CHAPTER 1

Hello, World

Consistency is the last refuge of the unimaginative.

—Oscar Wilde

Since time immemorial, which pretty much dates back to the release of the Kernighan and Richie book on C, there has been a tradition of opening a book on C or its descendants with a short example of how easy it is to display "Hello World". This book is no exception. Let's examine the C# and C++ versions of "Hello World" side by side (see Table 1-1).

Table 1-1. *"Hello World" in C# and C++*

C#	C++
`using System;`	`using namespace System;`
`class HelloWorld`	
`{`	
`    static void Main()`	`void main()`
`    {`	`{`
`        Console.WriteLine("Hello World");`	`    Console::WriteLine("Hello World");`
`    }`	`}`
`}`	

As you can see in Table 1-1, the languages are clearly different. On the other hand, C# and C++ are like French and Italian; although the C++ syntax may appear foreign, the meaning is clear.

Here are some things to notice:

- In C#, `Main()` is always a method of a class. In C++/CLI (Common Language Infrastructure), `main()` is not a class method; it is a global function. It's easy—just remember that global functions have no class.
- In the same way that you have a unique static member function named `Main()` in any C# program, you have a unique global function named `main()` in any C++ program. It is possible to get around this requirement in C# and have multiple `Main()` methods by embedding them in different classes. You can then tell the compiler using the `/main:<type>` option which class contains the startup method. This trick does not work in standard C++ since `main()` must be a global function and any versions of `main()` would have the same signature and clash in the global namespace.

- C++ uses `::` (colon-colon) to separate namespaces and class names and a dot (`.`) to access class members; C# uses a dot for everything. C++ expects you to be more specific about what you're doing.
- The C++/CLI `using` statement requires the additional keyword `namespace`.

■ **Note** In Microsoft Visual C++, the entry point can be any function as long as it meets certain restrictions defined in the linker documentation. It can be a global function or a member function. You do this by specifying the /entry:<function_name> linker option. Standard C++ requires a unique global function named main with an integer return value and an optional argument list. See Section 3.61 of the C++ standard, ISO/IEC 14882:2003(E). A PDF version of this standard can be downloaded from `http://webstore.ansi.org` for a small fee.

Starting the Visual Studio 2013 Console

I bet you're just itching to give this a try. "Real programmers" use the command line, so let's start there. We're now going to construct a console application.

Click Start, open the Visual Studio Tools folder as in Figure 1-1, then double-click Developer Command Prompt for VS2013.

Figure 1-1. *Open the Visual Studio Tools folder*

This spawns a new command prompt with the environment variables set to work with Visual Studio 2013. All the Visual Studio compilers may be run from the command line, including Visual C++, Visual C#, and Visual Basic.

Retrieving the Source Files

Either pop up `notepad.exe` (surely your favorite editor) and start typing, or fetch the source from the Source Code section of the Apress website. Go to `www.apress.com`, and search for this book using the ISBN, 978-1-4302-6706-5.

Executing HelloCpp.cpp

Navigate to the sample directory for this Chapter 1, and go to the `HelloWorld` subdirectory. Here is `HelloCpp.cpp`:

```
using namespace System;
void main()
{
    Console::WriteLine("Hello World");
}
```

Enter the following command:

```
cl /nologo /clr HelloCpp.cpp
```

This command directs the C++ compiler to compile this file targeting the Common Language Runtime (CLR) and creates a C++/CLI executable. The executable is a managed assembly that contains metadata and Common Intermediate Language (CIL), just like C# executables. CIL is also known as MSIL on the CLR.

Let's execute this example. First, type

```
HelloCpp
```

Next, press Enter. You should see the following:

```
Hello World
```

and that's a good thing.

A Quick Tour of the Visual C++ IDE

In this section, we go over the steps for making an elementary C++/CLI project using the Visual Studio 2013 C++ Integrated Development Environment (IDE). This is very similar to creating a C# project.

1. Load Visual Studio 2013.
2. From the File menu, select New Project. My system is set up with Visual C++ as the default language, so my New Project dialog box looks like the one shown in Figure 1-2.

