

Lab 9 - Debugging

1. Find the bug

- ◆ It's tempting to believe a program works if:
 - It runs without error, and
 - Its output seems reasonable.
- ◆ However, to show a program works, you must methodically test it:
 - Test using inputs which should generate outputs which you can predict.
 - Validate the program against these predictions.
Don't just think: *"The output isn't absurd, therefore the program's right."*
- ◆ Download the file `lab9-debugMealCost.cpp`. It has a bug. Can you find it?
 - Start your testing with a few inputs which are easy to check the answers.
Hint: Why? I is my favourite number, I is my second favourite number.
- ◆ **Understanding**
 - What is the incorrect behaviour in the `lab9-debugMealCost.cpp` program?
 - How to carefully test a program is correct.

2. Integrated Debugger

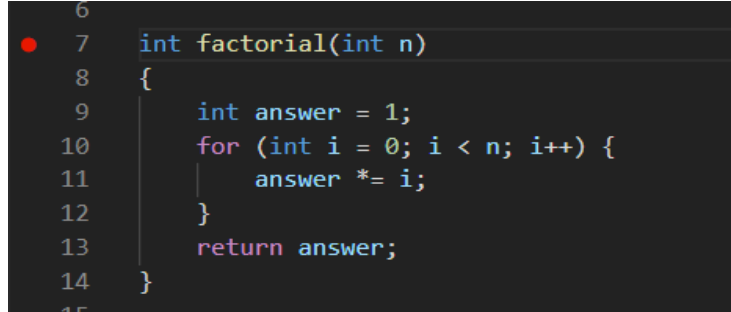
One benefit of using an integrated development environment (IDE) such as VS Code is the integrated graphical debugger. Now, we'll debug a reasonably simple program using the IDE. **The goal is to learn to use the debugger, not to debug the program. Please make an effort to explore with the debugger, even if you can spot the bugs by just looking at the code!**

1. Download `lab9-debugFactorial.cpp`. It contains two functions, each has bugs:
 1. `factorial()` which should, given a positive integer n , will compute $n!$ ("n factorial"). For example, given $n=3$, it should computes $3 * 2 * 1 = 6$.
 2. `findMax()` which should return the maximum number from a vector of integers.
2. Run the code and notice that neither function works correctly.

3. Debug the factorial function:

1. Set a break point on line 7, the first line of the `factorial()` function by clicking to the left of the line number ("7") on the left of the window.

- You'll then see a red circle beside the line.



```
6
7 int factorial(int n)
8 {
9     int answer = 1;
10    for (int i = 0; i < n; i++) {
11        answer *= i;
12    }
13    return answer;
14 }
```

2. Start the program in the debugger:

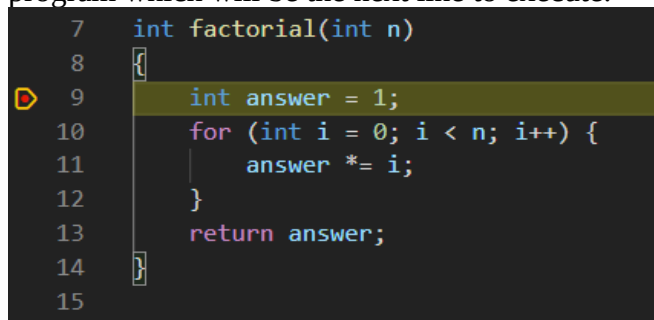
Run → Start Debugging

If asked about what debug configuration you want, select:


- C++ (GDB/LLDB)
- g++ - Build and debug active file

If this does not work, try deleting the file `.vscode/launch.json` from your project and trying again.

