Exercise VI: Debugging II – Walking through a program

In Exercise V we looked at a program that had crashed and how to determine where it had crashed. This exercise is actually a collection of exercises intended to introduce features of the Visual C++ debugger and the ways in which a programmer can walk through the execution of a program in as much or as little detail as they choose.

There is a zip file named **e6.zip** at the anonymous FTP server **ftp.cs.umd.edu** in the **/pub/egolub/VC.workbook** directory. Download this file and extract the files which it contains. Unzip those files to a temporary directory on your machine.

Launch Visual C++ on your computer. Create a new, empty, **Win32 Console Application** named **exercise6_1**. Go to the project settings and disable the language extensions as shown in Exercise II. Go to your Windows environment and copy the file **main1.cpp** that you extracted from e6.zip into the exercise6_1 directory. Return to the Visual C++ environment and that file to the project. Compile and run the program. Your output should be the following:

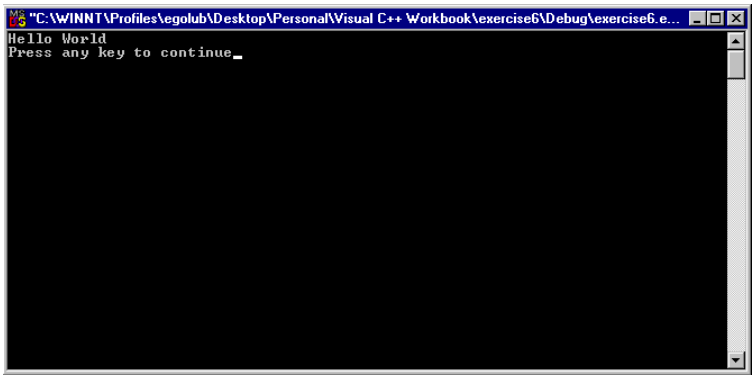

Figure VI.1

In Visual C++ ***double click*** on **main1.cpp** in the FileView to bring that file into the editor. Notice that there are actually four cout statements used to create the displayed output. In this first main program for this exercise, we will observe the behavior of an output stream.

Our goal is to have the program pause when it reaches the line of code that prints the word **World** to the screen and observe what has been printed to the console so far. We will then use the debugger's tools to step through the execution of that line of code and the remaining lines of code to observe each step that happens.

First, we need to inform Visual C++ that we want the program to pause at this line of code when we are running the program via the debugger. This is called inserting a

breakpoint in the program. You can instruct Visual C++ to insert (or later remove) a breakpoint in either of the following ways:

- Position the mouse in the gray strip to the left of the source code so that it is aligned with the line of code on which you want to set the breakpoint or position the mouse over the line of code itself. Right click the mouse to bring up the context menu and select the **Insert/Remove Breakpoint** option

- Position the cursor somewhere on the line of code on which you want to set the breakpoint and click on the breakpoint button (shown in Figure VI.2)

Figure VI.2

After specifying to Visual C++ to insert the breakpoint, a solid red circle will appear next to the line of code with which the breakpoint is associated. In this example, your screen should appear similar to that in Figure VI.3. If you insert a breakpoint in the incorrect location, you can right click on the red circle and select **Remove Breakpoint** from the context menu that appears.

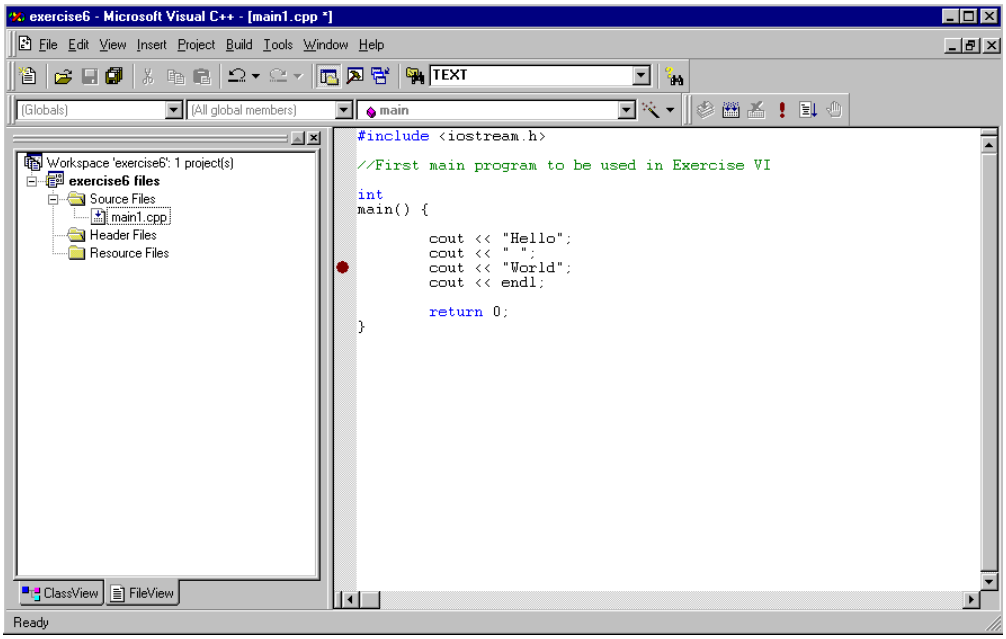

Figure VI.3

You can now start the program running in the debugger by using the **Go** command (see page V-2). The program will execute normally until it arrives at the marked line of code. At that point, execution of the program will be paused, but the program will still be "alive" – the entire state of the program (variables, etc.) will remain and you will be able to resume execution of the program as if it had not been stopped. At this point, Visual C++ will be brought back to the foreground and should appear similar to Figure VI.4.

