Exercise VII: Debugging III – Tracing Constructors, Copy Constructors and Destructors

This exercise is intended to introduce the user to the Visual C++ debugging environment via a program that does not have any errors. In Exercise V, we saw that once a program crashes, we can observe things such as the line on which the program has terminated, the values in local variables or the program's stack of activation records. In Exercise VI, we saw that a debugging environment can be used for more than this. A debugging environment can be used to walk through code and observe it as it is being executed. In this exercise, we will use the commands shown in Exercise VI along with an example program to observe how and when constructors, copy constructors and destructors are invoked.

There is a zip file named **e7.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 **exercise7**. 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 e7.zip into the exercise7 directory. Return to the Visual C++ environment and add those files to the project. Compile and run the program. Your output should be the following:

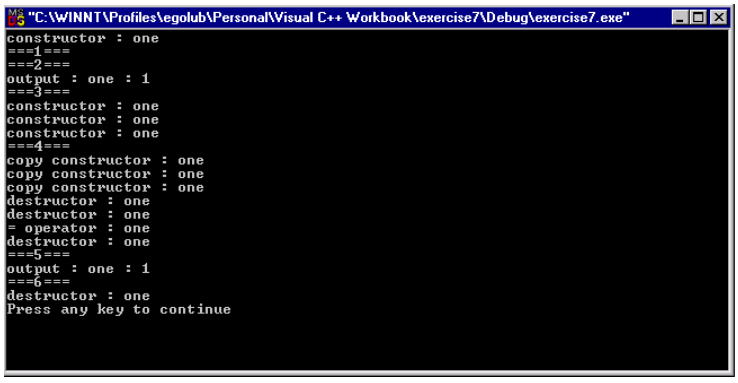

Figure VII.1

This program creates several objects and pointers to objects of type **one**. The type **one** itself is a simple class which serves no purpose other than to print tracing messages as each method (eg: constructor, overloaded = operator) is invoked. In this exercise, we will use the debugger to have our program pause at a specific line of code, and then walk through the execution of that single line of code in great detail. This will give you more practice using the features and functionality of the Visual C++ debugger. It might also

provide more insight into the actual workings of the subject of our debugging exercise – parameter passing and local variables in functions.

The part of the program that we will be looking at in detail in this exercise is the line of code ***B = fun1(A);** in the main program. If you look at the output of the program, this one line of code generates the follow output statements:

```
copy constructor : one
copy constructor : one
copy constructor : one
destructor : one
destructor : one
= operator : one
destructor : one
```

Figure VII.2

Our goal is to have the program pause when it reaches this line of code, and then use the debugger's tools to step into the execution of that line 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. After specifying to Visual C++ to insert the breakpoint, your screen should appear similar to that in Figure VII.3.