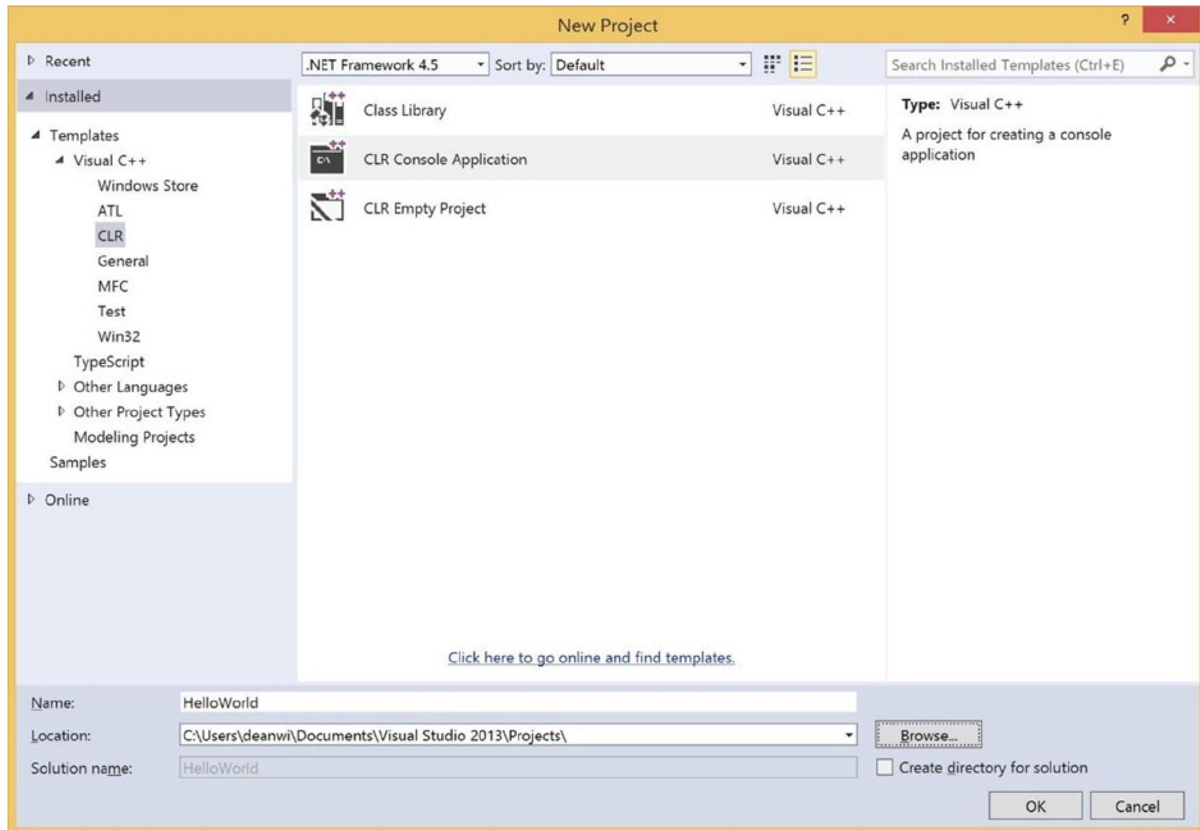


Figure 1-2. *Creating a new HelloWorld project and solution*

3. Navigate to the CLR project types under Visual C++.
4. Select CLR Console Application.
5. Enter **HelloWorld** in the Name text box.
6. Click OK.

By default, Visual Studio 2013 creates new projects in C:\Users\%USERNAME%\Documents\Visual Studio 2013\Projects. Feel free to change the directory and place the project elsewhere if you like. Click OK.

Understanding Projects and Solutions

The Visual C++ CLR Console Application Wizard creates a new project called HelloWorld in a solution also called HelloWorld. What is the difference between the project and the solution?

The basic paradigm used in Visual Studio is that you create a solution, which is the container for what you are working on. A solution can consist of several projects, which can be class libraries or executables. Each project is language specific, though it is also possible to mix languages within a single project using custom build rules.

In our case, we want a single Visual C++ project that will generate a single executable named `HelloWorld.exe`, so our solution has a single project. By default, the project is created in a subdirectory, but we can change this behavior by deselecting Create directory for solution. Later in this book, we'll have more sophisticated solutions that depend on several projects.

Now you should see two tiled windows: the Solution Explorer and the editor window containing `HelloWorld.cpp`. It appears that Visual C++ 2013 has gone to all the trouble of writing the program for us–now isn't that nice?