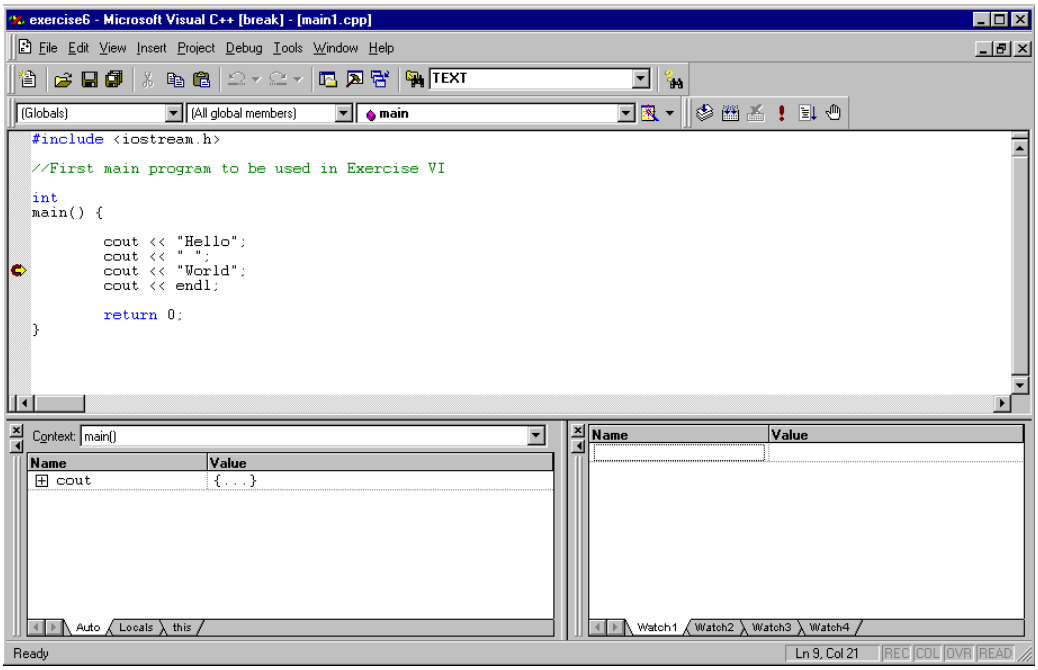

Figure VI.4

The yellow arrow shows the line of code on which execution of the program was stopped. At this point, the characters Hello and a blank character have been sent to the standard output stream. However, as you may have experienced before in your programming, output streams have buffers to which information is put before actually being sent to the stream. Until that buffer is flushed either explicitly or by the internals of the program, the information is not displayed. In this environment, we can observe this by switching to the DOS console window in which our output is displayed. If you do this, you will see an empty console window such as is shown in Figure VI.5.

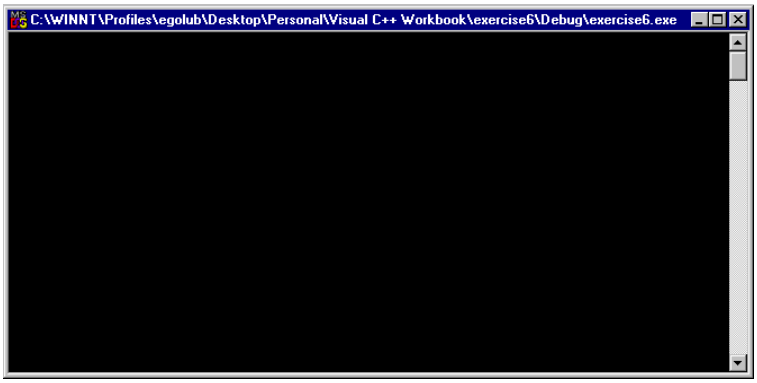

Figure VI.5

We can now step through the program line by line and observe what happens after each line. The command we will use in this first experience is the **Step Over** command. This command instructs the debugger to execute the line of code on which the program has been paused in total, and then pause at the beginning of the next line of code that would be executed. You can instruct Visual C++ step over a line of code by pressing the **F10**

key a single time. Switch back to the Visual C++ window and press the **F10** key a single time now. Now that you have instructed the debugger to step over the current line of code, it has been executed and the yellow arrow will point to the next line of code that is to be executed. In Figure VI.6 notice that the red circle still represents where we set the breakpoint and that the yellow arrow is pointing to the next line of code that is expected.

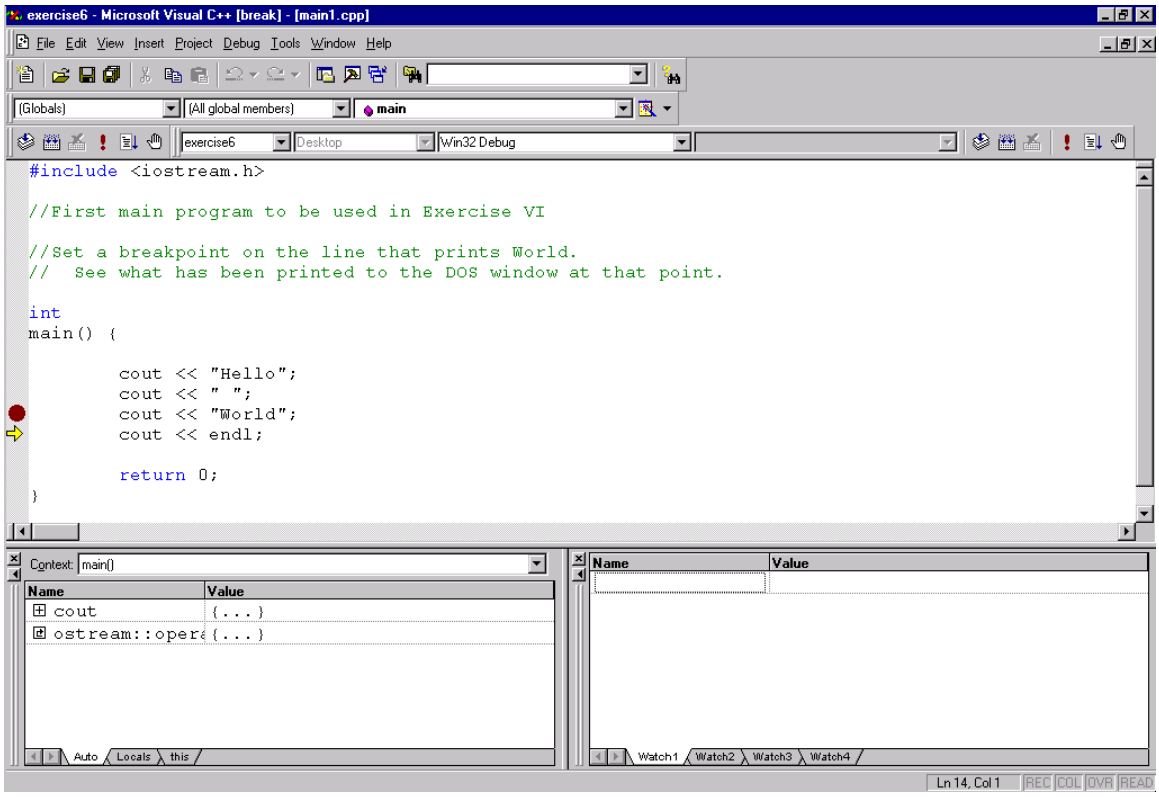

Figure VI.6

If you switch back to the console window, you will see that nothing has appeared yet. This is because the output buffer still has not been flushed. Switch back to the Visual C++ window and press **F10** one more time to execute the current line of code.

