Using the Visual Studio Debugger

F

Objectives

In this appendix you'll learn:

- To set breakpoints and run a program in the debugger.
- To use the **Continue** command to continue execution.
- To use the **Locals** window to view and modify the values of variables.
- To use the **Watch** window to evaluate expressions.
- To use the **Step Into**, **Step Out** and **Step Over** commands to control execution.
- To use the **Autos** window to view variables that are used in the surrounding statements.

Outline

F.1 Introduction

In Chapter 2, you learned that there are two types of errors—compilation errors and logic errors—and you learned how to eliminate compilation errors from your code. Logic errors (also called **bugs**) do not prevent a program from compiling successfully, but can cause the program to produce erroneous results when it runs. Most C compiler vendors provide software called a **debugger**, which allows you to monitor the execution of your programs to locate and remove logic errors. The debugger will be one of your most important program development tools. This appendix demonstrates key features of the Visual Studio debugger. Appendix G discusses the features and capabilities of the GNU debugger.

F.2 Breakpoints and the Continue Command

We begin our study of the debugger by investigating **breakpoints**, which are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether a logic error exists. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Note that attempting to set a breakpoint at a line of code that is not executable (such as a comment) will actually set the breakpoint at the next executable line of code in that function.

To illustrate the debugger's features, we use the program in Fig. F.1, which finds the maximum of three integers. Execution begins in `main` (lines 8–22). The three integers are input with `scanf` (line 15). Next, the integers are passed to `maximum` (line 19), which determines the largest integer. The value returned is returned to `main` by the `return` statement in `maximum` (line 38). The value returned is then printed in the `printf` statement (line 19).

```
1  /* Fig. G.1: figG_01.c
2     Finding the maximum of three integers */
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    int number1; /* first integer */
11    int number2; /* second integer */
12    int number3; /* third integer */
```

Fig. F.1 | Finds maximum of three integers. (Part 1 of 2.)

```
13
14     printf( "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     /* number1, number2 and number3 are arguments
18        to the maximum function call */
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20  } /* end main */
21
22  /* Function maximum definition */
23  /* x, y and z are parameters */
24  int maximum( int x, int y, int z )
25  {
26     int max = x; /* assume x is largest */
27
28     if ( y > max ) { /* if y is larger than max, assign y to max */
29        max = y;
30     } /* end if */
31
32     if ( z > max ) { /* if z is larger than max, assign z to max */
33        max = z;
34     } /* end if */
35
36     return max; /* max is largest value */
37  } /* end function maximum */
```

Fig. F.1 | Finds maximum of three integers. (Part 2 of 2.)

Creating a Project

In the following steps, you'll create a project that includes the code from Fig. F.1.

1. In Visual Studio select **File > New > Project...** to display the **New Project** dialog.
2. In the **Installed Templates** list under **Visual C++**, select **Win32**, and in the center of the dialog, select **Win32 Console Application**.
3. In the **Name:** field, enter a name for your project and in the **Location:** field, specify where you'd like to save the project on your computer, then click **OK**.
4. In the **Win32 Application Wizard** dialog, click **Next >**.
5. Under **Application type:**, select **Console application**, and under **Additional options:**, select **Empty project**, then click **Finish**.
6. In the **Solution Explorer**, right click your project's **Source Files** folder and select **Add > Existing Item...** to display the **Add Existing Item** dialog.
7. Locate the folder containing the Appendix F example code, select the code file and click **Add**.

Enabling Debug Mode and Inserting Breakpoints

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `number1` declared in Fig. F.1.

1. ***Enabling the debugger.*** The debugger is normally enabled by default. If it isn't enabled, you have to change the settings of the *Solution Configurations* combo

box (Fig. F.2) in the toolbar. To do this, click the combo box's down arrow, then select **Debug**.

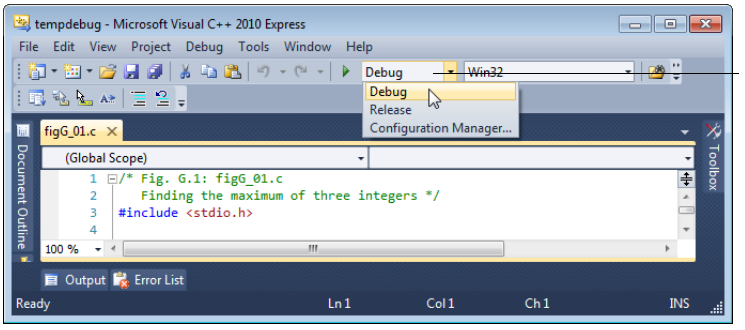

Fig. F.2 | Enabling the debugger.

