1

Deitel® Dive-Into™ Series: Dive Into Visual C++ .NET

Objectives

- To understand the relationship between C++ and Visual C++.
- To be able to use Visual C++ to create, compile and execute C++ console applications.
- To understand and be able to use the Microsoft's Visual Studio .NET integrated development environment.
- To be able to search Microsoft's on-line documentation effectively.
- To be able to use the debugger to locate program logic errors.

Outline

1.1 Introduction

Welcome to the Visual C++® .NET integrated development environment (part of the *Microsoft Visual Studio® .NET* suite of development tools). In this chapter you will learn how to create, compile, execute and debug C++ programs using the powerful C++ development environment from Microsoft—*Visual C++ .NET*. When you complete this chapter, you will be able to use Visual C++ to begin building applications.

This guide does not teach C++; rather, it is intended as a companion to our textbook *C++ How To Program, Fourth Edition* or any other ANSI/ISO C++ textbook. *C++ How To Program, Fourth Edition* does not teach GUI programming simply because ANSI/ISO C++ does not provide libraries to create GUIs. Compiler vendors such as Microsoft, Borland and Symantec normally provide their own libraries that support creation of applications with GUIs.

Before proceeding with this tutorial, you should be familiar with the topics in Chapter 1, "Introduction to Computers and C++ Programming," and Chapter 2, "Control Structures," of *C++ How to Program, Fourth Edition*. A few of the examples in this chapter make reference to *functions*. For these examples, you should be familiar with the material through Section 3.5 of Chapter 3, "Functions," in *C++ How to Program, Fourth Edition.* For the programs in Section 1.6 of this guide, you should be familiar with the material in Chaper 6, "Classes and Data Abstraction," in *C++ How to Program, Fourth Edition.*

We hope you enjoy learning about the Visual C++ .NET integrated development environment.

1.2 Installation of Visual Studio .NET

This section will guide the user through the installation process for the Visual Studio .NET programming environment.

1. Insert the Visual Studio .NET disk 1 in the CD-Rom drive. A dialog will pop up and display several options (Fig. 1.1). The first option, **Windows Component Update**, updates the computer as needed to complete the installation. The setup program automatically determines if this step should be performed. If **Windows Component Update** is disabled, proceed to step 8.

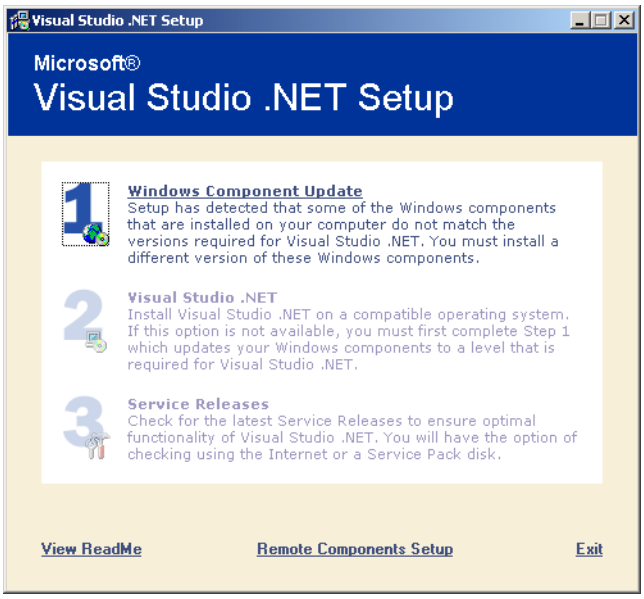

Fig. 1.1 Visual Studio .NET setup screen.

2. Clicking the **1** will display the **Web Projects Requirement** window (Fig. 1.2). This is installed if the programmer plans to host Web projects. For this example, click **Continue** to skip that part of the installation.

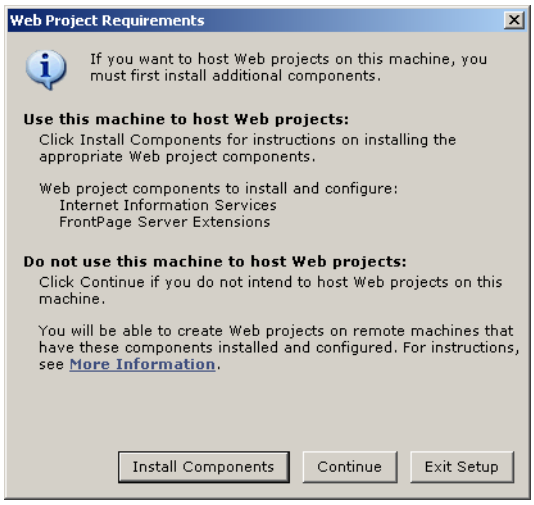

Fig. 1.2 Setup for hosting Web projects.

3. The installation prompts the user for the **Windows Component Update** disk (Fig. 1.3). Insert the disk into the CD drive and press *enter*. If multiple drives are used, be sure to appropriately specify the letter of the drive.

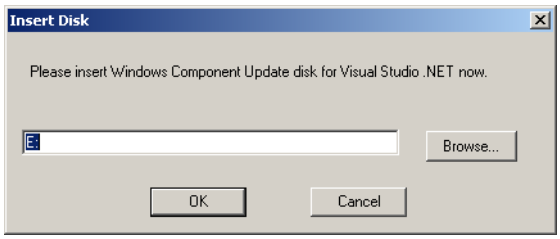

Fig. 1.3 Windows Component update CD prompt.

4. Setup will then prepare the computer for installation. Once setup prepares for installation, a **License Agreement** is displayed (Fig. 1.4). Read the agreement to understand the terms and conditions for using Windows Components. If you agree to the license, select **I accept the agreement** and click **Continue**.

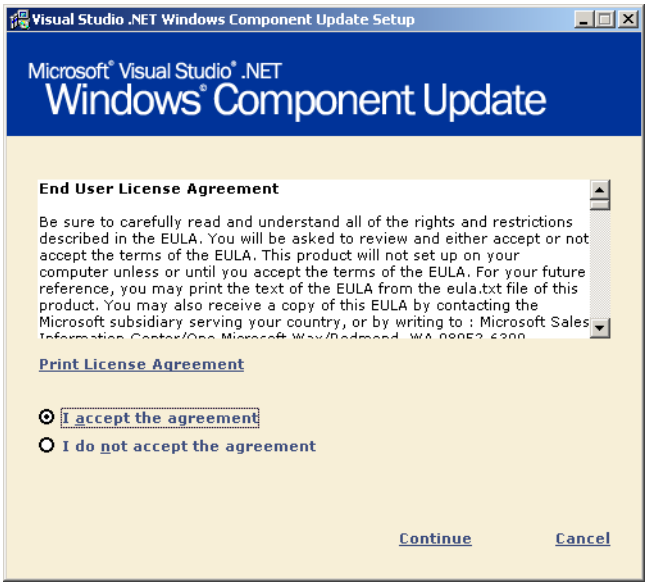

Fig. 1.4 License agreement for the Windows Component updates.

5. A new window is displayed that lists the components to be installed (Fig. 1.5). Information about each component is accessible on the Internet via the **More Information** button.

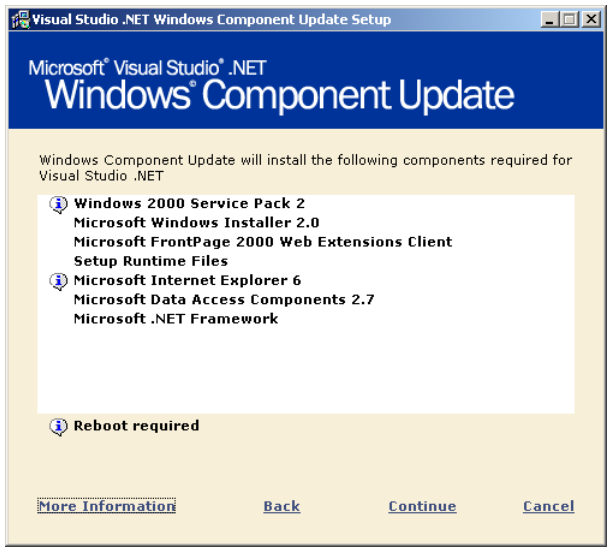

Fig. 1.5 List of components to be installed.

6. Installation asks for the user name and password information required to log into the machine. This speeds the process of installation because the machine is required to reboot. The personal information is not saved. To use this feature, check the **Automatically log on** check box and enter the user name and password in the appropriate fields (Fig. 1.6).

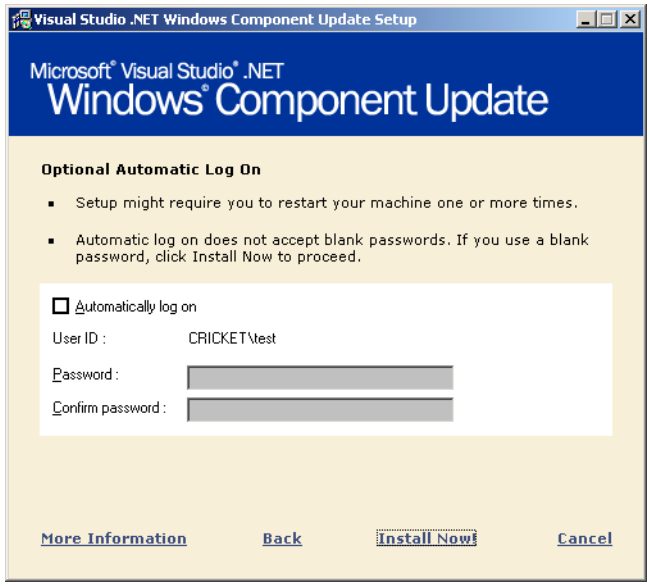

Fig. 1.6 Automatically logging back in as a specific user.

7. Click **Install Now!** to begin the installation (Fig. 1.7).

Fig. 1.7 Installing the needed Windows Components.

8. After the computer reboots, it is ready to perform part 2 (Fig. 1.8), which is the actual installation of **Visual Studio .NET**. Clicking the **2** will prompt the user to reinsert the 1st installation CD. Do so, and click **OK** to begin the installation process.