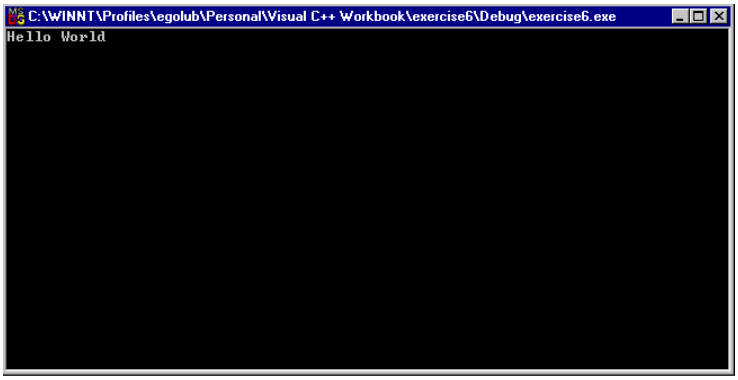

Figure VI.7

When **endl** is sent to the output buffer, in addition to sending a new line to the output buffer, it causes the buffer to be flushed. We can confirm this by switching to the console window, which will appear like the one in Figure VI.7. If you would like to experiment with something later, you can modify this program so that rather than **endl** the program uses '\n' to insert the end of line character. You can then walk through the execution and see when the output is flushed in that situation.

At this point, if you switch back to the Visual C++ window, the yellow arrow will be pointing to the **return** statement at the end of our program. If you continue to use **F10** to step over the execution of code, you will reach the end of the main program and will be brought into system code. Although this is something we do not typically do, it is a good exercise to "accidentally" do it now to see how it appears and also to see how to return to familiar ground. Currently, the yellow arrow should be pointing to the **return** statement. Press the **F10** key twice now. Your screen should look similar to Figure VI.8. (Note: Depending upon how Visual C++ was installed on your machine, you may get a dialog box at this point asking you to give the path to a file CRT0.C. If this occurs, use your system's find tool to find location of that file on your hard drive, and use the interface Visual C++ provides to identify that path. If you simply press **Cancel**, Visual C++ will take you to the system-level code rather than the code shown in Figure VI.8 below.)

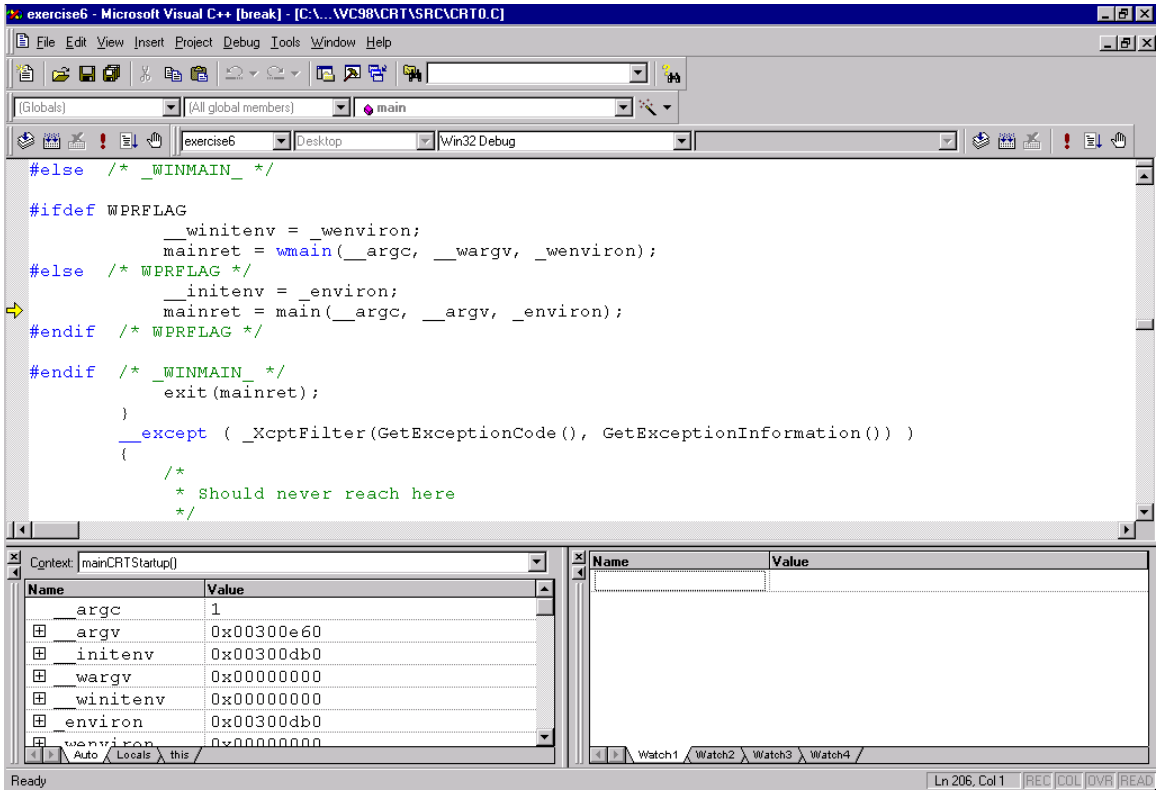

Figure VI.8

What you are now seeing is the code that is inserted by Visual C++ as a wrapper around your code. This code establishes the console window and invokes your program within that window. We are no longer at a place where we are interested in walking through the

code being executed. In this case, the **Go** command can come in handy again. Recall that we used the **Go** command to start the program in the debugger. It also serves the purpose of instructing the debugger to resume executing the program normally until the next breakpoint is reached. If there are no more breakpoints (as will be the case here) then it will run the program to completion. Use the **Go** command now.

The program has now completed its execution. There are two things worth noting at this point; (1) the console window has closed automatically and (2) the editor is now displaying the file in which we were last located during debugging. We can bring our code back up in the editor window (if it is not currently displayed) by ***double clicking*** on its name under FileView.

We are now ready to move to our second task in this exercise. Close the current project and create a new project called **exercise6_2**. Go to your Windows environment and copy the file **main2.cpp** that you extracted from e6.zip into the exercise6_2 directory. Return to the Visual C++ environment and that file to the project and compile it. With this second main function, we are going to observe conditional statements via our debugger.

Recall that with a conditional statement, there is code that will not always be executed – the execution depends on the test condition. In **main2.cpp** we have a program that asks the user to enter a number and then either prints the word **TRUE** or the word **FALSE** based on whether the entered value was less than **15**. Go the FileView and double click on **main2.cpp** to bring that file into the editor. Insert a breakpoint on the line of code that prints out the word **TRUE** as shown in Figure VI.9.

