1

Deitel® Dive-Into™ Series: Dive Into Microsoft Visual C++ 6

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 6 integrated development environment.
- To be able to search Microsoft's online documentation effectively.
- To be able to use the debugger to locate program logic errors.

Outline

1.1 Introduction

Welcome to the Visual C++® 6 Integrated Development Environment (part of the *Microsoft Visual Studio® 6* 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++ 6*. When you complete this tutorial, you will be able to use Visual C++ to begin building applications.

This guide does not teach C++; rather, it is intended to be used 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 any libraries for creating GUIs. Compiler vendors such as Microsoft, Borland and Symantec normally provide their own libraries that support creation of applications with GUIs. Our readers asked us to use the same "live-code" approach with outputs that we employ in all our *How to Program Series* textbooks.

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 guide 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 3.6 of this guide, you should be familiar with the material in Chapter 6, "Classes and Data Abstraction," in *C++ How to Program, Fourth Edition*.

We hope you enjoy learning about the Visual C++ 6 integrated development environment.

1.2 Installation of Visual C++ Introductory Edition

This section will guide the user through the installation process for Visual C++ Introductory Edition. Visual C++ Introductory Edition is included on the CD accompanying *C++ How to Program, Fourth Edition*. To install this programming environment perform the following steps:

1. Insert the CD into the CD-ROM drive. A dialog will pop up and display three buttons. Selecting the **Welcome** button will take you to another page containing installation information, license agreement as well as technical support. Selecting

the **Examples** button will allow you to view all the examples from *C++ How to Program, Fourth Edition*. Selecting the **Links** button will direct you to the extra resources available on the World Wide Web.

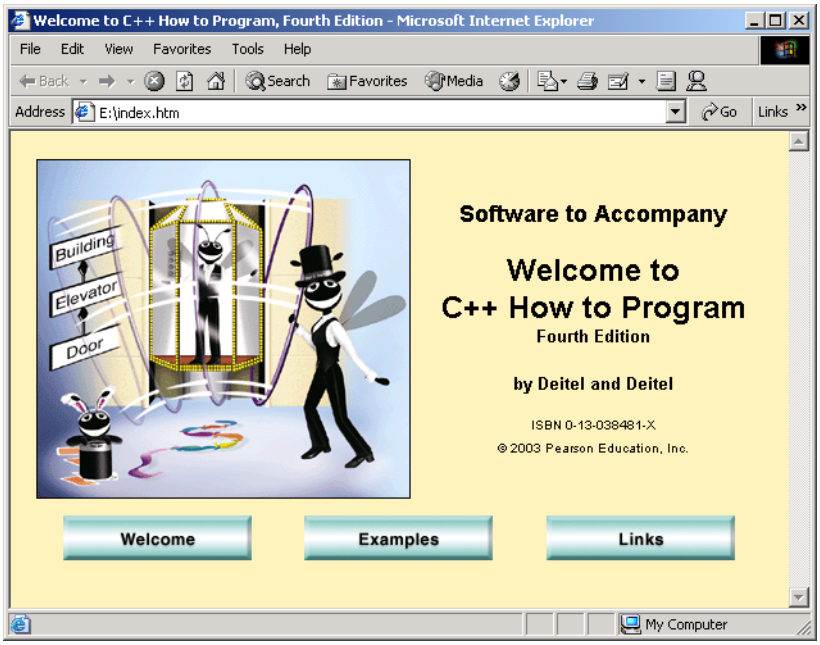

Fig. 1.1 Welcome screen

2. From the taskbar click **Start**, select **Run...**, enter the following path into the **Open:** textfield, **`E:\VCB600ENU1\SETUP.EXE`** and click **OK**. [*Note:* we are assuming that **E** is the letter of the CD-ROM drive.]

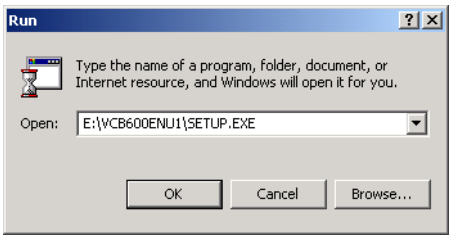

Fig. 1.2 Executing **`SETUP.EXE`**

3. This displays the **Installation Wizard for Visual C++ 6.0 Introductory Edition**. Click **Next >** to continue with the installation.

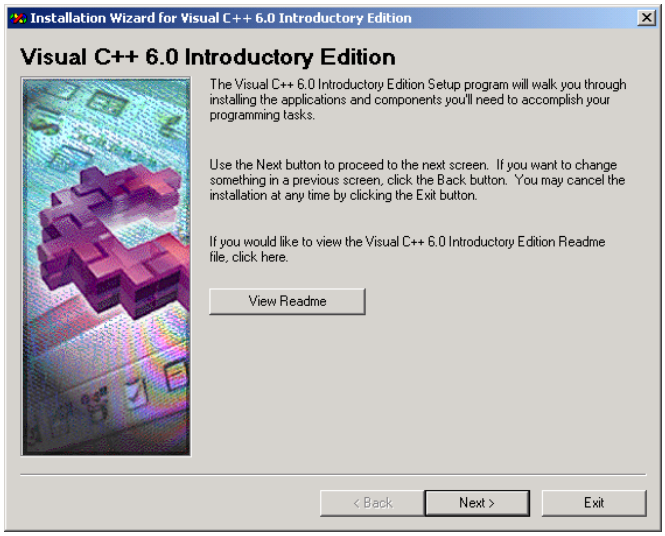

Fig. 1.3 Installation wizard

4. Read the license agreement to understand the terms and conditions for using the software. If you agree to the license, select **I accept the agreement** and click **Next >.**
5. You will be prompted to enter your name and your company's name. Since this is an introductory edition, you do not have to provide a **Product Number.** Click **Next >** to continue.

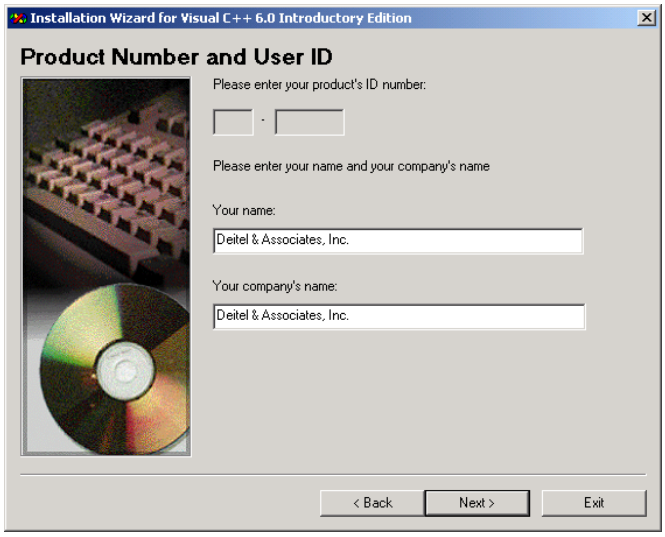

Fig. 1.4 Product Number and User ID

6. The **Choose Common Install Folder** screen will appear. Here you can specify a directory to store files common among Visual Studio 6.0. Specify a new location or click **Next >** to accept the default directory.
7. The **Visual C++ 6.0 Introductory Setup** screen appears. Click **Continue** to begin the setup. You are presented a **Product ID** number**,** write this number down, it will be important if you need to contact Microsoft. Click **OK**.

