Symantec C++

Learning Symantec C++ by Example

Part Three

Introduction to the Tutorial

9

Welcome to Symantec C++ version 7. This section of the manual contains a tutorial designed to introduce you to the important components and features of the Integrated Development and Debugging Environment (IDDE)—the “shell” within which most of your application development takes place.

The tutorial is designed to complement Part Two, “Creating an Application with Symantec C++.” The tutorial provides a quick tour of the IDDE that shows you how to perform the most common tasks. Part Two contains more in-depth information, to show you procedures for less common tasks and alternative ways of accomplishing things.

Prerequisite Knowledge

This tutorial assumes that you are familiar with the Windows environment—that you can start applications from the Program Manager, move and resize windows, operate menus and dialog boxes, and perform simple text editing tasks (such as cut, copy, and paste). The tutorial also assumes some familiarity with C, C++, and Windows programming basics. You need not know anything about the Microsoft Foundation Class (MFC) library; MFC basics are introduced here.

For more information, consult the references listed in Chapter 1, “Introducing Symantec C++.”

The Tutorial Application

The application built in the tutorial lets you read and navigate through hypertext. Two versions are built: a DOS version (in Lesson 1) and a Windows 3.1 version (in Lessons 2-5). Most of the code for the application has been written; the tutorial just shows you certain stages in the development process to familiarize you with IDDE tools.

The hypertext files that the tutorial applications accept as input are text files containing simple commands that control document formatting and show images as well as commands that define links to other such documents. The markup language recognized by the tutorial applications, referred to throughout the tutorials as TML, is a subset of the Hypertext Markup Language (HTML). HTML is a format that has become a standard for information interchange on the World-Wide Web (WWW), a distributed hypermedia system accessible through Internet connections.

Special WWW browser programs enable users worldwide to access and share text, graphics, audio, and other data. The tutorial applications only hint at the richness of full-featured WWW browsers. The DOS TML Reader built in Lesson 1 is called TMLDOS; the Windows version of Lessons 2 through 5 is called TMLRead.

Tutorial Structure

The tutorial comprises five lessons that include instructions for performing various tasks. Following these instructions will teach you basic procedures and familiarize you with IDDE tools. Each lesson builds on concepts and procedures introduced in previous lessons, so it is best to work through the lessons in order.

Lesson 1 teaches you the basics: how to start the IDDE, open Source windows for editing text, and compile and run a program. In addition, the first lesson shows you how to run in debugging mode and perform fundamental debugging tasks. The example program in Lesson 1 is a DOS application; however, the skills you learn are equally applicable to Windows application development.

Lesson 2 shows you how to use AppExpress to generate an application framework for a Windows program. You also learn to use precompiled headers and to use `TRACE` calls within your program to track its progress. Lesson 2 concludes with a brief introduction to MFC and describes the classes that constitute the application framework you generated.

Lesson 3 teaches you how to use the ResourceStudio. You modify the menu and accelerator table generated by AppExpress, and attach a new toolbar bitmap to your application's resources. You then edit the source code to make use of the new toolbar.

Lesson 4 shows you how to use ClassExpress to add message handlers to your application. You add handlers for Windows messages, such as scrolling, mouse button clicks, and keypresses, then monitor the message handlers as the application framework calls them.

Lesson 5 returns to the ResourceStudio, with which you add a menu item to open a simple **Preferences** dialog box. You use ClassExpress to create a new class for the dialog box and add message handlers. Finally, you add code to connect the menu item to the dialog box and to exchange information between the dialog box and the main program.

Tutorial Source Code

The source code for the tutorial is located in `samples\tutorial`, under the directory in which you installed Symantec C++ (by default, this is `c:\sc\samples\tutorial`). The `samples\tutorial` directory contains a subdirectory corresponding to each lesson (these subdirectories are named `lesson1`, `lesson2`, `lesson3`, `lesson4`, and `lesson5`).

Each lesson's subdirectory (except `lesson2`) contains three subdirectories, named `start`, `finish`, and `backup`.

- The `start` subdirectory is your working directory during the tutorial; it contains the project and source code that you change as part of the lesson.
- The `finish` subdirectory contains the project as it should appear after the steps in the lesson are performed correctly.
- The `backup` subdirectory is a copy of the initial contents of the `start` subdirectory. If you want to redo the lesson from scratch, delete the contents of the `start` subdirectory and copy all the files in the `backup` subdirectory to the `start` subdirectory.

The subdirectory for Lesson 2 contains only a `finish` subdirectory. The `start` subdirectory is created as part of the lesson.

The source and executable of the final DOS version of the TML Reader is contained in `tutorial\tmldos`. The source and executable of the final Windows version is located in `tutorial\tmlread`.

Lesson 1: Create the DOS Application

10

In this chapter you learn basic procedures for using the Integrated Development and Debugging Environment (IDDE) while building a DOS version of the TML Reader. This lesson teaches you to:

- Start the IDDE and load a project
- Edit source code
- Build and run the application
- Create a debugging workspace
- Run the application in debugging mode

Most of the code for the application has already been written; during the lesson you will add a few final lines. At the end of the lesson, you will have a DOS executable that can read and display TML files.

Starting the IDDE and Loading a Project

To start the IDDE:

First, start the IDDE. If you haven't yet installed Symantec C++, please see the *Getting Started Guide* for installation instructions.

1. Double-click on the Symantec C++ 16-bit group icon in the Windows Program Manager. (If the Symantec C++ 16-bit group is already open, you may omit this step.)

2. Double-click on the Symantec C++ icon. The IDDE main window opens at the top of the screen (see Figure 10-1).

Figure 10-1 The IDDE main window and Workspace toolbox

The IDDE main window contains a title bar and a menu bar. Below the main window is the Workspace toolbox, used to switch between workspaces (described later in this chapter). The Workspace toolbox is currently docked under the IDDE menu bars. Additional IDDE windows open below the main window and Workspace toolbox. You can open other IDDE windows by selecting their names from the **Goto View** submenu of the **Window** menu; close any window by clicking the Close button in its upper-left corner.

Next, you must load the DOS TML Reader project. (A project is a collection of source and other support files from which an executable is generated.) To open the project:

1. Choose **Open** from the IDDE's **Project** menu.
2. Use the **Open Project** dialog box to find `samples\tutorial\lesson1\start\tmldos.prj`, located under the directory in which you installed Symantec C++.
3. Click OK.

You can view the project's source file list in the Project window. If the Project window is not already open, choose **Project** from the **Goto View** submenu of the **Window** menu.

In the next section you edit one of the source files.

Editing Source Code

The DOS TML Reader's source code needs a minor addition before compilation. To add the missing lines, you must edit the source code in a Source window.

1. In the Project window, double-click on `display.c`.

A Source window opens, showing the contents of the source file (see Figure 10-2).

Figure 10-2 Source window

2. Choose **Function** from the Source window's **Goto** menu. The **Goto Function** dialog box opens.

3. Select `Display` from the Function Name listbox, then click OK. The editor moves to the start of `Display()`.

4. Choose **Find** from the **Edit** menu. The **Find** dialog box opens.

5. In the Pattern textbox, type `LESSON 1`.

6. Click on Next.

7. You should see the following line in the source code:

```
/* INSERT CODE FOR LESSON 1 HERE! */
```

   Just after this comment, but before the procedure's closing brace, insert the following two lines of code:

```
disp_move(disp_numrows-1, 0);
disp_eeol();
```

8. Choose **Save** from the **File** menu.

The DOS TML Reader is now ready to be compiled.

In the next section you learn to build and run the project, and you look at a sample document.

Building and Running the Application

To build and run the DOS TML Reader:

1. From the IDDE's **Project** menu, choose **Rebuild All**.

The Output window opens automatically. This window informs you of build progress and displays any warning or error messages. If no errors exist, you see the message "Successful build." (You can still work in the IDDE while the build is in progress, because the process of building is multitasked. The Output window can be behind other IDDE windows, even when messages are being written to it.)

2. From the IDDE's **Project** menu, choose **Arguments**.
   The **Run Arguments** dialog box opens.

3. Type `sample.tml` into the textbox and click OK.

4. From the **Project** menu, choose **Execute Program**.

The TML Reader opens in full-screen mode, showing the formatted contents of `sample.tml` (see Figure 10-3). You can use Page Up, Page Down, and the arrow keys to scroll through the file. To execute a hyperlink, position the cursor over text shown in reverse video and press Enter. Press Escape to exit the program and return to the IDDE.

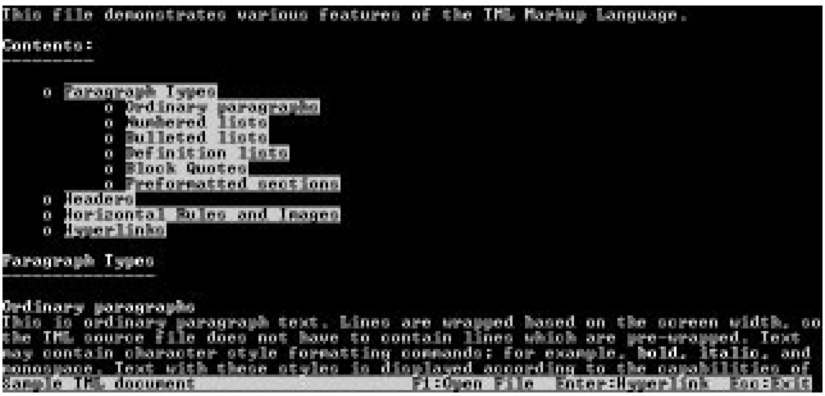

Figure 10-3 The DOS TML Reader

The DOS TML Reader is believed to be without any significant bugs. However, most applications have bugs during at least some of their development phase. To help you locate and correct incorrect code as quickly as possible, the IDDE has powerful debugging features.

The following sections show you how to set up a workspace for debugging and debug your source code.

Setting Up a Workspace for Debugging

A workspace is an arrangement of windows on the screen. You can set up several useful configurations of IDDE windows, then switch between them by clicking on workspace names in the Workspace toolbox.

The IDDE predefines a workspace named Debugging, which contains a configuration of windows useful for typical debugging sessions. Here, however, we define a new workspace specific to this lesson, to show you how this is done and to avoid disturbing any customizations you may have already made to the Debugging workspace.

To create a workspace for debugging:

1. Choose **New** from the **Workspace** submenu of the IDDE's **Environment** menu.
2. When prompted for a workspace name, type `Lesson1`.
3. Click OK.

The IDDE creates a new workspace and adds the workspace name to the Workspace toolbox. Initially the new workspace is empty, with the exception of a few toolboxes that are not needed here. To close the toolboxes, click the Close buttons in the upper-left corners.

Next, you need to add windows to the workspace. During debugging, the Source window follows program execution, the Function window views a list of functions in a particular program module, the Data/Object window views program data, and the Call window displays the program call chain.

1. To open the Project window, press Ctrl+Shift+P.
2. In the Project window, double-click on `main.c`. A Source window opens.
3. To open the Function window, press Ctrl+Shift+F.
4. To open the Data/Object window, press Ctrl+Shift+D.

5. To open the Call window, press Ctrl+Shift+L.

Other debugging windows are available as well; see Chapter 24, “Commands Available in Debugging Mode,” for more information.

Arrange the windows the way you want them (one example is shown in Figure 10-4), then choose **Save Workspace Set** from the **Workspace** submenu of the IDDE’s **Environment** menu.

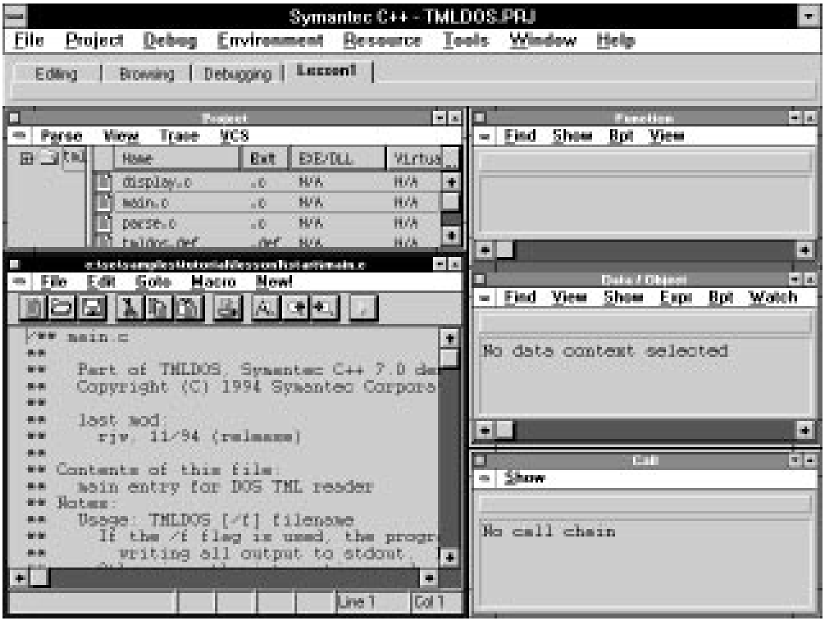

Figure 10-4 A possible Lesson 1 workspace

Now that the workspace is ready, you can begin debugging. The following section shows you how to run the application in debugging mode and how to set a breakpoint on a function, how to view program data, and how to step through code.