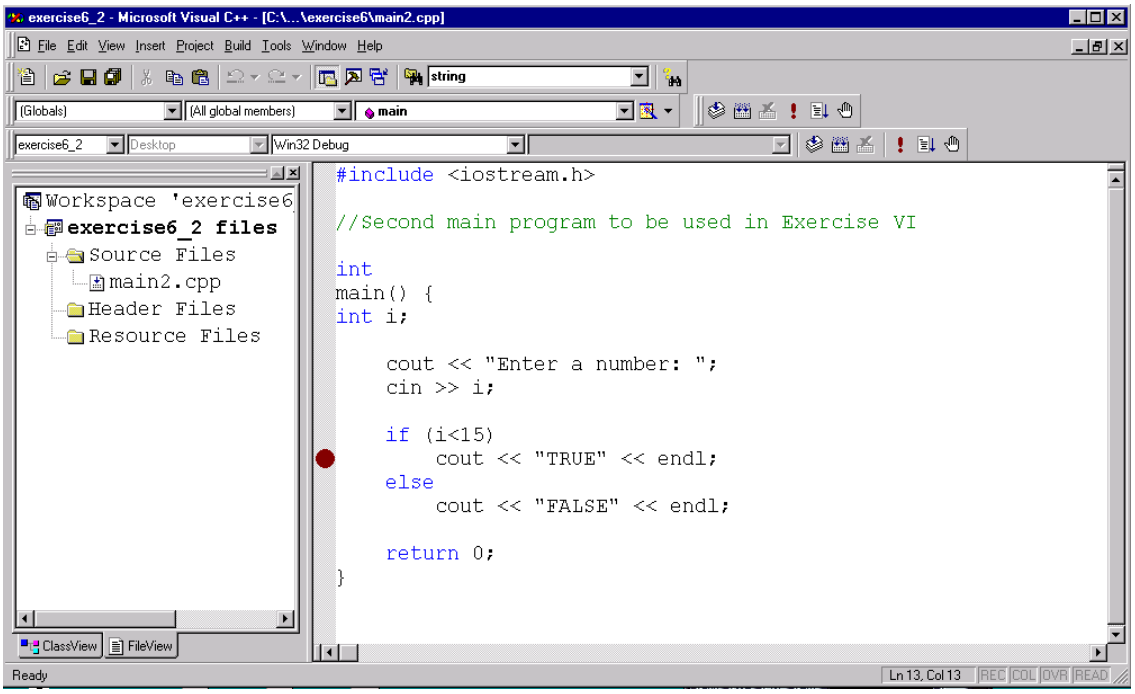

Figure VI.9

Next, compile the project and begin running the program using the **Go** command to start the debugger. The program will begin to run and the console window will appear in the foreground with a prompt for you to enter a number. Enter the number **10** and press the **enter** key. At this point, the Visual C++ window will come to the foreground and appear similar to Figure VI.10. The program has been paused on the line of code that is going to print the word **TRUE** to the screen.

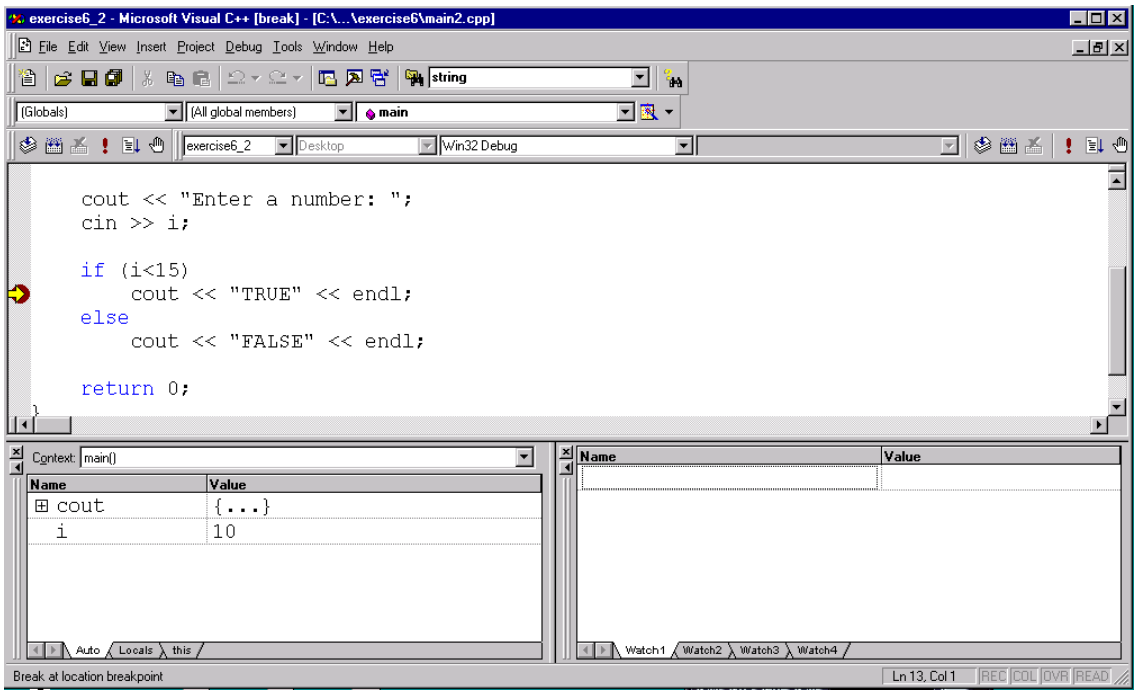

Figure VI.10

Press **F10** to instruct the debugger to step over this line of code and bring us to the next line of code that is to be executed. Notice that the yellow arrow is now pointing at the line that says **else**. However, if you press **F10** again, it does not take us to the contents of the else, but rather jumps over them to the return statement. This is because in an **if-then-else** conditional, only one or the other will have its body executed. Now that you are at the return statement, use the **Go** command to resume execution of the remainder of the program.

Without making any changes, use the **Go** command to begin running the program in the debugger again. This time, when prompted for a number, enter the number **20**. Notice that after you entered the number, the program continued on until completion – it did not stop at the breakpoint that exists in the program. The reason is that the line of code on which the breakpoint is set was never executed.

We are now ready to move to our third task in this exercise. Close the current project and create a new project called **exercise6_3**. Go to your Windows environment and copy the file **main3.cpp** that you extracted from e6.zip into the exercise6_3 directory. Return to the Visual C++ environment and that file to the project. Now add **main3.cpp** to the project and compile it. With this third main function, we are going to observe function

calls via our debugger and explore another command within the debugger – the **step into** command.

First, set a breakpoint on the first **cout** statement in the main function. Your window should appear similar to Figure VI.11.

