo* 3-7
- Figure 3-10.** Basic text editor form 3-7
- Figure 3-11.** The main elements of the **Actions** page in the Action Manager 3-8
- Figure 3-12.** Adding a New Standard Action 3-8
- Figure 3-13.** Selecting the Standard Actions that implement the basic file and text operations 3-9
- Figure 3-14.** Arranging the actions in the **File** menu and finishing adding the Standard Actions 3-9
- Figure 3-15.** The final look of the **File** menu 3-10
- Figure 3-16.** Editing the contents of the memo 3-11
- Figure 3-17.** Panel editor showing the list of added status panels 3-12
- Figure 3-18.** Defining the *CurrentFile* private variable (Delphi view) 3-13
- Figure 3-19.** Defining the *currentFile* private variable (C++Builder view) 3-14
- Figure 3-20.** Opening the **Events** tab in the *Object Inspector* 3-14
- Figure 3-21.** Automatic generation of the code skeleton for the *OnExecute* event (Delphi view) 3-15
- Figure 3-22.** Automatic generation of the code skeleton for the *OnExecute* event (C++Builder view) 3-15
- Figure 3-23.** **Project** menu options for compiling and building the project 3-22
- Figure 3-24.** Dialog showing the success of compiling the application 3-22

- Figure 3-25.** Dialog showing the success of building the application 3-22
- Figure 3-26.** Running the application from the **Run** menu 3-23
- Figure 3-27.** Debugging the *FileSaveAs1Accept* procedure (Delphi code) 3-24
- Figure 3-28.** Debugging the *FileSaveAs1Accept* function (C++Builder code) 3-24
- Figure 3-29.** Application stopping at the specified breakpoint (Delphi view) 3-25
- Figure 3-30.** Dragging the *FileName* variable to the Watch List (Delphi view) 3-25
- Figure 3-31.** Dragging the *fileName* variable to the Watch List (C++Builder view) 3-26
- Figure 3-32.** Advancing to the next line of code to change the value of *FileName* (Delphi view) 3-26
- Figure 3-33.** Advancing to the next line of code to change the value of *FileName* (C++Builder view) 3-27
- Figure 3-34.** Jumping over the **if** statement (Delphi view) 3-27
- Figure 3-35.** Jumping over the **if** statement (C++Builder view) 3-28
- Figure 3-36.** Viewing the value of *CurrentFile* (Delphi view) 3-28
- Figure 3-37.** Viewing the value of *CurrentFile* (C++Builder view) 3-29
- Figure 3-38.** The **Debug** toolbar 3-29

Typographic conventions

Typeface	Meaning
Monospace type	Monospaced type represents text as it appears on screen or in code. It also represents anything you must type.
Boldface	Boldfaced words in text or code listings represent reserved words, compiler options, menus, commands, and dialog boxes.
<i>Italics</i>	Italicized text represents Delphi identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.
Keycaps	Text in keycaps indicates a key on your keyboard. For example, "Press Esc to exit a menu."

Table 1-1. Typographic conventions

Introduction

This guide provides an overview of the CodeGear™ RAD Studio 2009 development environment to get you started using the product right away. It also tells you where to look for details about the tools and features available in RAD Studio.

Chapter 2, “Tour of the IDE” describes the main tools on the CodeGear™ RAD Studio 2009 desktop, or integrated development environment (IDE). Chapter 3, “Starting Your First Visual Application” explains how to use some of these tools to create an application. Chapter 4, “More Advanced Topics” describes the more advanced features in RAD Studio, like VCL, RTL, or the included third party add-ins. Chapter 5, “EDN and Partners”, displays a list of code-related articles on various products and the partners of Embarcadero.

For various examples on using CodeGear™ RAD Studio 2009 to write programs such as a text editor or database application, see the Demos directory of your CodeGear™ RAD Studio 2009 installation, also accessible from the Start menu folder. Other online resources are available at the following address: www.embarcadero.com.

What is RAD Studio?

RAD Studio is an object-oriented, visual programming environment for rapid application development (RAD). Using CodeGear™ RAD Studio 2009, you can create highly efficient visual applications with a minimum of manual coding, using either the Delphi, C++, or Delphi Prism programming languages. To learn about using Prism to create .NET applications, see the Prism Primer at prismwiki.codegear.com.

CodeGear™ RAD Studio 2009 provides all the tools you need to model applications, design user interfaces, automatically generate and edit code, and also the tools needed to compile, debug, and deploy applications. The tools available in the IDE depend on the version of RAD Studio you are using.

Finding information

You can find information about CodeGear™ RAD Studio 2009 in the following ways:

- Web-based product documentation
- *F1* help and online help system

For information...

about new features in this release, refer to the www.embarcadero.com web site.

Web-based product documentation

You can get help online by visiting the www.embarcadero.com web site and navigating to one of the following:

- Developer network—<http://dn.embarcadero.com>—where you can find news and articles about Embarcadero products.
- QualityCentral—<http://qc.embarcadero.com>—where you can read, create, update, or manage reports about issues in the Embarcadero products.
- CodeCentral—<http://cc.embarcadero.com>—where you can find, comment upon, upload, and download code snippets for the Embarcadero products.
- Blogs—<http://blogs.embarcadero.com>—you can find useful information in articles written by the Embarcadero employees.

The <http://docs.embarcadero.com> web site also includes a list of books and additional technical documents for all of the Embarcadero products.

F1 Help and online help system

You can get context-sensitive help in any part of the development environment, including in menu items, in dialog boxes, in toolbars, and in components by selecting the item and pressing *F1*.

Pressing the **F1** key while a menu item is selected displays context-sensitive help for the item. For example, pressing **F1** on the **Trace Into** menu item...

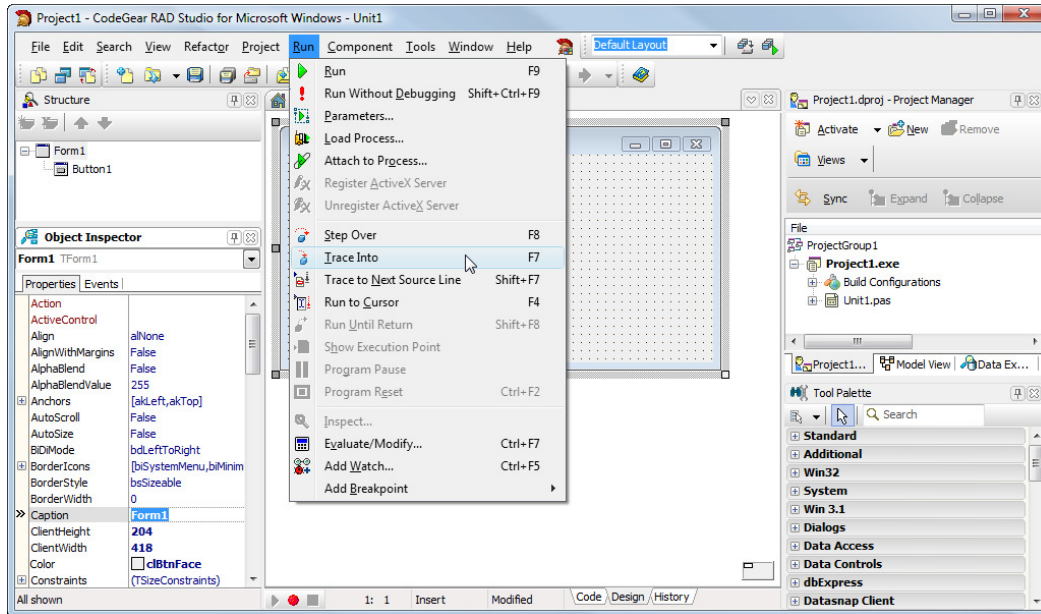


Figure 1-1. The **Trace Into** menu item in **Run**

...displays the following help page.

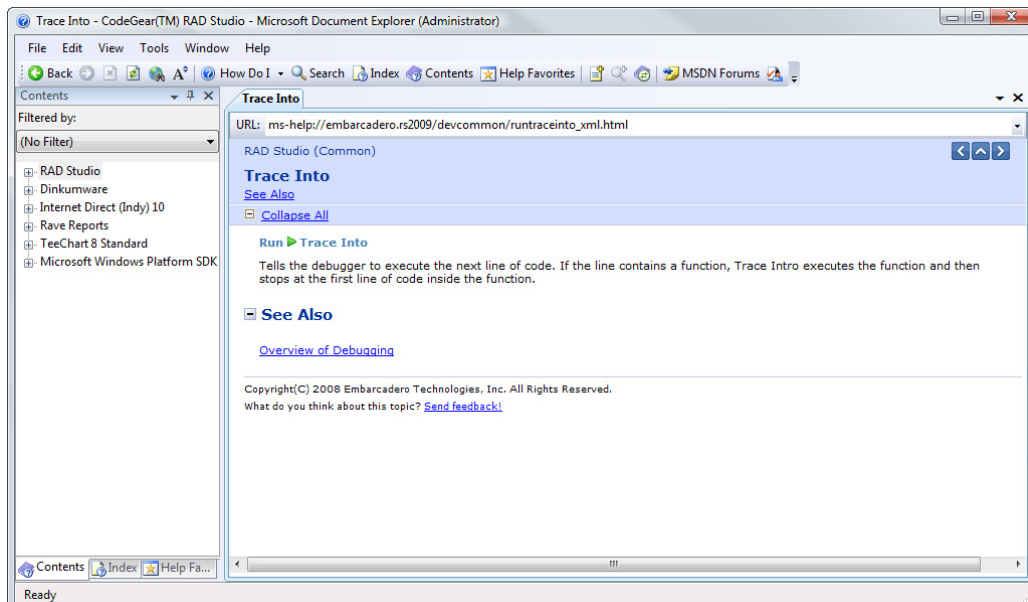


Figure 1-2. **F1** Help for the **Trace Into** menu item

Finding information

Error messages from the compiler and linker appear in a special window below the *Code Editor*. To get help with compilation errors, select a message from the list and press **F1**.

Useful information about using the help viewer can be found at <http://edn.embarcadero.com/article/37562>.

Tour of the IDE

First look

When you start CodeGear™ RAD Studio 2009, the integrated development environment (IDE) launches and displays several tools and menus.

The IDE helps you visually design user interfaces, set object properties, write code, view, and manage your application in various ways.

The default IDE desktop layout includes some of the most commonly used tools. You can use the **View** menu to display or hide certain tools. You can also customize your desktop by moving or deleting elements, and you can save the desktop layouts that work best for you.

Welcome Page

When you open RAD Studio, the *Welcome Page* appears with a number of links to developer resources, such as product-related articles, training, and online Help.

As you develop projects, you can quickly access them from the list of recent projects at the top of the page. To return to the Welcome Page from another main window such as the *Code Editor* or *Design* window, click the Welcome Page tab at the top of the window. If you close the Welcome Page, you can reopen it by choosing **View > Welcome Page**.

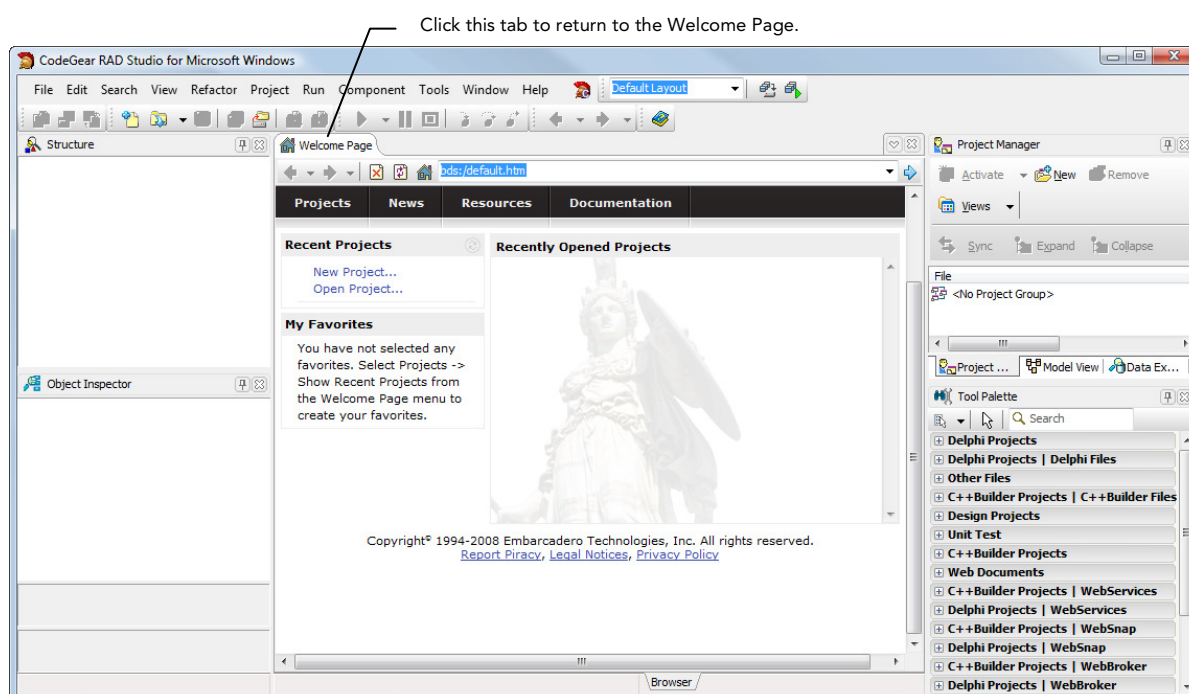


Figure 2-1. The RAD Studio Welcome Page

The following pages describe and show screenshots of the various available options when a RAD Studio project is open. You can create a new project by clicking **File > New > VCL Forms Application - Delphi** or **File > New > VCL Forms Application - C++Builder**, for Delphi and C++Builder, respectively. A more detailed explanation on how to create a project is given in Chapter 3, "Starting your first RAD Studio application".

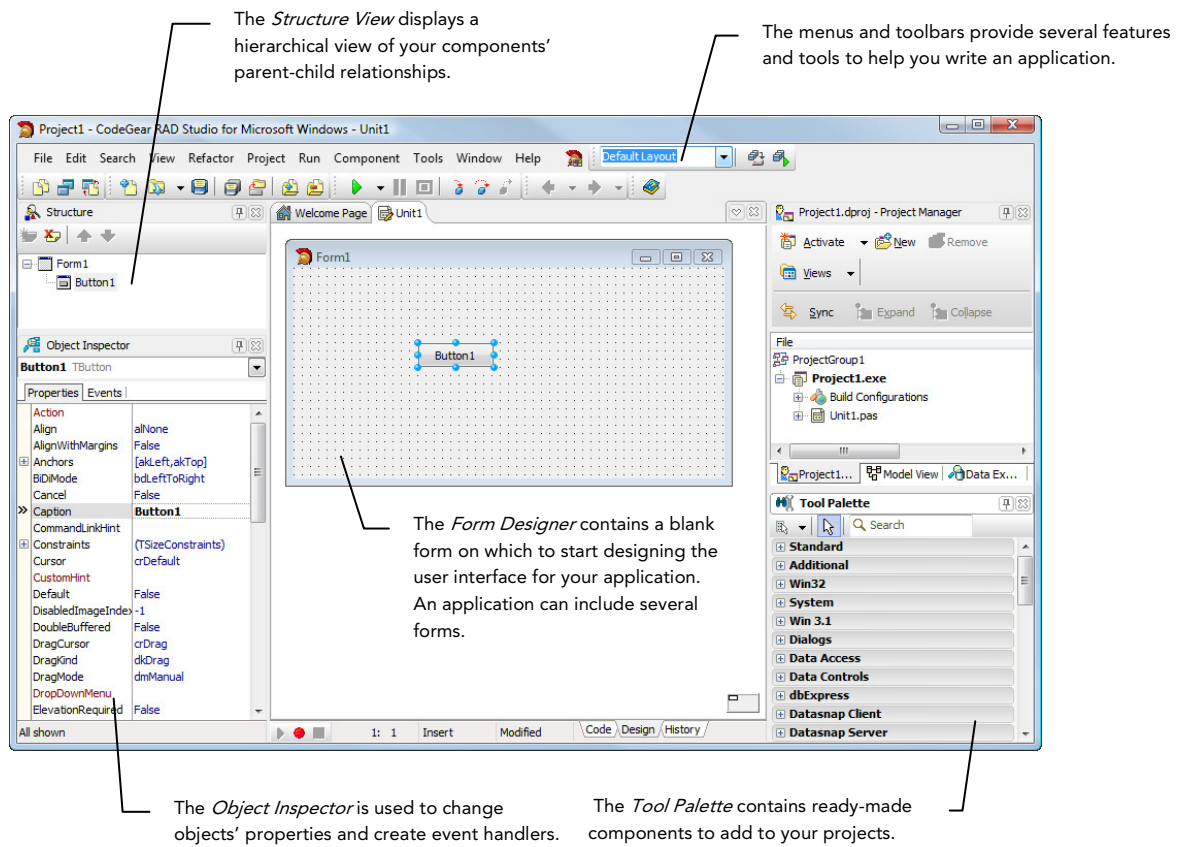


Figure 2-2. The default layout for creating a RAD Studio application

The main window, which occupies the top of the screen, contains the menu bar and the toolbars.

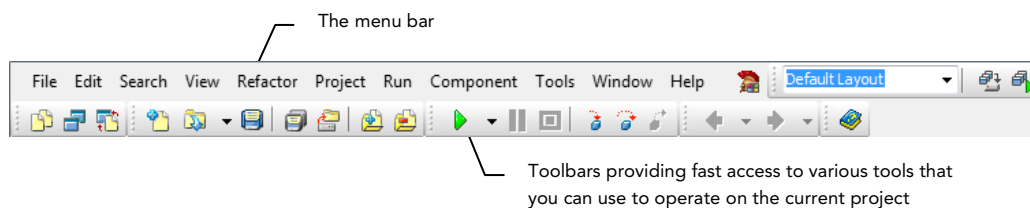


Figure 2-3. The main menu bar

Toolbars

RAD Studio toolbars provide quick access to frequently used operations and commands. The toolbars are displayed below in more detail.

Most toolbar operations are duplicated in the drop-down menus.

