

Data Structures and Algorithms

LAB #5

Advanced Debugging Lab

(On Classes and Linked Lists)

Objectives

After this lab, the student should:

- Know how to make the basic debugging in Visual Studio
- Know some advanced debugging notes that make finding code problems easier
- Solve some advanced exercises on debugging of classes and linked lists

What is the Main Purpose of Debugging?

1. It enables us to run our program line by line and trace the values of each variable after executing each line.
2. We can also add breakpoints and make our program run till reaching breakpoints.
3. This helps us in easily finding the causes of runtime and logical errors.

Note: You have to solve all syntax errors first to be able to debug your program.

What are the Main Visual Studio Shortcuts of Debugging?

4. There are some points you need to know before the shortcuts:
 1. **The Debug arrow (the Yellow Arrow):** Points to the line that will be executed next (not executed yet).
 2. **Breakpoint (the Red Circle):** Is a point or mark that can be used later to make the debug arrow executes normally till reaching it. We can add/remove breakpoints by clicking on the left grey column beside the line or by pressing (**F9**) after clicking on the line we want the breakpoint to be beside. We can add as many breakpoints as we want.
5. **Step Over (**F10**)** → Runs the program **line by line** (i.e. one press on F10 makes the program executes one line from the current location of debug arrow and advances the debug arrow one step forward).
6. **Continue (**F5**)** → Runs the program from the current execution location (the location of the **debug arrow**) to the **first breakpoint** on its way.

Note 1: you can start debugging using F10 (the debug arrow location will be in the first main line) or F5 (the program is executed normally from the beginning of the main function and the debug arrow will stop at the first break point on its way).

Note 2: if you pressed F5 and no breakpoints are found from the current location of the debug arrow till the end, the program will finish execution and the console is closed quickly.

7. **Step Into (F11)** → When pressed at a function call, it makes the debug arrow enters the inner code of the function to be able to debug its inner code step by step.

Note: If you pressed F10 in a function call line, this will execute the function normally but without moving the debug arrow inside it which means that the function will be executed all at once and the arrow will return to the function call line ready to move on.

8. **Step Out (Shift + F11)** → When pressed while the debug arrow is inside a function code, it will continue executing the whole function and moves the debug arrow outside the function on the function call line that called this function (ready to continue debugging the lines after the call).

9. **Stop Debugging (Shift + F5)** → Stops debugging.

Note: If you edited the code while debugging, make sure to stop debugging and start it again to be able to see the effect of your edits.

What are the Main Tabs/Windows in Debugging?

Note: If any of the following windows is not displayed, you can view them from: (The "Debug" menu + "Windows")

but you need to start the debugging itself first to be able to find those windows inside the "Windows" of the "Debug" menu.

1. **Locals Window:** Shows the local variables of the *current* function (the function that the debug arrow is currently in) along with their *current* values (their values of the current location of the debug arrow).
2. **Watch Window:** the same as the "*Locals Window*" but you write the name of the variables/expressions you want to watch their values (it does not list all the local variables).

Note 1: if you want to watch the value of a variable or expression, you can either write it in the Watch Window and press enter

OR Highlight this variable or expression in any location in the code then Right Click it and choose "Add Watch".

This will give you the current value of the variable/expression normally depending on the current location of the debug arrow (not on the location where you right clicked and Added Watch)

Note 2: to view the array elements of a dynamically allocated array or an array passed to a function, you need to Watch each element of it separately in the "Watch Window" (arr[0], arr[1], ...etc.)

3. **Call Stack:** Shows the sequence of **function calls** till the current location of the debug arrow *such that*:

- when the arrow **enters** a function call, a row with this function is **pushed** on the top of the call stack for that function
- when the arrow **goes outside** a function call, a row is **popped** from the call stack (the row of the function the arrow just goes outside it).

For example, the call stack shown in the right side, means that:

- **Func_1** calls **Func_2** which in turn calls **Func_3**
- The debug arrow now is still **inside Func_3**
- When the debug arrow goes outside Func_3, its row will be popped from the stack

Func_3
Func_2
Func_1

What are the Magic Debug Values?