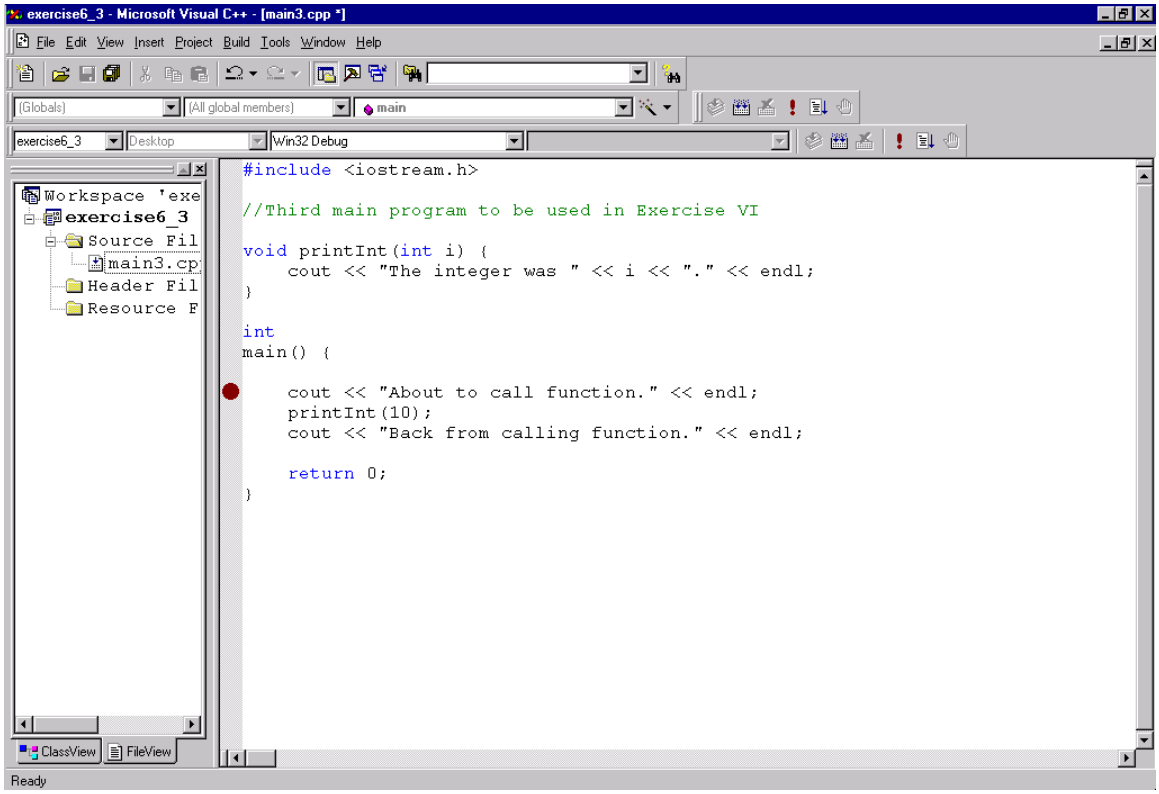

Figure VI.11

Begin running the program in the debugger. The program will pause when it is about to execute that first **cout** statement. Use **F10** to step over each of the three lines of code in the main function. When you are positioned at the return statement, use the **Go** command to run the program to completion. Notice that with the step over command, each line of code appeared to be executed as a single entity. However, each of these three lines has several components. The **cout** statements have several calls to the overloaded **<< operator** and the line of code between them calls the user defined function **printInt()**. It is often useful to **step into** a line of code that is about to be executed in order to observe the details. To do this we can use **F11** to **step into** the current line of code.

Begin running the program again using the **Go** command, but this time when you reach the breakpoint, press the **F11** key once. This will **step into** the first call to the **<< operator**. (Note: As was true previously with the file CRT0.C, Visual C++ might ask you to identify the path to OSTREAM.CPP on your system – use same technique to do so.) Your window should appear similar to Figure VI.12. Notice that you are now looking at the actual code for the **ostream << operator**.

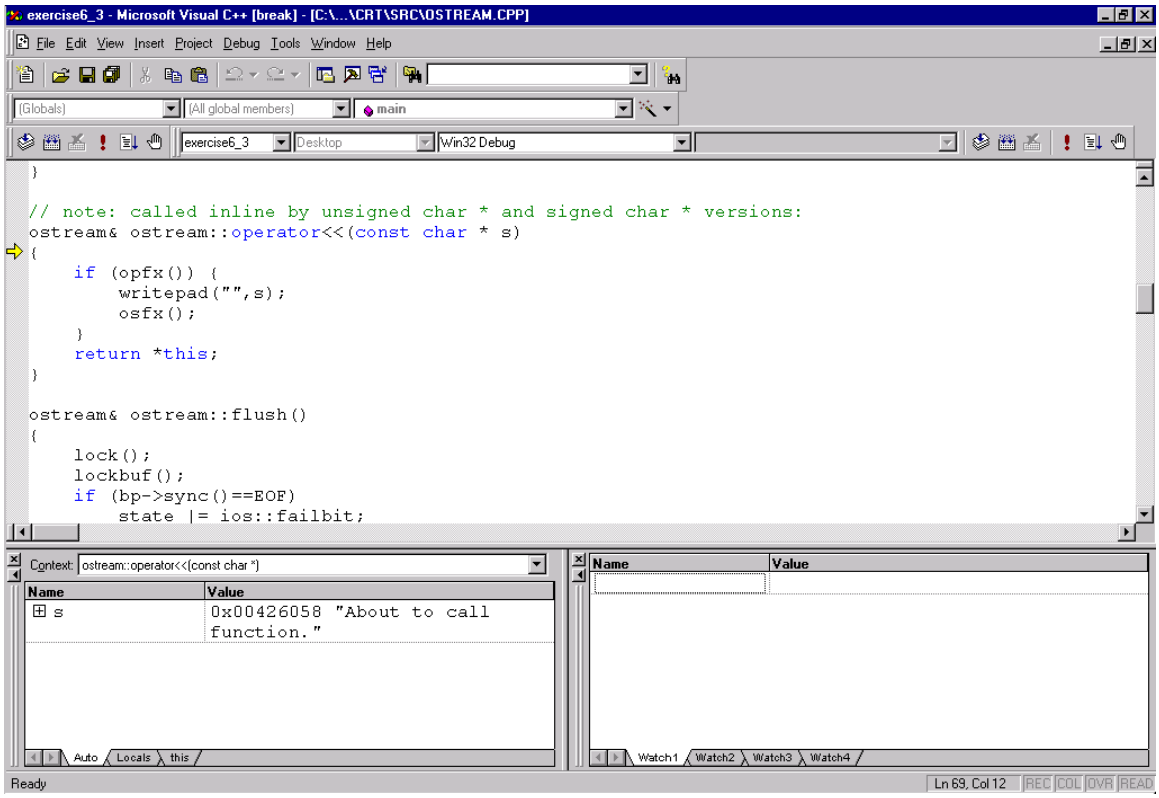

Figure VI.12

In this case, stepping into the execution of the line of code really isn't going to help us during debugging. However, it is good to experience stepping into a system function so that we know what it looks like and also so that we can see how to step back out of it. You will find that you will occasionally step into a line of code and then decide that you would rather have stepped over it. To step over a line of code that you stepped into, you can use the **step out** command by depressing the **Shift** key and then pressing **F11** with the **Shift** key held down (**Shift**+**F11**). Doing this will execute the remainder of the function which you stepped into, and return you to the line of code which invoked it.