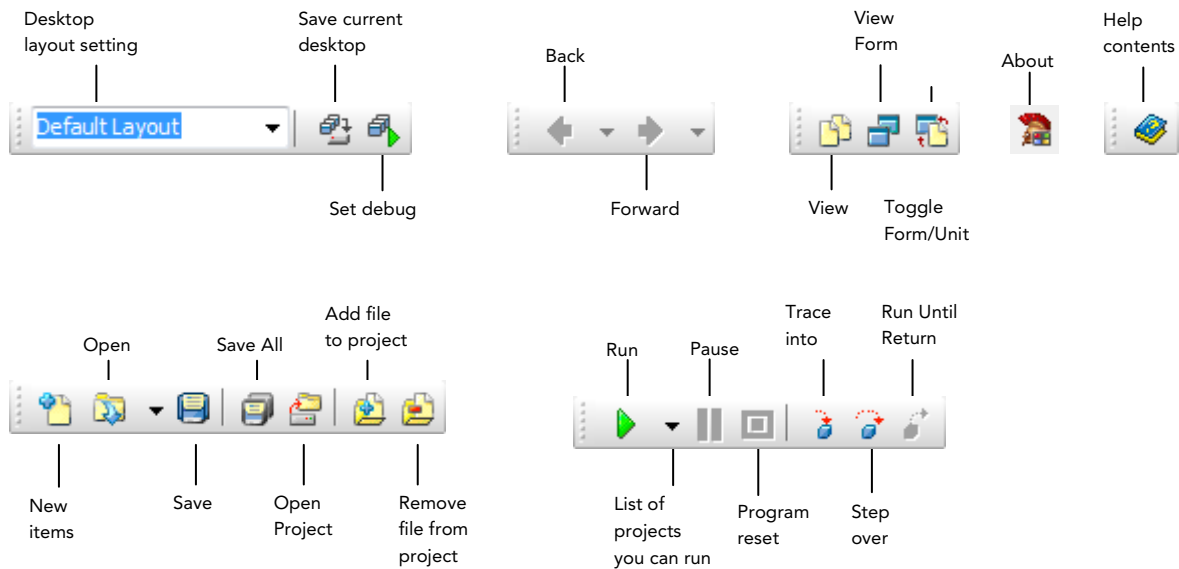


Figure 2-4. The main toolbars

To find out what a button does, hover the mouse over it for a moment until a tooltip appears. You can hide any toolbar by right-clicking the toolbar and selecting the context menu command **Hide**. To display a toolbar that is not showing, choose **View > Toolbars** and check the toolbar you want.

Many operations have keyboard shortcuts as well as toolbar buttons. When a keyboard shortcut is available, the dropdown menu displays the shortcut next to the command.

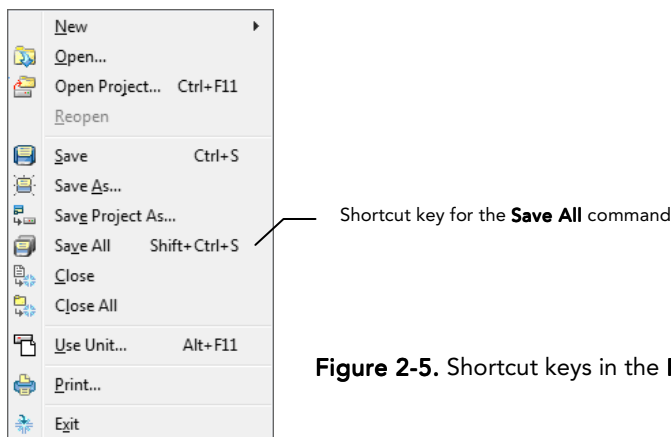


Figure 2-5. Shortcut keys in the **File** menu

You can right-click on many tools and icons to display a menu of commands appropriate to the object you are highlighting. These are called *context menus*. The toolbars are also customizable. You can add the commands you want to the toolbar or move the commands to different locations on the toolbar.

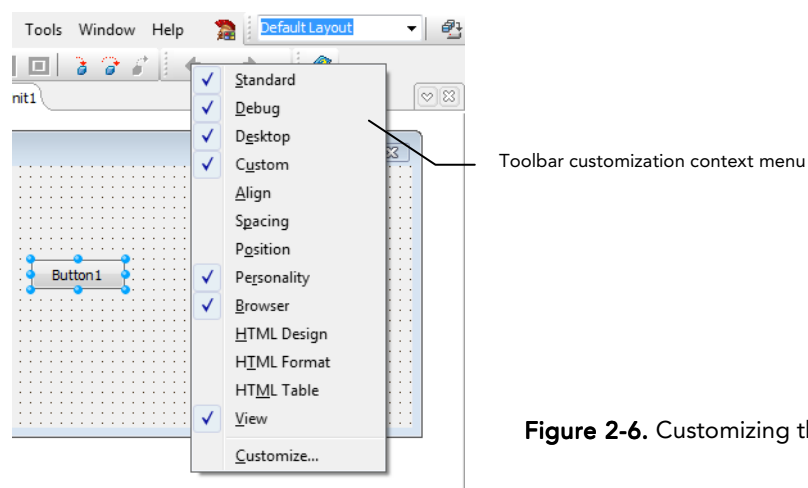


Figure 2-6. Customizing the toolbars

Tools

The tools available in the RAD Studio IDE depend on the version of RAD Studio you are using. Every SKU of RAD Studio contains the following tools:

- Accessibility Options
- Form Designer
- Tool Palette
- Object Inspector
- Project Manager
- Data Explorer
- Structure View
- History Manager
- Code Editor
- File Browser

The following sections describe each of these tools.

Accessibility options

The IDE's main menu supports *MS Active Accessibility* (MSAA). This means that you can use the Windows accessibility tools from the **Start** menu by choosing **All Programs > Accessories > Accessibility**.

Form Designer

The *Form Designer* in RAD Studio allows you to rapidly prototype, build, and modify the user interface of your application. Typically, a form represents a window or an HTML page in the user interface.

Select the form that best suits your application design, whether it is a Web application that provides business logic functionality over the Web or a Windows application that provides processing and high-performance content display.

In RAD Studio, the user interface of an application is built using components that can be either visual or nonvisual and can be added to the form using the Tool Palette, which is discussed in the next section. Visual components appear on your form at the time the program is run. Nonvisual components do not appear on the form at run time, but they change the behavior of your application. Both types of components can be accessed at run time from your application's code.

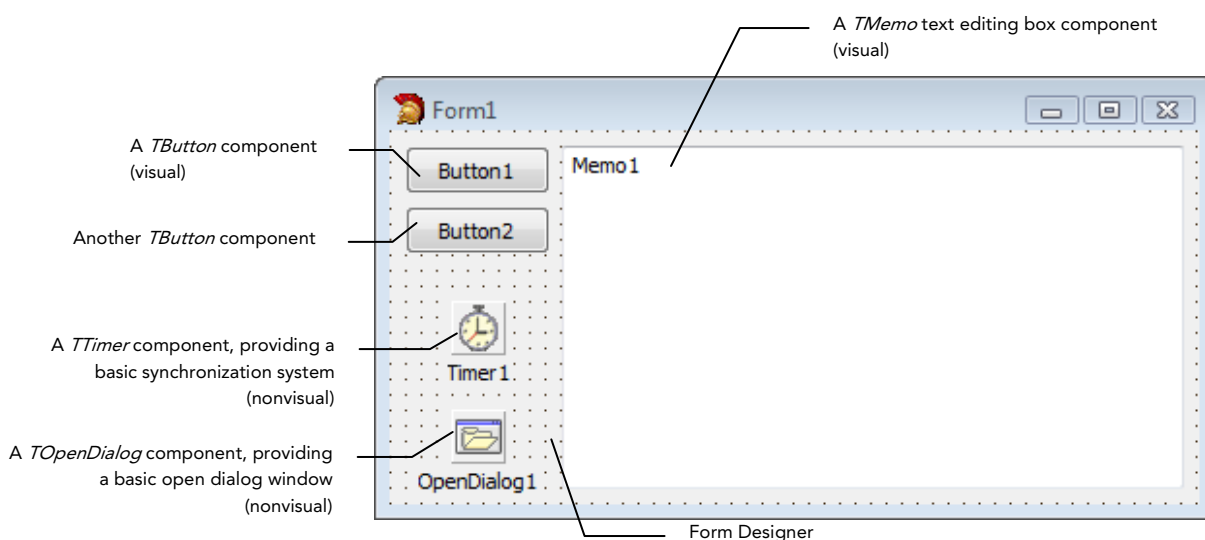


Figure 2-7. Creating a basic RAD Studio application using the Form Designer

The RAD Studio Form Designer is based on the *WYSIWYG* (What You See Is What You Get) concept, allowing you to design your application's user interface with as little effort as possible.

The same concept applies to the possibility of seeing the way components will behave at run time before compiling your applications.

This is the case, for instance, of database-aware components. You can develop live database queries and connections at design time. Database viewing components can show data from a selected database. This way you can check if the behavior at design time will be the intended one at run time.

The Form Designer also allows you to build user interfaces for your VCL for the Web applications, making the development as simple as possible.

To start using the Form Designer, you must first create a VCL for the Web or a VCL for Win32 form using the project templates from the Object repository.

After you place components on the form, or Form Designer, you can arrange them the way they should look on your user interface. Every component's attributes can be viewed and changed with the Object Inspector pane. You can use the Object Inspector for many purposes, including the following:

- To set design-time properties for the components you place on the form.
- To create event handlers, filter-visible properties, and events, making the connection between your application's visual appearance and the code that makes your application run.

For more information...

See "Adding the components using the Form Designer" and "Customizing the components using the Object Inspector" in the next chapter.

Tool Palette

The *Tool Palette* contains items to help you develop your application. The Tool Palette is displayed as a category panel group, usually located in the right column. Each of the categories in Tool Palette contains icons that represent visual or nonvisual components.

The categories divide the components into functional groups. For example, in Design mode, the **Standard** and **Win32** categories include Windows controls such as a button, or an edit box; the **Dialogs** category includes common dialog boxes to use for file operations such as opening and saving files.

The contents of the Tool Palette change when switching between Design mode and Code Editor mode. More information on the Code Editor is given in a later section. Thus, if you are viewing a form in Design mode, the Tool Palette displays components that are appropriate for that form. You can double-click a control to add it to your form. You can also drag a control to a desired position on the form.

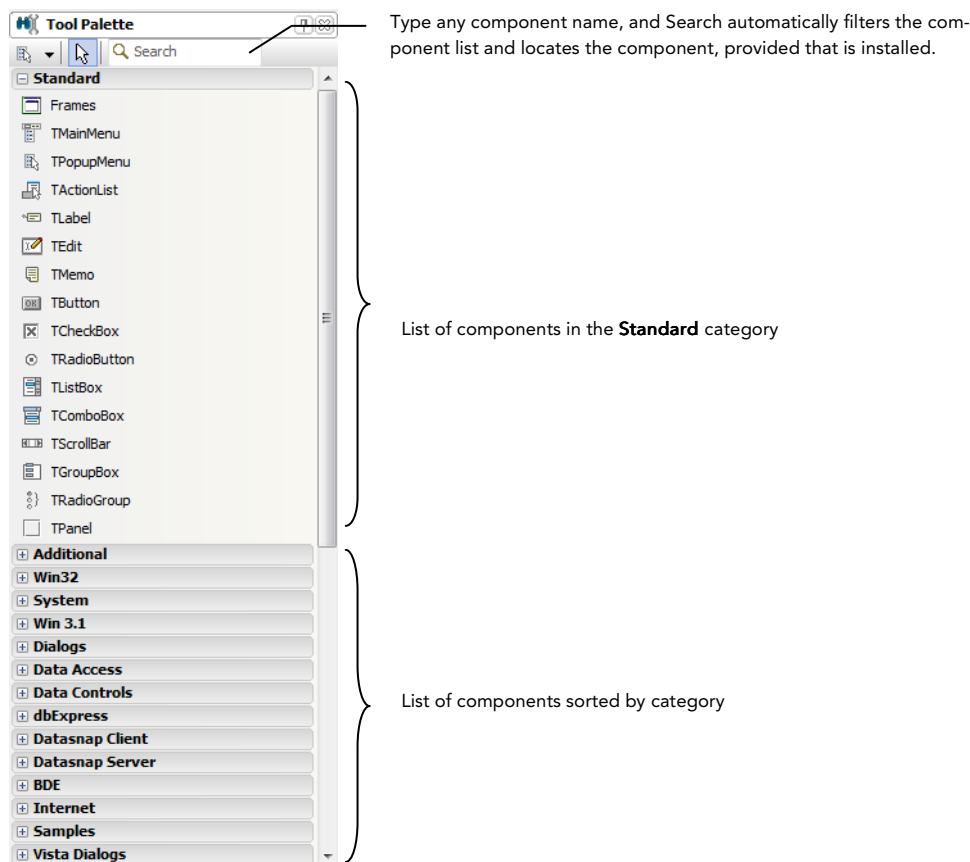


Figure 2-8. The Tool Palette showing the **Standard** components category

If you are viewing code in the Code Editor, the Tool Palette displays project types that you can add to your project group and file types that you can add to your project.

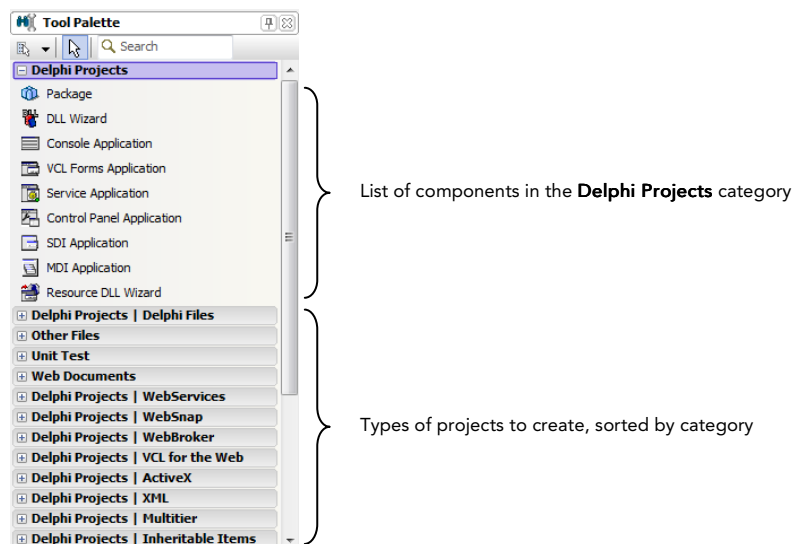


Figure 2-9. The Tool Palette in Code Editor mode, showing the standard **Delphi Projects** templates

In addition to the components that are installed with RAD Studio, you can add customized or third-party components to the Tool Palette and save them in their own category.

You can also create templates that are composed of one or more components. After arranging components on a form, setting their properties, and writing code for them, you can save them as a component template.

Later, by selecting the template from the Tool Palette, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time. You can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

Each component has specific attributes—properties, events, and methods—that enable you to control your application.

After you place components on the form, or Form Designer, you can arrange components the way they should look on your user interface. For the components you place on the form, use the Object Inspector to customize their behavior, as shown in the following section.

Object Inspector

The *Object Inspector* allows you to customize the properties of the components that make up the application user interface and create event handlers at design time. Each visual and nonvisual component has a set of published properties and events, which the Object Inspector displays and allows to be modified visually, using the **Properties** and **Events** tabs. User interfaces created with RAD Studio are event-driven, meaning that any component can react to an externally or internally generated event. The Object Inspector allows you to automatically generate code that is executed when such an event is fired.

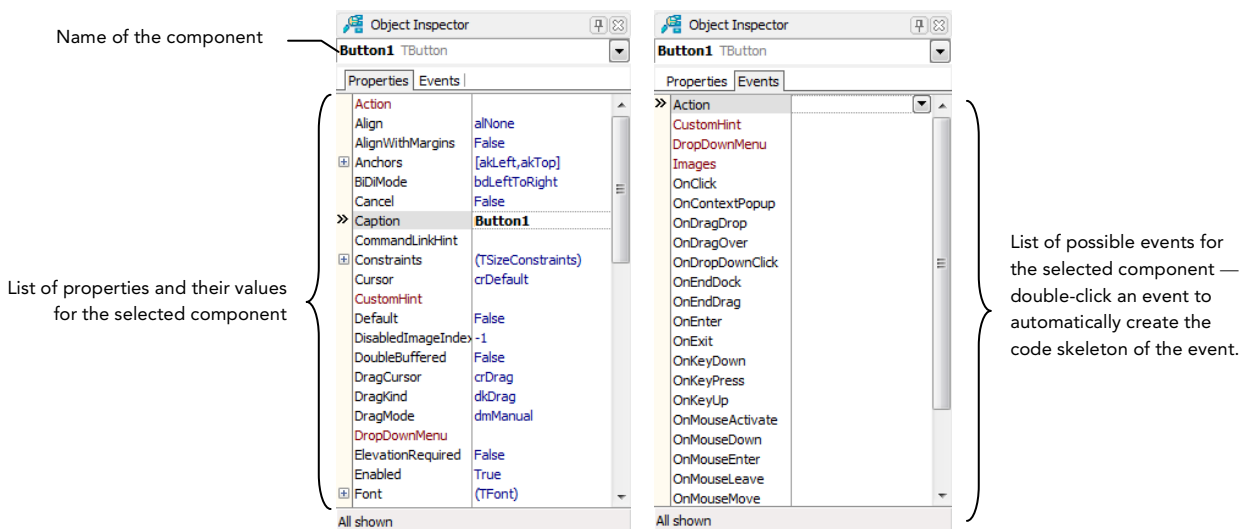


Figure 2-10. Properties tab in the Object Inspector

Figure 2-11. Events tab in the Object Inspector

You can customize the Object Inspector by right-clicking it. A popup menu will be shown with a list of customization options such as the arrangement style of properties or the filtering options.

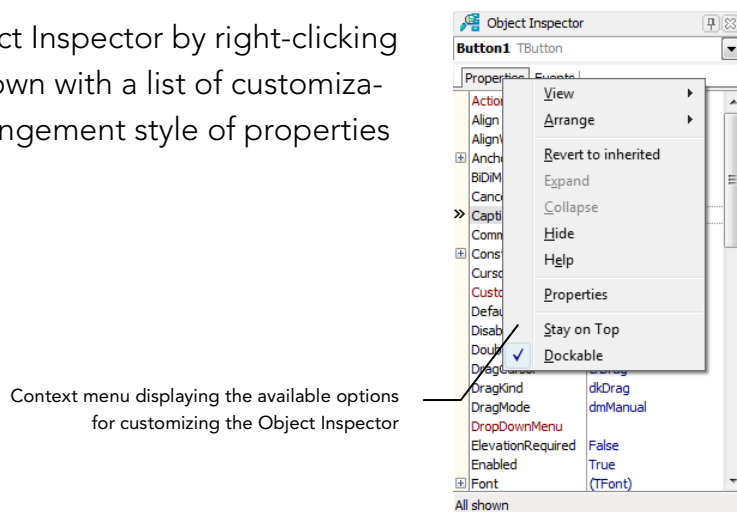


Figure 2-12. Customizing the Object Inspector

Project Manager

To build an application or a DLL using Delphi or C++Builder, you need to create a project. The *Project Manager* displays a hierarchical file list of your project or project group, so you can view and organize the files.