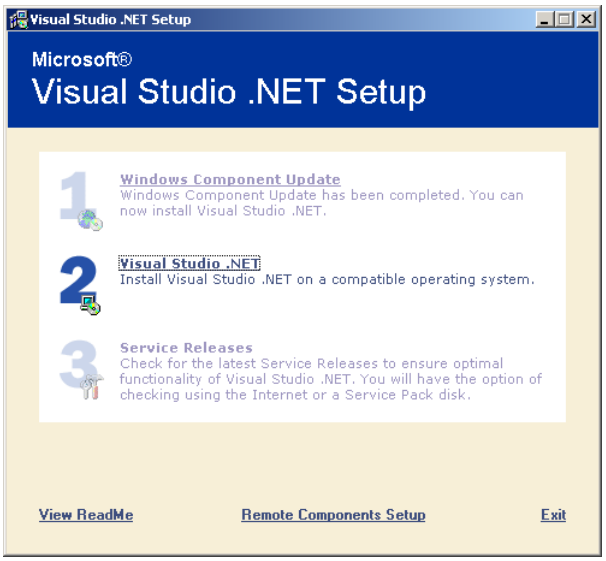

Fig. 1.8 Installing Visual Studio .NET Setup screen.

9. Setup again prepares the computer for installation. When complete, the **Visual Studio .NET license agreement** is presented and the user is asked for the **Product Key** and a name. After reading the agreement, select **I Agree** to accept the terms in the document. Enter the **Product Key** and click **Continue**. The **Product Key** is located on the first installation CD. If the key is entered incorrectly, setup instructs the user to validate the key and click **Continue** once more. Note that **Continue** is not enabled until the user agrees to the license agreement and a **Product Key** is provided (Fig. 1.9).

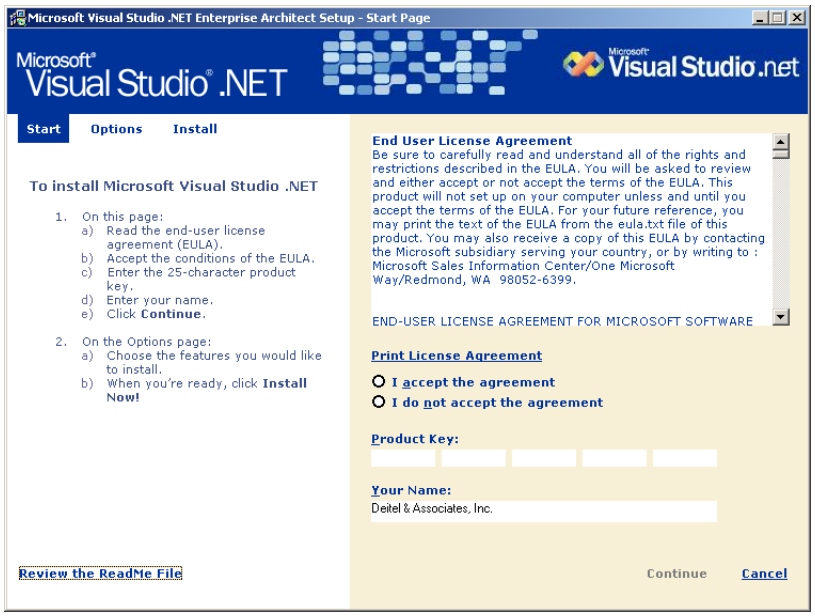

Fig. 1.9 Visual Studio .NET license agreement and CD key validation.

10. An options window is presented next. This allows the user to see how much space is available and how much Visual Studio .NET occupies. It also allows the user to specify a directory for installation—the default is **`C:\Program Files\Microsoft Visual Studio .NET\`**. There is a list of components that will be installed at the center of the screen. On the left is a detailed list of items that can be installed if the user so desires (Fig. 1.10).

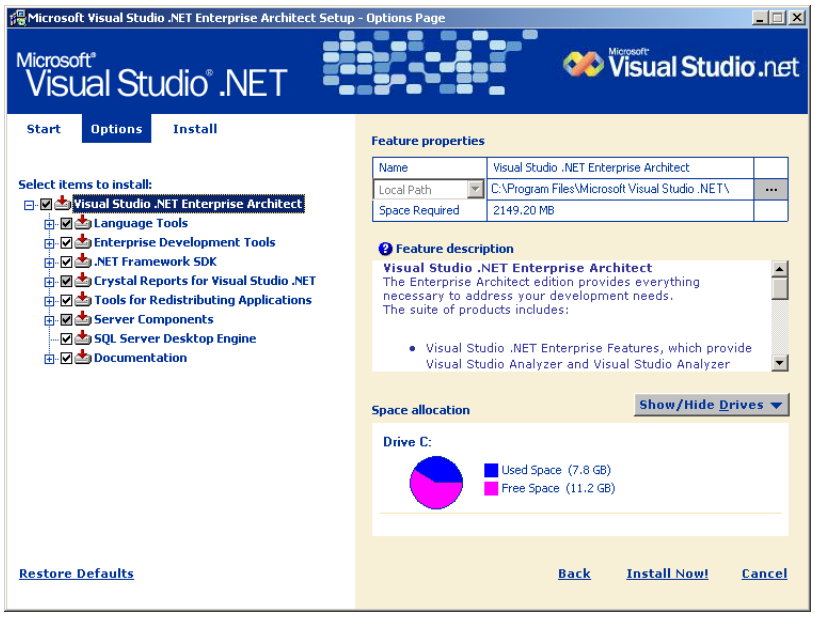

Fig. 1.10 Visual Studio .NET options screen.

11. Click **Install Now!** to begin the installation process. Insert CDs 2, 3 and 4 when prompted to do so. While the installation commences (Fig. 1.11), links are displayed that bring up additional information about certain topics. These topics include the .NET Framework, XML Web Services, Language Enhancements, Integrated Development Environment and MSDN Subscriptions. Click **Done** when installation completes.

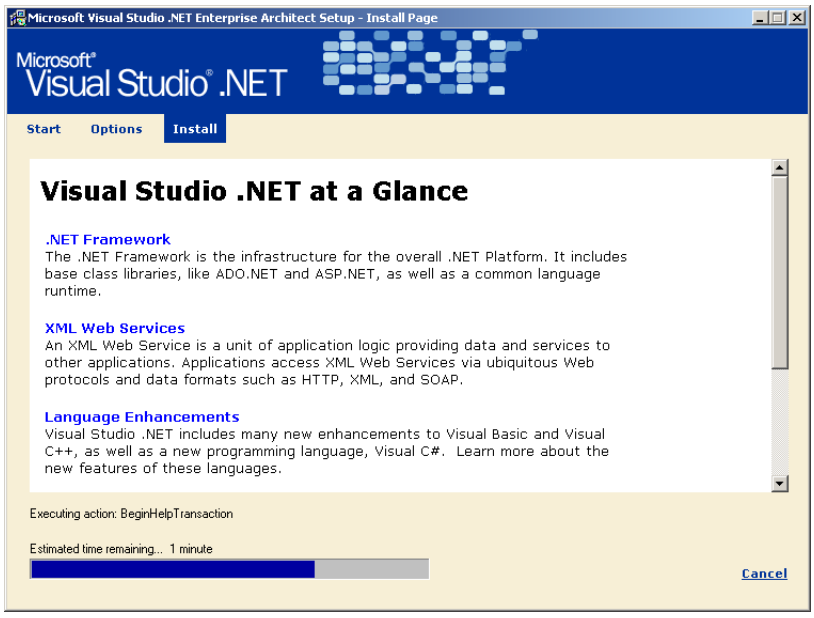

Fig. 1.11 Installing Visual Studio .NET.

The last step, **Service Releases**, obtains updates that might have been released for the Visual Studio environment. This step is not covered in the tutorial, but is as simple as the previous steps. Visual Studio .NET is now installed and ready for use.

1.3 Integrated Development Environment Overview: Visual C++

Figure 1.12 shows the initial screen image of the *Microsoft Visual C++* integrated development environment (*IDE*). This environment contains everything you need to create C++ programs—an *editor* (for typing and correcting your C++ programs), a *compiler* (for translating your C++ programs into machine language code), a *debugger* (for finding logic errors in your C++ programs after they are compiled) and much more. The environment contains many buttons, menus and other graphical user interface (GUI) elements you will use while editing, compiling and debugging your C++ applications.

1.4 On-line Visual C++ Documentation

Visual C++ .NET uses the *Microsoft Developer Network (MSDN™)* documentation (Fig. 1.13), which is accessible by selecting **Microsoft Visual Studio .NET Documentation** from the **Start > Programs > Microsoft Visual Studio .NET** menu. The documentation is also accessible from within Visual Studio .NET. To access this embedded version, select **Contents** from the **Help** menu. Microsoft has combined the documentation for all their development tools into MSDN just as they have combined the development tools (e.g., Visual Basic®, Visual C++, Visual J++®, etc.) into one product suite called *Vi-*

sual Studio®. The on-line documentation for a C++ term is also displayed by clicking the word in an editor window and pressing the *F1 key*.

The Visual C++ documentation is also accessible via the World Wide Web at the *Microsoft Developer Network* Web site

```
http://msdn.microsoft.com/library
```

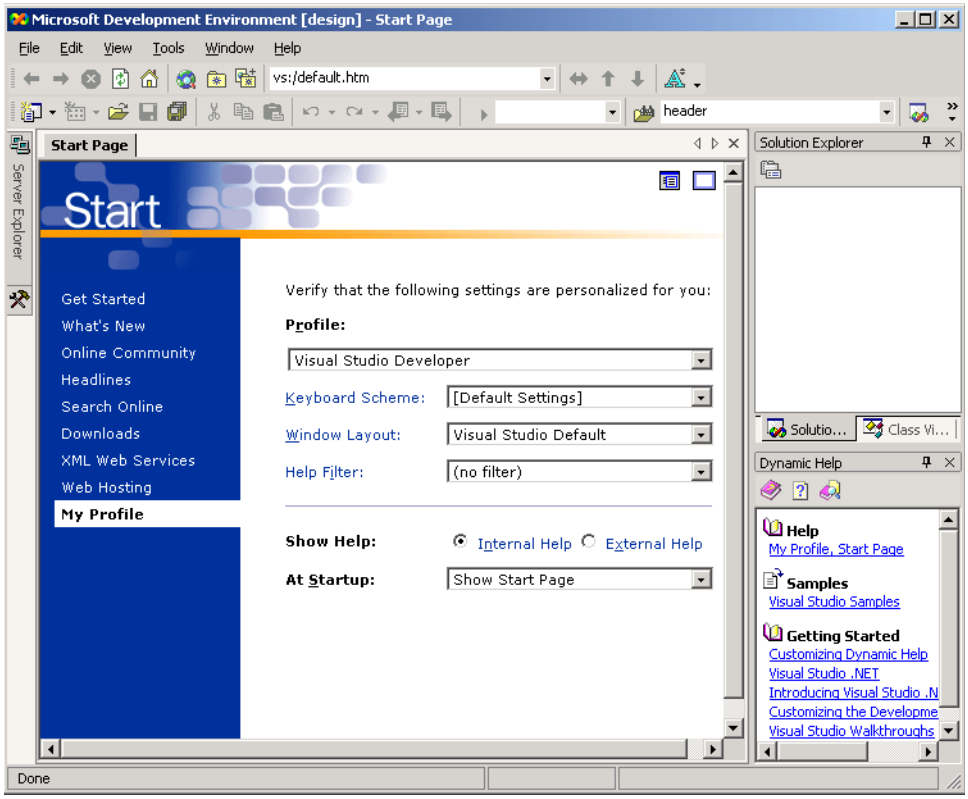

Fig. 1.12 Visual Studio start page.

