Exercise VIII: Debugging IV – Tracing Infinite Loops

By this exercise you should be fully comfortable using the Visual C++ debugging environment. In this exercise, we will be working with a program that has an error that will cause the program to get trapped in infinite loops. We will see a new command within the debugger that will allow us to manually pause a program that is running as well as one that will allow us to run the program up until a specific line of code without needing to insert a breakpoint.

There is a zip file named **e8.zip** at the anonymous FTP server **ftp.cs.umd.edu** in the **/pub/egolub/VC.workbook** directory. Downloaded 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 **exercise8**. Go to the project settings and disable the language extensions as shown in Exercise II. Go to your Windows environment and copy the files that you extracted from e8.zip into the exercise8 directory. Return to the Visual C++ environment and add those files to the project. Compile the program.

The program will request the input of three positive integers. After entering three values, your program should get stuck in an infinite loop! An important question that you might ask at this point is "How can I tell that it is stuck in an infinite loop and not working hard at doing its job?" This question must be answered in part by ones knowledge of the task being performed by the program and how long that task is expected to take to complete. In this exercise, the program will generate some rational numbers and insert them into a set – this should occur quite quickly. However, in reality, it is sometimes difficult to know whether a program is in an infinite loop or just doing a particularly long task. If a program is running for a long time and you are not sure which is the case, you might want to start it running in the debugger. This will enable you to observe the program's state after some time to see whether it appears to be making progress towards an end goal. This exercise is intended to introduce **how** to accomplish this rather than **when** to use the tool.

Before starting the program, familiarize yourself with the **Rational** and **RationalSet** classes and look over the main function briefly to get the general idea of what it is attempting to accomplish. Now, begin running the program normally. Enter **3**, **4** and **5** as the three positive integers. Your screen should now appear similar to Figure VIII.1. It will continue to appear this way. The program is stuck in a loop. You can help confirm this by using your mouse to move the console window around the screen – you will probably notice a very slow refresh rate. Since we began running the program normally, there is nothing to do at this point except to use **Control**+**C** to stop the program's execution. Ideally, we would like to be able to pause the execution and observe the state

of the program, and perhaps walk through some of the execution of the program to help determine what is happening.