Now you have been returned to the line of code which invoked the << **operator**, but the line of code has not finished running yet – only the first call to the << **operator** has run. If you were to use **F11** again, you would step into the second call to the << **operator**. Rather than doing that, use **F10** to step over the remainder of this line of code. That will bring you to the call the **printInt()**.

Now that the yellow arrow is positioned at the call to **printInt()** press **F11** to step into that function call. The yellow arrow indicating the line of code about to be executed has now jumped to the beginning of the **printInt()** function. We can now step through each line of code in this function as we choose – either using **F10** or **F11**. We leave this choice to you. Continue step through this program until it has run to completion in any way that you choose. The important thing is that after completing this task, you should be comfortable with stepping **over**, **into** and back **out of** code.

We are now ready to move to our fourth task in this exercise. Close the current project and create a new project called **exercise6_4**. Go to your Windows environment and copy the file **main4.cpp** that you extracted from e6.zip into the exercise6_4 directory. Return to the Visual C++ environment and that file to the project. Now add **main4.cpp** to the project and compile it. With this fourth main function, we are going to observe the execution of code within a loop via our debugger. We will also experiment with reference -*vs*- value parameters as well as looking at the address of a variable in memory using **watch** lists.

In **main4.cpp** add a breakpoint to the line of code within the while loop that calls **printInt()** and begin running the program using the debugger. The program will pause when it gets to that line of code. Use **F5** to resume execution of the program. Notice that the program pauses once again on that same line. However, this time you are in the second iteration of the loop. Each time a line of code with a breakpoint is reached, the debugger will pause.

Notice that in Figure VI.13 (the first time we hit the breakpoint) the value of **loopControl** as displayed in the lower left-hand panel is **0** while in Figure VI.14 (the second time we hit the breakpoint) is **1**. Each time you press **F5** to resume running the program, the program will come back to pause at the same line of code and the value of **loopControl** will reflect the iteration of the loop. After pressing **F5** a few times to observe the behavior, you can press **Shift**+**F5** to stop the execution of the program.