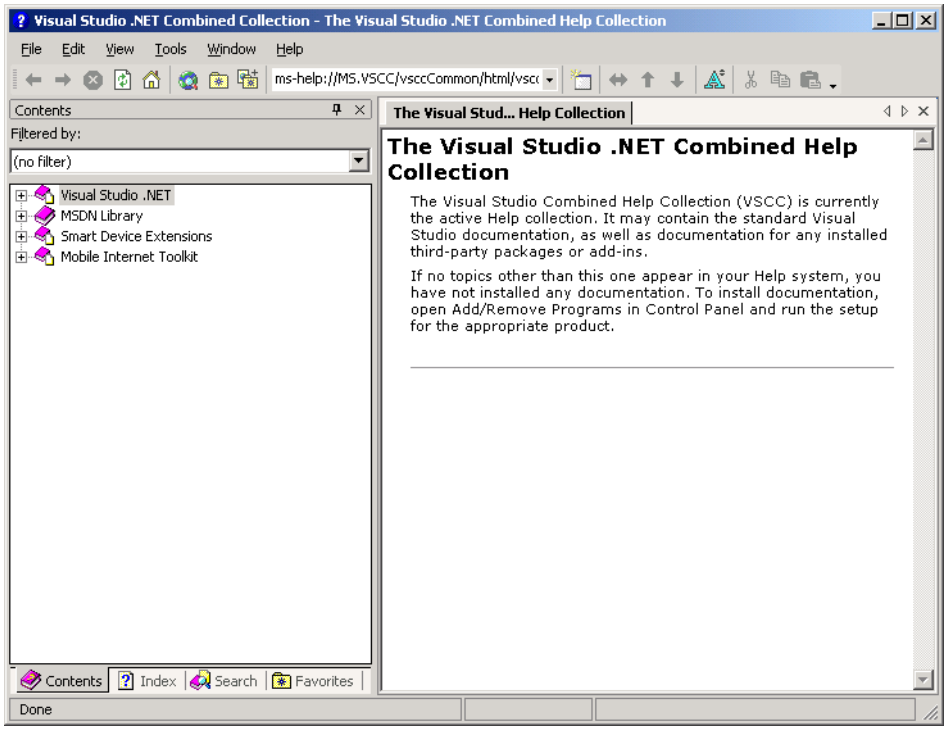

Fig. 1.13 On-line documentation.

If you have not already registered for the Microsoft Developer Network, you will be asked to register. There is no charge for registration at the Web site. The documentation is arranged hierarchically. The Visual C++ documentation is found under

```
.NET Development
   Visual Studio .NET
      Product Documentation
         Visual C++
```

Information about all aspects of Visual C++ is available. Topics range from the Standard C++ Library to the *Microsoft Foundation Classes* (*MFC*). Topics are displayed in *tree-view format* (see the leftmost portion of Fig. 1.13). Clicking the left mouse button on the plus (+) sign next to a topic expands its subtopics. [*Note:* For the rest of this chapter, we refer to *"clicking the left mouse button"* simply as *clicking*.]

The ***Visual Studio .NET Combined Collection*** toolbar (the row of icons near the top of the window in Fig. 1.13) is used to navigate through the on-line documentation in a manner similar to viewing pages in a Web browser. In fact, a modified version of Microsoft's **Internet Explorer** Web browser is used to view the documentation. Clicking the left and right arrows on this toolbar move back and forward, respectively, through any previously viewed pages. The ***Stop*** button causes the program to stop loading the current

topic. The ***Refresh*** button reloads the current topic from the document's source. The toolbar also provides a ***Home*** button that displays the **MSDN Library Visual Studio .NET Help Collection** page.

In the left panel, the user can control the display in the right panel by selecting the ***Filter*** to use and selecting a tab for viewing the ***Contents***, ***Index***, ***Search*** or ***Favorites***. The ***Contents tab*** displays the tables of contents. The ***Index tab*** displays a list of key terms from which to select a topic. The ***Search tab*** allows a programmer to search the entire on-line documentation contents for a word or phrase. The ***Favorites tab*** allows the user to save links to interesting topics for future reference.

On-line information is divided into categories. Each category is preceded by a book icon. ***Visual C++***, under ***Visual Studio .NET***, is the starting point for navigating the on-line documentation for C++.

The **Getting Started** section (Fig. 1.14) contains links to various topics in the documentation, including **Porting and Upgrading**, **Visual C++ Walkthroughs** and **Getting Assistance**. These topics cover a broad range of subjects such that a programmer new to Visual C++, regardless of programming background, can find something of interest.

Getting Started includes multiple topics, such as **What's New in Visual Studio .NET**, **Visual C++ Standard Edition** and **Walkthroughs**. The ***What's New in Visual Studio .NET*** topic explains the newest features introduced in Visual C++ .NET. The **Visual C++ Standard Edition** includes the **Visual C++ Standard Edition Features** topic, which outlines and describes the latest features that relate to libraries, environments, wizards and more.

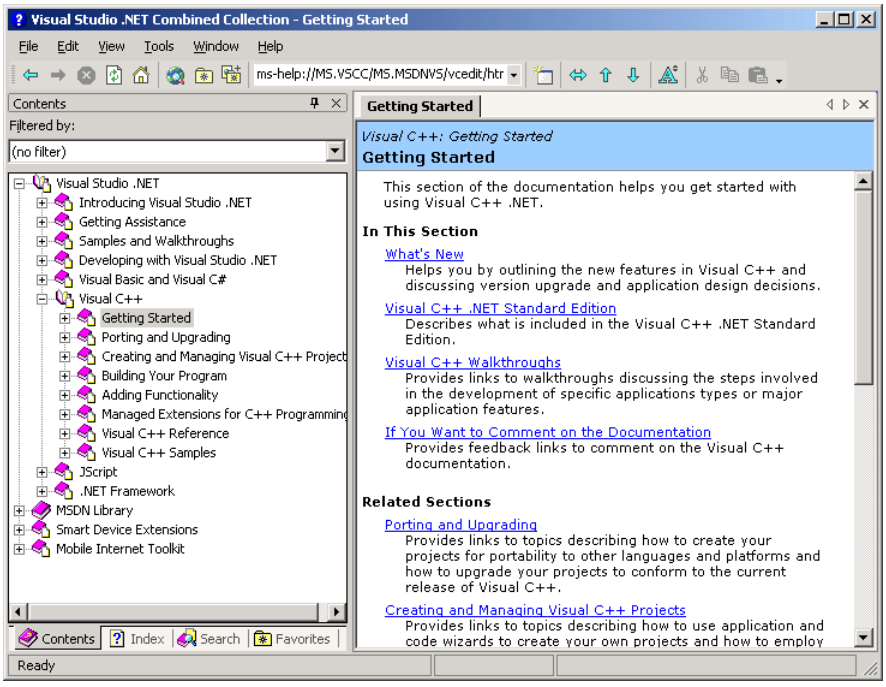

Fig. 1.14 Expanding the Visual C++ topics.

The ***Visual C++ Reference*** category contains multiple subcategories—two of which are ***Visual C++ Libraries*** and ***C/C++ Languages***. The ***Visual C++ Libraries*** category contains information about the *Active Template Library* (*ATL*)—a set of C++ class templates used to develop *distributed applications* (i.e., programs that communicate with each other over a network to perform a task). The category also includes information about the *Run-time library* and *Standard C++ library*—sets of header files that automate programming tasks and provide necessary functionality, such as input and output. The ***C/C++ Languages*** category contains information about Microsoft's implementation of C++. It provides a reference for keywords and operators that are a part of the C++ language. The text in a topic is hyperlinked to related text via the Hypertext Markup Language (HTML) technique of highlighting a term with color and underlining it to indicate which words can be clicked to display a definition or other details about a term.

The ***Visual C++ Samples*** category provides subcategories with example programs for some of the most important features in Visual C++.

1.5 Creating and Executing a C++ Application

You are now ready to begin using the Visual C++ IDE to create a simple *Win32 console application*. When executed, Win32 console applications get input from a *console window* (a text-only display that predates Windows) and display data to a console window. This type of application is used for the example and exercise programs in *C++ How to Program, Fourth Edition*.

Program files in Visual C++ are grouped into *projects*. A project is a text file that contains the names and locations of all its program files. Project file names end with the ***.dsp*** (**des**cribe **p**roject) extension. Before writing any C++ code, you should create a project. Click ***File***, ***New*** and select the **Project...** *menu item* to display the ***New Project*** *dialog* of Fig. 1.15. The **New Project** dialog lists the available Visual C++ project types. Note that your **New Project** dialog may display different project types depending on which Microsoft development tools are installed on your system. When you create a project, you can create a new *workspace* (a folder and control file that act as a container for project files) or combine multiple projects in one workspace. A workspace is represented by a ***.dsw*** (**des**cribe **w**orkspace) file. The examples in this guide have one project per workspace.

Starting with the **New Project** window, a series of dialog windows guides the user through the process of creating a project and adding files to the project. The IDE creates the folders and control files necessary to represent the project.

From the list of project types, click **Visual C++ Projects** and select **Win32 Project** from the list of templates. The ***Name*** field is where you specify the name of the project. Click in the **Name** field and type **Welcome** for the project name.