You can use the Project Manager to combine and display information on related projects into a single project group. By organizing related projects into a group, such as multiple executables, you can compile them at the same time. You can also set project options to resolve the dependencies between projects.

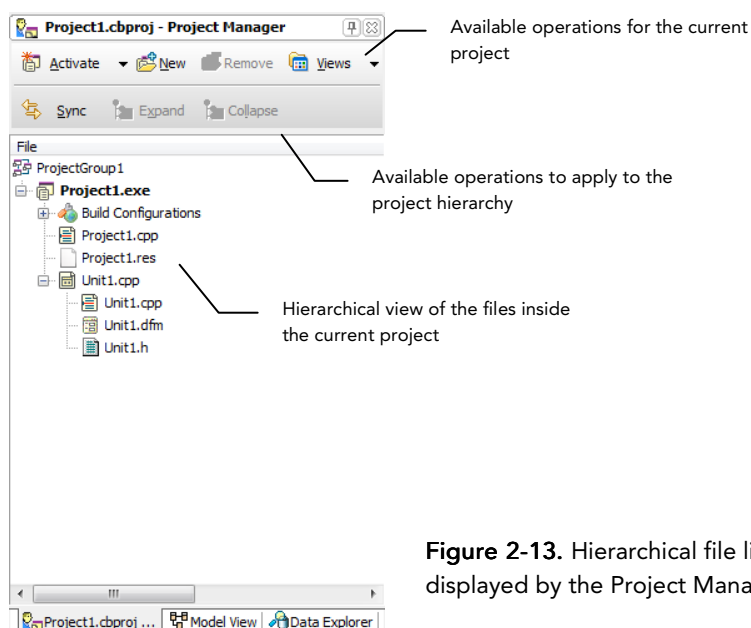


Figure 2-13. Hierarchical file list of the project, displayed by the Project Manager

The buttons at the top of the Project Manager enable you to perform the following tasks:

Activate—Activate the currently selected project.

New—Add another project to the current project group. If only one project currently exists, a project group is created for you automatically.

Remove—Remove a project from the current project group.

View—View the file tree hierarchy in multiple ways.

Sync—Synchronize the project manager with the medium where the actual project or project group files are stored.

Expand—Expand all child nodes from the one that is currently selected.

Collapse—Collapse all child nodes from the one that is currently selected.

File Browser

The *File Browser* allows you to conveniently manage files at a specified path. All the files are displayed in a tree view, allowing easy hierarchical browsing. To show the File Browser, choose **View > File Browser**.

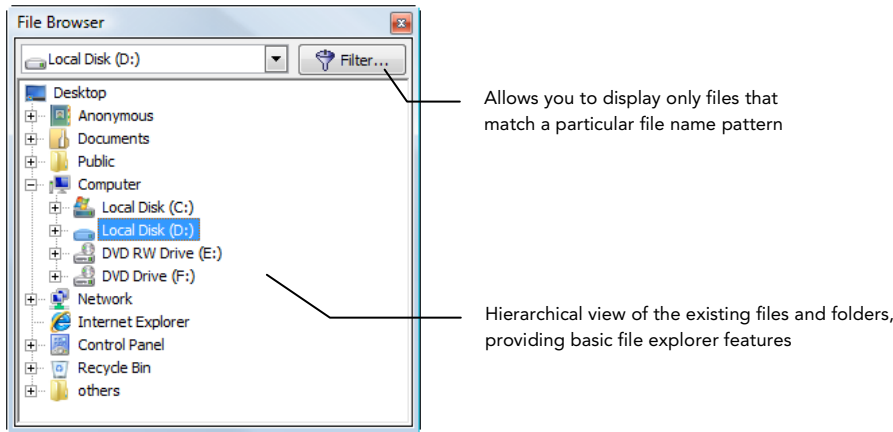


Figure 2-14. Browsing files and folders using File Browser

The File Browser is especially useful for managing files that are not normally part of the project itself, and thus not present in the Project Manager. The context menu shown by the File Browser when a file is right-clicked is based on the Windows Explorer menu, with two new options specific to RAD Studio. These two options allow you to either open the selected files with RAD Studio itself or to add them to the currently open project.

A useful feature of the File Browser is the ability to filter the displayed files based on a set of masks. After clicking the **Filter** button at the top of the File Browser window, a new dialog box asking for a semicolon-separated list of masks appears. As an example, setting the mask to *.txt; *.exe displays only executables and text files.

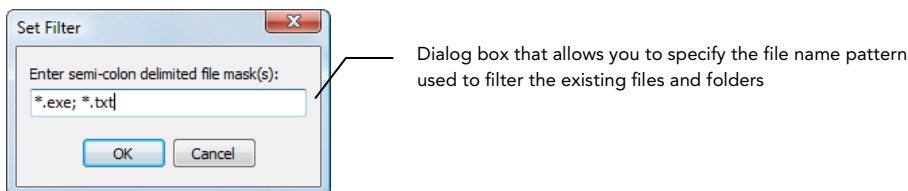


Figure 2-15. Setting up the file filter used in the File Browser

Structure View

The contents of the *Structure View* reflect the current mode in the IDE. The Structure View shows either the hierarchy of the source code that is currently open in the Code Editor, or the components currently displayed in the Designer. The tree diagram is synchronized with the Object Inspector and the Form Designer so that when you change mode in the Structure View, the mode also changes for both the Object Inspector and the form itself.

To display the Structure View, choose **View > Structure**.

If the Structure View is displaying the structure of Designer components, you can single-click a component in the tree diagram to focus it on the form.

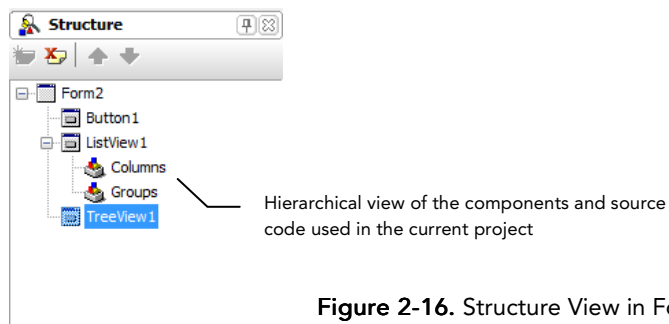


Figure 2-16. Structure View in Form Designer mode

If displaying the structure of source code or HTML, you can double-click an item in the list to jump to its declaration or location in the Code Editor.

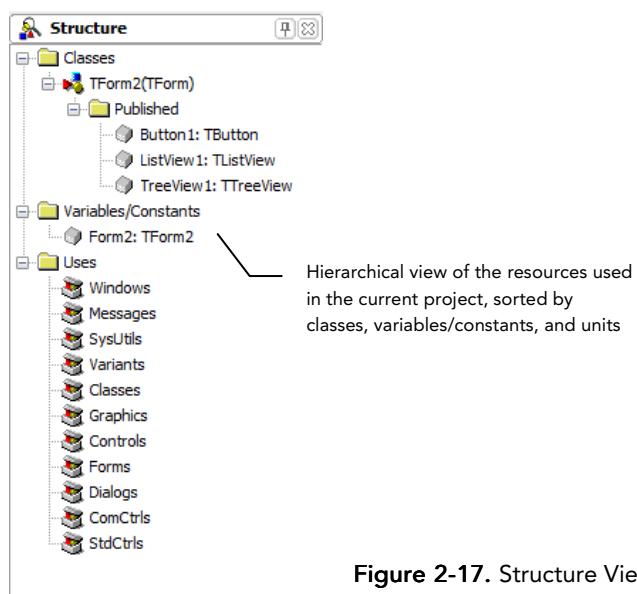


Figure 2-17. Structure View in Code Editor mode

Structure View

If your code contains syntax errors, they are displayed in the Errors pane of the Structure View. To locate an error in the Code Editor, double-click it in the Errors pane.

You can also use the Source View to change related components' relationships. For example, if you add a panel and a check box component to your form, the two components are siblings. In the Structure View, however, if you drag the check box on top of the panel icon, the check box becomes the child of the panel.

The Structure View is useful for displaying the relationships between database objects.

You can also double-click any Designer component in the Structure View to open the Code Editor to a place where you can write an event handler for that component.

You can control the content and appearance of the Structure View by choosing **Tools > Options > Environment Options > Explorer** and changing the settings. The following options page should be displayed.

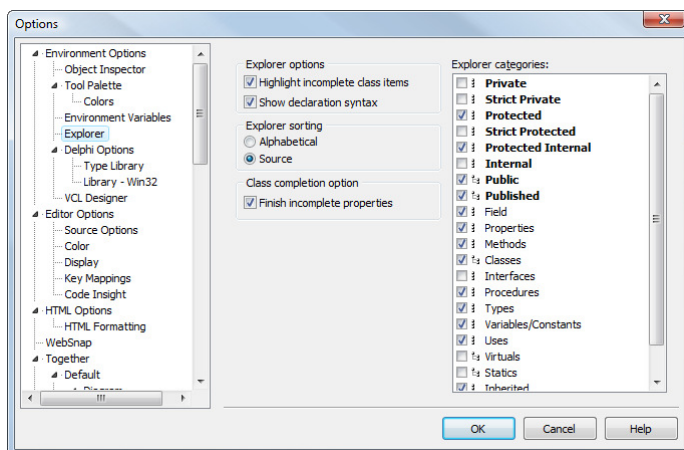


Figure 2-18. Structure View explorer options

For more information...

Access the "Structure View" help page.

The Code Editor

The Code Editor occupies the center pane of the IDE window. The Code Editor is a full-featured, customizable, UTF8 editor that provides syntax highlighting, source code browsing, multiple-undo capability, and context-sensitive Help for language elements.

As you design the user interface for your application, RAD Studio generates portions of the underlying code. Whenever you modify the properties of an object, your changes are automatically reflected in the source files.

Because all of your programs share common features, RAD Studio auto-generates code to get you started. You can think of the auto-generated code as an outline that you can use to create your program.

The Code Editor provides the following features to help you write code:

- Code Navigation
 - Method Hopping, Bookmarks, Finding Classes, Finding Units
- Code Folding
- Change Bars
- Block Comments
- Live Templates
- SyncEdit
- Code Insight
 - Code Parameter Hints, Code Hints, Help Insight, Class Completion, Block Completion, Error Insight, Code Browsing
- Refactoring
- Keystroke Macros
- To-Do Lists
- Custom Template Libraries

Code Navigation

To navigate code while you are using the Code Editor, you can use one of the following methods.

Method Hopping

You can navigate between methods using a series of editor hotkeys. You can also limit hopping to the methods of the current class by setting class lock.

For example, if class lock is enabled and you are in a method of *TComponent*, then hopping is only available within the methods of *TComponent*. The keyboard shortcuts for *Method Hopping* are as follows.

Keyboard shortcut	Effect
CTRL+Q followed by L	toggles class lock
CTRL+ALT+HOME	moves to the first method in the file
CTRL+ALT+END	moves to the last method in the file
CTRL+ALT+MOUSE WHEEL	scrolls through methods

Table 2-1. Method Hopping keyboard shortcuts

Bookmarks

Bookmarks provide a convenient way to navigate long files. You can mark a location in your code with a Bookmark and jump to that location from anywhere in the file.

You can use up to ten Bookmarks, numbered 0 through 9, within a file. To toggle a Bookmark at the current line, simply  press **CTRL+SHIFT+digit**.

When you set a Bookmark, a book icon is displayed in the left gutter of the Code Editor, as in the following image.

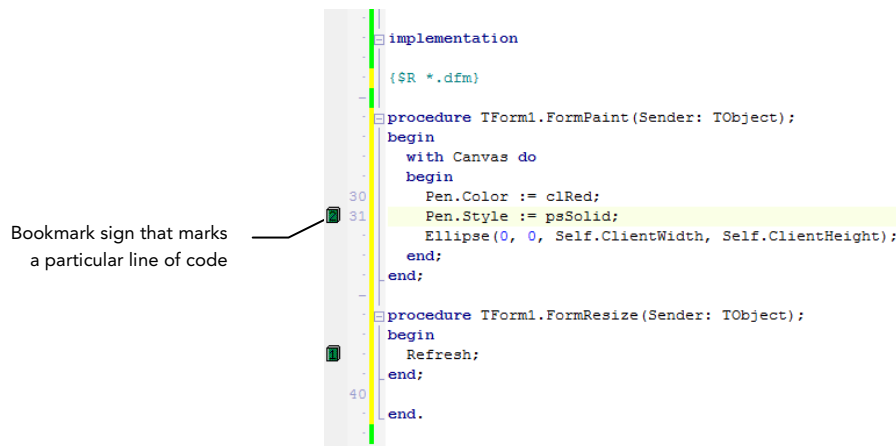


Figure 2-19. Setting Bookmarks in the source code

Finding Classes

Use the **Search > Find Class** command to see a list of available classes that you can select. If you double-click a class, the IDE automatically navigates to its declaration.

Finding Units

Depending on your programming language, you can use a refactoring feature to locate namespaces or units. If you are using the Delphi language, you can use the **Search > Find Unit** command to locate and add units to your code file. The Find Type window allows regular expressions.

Code Folding

Code Folding lets you collapse sections of code to create a hierarchical view of your code and to make it easier to read and navigate. To create Code Folding regions, see the help topic "Using Code Folding".

Code Folding regions have plus (+) and minus (-) signs located on the gutter of the Code Editor, used to collapse and expand a region of code, respectively.

The collapsed code is not deleted, but hidden from view.

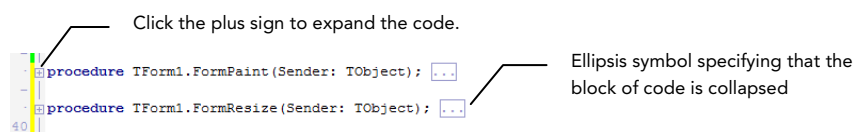


Figure 2-20. Collapsed blocks of code

Change Bars

The left margin of the Code Editor displays a yellow change bar to indicate lines that have been changed but not yet saved in the current editing session. A green change bar indicates the changes that have been made since the last **File > Save** operation. You can, however, customize the Change Bars to display in colors other than the default green and yellow. To do this, go to **Tools > Options > Editor Options > Color**, select the **Modified Line** element and change the foreground and the background colors.

Block Comments—**CTRL+ /**

You can comment-out a section of code by selecting the code in the Code Editor and pressing **CTRL+ /** (slash). Each line of the selected code is then prefixed with `//` and is ignored by the compiler. Pressing **CTRL+ /** adds or removes the slashes, based on whether the first line of the selected code is prefixed with `//`.

When using the Visual Studio or Visual Basic key mappings, use **CTRL+K+C** to add and remove comment slashes.

Live Templates

Live Templates allow you to have a dictionary of pre-written code that can be inserted into your programs while you are working with the Code Editor. You can access Live Templates by going to **View > Templates**.

Using Live Templates reduces the amount of typing that you must do.

You can find further information concerning Live Templates in the help, by accessing the pages entitled “Creating Live Templates” and “Using Live Templates”.

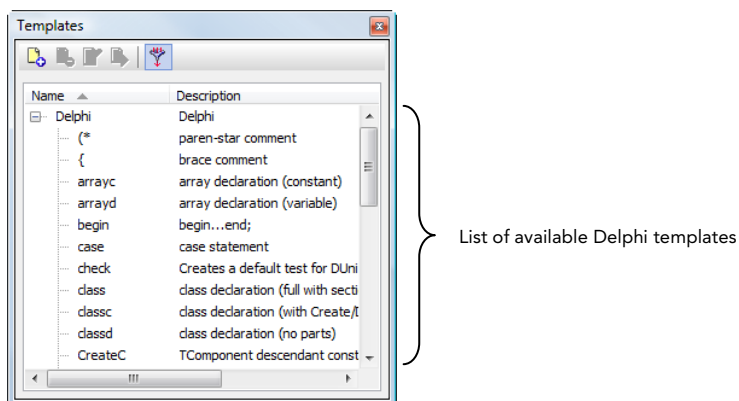


Figure 2-21. Expanding the list of Live Templates for Delphi

SyncEdit

The *SyncEdit* feature lets you simultaneously edit identical identifiers in the code.

As you change the first identifier, the same change is performed automatically on the other identifiers. For example, in a procedure that contains three occurrences of *Self*, you can edit the first occurrence only and all the other occurrences will change automatically.

To use SyncEdit:

1. In the Code Editor, select a block of code that contains identical identifiers.

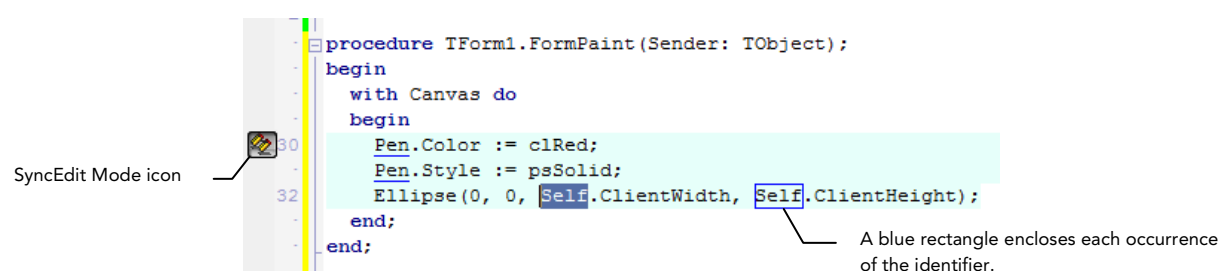



Figure 2-22. Highlighting all the occurrences of an identifier in a section of code

2. Click the SyncEdit Mode icon  that appears in the left gutter. The first identical identifier is highlighted and the others are outlined. The cursor is positioned on the first identifier. If the code contains multiple sets of identical identifiers, you can press **TAB** to move between each identifier in the selection.
3. Begin editing the first identifier. As you change the identifier, the same change is performed automatically on the other identifiers. By default, the identifier is replaced. To change the identifier without replacing it, use the arrow keys before you begin typing.
4. When you have finished changing the identifiers, you can exit Sync Edit mode by clicking the SyncEdit Mode icon, or by pressing the **Esc** key.