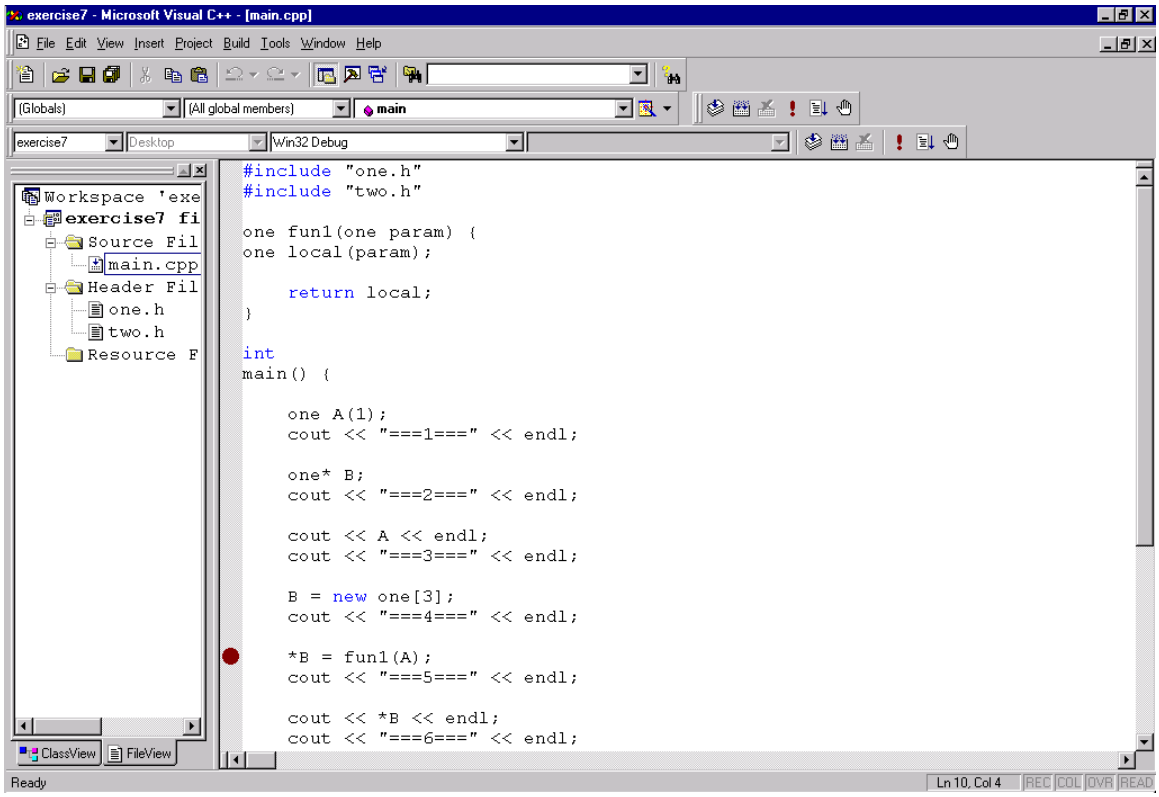

Figure VII.3

Take a few moments to read through the program and familiarize yourself with the main function, the user defined function **fun1()** as well as the class **one**. Notice that part of the main program has been commented out. That is there to use for your own further explorations as desired.

Now, start the program running in the debugger using the **Go** command. The last thing to be printed to the console window when the breakpoint is reached will have been the line **===4===**. After making some observations about the current environment, we will step into the execution of this line of code in great detail. It will be helpful at this point if you take out a piece of paper on which to write some information - one of the things that we will be observing is the order in which objects are created and destroyed. We will observe this using the ability to watch the addresses of individual variables and objects as we walk through the execution of the code.

On the **Watch 1** page, enter **&A** and **B.** This will enable us to make note of the addresses at which the objects of type **one** that exist in this program are being stored. Since **A** is an object, we need to use **&A** to observe the address of that object in memory. Since **B** is a pointer to an array of objects, we only need to use **B** to observe the address of the first array object in memory (since that is what a pointer stores).

After adding these watch values, your window should look similar to Figure VII.4. We will look back to this information while within the scope of the execution of this line of code to see when and how **A** and **B** are being used.