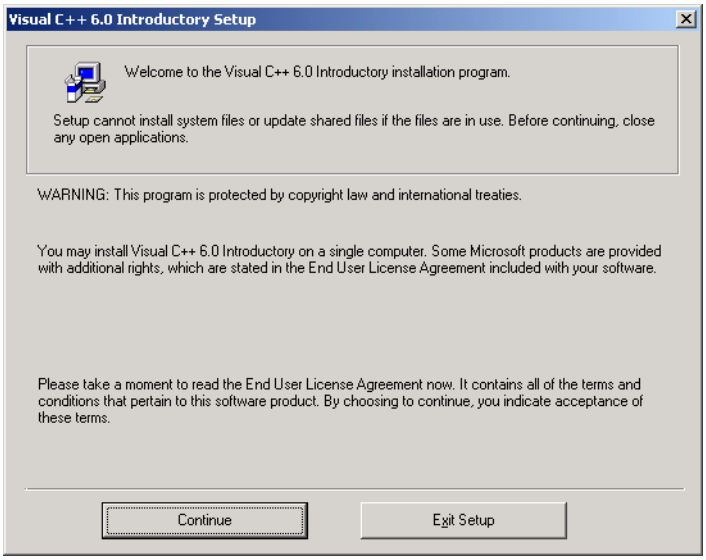

Fig. 1.5 **Setup** screen

8. Choose the installation type and folder to install Visual C++ 6.0 Introductory Edition. Choosing **Typical** installation will install all the basic tools needed to run Visual C++ 6.0. Choosing **Custom** installation allows you to specify which components are to be installed. Accept the default installation directory and select the **Typical** installation button.

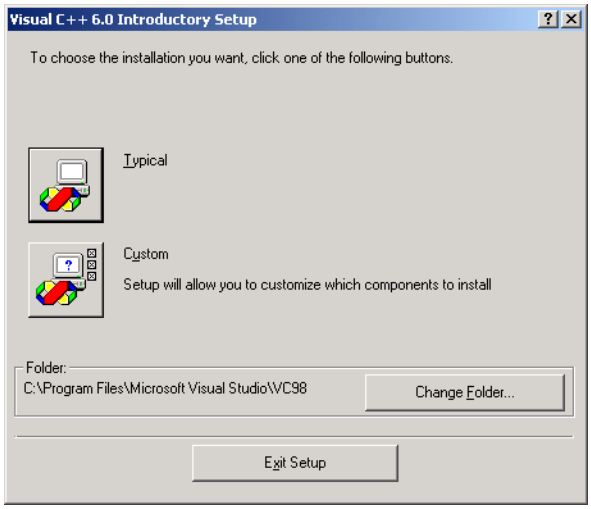

Fig. 1.6 Choosing installation type

9. The **Select Environment Variables** screen allow the option to register enrivornment variables so Visual C++ can be used from the command line. Decide if you want to enable this, then click **OK** to continue with the installation.

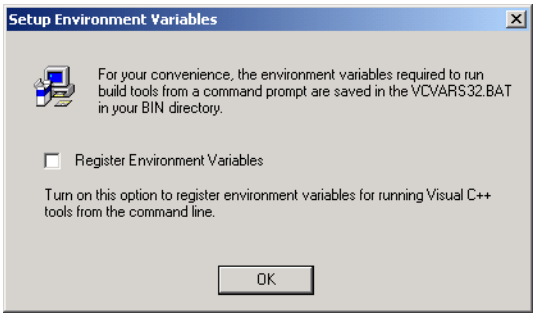

Fig. 1.7 **Setting Environment Variables**

10. The **Windows NT Debug Symbols** screen will notify the user to install a subset of system symbols for the debugger to function properly. Click **OK** to acknowledge this and click **OK** again to complete the **Visual C++ 6.0 Introductory Setup**.
11. You will now be prompted to install the MSDN Library. Click **Next >** to proceed. Follow the same procedures (starting at step 7) to complete the installation.

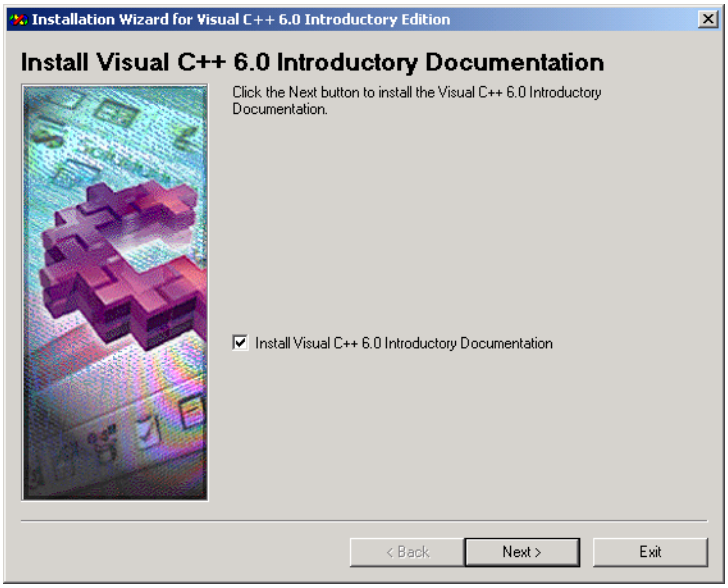

Fig. 1.8 Install documentation

12. Click **Finish** to complete the installation.

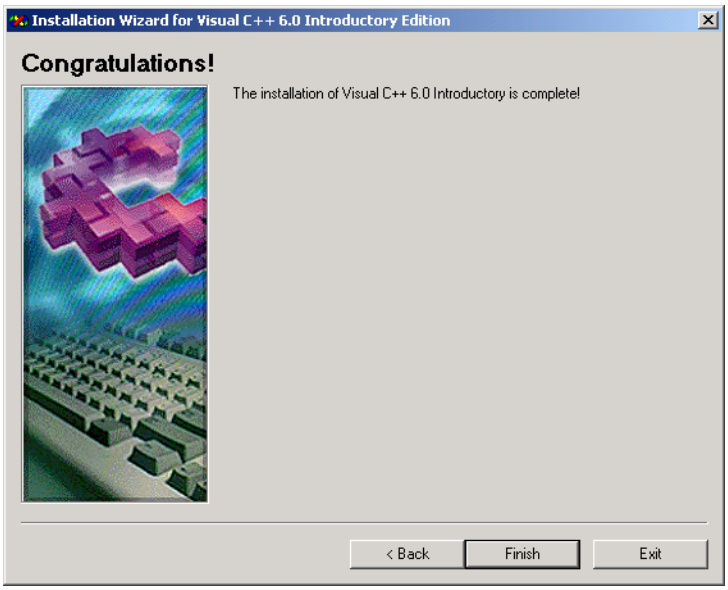

Fig. 1.9 Installation is complete

1.3 Integrated Development Environment Overview: Visual C++

Figure 1.10 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.