Note: SyncEdit determines identical identifiers by matching text strings; it does not analyze the identifiers. For example, it does not distinguish between two like-named identifiers of different types in different scopes. Therefore, SyncEdit is intended for small sections of code, such as a single method or a page of text. For changing larger portions of code, consider using refactoring, which is a more advanced method of improving your code, without changing its functionality. Further details on refactoring are given in a later section.

Code Insight

Code Insight refers to a subset of features embedded in the Code Editor (such as Code Parameter Hints, Code Hints, Help Insight, Code Completion, Class Completion, Block Completion, and Code Browsing) that aid in the code writing process. These features help identify common statements you want to insert into your code, and assist you in the selection of properties and methods. Some of these features are described in more detail in the following sub-sections.

To enable and configure Code Insight features, choose **Tools > Options > Editor Options** and click Code Insight.

Code Completion—**CTRL+SPACE**

To invoke *Code Completion*, press **CTRL+SPACE** while using the Code Editor. A popup window displays a list of symbols that are valid at the cursor location. You can type characters to match those in the selection and press **Return** to insert the text in the code at the cursor location.

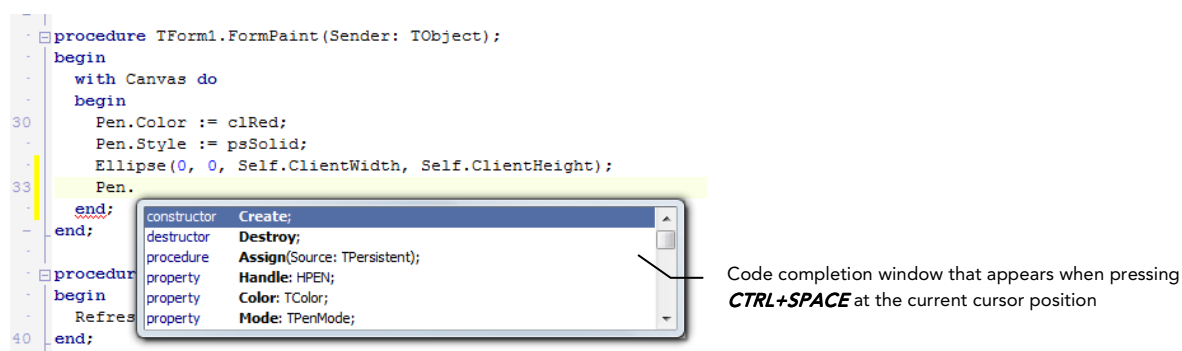


Figure 2-23. Code Completion popup window showing the list of available options

Help Insight—**CTRL+SHIFT+H**

Help Insight displays a hint containing information about the symbol, such as type, file, line number where declared, and any XML documentation associated with the symbol (if available).

Invoke Help Insight by hovering the mouse over an identifier in your code, while working in the Code Editor. You can also invoke Help Insight by pressing **CTRL+SHIFT+H**.

Class Completion—**CTRL+SHIFT+C**

Class Completion simplifies the process of defining and implementing new classes by generating skeleton code for the class members that you declare.

Position the cursor within a class declaration in the interface section of a unit and press **CTRL+SHIFT+C**. Any unfinished property declarations are completed.

For any methods that require an implementation, empty methods are added to the implementation section.

Class Completion can also be achieved by choosing the option **Complete class at cursor** from the editor context menu.

Block Completion—**ENTER**

When you press **ENTER** in a block of code that was incorrectly closed (while working in the Code Editor), a closing block token is inserted at the first empty line following the cursor position.

For instance, if you are using the Code Editor with the Delphi language, and you type the token `begin` and then press **ENTER**, the Code Editor automatically completes the statement so that you have: `begin end;`

Block Completion also works for the C++ language.

Code Parameter Hints—**CTRL+SHIFT+SPACE**

Code Parameter Hints display a hint containing argument names and types for method calls. You can invoke Code Parameter Hints by pressing **CTRL+SHIFT+SPACE**, after opening a left bracket of a function call.

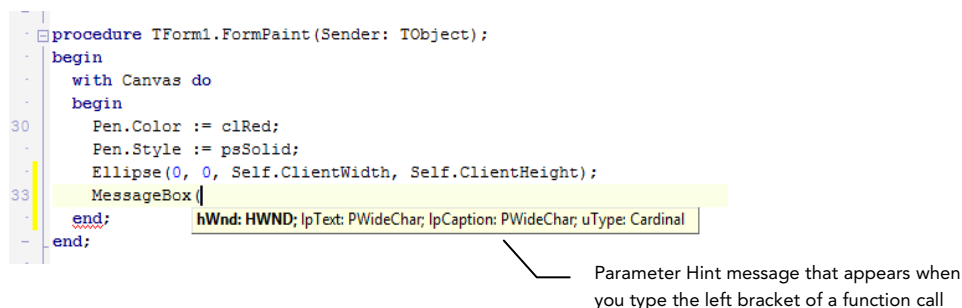


Figure 2-24. Using Code Parameter Hints to show the required types for the parameters

Code Hints

Code Hints display a hint containing information about the symbol such as type, file, and line number where declared. You can display Code Hints by hovering the mouse over an identifier in your code, while working in the Code Editor.

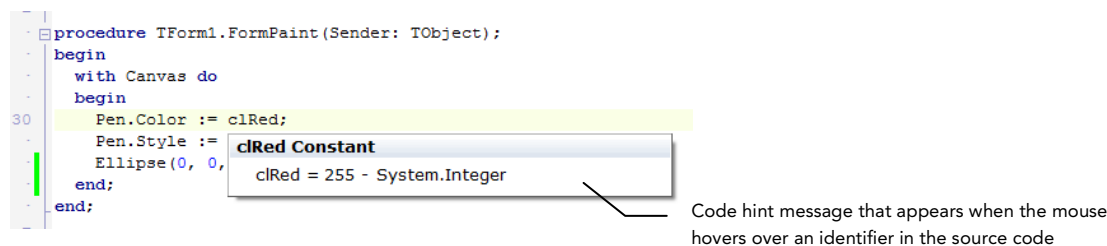


Figure 2-25. Displaying in-place Code Hints

Note: Code Hints only work when the Help Insight feature is disabled.

Error Insight

When you type an expression that generates compiler errors, the expression is underlined in red.

Also, the list of errors generated by the expression appears in the Errors pane of the Structure View.

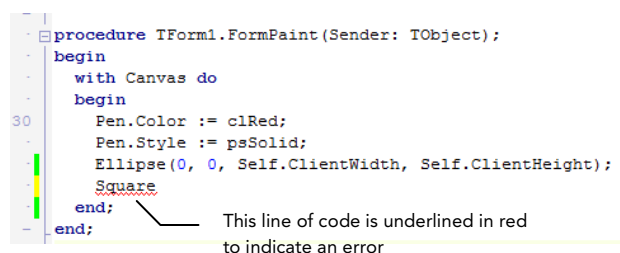


Figure 2-26. Automatic marking of errors in the code

Code Browsing—*CTRL+Click*

While using the Code Editor to edit a VCL Form application, you can hold down the **CTRL** key while hovering the mouse over the name of any class, variable, property, method, or other identifier.

The mouse pointer turns into a hand and the identifier appears highlighted and underlined. Click the identifier and the Code Editor jumps to the declaration of the identifier, opening the source file, if necessary. You can do the same thing by right-clicking an identifier and choosing **Find Declaration**.

Code browsing can only find and open units in the project Search path or Source path, or in the product Browsing or Library path. Directories are searched in the following order:

1. The project Search path
2. The project Source path, the directory in which the project was saved
3. The global Browsing path
4. The global Library path
5. The Library path, that is searched only if there is no project open in the IDE

These paths can be modified by editing the corresponding values in the list of Directories, available by going to: **Tools > Options > Environment Options > Delphi Options > Library - Win32**.

Refactoring

Refactoring is the process of improving your code without changing its external functionality.

For example, you can turn a selected code fragment into a method by using the extract refactoring method. The IDE moves the extracted code outside of the current method, determines the needed parameters, generates local variables if necessary, determines the return type, and replaces the code fragment with a call to the new method.

Several other refactoring methods, such as renaming a symbol and declaring a variable, are also available.

Keystroke Macros

You can record a series of keystrokes as a macro while editing code. The red button at the bottom of the code window starts the recording. After you record a macro, you can play it back to repeat the keystrokes during the current IDE session. Recording a macro replaces the previously recorded macro.

To-Do Lists

A *To-Do List* records the tasks that need to be completed for a project. After you add a task to the To-Do List, you can edit the task, add it to your code as a comment, indicate that it has been completed, and then remove it from the list.

You can filter the list to display only the tasks that are of interest to you.

To display the To-Do List window, select **View > To-Do List**.

Custom Template Libraries

RAD Studio allows you to create multiple custom template libraries to use as the basis for creating future projects. Template libraries let you declare how projects can look, and they enable you to add new types of projects to the Object Repository.

History Manager

The *History Manager* lets you compare versions of a file, including multiple backup versions, saved local changes, and the buffer of unsaved changes for the active file.

If the current file is under version control, all types of revisions are available in the History Manager.

The History Manager is displayed on the **History** tab, which lies at the center of the IDE to the right of the **Code** tab.

The History Manager contains the following tabbed pages.

Page	Description
Contents	Displays the current and previous versions of the file.
Info	Displays all labels and comments for the active file.
Diff	Displays the differences between the selected versions of the file.

Table 2-2. History Manager pages

The following image shows the Diff page of the History Manager, comparing two different versions of a .dfm file. The differences are shown using plus/minus signs and the corresponding lines are highlighted in green/magenta.

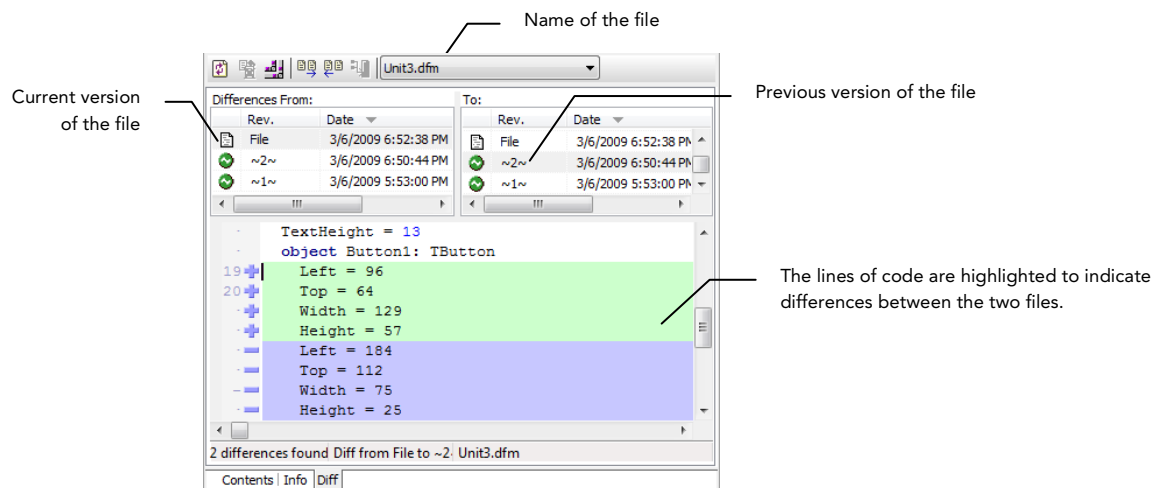


Figure 2-27. Comparing two versions of a file using the **Diff** page

Revision icons are used to represent file versions in the revision lists and they are described in the following table.






Icon	Description
	The latest saved file version.
	A backup file version.
	The file version that is in the buffer and includes unsaved changes.
	A file version that is stored in a version control repository.
	A file version that you have checked out from a version control repository.

Table 2-3. Revision icons on the **Diff** page

Data Explorer

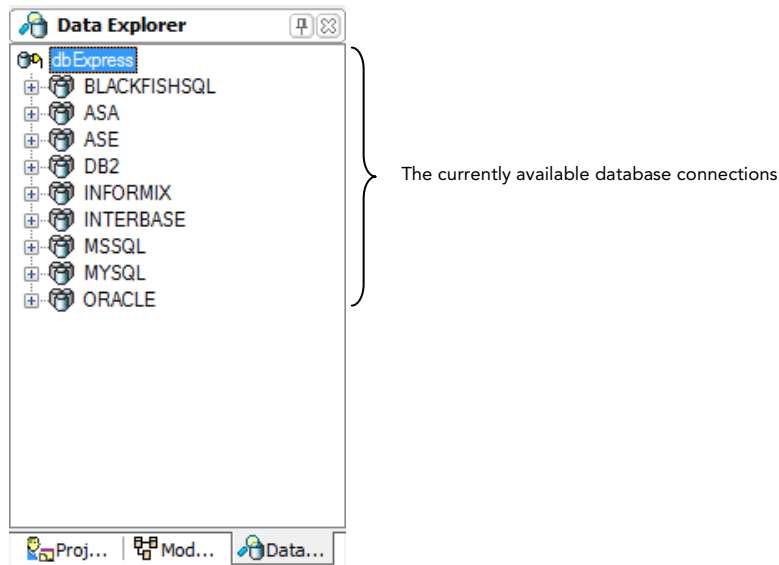


Figure 2-28. Exploring the list of available database connections

RAD Studio offers a variety of database and connectivity tools to simplify the development of database applications.

The *Data Explorer* is located, by default, in the upper right corner of the IDE. The Data Explorer allows you to create and modify database connections that can easily be used later in your database applications.

Note: Data Explorer works for databases that use the DBExpress connection type.

After you create a database connection, you can use the Data Explorer to create, view and modify tables, views, procedures, function, and synonyms. You can click an item from the expanded connection type entry. A menu that allows you to refresh the data or create a new item will appear.

Starting your first RAD Studio application

This chapter explains how to use the Rapid Application Development tools of CodeGear™ RAD Studio 2009 to create a GUI (Graphical User Interface) application. You start with creating the main form, customizing it, and adding the necessary visual and nonvisual components.

The sections of code in the application to handle user actions are called *event handlers*, which you also need to implement. After following the steps, given both for Delphi and C++, you will have a basic text editor, with a few additional features like word-wrapping and the ability to change the font and display the current cursor position in the status bar. The event handlers that implement these features refer to clicking options in the main menu and to typing or clicking inside the edit box.

Using project templates from the Object Repository

First, create a project in CodeGear™ RAD Studio 2009 by clicking **File > New** and choosing **VCL Forms Application > C++Builder** or **VCL Forms Application > Delphi**, depending on the language you want to use to develop the text editor. At this point, the **File** menu and its **Open** command submenu should be displayed as in the following image.

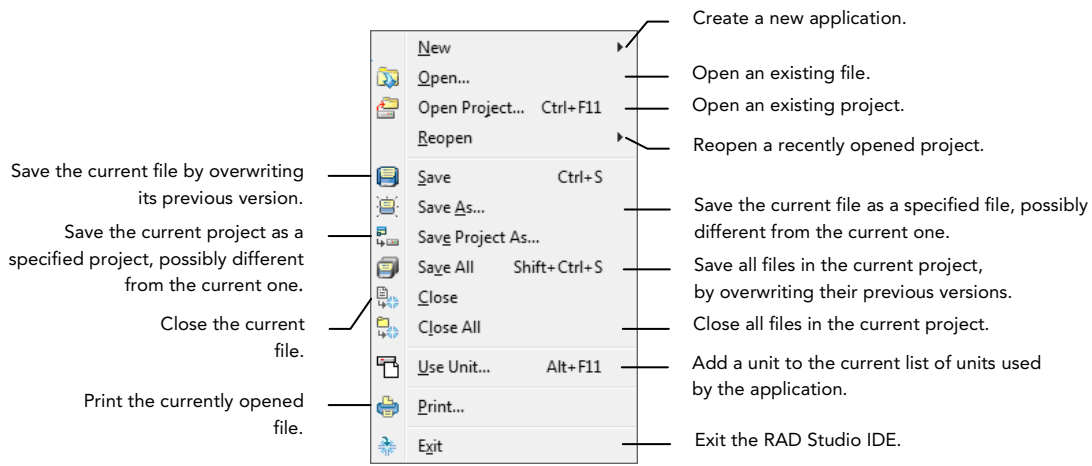


Figure 3-1. Description of all options in the **File** menu

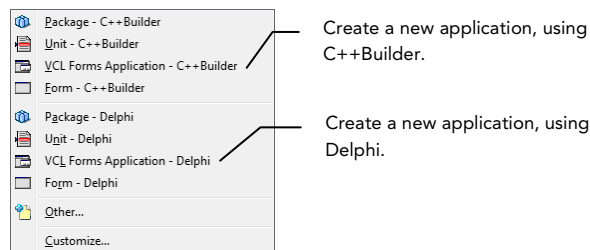


Figure 3-2. Expanding the **New** option in the **File** menu

After clicking the menu item to create a project, several files are automatically generated, using the VCL Forms Application template, also including the main form. After the files are generated, the main form is displayed in the Form Designer. The next images give a screenshot of the IDE at this step, using Delphi or C++Builder, respectively.