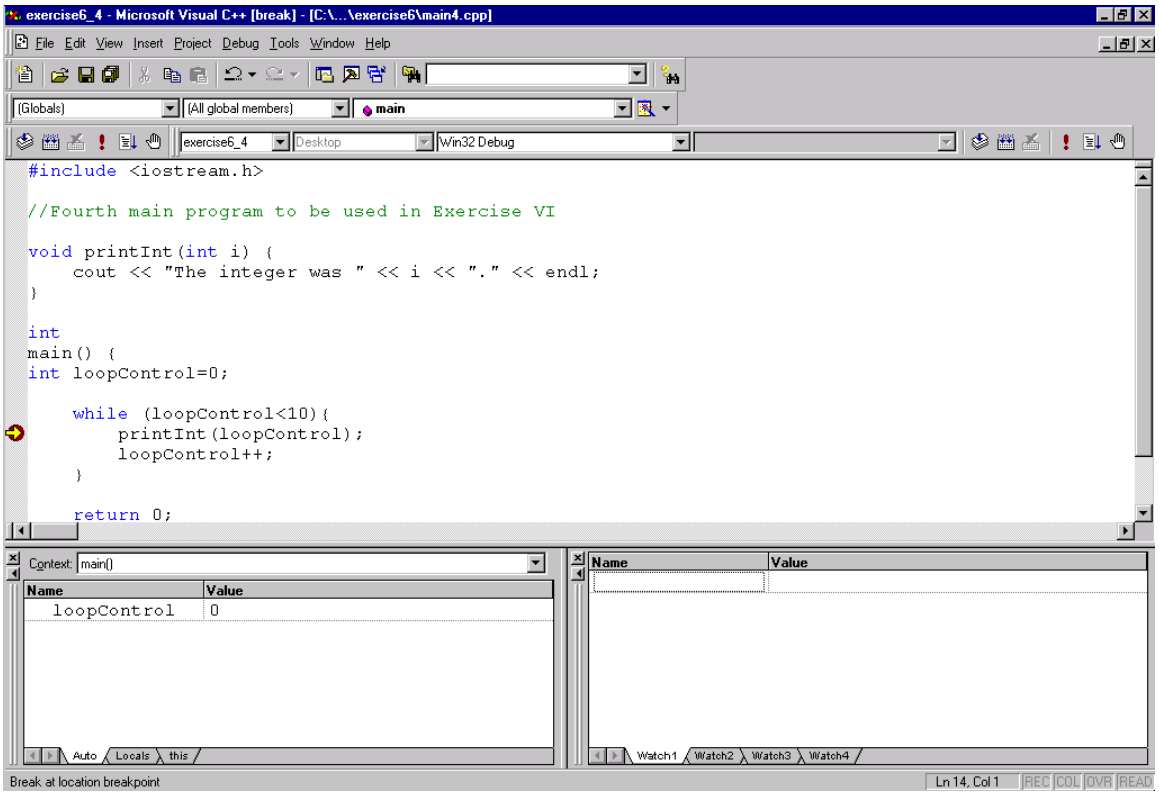

Figure VI.13

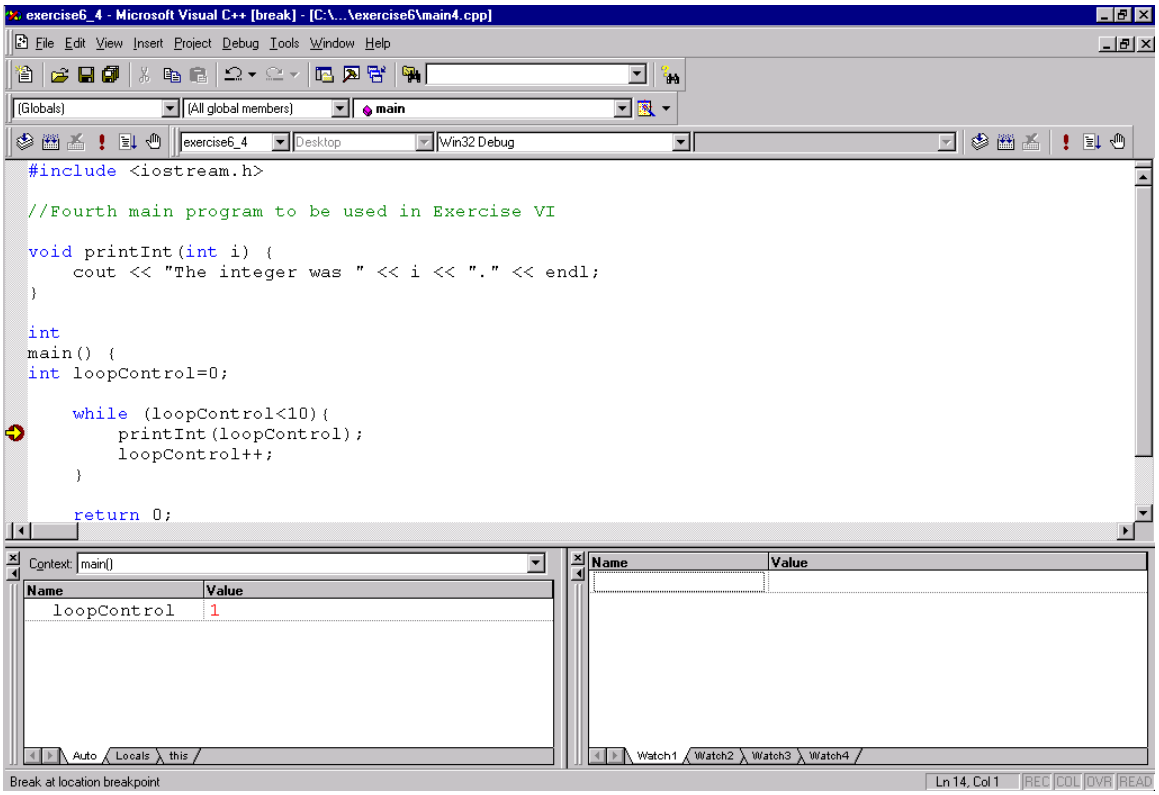

Figure VI.14

At this point, you should be in the editing environment of Visual C++. Start the program running using the debugger again. When you arrive at the first breakpoint, you will be inserting a new watch in the watch panel in the lower right-hand corner of your screen. The watch panel should have four tabs labeled **Watch 1** through **Watch 4**. The first thing you will add is a watch for the address of the variable named **loopControl** in **Watch 1**. To accomplish this, single click in the empty text entry box at the top of the **Watch 1** page and then type **&loopControl** into that box and press the **enter** key. After doing this, your watch panel should appear similar to Figure VI.15. The address shown in the **Value** column is the address of the variable in **virtual memory**. The actual value is dependent on several things, so it might not be the same value as the one shown in the screen shots. However, since in our exercises we will be looking at issues such as whether two variables have the same memory, the exact memory location will not be our direct concern. At this time, please make a note of the address of **loopControl**.

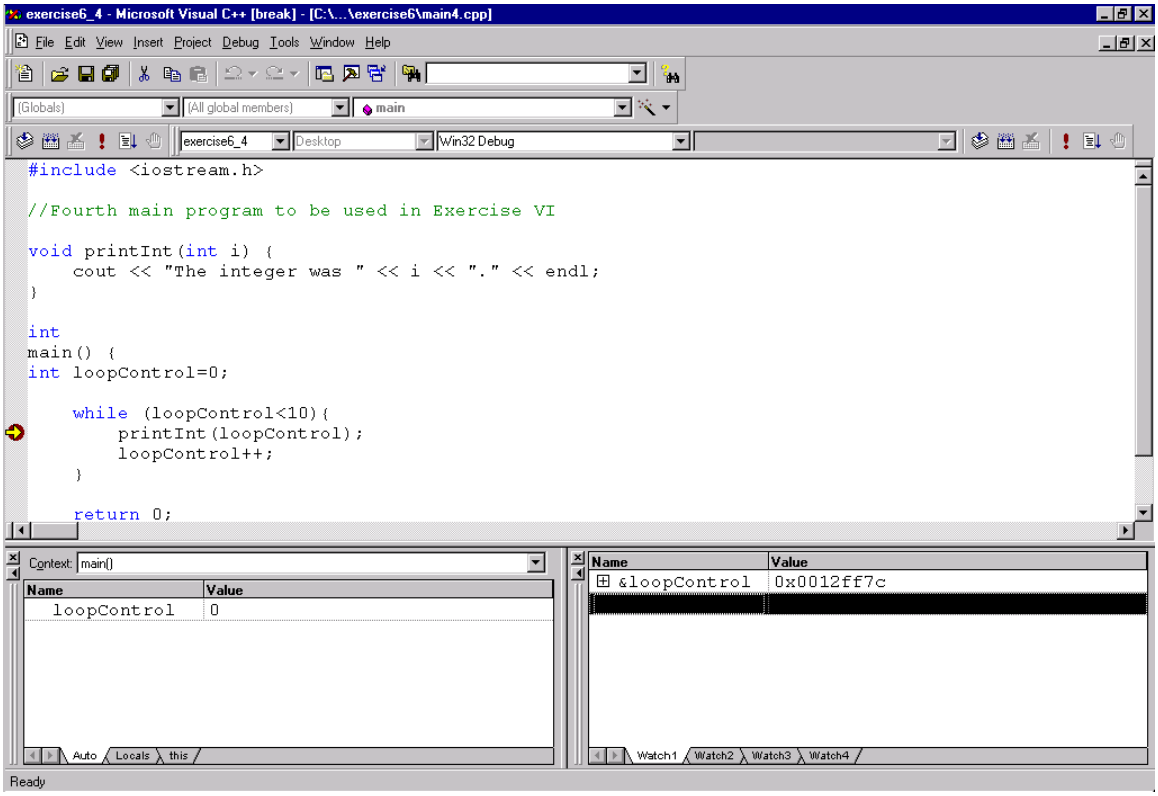

Figure VI.15

Next, use **F11** to step into the execution of this line of code. When you are taken to the **printInt()** function, use **F10** to walk to the **cout** statement. At this point if you look at the **Watch 1** page, you will see an error message in the **Value** column for **loopControl**. This makes sense since the variable **loopControl** does not exist within the scope of this function. At this point, let's look at the address of the local variable **i**. ***Single click*** on the tab labeled **Watch 2** to bring up another page on which watches can be added. Add a watch for **&i** to the **Watch 2** page. Notice that the address of **i** is different than the address of **loopControl**. Since **i** is a *by-value* parameter, this should not come as a surprise.

Now that you have seen how *by-value* parameters look, stop the program using **Shift**+**F5** and edit the **printInt()** function so that the integer is passed in *by-reference*. Now, compile the modified program and start running it using the debugger. When you reach the first breakpoint ***single click*** on the **Watch 1** tab and make a note of the address of **loopControl**. Next, use **F11** to step into the function call once again. Now, ***single click*** on the **Watch 2** tab and look at the address of **i**. Notice that this time it has the same address as **loopControl** does in the main function.