2. ***Inserting breakpoints.*** Open the file `figG_01.c` by double-clicking it in the **Solution Explorer**. To insert a breakpoint, click inside the **margin indicator bar** (the gray margin at the left of the code window in Fig. F.3) next to the line of code at which you wish to break or right click that line of code and select **Breakpoint > Insert Breakpoint**. You can set as many breakpoints as necessary. Set breakpoints at lines 14 and 19 of your code. A red circle appears in the margin indicator bar where you clicked, indicating that a breakpoint has been set (Fig. F.3). When the program runs, the debugger pauses execution at any line that contains a breakpoint. The program is said to be in **break mode** when the debugger pauses the program. Breakpoints can be set before running a program, in break mode and while a program is running.

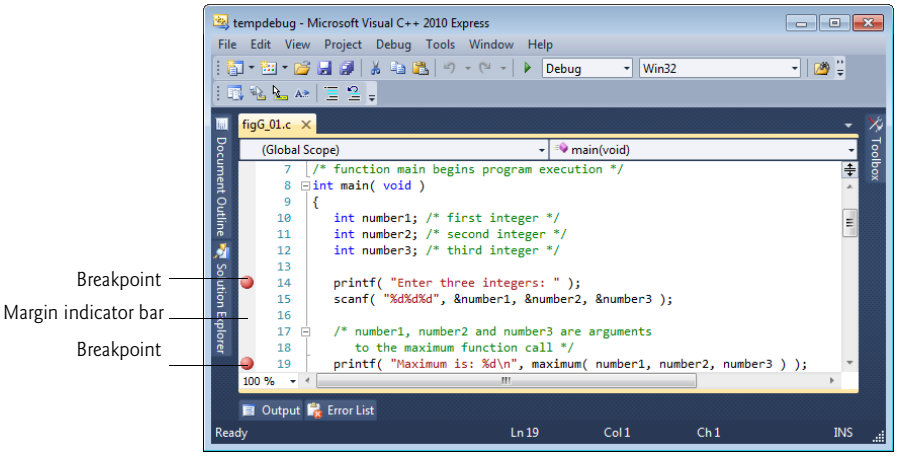

Fig. F.3 | Setting two breakpoints.

3. ***Starting to debug.*** After setting breakpoints in the code editor, select **Debug > Build Solution** to compile the program, then select **Debug > Start Debugging** to begin the debugging process. [*Note:* If you do not compile the program first, it will still be compiled when you select **Debug > Start Debugging**.] When you debug a console application, a **Command Prompt** window appears (Fig. F.4) in which you can specify program input and view program output. The debugger enters break mode when execution reaches the breakpoint at line 14.

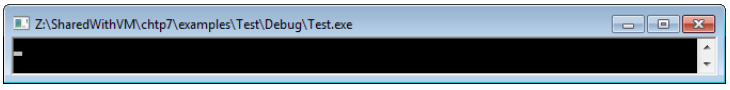

Fig. F.4 | **Maximum Numbers** program running.

4. ***Examining program execution.*** Upon entering break mode at the first breakpoint (line 14), the IDE becomes the active window (Fig. F.5). The **yellow arrow** to the left of line 14 indicates that this line contains the next statement to execute.