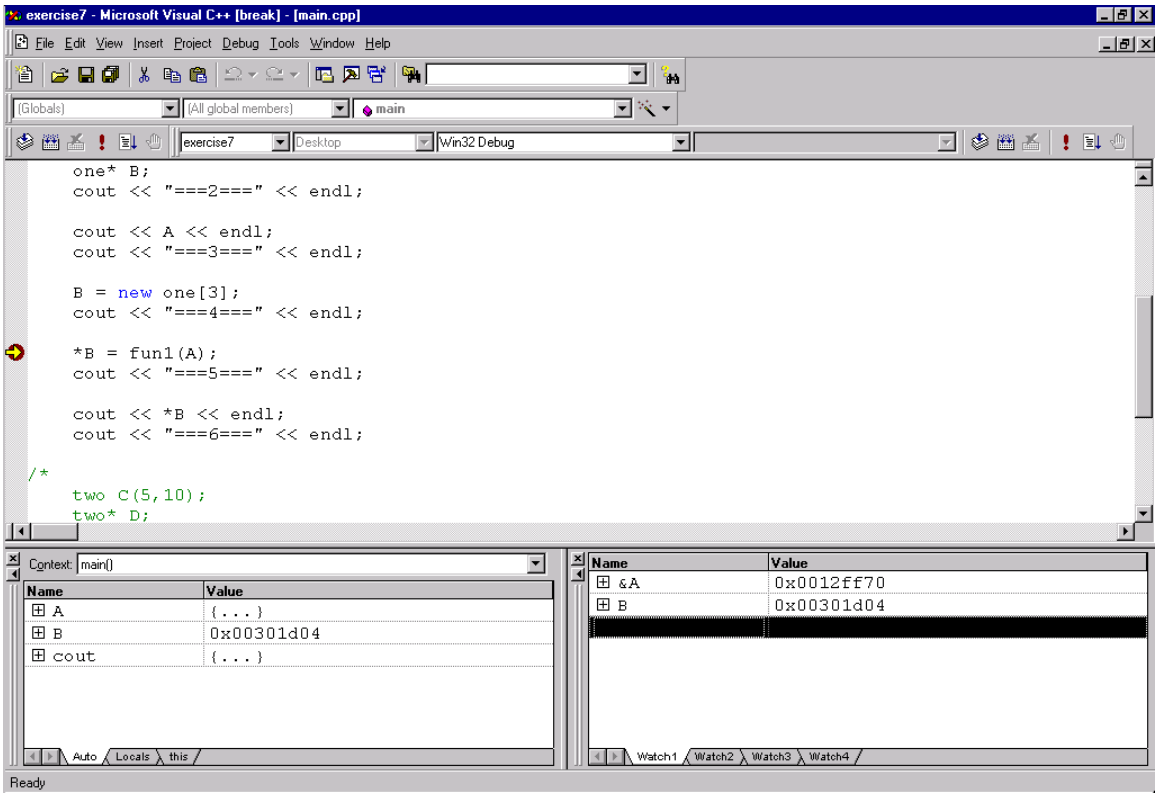

Figure VII.4

At this point, we are ready to step into this line of code. Press **F11** to step into the first function call that this line of code invokes. Figure VII.5 shows that the function into which we step is the copy constructor for the class one. This is because the function **fun1()** takes a *by-value* parameter of type **one**.