Our final experiment with **main4.cpp** will be to modify the **printInt()** function to be recursive (though silly). Edit your **main4.cpp** so that the **printInt()** function appears as the one in Figure VI.16.

```
void printInt(int i) {
        if (i==0)
                cout << "The integer was " << i << "." << endl;
        else
                printInt(i-1);
}
```

Figure VI.16

After making this change, compile the program again and start running it using the debugger. The first two times the breakpoint is reached, use **F5** to resume the execution of the program. The third time it is reached (**loopControl** will have the value **2**) use **F11** to step into the function call. Now that you have stepped into **printInt()**, use **F10** to step over each line of code until you are returned to the main function. After control returns to the main function, use **F5** to resume execution of the program once again. When it reaches the breakpoint again, use **F11** to step into the function call one more time. This time, when in **printInt()** use **F11** to step into each line of code. By doing this, we can observe recursion in action. Notice that when you get to the recursive call to **printInt()** and use **F11** to step into it, you arrive at the beginning of the function once more. However, if you go to **Watch 2** and look at the address of **i**, it is different each time **printInt()** calls itself. This is exactly what we would expect, since each activation record will have its own local variable called **i**. If you want to step out of the calls into **printInt()** recall that you can use **Shift**+**F11** to do so.

Again, take this opportunity to explore the debugger using this program and do not proceed on to the fifth and final task of this exercise until you are comfortable with stepping **over**, **into** and back **out of** lines of code.

We are now ready to move to our fifth and final task in this exercise. Close the current project and create a new project called **exercise6_5**. Go to your Windows environment and copy the file **main5.cpp** that you extracted from e6.zip into the exercise6_5 directory. Return to the Visual C++ environment, add that file to the project and compile it. With this fifth main function, we are going to observe the execution of a program with a bug intentionally inserted via our debugger. This is a contrived example, so the "expected" output of the program appears in Figure VI.17.

```
Original=0 and Modified=14
Original=14 and Modified=80
Original=80 and Modified=177
```

Figure VI.17

Take a moment to read through the program as given to familiarize yourself with it. Notice that I have placed each **if** and **else** on its own line to assist in observing the execution of the code within the debugger since the debugger will only tell which line (not which part of the line) is being executed.

If you run the program normally, you will see that the output does not match what we expected to have printed. The information printed by the first call to the **doStuff()** function is correct, but the subsequent call leads to errors.

Let us assume that we would like to use the debugger to assist us in finding the problem. At this point, we need to decide where a good place to insert a breakpoint would be. We could insert a breakpoint at the output statement, but by that point, we are past the point where the error has occurred. The **doStuff()** function has several mathematical computations within it, and it is a good guess that the error is in one of these. It might be useful to see which computation is used in the second call to the function, since we know from the output that this is where the problem appears to occur. For this exercise, we will insert a breakpoint at the call to the **doStuff()** function. (As with the previous tasks, you are encouraged to experiment on your own later using this example.)

After inserting the breakpoint at the call to **doStuff()**, your window should appear similar to Figure VI.18.

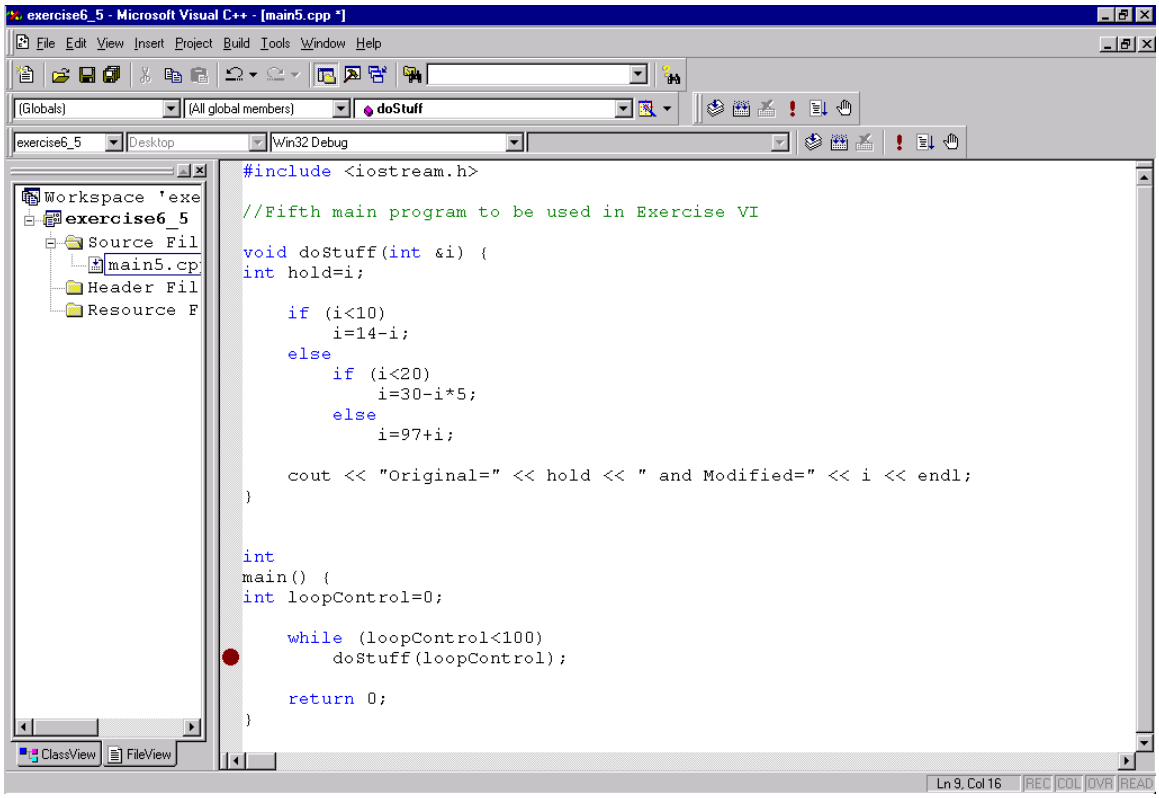

Figure VI.18

Start the program running in the debugger using the **Go** command. When it reaches the breakpoint for the first time, use **F11** to step into the function call. Once inside **doStuff()** use **F10** to step over each line and observe which computation is used. When you get to the output statement, use F5 to resume execution of the program until the next breakpoint is reached. When the breakpoint is reached again, the program is about to enter the **doStuff()** function for the second time – this is where the error occurs. Step into the function and determine which computation is used. You now know the line of code that has the error.

The error on that line is that there should have been parenthesis around **30-i** in the formula. Stop the debugger using **Shift**+**F5** and modify the program to have the correct formula. Now, compile the modified program and run in normally. Notice that when you ran it normally, the program did not pause at the breakpoints – they are only used when the program is run using the **Go** command.

Congratulations! You have now completed your second debugging exercise.

To leave the Visual C++ environment, go to the **FILE** menu and select **Exit**.