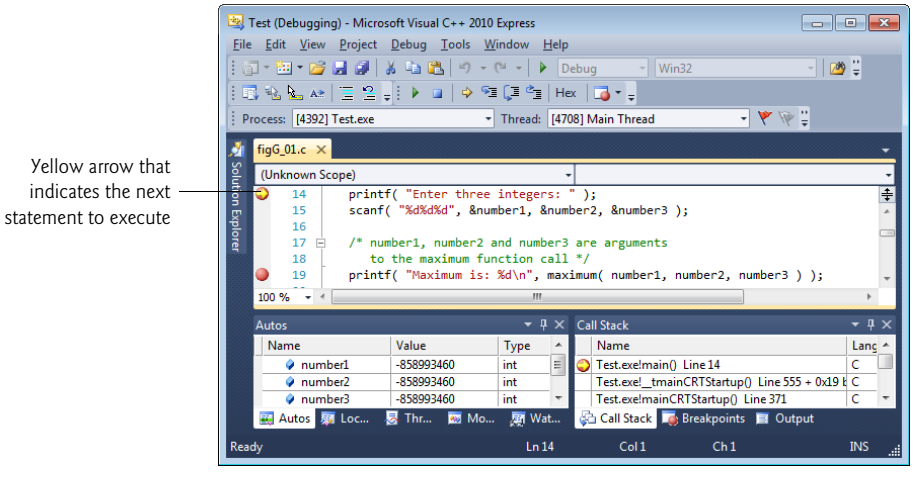

Fig. F.5 | Program execution suspended at the first breakpoint.

5. ***Using the* Continue *command to resume execution.*** To resume execution, select **Debug > Continue**. The **Continue command** resumes program execution until the next breakpoint or the end of `main` is encountered, whichever comes first. The program continues executing and pauses for input at line 15. Enter the values `22`, `85`, and `17` as the three integers separated by spaces. The program executes until it stops at the next breakpoint (line 19). When you place your mouse pointer over the variable name `number1`, the value stored in the variable is displayed in a ***Quick Info* box** (Fig. F.6). As you'll see, this can help you spot logic errors in your programs.

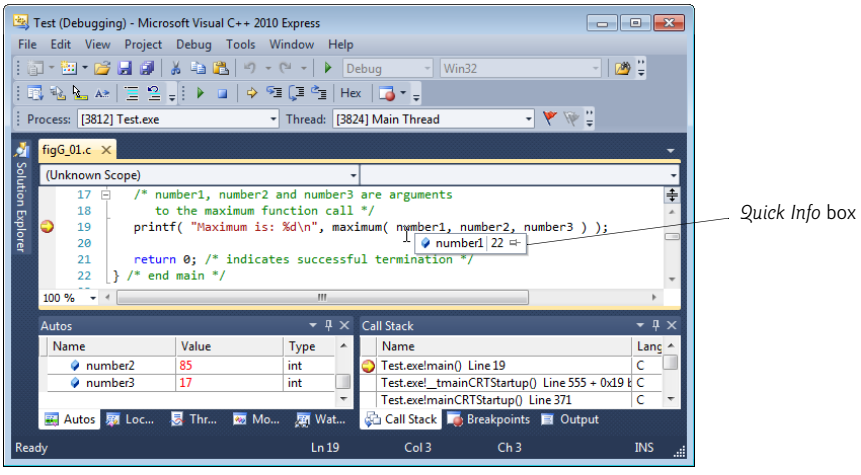

Fig. F.6 | *Quick Info* box showing the value of a variable.

6. ***Setting a breakpoint at `main`'s closing brace.*** Set a breakpoint at line 22 in the source code by clicking in the margin indicator bar to the left of line 22. This will prevent the program from closing immediately after displaying its result. When there are no more breakpoints at which to suspend execution, the program will execute to completion and the **Command Prompt** window will close. If you do not set this breakpoint, you won't be able to view the program's output before the console window closes.
7. ***Continuing program execution.*** Use the **Debug > Continue** command to execute the code up to the next breakpoint. The program displays the result of its calculation (Fig. F.7).

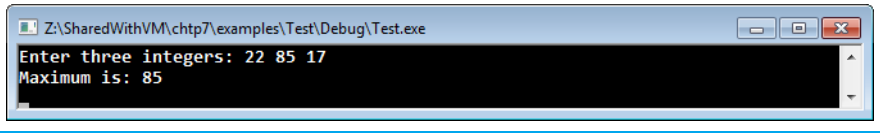

Fig. F.7 | Program output.

8. ***Removing a breakpoint.*** Click the breakpoint in the margin indicator bar.
9. ***Finishing program execution.*** Select **Debug > Continue** to execute the program to completion.

In this section, you learned how to enable the debugger and set breakpoints so that you can examine the results of code while a program is running. You also learned how to continue execution after a program suspends execution at a breakpoint and how to remove breakpoints.