Running in Debugging Mode

To execute the application in debugging mode, click in the Source window, then choose **Start/Restart Debugging** from the IDDE’s **Debug** menu. Several things happen:

1. The IDDE opens a window (the Symantec Application Window), in which the application runs. This window may be behind other windows.

2. The application is executed to the start of `main`.

3. The Source window changes to debugging mode (see Figure 10-5). Arrows indicating execution points within functions, and flags indicating breakpoints are shown in the left margin. You cannot edit the code while in debugging mode.

Figure 10-5 Source window in debugging mode

4. The Project window changes to debugging mode (see Figure 10-6). The icons next to the source module names change to indicate their status:

- A bug symbol indicates that the module contains debugging information.
- A small “T” indicates that tracing is enabled in the module.
- A green dot indicates that a breakpoint is set and enabled in the module.

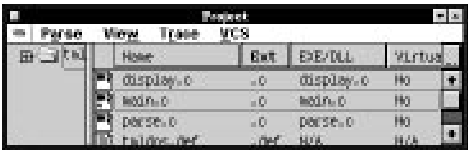

Figure 10-6 Project window in debugging mode

5. The Function window is updated to show a list of functions in the program (see Figure 10-6). An arrow

next to `main` indicates that it is in the call chain; a diamond indicates that a breakpoint is set.

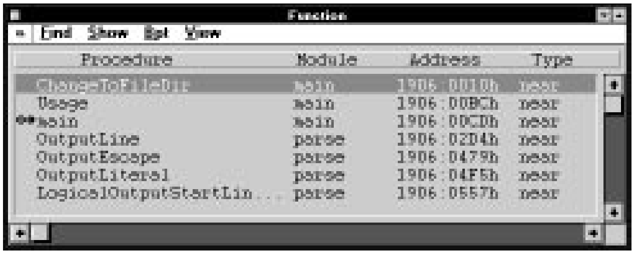

Figure 10-7 Function window

6. The Call window is updated to show the call chain and execution status (see Figure 10-8).

Figure 10-8 Call window

You can now perform the following simple debugging tasks.

Setting and running to breakpoints

To set a breakpoint on a function and execute to it:

1. Click and drag `display.c` from the Project window to the Source window.

2. Choose **Set/Clear Breakpoint** from the **Bpt** menu. A breakpoint is set at the start of `ShowScreen`.

3. Choose **Go Until Breakpoint** from the IDDE's **Debug** menu. The program is executed to the start of `ShowScreen`.

The Source window shows the line at which execution stopped. The Call window shows functions in the call chain (see Figure 10-9).

Figure 10-9 Call chain to function ShowScreen

Viewing data

You can view program data using the following commands:

1. Choose **Data** from the **Show** submenu of the Source window's pop-up menu. The Data/Object window is updated to show the local data in `ShowScreen` (see Figure 10-10).

Figure 10-10 Data/Object window showing local data

2. In the Call window, click on `main`, then choose **Data** from the **Show** menu. The Data/Object window is updated to show the local data in `main`.

3. Choose **Local/Global Data** from the Data/Object window's **View** menu. The Data/Object window is updated to display global data accessible in the current module. Choose **Local/Global Data** again to resume display of local data.

Stepping through code

To step through the source code line-by-line:

1. Choose **Step Into** from the IDDE's **Debug** menu (or press F8) seven times. The IDDE executes seven lines of code. The seventh step takes execution into `ShowLine`;

at that point the Call window adds `ShowLine` to the call chain, and the Data/Object window is updated to display the function's local data.

2. Choose **Return from Call** from the IDDE's **Debug** menu to execute the rest of `ShowLine` and return to the calling function.

3. Choose **Step Over** from the IDDE's **Debug** menu (or press F10) several times. The IDDE executes some lines of code in the function, but it does not step into subroutines. `ShowScreen` is in a loop that writes lines to the screen one-by-one. The results appear in the Symantec Application Window.

Running to the end

To execute the rest of the program, ignoring breakpoints, choose **Go Until End** from the IDDE's **Debug** menu. Bring the Symantec Application Window to the front and test the program as usual. You can press Alt+Ctrl+SysRq to break to the debugger, or let the program terminate before resuming your debugging session. (In the DOS TML Reader, you must press Escape to end the program.)

Ending the debugging session

To exit debugging mode, choose **Stop Debugging** from the IDDE's **Debug** menu. You can do this at any time during debugging; you don't have to choose **Go Until End** first.

In this chapter you have learned how to start the IDDE, load a project, edit source code, build and run the application, and run in debugging mode. These are the fundamental tasks you perform repeatedly as you develop your own applications, whether you are developing in C or C++, for DOS or for Windows. In the following lessons, you learn about the IDDE's more advanced features, many of which are designed specifically for Windows application development in C++.

Lesson 2: Generate an Application Framework

11

In this chapter you begin the process of building a Windows version of the TML Reader. This version is built around a Microsoft Foundation Class (MFC) version 2.5 Single Document Interface (SDI) framework. In this lesson you:

- Use AppExpress to generate a new project containing the SDI framework
- Build and run the bare application framework, a standardized skeleton composed of C++ classes derived from MFC library base classes.
- Use precompiled headers to speed build time
- Add calls to the TRACE macro with Class Editor
- Follow TRACE output in the Trace Messages window

At the end of the lesson you will have a working application framework with menus, a toolbar, and a status bar. The framework itself will respond to certain commands, but the functionality needed to read and display TML files is not yet included. In subsequent chapters you modify the user interface for the TML Reader and add the necessary file input and display routines.

There are many ways to create an application framework. This lesson provides the most straightforward way.

Generating the Framework

In this lesson you use AppExpress to generate a new project containing an application framework. To start AppExpress:

1. Start the IDDE and close any open project by choosing **Close** from the **Project** menu.

2. Start AppExpress by choosing **AppExpress** from the **Tools** menu.

The AppExpress window opens (shown in Figure 11-1).

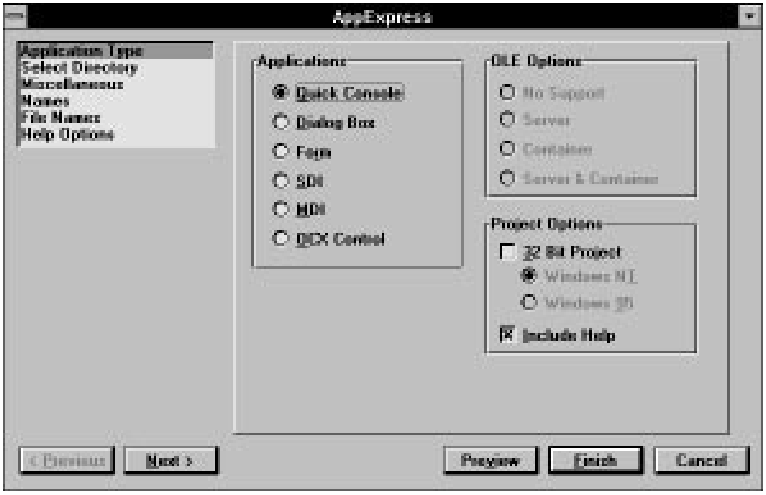

Figure 11-1 Setting up an SDI application with AppExpress

AppExpress contains six pages of options that together define the project to be generated. You define these options in six steps, listed in the upper-left portion of the window. For this project, you need to change options on only four of the six pages. Set up the project options as follows:

1. On the Application Type page (shown when AppExpress is started), select SDI.

2. Deselect Include Help to suppress generation of Windows Help support.

3. Click on Next to switch to the Select Directory page. Change to the `samples\tutorial\lesson2` directory under the directory in which you installed Symantec C++, then click on Create New Directory.

4. Type `start` in the textbox and click Create.

Note

To avoid filename conflicts, it is a good idea to keep each project you create in a separate directory.

5. Click on Next to switch to the Miscellaneous page. Type your company name and suffix (or your own name) in the appropriate textboxes. This information is displayed in the automatically generated **About** dialog box, and in comments at the beginning of each source file.

6. In the Project Name textbox, type `TMLRead`.

7. Click on Next to switch to the Names page. You can customize the names of automatically generated classes here. From the Name drop-down list, select CTMLReadApp. In the Edit textbox, type `CTMLReadApp`.

8. Click Finish.

AppExpress generates a new project in the directory you created. The directory contains:

- Source and header files for the MFC-derived classes
- Resource script and binary files
- Project options and other support files

After generating the project, AppExpress closes and control returns to the IDDE. The new project is loaded automatically. The IDDE parses the new project's source files for browsing with the Class and Hierarchy Editors; parsing progress is displayed in the Output window.

Building and Running the New Project

Next you build the project and learn what the default application framework can do.

1. From the IDDE's **Project** menu, choose **Rebuild All**.

2. From the **Project** menu, choose **Execute Program**.

The IDDE minimizes itself and the TMLRead application opens (shown in Figure 11-2).

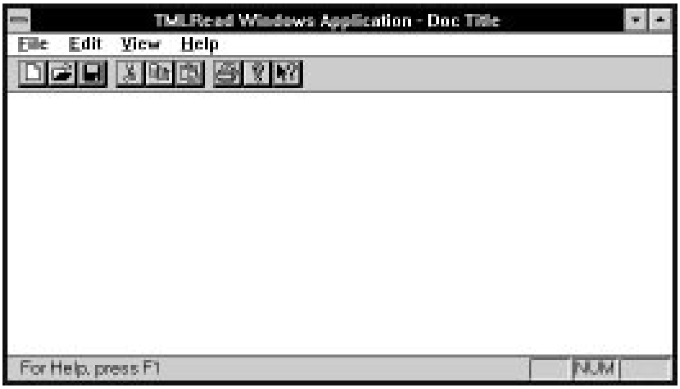

Figure 11-2 New application framework generated by AppExpress

At present, default functionality for several menu commands is provided by the MFC base classes. For example, if you choose **Open** from the **File** menu, a standard Windows **File Open** dialog box opens. You can select a file in this dialog box, but the code needed to read data from the file is not yet installed.

To close TMLRead and return to the IDDE, choose **Exit** from the **File** menu.

The project can take a considerable amount of time to compile. Next you learn how to decrease compilation time by using precompiled headers.

Using Precompiled Headers

In the last section, over 70,000 lines of code were read during compilation. Many of these lines are in Windows and MFC header files that are changed infrequently (if ever), but still must be included by almost every source file. To speed compilation, you can precompile header files; thereafter, the symbols generated by the compiler can be loaded directly.

To precompile the Windows and MFC header files:

1. Choose **Settings** from the **Project** menu. The **Project Settings** dialog box opens.

2. Click on the Build tab.
3. In the left listbox, click on Header Files.
4. In the Precompile section of the right pane, select Specific Header.
5. In the textbox below the Specific Header selection, type `stdafx.h`.

Note

You can specify multiple specific headers to precompile by entering in this textbox a list of their names separated by semicolons or spaces.