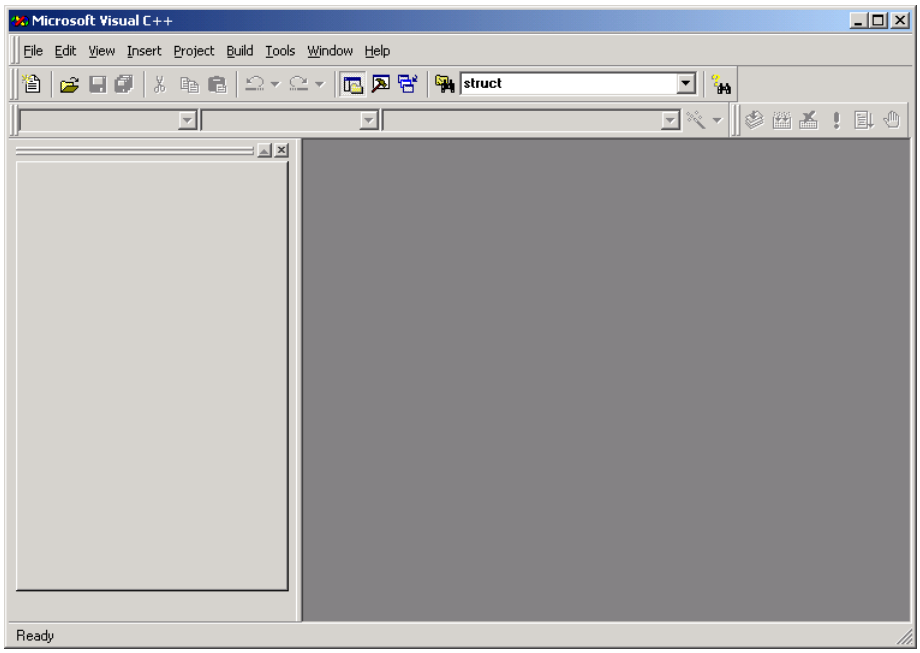

Fig. 1.10 Visual C++ IDE

1.4 Online Visual C++ Documentation

Visual C++ 6.0 uses the *Microsoft Developer Network (MSDN™)* documentation, which is accessible by selecting **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 *Visual Studio®*. Selecting **Contents** displays the ***MSDN Library Visual C++*** dialog (Fig. 1.11). The online 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
```

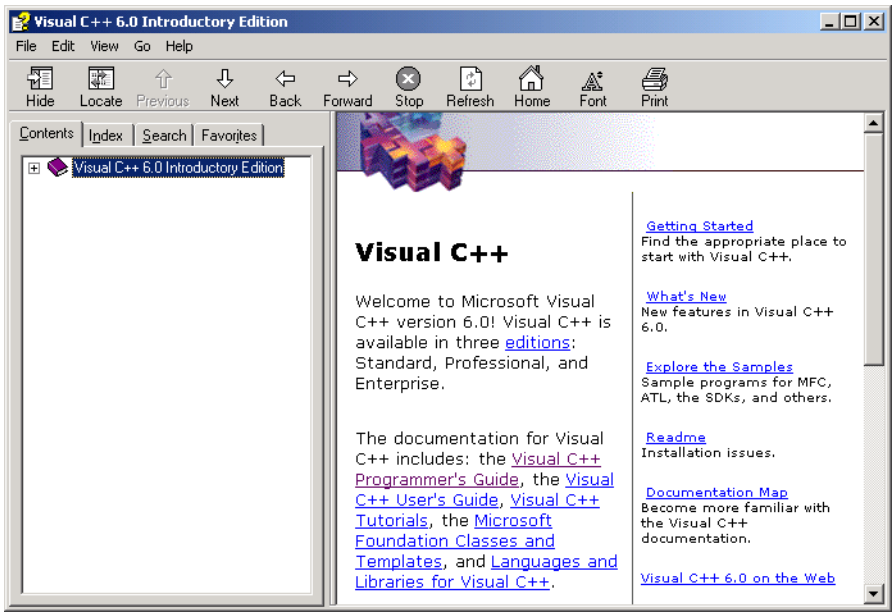

Fig. 1.11 Microsoft Developer Network on-line documentation.

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