The ***Location*** field is where you specify the location on disk where you want your project to be saved. If you do not modify the directory path, Visual C++ adds the project name to the value in the **Location** field and stores your projects in this directory. Although this is not shown, we selected our **D:** drive (we edited the **Location** field to display **D:\Welcome**). You may, of course, choose a different location. You can do this by pressing the **Browse** button (Fig. 1.15) and navigating to the desired location. Pressing **OK** closes the **New Project** dialog and displays the **Win32 Application Wizard** dialog (Fig. 1.16).

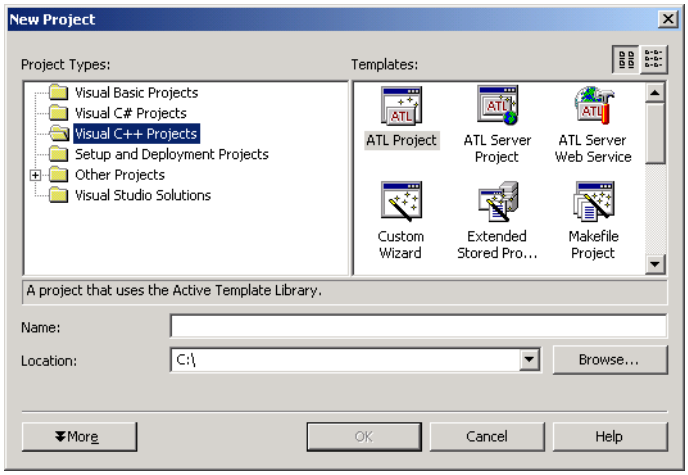

Fig. 1.15 **New Project** dialog displaying lists of Project Types and Templates.

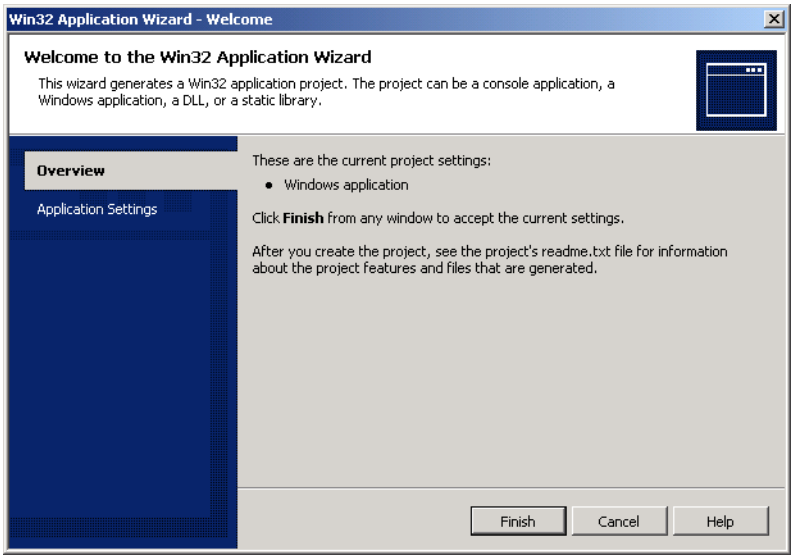

Fig. 1.16 **Win32 Application Wizard Welcome** dialog.

The **Win32 Application Wizard** dialog displays the current project settings. On the left section of the dialog, select **Application Settings** for additional project configuration options, shown in Fig. 1.17. Under **Application type**, select **Console application**

and under **Additional options**, check **Empty project**. A console application simply runs in a console window, rather than in a Windows GUI component, which is the default setting. The **Empty project** option prevents Visual C++ from automatically generating and adding files to the project. The programmer must add source code files to the project. The **Add support for** list allows the programmer to utilize the Microsoft Foundation Classes or the *Standard Template Library*. These options are intended for more complicated programs than this example.

Figure 1.18 shows the Visual C++ IDE after creating an empty **Win32 Project**. The IDE displays the project name (i.e., **Welcome**) in the title bar, and shows the *Solution Explorer pane* and the *Properties pane*. Other panes are available that might not always be open. To open a pane, select it from the **View** menu. For instance, if the *Output pane* is not visible, select **Output** from the **View > Other Windows** menu. The Output pane displays various information, such as the status of your compilation and compiler error messages when they occur.

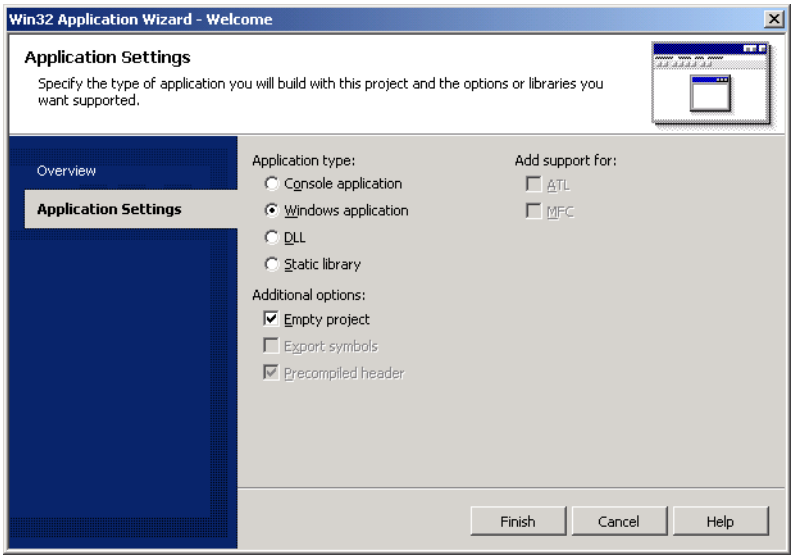

Fig. 1.17 **Win32 Application Wizard**'s Application Settings.

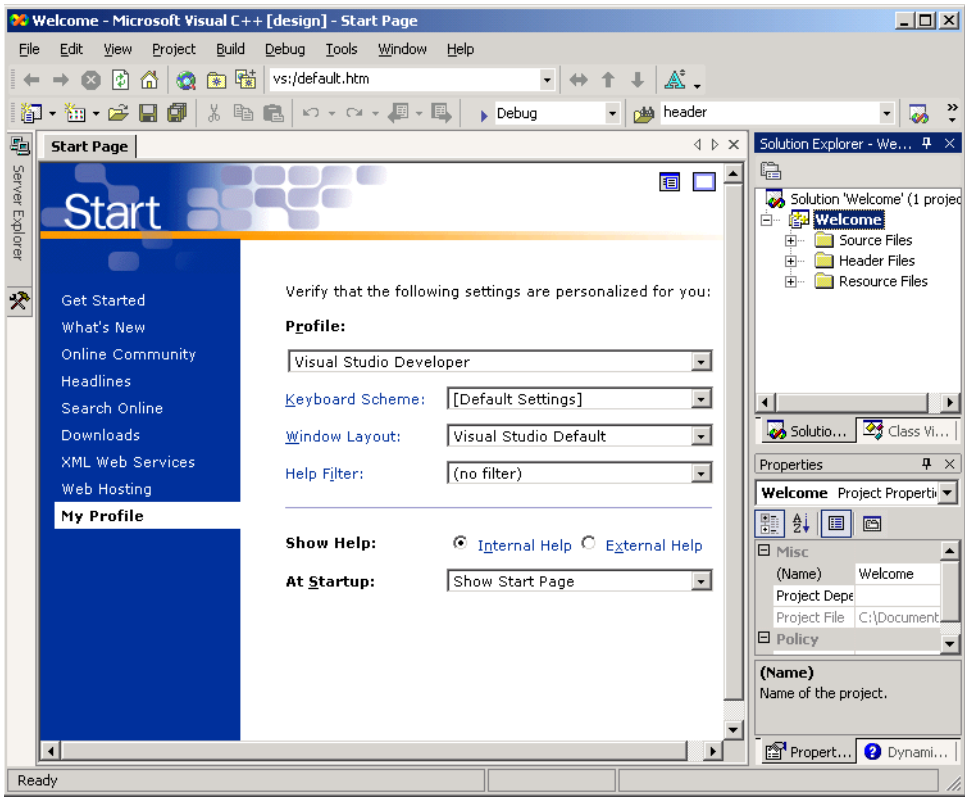

Fig. 1.18 Visual C++ IDE displaying an empty project.

The Solution Explorer pane displays the names of the files that make up the project. Clicking the plus sign, **+,** to the left of **Welcome** displays three empty folders: **Source Files**, **Header Files** and **Resource Files**. **Source Files** displays C++ source files (i.e., **.cpp** files), **Header Files** displays header files (i.e., **.h** files) and **Resource Files** displays resource files (i.e., **.rc** files that define window layouts). For this example we only use the **Source Files** folder. Select the **Class View** tab at the bottom of the Solution Explorer pane to view the classes, class members (discussed in Chapter 6 of *C++ How To Program, Fourth Edition*), and functions (discussed in Chapter 3 of *C++ How To Program, Fourth Edition*) in your project.

The next step is to add a C++ file to the project. Right-click on **Source Files** in the Solution Explorer and select **Add**, then **Add New Item**… to display the **Add New Item…** dialog (Fig. 1.20). The **Add New Item**… dialog displays a list of **Categories** and **Templates**. The options might vary based on the Microsoft development tools installed on your system.

Select ***C++ File*** for a C++ file. The ***Name*** field is where you specify the name of the C++ file. Enter **welcome** in the **Name** field. You do not have to enter the file name suffix "**.cpp**" because it is implied when you select the file type. Do not modify the **Location**

text box. Click **Open** to close the dialog. The C++ file is now saved to disk and added to the project. In our example, the file **welcome.cpp** is saved to the location **D:\Welcome** (the combination of the **Location** field and the project name). Figure 1.21, shows the IDE after adding **welcome.cpp** to the project. In Fig. 1.21 we clicked the **+** character next to **Source Files** to see that the C++ source file is indeed part of the project. The plus **+** becomes a minus **-**, and vice versa, when clicked.

Common Programming Error 1.1

Forgetting to add a C++ source file that is part of a program to the project for that program prevents the program from compiling correctly.

We are now ready to write a C++ program. Type the following sample program into the source code window (Fig. 1.19). [*Note:* The code examples for this guide are available at the Deitel & Associates, Inc. Web site (**www.deitel.com**). Click the "downloads" link to go to our downloads page.]

```
1  // welcome.cpp
2  #include <iostream>
3
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     cout << "Welcome to Visual C++!" << endl;
10
11    return 0;
12
13 }
```

Fig. 1.19 Code for **welcome.cpp**.