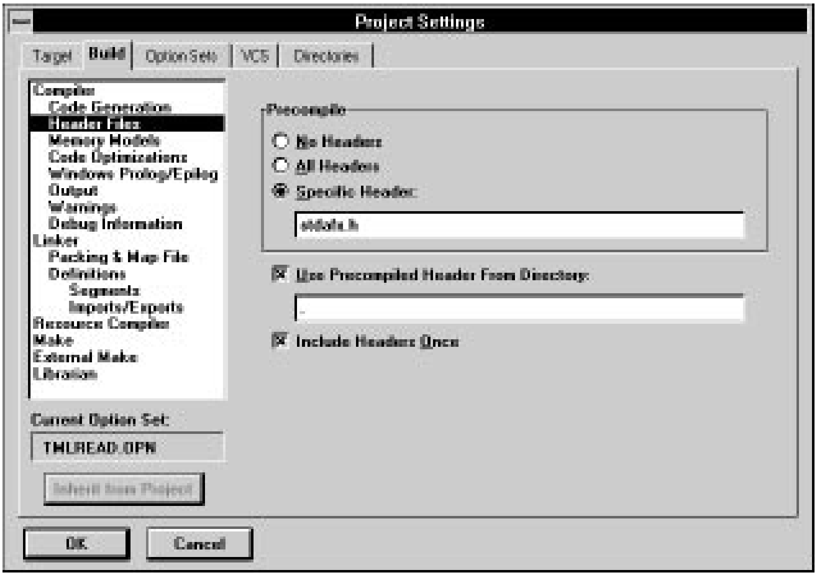

Figure 11-3 Using precompiled headers

6. Click OK.
7. An Editor/Browser message box is opens, noting that project settings have changed and asking if you want to reparse all files. Click No, because you are about to build in the next step, which will stop any parse in progress.
8. From the IDDE **Project** menu, choose **Rebuild All**.

During the rebuild, the header file `stdafx.h` is precompiled, then the source files are compiled using the precompiled symbol table. You should notice a significant reduction in the number of lines compiled, as well as a corresponding increase in compile speed.

Now that you have created, compiled, and executed the application framework, the rest of the chapter helps you to understand the framework's structure. The following three sections are optional.

Adding TRACE Calls with Class Editor

In the remainder of this chapter, you investigate calls to member functions of the MFC-derived classes created by AppExpress to understand the structure of the application framework and the relationships between the classes. To do this, you use the Class Editor to insert calls to the MFC global `TRACE` macro into the member functions, then watch the output in the Trace Messages window.

To add a `TRACE` call to the application class's constructor:

1. Choose **Class Editor** from the **Goto View** submenu of the IDDE **Window** menu. The Class Editor window opens (see Figure 11-4).

2. Under Classes, click on CTMLReadApp. The application class's members appear in the Members list.

3. Under Members, double-click on CTMLReadApp. The application class constructor's source code is shown in the source pane.

4. Add the following line to the constructor:

   ```
   TRACE("CTMLReadApp::CTMLReadApp()\n");
   ```

   The constructor should now appear as follows:

```
////////////////////////////////////////////////////////////////////
// CTMLReadApp construction

CTMLReadApp::CTMLReadApp()
{
    TRACE("CTMLReadApp::CTMLReadApp()\n");
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}
```

5. Press Ctrl+S to save the modified constructor.

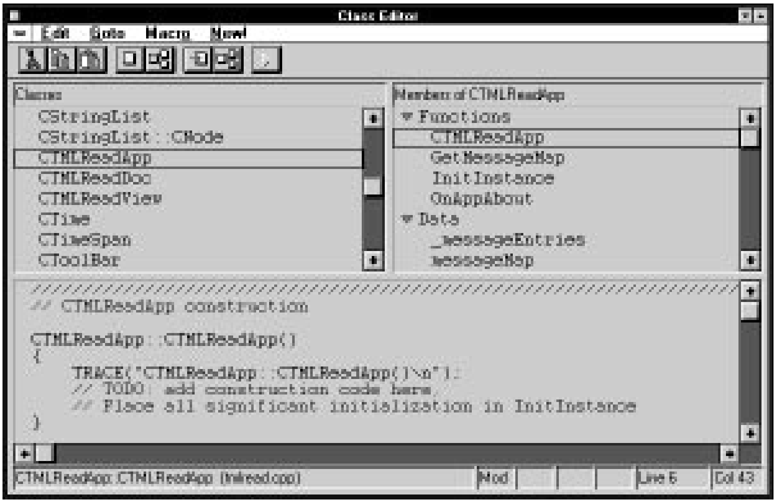

Figure 11-4 Adding TRACE calls with Class Editor

You can add similar TRACE calls to these other member functions:

- `CTMLReadApp::InitInstance()`
- `CMainFrame::CMainFrame()`
- `CMainFrame::~CMainFrame()`
- `CMainFrame::OnCreate()`
- `CTMLReadDoc::CTMLReadDoc()`
- `CTMLReadDoc::~CTMLReadDoc()`
- `CTMLReadDoc::OnNewDocument()`
- `CTMLReadDoc::Serialize()`
- `CTMLReadView::CTMLReadView()`
- `CTMLReadView::~CTMLReadView()`
- `CTMLReadView::OnDraw()`

When you have finished, choose **Build** from the IDDE **Project** menu to recompile the source files you have changed.

In the next section you execute the application and watch the TRACE output in the Trace Messages window.

Watching TRACE Output in the Trace Messages Window

To watch TRACE output, first you must set up the Trace Messages window to receive and display TRACE messages.

1. Choose **Trace Messages** from the **Goto View** submenu of the IDDE's **Window** menu. The Trace Messages window opens.

2. Choose **Output to Window** from the **Options** menu of the Trace Messages window.

3. Choose **MFC Debug Messages** from the **Options** menu. The **MFC Trace Debug Options** dialog box opens.

4. Check Enable Tracing and uncheck any other options that are checked. Click OK.

You are now prepared to run the application and view the output of the TRACE macro calls.

1. Choose **Execute Program** from the IDDE's **Project** menu.

2. The IDDE is minimized and the application window opens. Double-click on the IDDE icon to reopen the IDDE windows. Position the windows so you can watch the Trace Messages window while the program executes.

The Trace Messages window has already received several messages from the application's class constructors and initialization code (Figure 11-5). You see more messages if you choose **Open** or **New** from TMLRead's **File** menu, or when the window needs to be repainted.

When you close TMLRead, the Trace Messages window receives messages from the application's class destructors.

Figure 11-5 TRACE output in the Trace Messages window

The next section gives an overview of the classes in the application framework and explains the messages you see in the Trace Messages window.

The Application Framework and MFC Classes

In this chapter, you have used AppExpress to build an application framework, a skeleton on which you can build a Windows application, consisting of C++ classes contained in and derived from classes in the Microsoft Foundation Class (MFC) library.

The MFC library is a C++ class library that supports programming for Windows. It encapsulates most of the Windows Application Programming Interface (API), and provides additional C++ programming support such as container and string classes. The MFC library makes it easy to work with Windows elements in an object-oriented manner. For example, MFC library classes exist to represent objects such as windows, dialog boxes, controls, device contexts, Graphic Device Interface (GDI) objects, and so on. Windows API functions are implemented as member functions of the classes with which they are logically associated.

TMLRead is built on a Single Document Interface (SDI) framework. The SDI framework contains five fundamental objects:

- Document: The document contains data, reads the data from disk, and provides access to the data. In TMLRead, the document is an object of type CTMLReadDoc, derived from the MFC library class CDocument.

- View: The view displays the data contained by the document. In TMLRead, the view is an object of type CTMLReadView, derived from the MFC library class CView (which is, in turn, derived from CWnd, the base class for all types of windows).

- Document Template: The document template defines the association between document and view classes. In an SDI application, this is an object of type CSingleDocTemplate.

- Frame Window: The frame window object is the application's main window. In TMLRead it contains a toolbar, a status bar, and the view. The frame window is an object of type CMainFrame, derived from the MFC library class CFrameWnd.

- Application: The application object creates and controls the other objects, and takes care of general program initialization and cleanup. In TMLRead the application is an object of type CTMLReadApp, derived from the MFC library class CApplication.

The framework itself provides standard user-interface implementations for some commands (for example, file open and save); you must add support for certain framework functions (such as file input and output), as well as add other command and message-handling capability specific to your application.

The `TRACE` output from the previous section lets you see the creation, use, and destruction of objects in the application. For example, when you start the application, you see the following messages:

```
[00001] NOTIFY(StartTask)
[00002] CTMLReadApp::CTMLReadApp()
[00003] CTMLReadApp::InitInstance()
[00004] CTMLReadDoc::CTMLReadDoc()
[00005] CMainFrame::CMainFrame()
[00006] CMainFrame::OnCreate()
[00007] CTMLReadView::CTMLReadView()
[00008] CTMLReadDoc::OnNewDocument()
[00009] CTMLReadView::OnDraw()
```

The application object is created first. Using a document template, the application object creates the document, frame window, and view objects. It then calls the document object to set up a new document. Finally, when the window is shown, the framework calls the view's `OnDraw()` function to repaint the window.

If you choose **New** from TMLRead's **File** menu, you see the following messages:

```
[00010] CTMLReadDoc::OnNewDocument()
[00011] CTMLReadView::OnDraw()
```

Or, if you choose **Open** from the **File** menu, then select a file, you see:

```
[00012] CTMLReadDoc::Serialize()
[00013] CTMLReadView::OnDraw()
```

The document object is called either to create a new document or to read a document from a file, depending on the menu item that was chosen. Note that neither the document object nor the view object is destroyed; in an SDI application, they are reused continually.

When you choose **Exit** from TMLRead's **File** menu, the application's objects are destroyed in reverse order in which they were created. You see these messages in the Trace Messages window:

```
[00014] CTMLReadView::~CTMLReadView()
[00015] CMainFrame::~CMainFrame()
[00016] CTMLReadDoc::~CTMLReadDoc()
[00017] NOTIFY(ExitTask)
[00018] NOTIFY(DelModule)
```

There is no message from the application object's destructor, because it is not defined explicitly in the framework.

You may find it useful to continue to add `TRACE` calls to the application as it is built. It is often difficult to follow the workings of a message-driven system; the Trace Messages window, however, acts as a kind of passive debugger that keeps you informed of the internal workings of your application.

In the next chapter, you begin to shape the application framework to the specific needs of the application. The first step in this process is to customize the user interface with the Resource Editor.

Lesson 3: Customize the Interface 12

After an application framework has been generated with AppExpress, it is usually necessary to customize the user interface. This typically is an ongoing process; as your application evolves, you can add and remove interface elements. In this lesson you perform the initial stage of customization. You will:

- Use ResourceStudio to customize TMLRead's menu and toolbar
- Edit the code that sets up the toolbar

AppExpress generates an SDI application framework with a full set of resources. TMLRead needs only a subset of the standard user interface, so most of the work in this lesson involves trimming down the interface. At the end of the lesson you will have a modified version of TMLRead, supporting only the commands that are needed.

Before starting this lesson, start the IDDE and open the project `tmlread.prj` found in directory `samples\tutorial\lesson3\start`.

Launching ResourceStudio

An application's resources are contained in a resource script file, which includes descriptions of menus, dialog boxes, and other user interface elements. The resource script file is edited with ResourceStudio. To launch ResourceStudio:

1. Open the Project window by pressing Ctrl+Shift+P.
2. In the Project window, double-click on `tmlread.rc`.

3. When asked if you want to use ResourceStudio to edit this file, click Yes. ResourceStudio starts, with its Shell window minimized. (The Shell window is a control center from which you can create and open resource files. You will use it later in this lesson.)

The Browser window of ResourceStudio opens (Figure 12-1). The window controls editing of the resources within an individual resource file. On startup, its upper-left pane contains a list of resource types found in `tmlread.rc`.

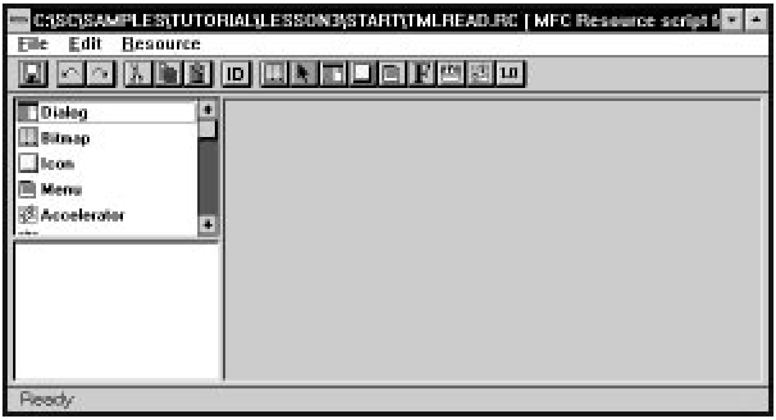

Figure 12-1 The Browser window of ResourceStudio

The TML Reader is only able to read and display TML files; it cannot edit or write them. Thus, you can remove menu items related to editing and saving.

Customizing the Menu

In this section you customize TMLRead's menu.