```
Microsoft Developer Network Library
   Visual Studio 6.0 Documentation
      Visual C++ Documentation
```

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. 1.12). 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 toolbar (the row of icons near the top of the window in Fig. 1.12) 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 **Visual C++** page (Fig. 1.11).

In the left panel, the user can control the display in the right panel by selecting the ***Active Subset*** of the **MSDN Library** to use and selecting a tab for viewing the ***Contents***, ***Index***, ***Search*** or ***Favorites***. The ***Contents tab*** displays the table 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*** lets the user save links to interesting topics and later return to them.

Online information is divided into categories. Each category is preceded by a book icon. The ***Visual C++ Start Page*** is the starting point for navigating the online documentation. The ***Visual C++ Documentation Map*** outlines the various sections of the Visual C++ documentation by category. The ***What's New in Visual C++ 6.0*** topic explains the newest features introduced in Visual C++ 6.0.

Getting Started with Visual C++ 6.0 contains links to various topics in the documentation, including **Beginning your Program**, **Porting and Upgrading**, **Visual C++ Tutorials**, **What's New for Visual C++ 6.0**, **Getting Help** and **Visual C++ Home Page**. 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.

The ***Using Visual C++*** category is composed of four subcategories—***Visual C++ Tutorials,*** which contains tutorials on how to develop applications that use advanced language features (e.g., OLE server, OLE containers, ActiveX controls, etc.); ***Visual C++ Programmers Guide,*** which contains information on various programming topics (e.g., portability issues, debugging, errors, etc.); ***Visual C++ User's Guide,*** which contains information about the Visual C++ IDE (e.g., projects, classes, editors, utilities, etc.); and a ***Glossary,*** which contains acknowledgments and terminology.

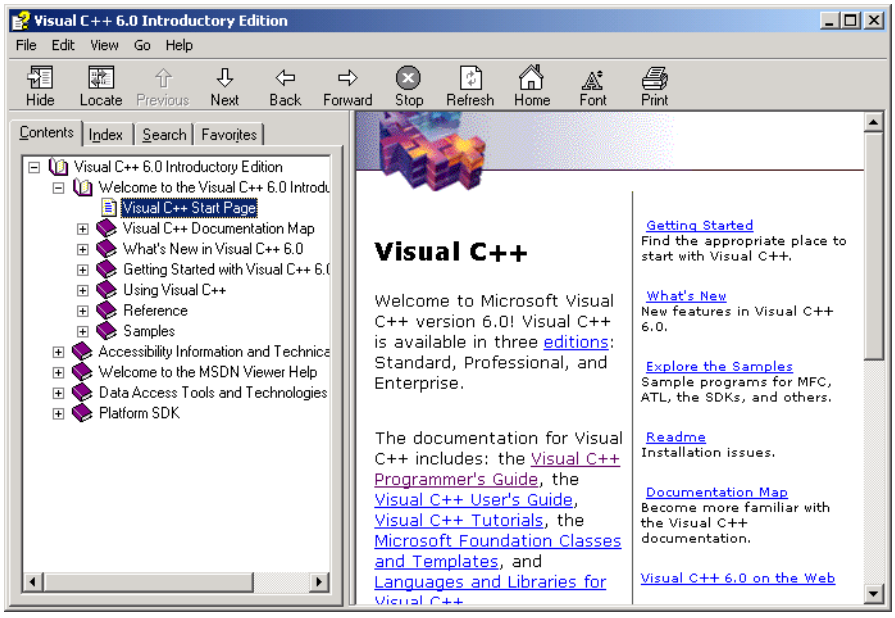

Fig. 1.12 Expanding the Visual C++ topics.

The ***Reference*** category contains two subcategories—***Microsoft Foundation Class Library and Templates*** and ***C/C++ Language and C++ Libraries***. The

Microsoft Foundation Class Library and Templates 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 sample programs, technical notes, a hierarchy chart (showing the relationships among the MFC classes) and a class library reference. The ***C/C++ Language and C++ Libraries*** category contains information about the *Standard Template Library* (*STL*) (a set of reusable C++ template components that make programming easier), the C++ language, the C language and the standard C++ libraries. 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 C++ program. In this guide, we do not create Windows applications that use graphical user interfaces (GUIs) with menus, buttons, etc. Rather, we create *Win32 console applications*. When executed, Win32 console applications get input from and display data to a *console window* (a text-only display that predates Windows). 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. Clicking the ***File*** *menu*'s ***New...*** *menu item* displays the ***New*** *dialog* of Fig. 1.13. The **New** dialog lists the available Visual C++ project types. Note that your **New** 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 book have one project per workspace.

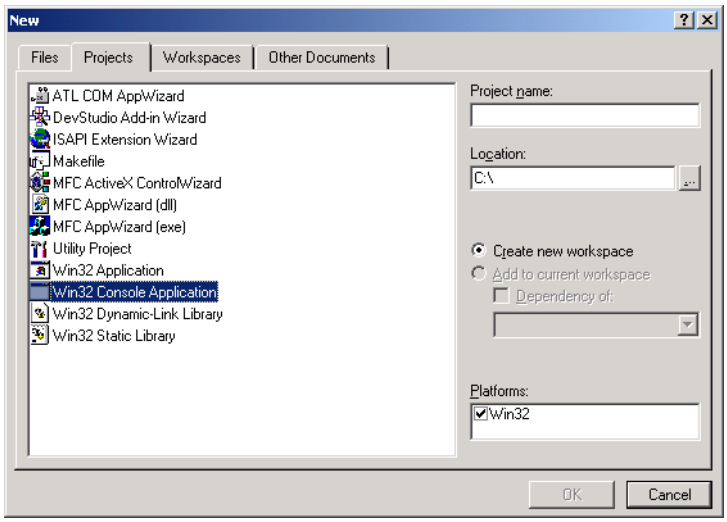

Fig. 1.13 **New** dialog displaying a list of available project types.

Starting with the **New** 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, select **Win32 Console Application**. The ***Project name*** field (in the upper-right corner of the dialog) is where you specify the name of the project. Click in the **Project 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 click in the **Location** field and scroll through it using the right arrow key, you will notice that the project name you typed (**Welcome**) is at the end of the directory path. If you do not modify this directory path, Visual C++ stores your projects in this directory. Pressing **OK** closes the **New** dialog and displays the **Win32 Console Application - Step 1 of 1** dialog (Fig. 1.14).

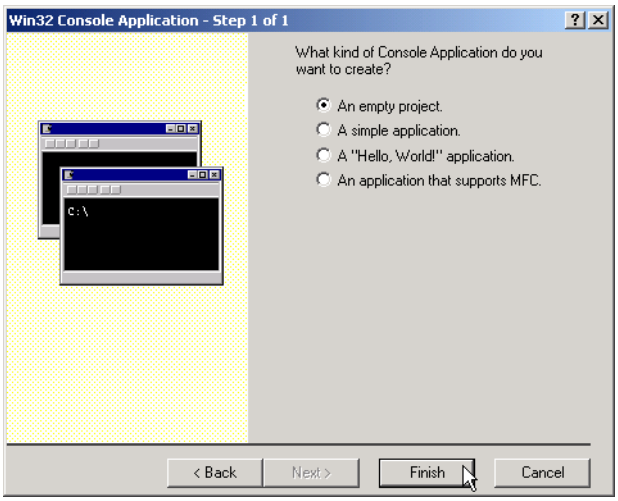

Fig. 1.14 **Win32 Console Application - Step 1 or 1** dialog.

The **Win32 Console Application - Step 1 of 1** dialog displays four choices. Selecting **An empty project** creates a project that does not contain any files. The programmer must add source code files to the project. **A simple application** creates a **.cpp** file (i.e., a file—called a *C++ source file* into which the programmer writes C++ code) and a few support files. Selecting **A "Hello, World" application** creates a **.cpp** file (which contains the code to print **Hello World**) and a few support files. Selecting **An application that supports MFC** creates several files which add support for Visual C++'s Windows programming library called *MFC*. At this point, you should select **An empty project** and click **Finish** to display the ***New Project Information*** dialog (Fig. 1.15). If you selected the wrong project type (i.e., a type other than **Win32 Console Application**) in the **New** dialog and do not see the choices in Fig. 1.14, click **<Back** to view the **New** dialog.

The **New Project Information** dialog provides a summary of the project about to be created. For our **Win32 Console Application**, the dialog specifies that the project is empty (i.e., no additional files were created). This dialog also specifies in the lower-left corner the location on disk for this project. Clicking **OK** closes the dialog and creates the project. If you created the wrong project type, click **Cancel** to go back to the **Win32 Console Application - Step 1 of 1** dialog.

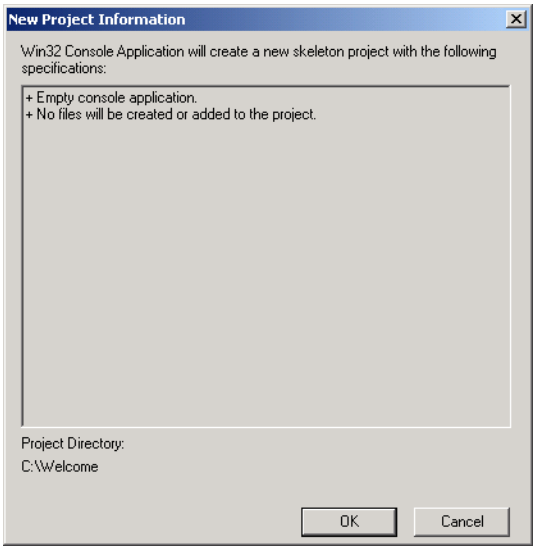

Fig. 1.15 New Project Information dialog.

Figure 1.16 shows the Visual C++ IDE after creating an empty **Win32 Console Application**. The IDE displays the project name (i.e., **Welcome**) in the title bar, and shows the *workspace pane* and the *output pane*. If the output pane is not visible, select **Output** from the **View** menu to display the output pane. The output pane displays various information, such as the status of your compilation and compiler error messages when they occur.

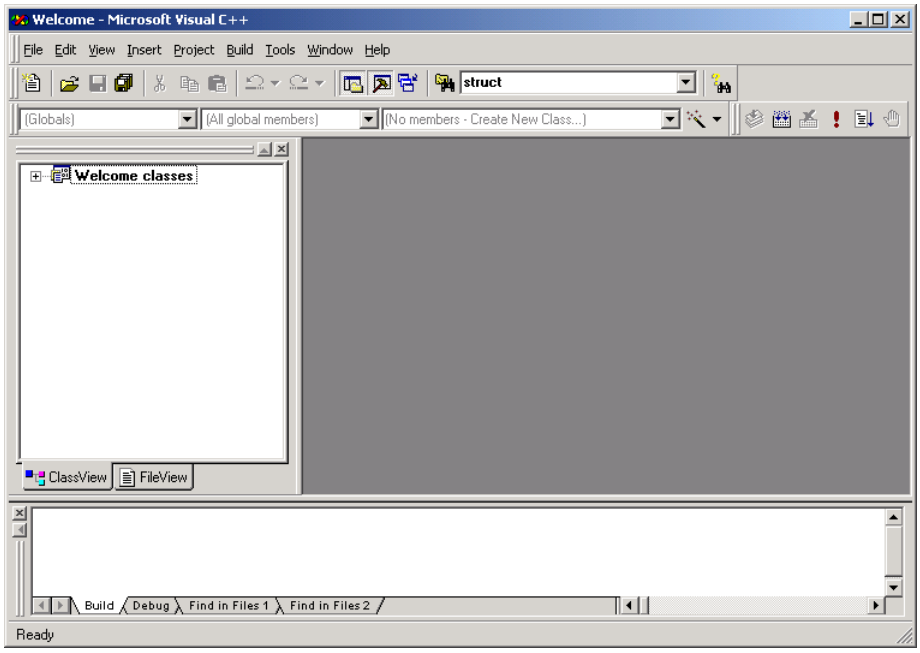

Fig. 1.16 Visual C++ IDE displaying an empty project.

At the bottom of the workspace pane are two tabs, ***ClassView*** and ***FileView***. Clicking **ClassView** displays classes and 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. **FileView** displays the names of the files that make up the project. **FileView** initially displays the project name followed by the word **files** (e.g., **Welcome files**). Clicking the plus sign, **+,** to the left of **Welcome files** 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.

The next step is to add a C++ file to the project. Selecting **New...** from the **File** menu displays the **New** dialog (Fig. 1.17). When a project is already open, the **New** dialog displays the **Files** tab containing a list of file types. The file types will vary based on the Microsoft development tools installed on your system.

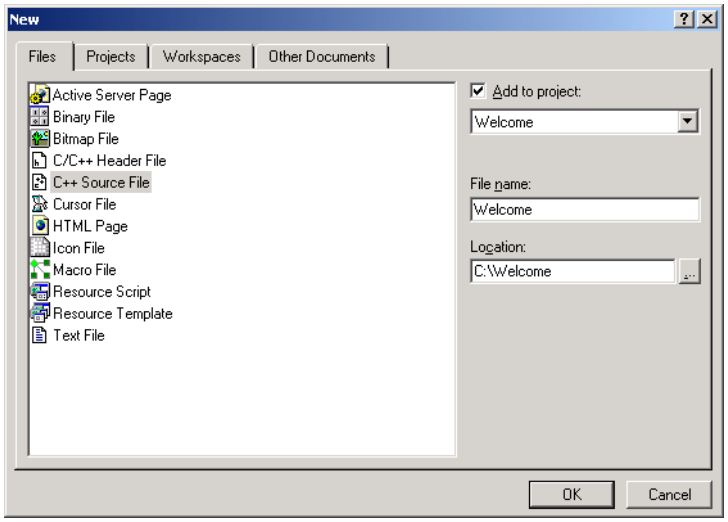

Fig. 1.17 **New** dialog displaying the **Files** tab.

Select ***C++ Source File*** for a C++ file. The ***File name*** field is where you specify the name of the C++ file. Enter **Welcome** in the ***File 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. When checked, **Add to Project** adds the file to the project. If **Add to Project** is not checked, check it. Click **OK** 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 **C:\Welcome** (the combination of the **Location** field and the project name). Figure 1.18, shows the IDE after adding **Welcome.cpp** to the project. In Figure 1.18 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.

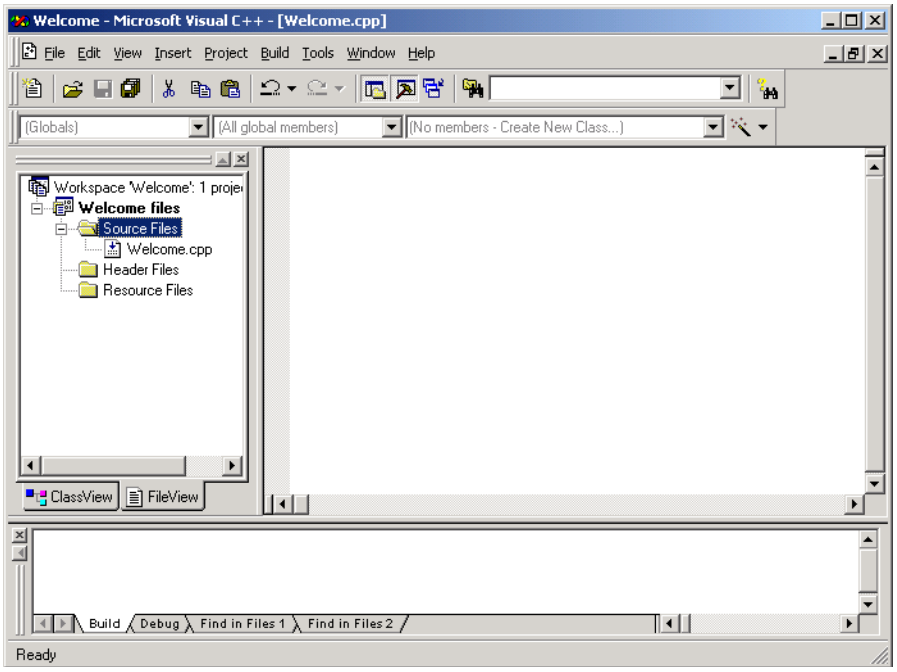

Fig. 1.18 A newly created **Welcome** project

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. [*Note:* The code examples for this book 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
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    cout << "Welcome to C++!" << endl;
11
12    return 0;   // indicates successful termination
13
14 } // end main
```