The source code window is maximized (also called *docked*) in Fig. 1.21. Click the ***Restore*** button to restore the source code window to its default size.

Testing and Debugging Tip 1.1

Click in front of a brace (i.e., [or {) or a parenthesis (i.e., () and press Ctrl +] to find the matching brace or parenthesis.

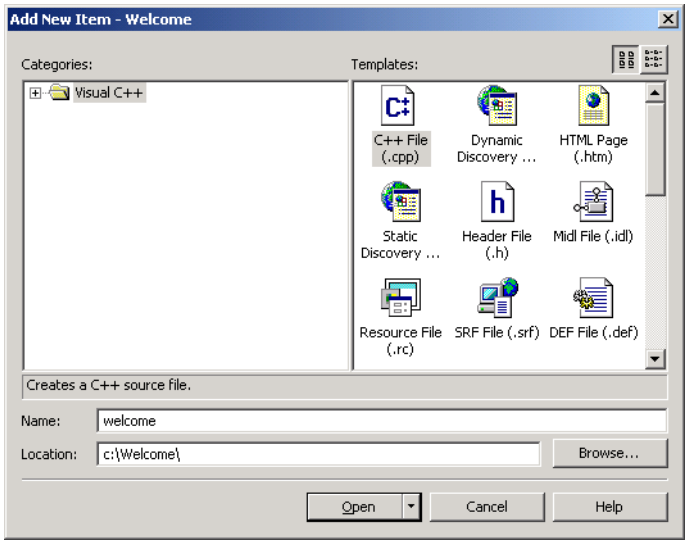

Fig. 1.20 The **Add New Item** dialog displaying a list of templates.

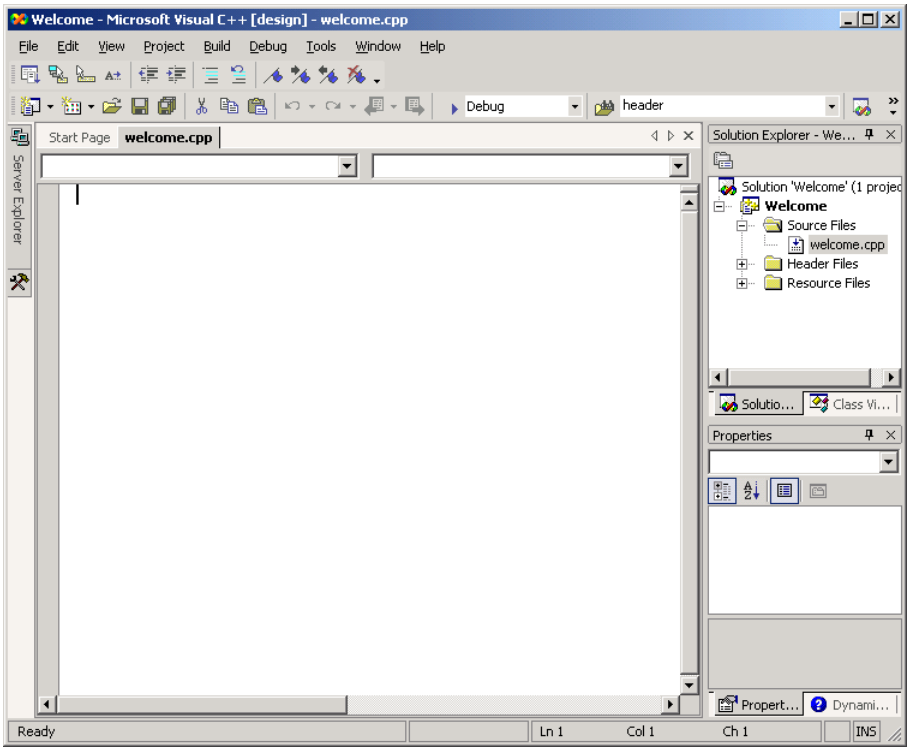

Fig. 1.21 IDE after adding a C++ source file.

The Visual C++ IDE has a highlighting scheme called *syntax coloring* for the keywords and comments in a C++ source file—you may have noticed this while you were typing the program. Syntax color highlighting is applied as you type your code and is applied to all source files opened in Visual C++. By default, keywords appear in blue, comments in green and other text in black, but you can set your own color preferences.

Testing and Debugging Tip 1.2

Visual C++'s syntax highlighting helps the programmer avoid using keywords accidentally as variable names. If a name appears blue (or whatever color you have selected for keywords), it is a keyword and you should not use it as a variable name or other identifier.

Another useful editor feature is *IntelliSense*®. When typing certain language elements, *IntelliSense* displays help automatically to let the programmer select a symbol from a list of names that can appear in the current context in the program; this saves typing time as well as the time it might otherwise take the programmer to look up options.

Figure 1.22 shows an example of *IntelliSense*. When the **c** in **cout** is typed, Visual C++ automatically displays a list of available functions and operators.

Testing and Debugging Tip 1.3

IntelliSense helps the programmer type a correct program.

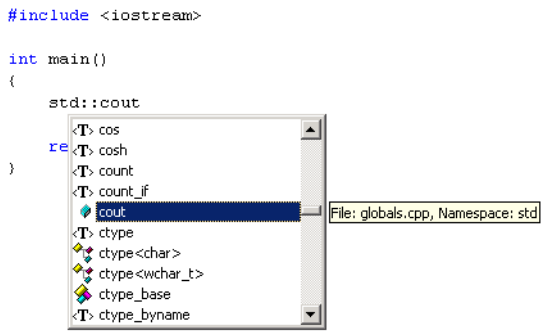

Fig. 1.22 IntelliSense.

After you have typed the program, click **Save** (in the **File** menu) or click the **Save** button (the one that resembles a floppy disk) on the tool bar to save the file.

Before executing a program, you must eliminate all *syntax errors* (also called *compilation errors*) and create an *executable file*. A syntax error indicates that code in the program violates the syntax (i.e., the grammatical rules) of C++.

To compile the C++ file into an executable, click the **Build** menu's **Build Welcome** command. Compiler messages and errors appear in the output pane (Fig. 1.23). If there are no errors when compilation is complete, **Welcome: 0 error(s), 0 warning(s)** should appear in the output pane, as shown in Fig. 1.23 (this is sometimes called the "happy window").

If an error message appears in the output pane, double-clicking anywhere on the error message displays the source file and places a *black arrow marker* in the *margin indicator bar* (i.e., the gray strip to the left of the source code), indicating the offending line as shown in Fig. 1.24. The error in this particular case is a missing **<** character after **cout**.

Error messages are often longer than the output pane's width. The complete error message can be viewed either by using the horizontal scrollbar above the tabs at the bottom of the screen or by reading the *status pane*, located at the bottom of the IDE. The status pane displays only the selected error message.

If you do not understand the error message, highlight the error message number by dragging the mouse over the number, then press the *F1 key*. This displays a help file that provides information about the error and some helpful hints as to the cause of the error. Please keep in mind that C++ compilers may mark a line as having an error when, in fact, the error occurs on a previous line of code.

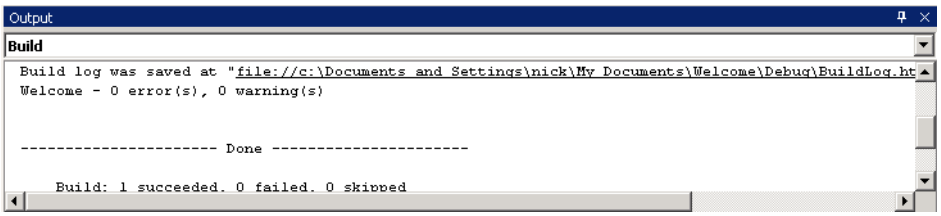

Fig. 1.23 **Output** pane showing a successful build.

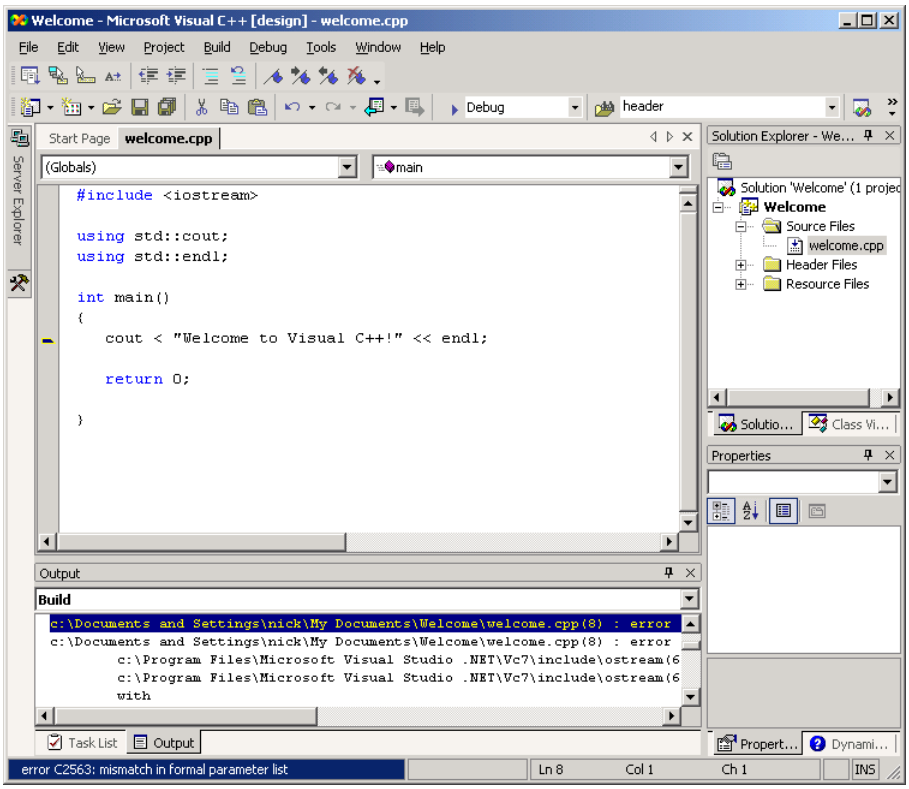

Fig. 1.24 Black marker indicating that a line contains an error.

After fixing the error(s), recompile the program. C++ compilers often list more errors than actually occur in the program. For example, a C++ compiler may locate a syntax error in your program (e.g., a missing semicolon). That error may cause other errors in the program when, in fact, there may not be any other errors.