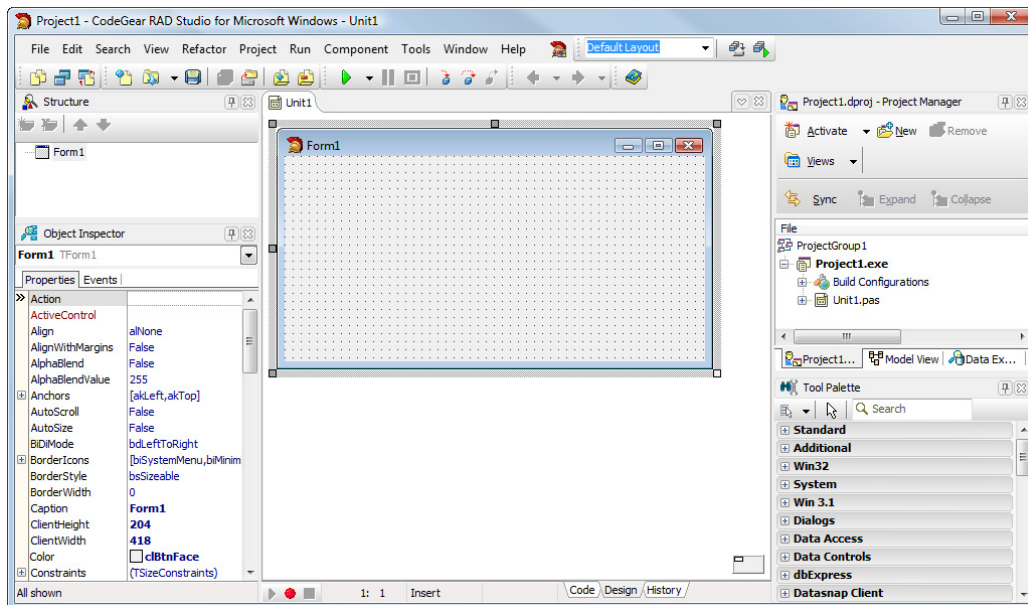


Figure 3-3. The default layout for creating a RAD Studio application (Delphi view)

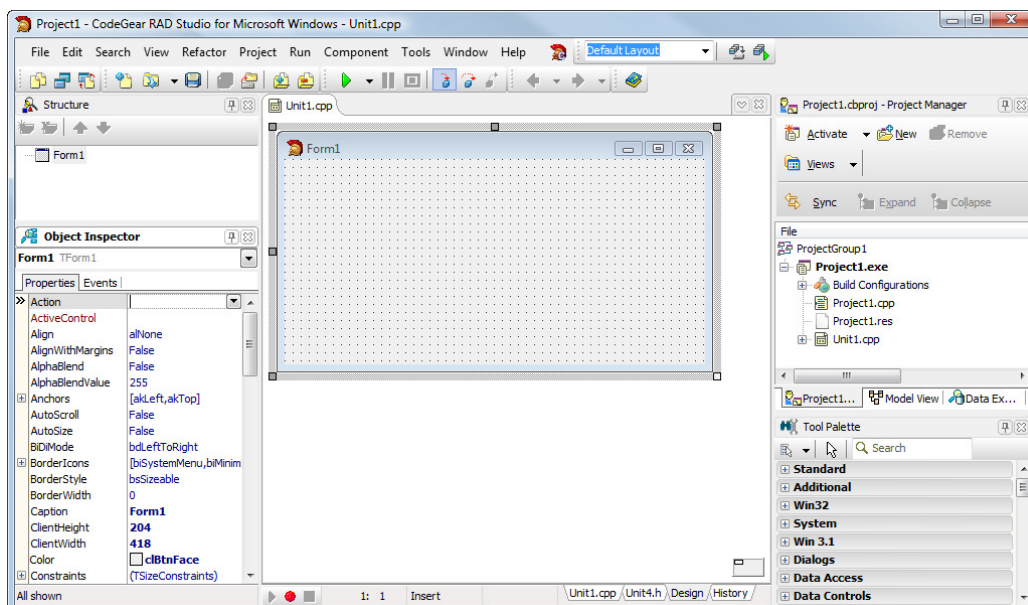


Figure 3-4. The default layout for creating a RAD Studio application (C++Builder view)

Basic customization of the main form

Before adding any components, you should start by doing some basic customization to the form. Make sure that the main form is activated (click it once otherwise) and that the Object Inspector window is visible in the lower left quadrant (if not, press **F11** to display it). With the **Properties** tab selected, look for the *Caption* property and change its value to `Text Editor`; also, change the value of the *Name* property to `TextEditorForm`.

To make the design of the project more visually balanced, set the main form to initially be positioned in the center of the screen. To do this, change the value of the *Position* property to `poScreenCenter` by clicking the value field for *Position* and then selecting the value from the dropdown list. For the same reason, make the form square-shaped, by changing the values of both *Width* and *Height* to `400`, or any other number you prefer, as long as the number does not exceed the current screen size.

After making these changes, the main form should look like the following images, using Delphi or C++Builder, respectively.

Basic customization of the main form

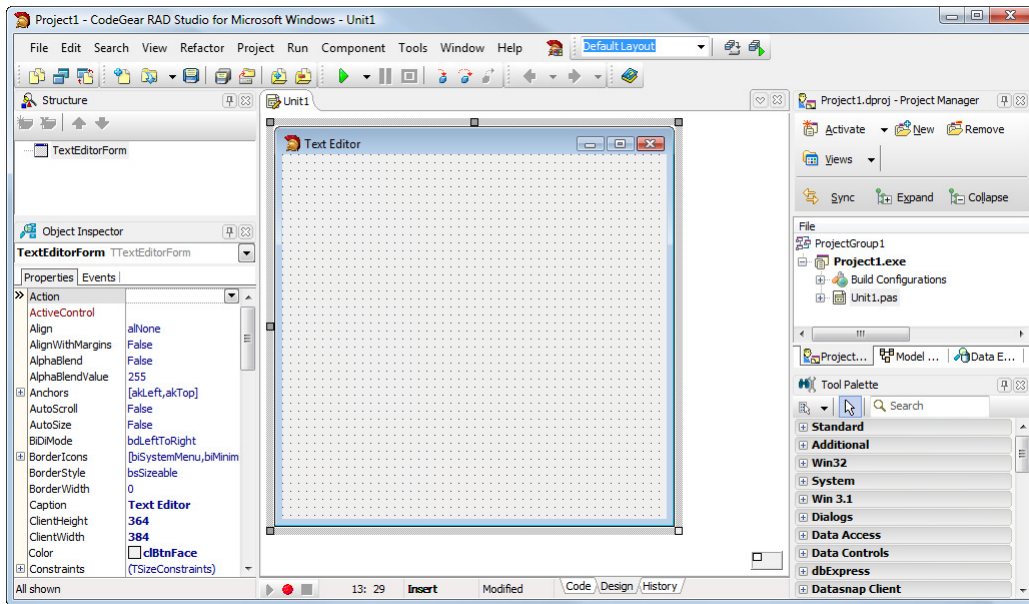


Figure 3-5. Basic customization of the main form (Delphi view)

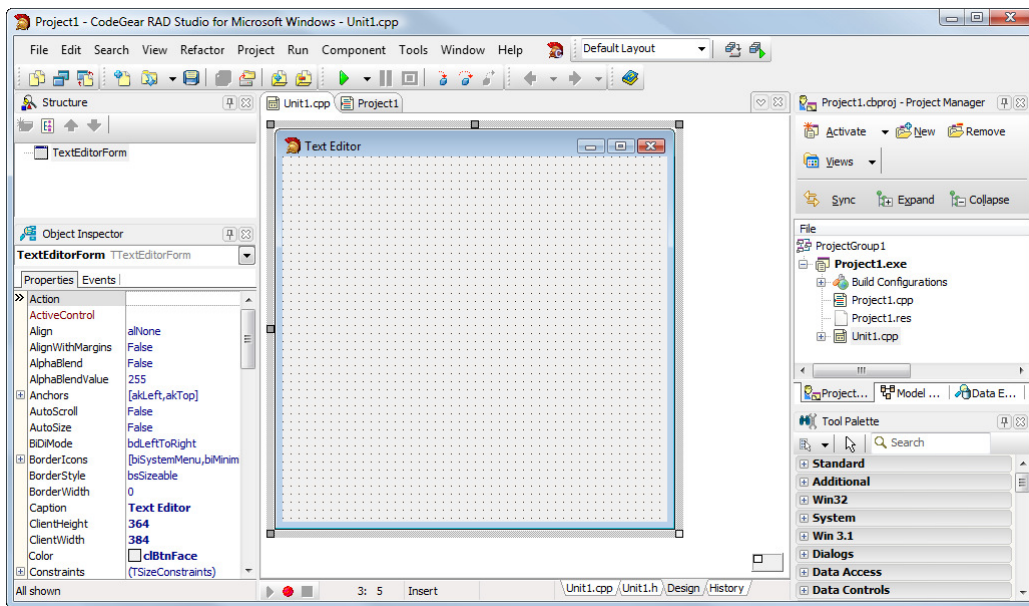


Figure 3-6. Basic customization of the main form (C++Builder view)

Adding the components using the Form Designer

Adding the components using the Form Designer

Now that you have set up the main form, you can proceed with arranging the necessary components to create your text editor application. First, you need to add a menu bar providing the basic options for file manipulation, editing, and also other options like changing the font or toggling word wrap.

Adding an action manager

Add an action manager to the form to automatically provide the basic functionality of your application. To do so, make sure the **Design** tab is selected, go to the Tool Palette and type the text `action` in the search box, in order to locate the `TActionManager` component. The Tool Palette should look as in the following image.

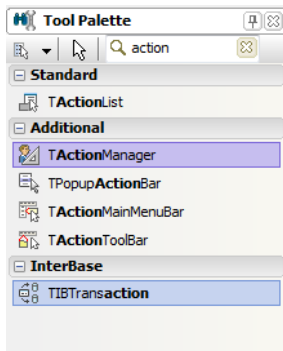



Figure 3-7. Using the `action` filter in the Tool Palette to select `TActionManager`

Double-click the `TActionManager` button to add it to the form. Now you should change the name of the `ActionManager` to suit your application. To do this, make sure Object Inspector is visible (if not, press **F11** to display it) and click the Action Manager icon  to activate it. Click the `Name` property and change its value to `ActionMgr`.

Adding the main menu

To put a main menu bar on the form, simply locate the `TActionMainMenuBar` component in the Tool Palette. Double-click `TActionMainMenuBar` to add it to the form.

Adding a Status Bar

Next, you should also put a status bar on the form. To do this, type `status` in the search box of the Tool Palette to locate the `TStatusBar` component. Using this filter, the Tool Palette will look like the following image.

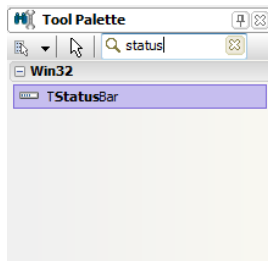


Figure 3-8. Using the `status` filter in the Tool Palette to select `TStatusBar`

As with the previous components, double-click `TStatusBar` to add it to the form.

Adding a text box

The only component left to add is a text box, giving your application its main functionality—that of a text editor. Type `memo` in the search box and locate the `TMemo` component. The Tool Palette should now display the components whose names include the word `memo`, as in the following image.

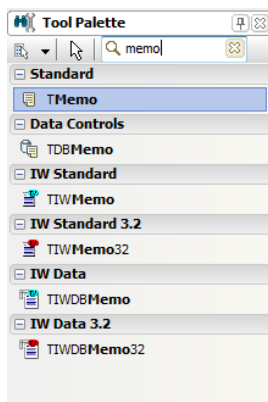


Figure 3-9. Using the `memo` filter in the Tool Palette to select `TMemo`

Double-click `TMemo` to add it to the form. The main form should now display the action manager, the action main menu bar, the status bar, and the memo we have previously added to the form, similar to the following image.

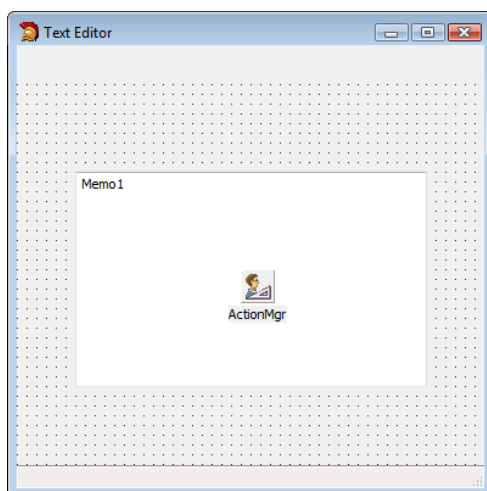


Figure 3-10. Basic text editor form

Adding the components using the Form Designer

Adding the main menu commands

To finish designing the form, you must add the options to be displayed in the main menu. Start by double-clicking the action manager component on the form to open the **Actions** editor.

The following window should be displayed.

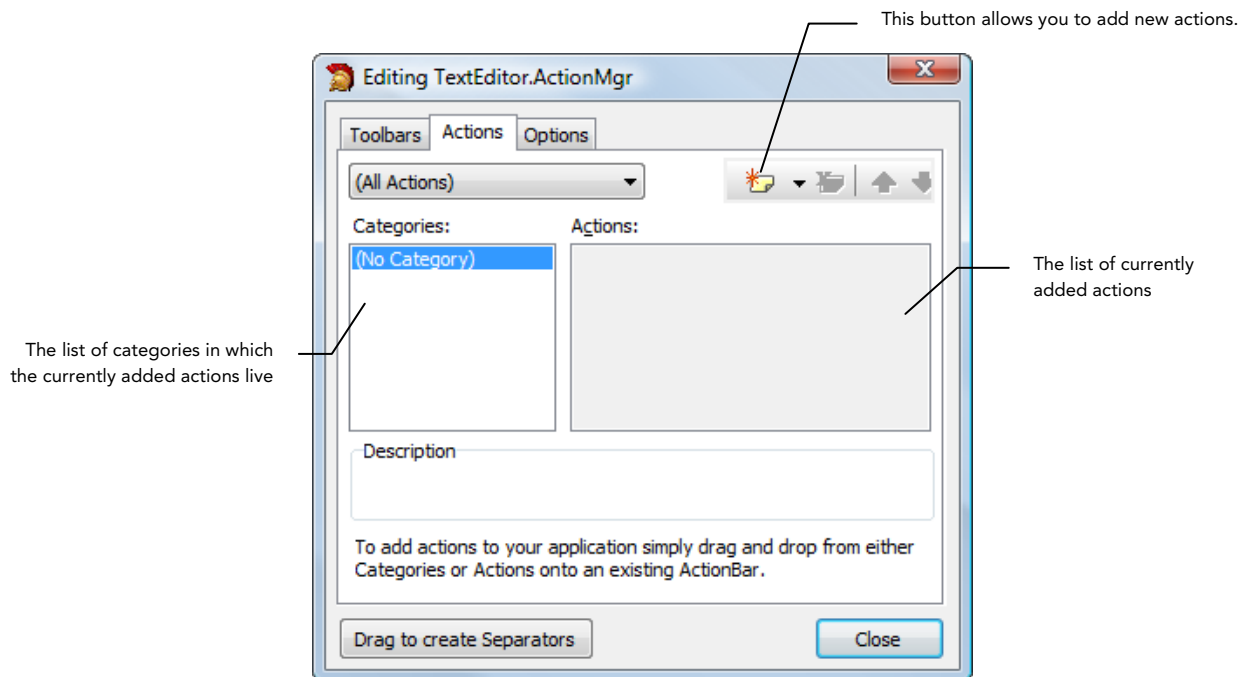


Figure 3-11. The main elements of the **Actions** page in the Action Manager

You are now ready to create the items in the main menu. Press **CTRL+Insert** to add new standard actions or click the down-arrow of the **New Action** icon and choose **New Standard Action...** from the menu. The image below shows this menu.

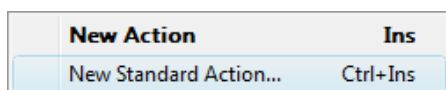


Figure 3-12. Adding a New Standard Action

While pressing **CTRL**, select all items in the *Edit* category and also *TFileOpen*, *TFileSaveAs*, and *TFileExit* from the *File* category, then click **OK**.

The following image shows how the Standard Action Classes list should display, with the items in the *File* category selected.

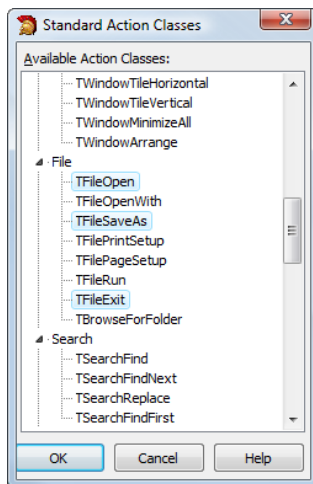


Figure 3-13. Selecting the Standard Actions that implement the basic file and text operations

After clicking **OK**, wait until the menu items are automatically generated. You may notice that the object inspector displays the properties of each menu item, as they are created.

Defining Action Properties

Since you need other options as well, you must define your own custom actions. To do this, from the Actions Editor, select *File* from the **Categories** list and click the **New Action** button twice to create two new, non-standard actions.

You can now customize the newly created actions. Click *Action1* in the **Actions** list and use the Object Inspector to change its *Name* property to *New* and its *ShortCut* to **CTRL+N**. Also, click *Action2* and change its *Name* to *Save* and its *ShortCut* property to **CTRL+S**.

Now, use the **Move Up** and **Move Down** arrows to put the actions in the right order, as in the following image.

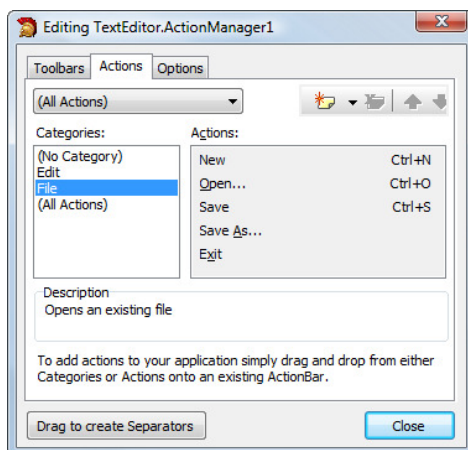


Figure 3-14. Arranging the actions in the **File** menu and finishing adding the Standard Actions

Adding word wrap and font capabilities

To give your text additional features—word-wrapping and the ability to change the font—you need to add another main menu option. Click *(No Category)* from the **Categories** list and press **CTRL+Insert** on the keyboard to create a new standard action. The **Standard Actions Classes** list is displayed.

Select TFontEdit from the *Dialogs* category and click **OK**. In the **Actions** list, click *Select Font* and use the Object Inspector to change its *Category* property to `Format`. Do this by selecting **Category** and type the word `Format`. Also, write `Font` as its *Caption* property.

With the *Format* category selected in the **Categories** list, press the *New Action* button to define a new action. Change its *Name* to `WordWrap` and its *Caption* to `Word Wrap`, using the Object Inspector.

Now drag each item from the **Categories** list to the menu bar at the top of the main form, in this order: **File, Edit, Format**.

The following image shows how the **File** menu should look.