Fig. 1.19 Code for **Welcome.cpp**

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.*

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 1.20 demonstrate the use of *IntelliSense*. When function **`srand`**'s opening parenthesis is typed, Visual C++ automatically displays its function header as a tip.

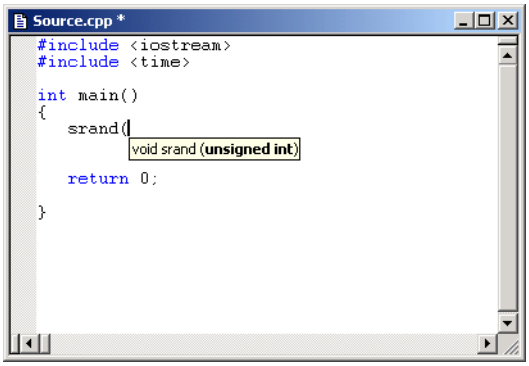

Fig. 1.20 Demonstrating intelliSense.

Testing and Debugging Tip 1.3

IntelliSense helps the programmer type a correct program.

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.exe** command or press the *F7 key*. Compiler messages and errors appear in the output pane's **Build** tab. If there are no errors when compilation is complete, the **Output**

pane should appear in the **Build** tab as shown in Fig. 1.21 (this is sometimes called the "happy window").

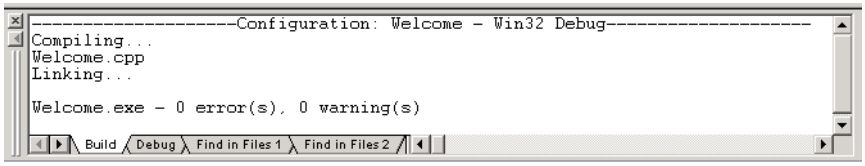

Fig. 1.21 **Output** pane showing a successful build.

If an error message appears in the **Output** pane's **Build** tab, double-clicking anywhere on the error message displays the source file and places a *blue 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. 1.22. The error in this particular case is that the ending quote around the string "Welcome to C++!" is missing (an illegal change).

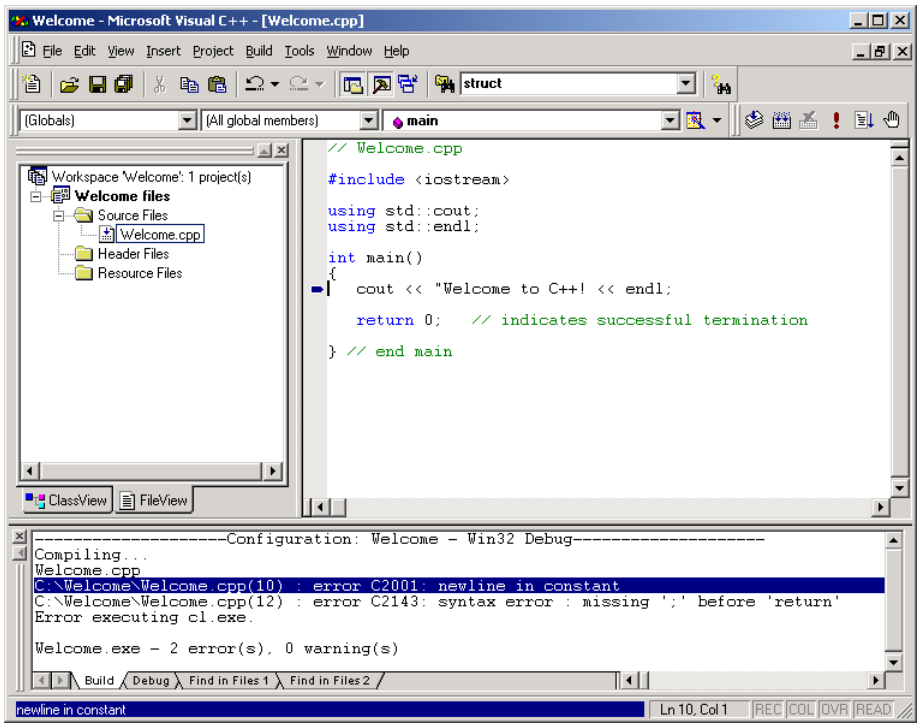

Fig. 1.22 Blue marker indicating that a line contains a syntax error.

Error messages are often longer than the output pane's width. The complete error message can be viewed either by using the horizontal scrollbar to the right of the tabs at the bottom of the screen (e.g. **Build**, **Debug**) or by reading the *status pane*. 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.

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 the compiler to report 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 **Execute Welcome.exe** in the **Build** menu. The *Visual C++ 6 Introductory Edition* displays the message as in Fig. 1.23.