Testing and Debugging Tip 1.4

When a syntax error on a particular line is reported by the compiler, check that line for the syntax error. If the error is not on that line, check the preceding few lines of code for the cause of the syntax error.

Testing and Debugging Tip 1.5

After fixing one error, recompile your program. You may observe that the number of overall errors perceived by the compiler is significantly reduced.

Once the program compiles without errors, you can execute the program by clicking **Start Without Debugging** in the **Debug** menu. The program is executed in a console window as shown in Fig. 1.25. Pressing any key closes the console window.

Fig. 1.25 C++ program executing in a console window.

To create another application, follow the same steps outlined in this section using a different project name and directory. Before starting a new project, close the current project by selecting the **File** menu's ***Close Solution*** menu item. If a dialog appears asking if all document windows should be closed or if a file should be saved, click **Yes**. You are now ready to create a new project for your next application or open an existing project. To open an existing project, in the **File** menu you can select the **Recent Projects** option to select a recent workspace or select **Open** then **Project...** to see a dialog and select a workspace (**.dsw** file) to open.

1.6 Compiling Programs with Multiple Source Files

More complex programs often consist of multiple C++ source files. We introduce this concept, called *multiple source files*, in Chapter 6 of *C++ How To Program, Fourth Edition*. This section explains how to compile a program with multiple source files using the Microsoft Visual Studio .NET IDE.

Compiling a program that has two or more source files will be demonstrated using the **`Time`** class (Fig. 6.5–Fig. 6.7 from Chapter 6 of *C++ How To Program, Fourth Edition*), shown in Figure 1.26–Figure 1.28.

```
1  // Fig. 6.5: time1.h
2  // Declaration of class Time.
3  // Member functions are defined in time1.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME1_H
7  #define TIME1_H
8
9  // Time abstract data type definition
10 class Time {
11
12 public:
13    Time();                          // constructor
14    void setTime( int, int, int ); // set hour, minute, second
15    void printUniversal();           // print universal-time format
16    void printStandard();            // print standard-time format
17
18 private:
19    int hour;     // 0 - 23 (24-hour clock format)
20    int minute;   // 0 - 59
21    int second;   // 0 - 59
22
23 }; // end class Time
24
25 #endif
```

Fig. 1.26 **Time** class definition.

```
1  // Fig. 6.6: time1.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4
5  using std::cout;
6
7  #include <iomanip>
8
9  using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time1.h
13 #include "time1.h"
14
15 // Time constructor initializes each data member to zero.
16 // Ensures all Time objects start in a consistent state.
17 Time::Time()
18 {
19    hour = minute = second = 0;
20
21 } // end Time constructor
22
23 // Set new Time value using universal time. Perform validity
24 // checks on the data values. Set invalid values to zero.
```

Fig. 1.27 **Time** class member-function definitions.

```
25  void Time::setTime( int h, int m, int s )
26  {
27     hour = ( h >= 0 && h < 24 ) ? h : 0;
28     minute = ( m >= 0 && m < 60 ) ? m : 0;
29     second = ( s >= 0 && s < 60 ) ? s : 0;
30
31  } // end function setTime
32
33  // print Time in universal format
34  void Time::printUniversal()
35  {
36     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
37          << setw( 2 ) << minute << ":"
38          << setw( 2 ) << second;
39
40  } // end function printUniversal
41
42  // print Time in standard format
43  void Time::printStandard()
44  {
45     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
46          << ":" << setfill( '0' ) << setw( 2 ) << minute
47          << ":" << setw( 2 ) << second
48          << ( hour < 12 ? " AM" : " PM" );
49
50  } // end function printStandard
```

Fig. 1.27 `Time` class member-function definitions.

```
 1  // Fig. 6.7: fig06_07.cpp
 2  // Program to test class Time.
 3  // NOTE: This file must be compiled with time1.cpp.
 4  #include <iostream>
 5
 6  using std::cout;
 7  using std::endl;
 8
 9  // include definition of class Time from time1.h
10  #include "time1.h"
11
12  int main()
13  {
14     Time t;  // instantiate object t of class Time
15
16     // output Time object t's initial values
17     cout << "The initial universal time is ";
18     t.printUniversal();   // 00:00:00
19     cout << "\nThe initial standard time is ";
20     t.printStandard();    // 12:00:00 AM
21
22     t.setTime( 13, 27, 6 );   // change time
23
24     // output Time object t's new values
```

Fig. 1.28 Program to test class `Time`.

```
25     cout << "\n\nUniversal time after setTime is ";
26     t.printUniversal();   // 13:27:06
27     cout << "\nStandard time after setTime is ";
28     t.printStandard();    // 1:27:06 PM
29
30     t.setTime( 99, 99, 99 );  // attempt invalid settings
31
32     // output t's values after specifying invalid values
33     cout << "\n\nAfter attempting invalid settings:"
34          << "\nUniversal time: ";
35     t.printUniversal();   // 00:00:00
36     cout << "\nStandard time: ";
37     t.printStandard();    // 12:00:00 AM
38     cout << endl;
39
40     return 0;
41
42  } // end main
```

Fig. 1.28 Program to test class **`Time`**.

1. The first step to compiling multiple source files is to create a project. Follow the steps in Section 1.5 of this guide to create a project.
2. Next create the **`.cpp`** source files using the same method described in Section 1.5.
3. If your program contains **`.h`** header files, follow the steps for adding a **`.cpp`** file, only right click on the **Header Files** folder instead of the **Source Files** folder in the **Solution Explorer**. Figure 1.29 shows an example project containing the **`Time`** class example.

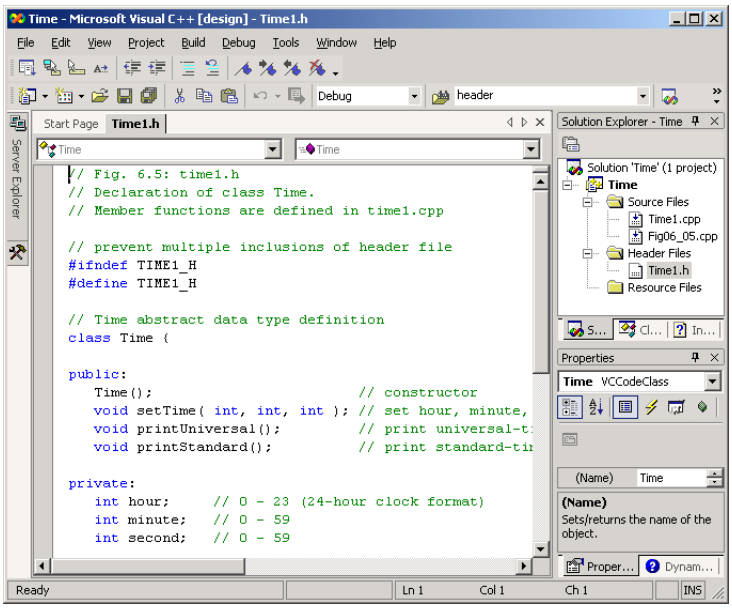

Fig. 1.29 Visual Studio project containing multiple source files.

4. Next, use the method described in Section 1.5 to compile and run the program. Figure 1.30 shows the output for the **`Time`** class example.

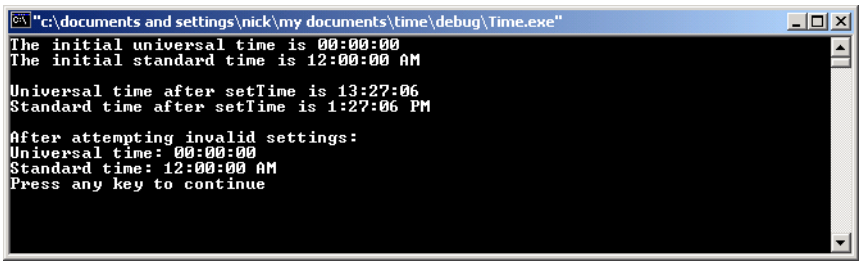

Fig. 1.30 Output for **`Time`** class example.

1.7 Debugger

The Visual C++ IDE provides a *debugger* tool to help the programmer find run-time logic errors in programs that compile and link successfully but do not produce expected results. The debugger lets the programmer view the executing program and its data as the program runs either one step at a time or at full speed. The program stops on a selected line of code or upon a fatal run-time error. When the programmer does not understand how incorrect results are produced by a program, running the program one statement at a time and monitoring the intermediate results can help the programmer isolate the cause of the error. The programmer can then correct the code.

To use the debugger, set one or more *breakpoints*. A breakpoint is a marker set at a specified line of code that causes the debugger to suspend execution of the program upon reaching that line of code. Breakpoints help the programmer verify that a program is executing correctly. A breakpoint is set by clicking the line in the program where the breakpoint is to be placed and clicking the **New Breakpoint...** button in the **Debug** toolbar (Fig. 1.31). The **New Breakpoint...** button is disabled unless the C++ code window is the active window (clicking in a window makes it active). When a breakpoint is set, a solid red circle appears in the margin indicator bar to the left of the line. Breakpoints are removed by clicking the line with the breakpoint and clicking the **Remove Breakpoint** button or pressing the *F9 key*. To toggle a breakpoint on or off, right-click on the appropriate line of code.

Fig. 1.31 Debug toolbar

Breakpoints are persistent, meaning when a project is closed and reopened, any breakpoints set during a previous debugging session remain set. You can gather information about breakpoints by selecting the ***Breakpoints*** tab from the bottom-right sub-window of the IDE. When selected, the **Breakpoints** tab can be used to display the ***New Breakpoint...*** dialog (Fig. 1.32), by clicking the **New Breakpoint** button.

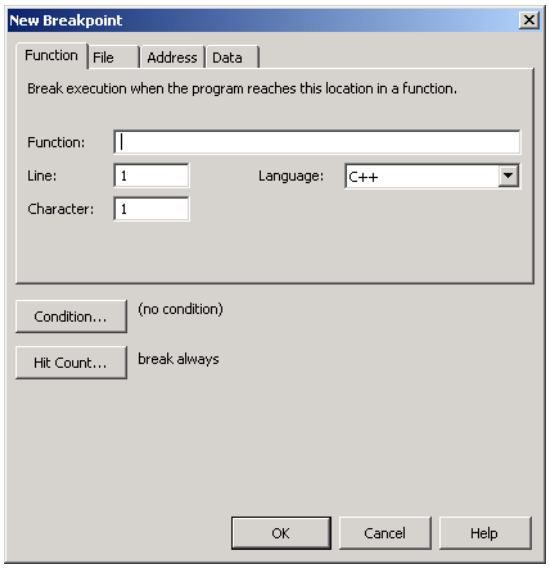

Fig. 1.32 **New Breakpoint** dialog.