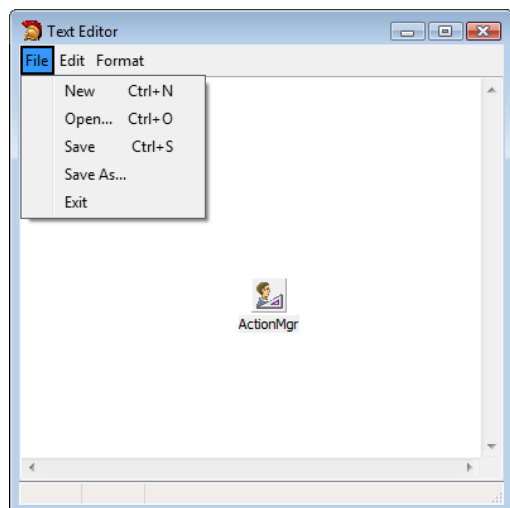


Figure 3-15. The final look of the **File** menu

Finally, close the Actions Editor to continue with customizing your application.

Customizing the components

In the previous section of this chapter, “Arranging the components in the Form Designer”, you have added all the required components to your form and then configured the action manager. Before you continue with writing code for the event handlers, you must first customize the properties of the newly placed components.

To customize a component, first select the component in the Form Designer. Then you can edit the properties of the component in the Object Inspector.

Follow these steps to customize the memo component:

1. Select the memo component in the Form Designer by clicking the memo component.
2. Find the memo component in the Object Inspector (if the Object Inspector is not visible, press **F11** or click **View > Object Inspector**).
3. Set the *Align* property to `alClient`. This makes the memo component occupy all the free space available on the form.
4. Set the *Name* property to `TextMemo`. Naming your component properly is very important because your code needs to access the component using that name. Setting a name you can easily remember is essential.
5. Set the *ScrollBars* property to `ssBoth`. This setting ensures that both the vertical and horizontal scroll bars are displayed in the memo and allows users to easily scroll through its contents.
6. Set *WordWrap* to `False`. *WordWrap* tells the memo to wrap all text on several lines if the text does not fit in a single line. A `False` value disables word wrapping.
7. Find the *Lines* property and press the '...' button located in the value box. A new edit dialog appears, allowing you to edit the initial contents of the memo, as in the image on the right. Delete all the text and then press **OK** to clear the memo.

Contents of the memo box

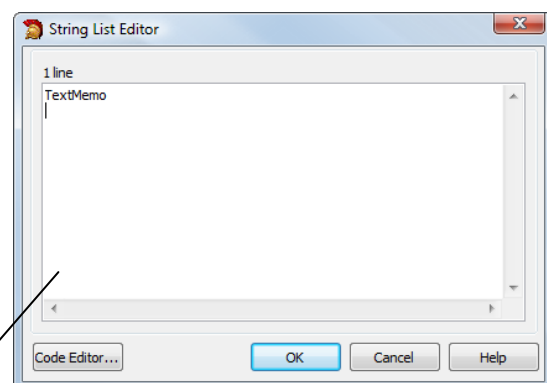


Figure 3-16. Editing the contents of the memo

Customizing the components

After you have customized the memo, select the status bar component and customize it as follows.

1. Select the status bar in the **Design** window.
2. Set the *Name* property to `TextStatus`.
3. Find the *Panels* property and press the "... " button at the right side of the value box. This will display a new dialog box that allows adding and customizing panels displayed in the status bar.
4. Press the **Insert** key three times to add three panels. The panel editor should look as in the following image. You do not need to customize these panels, so just close the dialog.

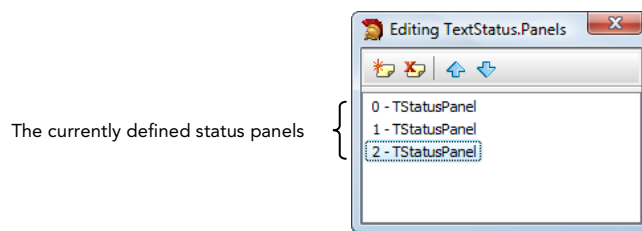


Figure 3-17. Panel editor showing the list of added status panels

This concludes all customization you need to perform on your components.

Before proceeding with writing any code, save all the changes you have made to the project, as follows. Click **File > Save As** and save the unit as `TextEditor`. Also, click **File > Save Project As** and save the project as `TextEditor_proj`.

Coding responses to user actions in the Code Editor

By following the instructions in this section, you will make your application interactive and provide it with the functionality you want. You will code event handlers, that is, the responses to clicking the various options in the main menu.

Before proceeding with writing any code, define the *String* variable, which you need throughout the execution of the application to retain the name of the currently opened text file. First make sure you are in Code Editor mode by selecting the **Code** tab, next to the **Design** tab in the status bar. To toggle between Form Designer and Code Editor mode, press **F12**.

In Delphi, define a *String* variable called *CurrentFile* in the private section of the *TTextEditorForm* class, in the interface part, as in the following image.

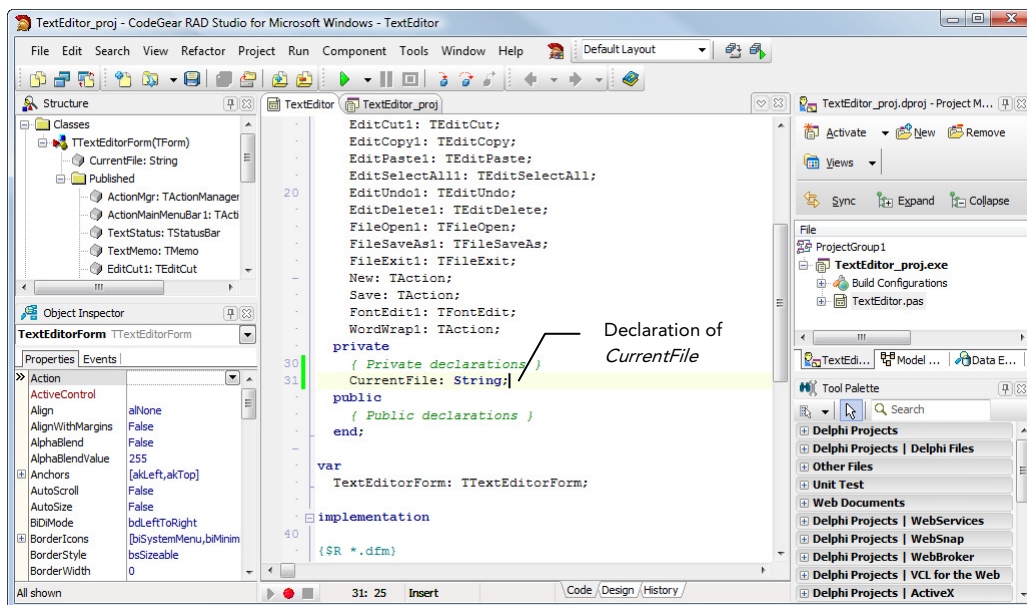


Figure 3-18. Defining the *CurrentFile* private variable (Delphi view)

In C++, use the tabs at the bottom of the Code Editor window to display the *TextEditor.h* file. Also declare the *currentFile* variable in the private section of *TTextEditorForm*, as in the following image.

Coding responses to user actions in the Code Editor

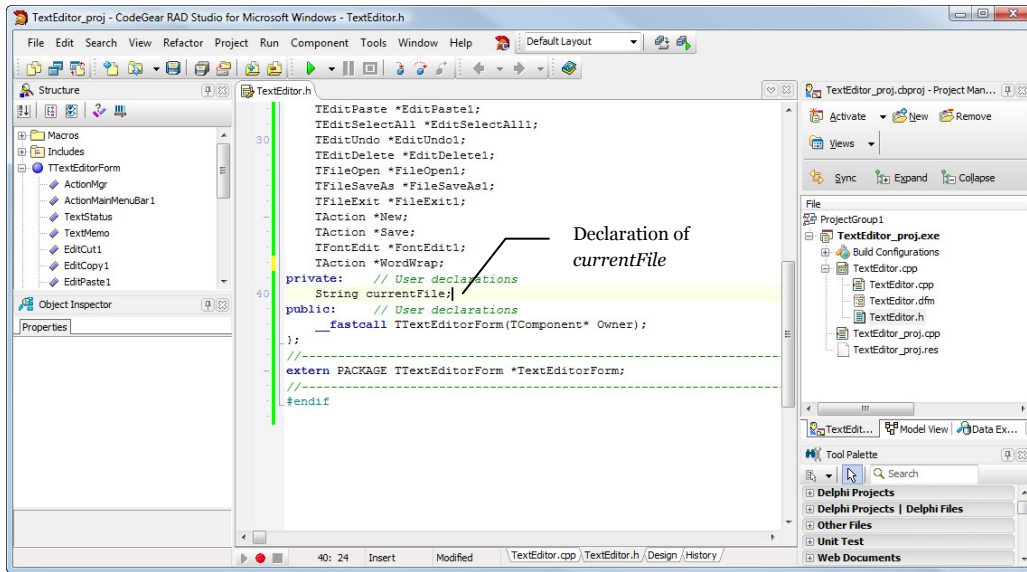


Figure 3-19. Defining the *currentFile* private variable (C++Builder view)

Creating an event handler for the New command

You are now ready to define the responses to clicking the menu items. In the Form Designer, click **File > New** on the menu bar in your text editor form. Then select the **Events** tab in the Object Inspector, as in the following image. Click the plus sign (+) to expand the **Action** list if necessary.

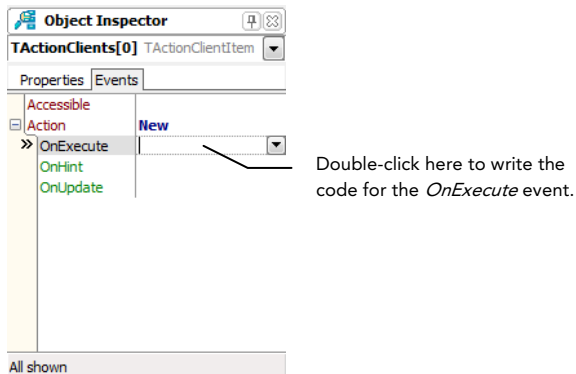


Figure 3-20. Opening the **Events** tab in the Object Inspector

Double-click the edit box corresponding to the *OnExecute* event. The Code Editor opens and displays the following function skeleton, using Delphi or C++Builder, respectively.

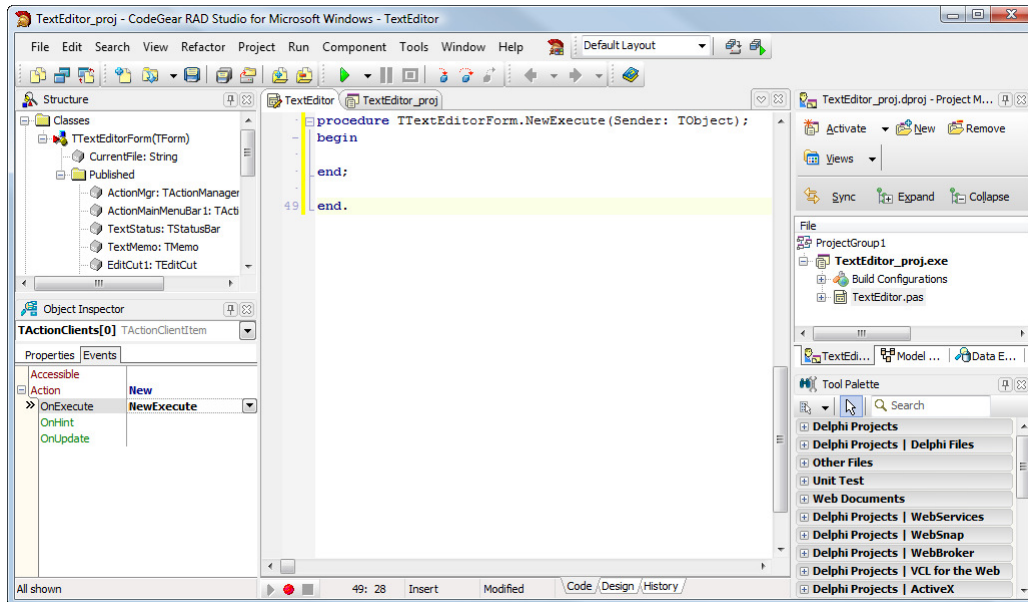


Figure 3-21. Automatic generation of the code skeleton for the *OnExecute* event (Delphi view)

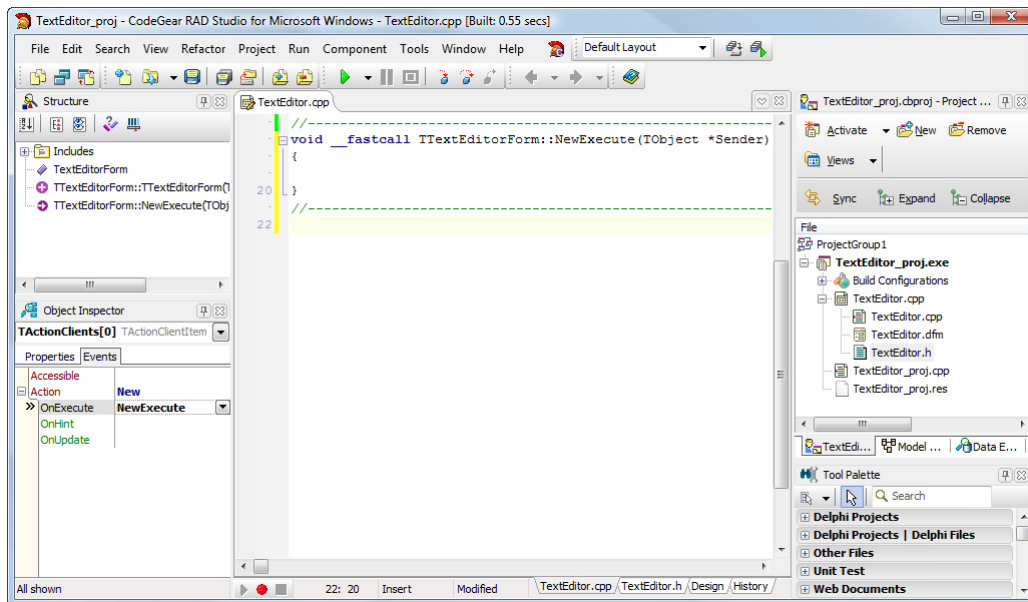


Figure 3-22. Automatic generation of the code skeleton for the *OnExecute* event (C++Builder view)

Coding responses to user actions in the Code Editor

Now write the code that executes when the user selects **File > New**, inside the code skeleton previously generated, as in the following lines of code.

```
procedure TTextEditorForm.NewExecute(Sender: TObject);
var
  UserResponse : Integer;
begin
  if TextMemo.Lines.Count > 0 then
  begin
    UserResponse := MessageDlg(
      'This will clear the current document. ' +
      'Do you want to continue?', mtInformation,
      mbYesNo, 0);

    if UserResponse = mrYes then
    begin
      TextMemo.Clear;
      CurrentFile := '';
    end;
  end;
end;
```

```
void __fastcall TTextEditorForm::NewExecute(TObject *Sender)
{
  if (TextMemo->Lines->Count > 0)
  {
    int userResponse = MessageDlg(
      String("This will clear the current document. ")
      + "Do you want to continue?", mtInformation,
      TMsgDlgButtons() << mbYes << mbNo, 0);

    if (userResponse == mrYes) {
      TextMemo->Clear();
      currentFile = "";
    }
  }
}
```

Creating the event handlers for the Open command

Return to the form and double-click the *OnAccept* event of the **File > Open** item and write the code displayed below.

```

procedure TTextEditorForm.FileOpen1Accept(Sender: TObject);
var
    FileName: String;
begin
    FileName := FileOpen1.Dialog.FileName;
    if FileExists(FileName) then
        begin
            TextMemo.Lines.LoadFromFile(FileName);
            CurrentFile := FileName;
            Self.Caption := 'Text Editor - ' + ExtractFileName(FileName);
        end;
end;

```

```

void __fastcall TTextEditorForm::FileOpen1Accept(TObject *Sender)
{
    String fileName = FileOpen1->Dialog->FileName;
    if (FileExists(fileName)) {
        TextMemo->Lines->LoadFromFile(fileName);
        currentFile = fileName;
        this->Caption = "Text Editor - " + ExtractFileName(fileName);
    }
}

```

Creating the event handlers for the SaveAs command

Double-click the *OnAccept* event of **File > SaveAs** and write the following code.

```
procedure TTextEditorForm.FileSaveAs1Accept(Sender: TObject);  
var  
    FileName: String;  
    UserResponse : Integer;  
begin  
    FileName := FileSaveAs1.Dialog.FileName;  
  
    if FileExists(FileName) then  
    begin  
        UserResponse := MessageDlg(  
            'File already exists. ' +  
            'Do you want to overwrite?', mtInformation,  
            mbYesNo, 0);  
        if UserResponse = mrNo then  
            Exit();  
    end;  
  
    TextMemo.Lines.SaveToFile(FileName);  
    CurrentFile := FileName;  
    Self.Caption := ExtractFileName(FileName);  
end;
```

```
void __fastcall TTextEditorForm::FileSaveAs1Accept(TObject *Sender)  
{  
    String fileName = FileSaveAs1->Dialog->FileName;  
  
    if (FileExists(fileName)) {  
        int userResponse = MessageDlg(  
            String("File already exists. ") +  
            "Do you want to overwrite?", mtInformation,  
            TMsgDlgButtons() << mbYes << mbNo, 0);  
        if (userResponse == mrNo) {  
            return;  
        }  
    }  
  
    TextMemo->Lines->SaveToFile(fileName);  
    currentFile = fileName;  
    this->Caption = ExtractFileName(fileName);  
}
```


Creating the event handlers for the Save command

Double-click the *OnExecute* event of **File > Save** and write the following lines of code.

```
procedure TTextEditorForm.SaveExecute(Sender: TObject);
begin
    if CurrentFile = '' then
        Self.FileSaveAs1.Execute()
    else
        TextMemo.Lines.SaveToFile(CurrentFile);
end;
```

```
void __fastcall TTextEditorForm::SaveExecute(TObject *Sender)
{
    if (currentFile == "") {
        this->FileSaveAs1->Execute();
    }
    else {
        TextMemo->Lines->SaveToFile(currentFile);
    }
}
```

Coding responses to user actions in the Code Editor

Creating the event handlers for the Font command

Double-click the *OnAccept* event of **Format > Font** and write the following code.

```
procedure TTextEditorForm.FontEdit1Accept(Sender: TObject);  
begin  
    TextMemo.Font := FontEdit1.Dialog.Font;  
end;
```

```
void __fastcall TTextEditorForm::FontEdit1Accept(TObject *Sender)  
{  
    TextMemo->Font = FontEdit1->Dialog->Font;  
}
```