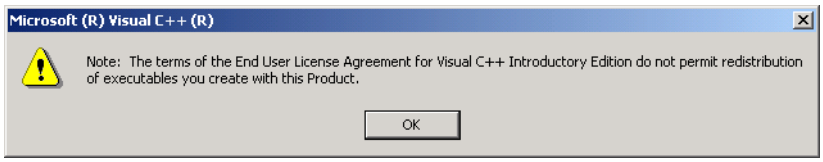

Fig. 1.23 Notice produced by Visual C++ 6 Introductory Edition.

The program is executed in a console window as shown in Fig. 1.24. [*Note*: The console window is referred to as the **Command Prompt** for Windows 2000, if you are using a different version of Windows then the console might be referred to as the **MS-DOS Prompt**.] Pressing any key closes the console window.

Fig. 1.24 Output for `Welcome.cpp`

To create another application, follow the same steps outlined in this section using a different project name and directory. Before starting a new project you should close the current project by selecting the **File** menu's ***Close Workspace*** option. 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 Workspaces** option to select a recent workspace or you can select **Open...** to see an **Open Workspace** 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 Visual C++ 6.

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 1.25–Figure 1.27.

```
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. 1.25 **Time** class header file.

```
1  // Fig. 6.6: time1.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4
5  using std::cout;
6
```

Fig. 1.26 **Time** class definitions.

```
 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.
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. 1.26 **Time** class 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;
```

Fig. 1.27 Main entry point for application

```
 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
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. 1.27 Main entry point for application

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. To create a **.h** header file, follow the same steps for creating a **.cpp** source file, but select **C/C++ Header File** under the **Files** tab, shown in Figure 1.28.

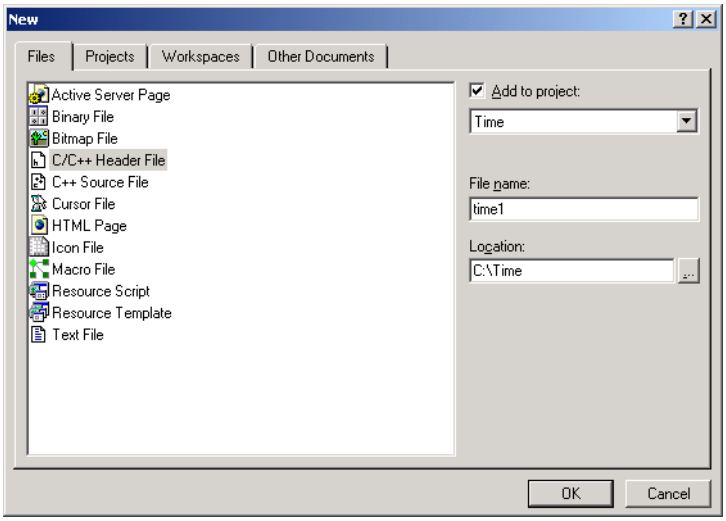

Fig. 1.28 Creating a header file.

4. Next, use the method described in Section 1.5 to compile and run the program. Figure 1.29 shows the output for the **`Time`** class example.

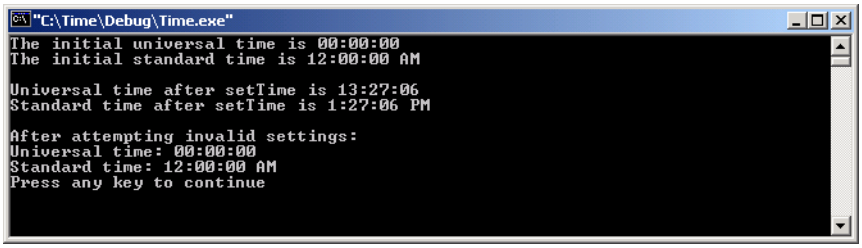

Fig. 1.29 Output for **`Time`** class example.

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 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 exe-

cuting correctly. A breakpoint can be set in two ways. You can set a breakpoint by clicking the line in the program where the breakpoint is to be placed and clicking the ***Insert/ Remove Breakpoint*** button in the **Build MiniBar** or by pressing the *F9 key*. The **Insert/Remove Breakpoint** button is grayed (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 (Fig. 1.30). Breakpoints are removed using the same method as adding a breakpoint.

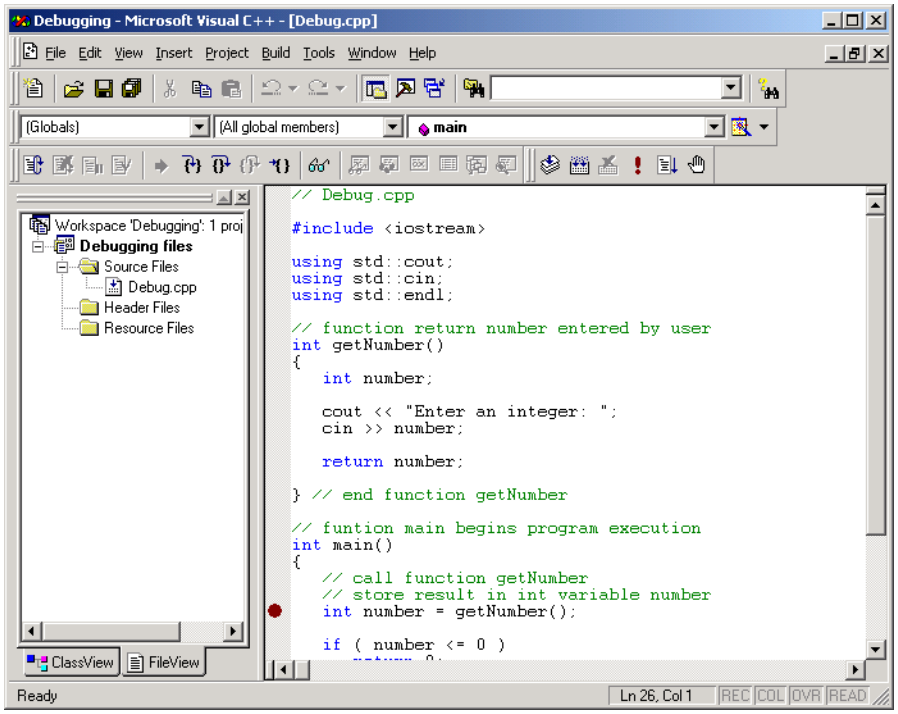

Fig. 1.30 Setting a breakpoint.

Selecting the ***Go*** *command* (in the **Build** menu's ***Start Debug*** *submenu*) starts the debugging process. Because we have chosen to debug a console application, the console window (i.e., command prompt) that contains our application appears. All program interaction (input and output) is performed in this window. Program execution suspends for input and at breakpoints. You may need to manually switch between the IDE and the console window to perform input. To switch between windows you can use *Alt + Tab* or click your program's panel on the Windows taskbar at the bottom of the screen.