Within ResourceStudio are several individual editors, each capable of editing a single type of resource. For example, the Dialog editor is used to edit dialog box resources, and the Menu editor to edit menu resources. By default, the editors use the right pane of the Browser window.

To start the Menu editor:

1. Click on Menu in the Browser window's upper-left pane.
2. Double-click on IDR_MAINFRAME in the lower-left pane.

The Menu editor opens in the Browser window (Figure 12-2). Its menu replaces the Browser window's menu, and its toolbar appears at the top of the right pane.

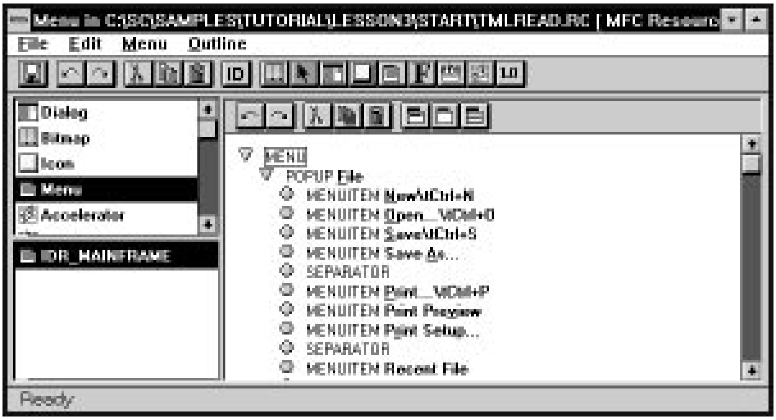

Figure 12-2 The Menu editor in the Browser window

As the Menu editor opens, the Property Sheet also opens (Figure 12-2). This window is used to edit resource and resource element properties and options.

Figure 12-3 The Property Sheet

The Test Menu pop-up window also opens simultaneously with the Menu editor (Figure 12-2). It is a top-level window whose menu is identical to the one you are editing. This window does nothing when you choose menu items that are not top-level menu items or submenus.

Because changes you make to the menu in the Browser window are immediately reflected in the menu of the Test Menu window, you can verify your changes as you make them.

Figure 12-4 The Test Menu window

Now you can remove unnecessary menus and menu items.

1. Click on "MENUITEM New" to select this menu item.
2. Press Delete.
3. Repeat the above steps to delete:

- "MENUITEM Save"
- "MENUITEM Save As"
- "POPUP Edit"
- "MENUITEM Index"
- "MENUITEM Using Help" and the following "SEPARATOR"

If you make a mistake, you can correct it by choosing **Undo** from the **Edit** menu.

To close the Menu editor, choose **Close Editing** from the **File** menu, or press Escape. The Browser window menu and toolbar return. The right pane shows a preview of the menu.

Although we have removed menu items for commands that will not be supported, at present it would still be possible to access some of those commands through their accelerators. In the next section we remove those commands from the accelerator table.

Customizing the Accelerator Table

To start the Accelerator Table editor:

1. Click on Accelerator in the Browser window's upper-left pane.

2. Double-click on IDR_MAINFRAME in the lower-left pane.

The Accelerator Table editor opens in the Browser window (shown in Figure 12-5). As with the Menu editor, the Accelerator Table editor's menu replaces the Browser window's menu, and its toolbar appears at the top of the right pane.

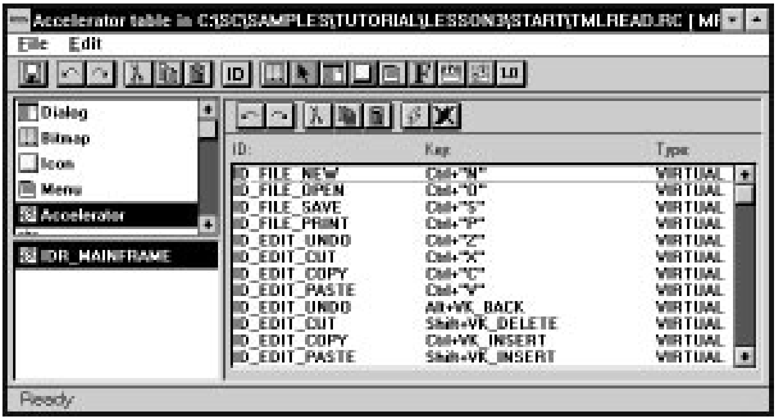

Figure 12-5 The Accelerator Table editor in the Browser window

Removing accelerators is quite similar to removing menu items. To remove the unnecessary accelerators:

1. Click on ID_FILE_NEW.
2. Press Delete.
3. Repeat the above steps for all the accelerators, leaving only ID_FILE_OPEN and ID_FILE_PRINT.

As in the Menu editor and elsewhere in ResourceStudio, if you make a mistake, you can correct it by choosing **Undo** from the **Edit** menu.

To close the Accelerator Table editor, press Escape.

Commands may be sent to TMLRead in one other way: by clicking on buttons in the toolbar. In the next section we replace the toolbar bitmap with one containing only commands that are supported by TMLRead.

Importing a New Toolbar Bitmap

Before importing a new toolbar bitmap, delete the current toolbar bitmap:

1. Click on Bitmap in the Browser window's upper-left pane.
2. Click on IDR_MAINFRAME in the lower left-pane.
3. Press Shift+Delete.

You must now copy a bitmap from another resource script file. To do this, you open a second Browser window to view the contents of a resource script file containing the new toolbar bitmap. Then you copy the bitmap to the clipboard and paste it into `tmlread.rc`.

1. Double-click the ResourceStudio icon to restore the Shell window.
2. In the Shell window, choose **Open** from the **File** menu.
3. Select the file `newbar.rc` and click OK.
4. In the upper-left pane of the new Browser window, click on Bitmap.
5. In the lower-left pane, click on IDB_NEWBAR. A preview of the bitmap appears in the right pane (Figure 12-6).

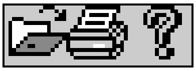

Figure 12-6 New toolbar bitmap

6. Choose **Copy** from the **Edit** menu. The bitmap is copied to the clipboard.
7. Choose **Close** from the **File** menu. The second Browser window closes. Activate the Browser window containing `tmlread.rc` by clicking on it.
8. Choose **Paste** from the **Edit** menu. The bitmap is added to your application's resources.

The new toolbar does not presently have the correct resource ID. For the application framework to use it correctly, this toolbar's resource ID must be identical to that of the application's menu and accelerator table resources. To set the new bitmap's resource ID:

1. Click on the Property Sheet to bring it to the front. The Property Sheet shows the resource ID and memory options for the bitmap resource.

2. From the ID drop-down list, select IDR_MAINFRAME.

The initial resource customization is now complete. Next you exit ResourceStudio and make a minor adjustment to update the source code that loads the toolbar.

Exiting ResourceStudio

To save your work and exit Resource Studio:

1. Choose **Save** from the Browser window's **File** menu.

2. Choose **Close** from the **File** menu. The Browser window closes, but the Shell window and Property Sheet remain open.

3. Choose **Exit** from the Shell window's **File** menu.

Setting Up the New Toolbar

The MFC framework uses a single bitmap to represent a set of toolbar buttons. The correspondence between the button images in the bitmap and the commands they represent is established in an array in the source code. Because you have replaced the original toolbar bitmap, you also must update this array.

1. In the Project window, double-click on `mainfrm.cpp`. A Source window opens, displaying the frame window class's source code.

2. Scroll downward a few lines until you find the following code:

```
// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] = {
    // same order as in the bitmap 'toolbar.bmp'
    ID_FILE_NEW,
    ID_FILE_OPEN,
    ID_FILE_SAVE,
        ID_SEPARATOR,
    ID_EDIT_CUT,
    ID_EDIT_COPY,
    ID_EDIT_PASTE,
        ID_SEPARATOR,
    ID_FILE_PRINT,
    ID_APP_ABOUT,
    ID_CONTEXT_HELP,
};
```

3. Remove and rearrange lines until the array looks like this:

```
// toolbar buttons - IDs are command buttons
static UINT BASED_CODE buttons[] = {
    // same order as in the new toolbar bitmap
    ID_FILE_OPEN,
        ID_SEPARATOR,
    ID_FILE_PRINT,
        ID_SEPARATOR,
    ID_APP_ABOUT,
};
```

4. Save the code by choosing **Save** from the Source window's **File** menu.

The `buttons[]` array now contains the command IDs of the three toolbar buttons. The ID_SEPARATOR entries indicate where spaces should be placed between the button images when drawing the toolbar.

In the last section you build the application and view the results of your labor.

Building and Running the Application

To build and run the application, choose **Execute Program** from the IDDE **Project** menu. Because no executable has yet been built, the IDDE automatically builds TMLRead and then runs it. (When you try to run an existing executable that needs to be rebuilt, the IDDE asks if you want to rebuild the program before running it.)

The source files are recompiled, the resource script is compiled by the resource compiler, and all the object files and resources are linked together to create the final application. Then the IDDE minimizes itself and executes the application (Figure 12-7).

You can verify that the changes you made to the menu and toolbar are correct. To close the application, choose **Exit** from the **File** menu.

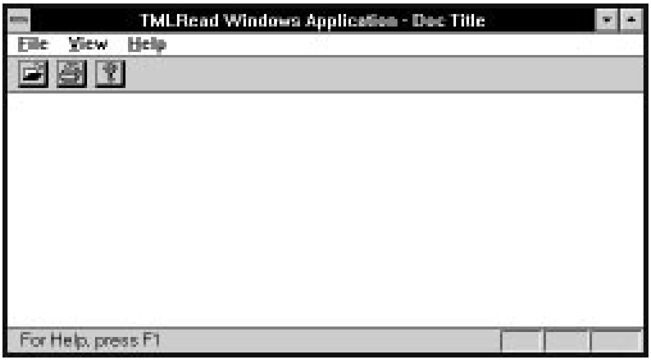

Figure 12-7 TMLRead with customized menu and toolbar

You now have completed initial customization of TMLRead's resources. You deleted excess menu items and accelerators and installed a new toolbar bitmap. More changes to the application's resources can be made as the need arises; in Lesson 5, for example, you will return to ResourceStudio to add support for a Preferences dialog box. In the next lesson, though, you will use ClassExpress to install functions to handle Windows messages, such as those generated by key presses and button clicks.

Lesson 4: Add Messages with ClassExpress 13

In Lessons 2 and 3, you learned how to generate an application framework and customize its user interface. However, the behavior of the resulting application is still generic, because it is supplied entirely by methods of MFC base classes. The inherited methods that handle Windows messages, in particular, often do nothing more than call `DefWindowProc`.

In this lesson, you use ClassExpress to expand the functionality of the TML Reader and add member functions that handle Windows messages announcing user actions. Member functions that handle Windows messages are called message handlers, or just handlers. You can think of them as callbacks, which are called by MFC when the message they are intended to process is received. A message handler is called in response to only one message.

The Windows message stream is the lifeblood of a Windows application, regardless of whether the application is constructed in an object-oriented or a procedural way. A well-behaved Windows application can interact with the user only if it taps into the message stream and responds to messages in nondefault ways. Using ClassExpress to add MFC message handlers demonstrates how easy it is to enhance the behavior of your MFC application.

In this lesson, you:

- Launch ClassExpress from the IDDE
- Use ClassExpress to add message handlers
- Edit message handlers in the IDDE Source window
- Rebuild and run the application

At the conclusion of this lesson, you will have:

- Seen how Windows messages are handled in an MFC application
- Used ClassExpress to perform the purely administrative chores associated with handling Windows messages

The next section provides a brief introduction to Windows message handling in MFC to help you understand how messages are handled in TML Reader.

Windows Message Handling in MFC

Regardless of how it is written, a Windows application receives messages that inform it of user actions and their consequences, changes in the state of other applications, or changes in the state of Windows itself. MFC transforms this message-driven model into the object-oriented model defined by its class hierarchy, thereby introducing several improvements to the design of applications. To discuss this transformation and its benefits, it is necessary first to review how messages are handled in a traditional Windows program, and to identify the shortcomings of that approach.

Message handling in a traditional Windows application

In a traditional Windows program written in C, you handle a Windows message by adding code directly to the window procedure to which that message is dispatched by your application's message loop. The window procedure usually consists of a large `switch` statement whose cases are different messages.