Creating the event handlers for the Word Wrap command

Next, double-click the *OnExecute* event of **Format > Word Wrap** and write the following code.

```
procedure TTextEditorForm.WordWrapExecute(Sender: TObject);  
begin  
    { Toggle the word wrapping state. }  
    TextMemo.WordWrap := not TextMemo.WordWrap;  
    WordWrap.Checked := TextMemo.WordWrap;  
    if TextMemo.WordWrap = True then  
        { Only vertical scrollbars are needed when word wrapping is set. }  
        TextMemo.ScrollBars := ssVertical  
    else  
        TextMemo.ScrollBars := ssBoth;  
end;
```

```
void __fastcall TTextEditorForm::WordWrapExecute(TObject *Sender)  
{  
    { Toggle the word wrapping state. }  
    TextMemo->WordWrap = !TextMemo->WordWrap;  
    WordWrap->Checked = TextMemo->WordWrap;  
    if (TextMemo->WordWrap == True) {  
        { Only vertical scrollbars are needed when word wrapping is set. }  
        TextMemo->ScrollBars = ssVertical;  
    }  
    else {  
        TextMemo->ScrollBars = ssBoth;  
    }  
}
```

Creating event handlers for the status bar

Finally, use the status bar to display the current cursor position and also the number of lines of the currently opened text file. To do this, double-click the *OnMouseDown* event of the *TextMemo* component and write the following code, in Delphi and C++ respectively. The *CaretPos* property is used to indicate the coordinates of the caret inside the text memo box.

```
procedure TTextEditorForm.TextMemoMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  TextStatus.Panels.Items[0].Text :=
    'L: ' + IntToStr(TextMemo.CaretPos.Y + 1);
  TextStatus.Panels.Items[1].Text :=
    'C: ' + IntToStr(TextMemo.CaretPos.X + 1);
  TextStatus.Panels.Items[2].Text :=
    'Lines: ' + IntToStr(TextMemo.Lines.Count);
end;
```

```
void __fastcall TTextEditorForm::TextMemoMouseDown(TObject *Sender,
  TMouseButton Button, TShiftState Shift, int X, int Y)
{
  TextStatus->Panels->Items[0]->Text =
    "L: " + String (TextMemo->CaretPos.y + 1);
  TextStatus->Panels->Items[1]->Text =
    "C: " + String (TextMemo->CaretPos.x + 1);
  TextStatus->Panels->Items[2]->Text =
    "Lines: " + IntToStr (TextMemo->Lines->Count);
}
```

Next, double-click the *OnKeyDown* event of *TextMemo* and write the code below. The *OnKeyDown* event is triggered whenever you press a key inside the text memo box.

```
procedure TTextEditorForm.TextMemoKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  TextMemoMouseDown(Sender, mbLeft, Shift, 0, 0);
end;
```

```
void __fastcall TTextEditorForm::TextMemoKeyDown(TObject *Sender,
  WORD &Key, TShiftState Shift)
{
  TextMemoMouseDown(Sender, mbLeft, Shift, 0, 0);
}
```

Coding responses to user actions in the Code Editor

Compiling and running the application

Before you can actually see your application running, you must first compile it. To compile your application, press **SHIFT-F9** or select **Project > Compile**. You then see a dialog box displaying the progress of the compilation. If your application contains any syntactical errors, you will have to correct them and then recompile.

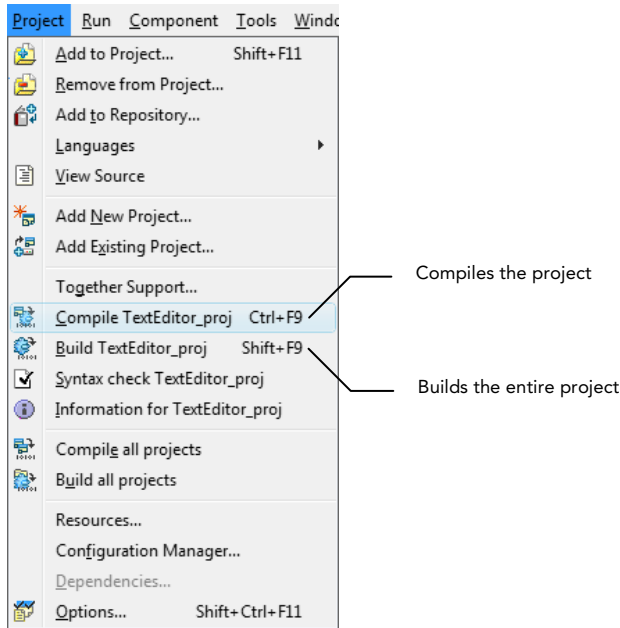


Figure 3-23. Project menu options for compiling and building the project

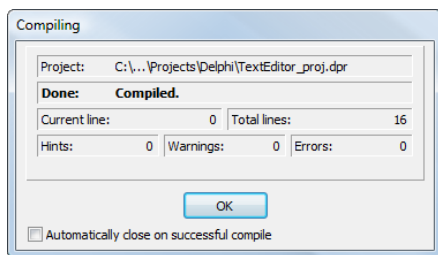


Figure 3-24. Dialog showing the success of *compiling* the application

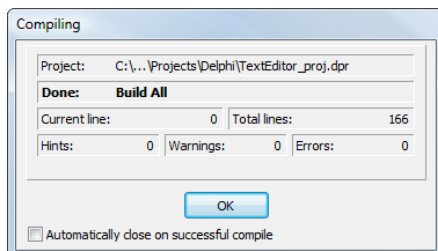


Figure 3-25. Dialog showing the success of *building* the application

After you compile the application, you can see how it behaves at run time. Press **F9** or click **Run > Run** to run your application in debug mode.

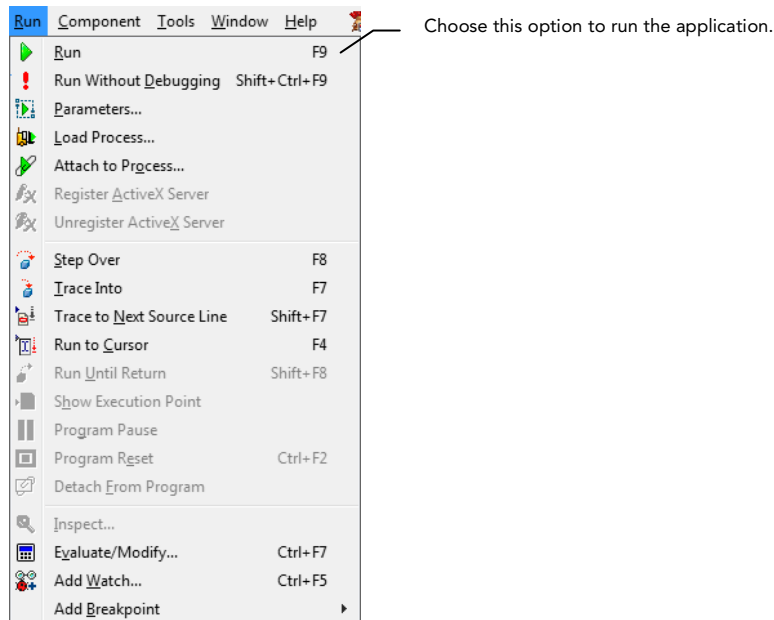


Figure 3-26. Running the application from the **Run** menu

There are a few other options available, but those options are beyond the scope of this book.

Note: You can directly run the application without compiling it first. RAD Studio automatically detects whether compilation is required and compiles the project if necessary.

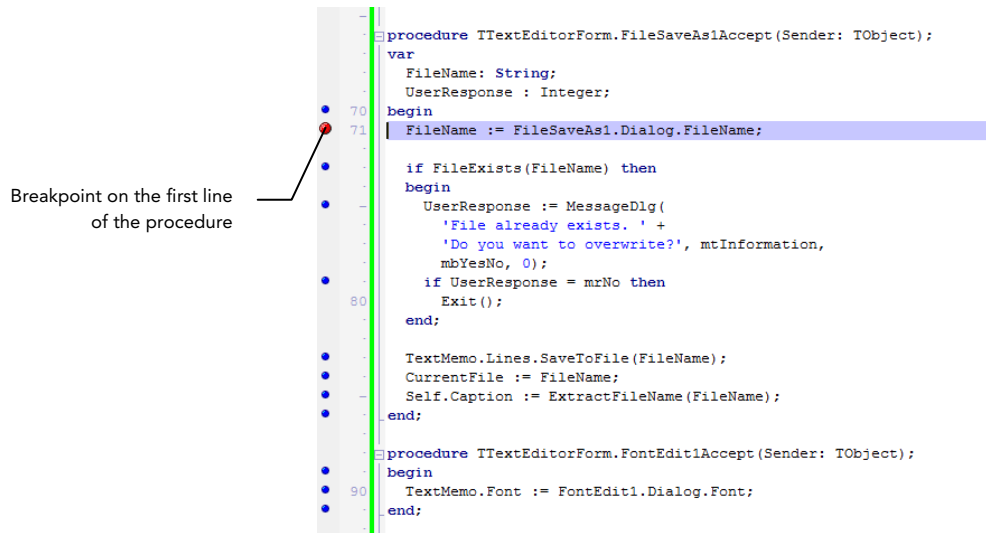
Even if the application successfully compiles and runs, it might still not perform as you intended. The next section in this chapter, called “Debugging the application”, describes how to use some of the RAD Studio debugging features to rapidly find and fix bugs.

Debugging the application

To get a glimpse of the basic debugging features in RAD Studio, first set a breakpoint on the first line of the *FileSaveAs1Accept* function, by clicking on the bar at the left of the line of code.

Debugging the application

The Code Editor window should look like the following image.



The screenshot shows the Delphi code editor with a procedure named `TTextEditorForm.FileSaveAs1Accept`. A red dot breakpoint is placed on the first line of the procedure's `begin` block. A callout box points to this breakpoint with the text "Breakpoint on the first line of the procedure". The code is as follows:


```
procedure TTextEditorForm.FileSaveAs1Accept(Sender: TObject);
var
  FileName: String;
  UserResponse : Integer;
begin
  FileName := FileSaveAs1.Dialog.FileName;

  if FileExists(FileName) then
  begin
    UserResponse := MessageDlg(
      'File already exists. ' +
      'Do you want to overwrite?', mtInformation,
      mbYesNo, 0);
    if UserResponse = mrNo then
      Exit();
  end;

  TextMemo.Lines.SaveToFile(FileName);
  CurrentFile := FileName;
  Self.Caption := ExtractFileName(FileName);
end;

procedure TTextEditorForm.FontEdit1Accept(Sender: TObject);
begin
  TextMemo.Font := FontEdit1.Dialog.Font;
end;
```

Figure 3-27. Debugging the *FileSaveAs1Accept* procedure (Delphi code)



The screenshot shows the C++Builder code editor with a function named `FileSaveAs1Accept`. A red dot breakpoint is placed on the first line of the function's body. A callout box points to this breakpoint with the text "Breakpoint on the first line of the function". The code is as follows:

```
//-----
void __fastcall TTextEditorForm::FileSaveAs1Accept(TObject *Sender)
{
  47 String fileName = FileSaveAs1->Dialog->FileName;

  if (FileExists(fileName)) {
    50 int userResponse = MessageDlg(
      String("File already exists. ") +
      "Do you want to overwrite?", mtInformation,
      TMsgDlgButtons() << mbYes << mbNo, 0);
    if (userResponse == mrNo) {
      return;
    }
  }

  TextMemo->Lines->SaveToFile(fileName);
  currentFile = fileName;
  60 this->Caption = ExtractFileName(fileName);
}

//-----
void __fastcall TTextEditorForm::SaveExecute(TObject *Sender)
{
  if (currentFile == "") {
    this->FileSaveAs1->Execute();
  }
  70 else {
    TextMemo->Lines->SaveToFile(currentFile);
  }
}
```

Figure 3-28. Debugging the *FileSaveAs1Accept* function (C++Builder code)

Press **F9** to run the application, write something in the text box of the text editor and click **File > Save As**. Name your text file, making sure that a file with the same name does not already exist at the current location. After clicking **Save**, the application should stop at the breakpoint you have previously set, and the code editor window should display as in the image below.

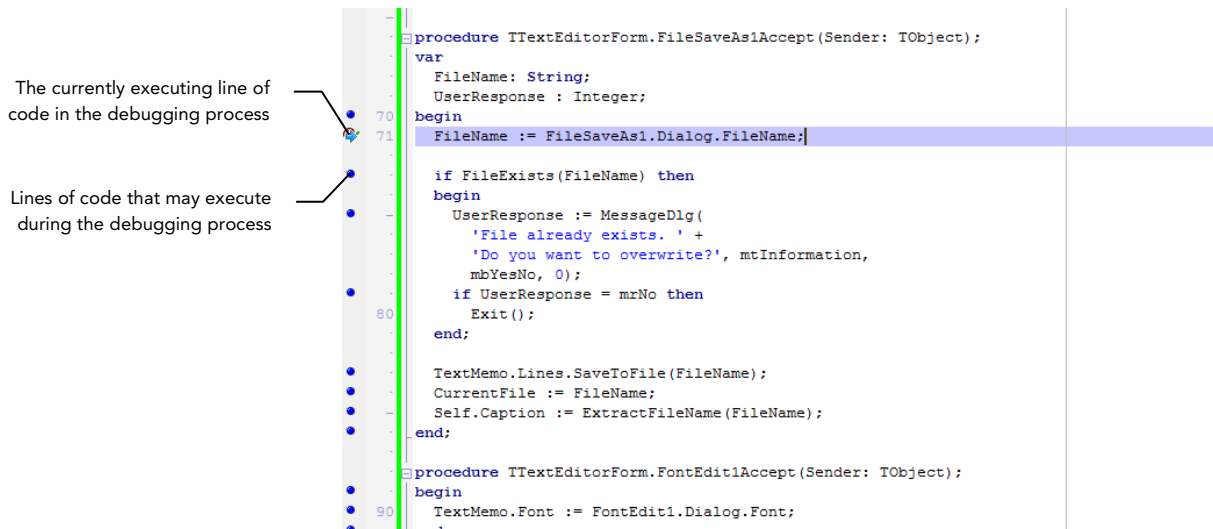


Figure 3-29. Application stopping at the specified breakpoint (Delphi view)

To see the value of the *FileName* variable, select the *FileName* word in the first line of *FileSaveAs1Accept* and drag it to the Watch List, as in the following images.

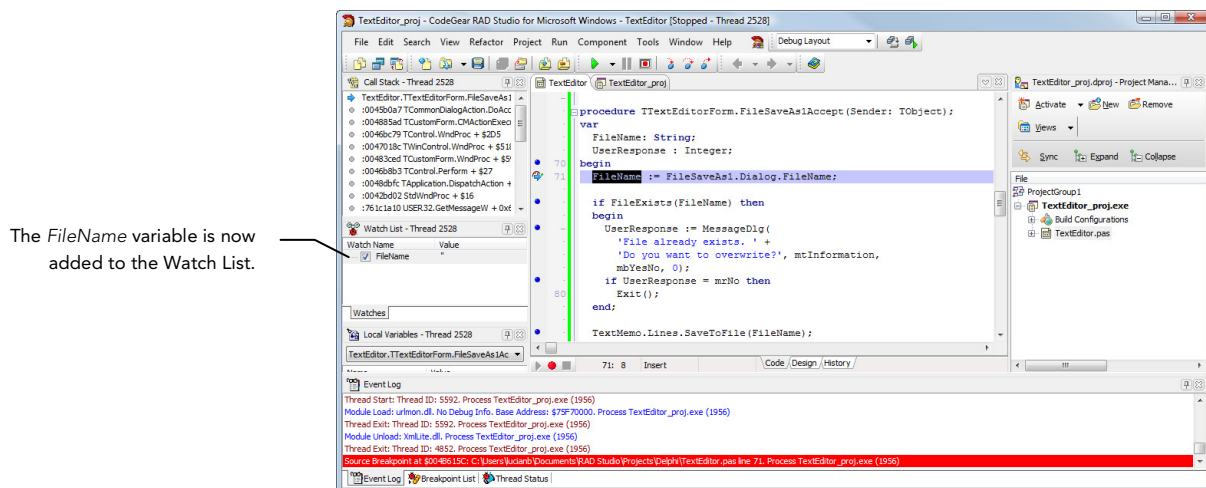


Figure 3-30. Dragging the *FileName* variable to the Watch List (Delphi view)

Debugging the application

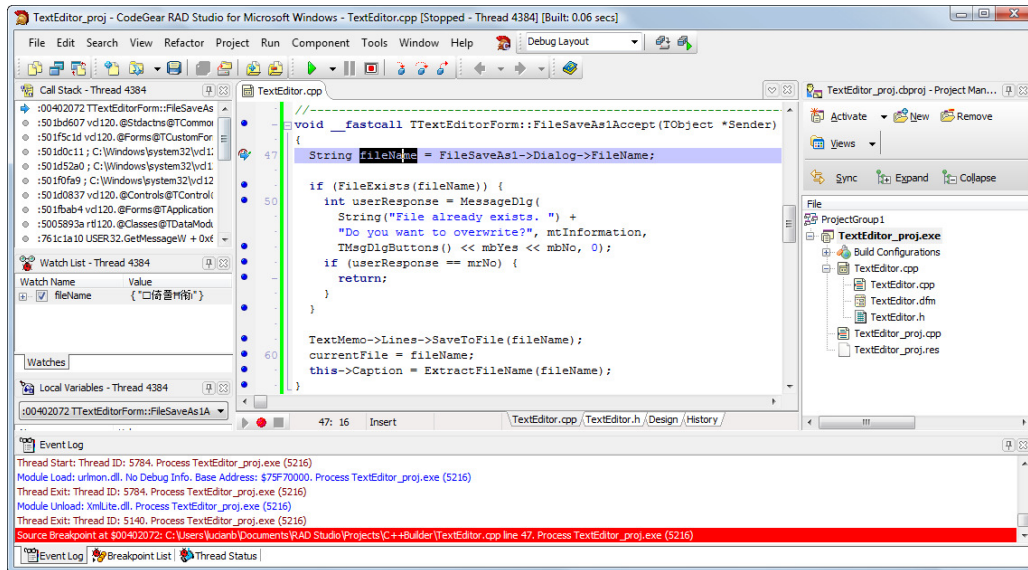


Figure 3-31. Dragging the `fileName` variable to the Watch List (C++Builder view)

Press **F8** to advance to the following line of code, so that the value of `FileName` is updated. To expand the value of `FileName`, hover the mouse cursor over its label in the Watch List and wait.

The result should look as in the following images.