The **New Breakpoint** dialog displays all the breakpoints currently set for the program. A checkbox appears next to each breakpoint. If the breakpoint is *active*, the checkbox contains a check. If the breakpoint is *disabled*, the checkbox is empty. A disabled breakpoint will not cause the debugger to stop but may be re-enabled at a later time. Clicking the checkbox allows the user to toggle the breakpoint *on* (checked) or *off* (unchecked). Additional breakpoints can be added by entering the desired line number into the ***Line*** field after clicking on the **New** button. Breakpoints can be set in several ways. Using the **Function** tab allows the programmer to set a breakpoint at a certain location within the scope of a function. The **File** tab allows the programer to set a breakpoint at a desired line of code within a specified file. A breakpoint can be set at a memory location by using the **Address** tab. Lastly, a breakpoint can be set to halt the program when a certain variable's value changes by using the **Data** tab.

Visual C++ also allows breakpoints to be enabled when certain conditions are true. The programmer specifies the line number in the **Line** field, in the **File** tab, and presses the **Condition...** button to display the ***Breakpoint Condition*** dialog (Fig. 1.33). A condition is specified in the **Condition** field. Figure 1.34 shows the **Breakpoints** dialog with the new breakpoint.

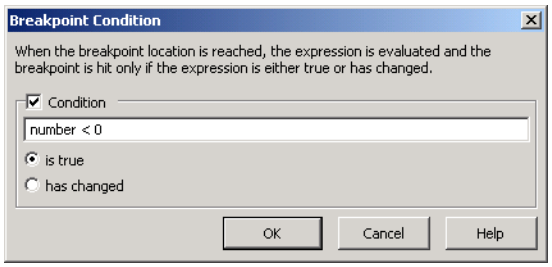

Fig. 1.33 Setting a condition for a breakpoint.

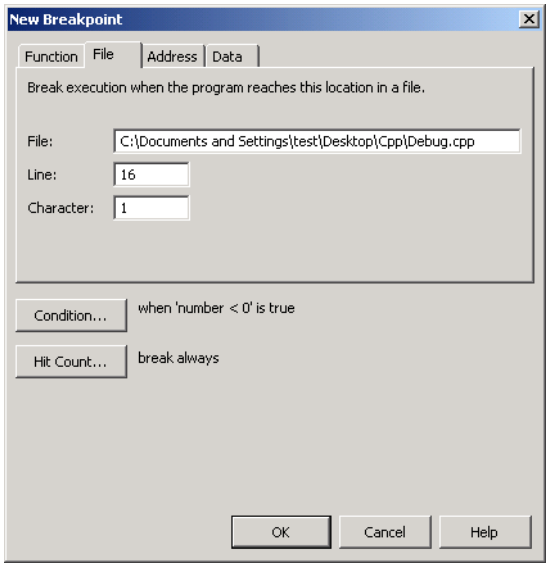

Fig. 1.34 **New Breakpoint** window after a condition has been set.

Figure 1.35 shows the debugging environment with a disabled breakpoint. Notice that the disabled breakpoint is still visible but it appears as an empty circle. To make the breakpoint active, click the empty checkbox next to the breakpoint in the **Breakpoints** dialog.

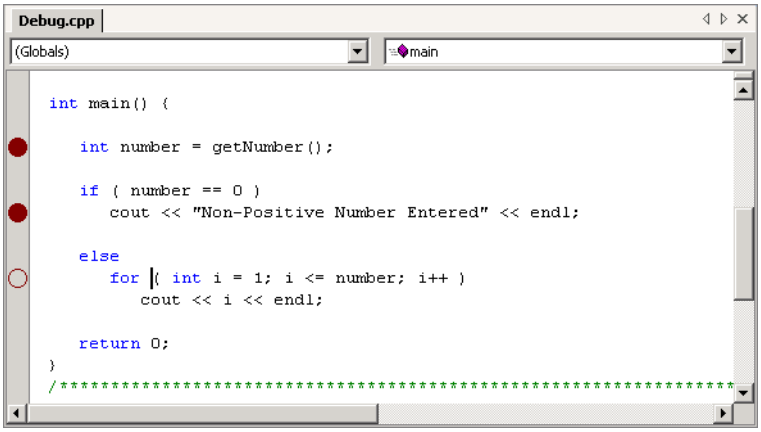

Fig. 1.35 Debugger with disabled breakpoint.

Testing and Debugging Tip 1.6

Loops that iterate many times can be executed in full (without stopping every time through the loop) by placing a breakpoint after the loop and selecting ***Go*** *from the* ***Debug*** *menu.*

Testing and Debugging Tip 1.7

Disabled breakpoints allow the programmer to maintain breakpoints in key locations in the program so they can be used again when needed. Disabled breakpoints are always visible.

The bottom portion of the IDE is divided into two windows—the **Autos** window (i.e., the left window) and the ***Call Stack*** window (i.e., the right window). The **Autos** window contains a list of the program's initialized variables. Note that different variables can be viewed at different times, by clicking either the ***Autos***, ***Locals*** or ***Watch 1*** tabs. The **Autos** tab displays the name and value of the variables or objects (discussed in Chapter 6 of *C++ How To Program, Fourth Edition*). The **Locals** tab displays the name and current value for all the local variables or objects in the current function's scope. The **Watch 1** tab displays data about any objects or variables being watched.

The variable values listed in any of the tabs can be modified by the user for testing purposes. To modify a variable's value, click the ***Value*** *field* and enter a new value. Any modified value is colored red to indicate that it was changed during the debugging session by the programmer.

Often, certain variables are monitored by the programmer during the debugging process—a process known as *setting a watch*. The **Watch 1** tab allows the user to watch variables as their values change. Changes are displayed in the **Watch 1** tab.

Variables can be typed directly into the **Watch 1** tab or dragged with the mouse from either the other tabs or the source code window and dropped into the **Watch 1** tab. A variable can be deleted from the **Watch 1** tab by selecting the variable name and pressing the *Delete* key.

Like the **Autos** tab, variable values can be modified in the **Watch 1** tab by editing the **Value** field. Changed values are colored red. Note also that the current value of a variable during debugging can also be viewed by resting the mouse cursor over the name of that variable in the source code window (Fig. 1.36).

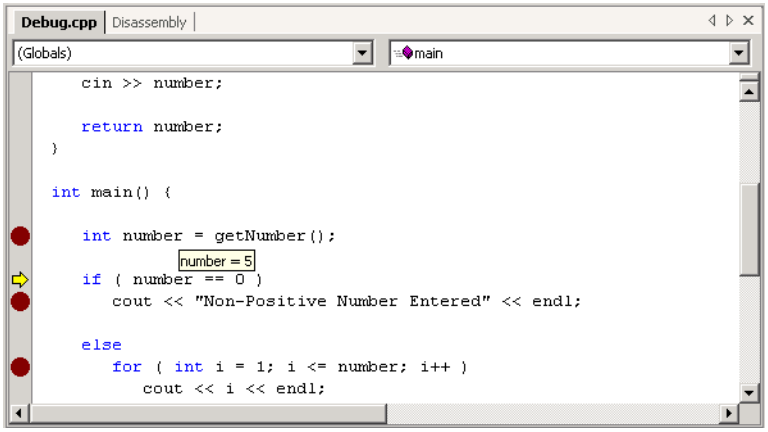

Fig. 1.36 Displaying a variables value using the mouse pointer.

The **Debug** toolbar(Fig. 1.31) contains buttons that control the debugging process. These buttons perform the same actions as the **Debug** menu items. The **Debug** toolbar can be displayed by positioning the mouse pointer over an empty region of the main menu or any toolbar, right-clicking the mouse and selecting the **Debug** option in the popup menu.

The ***Restart*** button restarts the application, stopping at the beginning of the program to allow the programmer to set breakpoints before starting to execute the code. The ***Stop Debugging*** button ends the debugging session to let the programmer edit and rebuild the program before running another test.

Code can be altered when the program is suspended in debug mode. Once the debugger resumes, the IDE asks if the programmer would like to continue using the changes made. This might require Visual Studio .NET to re-build the program. [Refer to the on-line documentation for limitations on this feature.] **Show Next Statement** places the cursor on the same line as the yellow arrow that indicates the next statement to execute. ***Show Next Statement*** is useful to reposition the cursor to the same line as the yellow arrow when viewing the source code during debugging.

The ***Step Into*** button executes program statements, one per click, including code in functions that are called, allowing the programmer to confirm the proper execution of the function, line-by-line. Functions that can be stepped into include programmer-defined functions and C++ library functions. If you want to step into a C++ library function, Visual C++ may ask you to specify the location of that library.

Testing and Debugging Tip 1.8

The debugger allows you to "step into" a C++ library function to see how it uses your function call arguments to produce the value returned to your program.

The ***Step Over*** button executes the next executable line of code and advances the yellow arrow to the following executable line in the program. If the line of code contains a function call, the function is executed in its entirety as a single step. This allows the user to execute the program one line at a time and examine the execution of the program without seeing the details of every function that is called. This is especially useful at **`cin`** and **`cout`** statements.

The ***Step Out*** button allows the user to step out of the current function and return control back to the line that called the function. If you **Step Into** a function that you do not need to examine, click **Step Out** to return to the caller.

Testing and Debugging Tip 1.9

Loops that iterate many times can be executed in full by placing the cursor after the loop in the source code window, right clicking and selecting the ***Run to Cursor*** *item.*

Testing and Debugging Tip 1.10

If you accidently step into a C++ library function, click ***Step Out*** *to return to your code.*

The ***QuickWatch*** button displays the ***QuickWatch*** *dialog* (Fig. 1.37), which is useful for monitoring expression values and variable values. The **QuickWatch** dialog provides a "snapshot" of one or more variable values at a point in time during the program's execution. To watch a variable, enter the variable name or expression into the ***Expression*** field and press *Enter*. As with the **Autos** window and **Watch 1** window, values can be edited in the **Value** field, but changed values are not color coded red. Clicking ***Recalculate*** is the same as pressing *Enter*.