F.3 Locals and Watch Windows

In the preceding section, you learned that the *Quick Info* feature allows you to examine a variable's value. In this section, you'll learn to use the **Locals window** to assign new values to variables while your program is running. You'll also use the **Watch window** to examine the value of more complex expressions.

1. ***Inserting breakpoints.*** Clear the existing breakpoints by clicking each one in the margin indicator bar. Then, set a breakpoint by clicking in the margin indicator bar to the left of line 19 (Fig. F.8).

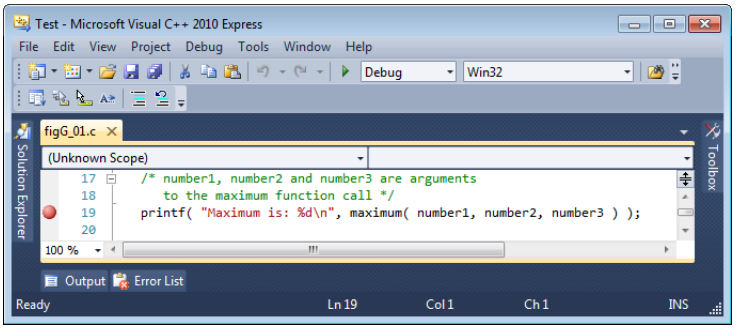

Fig. F.8 | Setting breakpoints at lines 25 and 28.

2. ***Starting debugging.*** Select **Debug > Start**. Enter the values `22`, `85`, and `17` at the **Enter three integers:** prompt and press *Enter* so that your program reads the values you just entered.
3. ***Suspending program execution.*** The debugger enters break mode at line 19 (Fig. F.9). At this point, line 15 read the values that you entered for `number1` (22), `number2` (85) and `number3` (17). Line 19 is the next statement to execute.

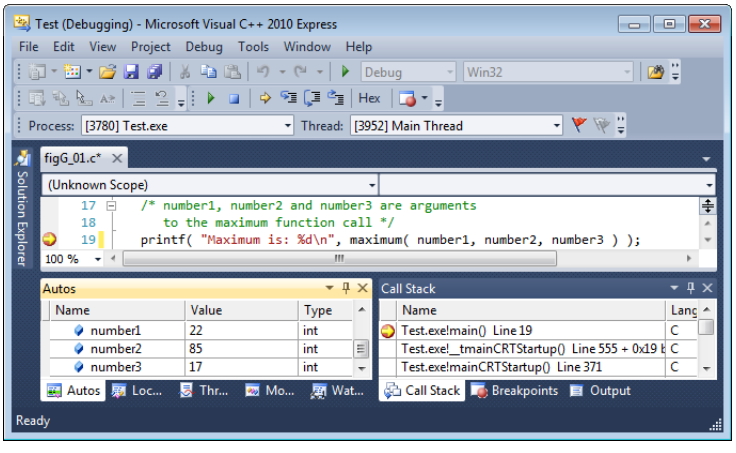

Fig. F.9 | Program execution suspended when debugger reaches the breakpoint at line 19.

4. ***Examining data.*** In break mode, you can explore the values of your local variables using the debugger's **Locals** window, which is normally displayed at the bottom left of the IDE when you are debugging. If it's not shown, you can view the **Locals** window, select **Debug > Windows > Locals**. Figure F.10 shows the values for `main`'s local variables `number1` (22), `number2` (85) and `number3` (17).

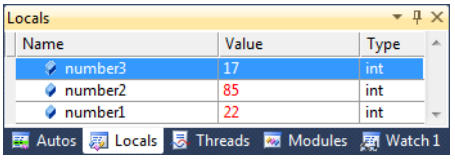

Fig. F.10 | Examining variables `number1`, `number2` and `number3`.

5. ***Evaluating arithmetic and boolean expressions.*** You can evaluate arithmetic and boolean expressions using the **Watch** window. You can display up to four **Watch** windows. Select **Debug > Windows > Watch 1**. In the first row of the **Name** column, type `(number1 + 3) * 5`, then press *Enter*. The value of this expression (`125` in this case) is displayed in the **Value** column (Fig. F.11). In the next row of the **Name** column, type `number1 == 3`, then press *Enter*. This expression determines whether the value of `number1` is `3`. Expressions containing the `==` operator (or any other relational or equality operator) are treated as `bool` expressions. The value of the expression in this case is `false` (Fig. F.11), because `number1` currently contains `22`, not `3`.