- **Magic debug values** are specific values written to memory during allocation or deallocation, so that it will later be possible to tell whether or not they have become **corrupted**, and to make it obvious when values taken from **uninitialized** memory are being used (*the reference is the Wikipedia link below*).
- Here are some special debug addresses assigned to pointers during allocation/deallocation (with examples of runtime errors caused by them):

1. **0xCCCCCCCC**: Used to mark uninitialized stack memory

```
Complex * Arr[3];
Arr[0]->Print();
```

2. **0xCDCDCDCD**: Used to mark uninitialized heap memory

```
Complex ** Arr = new Complex*[3];
Arr[0]->Print();
```

3. **0xFEEEFEEE**: Used to mark freed heap memory

[Advanced]:

```
Complex** Arr[3];
Arr[0] = new Complex*;
delete Arr[0];
(*Arr[0])->Print();
```

4. **0x00000000**: Means **NULL** (address 0 in memory) (we added this here to complete the list of special address values)

- **Complete List (The Reference):**

[https://en.wikipedia.org/wiki/Magic_number_\(programming\)#Magic_debug_values](https://en.wikipedia.org/wiki/Magic_number_(programming)#Magic_debug_values)

Debugging Exercises

Exercise_1 (Debugging on Classes)

Open the following Exercise and apply the steps mentioned below:

 See Code Exercises: Exercise_1

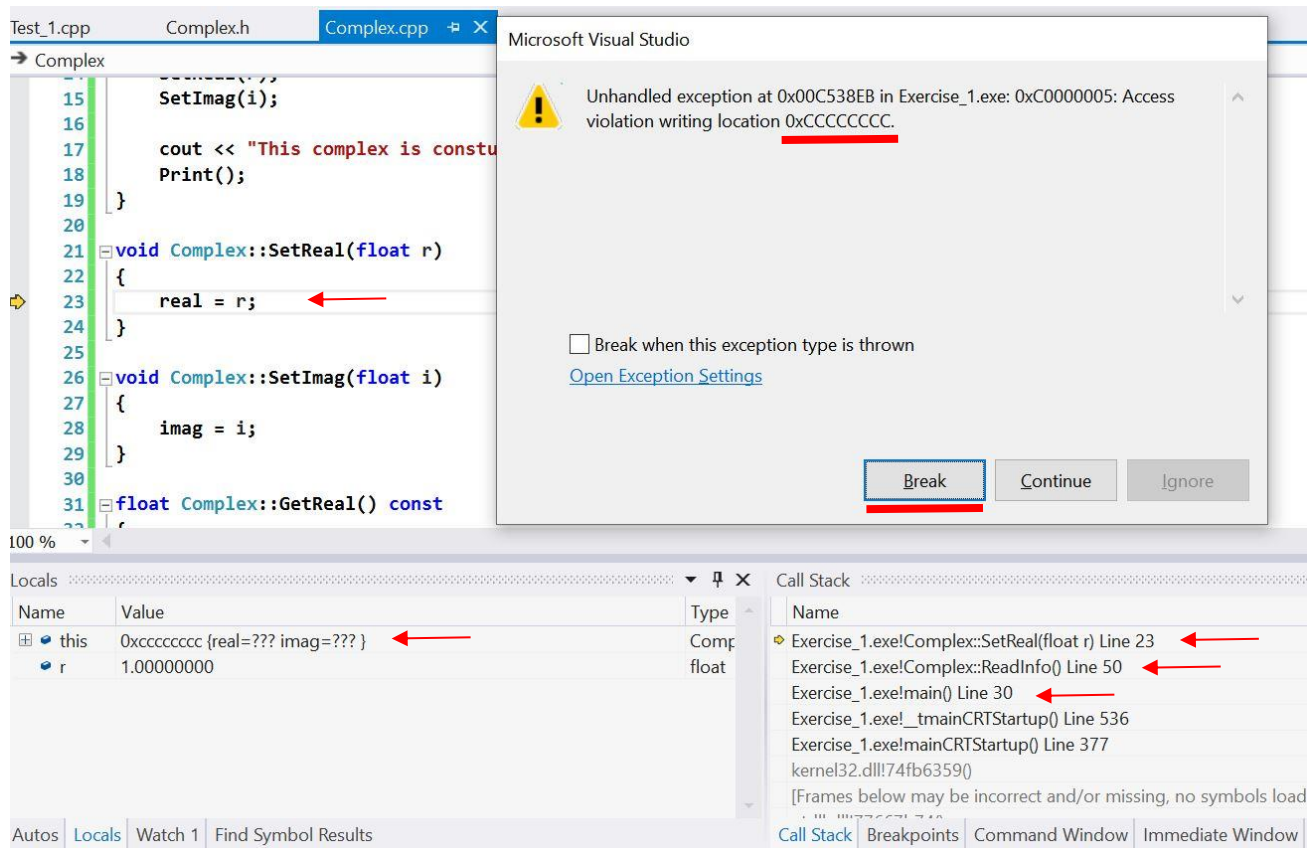
1. Run without debugging using (Ctrl + F5)
2. Try to enter the following complex numbers (real then imag):

1 2
3 4
5 6

Does this run normally? what happened then?