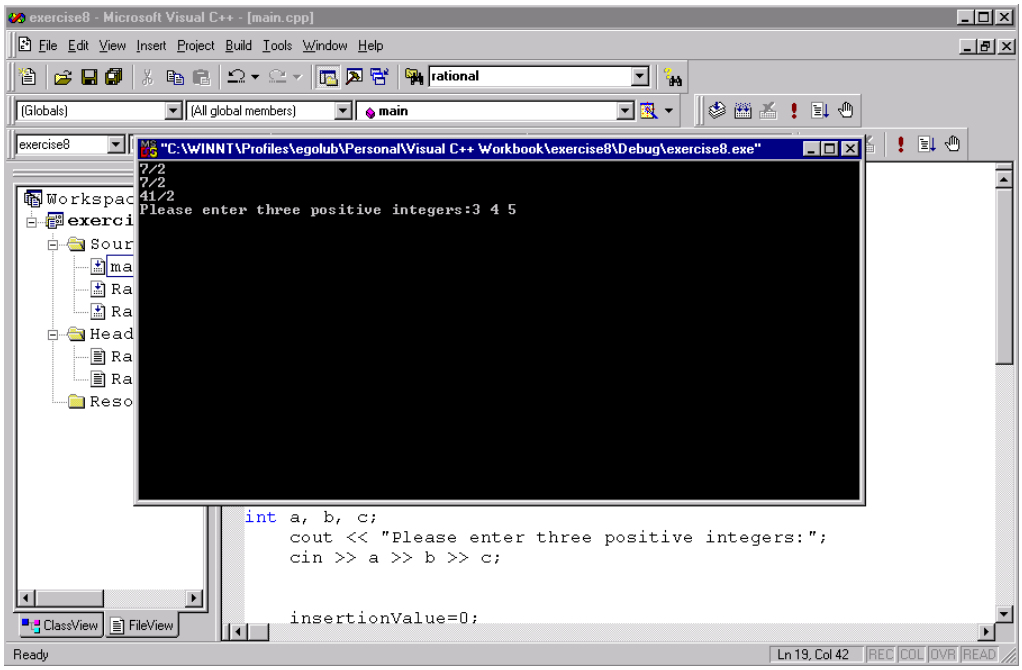

Figure VIII.1

This time, start the program running in the debugger using the **Go** command. Again, enter **3**, **4** and **5** as the values. The program will once again become trapped in a loop. However, this time, your screen will appear slightly differently. In Figure VIII.2, notice that the menu bar of Visual C++ in the background has **DEBUG** listed.

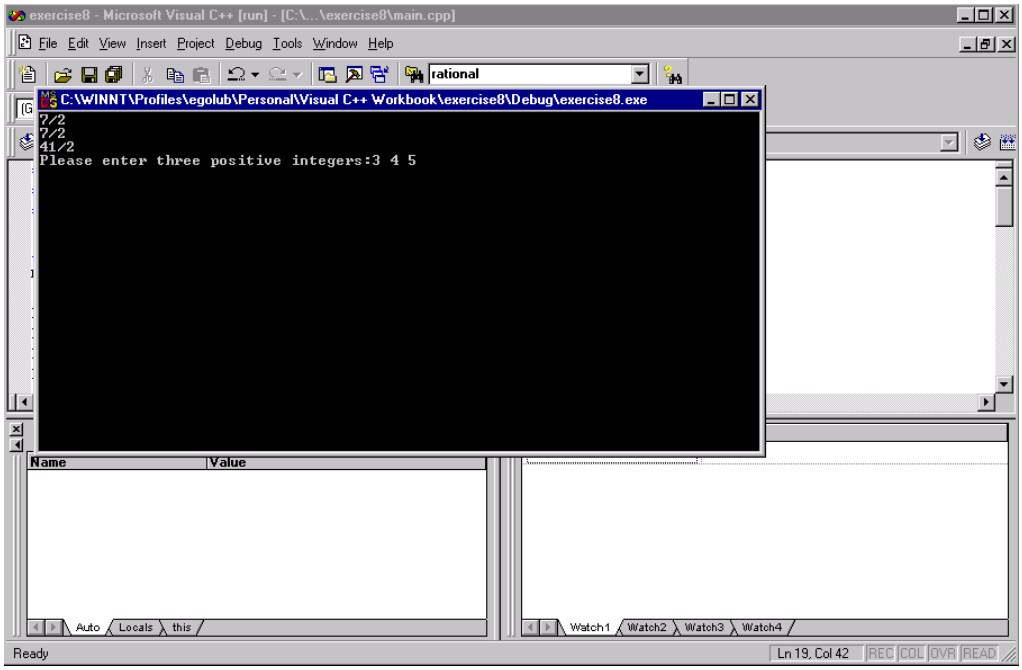

Figure VIII.2

If you bring Visual C++ to the foreground, you can then go to the **DEBUG** menu and select the **Break** option. This will pause the program's execution and allow us to use all of the debugger commands to which we have previously been introduced.

At this point, we can not predict exactly where within the program's execution it will pause. (Note: If Visual C++ asks for the path to a file, use the same technique discussed in Exercise VI in reference to CRT0.C to identify that path.) With the crash in Exercise V and the breakpoints in Exercises VI and VII this workbook was able to walk through the programs in the same way as they appeared on your screen. Here, however, we are unable to do this. Also, since the program may have been executing lower level code at the time you selected to break, you might see something similar to Figure VIII.3, Figure VIII.4 or Figure VIII.5 on-screen. Each of these figures shows one of many different places at which your program could have paused.