Understanding the Differences

There are a few differences between our basic HelloCpp application and the HelloWorld application created by the Visual Studio C++ CLR Console Application Wizard, shown in Figure 1-3. The most obvious difference is that the wizard created several additional supporting files.

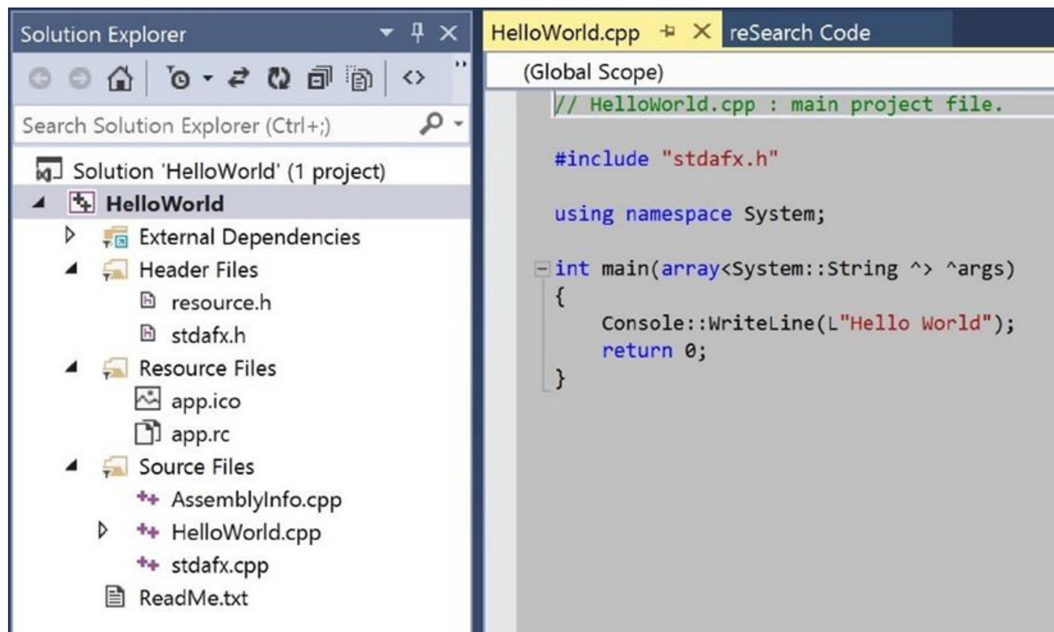


Figure 1-3. *The HelloWorld application as created by the CLR Console Application Wizard*

Let's have a look at those new files.

Resources

These files outfit your application with a snappy little icon and pave the way for future application development. Visual C++ allows you to embed resources in your binary files. They can be bitmaps, icons, strings, and other types. For more information, consult the Visual C++ documentation.

- `resource.h`
- `app.ico`
- `app.rc`

Precompiled Headers

These files improve compilation speed by avoiding multiple compilations of common code:

- `stdafx.h`
- `stdafx.cpp`

One topic that surfaces again and again throughout this book is the distinction between declarations and definitions in C++. Unlike C#, class prototypes, called declarations, may be separated from class definitions into distinct files. This improves compilation speed, avoids circular dependencies, and provides an object-oriented abstraction layer for complex projects. In many C++ projects, it is common that files containing just declarations, called *header files* and terminated with the `.h` extension, are compiled as a unit at the start of every source file. If the headers are identical across the project, the compiler ends up compiling the same chunk of code with each source file. One optimization provided by Visual C++ is to compile the headers referenced in the `stdafx.h` file en masse into a binary PCH (precompiled header) file in advance of all other compilation. This is called *precompiling* the headers. As long as the headers are not modified, subsequent compilations of source files are sped up considerably as the precompiled headers are loaded from disk as a unit rather than being recompiled individually. Two files, `stdafx.h` and `stdafx.cpp`, are generated by Visual C++ to assist in this mechanism. For more information, consult the Visual C++ documentation.

It is possible to disable precompiled headers by changing the project properties. To modify the project settings, right-click the `HelloWorld` project in the Solution Explorer. Navigate to Configuration Properties, and click the triangle to expand the list. Then expand the triangle next to C/C++, and select Precompiled Headers. The Property Pages window, shown in Figure 1-4, appears on the screen, which allows you to configure precompiled headers within your application.