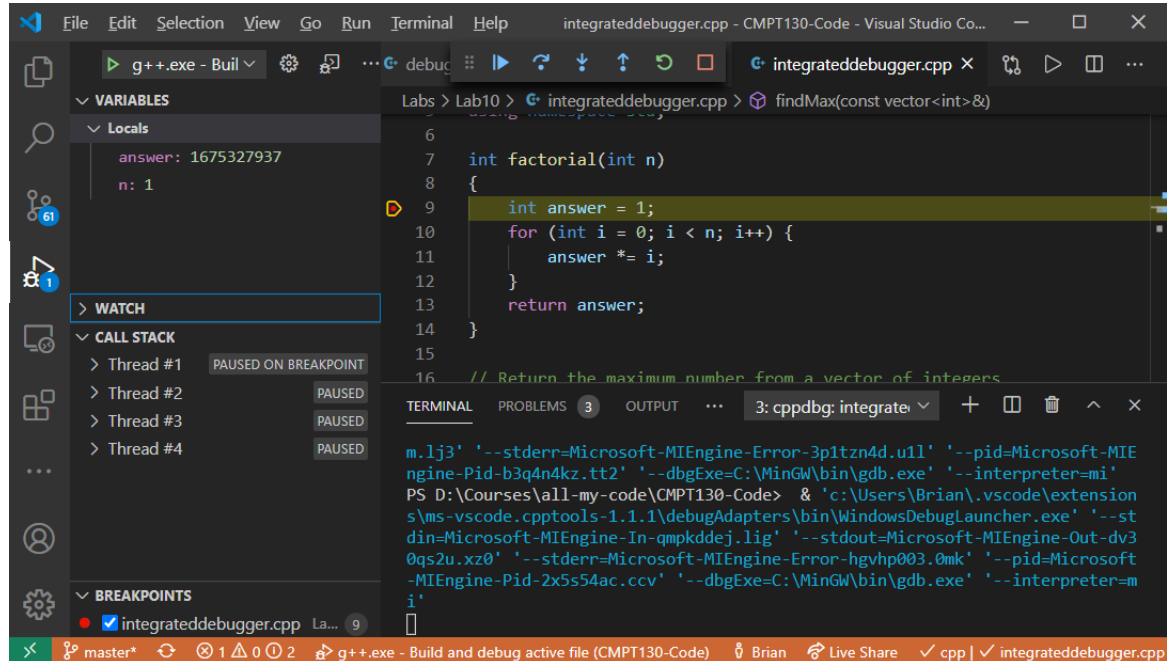
3. You'll now see a yellow bar at line 9 of your code. This shows you the line of the program which will be the next line to execute:



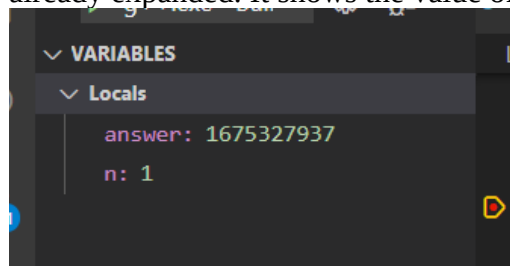
```
7 int factorial(int n)
8 {
9 int answer = 1;
10 for (int i = 0; i < n; i++) {
11     answer *= i;
12 }
13 return answer;
14 }
```

4. If VS Code did not do so automatically, switch to the run view on the left by clicking the icon:  on the very left tool bar.

You should now see



5. You can “step” (or “single step”) your program to execute one line of code at a time:
Step Over: Executes the line of code highlighted in yell. If that line calls a function, just execute the function (without interactively single-stepping through it).
Step Into: Same as step-over, but interactively steps into code in function calls. If the current line does not call another function, step over and step into are the same.
6. Experiment stepping:
 - Step over the current line by pressing its hotkey (look it up on the menu!).
 - Note that the yellow line changes to the next line.
 - Step-over a few times to watch what lines are executed.
7. On the left, notice the Variables section which includes Locals. Expand this if it’s not already expanded. It shows the value of some local variables.



8. Resume the program a couple times.
 - **Continue** (look up hotkey on menu) let the program run until it either finishes or hits a break point.
 - Since `main()` calls `factorial()` multiple times, your break-point in `factorial()` will be encountered for each call. Check the variables window to see the current argument value.
9. **Stop** debugging by either clicking the red square at the top, or Run → Stop Debugging.
 - This will end your current debugging run.
 - NOTE: You will have to stop debugging in order to rebuild your code after a change because debugging is running the program: it prevents the compiler from changing the executable file.
10. Re-run the debugger. It will encounter your breakpoint in `factorial()`.
11. Mouse over any variable in scope to see its current value in a tool-tip.

Watch the values of `answer`, and `i` in `factorial()` to look for bugs while stepping through the code.