To maintain a longer watch, click the ***Add Watch*** button to add the variable to the **Watch 1** tab. When the **QuickWatch** dialog is dismissed by clicking **Close**, variables in the dialog are not preserved. The next time the **QuickWatch** dialog is displayed, the ***Name*** and ***Value*** fields are empty. The **QuickWatch** window can also be used to evaluate expressions such as arithmetic calculations (e.g., **a + b - 9**, etc.) and variable assignments (e.g., **number = 20**, etc.) by typing the expression into the **Expression** field.

The **Call Stack** window contains the program's *function call stack*. A function call stack is a list of the functions that were called up to the current line in the program. This helps the programmer see the flow of control that led to the current function being called.

The right sub-window also contains tabs for switching to the **Command** window and the **Output** window. The **Command** window allows the programmer to enter input from within IDE. The **Output** window allows the programmer to see the program output from within the IDE as well.

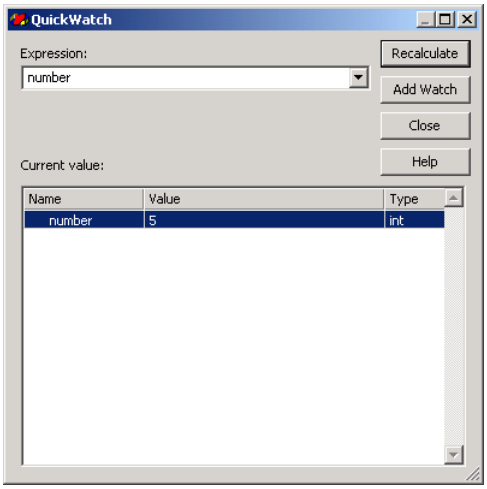

Fig. 1.37 **QuickWatch** dialog.

Testing and Debugging Tip 1.11

When using the debugger to run a program at full speed, certain problems such as infinite loops can usually be interrupted by selecting ***Break All*** *from the* ***Debug*** *menu.*

1.7.1 Debugging an Application

This section guides the programmer through the debugging process for a simple C++ application, **`Debug.cpp`** (Fig. 1.38). This applications obtains a number from the user and counts from 1 to that number.

```
1  // Debug.cpp
2
3  #include <iostream>
4
5  using std::cin;
6  using std::cout;
7  using std::endl;
8
9  // function that gets an integer from the user
10 int getNumber()
11 {
12    int number;    // holds user input integer
13
14    // ask user for and store integer
15    cout << "Enter an integer: ";
16    cin >> number;
17
18    return number; // return integer entered by user
19
20 } // end function getNumber
```

Fig. 1.38 Code for **`Debug.cpp`**

```
21
22  int main()
23  {
24     // get integer from user
25     int number = getNumber();
26
27     // end program if user does not enter positive number
28     if ( number <= 0 )
29        return 0;
30
31     // display integers from one to user input number
32     else
33        for ( int i = 1; i <= number; i++ )
34           cout << i << endl;
35
36     return 0;
37
38  } // end main
```

Fig. 1.38 Code for **Debug.cpp**

1. In order to debug the code, **Debug.cpp** needs to be compiled into an executable file. Create a project for **Debug.cpp** as described in Section 1.5. Next, compile the program into an executable file.
2. In the window that contains the source code, a breakpoint is added by clicking the line in the program where the breakpoint is to be placed and clicking the **New Beakpoint...** button in the **Debug** toolbar. A breakpoint can also be added by clicking in the gray box to the left of the line of code where the breakpoint is to be placed. Click in the gray box to the left of line 25. The red circle that appears indicates that a breakpoint has been set at that line.
3. Repeat step 2, only this time set breakpoints at lines 29, 33 and 36. When complete the window should appear as shown (Fig. 1.39).

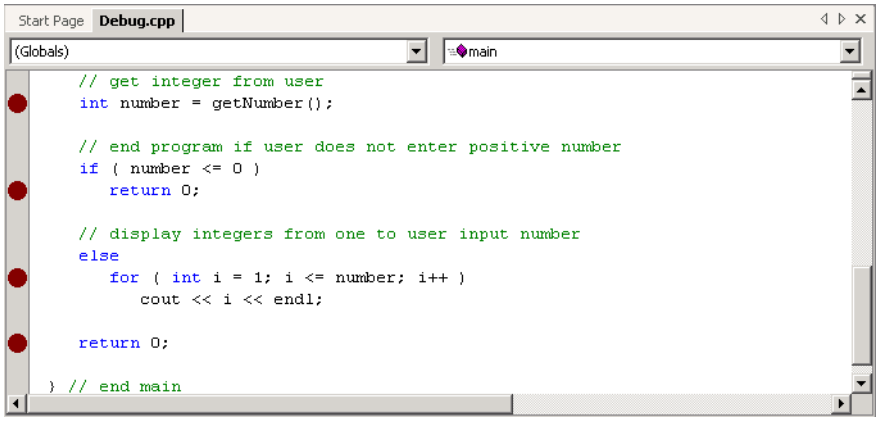

Fig. 1.39 Breakpoints set in a program.

4. Click **Start** from the **Debug** menu to start the debugger. Because we have chosen to debug a console application, the console window (i.e., command prompt) appears (Fig. 1.40). All program interaction (input and output) occurs within this window. Program execution suspends for input and at breakpoints. You might need to manually switch between the IDE and the console window to perform input. To switch between windows, use *Alt* + *Tab* or click the program's panel on the Windows taskbar at the bottom of the screen.
   Figure 1.41 shows program execution suspended at a breakpoint. The *yellow arrow* to the left of the statement

   ```
   int number = getNumber();
   ```

   indicates that execution is suspended at this line. This statement will be the next statement executed. Note in the IDE that the title bar displays **[break]** to indicate that the IDE is in *debug mode*.

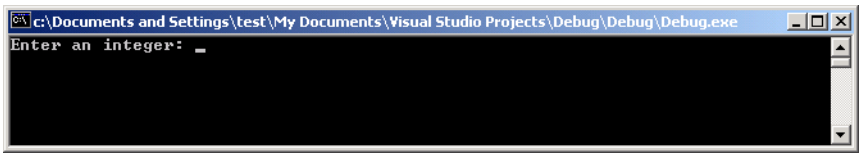

Fig. 1.40 C++ program executing in a command window during debug mode.

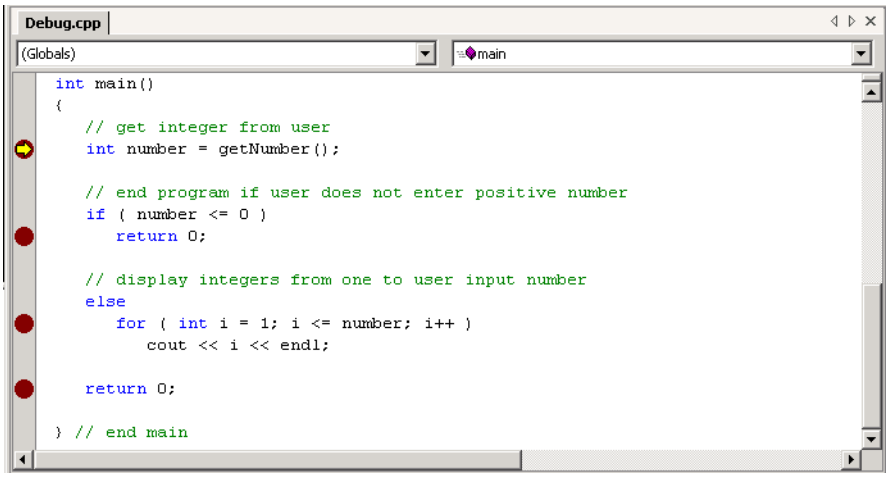

Fig. 1.41 Execution halted at a breakpoint.

5. To add a watch, type the text number into the name field of the **Watch1** tab located at the bottom of the IDE. Notice that the value for **number** is its memory address because it has not been given a value.

6. Click **Continue**. At this point, the program is executing and the **Input** dialog is displayed. Enter 10 into the text field and hit Enter. The program briefly resumes execution and then suspends. Add another watch for variable **`i`**. This watch can only be added when within the scope of **`i`**, meaning inside the **`for`** loop. If the program is not within the scope of **`i`**, an error will be displayed for the value of **`i`** in the **Watch 1** pane. The **Watch 1** pane now displays information about the integer **`number`** and the variable **`i`**. The text is changed to a red font to indicate that changes have been made to that variable. **`number`** is used to store the number entered by the user, therefore it always maintains its original value. Figure 1.42 shows watches set for **`number`** and **`i`**.

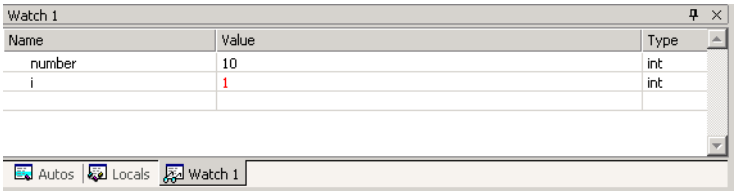

Fig. 1.42 Watches set for **`number`** and **`i`**.

7. Click **Step Over** to execute the individual steps of the loop each time the loop is executed. Or click **Step Out** to perform the whole loop in one step. Clicking **Continue** will go to the next break point.
8. The main window indicates that the debugger has completed and the count from 1 to 10 is displayed in the output window (Fig. 1.43). Even though a breakpoint was set on line 29, the program never suspended on that line because the code on line 29 never executed. The code on lines 29 and 34 either end the program or display the numbers to the outut window, depending on the number entered by the user. Start the debugger again, by clicking **Restart** in the **Debug** menu, but enter a non-positive number for the value of number into the program input dialog and observe how the debugger operates.

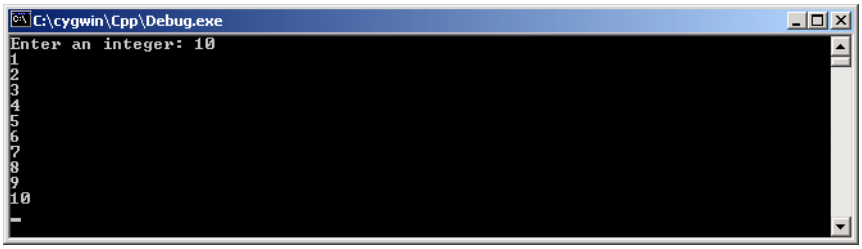

Fig. 1.43 Output of Debug.cpp.

9. When you have finished your debugging session, click the ***Stop Debugging*** button on the **Debug** toolbar. The environment returns to the pre-debugging layout. Refer to the on-line documentation for additional debugger features.