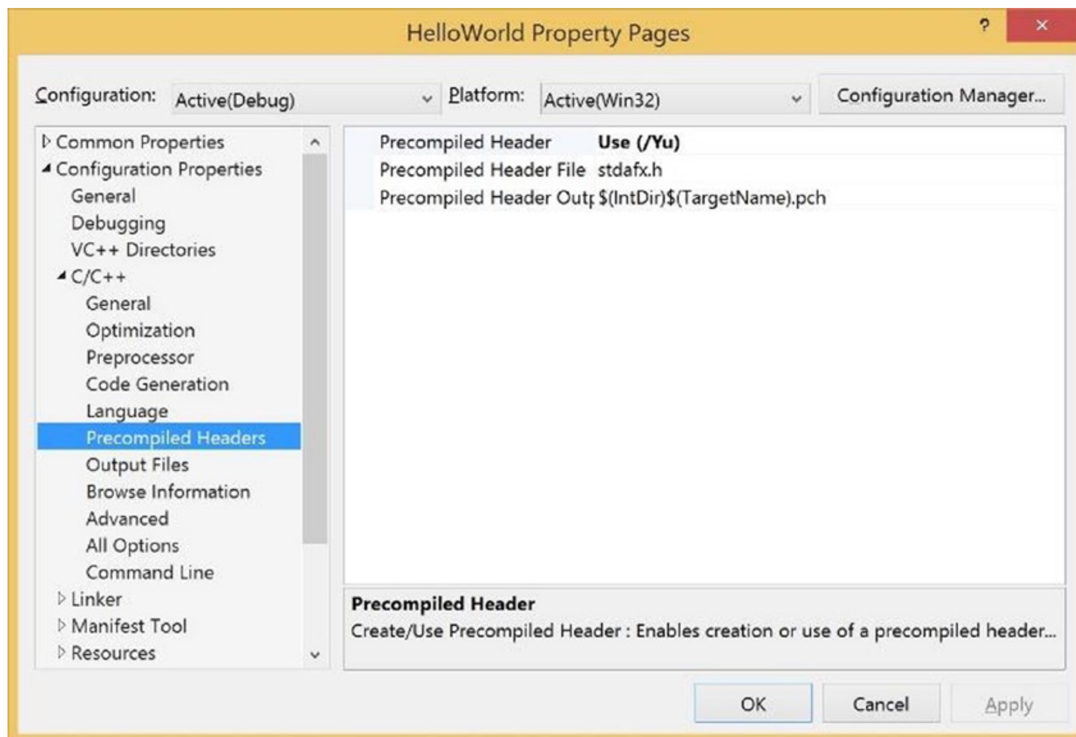


Figure 1-4. *Configuration of precompiled headers from the Property Pages window*

AssemblyInfo.cpp

The file AssemblyInfo.cpp contains all the attribute information for the assembly. This is similar to the C#-produced AssemblyInfo.cs. This includes, but is not limited to, the copyright, version, and basic assembly description information. The default values are fine for development, but you need to fill out some of the information before you ship, including the Copyright Attribute. Figure 1-5 shows an excerpt from a sample AssemblyInfo.cpp.

```
#include "stdafx.h"

using namespace System;
using namespace System::Reflection;
using namespace System::Runtime::CompilerServices;
using namespace System::Runtime::InteropServices;
using namespace System::Security::Permissions;

//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
//
[assembly:AssemblyTitleAttribute(L"HelloWorld")];
[assembly:AssemblyDescriptionAttribute(L"")];
[assembly:AssemblyConfigurationAttribute(L"")];
[assembly:AssemblyCompanyAttribute(L"")];
[assembly:AssemblyProductAttribute(L"HelloWorld")];
[assembly:AssemblyCopyrightAttribute(L"Copyright (c)  2014")];
[assembly:AssemblyTrademarkAttribute(L"")];
[assembly:AssemblyCultureAttribute(L"")];
```

Figure 1-5. *An excerpt from AssemblyInfo.cpp*

HelloWorld.cpp

The main source file has a few significant differences as well, as Figure 1-6 shows:

- The main function is defined to accept a managed array of System::String, which is equivalent to the C# Main(string[] Args). This allows you to access command-line arguments.
- The precompiled header file stdafx.h is included to support the use of precompiled headers.
- The literal string © "Hello World" is prepended with an L to indicate a wide character string. In native C++, strings are byte arrays by default. When compiling C++/CLI, the compiler attempts to distinguish between wide character strings and byte arrays by context. Whether or not you have an L in this context, a wide character System::String is created.

```
HelloWorld.cpp    Welcome to WinIDE    AssemblyInfo.cpp
(Global Scope)
// HelloWorld.cpp : main project file.

#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hello World");
    return 0;
}
```

Figure 1-6. *HelloWorld.cpp*

Window Layout

One of the well-designed features of Visual Studio is the ability to customize the appearance of the IDE by rearranging windows using simple mouse movements. In this section, we learn how to dock and position windows.

Docking the Window

The Solution Explorer naturally appears on the left or right of Visual Studio, depending on which settings are chosen by default. Luckily, custom rearrangement is easy and intuitive. Right-click on the title bar, and a pop-up window appears as in Figure 1-7, which allows you to dock the window, dock as a tabbed document, or float on top.

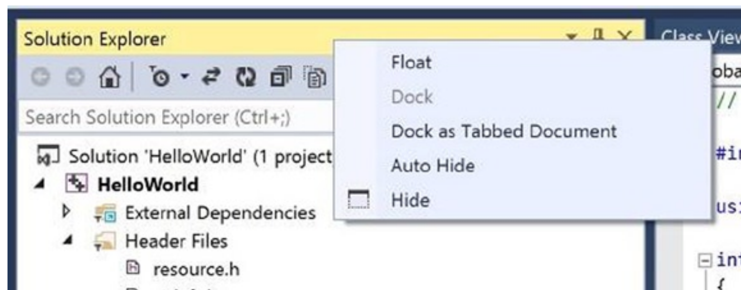
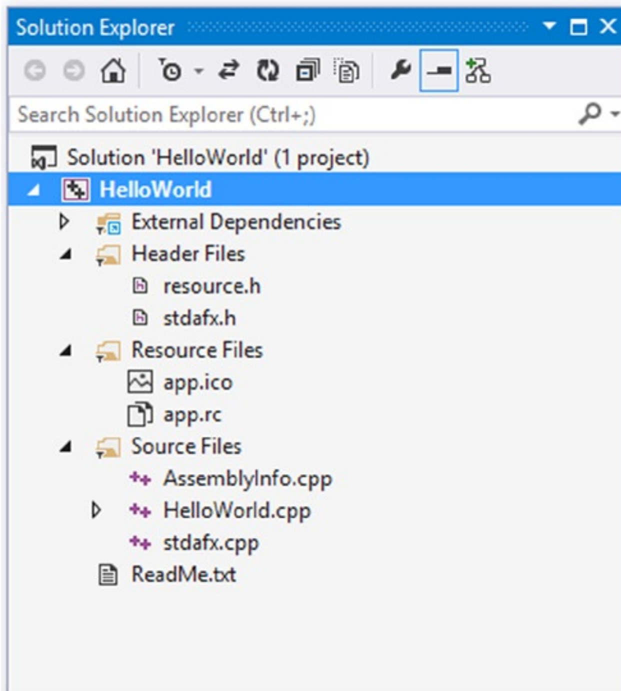


Figure 1-7. *Right-clicking on the title bar reveals options for displaying the window*

Now when you click and hold the title bar, you see a small compass in the frame that the cursor is hovering over, as well as reference markers on each of the other window frames. The compass allows you to direct the placement of the window with respect to the frame you are hovering over. Move the window over another frame, and the compass hops to that one.



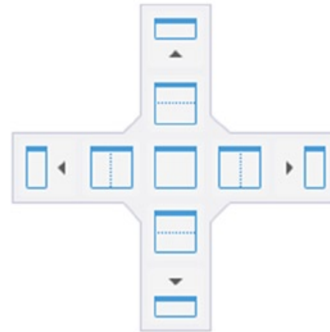


Figure 1-8. *Clicking and holding down the title bar reveals a compass*

The Center of the Compass

The compass itself has tabs for the directions (north, south, east, and west) as well as a center box. If you release the mouse over the center box, the window becomes tabbed within the current frame. Go ahead and drop it over the main frame, where the documents are edited. You can see now that it shares a frame with the other main windows.

