Exercise V: Debugging I – When programs crash…

We are probably all familiar with the fact that during program development, programs crash. In this exercise, we will use Visual C++ to explore a program that has an error that will cause the program to crash.

There is a zip file named **e5.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 **exercise5**. 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 e5.zip into the exercise5 directory. Return to the Visual C++ environment and add those files to the project. Take a few minutes to look through the program to get the basic idea of what the program does. Now, compile the program and then run the program. It will crash and you should see two different windows on the screen. The first (Figure V.I) is the console window which has the output of the program. The second (Figure V.II) is an error message dialog box informing you where and why (at a machine level) the program crashed. (Note: Different versions of Microsoft Windows display different messages.)

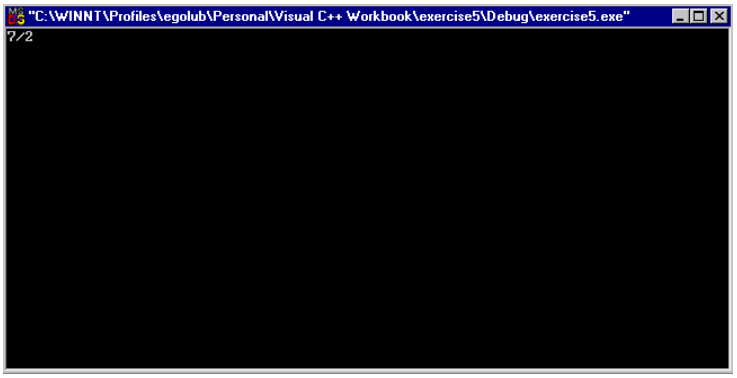

Figure V.1

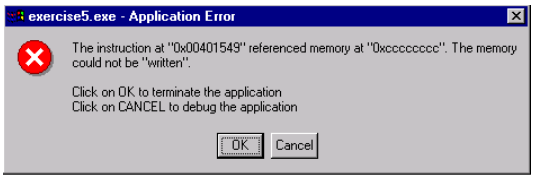

Figure V.2

If you click on the **Cancel** button (don't do that) another instance of Visual C++ would be launched for debugging. It is better for us if we keep only a single instance of Visual

C++ open for both organization as well as system usage. If you click on the **OK** button, the program will continue to terminate. In this case, the error message dialog box shown in Figure V.III will appear.

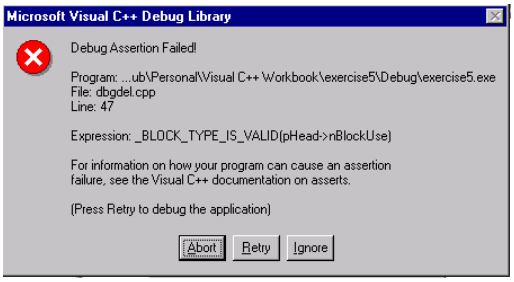

Figure V.3

If you click on the **Abort** button, the program will continue to terminate. That is what we want to do at this time.

Now that the program has terminated, the console window should still be on the screen, with the **Press any key to continue** message displayed. Go ahead and press a key to continue. You should now have been returned to the Visual C++ environment.

We will now execute the program again, but this time we will do so in a manner that executes the program inside the debugger. You can instruct Visual C++ to start the program inside the debugger using the **Go** command in any one of the following ways:

- Go to the **BUILD** menu, go to the **Start Debug** sub-menu and select **Go**
- Press the **F5** key
- Single click on the go button (shown in Figure V.4)

Figure V.4

This time, when the program crashes, you should see an error message dialog box similar to the following:

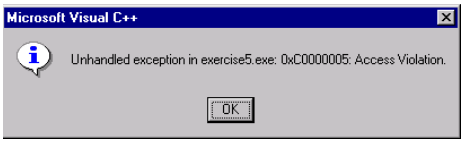

Figure V.5

When you click on the **OK** button, you will be returned to the Visual C++ environment, with the program still active. You are now viewing the debugging environment. It should look similar to Figure V.6 below. Although the program can not proceed any further due to the error, in this situation, you can at least observe the state of the program at the time the error was encountered. This will typically allow you (at a minimum) to

determine the line of code on which the program terminated. It is also often possible to discover quite a bit more about the state of your program, as we will in this exercise.