3. Run with debugging using (F5) only (make sure you did not add any breakpoints)

What is the difference between what appeared here and what was with (Ctrl + F5)?



Answer:

- This is a runtime error. Its message says: **"Access violation writing location 0xCCCCCCCC"**.
 - Notice that from the Magic Debug Values, 0xCCCCCCCC means uninitialized pointer, so this gives us a hint on the cause of the problem.
- The **"Yellow Debug arrow"** will appear and point to the line that caused the error when the runtime error occurred.

C. The "Locals" tab, *bottom left*, shows that:

"**this = 0xc0000000 (real=???, imag=???)**, **r = 1**"

- You can notice that the address of the calling object (**this**) equals **0xc0000000** which means that the pointer that calls the current function (SetReal) is NOT initialized.

D. The "Call Stack" tab, *bottom right*, contains set of rows above each other.

- The first row on top contains:
"Exercise_1.exe!Complex::SetReal(float r) Line 23"
- The second row from top contains:
"Exercise_1.exe!Complex::ReadInfo() Line 50"
- The third row from top contains:
"Exercise_1.exe!main() Line 30"

This means that:

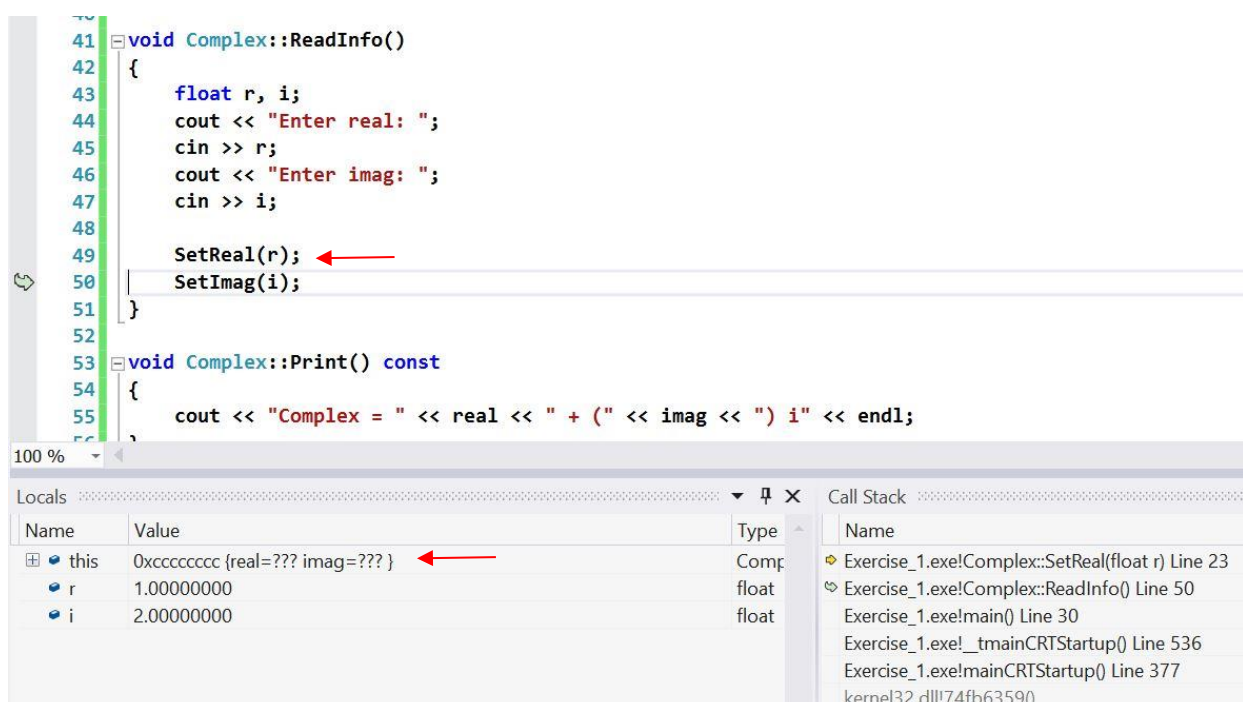
- the **main()** called → **ReadInfo()** before Line 30 of the main function's file
- then **ReadInfo()** called → **SetReal()** inside it before Line 50 of its file
- and now the **debug arrow** is in **Line 23** of the file of **SetReal()** where the program faces the runtime error

4. Click "Break" from the runtime error window.

5. Double click on "ReadInfo ()" line in the call stack

- this will show you the function call line in **ReadInfo ()** which caused that error
- and the "**Locals**" window will contain the values of the local variables of **ReadInfo()** by then, so you can check them too.