When you hover over one of the compass direction tabs, the corresponding portion of the target frame is grayed out, so that you can preview the new window arrangement. If you drop the window in the wrong place, you can always either tear it off or manually set it to Dockable or Floating, depending on its state.

Play around with this a bit. In Figure 1-9, you can see the Solution Window as a tabbed document in the main window.

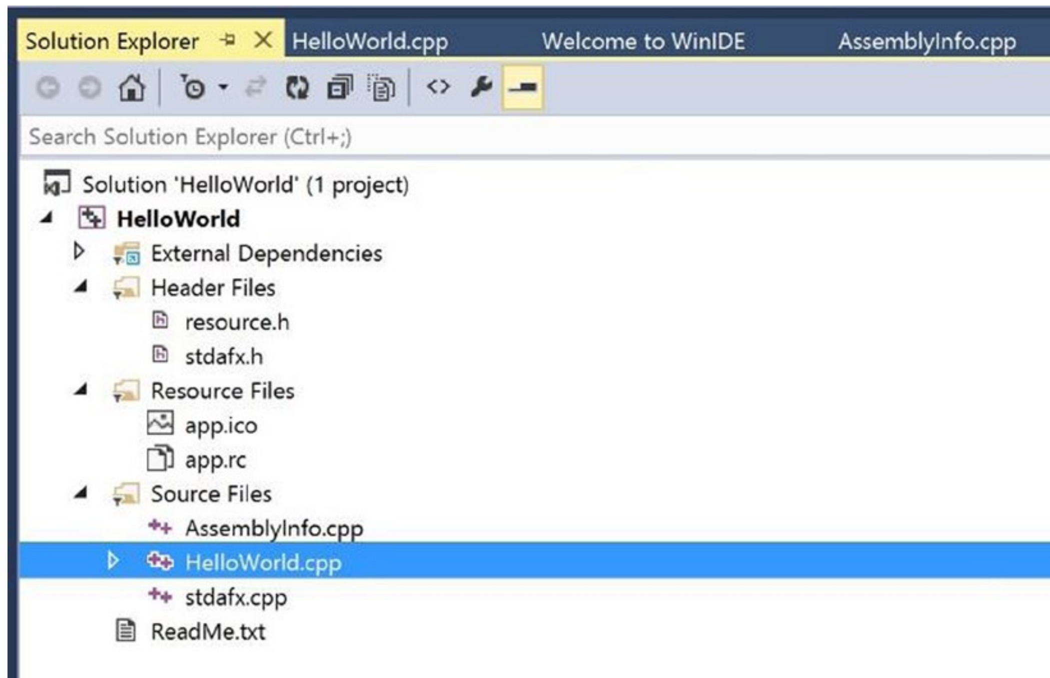


Figure 1-9. *Solution Explorer as a tabbed document in the main frame*

Building, Executing, and Debugging

Let's take a quick tour of some key Visual C++ IDE commands (see Table 1-2) as we build and test HelloWorld.

Table 1-2. *Common IDE Commands Quick Reference*

C#	C++	Explanation
F3	F3	Find next
F8	F4	Go to the next compilation error in the source
Shift-F8	Shift-F4	Go to the previous compilation error in the source
F5	F5	Execute with debugging
Ctrl-F5	Ctrl-F5	Execute without debugging
F6	F7	Build
F9	F9	Toggle breakpoint
F10	F10	Step over
F11	F11	Step into

Building the Program

Depending on our key bindings, we can use either F6 or F7 to build. If there are any errors, they appear in the Output window at the bottom of the screen, and you can use either F8 or F4 to cycle through them.

In C++, just as in C#, multiple compilation errors are often spurious; the compiler tries to compile beyond the first detected problem and may get lost. Often this allows you to see two or three errors and fix them all in a single editing pass. Just as often, the extra errors are an artifact of the compiler going out to lunch based on incorrect syntax, and fixing the first error or two may make the remainder disappear. I recommend building often.

Executing HelloWorld

The F5 key is the execute command. Because this is a console application, execution spawns a command window that displays "Hello World" and then quickly closes, which is somewhat unsatisfying. There are several ways around this. One approach is to create another Developer Command Prompt, navigate to the debug directory where the executable was created, and run the program manually, as we did earlier. Another way is to add the following call to the end of the `main()` function:

```
Console::ReadLine()
```

This method asks for a line of input from the user and keeps the console window open until the user presses the Enter key.