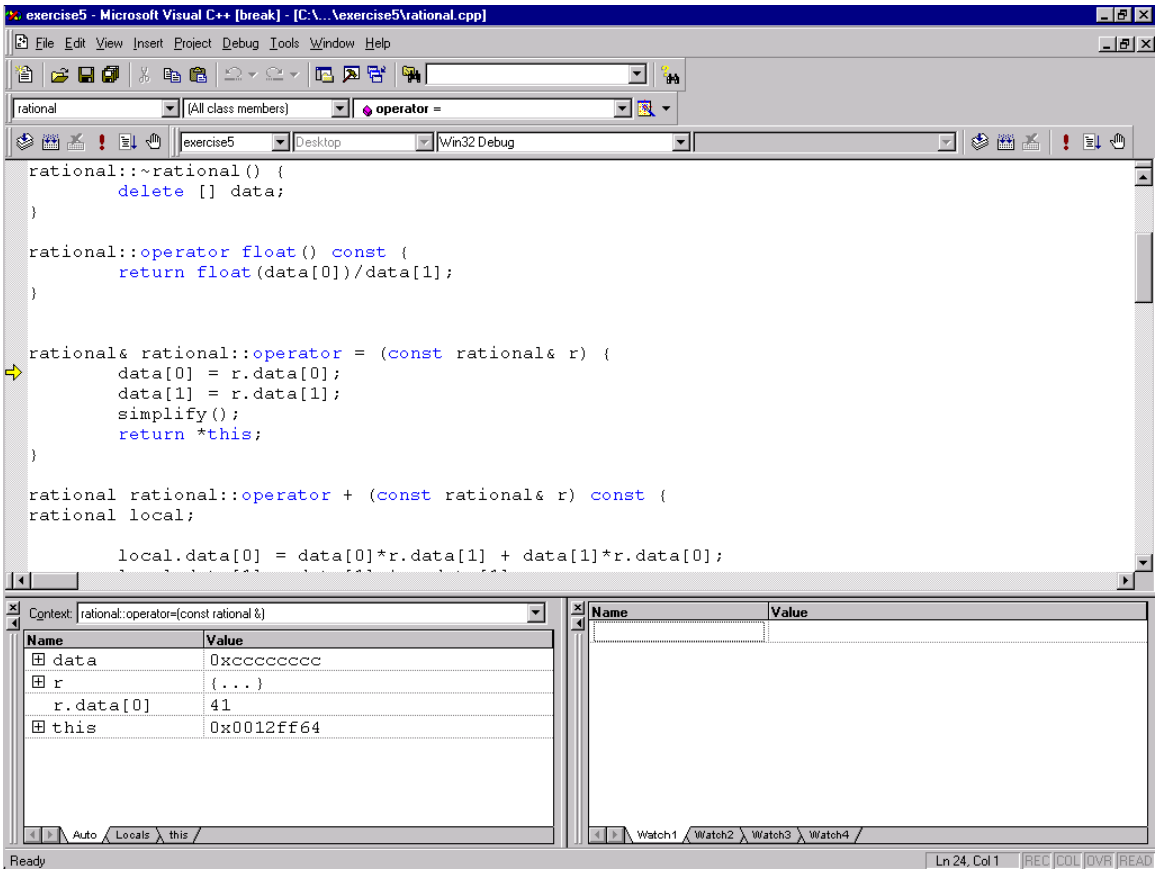

Figure V.6

The yellow arrow indicates the line on which the program encountered the problem. In this case the line **data[0] = r.data[0];** looks perfectly fine. However, the execution of this line of code caused the program to crash. It is often useful to investigate the context in which this line of code was executed. This context includes current variable values as well as the program's function stack.

The lower left-hand corner of your Visual C++ environment should appear similar to the lower left-hand corner of Figure V.6 above. If it does not, go to the **VIEW** menu, then to the **Debug Windows** sub-menu and select **Variables**.