Your screen will probably appear different than any of these three, but it should appear similar in nature. The important thing to understand is that you have paused the program in the middle of its execution. If we have paused at a point in the program execution which is not user-written code (eg: **Rational**, **RationalSet** or **main()**), it is usually our goal to return to code that we ourselves wrote, and then walk through the code starting from that point. To accomplish this, go to the **Context** pop-up menu located in the lower left-hand panel. ***Single click*** on the menu to bring it up, and scroll through the activation stack until you see a function that is part of the **Rational** or **RationalSet** class or the main function itself.

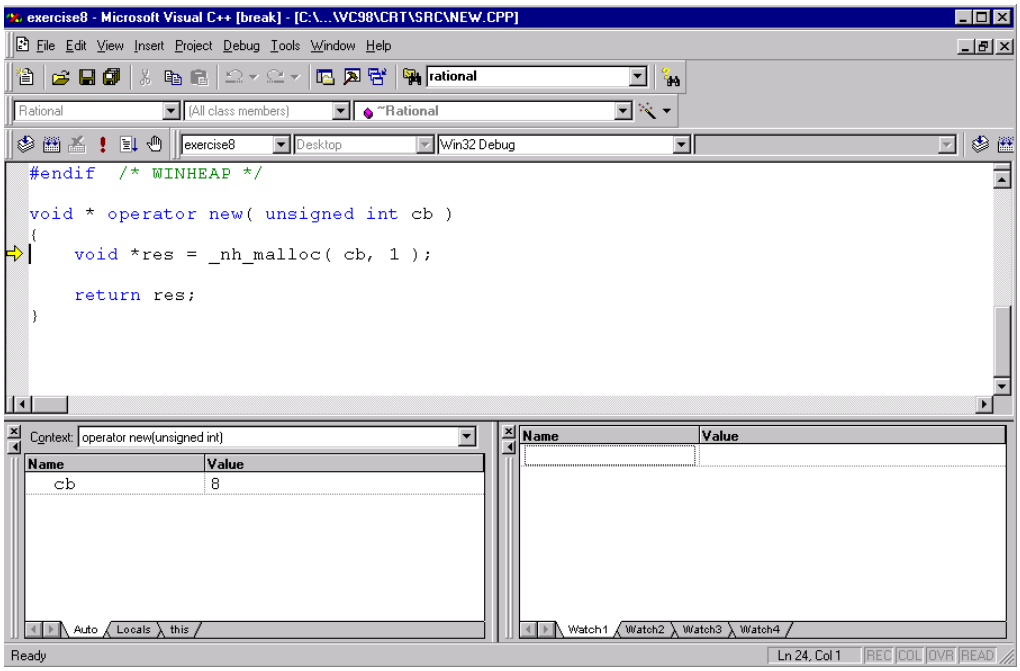

Figure VIII.3

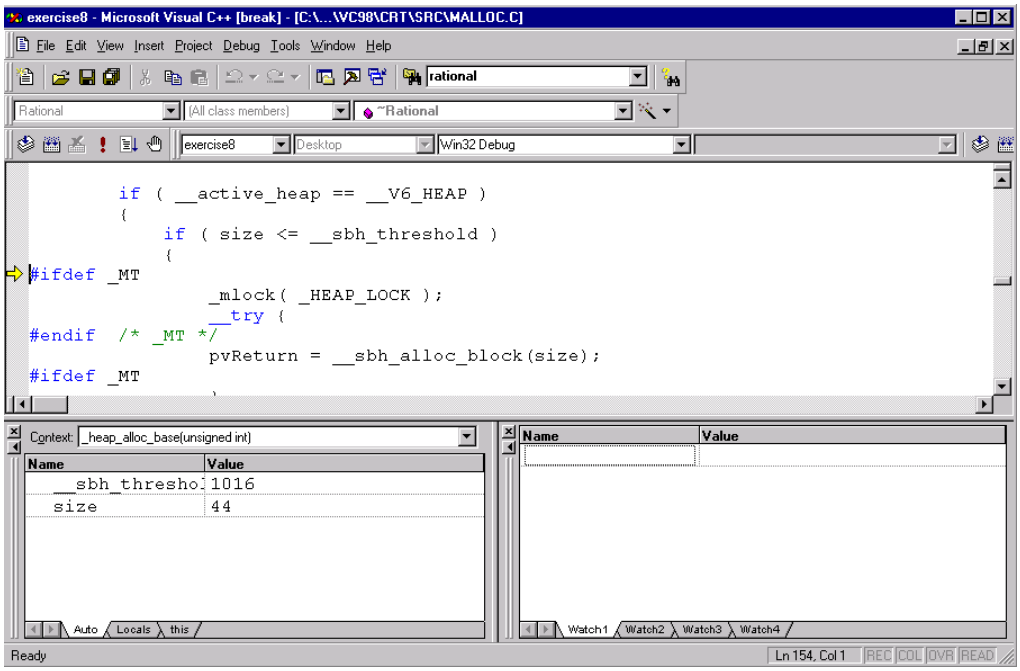

Figure VIII.4

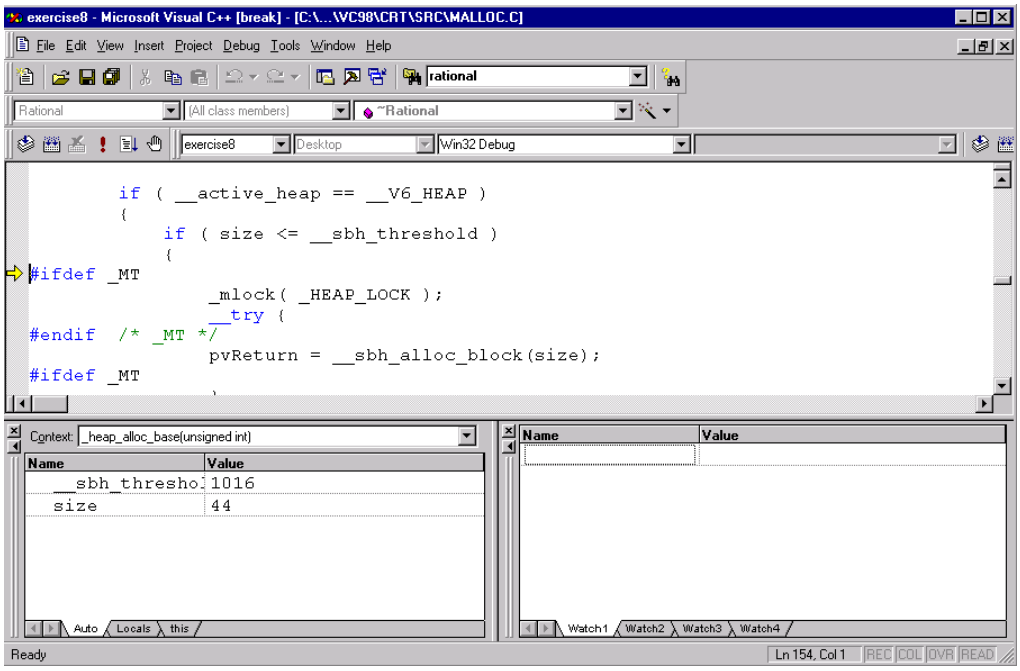

Figure VIII.5

In the example from Figure VIII.5, the activation stack was several layers deep (as shown in Figure VIII.6) but in this case, I can choose to ***single click*** on the entry for **Rational::Rational(int,int)** to bring up that function.