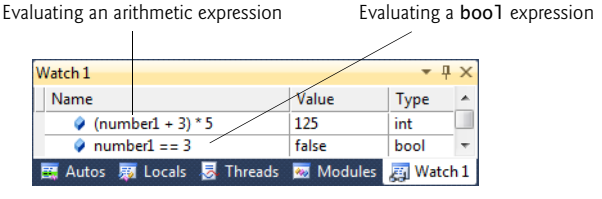

Fig. F.11 | Examining the values of expressions.

6. ***Modifying values.*** Based on the values input by the user (`22`, `85` and `17`), the maximum number output by the program should be `85`. However, you can use the **Locals** window to change the values of variables during the program's execution. This can be valuable for experimenting with different values and for locating logic errors. In the **Locals** window, click the **Value** field in the `number1` row to select the value `22`. Type `90`, then press *Enter*. The debugger changes the value of `number1` and displays its new value in red (Fig. F.12).
7. ***Setting a breakpoint at `main`'s closing brace.*** Set a breakpoint at line 22 in the source code to prevent the program from closing immediately after displaying its result. If you do not set this breakpoint, you won't be able to view the program's output before the console window closes.

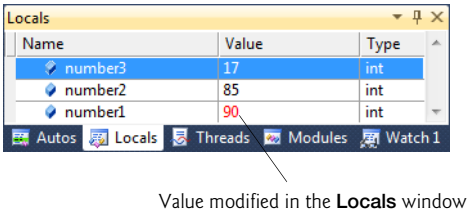

Fig. F.12 | Modifying the value of a variable.

8. ***Resuming execution and viewing the program result.*** Select **Debug > Continue** to continue program execution. Function `main` executes until the `return` statement in line 21 and displays the result. The result is `90` (Fig. F.13). This shows that *Step 6* changed the value of `number1` from the original vlaue (`85`) to `90`.

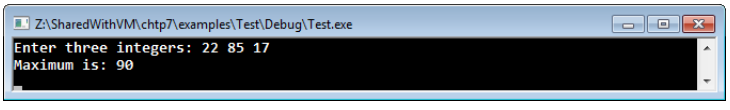

Fig. F.13 | Output displayed after modifying the `number1` variable.

9. ***Stopping the debugging session.*** Select **Debug > Stop Debugging**. This will close the **Command Prompt** window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's **Watch** and **Locals** windows to evaluate arithmetic and boolean expressions. You also learned how to modify the value of a variable during your program's execution.

F.4 Controlling Execution Using the Step Into, Step Over, Step Out and Continue Commands

Sometimes executing a program line by line can help you verify that a function's code executes correctly, and can help you find and fix logic errors. The commands you learn in this section allow you to execute a function line by line, execute all the statements of a function at once or execute only the remaining statements of a function (if you've already executed some statements within the function).

1. ***Setting a breakpoint.*** Set a breakpoint at line 19 by clicking in the margin indicator bar to the left of the line.
2. ***Starting the debugger.*** Select **Debug > Start**. Enter the values `22`, `85` and `17` at the **Enter three integers:** prompt. Execution will halt when the program reaches the breakpoint at line 19.
3. ***Using the Step Into command.*** The **Step Into command** executes the next statement in the program (line 19), then immediately halts. If that statement contains a function call (as is the case here), control transfers into the called function. This enables you to execute each statement inside the function individually to confirm

the function's execution. Select **Debug > Step Into** to enter the `maximum` function. Then, Select **Debug > Step Into** again so the yellow arrow is positioned at line 28 as shown in Fig. F.14.