In this panel, the values of variables that are active in the current scope are displayed. There are three different tabs in this panel; **Auto**, **Locals** and **this**. The **Auto** tab represents a page on which Visual C++ will display the variables which it determines are most likely to be of interest at the current time. This will typically be a list of any variables that are being accessed by the current line of code. Notice that variables which are pointers (such as **this** and **data** in the current method) display the address to which they are pointing. Recall that **data** is a pointer, since we are using an **int** pointer to indicate a dynamically allocated array of **int**s. To view the information at that address,

one can click on the + sign to expand the information to be shown. However, just looking at the actual address can be useful. For example, if the address is **0x00000000** then you can tell that it is a NULL pointer. Something else worth knowing is that if a pointer's value is **0xcccccccc**, then it is a good guess that the pointer has never had its value initialized to something other than the default value a new pointer is assigned. In this example, **data** has such a value, implying that it was not initialized before we attempted to use it in this line of code. This implies to us that the problem may not be in this line of code. It might instead be in the lines previous to it, or possibly the way in which this method has been used. Since this is an overloaded = operator, it is assumed that both the left-hand side and right-hand side operands are valid objects. The fact that the data member of the current object has not been initialized implies that the left-hand side operand was not properly initialized.

From within the = operator, we can not know much about the left-hand operand which invoked the = operator. We would need to see the line of code that invoked the operator. When a program is running, each time a function or method is invoked, it is placed on the top of the program stack and executes. When it is done, it is popped off of that stack, and control is returned to the previous item on the stack at the place where it had invoked the routine that just terminated. We want to move up one level in the program stack to see how the = operator was invoked.

There is a pop-up list labeled **Context:** that can be used to display the program stack as well as move through that stack. In Figure V.6, we see that the current location is the = operator of the rational class. By viewing the full list, we can determine from where this operator was invoked, as well as from where the function or method that invoked it was itself invoked, all the way back up to our main function. The view this, ***single click*** on the arrow at the right of the box. Figure V.7 shows this list as it appears for the current program.

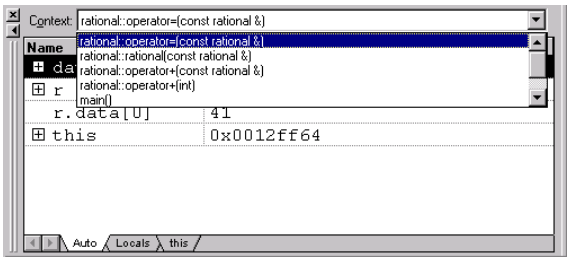

Figure V.7

In this case, the = operator was invoked in the copy constructor of the rational class. We can go to the line of code that invoked the = operator by moving our mouse over the item in the list **rational::rational(const rational&)** and ***single clicking*** on that option. Your screen should now appear similar Figure V.8.