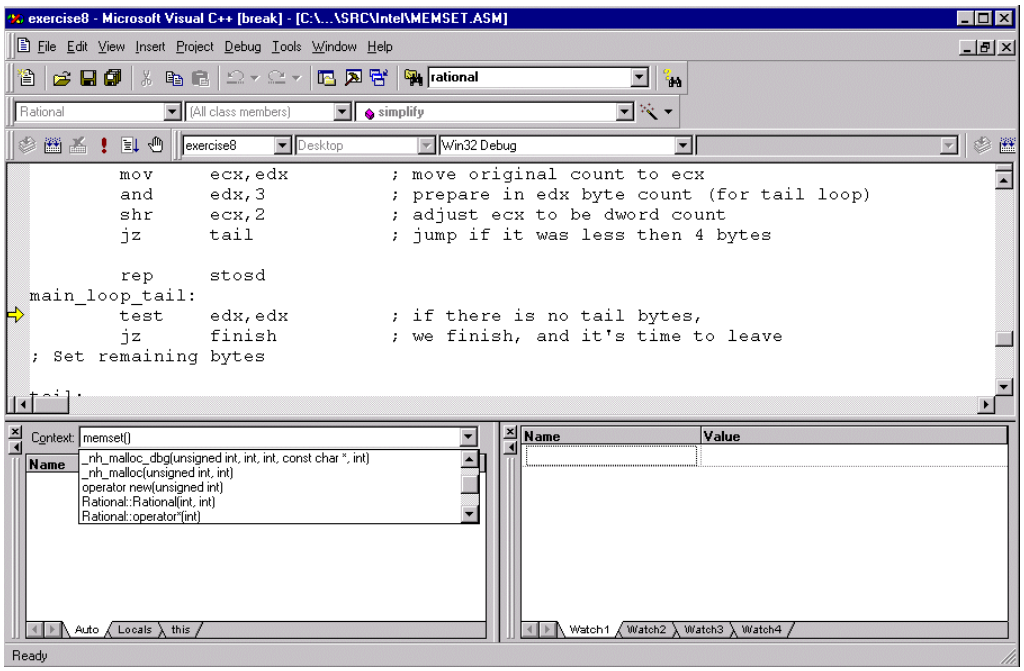

Figure VIII.6

After doing so, the line of code that is in the process of being executed will be indicated with a green triangle (as shown in Figure VIII.7). Occasionally, the green triangle will actually indicate the line of code after the one currently being executed.

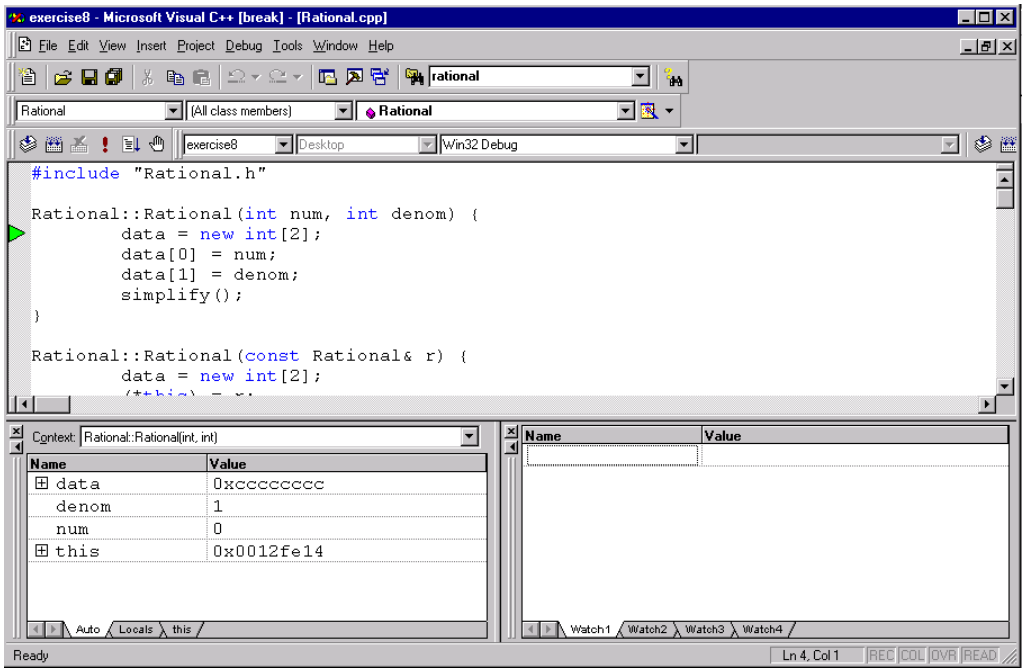

Figure VIII.7

At this point we want to use a new command, **Run to Cursor**, to instruct the debugger to execute the program behind the scenes until it reaches the location at which the cursor is currently located – in this case by the green triangle. This allows us to jump out of how many ever levels down in the code the program had been when we had paused. You can instruct the debugger to **Run to Cursor** in either of the following ways:

- Go to the **DEBUG** menu and select **Run to Cursor**
- While holding down the **Control** key, press the **F10** key

Once you have done this, you can use the commands previously introduced to walk through the code. For this part of the exercise, use **F10** to step over the individual lines of code for a while until you arrive at a point where you appear to be trapped within a loop. You should find yourself trapped in the first **while** loop of the program. Use F10 as many times as required to bring the yellow arrow to the **while** statement itself as shown in Figure VIII.8.

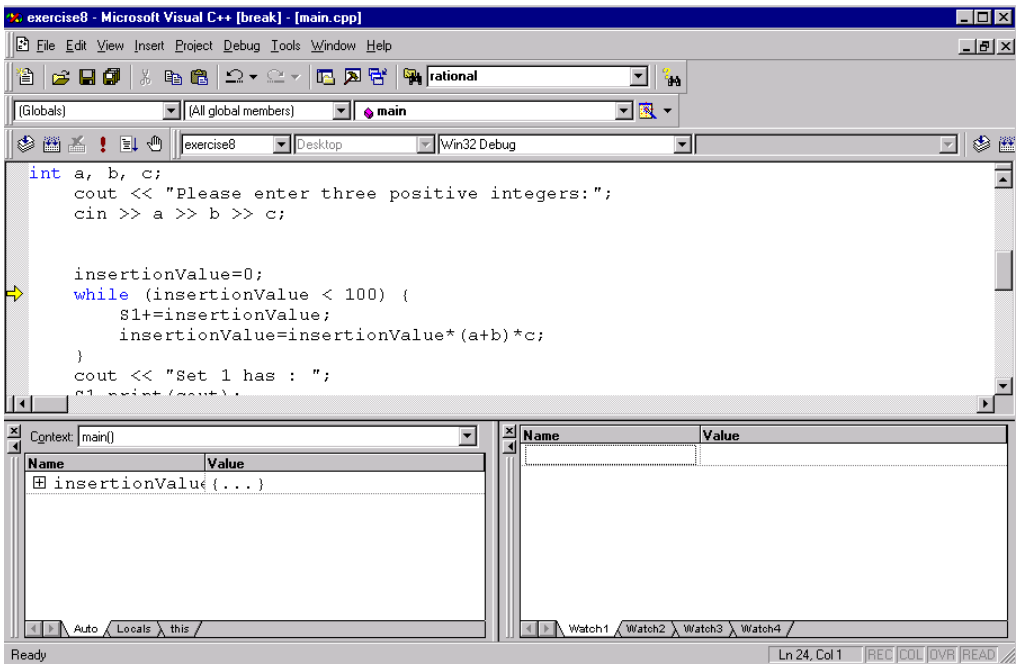

Figure VIII.8

The loop control variable is **insertionValue**, and the loop should terminate once this value is no longer less than 100. From this point, we should walk through this loop several times to observe the behavior of **insertionValue**. One way to accomplish this would be to insert a **cout** statement within the loop, but via the debugger we will want to add **insertionValue** to **Watch 1** and observe its changes. However, since **insertionValue** is an object with several pieces of data, we want to make sure that all of the desired data is shown. In this case, the numerator and denominator of the rational number is stored in a two-element array or integers. In order for us to see both of these values, we will need to add **insertionValue.data[0]** as well as **insertionValue.data[1]** to the watch list. Notice that we are able to ignore the fact that data is a private member when building our watch list. If we had tried to do this with **cout** statements, we would not have been able to directly access the data array. Add those two to the watch list now.