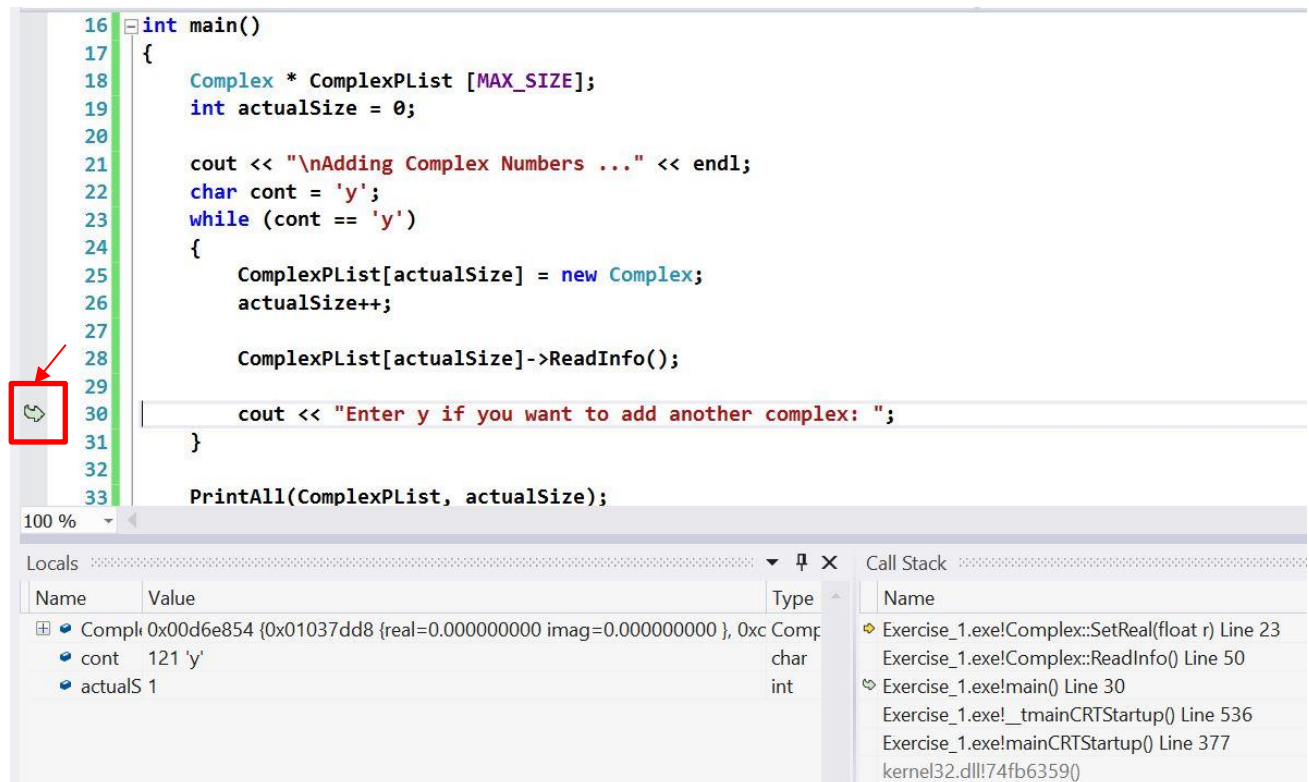
See the screenshot below and notice that you can view here the values of **r** and **i** variable which are local variables inside the **ReadInfo()** function.




6. Repeat the previous one step but on clicking on "main()" line instead.

Similarly, the following screenshot will appear.

Note: the green curved arrow shown below points to the line that should be executed after the return of the function call. Here, it is Line 30 which was written in the Call Stack row beside the main() function.



TO DO:

1. Use debugging to solve all the runtime/logical errors with **Exercise_1**
 - a. A new file is opened by the runtime error of Test 4: `cout << strlen(name);`
How to know the line in “my code” that caused this error?
2. Open **Exercise_2 (Debugging on Linked List)** and use debugging to solve all the runtime/logical errors with it
 See Code Exercises: **Exercise_2**
3. From now on, any runtime/logical problems face you, consider them as extra debugging exercises and try to figure out by yourself using debugging how to solve them.

Good Luck 😊