For example, to handle the `WM_SIZE` message, you must add a `case WM_SIZE` to the `switch`. The statements following that `case` statement must have to unpack and coerce the window procedure's parameters, `wParam` and `lParam`, into constituent parts in a way that is specific to the `WM_SIZE` message. The result would look much like the following:

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT msg,
   WPARAM wParam, LPARAM lParam )
{
   // local and static variables...

   switch ( msg )
   {
      // ...

      case WM_SIZE:
      {
         UINT nType = (UINT)wParam;
         int  cx    = LOWORD( lParam );
         int  cy    = HIWORD( lParam );
         // Now do what you really want to do
         // ...
      }
      break;

      // ...

      default:
         break;
   }

   return DefWindowProc( hwnd, msg,
                         wParam, lParam );
}
```

The `switch` statement usually sprawls across several pages, and the bodies of `case` statements often spill over the right margin of the page. Data that must be preserved across messages or shared between cases is usually stored in automatic or static variables visible to every `case`. The result enjoys none of the benefits of encapsulation or data hiding afforded by C++, and it is difficult to read, comprehend, and maintain.

MFC's design

In MFC, window procedures are part of MFC itself; you do not edit them. Instead, the window procedures route each message to a handler—a member function of some class—whose purpose is to process that message. MFC provides default handlers, which collectively define the default behavior of an MFC application. You supply handlers only for those messages you want to process. Each handler you supply is a member function of one of your derived classes and overrides the inherited handler in the base class.

However, you will sometimes find that your handler can call the inherited handler to perform the bulk of its work.

For example, to handle the WM_SIZE message, use ClassExpress to add a handler for this message to your View class. (By convention, the handler for the WM_SIZE message is named OnSize.) Whenever a WM_SIZE message is received by a window represented by an object of your View class, your handler is called. The prototype of this handler is as follows:

```
void OnSize(UINT nType, int cx, int cy);
```

Notice that the parameters of OnSize contain the same information that the WM_SIZE message bears in the wParam and lParam window procedure arguments, but in an immediately usable form. No unpacking or typecasting is required. In general, the signature of each handler—its return type, the number of its arguments and their types—is specific and appropriate to the message it processes. MFC parses the messages before calling the handler.

To route Windows messages to handlers, MFC consults tables, called message maps, that pair Windows messages with the member functions that handle them. You never have to edit message maps manually because AppExpress and ClassExpress create and maintain them for you. AppExpress creates message maps when you generate an application framework. Their definitions are placed in the implementation (.cpp) files of your derived classes. When you use ClassExpress to add a handler to a class, the message map is updated automatically with a new entry associating the chosen message with your handler. Furthermore, ClassExpress adds a prototype for the new handler to the class's header file and inserts a stub for the member function into the implementation file. All you need to do is to add code to the body of the handler.

Note

This introduction to MFC message handling has necessarily been brief and has glossed over several topics you will want to learn more about. For a definitive discussion of how MFC works, see the expository chapters of the *Microsoft Foundation Class Library Reference*.

Because you used AppExpress in Lesson 2 to generate an application framework for the TML Reader, message maps have already been created for your derived classes. During the course of this lesson, ClassExpress updates them. ClassExpress can be run either separately—from a Program Manager icon—or from within the IDDE. Because you will be using the Source window to edit the code that ClassExpress generates, you will launch ClassExpress from within the IDDE.

Launching ClassExpress

To launch ClassExpress from the IDDE:

1. If you are not already in the IDDE, launch it.
2. Open the project `tmlread.prj` in directory `samples\tutorial\lesson4\start`.
3. Choose **ClassExpress** from the **Tools** menu. This launches ClassExpress and automatically loads the project `tmlread.prj`.

The ClassExpress window features a multipage interface. Its left column presents a list of the six different pages that can be displayed in the area to the right. You will work on the Message Maps page, which is selected automatically when you launch ClassExpress. The ClassExpress window is displayed, as in Figure 13-1.

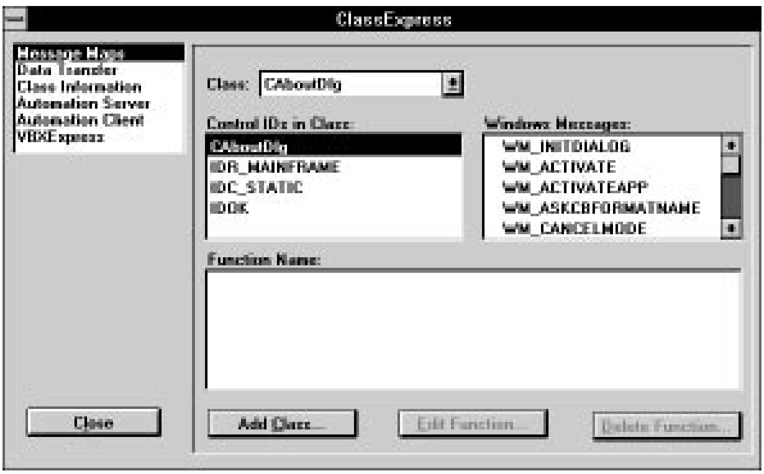

Figure 13-1 ClassExpress, displaying the Message Maps page

The lists on the Message Maps page

The drop-down combobox at the top of the page labeled Class contains a list of all derived classes in the TML Reader application. The class selected here directly determines the contents of two lists beneath it—those labeled Control IDs in Class and Function Name.

The list labeled Control IDs in Class contains an entry for the class itself, as well as the names of any commands and controls that the selected class could potentially handle.

The list labeled Function Name contains a list of message handlers already generated for the selected class by AppExpress. These methods are referenced in the class's message map.

The contents of the list labeled Windows Messages depends on what you select in the Control IDs in Class list. If you select the class name in the Control IDs list, then the Windows Messages list contains a list of Windows messages. However, if you select a control ID, the Windows messages list contains notification messages appropriate to that type of control. If you select a command ID (usually corresponding to a menu item), the rightmost list contains names of potential message map entries for commands.

Because the TML Reader handles messages that signal user input, the handlers must be added to the class corresponding to the window that receives those messages. Thus, you will add the handlers to the View class, `CTMLReadView`.

Adding Message Handlers

The requirements of the TML Reader dictate the messages you will add handlers for.

Requirement	Message
Detect when the window is resized, so it can recompute word-wrapping	`WM_SIZE`
Permit scrolling through a document using the vertical scrollbar	`WM_VSCROLL`

Requirement	Message
Permit scrolling through a document using the keyboard	WM_KEYDOWN
Change the cursor when it is positioned over a hyperlink	WM_SETCURSOR
Detect clicks on hyperlink jumps so it can change its display	WM_LBUTTONDOWN
Repaint the window background	WM_ERASEBKGND

Adding a handler for WM_SIZE

To add a handler for WM_SIZE to your View class:

1. From the drop-down list labeled Class, select the name of the View class, CTMLReadView.
2. Use the scrollbar to move through the Windows messages list until the message WM_SIZE is visible.
3. Double-click on WM_SIZE.

Notice that a new entry appears in the Function Name list:

```
afx_msg void OnSize(UINT nType, int cx, int cy);
```

This is the prototype of a handler for the WM_SIZE message, as it would appear within a class declaration.

Also notice that a solid square appears to the left of the message name, WM_SIZE, indicating that a handler now exists for that message.

The ClassExpress window is displayed, as in Figure 13-2.

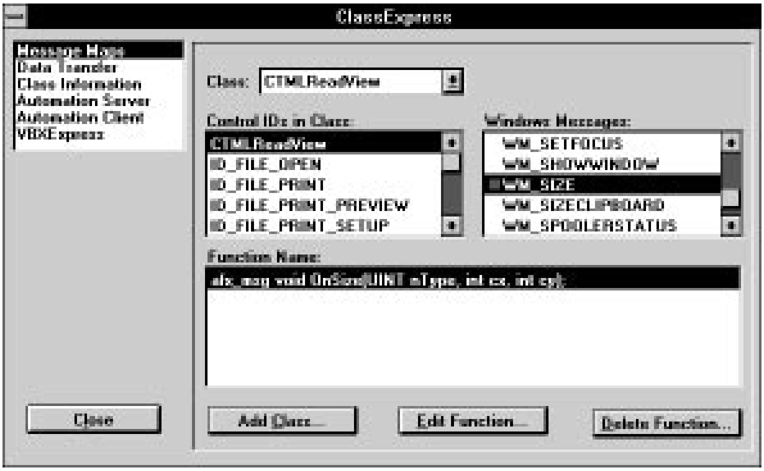

Figure 13-2 ClassExpress after adding a `WM_SIZE` handler

Adding other message handlers

Follow the procedure described in Steps 2 and 3 in the previous section to add a handler for each of the messages `WM_VSCROLL`, `WM_KEYDOWN`, `WM_SETCURSOR`, `WM_LBUTTONDOWN` and `WM_ERASEBKGND`: use the scrollbar of the Windows messages list to scroll to the message, then double-click on the message. Again, notice that for each message you double-click on, a new prototype is added to the Function name list, and a solid square appears to the left of the message name.

What you have just done

When you add a handler for a message `WM_`*messagename*, ClassExpress generates code in three places.

- The prototype for the handler is added to the declaration of the selected class, and the handler becomes a protected member function.
- An entry `ON_WM_`*messagename*`()`is added to the message map for the class in that class's implementation file.
- A stub function is created for the handler in the implementation file. The body of the function just calls the base class handler, and contains the comment:

```
// TODO: Add your message handler code
   here and/or call default
```

For example, when you add a handler for `WM_SIZE`, ClassExpress generates the following:

- The prototype for `OnSize` is added to the declaration of the class `CTMLReadView` in `tmlrdvw.h`.
- An entry `ON_WM_SIZE()`is added to the message map for the class `CTMLReadView` in its implementation file, `tmlrdvw.cpp`.
- A stub function for `CTMLReadView::OnSize` is added to the file `tmlrdvw.cpp`. The body of the function just calls the base class handler `CView::OnSize`.

Now, whenever a `CTMLReadView` window receives a `WM_SIZE` message, the member function `OnSize` is called to handle it.

Saving Your Work

To save your work and return to the IDDE, click Close at the bottom of the ClassExpress window.

Clicking Close updates your project files and returns you to the IDDE. To observe the handlers in action and confirm that they are indeed called when you expect them to be, you can add code that notifies you when they are called.

Adding Code to Handlers

To add code to the `OnLButtonDown` and `OnSize` handlers, follow these steps:

1. Open the file `tmlrdvw.cpp` in a Source window.
2. Find the function `CTMLReadView::OnLButtonDown`.
3. Add the following line to the top of the function:

```
AfxMessageBox( "Left button clicked!");
```

4. Now find the function `CTMLReadView::OnSize`.

5. Add the following line to the top of the function:

```
::MessageBeep( -1);
```

6. Type Ctrl+S to save your changes.

After the project is rebuilt, you'll want to confirm that the appropriate statement executes whenever you click in the client area or resize the window.

Building and Running the Project

To see the effect of what you've done, perform these steps:

1. Choose **Build** from the **Project** menu to update the executable.

2. Choose **Execute Program** from the **Project** menu to run the program.

3. With the mouse cursor in the client area of the program's window, press the left mouse button. A box containing the message "Left button clicked!" is displayed (Figure 13-3). This message is displayed each time you press the left mouse button in the client area. It is not displayed when you release the left mouse button, move the mouse, or click with the right mouse button. To remove the message box, click OK.

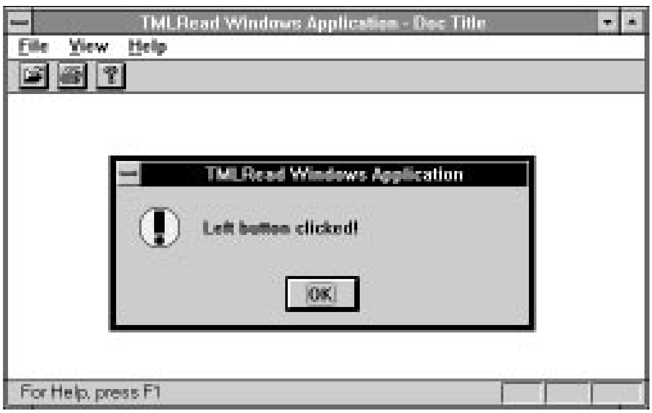

Figure 13-3 TML Reader running, with message box displayed

4. Now resize the window by dragging a corner of the window's border to a different position. You hear a beep when you release the mouse button.

Note

You will already have heard beeps during the initial sequence of messages received by the View window. This is normal: the window receives multiple `WM_SIZE` messages early in its life-cycle.

---

5. After you have confirmed to your satisfaction that the handlers are called when, and only when, they ought to be called, quit the program by choosing **Exit** from TMLRead's **File** menu.

Summary

In this lesson, you have learned:

- How Windows messages are handled in an MFC application
- How to launch ClassExpress from the IDDE and use it to add message handlers
- How to verify that the message handlers are called in response to the messages they handle

In the next lesson, you implement a **Preferences** dialog box. You learn how MFC and ClassExpress make it easy to create a dialog box, validate its data, and exchange data between the dialog box and your application. In TML Reader's present state, the only preference one could have would be for the application to actually do something. Toward that end, we have added code to the result of this lesson so that your starting point in the next lesson is a functioning TML Reader.

Lesson 5: Add a Dialog Box with ClassExpress

14

In Lesson 4, you learned how to use ClassExpress to add handlers for Windows messages. The application from that lesson is still skeletal. Its interface is complete, and it receives the Windows messages it needs to act upon, but every user action ultimately results in a call to a stub function. After completing Lesson 4, the next step is writing code that supplies functionality.

This lesson shows you how to use some additional, powerful features of Symantec C++. To accomplish this, a body of code that supplies some substantial functionality has been added to the application you built in Lesson 4. In between the end of Lesson 4 and the start of this lesson, only the Source window has been used to create or change C++ files in the project. That is, none of the new code has been automatically generated; it has all been manually entered.

Start this lesson with the project `samples\tutorial\lesson5\start\tmlread.prj`, located in the directory where you installed Symantec C++. The first task you will perform is to build that project. Once you have done so, you will have a fully functional TML Reader.

In the remainder of the lesson, you will implement a **Preferences** dialog box. In addition to supplying C++ code for the start of this lesson, we have also already used ResourceStudio to create a dialog resource that defines the **Preferences** dialog box. However, this dialog box is not yet connected to the application. It lies dormant within `tmlread.rc`. You will perform all the tasks necessary to animate it and make it fully functional.

Specifically, you'll use ResourceStudio to:

- Add a new menu item for invoking the Preferences dialog box
- Launch ClassExpress

Within ClassExpress, you'll:

- Create a new class that represents the **Preferences** dialog box
- Add a handler for the new menu item
- Add a handler to this new class for one of the buttons in the dialog box
- Add data members to the new class that transfer information between the application's data and the controls in the dialog box. You will also specify validation criteria for the values that the user enters in the dialog box, so that MFC can perform automatic data validation.

Finally, within the IDDE, you will:

- Add code for the two new handlers
- Rebuild the project and examine the results of your efforts

After completing this lesson, you will have:

- Added a new menu command
- Used MFC and ClassExpress to implement a dialog box, validate its data, and exchange data between the dialog box and your application

The next section walks you through building and running the TML Reader. It also discusses how the Reader displays files and how to create a **Preferences** dialog box which lets the user configure how the files are displayed.

Building and Exploring the TML Reader

This section acquaints you with the major features of the TML Reader. After building the Reader, you will use it to read sample files that contain TML formatting strings.

Before building the Reader, you must first copy the file `VIEWHDRS.H` from the `TUTORIAL\TMLREAD` directory to the `TUTORIAL\LESSON5\BACKUP` directories. Various TMLRead modules include this header file, and cannot be compiled without it.

Building the Reader

Follow the first four steps to build the TML Reader.

1. If you are not already in the IDDE, launch it.

2. Open the project `tmlread.prj` in the directory `samples\tutorial\lesson5\start` located beneath the directory where you installed Symantec C++.

3. Choose **Build** from the **Project** menu.

4. Choose **Execute Program** from the **Project** menu to run the program. The IDDE minimizes, and TMLRead is launched. No file is yet displayed.

Exploring the capabilities of the Reader

Now use the Reader to browse two sample files that illustrate the kinds of formatting that the Reader can display.

1. Choose **Open** from TMLRead's **File** menu.

2. In the **File Open** dialog box, select the file `sample.tml` in the `samples\tutorial\lesson5\start` directory, then click OK. TMLRead reads, parses, and displays the file.

3. Scroll within the document using the arrow keys and the Page Up and Page Down keys. Code within `CTMLReadView::OnKeyDown` causes the scrolling to occur.

4. Now scroll by using the vertical scroll bar. In this case, code within `CTMLReadView::OnVScroll` is responsible for the scrolling.

5. Drag either the left or right border of the window to resize it. Notice that the ends of lines are automatically adjusted so that paragraphs wrap properly, filling almost all of the window's width. The handler `CTMLReadView::OnSize` causes rewrapping to occur.

6. If you are not now displaying the beginning of the file, press the Home key. The heading "Contents" announces the table of contents of this document, presented as a hierarchical bulleted list of the major sections of `sample.tml`. Each line in the table of contents is underlined and displayed in green. These properties of the text indicate that it is a hyperlink. Place the cursor over any hyperlink; the cursor changes to a hand with an extended index finger, a visual cue that you can meaningfully click on the text. Move the cursor so that it is not over a hyperlink; the cursor reverts to the default cursor. This behavior is provided by `CTMLReadView::OnSetCursor` after a necessary initialization by `CTMLReadView::PreCreateWindow`.

7. Click on the last hyperlink in the Contents section, named "Hyperlinks," with the left mouse button. This displays the last section of the file. The method `CTMLReadView::OnLButtonDown` handles the mouse click by determining whether or not it occurred on a hyperlink; if a hyperlink is clicked, the jump occurs.

8. At the end of the last paragraph, there is the following hyperlink phrase: "here is a link to another sample document." Click on this phrase. The file `complex.tml` is read and displayed. Again, `CTMLReadView::OnLButtonDown` causes the jump to occur.

9. Choose **Previous File** from TMLRead's **File** menu to return to `sample.tml`.

10. Examine each section of `sample.tml` by clicking each hyperlink in the document's contents at the top of the file. This will give you a good sense of the features of TML, the subset of HTML (HyperText Markup Language) recognized by the Reader.

11. Choose **Print Preview** from TMLRead's **File** menu to see a WYSIWYG display of how `sample.tml` would look if it were printed on the current default printer. The presence and functionality of the buttons at the top of the Print Preview window has been supplied almost entirely by MFC. When you finish looking at the Print Preview display, click Close to return to the main view of the Reader.

12. Choose **Exit** from TMLRead's **File** menu. This closes the Reader and returns you to the IDDE.

Turning aspects of the Reader's display into preferences

A few numerical quantities that help determine how TMLRead displays documents are at present hard-coded. These numbers are stored as `int` data members of `CTMLReadView` and are described in the following table:

Table 14-1 Quantities which the Preferences dialog box manipulates

CTMLReadView data member	TMLRead display characteristic
`nParVSpace`	Vertical space between paragraphs
`nMargin`	Amount of horizontal space between the document and the edges of the window
`nIndent`	Amount by which items in a list are offset from the left margin

The `CTMLReadView` constructor calls `CTMLReadView::SetDefaultPrefs` to initialize these data members with default values provided by the following `enum` constants (defined in the class declaration):