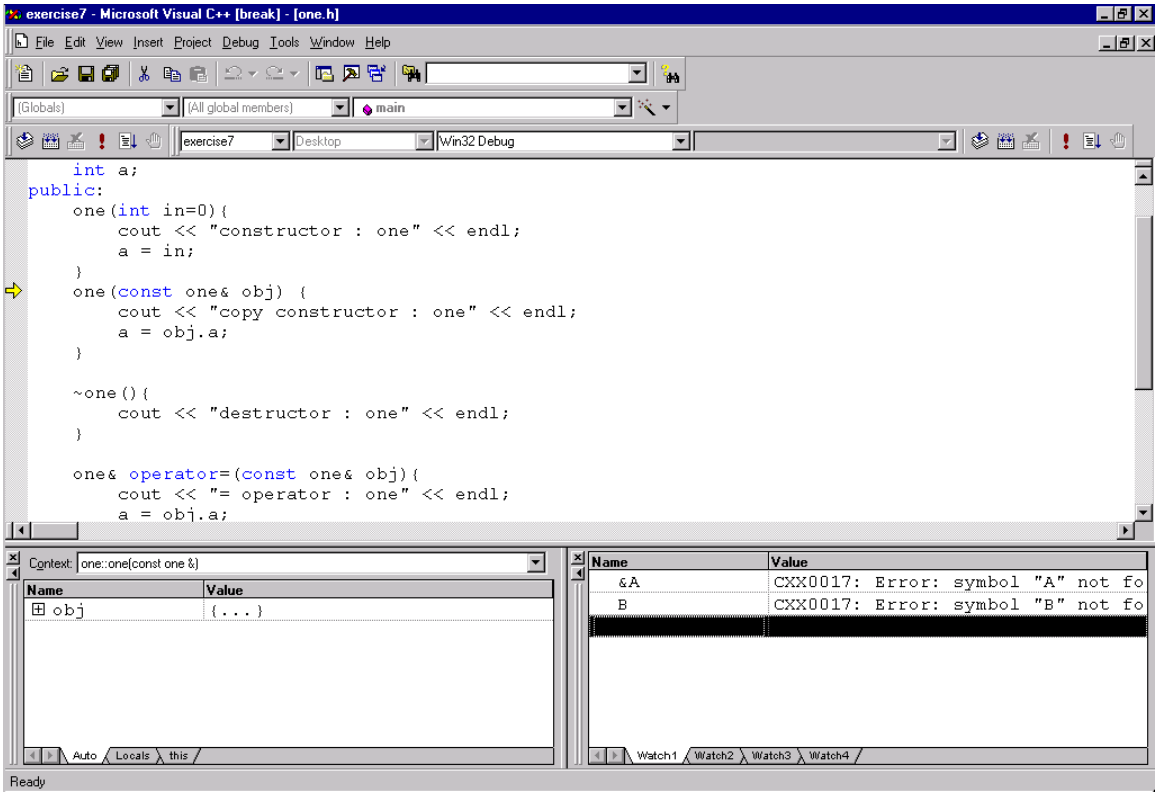

Figure VII.5

Notice that just as we saw in Exercise VI, once variables go out of scope (as **A** and **B** have done) the watch panel shows error messages in the **Value** column. At this point, we know based on our understanding of C++ that the variable within the scope of this member function named **obj** is simply a reference to the object that was passed in (in this case **A**). We can confirm that it is a reference to **A** by observing **obj**'s address. ***Single click*** on the **Watch 2** tab and add a watch for **&obj**. The address shown for **obj** will be the same as the one that had been shown for **A**. Something else that will be useful for us to know is the address of the current object within this member function. To obtain that information, add **this** to the watch list. Notice that since this is a **pointer** to the current object, we do not place an **&** in front of it. If you were to use an **&** in front of it, that would tell you the address of the pointer variable named **this**, rather than the address of the current object. In order for this to have a value, we must be inside the execution of the member function, so press **F11** to step into the function. The address of the current object is different than the address of **A** or **B** – this makes sense since the copy constructor is being called on a newly created object. That newly created object is the local parameter variable for this call to **fun1()**.

Since nothing of interest happens within this copy constructor, we can step back out of it using **Shift**+**F11**. At this point, the debugger returns us to the line of code that we had just stepped into. However, this line of code is still in the process of being executed. To continue stepping into its execution, press **F11** once again. We will now actually step into the call to **fun1()**. This shows us how the copy constructor is called on the *by-value* parameter before the body of the function is jumped to.

If you press **F11** at this point, the debugger will step into the function itself. At this point, single click on the **Watch 3** tab and enter **&param** on the page so that we know the address of the parameter. Notice that the address of **param** is the same as the address of the current object when the copy constructor was previously called. This is because that call to the copy constructor was invoked to initialize this parameter.

At this point, your window should appear similar to Figure VII.6. The line of code that is about to be executed is the declaration of the local variable named **local**. It will be created and initialized with the value in the object **param**. Press **F11** to step into the constructor that is called to initialize this new object.