Another set of solutions presents itself by taking advantage of the built-in Visual C++ debugger. You could either set a breakpoint on the last line of the program using the F9 command, or you could just step through the program line by line. Either way, you can switch to the spawned command prompt to see the output as desired.

Let's try using the debugger.

Using the Visual C++ 2013 Debugger

The debugger is integrated into Visual Studio 2013, so initiating debugging is very simple. Entering any debugging command launches your application under the debugger. The window layout is sure to change, as there are several status windows that are only visible while debugging by default.

■ **Note** There are different window configurations for editing and debugging. Each configuration must be customized separately.

The basic debugging commands are F5 (Execute with Debugging), F9 (Toggle Breakpoint), F10 (Step Over Source Line), and F11 (Step Into Source Line).

Stepping Through the Code

A Step command executes a line of code in the program. There are two varieties of the Step command: F10 (Step Over) and F11 (Step Into). These are similar, yet they differ when applied to a function call. F10 executes until the line after the function call, whereas F11 stops execution at the first line of the function body. Of course, using F11 is always dependent on whether debugging information is available for the binary the function came from. Because debugging information for `Console::WriteLine()` is not distributed with Visual C++ 2013, both F10 and F11 step over the function.

Press F10 to begin debugging HelloWorld with Visual C++ 2013. The title bar changes to show "HelloWorld (Debugging)" to indicate debugging mode. In addition, a command window is spawned in a separate window. At this point, it is blank because HelloWorld has yet to display any information.

A small yellow arrow appears on the left edge of the editor window, which indicates the current line of code that is executing. Figure 1-10 shows that execution has stopped at this point, and the debugger awaits the next command.

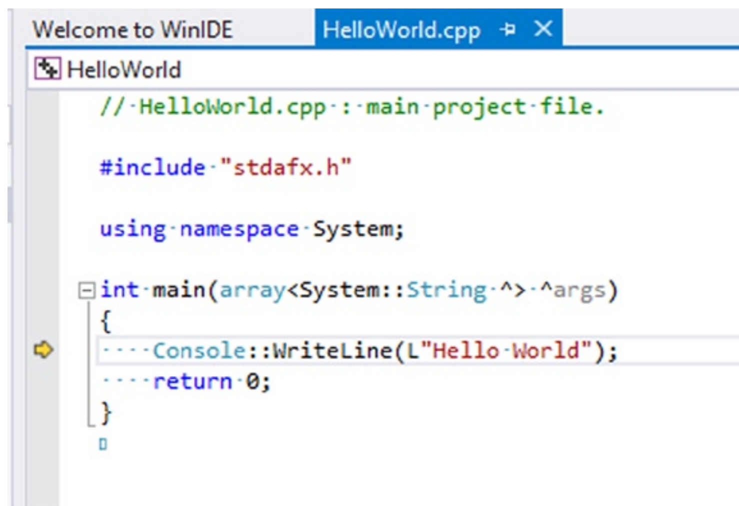


Figure 1-10. *Debugging HelloWorld*

The arrow indicates that we are beginning execution of the `main()` function, and the next line to be executed contains the `Console::WriteLine()` statement.

Press F10 again. The `Console::WriteLine()` function call executes, and "Hello World" appears in the separate command window.

If you dare to press F10 a couple more times, you create a nightmare on your screen. The first time, you execute over the return function. The next time, you return from the HelloWorld code into the C/C++ Runtime, or CRT. This module performs important tasks, including initializing your program in Windows, packaging the command-line arguments for your program, and handling the program's exit to Windows. Note that this code calls `main()` explicitly by name, which explains why every C++ program requires a global function called `main()`.

Completing Execution

Press F5 once to execute the remainder of the exit code and return to the editor. If `HelloWorld.cpp` is not visible, you can click the tab to reveal the source again. At this point, debugging has completed, and the title bar no longer indicates debugging.

Summary

This chapter provided you with a basic outline of how to create simple C++/CLI applications from the console and more sophisticated applications using the IDE. I also showed you how basic debugging can be performed in Visual C++ 2013 using the integrated debugger.

In the next chapter, we'll see how you can call C# from a simple C++ program.