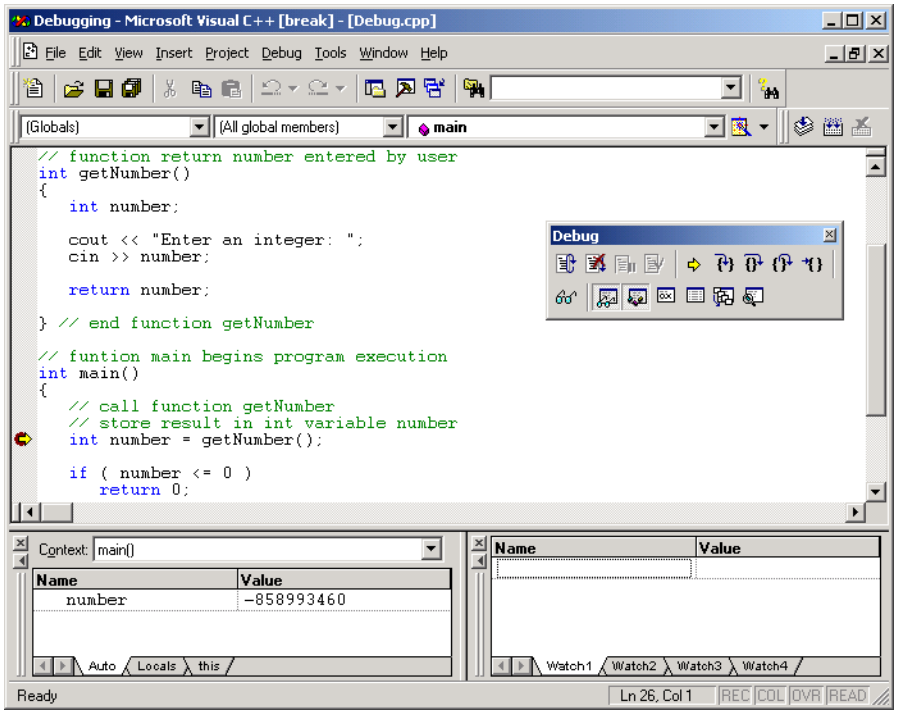

Fig. 1.31 Execution suspended at a breakpoint.

The *yellow arrow* to the left of a statement indicates that execution is suspended at this line. This statement will be the next statement executed (Fig. 1.31). Note in the IDE that the **Build** menu is replaced with the **Debug** menu and that the title bar displays **[break]** to indicate that the IDE is in *debug mode*.

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.*

The bottom portion of the IDE is divided into two windows—the ***Variables*** window (i.e., the left window) and the ***Watch*** window (i.e., the right window). The **Variables** window contains a list of the program's variables. Note that different variables can be viewed at different times, by clicking either the ***Auto***, ***Locals*** or ***this*** tabs. The **Auto** tab displays the name and value of the variables or objects (discussed in Chapter 6 of *C++ How To Program, Fourth Edition*) used in both the previous statement and the current statement. The **Locals** tab displays the name and current value for all the local variables or objects in the current function's scope. The **this** tab displays data for the object to which the executing function belongs.

The variable values listed in the **Variables** window 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** window allows the user to watch variables as their values change. Changes are displayed in the **Watch** window.

Variables can be typed directly into the **Watch** window or dragged with the mouse from either the **Variables** window or the source code window and dropped into the **Watch** window. A variable can be deleted from the **Watch** window by selecting the variable name and pressing the *Delete* key. The four tabs at the bottom of the **Watch** window are used by the programmer to group variables.

Like the **Variables** window, variable values can be modified in the **Watch** window by editing the **Value** field. Changed values are colored red. The current value of a variable during the process of debugging can also be viewed by resting the mouse cursor over the name of that variable in the source code window (Fig. 1.32).

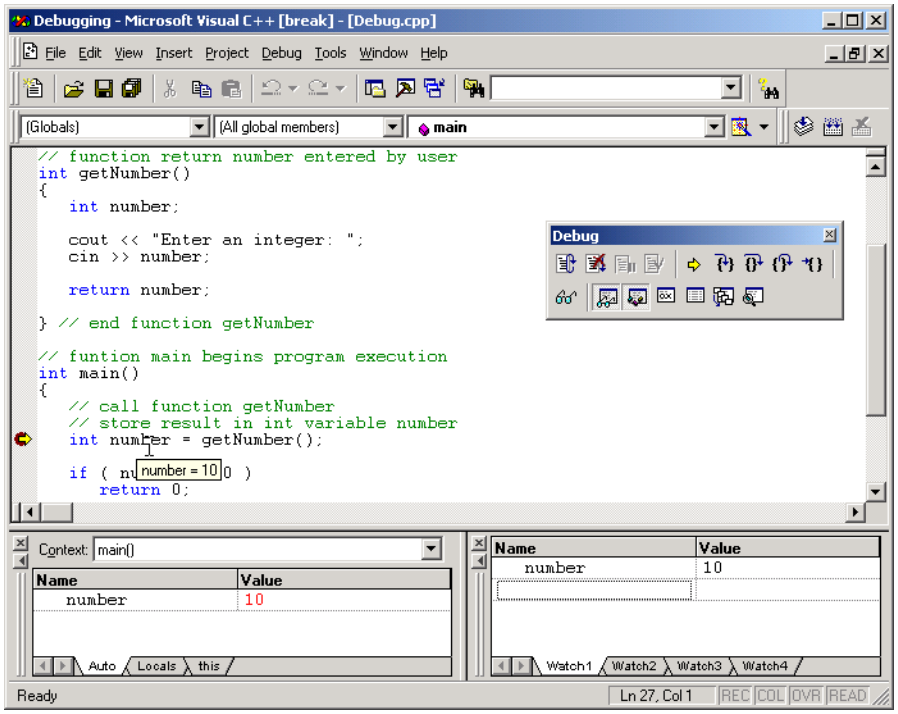

Fig. 1.32 The **Variables** and **Watch** windows

The **Debug** toolbar 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.

Break Execution suspends program execution at the current location. ***Apply Code Changes*** allows the programmer to modify the source code while in debug mode. [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.

Fig. 1.33 The **Debug** toolbar.

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 1.7

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 one 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 In** to a function that you do not need to examine, click **Step Out** to return to the caller.

Click the mouse on a line of code after a number of lines of code you do not wish to step through, then click the ***Run to Cursor*** button to execute all code up to the line where the cursor is positioned. This technique is useful for executing loops or functions without having to enter the loop or function.

Testing and Debugging Tip 1.8

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

Testing and Debugging Tip 1.9

*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. 1.34), which is useful for monitoring expression values and variable values. The **QuickWatch** dialog pro-

vides 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 **Variables** window and **Watch** 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*.

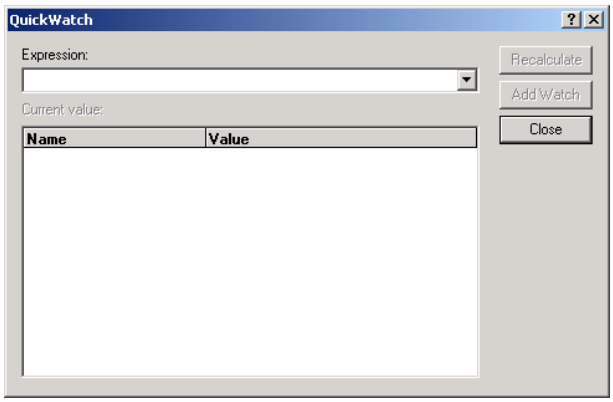

Fig. 1.34 The **QuickWatch** dialog.

To maintain a longer watch, click the ***Add Watch*** button to add the variable to the **Watch** window. 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., **x = 20**, etc.) by typing the expression into the **Expression** field.

The ***Watch*** button displays the **Watch** window. The ***Variables*** button displays the **Variables** window.

The ***Memory*** button displays the **Memory** window and the ***Registers*** button displays the **Registers** window (these buttons are beyond the scope of this book).

The ***Call Stack*** button displays a window containing the program's *function call stack*. A function call stack is a list of the functions that were called to get 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 ***Disassembly*** button displays the **Disassembly** window. Analyzing a program that has been disassembled is a complex process likely to be used by only the most advanced programmers. We do not discuss the **Disassembly** window in this book.

Each of the buttons (e.g., **Variables**, **Watch**, **Memory**, **Registers**, **Call Stack** and **Disassembly**) act as *toggle buttons*—clicking them hides the window (if it is visible) or displays the window (if it is invisible). Debug windows (e.g., **Variables**, etc.) can also be displayed during the debugging session by selecting the appropriate debug window from the **View** menu's submenu **Debug Windows**.

When a project is closed and reopened, any breakpoints set during a previous debugging session are still set. Breakpoints are persistent. You can gather information about breakpoints by selecting the **Edit** menu's **Breakpoints…** menu item. When selected, the **Breakpoints…** menu item displays the **Breakpoints** dialog (Fig. 1.35).