```
eDftParVSpace = 12
eDftMargin    = 10
eDftIndent    = 40
```

The **Preferences** dialog box lets the user alter these values, as shown in Figure 14-1.

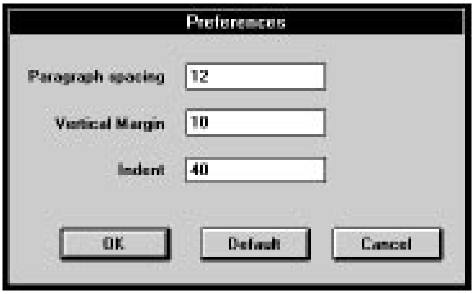

Figure 14-1 The Preferences dialog box

Of course, the user should be allowed to configure many other aspects of the Reader's display, such as colors and fonts. However, implementing this simple **Preferences** dialog box gives you the fundamental skills needed to realize more ambitious designs using the Symantec C++ tools.

Before you connect the Reader's data with the controls of the **Preferences** dialog box, you need to first outfit the user interface with a way to call the dialog box. In the next section, therefore, you'll use ResourceStudio to add a menu item to TMLRead.

Using ResourceStudio to Add a Menu Item

To add a **Preferences** item to the **View** menu of TMLRead, launch ResourceStudio from within the IDDE so that it loads `tmlread.rc`. To do this, follow these steps:

1. Open the Project window if it is not already open.

2. Double-click on `tmlread.rc` in the Project window. When asked if you want to use ResourceStudio to edit this file, click Yes.

   The Browser window of ResourceStudio opens, as shown in Figure 14-2.

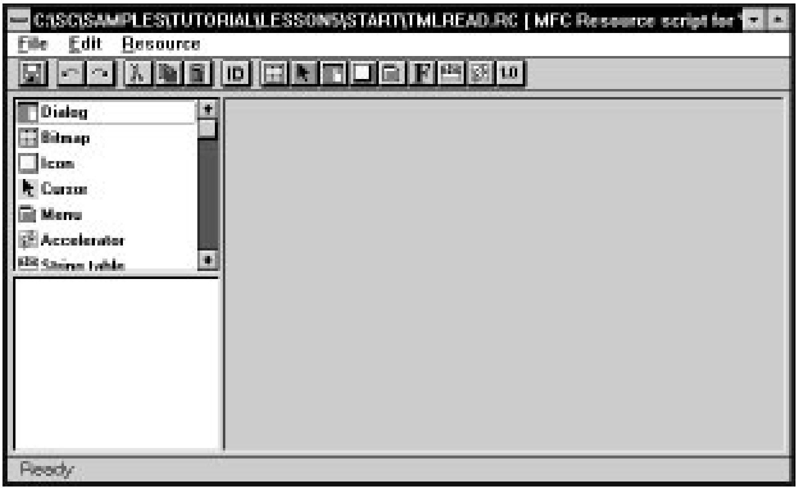

Figure 14-2 The Browser window of ResourceStudio

3. Select the item named Menu in the upper-left pane. The lower-left pane now displays the ID of the TMLRead menu, `IDR_MAINFRAME`.

4. Double-click on `IDR_MAINFRAME` in the lower-left pane. The pane on the right now contains a representation of the TMLRead menu, shown in Figure 14-3.

The Property Sheet and the Test menu window also open; they are described in Chapter 12, "Lesson 3: Customize the Interface."

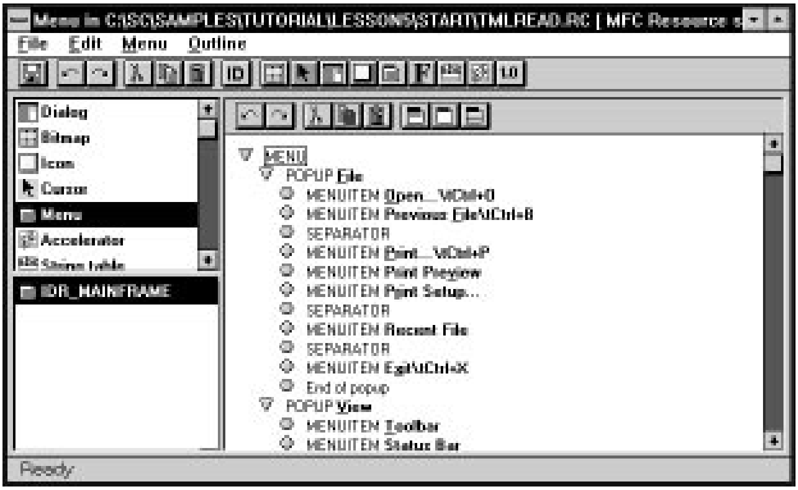

Figure 14-3 ResourceStudio displaying the TMLRead menu

5. In the right pane, click on the item `MENUITEM Status Bar` to select it. This item is the last one belonging to the `POPUP View` item. When a new menu item is added, it will appear beneath this one.

6. Choose **New Item** from the **Menu** menu in the Browser window. This inserts a new item whose text is `Item`, and creates a new ID for it. The Property Sheet, shown in Figure 14-4, shows these settings.

Figure 14-4 The Property Sheet after adding a new item

7. Click on the Property Sheet to make it active.

8. Click on the General tab near the top of the window to ensure that the General page is displayed.

9. In the ID textbox, type `ID_VIEW_PREFS` to change the ID name.

10. In the text textbox, type `&Preferences...`.

11. Click on the Connect tab.

12. On this page, type the string `Customize the appearance of the view`. This prompt will appear in the status bar whenever the **Preferences** item is selected.

13. Save your work by choosing **Save** from the Browser window's **File** menu.

14. Close the Browser window, close the ResourceStudio Shell window (and with it, the Property Sheet), and return to the IDDE.

You have now created the desired menu item. However, TMLRead does not yet contain a command handler for `ID_VIEW_PREFS`. In the next section, you will add one using ClassExpress, in which the bulk of the remaining work will take place.

Two command handlers must be added—one for `ID_VIEW_PREFS`, the other for the Default button of the **Preferences** dialog box. Thus, a class must be created that represents the **Preferences** dialog box in the same way that CAboutDlg represents the **About** dialog box. The handler for the Default button will be a method for this new class. In the next section, you use ClassExpress to add the CPrefDialog class.

Using ClassExpress to Create a New Dialog Class

Follow these steps to create a class that corresponds to the **Preferences** dialog box:

1. Launch ClassExpress by choosing **ClassExpress** from the IDDE's Tools menu.

2. Click the Add Class button. The **Add Class** dialog box opens, as shown in Figure 14-5. ClassExpress detects that a new dialog box has been added to the project, and assumes that you want to create a corresponding class.

   It initializes the selections in the Class Type and Dialog ID drop-down lists to reflect this assumption: Class Type is set to Dialog, and Dialog ID to `IDD_PREFDIALOG`.

   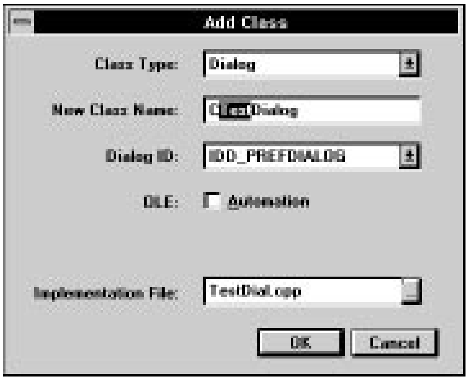

   Figure 14-5 The Add Class dialog box

   The focus is in the New Class Name editbox. ClassExpress has suggested the name `CTestDialog` and has selected the substring `Test` so that you can change it just by typing. It has also suggested `TestDial.cpp` as the name of the implementation file.

3. Type `Pref` to change the name of the dialog class to `CPrefDialog`. Notice that ClassExpress accordingly changes the suggested name of the implementation file to `PrefDial.cpp`.

4. Click OK in the **Add Class** dialog box to accept the settings. ClassExpress confirms the operation with a message box.

5. Click OK to close this message box. This returns you to the ClassExpress window, which displays the Message Maps page, as shown in Figure 14-6.

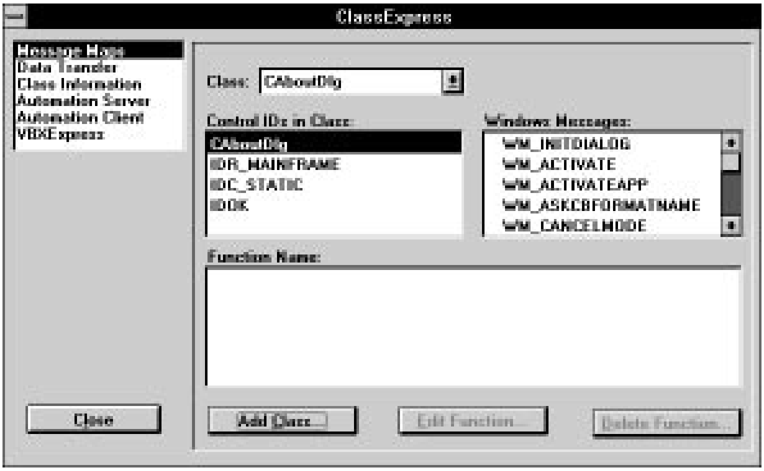

Figure 14-6 ClassExpress displaying the Message Maps page

Using ClassExpress to Add Methods

In this section, you'll add two methods that respond to user actions. One opens the **Preferences** dialog box when the **Preferences** menu item is chosen, and updates the Reader's display if the OK button is clicked in the dialog box. The other method handles the clicking of the Default button in the **Preferences** dialog box. You will see later why it is unnecessary to add handlers for the OK and Cancel buttons.

First you'll add a method that handles the command generated when **Preferences** is chosen from the **View** menu. This method also is responsible for updating the application to reflect any changed preferences. Then, you'll add a method that responds to a click on the Default button of the **Preferences** dialog box.

The procedure in each case is similar to the one that you used in Lesson 4. The results of your actions also will be similar: ClassExpress adds these methods to class declarations, add entries to Message Maps, and create stub functions for the methods. When you return to the IDDE, you will only need to write the code for these two methods.

Creating a handler for the Preferences command

You will use ClassExpress to bind the ID_VIEW_PREFS command ID to a new method that creates and initializes an object of class CPrefDialog, and then uses this object to display the dialog box. When the user closes the dialog box, this method determines whether the user clicked OK or Cancel. If OK was clicked, application data must be updated, and the view must be redrawn.

This method needs access to data members of CTMLReadView in order to initialize the dialog box and to update those data members if the user clicks OK. Because of these requirements, the method is added to the class CTMLReadView.

1. Select the class CTMLReadView in the Class combobox on the Message Maps page.

2. In the Control IDs in the Class listbox, select ID_VIEW_PREFS, the ID of the command issued when the **Preferences** item in the **View** menu is chosen. The listbox on the right, Windows Messages, now contains the two items, COMMAND and UPDATE_COMMAND_UI.

3. Double-click on COMMAND in the Windows Messages listbox. The **Method Name** dialog box opens, asking you for a name for the method that is called when **Preferences** is chosen. ClassExpress suggests the name OnViewPrefs for this method.

4. Click OK in the **Method Name** dialog to accept the name OnViewPrefs.

ClassExpress adds this method to the declaration of class CTMLReadView in the header file for the class. It also updates the class implementation file by adding an entry to the class's Message Map, and by adding a stub function CTMLReadView::OnViewPrefs.

Creating a handler for the Default button of the Preferences dialog box

You will now add a method to `CPrefDialog` for handling clicks on the Default button in the **Preferences** dialog box. This method must update the values displayed in the controls of the dialog box with the same default values used to initialize the `CTMLReadView` data members `nParVSpace`, `nMargin`, and `nIndent`. Because the default values are `public enum` constants, this method requires no access privileges to `CTMLReadView`.

Follow these steps to add this method:

1. Select the class `CPrefDialog` in the Class combobox on the Message Maps page.

2. In the Control IDs in Class listbox, select `ID_PREFS_DEFAULT`, the ID of the Default button. The listbox on the right, Windows Messages, now contains the two items `BN_CLICKED` and `BN_DOUBLECLICKED`.

Note

> These so-called messages are actually notifications that the button sends to its parent, the **Preferences** dialog box, via `WM_COMMAND` messages. `BN_`*xxx* stands for Button Notification. The `BN_`*xxx* identifiers are defined in `windows.h`.

---

3. In the Windows Messages listbox, double-click on `BN_CLICKED`, the notification sent when the Default button is clicked. The **Method Name** dialog box opens, suggesting `OnClickedPrefsDefault` for the method name.

4. Change this name to `OnDefault`.

5. Click OK in the **Method Name** dialog box to return to the ClassExpress main window.

As it did when you added `CTMLReadView::OnViewPrefs`, ClassExpress adds a prototype for `OnDefault` to the declaration of the class `CPrefDialog` in the class's header file. It also updates the class implementation file by adding an entry to the class's Message Map. It also adds a stub function `CPrefDialog::OnDefault`.

Once you are back in the IDDE, you must write code to implement these two handlers. To have the handlers to perform their intended tasks, you must first make it possible to transfer data into and out of the controls of the **Preferences** dialog box. MFC and ClassExpress make it surprisingly easy not only to transfer data to and from a dialog box, but also to validate data that the user has entered into a dialog's controls. Adding these capabilities is addressed in the next section, the last one in which you use ClassExpress.

Adding Dialog Data Exchange and Validation

Windows programs written in C traditionally exchange data with dialog boxes by fetching data from each control in a manner particular to the control type (edit control, radio button, check box, and so on) and particular to the intended type of the data (string, integer, long integer, and so on). Then, the extracted data is usually stored in `static` storage. Also, there are rarely any general-purpose facilities from Windows API for validating data that the user enters in a dialog box—placing another burden on the programmer.

MFC and ClassExpress add order, simplicity and elegance to this situation. To use the MFC model of dialog box data exchange and validation, you must first add data members to the dialog class. (This replaces the ad hoc collection of `static` data favored by the traditional approach.) You use ClassExpress to associate data members with controls in the dialog box. MFC then automates the transfer of data between the dialog's controls and the dialog class data members. When adding a data member with ClassExpress, you also specify its type and the validation criteria in the associated control. MFC uses this information to perform its automatic data validation.

In the next step, you'll add data members to the `CPrefDialog` class using the Data Transfer page of ClassExpress. After you complete this task, there is a review of the changes that ClassExpress has made to the `CPrefDialog` source files, and an explanation of how MFC accomplishes data exchange and validation.

Throughout this section, you'll work with the `CPrefDialog` class on the Data Transfer page of ClassExpress. Perform the following steps to set up ClassExpress for this part of the lesson:

1. In the upper-left listbox, select the second item, Data Transfer. ClassExpress opens the Data Transfer page in the larger pane on the right, as shown in Figure 14-7.

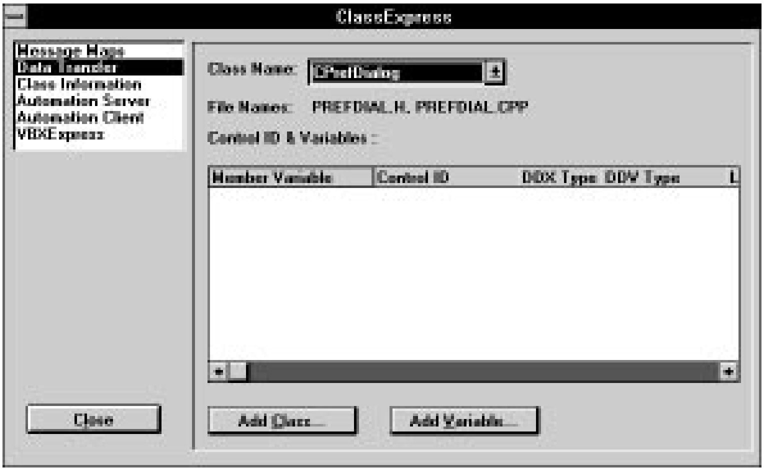

Figure 14-7 ClassExpress displaying the Data Transfer page

2. Be sure that `CPrefDialog` is selected in the Class Name combobox at the top of the Data Transfer page.

Adding data members to CPrefDialog

To add data members, to CPref Dialog, carry out the following steps:

1. Click on the Add Variable button. The **Add Member Variable** dialog box opens, as shown in Figure 14-8.

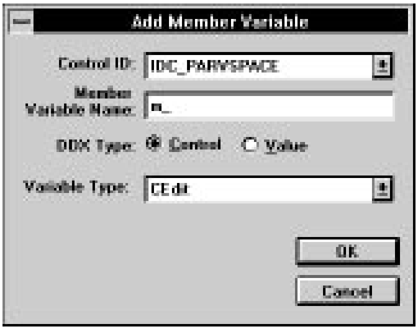

Figure 14-8 The Add Member Variable dialog box

2. Select IDC_PARVSPACE from the Control ID combobox.

3. Change the Member Variable Name to nParVSpace. (For simplicity, you can name the member variables of CPrefDialog the same as the members of CTMLReadView to which they correspond.)

4. For the DDX Type option, select Value.

   In fact, the Control Name combobox does not contain symbolic indentifiers for the three edit controls in the Preferences dialog box. Rather, you see only the integer resource ids of these controls. Select 3001, the resource id of the Paragraph Spacing edit control.

5. In the Variable Type combobox, select `int`. Two new textboxes, Minimum Value and Maximum Value, appear at the bottom of the dialog box, as shown in Figure 14-9.

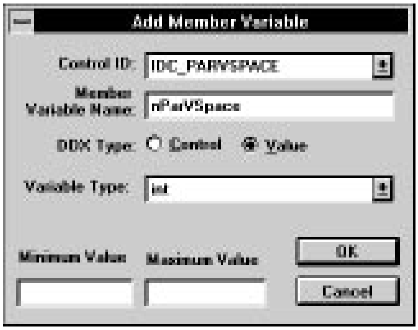

Figure 14-9 Add Member Variable with Variable Type set to int

6. Type `0` in the Minimum Value field.

7. Type `100` in the Maximum Value field.

8. Click OK in the **Add Member Variable** dialog box. The Data Transfer page now reflects this additional data member and its validation criteria in the Control ID and Variables list, as shown in Figure 14-10.

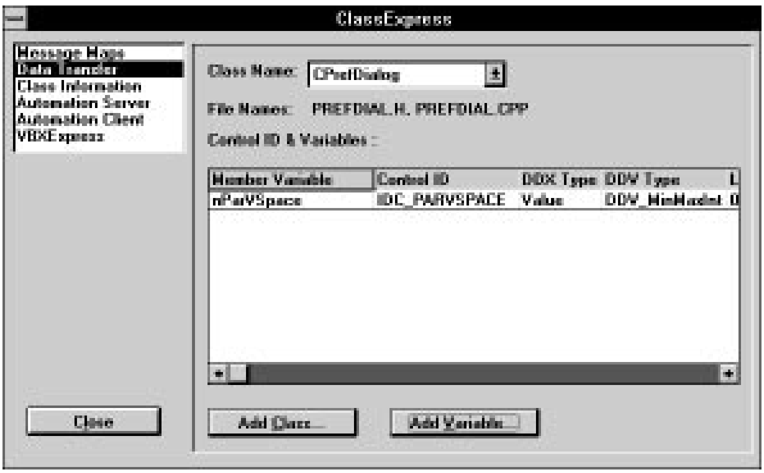

Figure 14-10 The Data Transfer page after adding nParVSpace

9. Follow the procedure described in steps 3 through 10 of the previous task to add an `int` variable named `nMargin` associated with the control IDC_MARGIN. Set its minimum and maximum values to `0` and `50`, respectively.

   In fact, the Control Name combobox does not contain symbolic identifiers for the three edit controls in the Preferences dialog box. Rather, you see only the integer resource ids of these controls. Therefore, add a data member corresponding to the control 3004, the resource id of the Margin edit control.

10. Then, follow steps 3 through 10 to add a final `int` variable, `nIndent`, associated with the control IDC_INDENT. Sets its minimum and maximum values to `0` and `120`, respectively.

    In fact, the Control Name combobox does not contain symbolic identifiers for the three edit controls in the Preferences dialog box. Rather, you see only the integer resource ids of these controls. Therefore, add a data member corresponding to the control 3005, the resource id of the Indent edit control.

To return to the IDDE, click the Close button. The IDDE's Project window now lists the source file `prefdial.cpp`.

Seeing the changes in the CPrefDialog source files

As a result of adding data members to `CPrefDialog`, ClassExpress makes various changes to the header and implementation files of the class `CPrefDialog` including:

Changes to the class definition

ClassExpress added the following lines to the declaration of `CPrefDialog` in `prefdial.h`:

```
// Dialog Data
   //{{AFX_DATA(CPrefDialog)
   enum { IDD = IDD_PREFDIALOG };
   int nParVSpace;
   int nMargin;
   int nIndent;
   //}}AFX_DATA

// Implementation
protected:
   virtual void DoDataExchange(CDataExchange*
                 pDX);// DDX/DDV support
```

The comments help ClassExpress find where subsequently added data members should be declared. (The variables that ClassExpress adds are public.) The `enum` constant `IDD` is used by the `CPrefDialog` constructor as an argument to the constructor of its base class, `CDialog`.

Changes to the constructor

ClassExpress changes the `CPrefDialog` constructor so that it appears as follows:

```
CPrefDialog::CPrefDialog(CWnd* pParent /*=NULL*/)
   : CDialog(CPrefDialog::IDD, pParent)
{

   //{{AFX_DATA_INIT(CPrefDialog)
   nParVSpace = 0;
   nMargin = 0;
   nIndent = 0;
   //}}AFX_DATA_INIT
}
```

The `AFX_DATA_INIT` comments are used by ClassExpress to delimit the location within the constructor where data members participating in data exchange and validation should be initialized. All the data members that you added are set to `0`. Therefore, before it displays the **Preferences** dialog box, the handler `CTMLReadView::OnViewPrefs` first sets these CPrefDialog data members to the values of the `CTMLReadView` data members to which they correspond.

Changes to the DoDataExchange function

The protected member function `DoDataExchange` is the workhorse that transfers data between the added data members and the controls of the **Preferences** dialog box. It also validates the values entered into the controls when the user clicks OK. (The acronym DDX stands for Dialog Data eXchange; DDV stands for Dialog Data Validation.) ClassExpress also writes the `DoDataExchange` function, using the information you gave it when you added each variable. The implementation of the function, in `PrefDial.cpp`, looks like this:

```
void CPrefDialog::DoDataExchange(CDataExchange*
                                   pDX)
{
   CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CPrefDialog)
   DDX_Text(pDX, IDC_PARVSPACE, nParVSpace);
   DDV_MinMaxInt(pDX, nParVSpace, 0, 100);
   DDX_Text(pDX, IDC_MARGIN, nMargin);
   DDV_MinMaxInt(pDX, nMargin, 0, 50);
   DDX_Text(pDX, IDC_INDENT, nIndent);
   DDV_MinMaxInt(pDX, nIndent, 0, 120);
   //}}AFX_DATA_MAP
}
```

Again, the `AFX_DATA_MAP` comments serve only as delimiters used by ClassExpress when adding data members.

The overloaded `DDX_`*xxx* and `DDV_`*xxx* functions are declared in the MFC include file `afxdd_.h`. These functions all take a pointer, `pDX`, to a `CDataExchange` object. This object contains two public data members that enable the functions.

The `CDataExchange` data member `m_bSaveAndValidate` informs the functions of the direction of the transfer. If `m_bSaveAndValidate` is `TRUE`, then the transfer is from the controls to the data members; if this data member is `FALSE`, the transfer is from the data members to the controls. The `DDX_`*xxx* functions perform the appropriate transfer in each case. If `m_bSaveAndValidate` is `TRUE`, each `DDV_`*xxx* validation function checks whether the data meets the validation criteria it implements. If the data fails to meet the criteria, the validation function opens a message box informing the user, and sets the focus to the control containing the invalid value. If `m_bSaveAndValidate` is `FALSE`, the validation functions do nothing.

The `CDataExchange` data member `m_pDlgWnd` is a pointer to the `CWnd` whose data is being transferred and validated. The presence of this data member saves `DoDataExchange` from having to pass the `this` pointer to every `DDX_`*xxx* and `DDV_`*xxx* function.

Note

> Because DoDataExchange is a CWnd member function and m_pDlgWnd is a CWnd*, DDX and DDV can be used with any window, and not just with those derived from CDialog.

DoDataExchange is called by the UpdateData function whenever data exchange must take place. You never call it directly. In particular, DoDataExchange is called when a window or dialog box is first opened to initialize its controls. It is also called by the CDialog member function OnOK when a user clicks on a button in the dialog box with an ID of IDOK. Thus in the **Preferences** dialog box, the inherited handler for the OK button already performs data transfer and validation, so you do not need to supply an override. Similarly, the CDialog member function OnCancel simply terminates a dialog box—behavior completely suitable for the **Preferences** dialog box.

With these concepts in mind, you can clearly see the purpose and effect of the handlers that you supply in the next task.

Writing Code for the New Handlers

Now, the only task left is to provide implementations of the handlers CTMLReadView::OnViewPrefs and CPrefDialog::OnDefault. Both handlers are straightforward.

Implementing CTMLReadView::OnViewPrefs

Carry out these steps to create the handler:

1. In the Project window, double-click on tmlrdvw.cpp. This opens a Source window in which you can edit the file tmlrdvw.cpp.

2. Find the group of #include statements toward the top of the file. Add this line after the last #include statement:

   ```
   #include "prefdial.h"
   ```

3. Find the function CTMLReadView::OnViewPrefs. (It is at the bottom of the file.)

4. Edit the function by entering the following code. (You may omit the lengthy comments.)

```
void CTMLReadView::OnViewPrefs()
{
   CPrefDialog dlgPref;

   // Initialize dlgPref data members with
   // current values from CTMLReadView
   //
   dlgPref.nParVSpace = nParVSpace;
   dlgPref.nMargin    = nMargin;
   dlgPref.nIndent    = nIndent;

   // Display the dialog modally.
   // If user clicks OK, DoDataExchange
   // will be called to validate data in
   // the controls. If the controls hold valid
   // values, their contents will be
   // transferred to the CPrefDialog data
   // members.  In that case, we must transfer
   // the values to the corresponding
   // CTMLReadView data members.
   //
   if ( dlgPref.DoModal() == IDOK )
   {
      // Transfer the data to our view class
      //
      nParVSpace = dlgPref.nParVSpace;
      nMargin    = dlgPref.nMargin;
      nIndent    = dlgPref.nIndent;

      // Make sure that the view is redrawn
      // to reflect the new preferences
      //
      bWordsWrapped = FALSE;
      OnUpdate( NULL, 0L, NULL );
   }
}
```

5. Save your work by choosing **Save** from the **File** menu of the Source window.

6. Close the Source window by clicking on the close box in the upper-left corner of the window (on the caption bar).

Implementing CPrefDialog::OnDefault

This handler is even simpler. To create it, perform these steps:

1. In the Project window, double click on `prefdial.cpp`. This opens a Source window in which you can edit the file `prefdial.cpp`.

2. Find the group of `#include` statements, toward the top of the file. Add this statement to the end of the group:

```
#include "viewhdrs.h"
```

3. Find the function `CPrefDialog::OnDefault`. (It is at the bottom of the file.)

4. Edit the function by entering the following code. (You may omit the lengthy comments.)

```
void CPrefDialog::OnDefault()
{
   // Set data members to default values
   //
   nParVSpace = CTMLReadView::eDftParVSpace;
   nMargin = CTMLReadView::eDftMargin;
   nIndent = CTMLReadView::eDftIndent;

   // Transfer these values to the controls.
   // UpdateData calls DoDataExchange to effect
   // the transfer. The argument determines the
   // direction of the transfer:
   // TRUE causes values to move from the
   // controls to the data members;
   // FALSE, used here, transfers the values of
   // the data members to the controls.
   // UpdateData uses its argument to set the
   // m_bSaveAndValidate data member of the
   // CDataExchange object whose address
   // it passes to DoDataExchange.
   //
   UpdateData( FALSE );
}
```

5. Save your work by choosing **Save** from the **File** menu of the Source window.

You have now completed all the tasks necessary for the **Preferences** dialog box to be fully functional. To conclude this lesson, you will rebuild TMLRead and test the dialog box.

Rebuild and Test TMLRead

To verify that your work has achieved the desired goal, rebuild and run the Reader, making it a point to use the new features. To do this, follow these steps:

1. Choose **Build** from the **Project** menu to incorporate the changes you have made into the executable file `tmlread.exe`.
2. Choose **Execute Program** from the **Project** menu to run the Reader.
3. Choose **Open** from the **File** menu of TMLRead. Again, load the file `sample.tml`.
4. Choose **Preferences** from the **View** menu of TMLRead.
5. Enter `25` into all three edit controls.
6. Click on the Default button to verify that the `CPrefDialog` data members are set to default values by `CPrefDialog::OnDefault`.
7. Click Cancel. You should see no change in the display of `sample.tml`.
8. Again, choose **Preferences** from the **View** menu of TMLRead.
9. Change the values to `150`, `40`, and `10`, respectively.
10. Click OK. Observe the message box informing you of invalid data.
11. Click OK to close the message box. The focus returns to the editbox containing `150`.
12. Continue experimenting with the **Preferences** dialog box, eventually clicking OK when valid data is entered in the controls. After you have confirmed that the dialog box is performing as you want, exit the program by choosing **Exit** from the Reader's **File** menu. This returns you to the IDDE.

Summary

In this lesson, you learned:

- How to use the ResourceStudio to add a new menu item
- How to use ClassExpress to add a handler that responds to the choice of that menu item
- How to use ClassExpress and MFC to implement a dialog box, validate its data, and exchange data between the dialog box and your application

This concludes the tutorial section. These lessons have been an instructive introduction to the capabilities of Symantec C++. By this point, you will have acquired the techniques and knowledge you need to apply the features of Symantec C++ to your own projects.