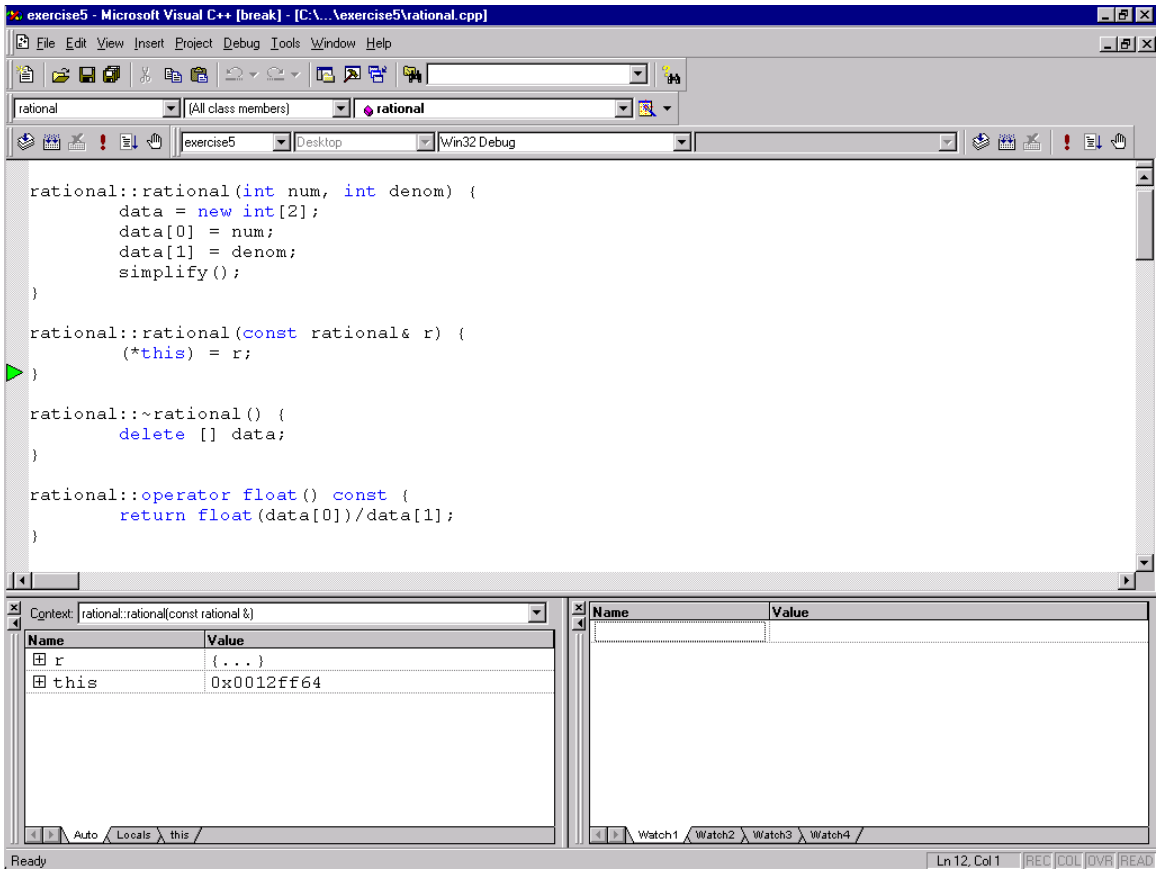

Figure V.8

After selecting **rational::rational(const rational&)** from the context list, Visual C++ displays the file which contains the method, and indicates where control would return to in this routine with a green triangle. This will either be the line of code that executed the call from which we are tracing back, or the line right after it. In this case, the green triangle points to the line after it. The line which invoked the = operator was **(*this) = r;**. Although the green arrow might point to the line which invoked a call or the one after it, you can usually tell which is the case by looking to see which line actually invokes the routine from which you are tracing back (in this case the = operator).

At this point, we have quite a bit of information; the left hand operand of the = operator was not initialized and the line that invoked the = operator is **(*this) = r;** in the copy constructor of the rational class. The left hand operand of **(*this) = r;** is ***this**. In a method of a class, ***this** refers to the current object. In a copy constructor, we are creating a brand new current object. Looking at this code further, we see that nothing is done previous to this call to the = operator, which in turn means that if any components of the current object are dynamically allocated components, they will not have been properly allocated yet. We know that the program terminated when we attempted to access the memory pointed to by the **data** pointer. However, we now also know that the data pointer has not been explicitly set to point to anything. This corresponds to what we had previously observed as a possible problem when we saw that the pointer **data** had the value **0xcccccccc**.

Now that we have determined what the problem most probably was (**data** was not initialized by the copy constructor before we tried to use it in the = operator) we can stop the debugger, go back to looking at the program to determine the correct way to initialize the **data** pointer.

You can instruct Visual C++ to stop the degugger in one of the following ways:

- Go to the **DEBUG** menu, and select **Stop Debugging**
- While holding down the **Shift** key, press the **F5** key

This will return you to the editing environment of Visual C++. The file that you were last viewing will be in the foreground of the editor, and the cursor will be positioned at the same line it had been left at during the debugging session.

We can now look at the program and determine how to correct the error we have now detected. The error was that the dynamically allocated array (**data**) was not initialized by the copy constructor before we tried to actually assign information to that array. This is a common error made by beginning C++ programmers. It is important to remember that the copy constructor will often need to do much of the same initialization as the constructor does. To correct this problem, we can add the line **data = new int[2];** to the beginning of the copy constructor. This allocates the two-dimensional array that we later access.

Add that line of code and recompile the program. After it has been recompiled, you can execute the program again to determine whether that fixed the problem. Since it may not have, you might want to start running the program using the debugger so that if it does crash again, we are already in the debugging environment. There is no harm in running it using the debugger, so even if the program has no problems, there is no loss in doing this.

Adding that line of code should have corrected the problem.

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

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