Use **F10** to step over the lines of code within the loop. You should notice something interesting about the value of **insertionValue** – it doesn't change from **0**. This would appear to be the problem. Now that the problem has been identified, we can attempt to correct it. In this case, the problem is that the formula will always stay at **0** if it starts from **0**. We should now change the starting value of **insertionValue** from **0** to **1**. Remember, this is an example that has been contrived to allow us to experiment with the debugger.

Before actually changing the code, it would probably be best to stop the debugger, since we will typically want to start the program over again once we have made our correction. We have already seen the **Shift**+**F5** can be used to stop the debugger. Make your modification and compile the program again.

Once again, start the program running in the debugger using the **Go** command and enter **3**, **4** and **5**. This time, the program will get further before getting trapped in a loop. Figure VIII.9 shows what the contents of the console window should be this time through.

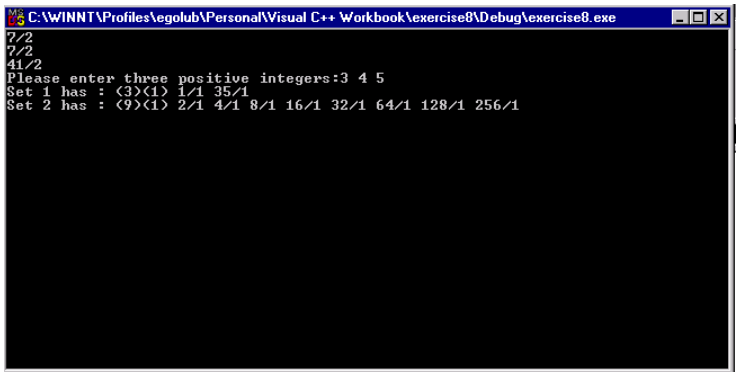

Figure VIII.9

Once again, we can go to the **DEBUG** menu and select **Break** to pause the program and observe its state. As in the previous example, you might find yourself in one of many places in the program. Go to the **Context** pop-up menu and select **main()** to return to the main program. Then **Run to Cursor** and use **F10** to step over lines of code until your window appears similar to Figure VIII.10 with the yellow arrow pointing to the **while** statement.