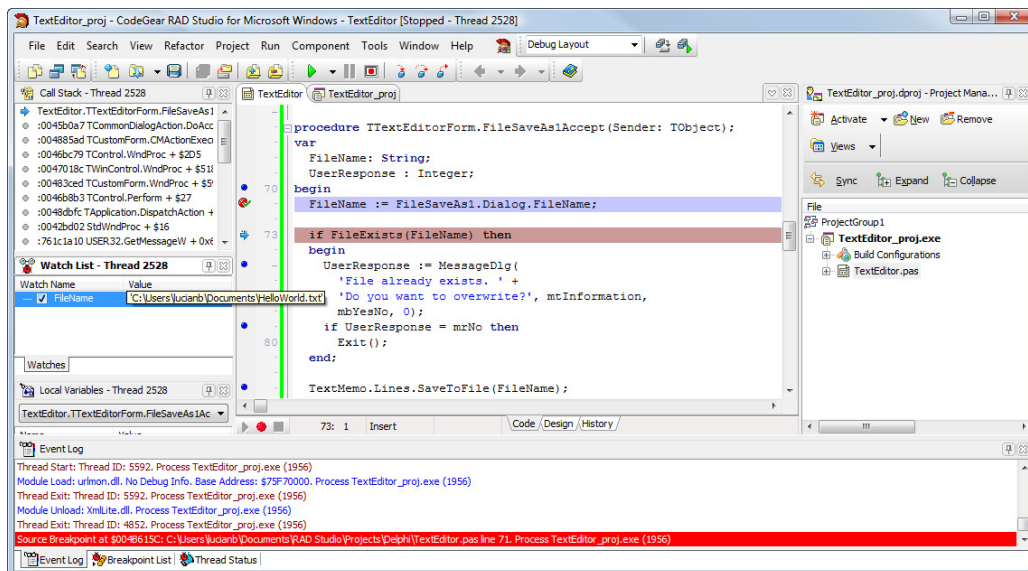


Figure 3-32. Advancing to the next line of code to change the value of `FileName` (Delphi view)

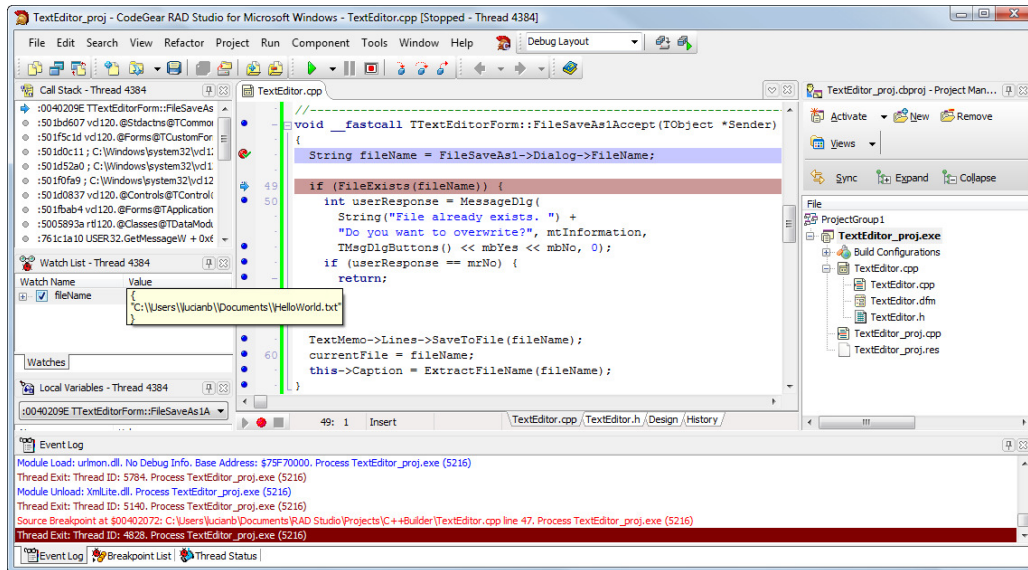


Figure 3-33. Advancing to the next line of code to change the value of *FileName* (C++Builder view)

Pressing **F8** once more jumps over the **if** statement, since a file with the given name does not already exist at the current location. The screen should now look like the following images.

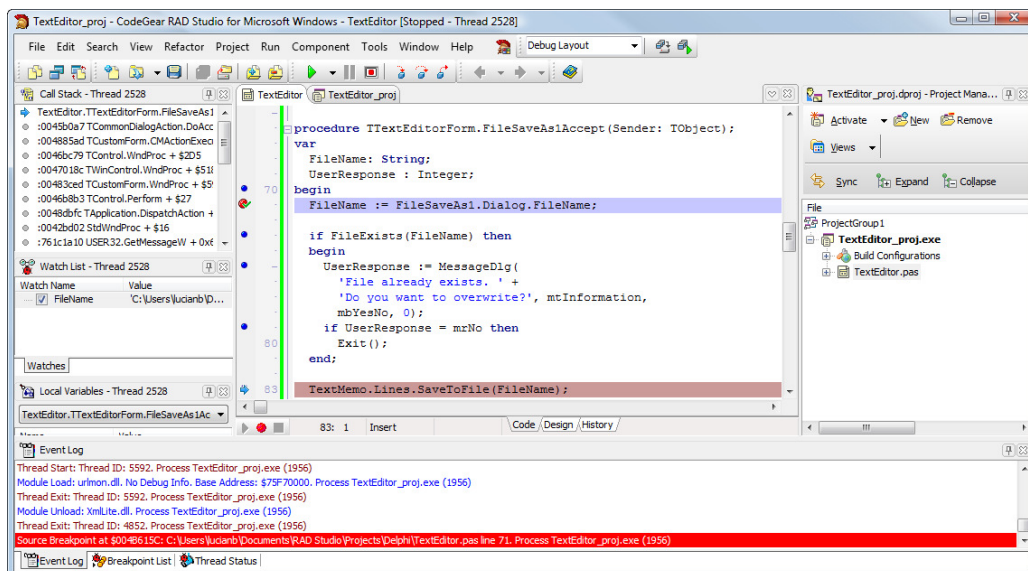


Figure 3-34. Jumping over the **if** statement (Delphi view)

Debugging the application

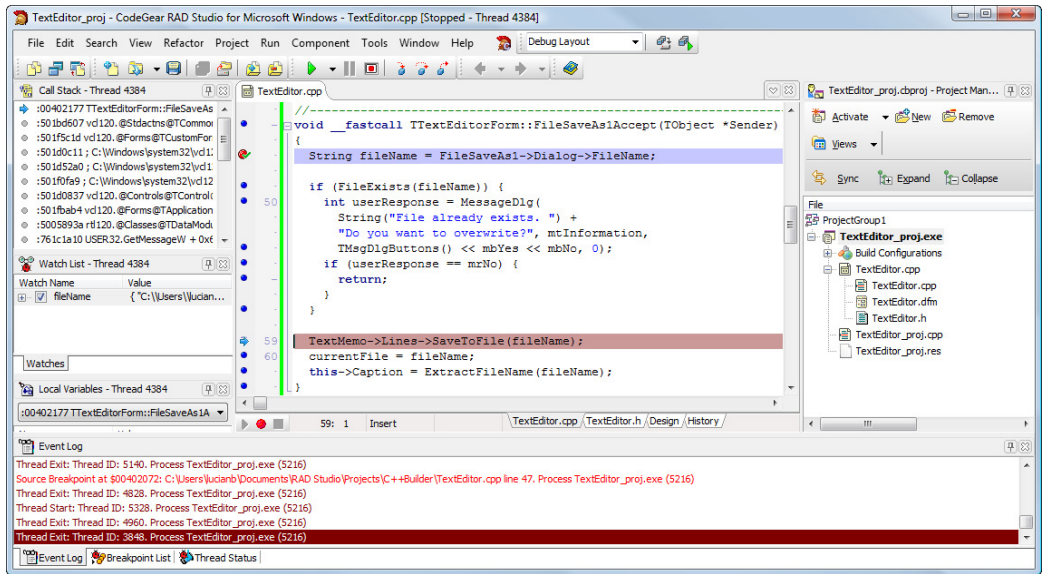


Figure 3-35. Jumping over the if statement (C++Builder view)

Press **F8** until you get to the last line of the *FileSaveAs1Accept* function. Now move the mouse cursor over the name of the *CurrentFile* variable to instantly view its value, as in the following images.

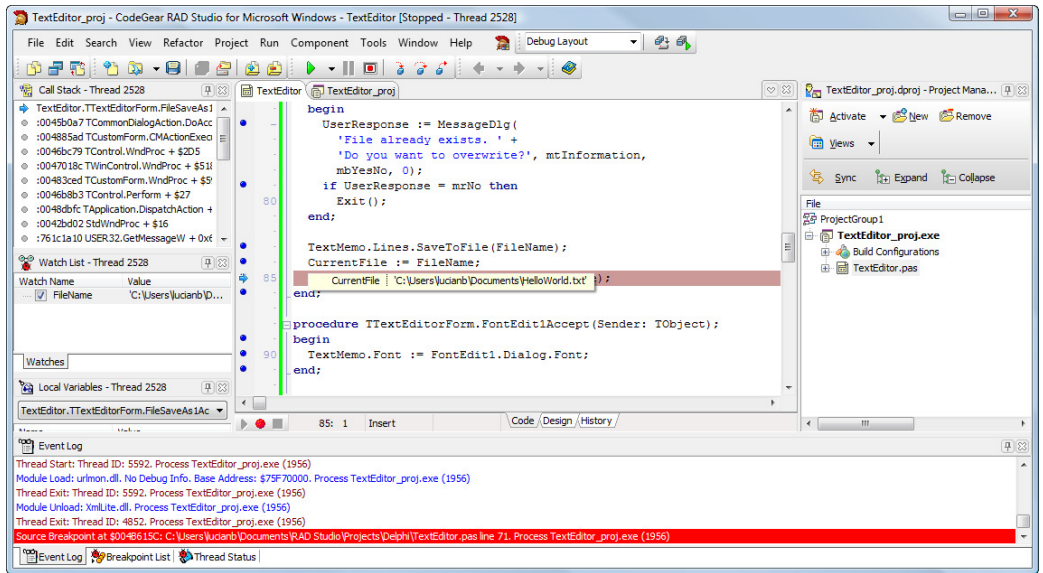


Figure 3-36. Viewing the value of *CurrentFile* (Delphi view)

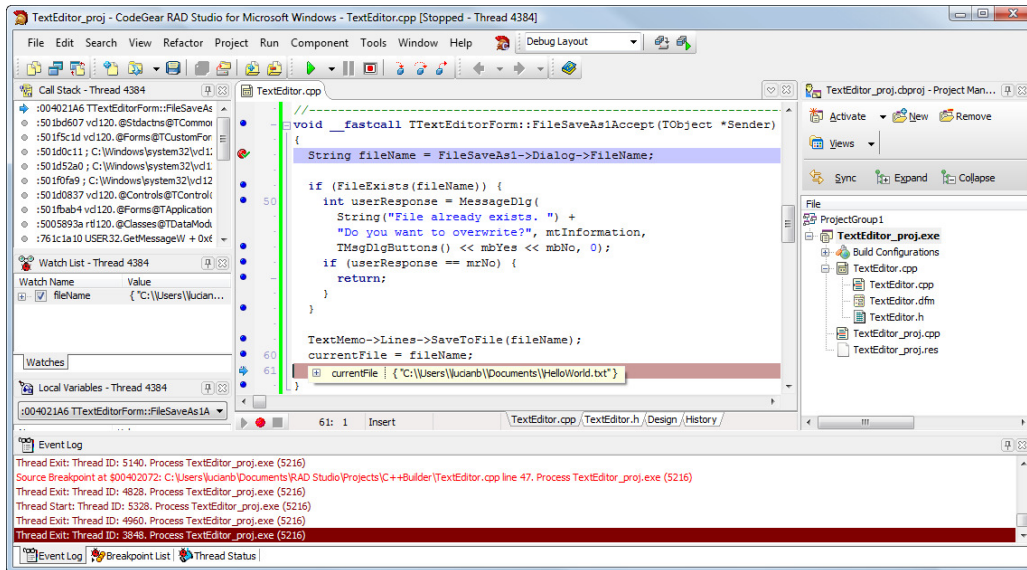


Figure 3-37. Viewing the value of *CurrentFile* (C++Builder view)

To end the debugging session, press the stop button on the **Debug** toolbar, also displayed in the following image.



Figure 3-38. The Debug toolbar

Debugging the application

More Advanced Topics

VCL and RTL

As seen in the previous chapters, CodeGear™ RAD Studio 2009 offers a powerful Integrated Development Environment that makes building native Windows applications extremely easy. The *Visual Component Library* (also known as VCL) offers a large number of visual and nonvisual components that can be used to build almost any desired user interface. Besides the VCL, RAD Studio provides an extensive library of routines and classes, called the *Run Time Library* (known as RTL), that provides the common functionality needed in all applications.

This chapter lists the most important classes, data types, and functions that can be found both in the VCL and RTL.

The most important components of the VCL are:

- A standard set of components that include all controls provided by the Windows UI framework. This set consists of components such as buttons, edits, menus, and so on. The VCL also extends some of these controls, offering you even more functionality than is normally provided by the Windows controls.
- An extended set of components not normally present in the Windows UI framework. These components are built on top of the standard set.
- *Actions*, which is a key concept extensively used in VCL applications, allow you to centralize all the interaction logic of your user interface.

VCL and RTL

- A number of data-aware controls that can be linked to a data source at design time. These components are widely used in database applications.
- *Ribbon controls* that allow you to build the next generation of user interfaces that integrate nicely with the Windows Vista and Microsoft Office 2007 look-and-feel.
- *DBExpress* and *dbGo* database frameworks. These frameworks can be used with all the data-aware controls, simplifying your application development more than ever.
- *Internet Direct*, also known as *Indy*, that provides an extensive number of components used in Internet-connected applications. *Indy* includes client and server components for today's most used communication protocols on the Internet.
- *DataSnap*, which allows you to build distributed applications.
- Easy integration of any exposed *OLE* and *ActiveX* objects in your application. RAD Studio provides a set of tools that allow creating a wrapper component over any public *ActiveX*. This wrapper component can be used as any normal VCL component inside your application.

Even though this is not the full list of components available in the VCL, the above mentioned are the most widely used and appreciated VCL components. To see all available components, check out the Tool Palette in RAD Studio.

The most useful features in the RTL, which are available both in Delphi and C++Builder, are:

- An extensive support for strings. This support includes handling of Unicode strings (the default encoding used by RAD Studio), ANSI and UTF-8 strings, various string handling routines, and much more.
- A large number of date and time manipulation routines.
- Extensive support for file and stream operations.
- Routines and classes that provide Windows API support. You, as a developer, will often be required to use Windows API directly because a certain functionality is not provided by the RTL. RAD Studio provides developers with the ability to use the full Windows API directly. RAD Studio also provides easy-to-use classes like *TRegistry* for registry handling.

- Variant data types and various support routines to make COM integration easy. Variant data types have long been used in Microsoft COM and OLE technologies. Variants are useful when you do not know the exact data type you are operating on. The Delphi language compiler provides native support for Variants, integrating some of the dynamic language concepts, found in other languages such as Java, PHP, and others.
- *Run-time Type Information*, also known as RTTI, that provides an easy way to obtain metadata about types, classes, and interfaces at runtime.

Another important part of the RTL is provided by the generic collections, which is specific to the Delphi language. This collection of generic classes can be used in any application that requires lists, dictionaries, and other container classes. There are also nongeneric counterparts for these classes.

The C++Builder equivalent of the generic collections is given by the STL library, provided by Dinkumware as a third party add-in. This is presented in the next section.

For more information...

See "Win32 Developer's Guide" in the bundled Help.

Third party add-ins

- **Dinkumware (STL)**
- **Boost**
- **Intraweb**
- **Indy**

Dinkumware (STL) is a collection of template libraries for C++, included in C++Builder. It includes containers such as vectors, lists, sets, maps, bitsets. It also includes algorithms for applying widely used operations, like sorting a container or searching inside a container. To implement the algorithms, STL introduces iterators in all five flavors for operating on a container: input, output, forward, backward, bidirectional. *Functors*, or function objects, are also introduced for overloading operators.

Boost is a set of libraries for C++ that significantly expand the language using template meta-programming. A fully tested and preconfigured subset of Boost is included in C++Builder. Include paths have been set for the Boost libraries, and any other necessary libraries should be automatically linked. As an example, to use the Boost *minmax* library, your code should specify:

```
#include <boost/algorithm/minmax.hpp>.
```

Intraweb is a collection of visual components, a framework designed to allow you to create web applications or Apache plug-ins. It allows you to create web applications with the same ease you use the VCL.

Indy is an Open Source group. The Indy project maintains several active Open Source projects which have evolved from the original Indy (Internet Direct) project. Indy offers client and server components using Internet protocols, such as tcp, udp, echo, ftp, http, telnet, and many others. It also provides components for I/O handling, intercepts, SASL, UUE, MIME, XXE encoders, and others.

EDN and Partners

Other resources

EDN

The Embarcadero Developer Network (EDN), located at dn.embarcadero.com, is a collection of code-related articles on various products, including 3rdRail, Turbo Ruby, Blackfish SQL, C++Builder, Delphi, Delphi for PHP, Delphi Prism, Interbase, and JBuilder.

The EDN website keeps an up-to-date calendar of the most important events related to Embarcadero products and also gives the latest news in product updates. As a feature of this website, the calendar can be customized to show the events concerning a single Embarcadero product.

An important developers' resource is the CodeCentral page, as part of the EDN. This is located at the following link: cc.embarcadero.com. CodeCentral is a collection of code snippets contributed by various members, using all the programming languages featured in EDN.

Partners

The companies behind all the included third party add-ins are mentioned in the following list:

- DINKUMWARE Ltd for Dinkumware, at <http://www.dinkumware.com>
- The Boost open source project, at www.boost.org
- ATOZED Software for IntraWeb, at <http://www.atozed.com/IntraWeb>
- The Indy Project, at <http://www.indyproject.org>

A complete list of the RAD Studio 2009 partners can be found at the following links:

- <http://cc.embarcadero.com/partners/delphicpp2009/CBuilder/index.html>
- <http://cc.embarcadero.com/partners/delphicpp2009/Prism/index.html>
- <http://cc.embarcadero.com/partners/delphicpp2009/Delphi/index.html>

2009 04 16



Copyright © 2009
Embarcadero Technologies, Inc.
Download a free trial
at www.embarcadero.com