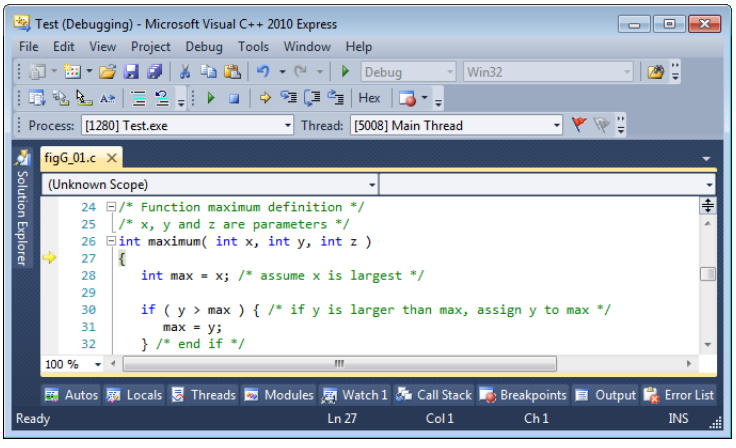

Fig. F.14 | Stepping into the `maximum` function.

4. ***Using the Step Over command.*** Select **Debug > Step Over** to execute the current statement (line 28 in Fig. F.14) and transfer control to line 30 (Fig. F.15). The **Step Over command** behaves like the **Step Into** command when the next statement to execute does not contain a function call. You'll see how the **Step Over** command differs from the **Step Into** command in *Step 10*.

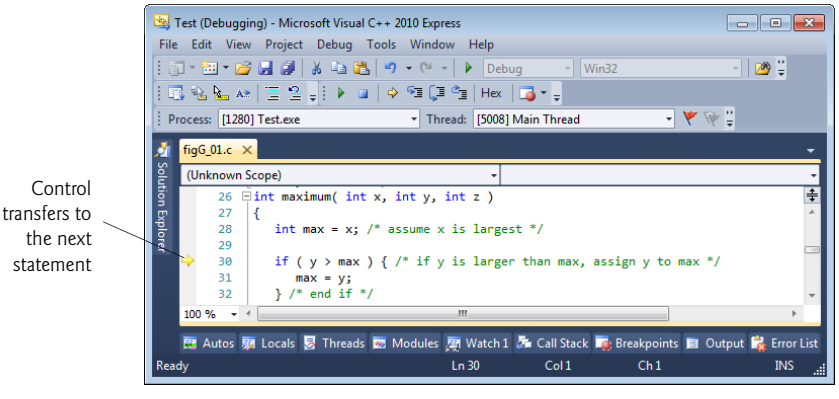

Fig. F.15 | Stepping over a statement in the `maximum` function.

5. ***Using the Step Out command.*** Select **Debug > Step Out** to execute the remaining statements in the function and return control to the next executable statement (line 21 in Fig. F.1). Often, in lengthy functions, you'll want to look at a few key lines of code, then continue debugging the caller's code. The **Step Out command**

enables you to continue program execution in the caller without having to step through the entire called function line by line.

6. ***Setting a breakpoint.*** Set a breakpoint at the end of `main` at line 22 of Fig. F.1. You'll make use of this breakpoint in the next step.
7. ***Using the* Continue *command.*** Select **Debug > Continue** to execute until the next breakpoint is reached at line 22. Using the **Continue** command is useful when you wish to execute all the code up to the next breakpoint.
8. ***Stopping the debugger.*** Select **Debug > Stop Debugging** to end the debugging session. This will close the **Command Prompt** window.
9. ***Starting the debugger.*** Before we can demonstrate the next debugger feature, you must start the debugger again. Start it, as you did in *Step 2*, and enter `22`, `85` and `17` in response to the prompt. The debugger enters break mode at line 19.
10. ***Using the* Step Over *command.*** Select **Debug > Step Over** (Fig. F.16) Recall that this command behaves like the **Step Into** command when the next statement to execute does not contain a function call. If the next statement to execute contains a function call, the called function executes in its entirety (without pausing execution at any statement inside the function), and the yellow arrow advances to the next executable line (after the function call) in the current function. In this case, the debugger executes line 19, located in `main` (Fig. F.1). Line 19 calls the `maximum` function. The debugger then pauses execution at line 21, the next executable line in the current function, `main`.