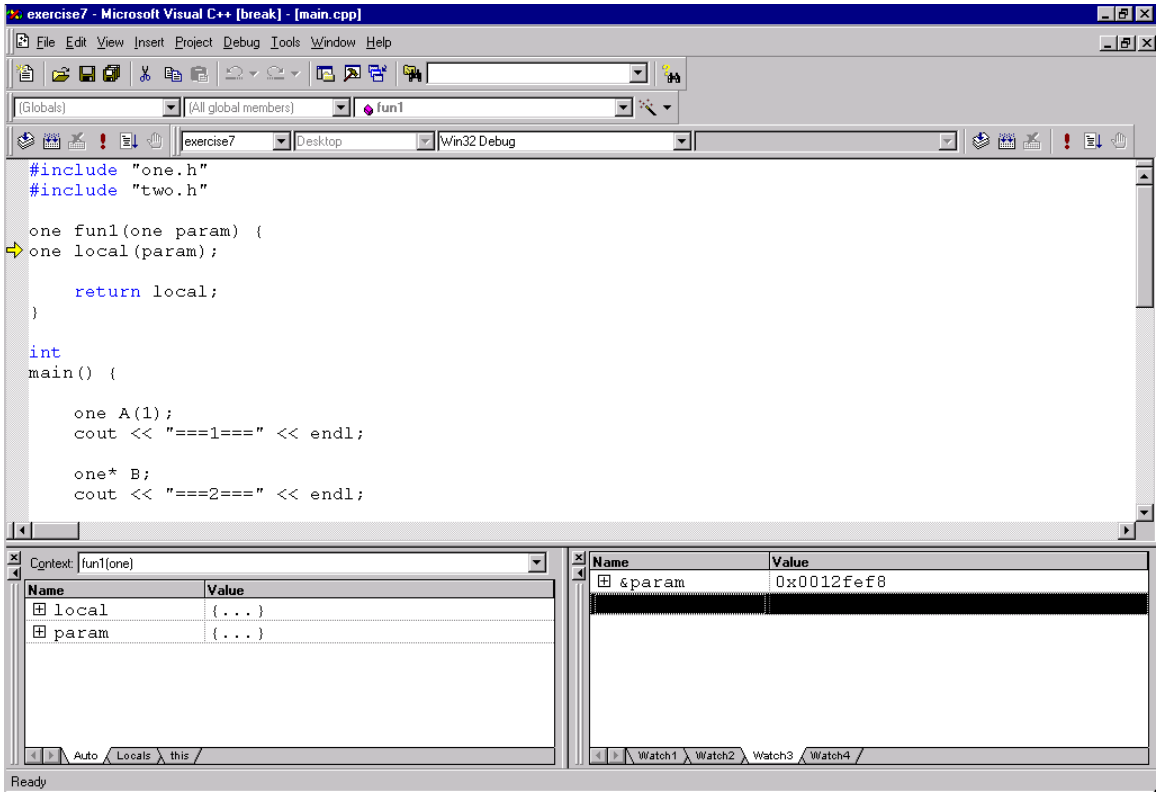

Figure VII.6

Since we gave an object of type **one** as the initial value of this new type **one** object, the initialization of **local** is done by a call to the copy constructor. Press **F11** to step into the body of the copy constructor. The last time we were inside the copy constructor, we set up **Watch 2** to show the address of the current object as well as the parameter. If you single click on the **Watch 2** tab, we can observe those values for the current invocation of the copy constructor. Notice that the address of the parameter **obj** is the address of **param** in **fun1()** and the address of the current object is one we have not seen before. That is because this is a new object, and the object that was passed in to provide the initial value was **param**.

At this point, we can use **Shift**+**F11** to step out of the copy constructor and return to **fun1()**. After stepping out of the copy constructor, the yellow arrow will be pointing to the declaration of **local**. Press **F11** once – since the declaration of **local** is finished, doing so will take us to the next line of code to be executed. That line is the **return** statement. This return statement will have many stages to be executed. Before doing so, note that you have the address of the object **param** as well as the object **local**. Since both of these objects are local to the function **fun1()**, we will soon observe when these objects are destroyed.

Press **F11** to step into the execution of the **return** statement. Doing this takes us into the copy constructor once again. The reason is that the object being returned is being sent *by-value*. This means that an object will be temporarily created to hold this information. This object has no name associated with it, so I will refer to it as **FRED** as needed. If you press **F11**, you will enter the body of the copy constructor, and be able to observe the address of **FRED** in **Watch 2**. Notice that this is another new address and note it down.

Press **Shift**+**F11** to step out of the copy constructor. This will return you to the **return** statement. Press **F11** to continue stepping into the execution of this line of code. Doing so will take you into the first of several calls to the destructor for class **one**. Since non-dynamic objects are destructed in the reverse order of creation, we expect **local** to be destroyed first, followed by **param** followed by **FRED**. To confirm which object is being destroyed, use **F11** to step into the function and look at the value of this in **Watch 2**. As expected, the object being destroyed is **local**. Realize that this is why we do not return local variables *by-reference*.

Press **Shift**+**F11** to step out of the destructor. This will return you once more to the **return** statement. Press F11 to continue stepping in the execution of this line of code. Doing so brings us to another destructor call. This time, the address of the current object is the address of **param**.

Press **Shift**+**F11** to step out of the destructor. This will return you once more to the **return** statement. Press **F11** to continue stepping into the execution of this line of code. Doing so brings us to the end of the function **fun1()** and we are ready to return from this function call. Notice that **FRED** has not been destroyed yet.

Press **F11** to continue stepping through the program. Doing so returns us to the line on which we started this – the call to **fun1()**. Our function has completed its execution now, and **FRED** has been returned. If you press **F11** to continue to step into the execution of this line of code, you will find yourself in the **= operator** of the **one** class. Press **F11** to step into the body of the **= operator**. Since **Watch 2** is displaying the addresses of the current object and an object named **obj** within the current scope, we can use **Watch 2** to determine what the left-hand operand (current object) and right-hand operand (**obj**) to the **= operator** were. As expected, the current object's address is the same as the address which **B** points to in the main function and the address of **obj** is the address of **FRED**.

Press **Shift**+**F11** to step out of the **= operator**. This will return us once again to the line on which we began. If you press **F11** once more to continue the execution of this line of code, the destructor gets called. This is the destruction of the object (**FRED**) that had been temporarily allocated when returning from **fun1()**. Looking at the address of the current address under **Watch 2** will confirm that **FRED** is now being destructed.

If you now press **Shift**+**F11** to step out of the destructor, you will finally move on to the next line of code in the main function.

This program's main function has a wide range of examples to experiment with to become more familiar with the debugger as well as the actions of constructors, copy constructors and destructors in a program. When tracing through a program "for real" it will often be useful (especially in programs with dynamic memory allocation) to be able to use the watch panel to track the objects being accessed.

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

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