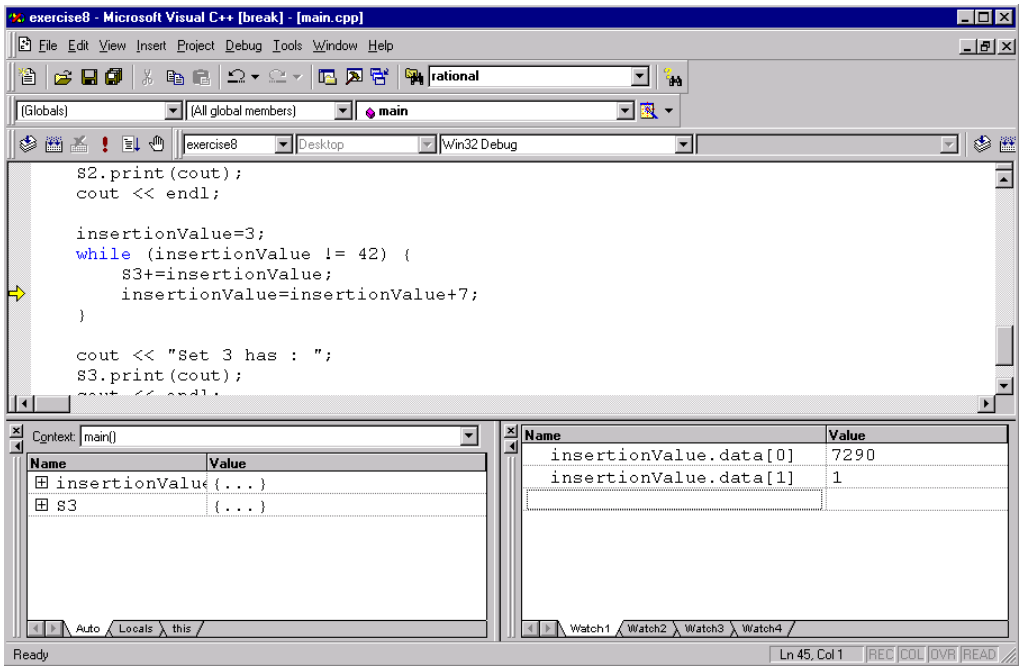

Figure VIII.10

Notice that the while loop is set to terminate when the value of **insertionValue** is equal to **42**. The current value (in Figure VIII.10) for **insertionValue** is **7290**. It would appear that something went wrong with our test. Since we have already gone past the point where the test would have failed, continuing to step through the program would not help us. Instead, stop the debugger and insert a breakpoint on the while loop (shown in Figure VIII.11) and start the program running using the debugger again.

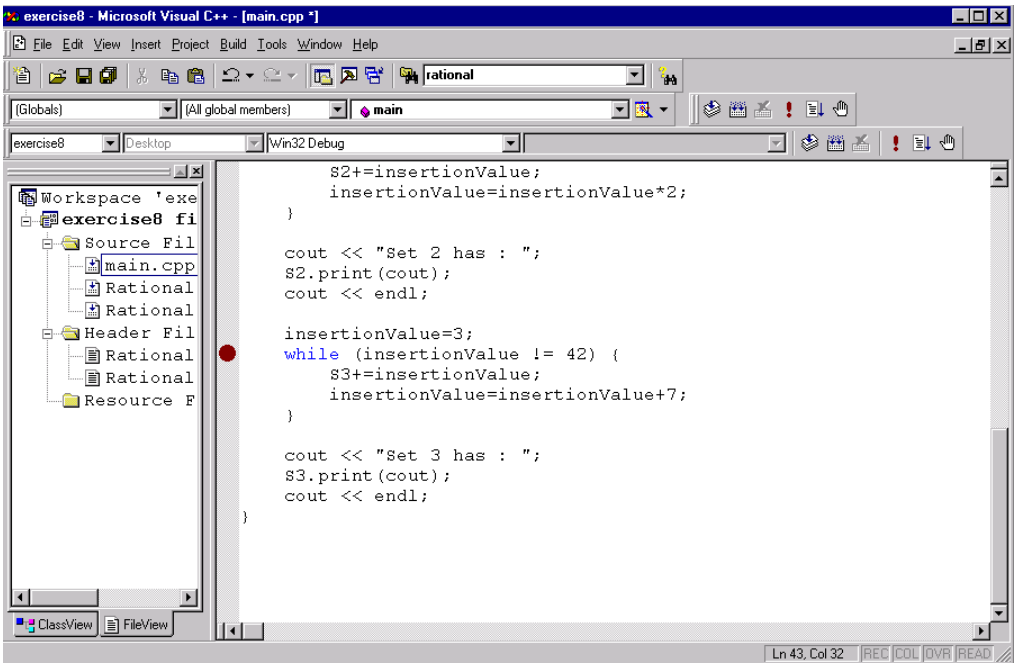

Figure VIII.11

Now, when we get to this loop, we can step over each execution of the loop and observe what happens when **insertionValue** approaches the point at which the loop was meant to terminate. Since the error might be in the **!= operator** of the **Rational** class, we want to observe the value of **insertionValue** in **Watch 1**, so that when it becomes **42**, we can step into the while test rather than over it. However, after stepping through the loop several times, you should notice that the value of **insertionValue** jumped from **38** to **45**. The loop did not terminate because the termination condition was never met.

Again, since we have now determined the cause of our problem, we can stop the debugger and determine how to correct the problem. Let's assume that the correct solution is to change the test from **!=** to **<**. Make that change and run the program once more in the debugger. This time, the program ran to completion.

Congratulations! You have now compiled and executed your fourth and final debugging exercise.

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