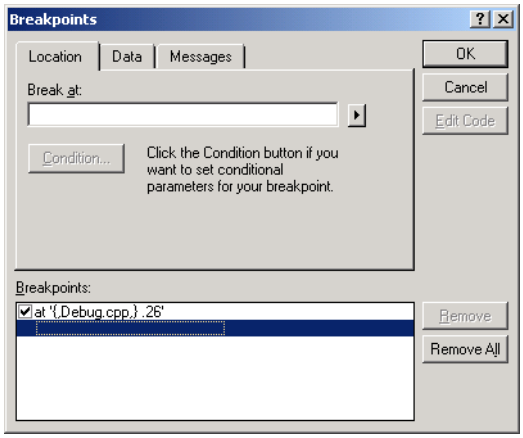

Fig. 1.35 The **Breakpoint** dialog box.

The **Breakpoints** 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 **Break at** field. Note that when entering the line number in the **Break at** field, precede the line number with a period. For example, to set a breakpoint at line 13 we enter **.13** in the **Break at** field.

Visual C++ also allows breakpoints to be enabled when certain conditions are true. The programmer specifies the line number in the **Break at** field and presses the **Condition...** button to display the **Breakpoint Condition** dialog. A condition is specified in the **Enter the expression to be evaluated** field and **OK** is pressed to set the condition.

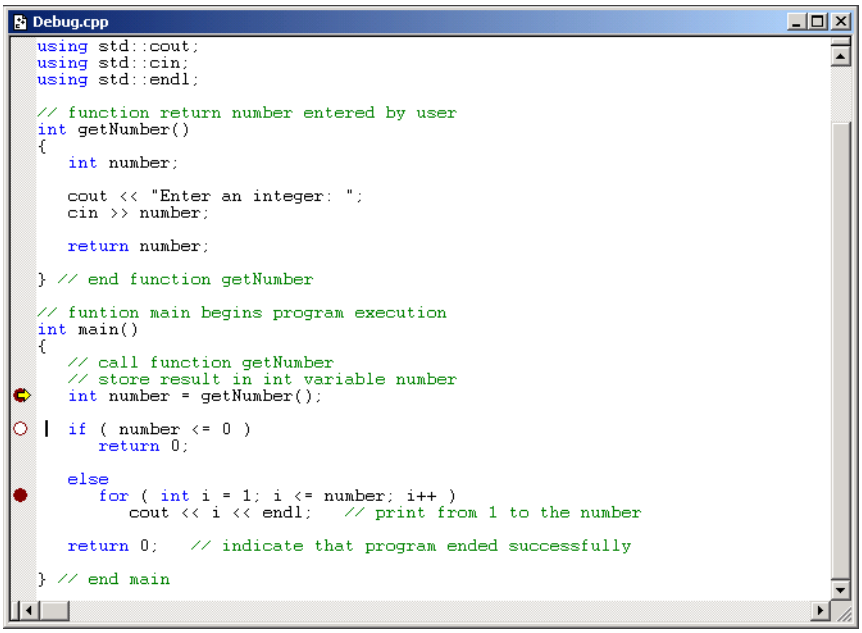

Fig. 1.36 Disabled breakpoint.

Figure 1.36 shows the debugging environment with a disabled breakpoint. Notice that the disabled breakpoint is still visible but it appears as a white circle. To make the breakpoint active, click the empty checkbox next to the breakpoint in the **Breakpoints** dialog.

Testing and Debugging Tip 1.10

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.

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** from the **Debug** menu.*

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

1.7.1 Debugging an Application

This section guides the programmer through the debugging process for a simple C++ application, **`Debug.cpp`** (Fig. 1.37). This application obtains a number from the user and counts from 1 to that number.

```
1  // Debug.cpp
2
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // function return number entered by user
10 int getNumber()
11 {
12    int number;
13
14    cout << "Enter an integer: ";
15    cin >> number;
16
17    return number;
18
19 } // end function getNumber
20
21 // funtion main begins program execution
22 int main()
23 {
24    // call function getNumber
25    // store result in int variable number
26    int number = getNumber();
27
28    if ( number <= 0 )
29       return 0;
30
31    else
32       for ( int i = 1; i <= number; i++ )
33          cout << i << endl;   // print from 1 to the number
34
35    return 0;   // indicate that program ended successfully
36
37 } // end main
```

Fig. 1.37 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, add a breakpoint to line 26 by clicking the line in the program and pressing the *F9 key*. 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, 32 and 35. When complete the window should appear as shown (Fig. 1.38).

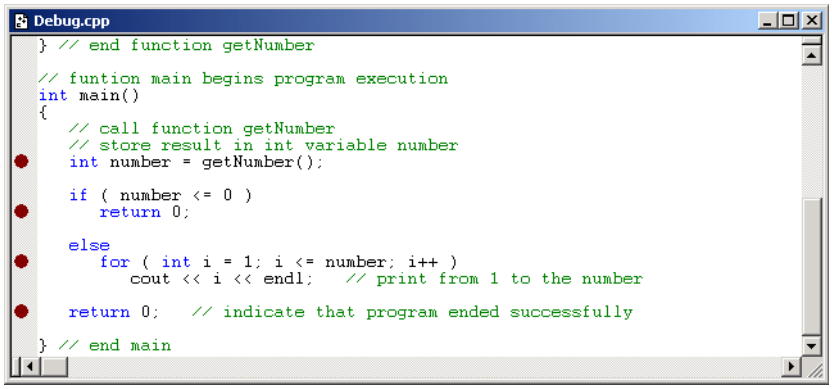

Fig. 1.38 Breakpoints set in a program.

4. Click **Go** (in the **Start Debug** submenu from the **Build** menu) to start the debugger. Because we have chosen to debug a console application, the console window (i.e., command prompt) appears. Program execution suspends for input and at breakpoints. The program suspends execution at line 26. The *yellow arrow* to the left of the statement

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

indicates that execution is suspended at this line.

5. To add a watch, type the **`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 **Go** again. At this point, the program is executing and the console window is displayed. Enter 10 into the console 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`**. **`number`** is used to store the number entered by the user, therefore it always maintains its original value. Figure 1.39 shows watches set for **`number`** and **`i`**.

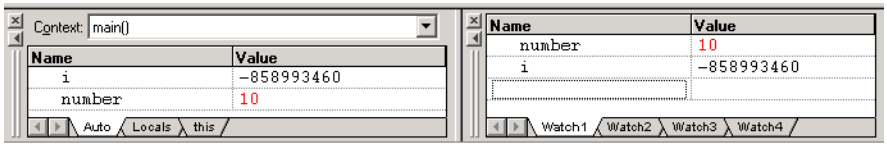

Fig. 1.39 Watches set for **`number`** and **`i`**.

7. Clicking **Step Over** would execute the individual steps of the loop each time the loop is executed. Clicking **Step Out** or **Go** would perform the whole loop in one step.
8. Once the `for` loop has completed, program execution suspends at line 35. The main window indicates that the debugger has completed and the count from 1 to 10 is displayed in the output window (Fig. 1.40). 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 35 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 into the console window and observe how the debugger operates.

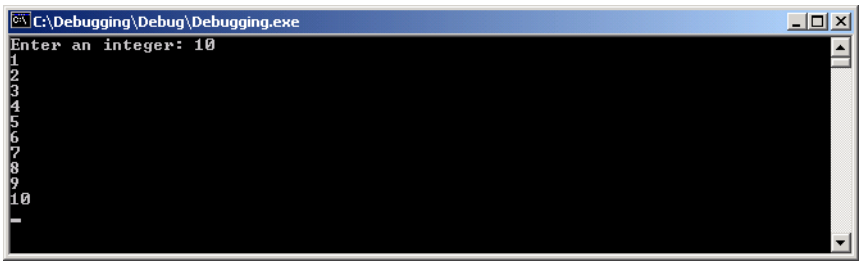

Fig. 1.40 Output for `Debug.cpp`

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