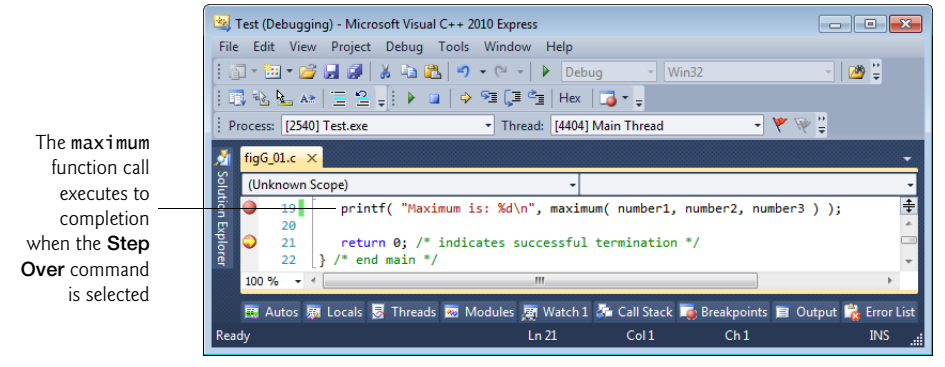

Fig. F.16 | Using the debugger's **Step Over** command.

11. ***Stopping the debugger.*** Select **Debug > Stop Debugging**. This will close the **Command Prompt** window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's **Step Into** command to debug functions called during your program's execution. You saw how the **Step Over** command can be used to step over a function call. You used the **Step Out** command to continue execution until the end of the current function. You also learned that the **Continue** command continues execution until another breakpoint is found or the program exits.

F.5 Autos Window

The **Autos window** displays the variables used in the previous statement executed (including the return value of a function, if there is one) and the variables in the next statement to execute.

1. ***Setting breakpoints.*** Set breakpoints at lines 14 and 19 in `main` by clicking in the margin indicator bar.
2. ***Using the Autos window.*** Start the debugger by selecting **Debug > Start**. When the debugger enters break mode at line 14, open the **Autos** window (Fig. F.17) by selecting **Debug > Windows > Autos**. Since we are just beginning the program's execution, the **Autos** window lists only the variables in the next statement that will execute—in this case, `number1`, `number2` and `number3`. Viewing the value stored in a variable lets you verify that your program is manipulating these variables correctly. Notice that `number1`, `number2` and `number3` contain large negative values. These values, which may be different each time the program executes, are the variables' uninitialized values. This unpredictable (and often undesirable) value demonstrates why it's important to initialize all C variables before they are used.

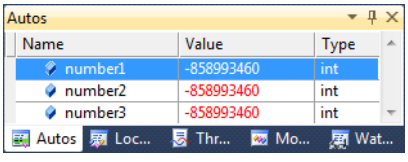

Fig. F.17 | **Autos** window displaying the values of `number1`, `number2` and `number3`.

3. ***Continuing execution.*** Select **Debug > Continue** to execute the program until the second breakpoint at line 19. At the program's input prompt, enter values for the three integers. The **Autos** window updates the values of `number1`, `number2` and `number3` after they're initialized. Their values appear in red to indicate that they've just changed. (Fig. F.18)

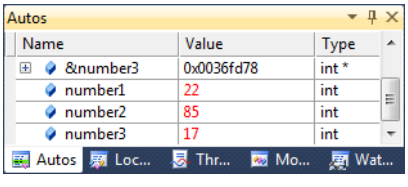

Fig. F.18 | **Autos** window displaying local variables `number1`, `number2` and `number3`.

4. ***Entering data.*** Select **Debug > Step Over** to execute line 19. The **Autos** window displays the return value of function `maximum` (Fig. F.19).

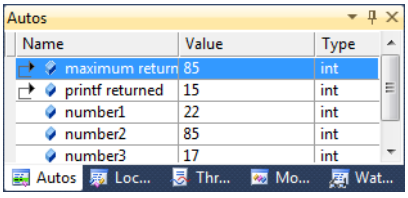

Fig. F.19 | **Autos** window displaying value returned by function `maximum`.

5. *Stopping the debugger.* Select **Debug > Stop Debugging** to end the debugging session. Remove all remaining breakpoints.

F.6 Wrap-Up

In this appendix, you learned how to insert, disable and remove breakpoints in the Visual Studio debugger. Breakpoints allow you to pause program execution so you can examine variable values. This capability will help you locate and fix logic errors in your programs. You saw how to use the **Locals** and **Watch** windows to examine the value of an expression and how to change the value of a variable. You also learned debugger commands **Step Into**, **Step Over**, **Step Out** and **Continue** that can be used to determine whether a function is executing correctly. Finally, you learned how to use the **Autos** window to examine variables used specifically in the previous and next commands.