Hint: Mouse-over `n` to see the value of the parameter to the function.

Hint: Use the hot key to step over lines of code while watching variables.
12. Correct bugs as you find them.

When you change the code, you'll need to restart the debugging session for the change to take effect.
4. Debug the `factorial()` and `findMax()` functions using the integrated debugger.

The purpose is not fix the bugs, but to learn how to use the integrated debugger!
5. **If using VS Code in Windows without WSL or DevContainer:**
 - The debugger under Windows when used without WSL or DevContainers does not work well with `cin` or `cout` statements.
 - If you want to use the interactive debugger with with a program that would use `cin`, you may need to comment out the `cin` statement and replace it with a hard-coded value for the purposes of tracking down a bug.
 - This is not an issue on Linux, Mac, or WSL, or using Dev Containers.
6. **Understanding:**
 1. What is a breakpoint?
 2. How do you run a single line of a program, one line at a time, to watch it execute?
 3. How can you see the values in a variable as it executes (without having to call `cout` all the time to print them)?
 4. What kind of bug would be best explored with the debugger? A bug known to exist in a small function, like `factorial()` or a bug somewhere in a much bigger program, such as your most recent assignment? Why?

3. Good test values

Many people test their programs as fast as possible in order to show that there are no bugs. **Testing and debugging involve trying to find bugs, not trying to hide from them.**

From the course website, download the file `lab9-debugTestScores.cpp`. This program reads in two test scores and averages them.

Test the program with the following 4 separate test runs. Each test enters a student, his/her scores and then quits (the 'Q' or 'q' on the 2nd row for the name makes the program quit).

	Name	Score 1	Score 2	Expected Outcome
Test 1	Mary Q	80	80	80.0 Program quits.
Test 2	Bill Q	70	80	75.0 Program quits.
Test 3	Tom q	80	90	85.0 Program quits.
Test 4	Sam q	-1, then 1	999 then 99	50.0 Program quits.

Sample output for Test 1:

```
Enter the first name of student 1 (or Q to quit): Mary
Enter score 1: 80
Enter score 2: 80
Mary 80.0
Enter the first name of student 1 (or Q to quit): q
```

Did following these test find any bugs?
No? But, there are many logic errors in the program!

Come up with more tests to try and find bugs.

Each test you do should check for some specific possible failure. It is better to have a few good tests that many many ineffective tests.

(More tests on next page).

Run the three tests below to help you narrow down the bugs. Fix all the bugs you find.

Hint: Use the integrated debugger as much as you can! It may take you a bit longer initially, but it's an exceptionally useful and powerful tool!

- ◆ Notice how each of these tests is targeted at one aspect of the program: each test is trying to break one thing.

	Name	Score 1	Score 2	Purpose	Expected Outcome
Test 1	Mary Bill Tom Q	80 70 80	80 80 91	Handles whole number results (80.0), and decimal results (85.5) And, works with multiple students.	80.0 75.0 85.5 Program quits.
Test 2	Sam Ted Q	-1, then 1 -1 then -2 then 1	101 then 99 200 then 500 then 99	Handles one and more than one bad input. Catches numbers just invalid (101), and way out of range (500).	50.0 50.0 Program quits.
Test 3	Bob q	0	100	Handles values at extreme of valid range.	50.0 Program quits.

Extract a function:

Have a good look at your code. Can you spot a part that would make a good function?

- ◆ *Hint: look for repeat code. Think about how you fixed the bugs in the code; did you have to make the same correction in multiple spots? Could this part of the code become a function?*
- ◆ Extract some (or all) of the repeat code into a function.
 - If you are having troubles with this, start by deciding what this extracted function will do.
 - Then figure out what data it needs to do this (if any). This data goes into the arguments.
 - Then figure out what data it will return (if any).
- ◆ After extracting your function, re-test the code. It should work exactly as it did before, except the code will be cleaner and easier to read!

Understanding

1. Explain why the initial 4 tests did not find any of the bugs.
2. Explain how the second set of tests exercises different parts of the code than the first set.
3. Briefly explain the value of a function for reducing duplicate code. Why can this reduce the number of bugs in a program?

Testing Mentality (not to be submitted)

Pretend that you are given your friend's code and being paid \$100 per bug. How much money can you make? When testing, you are searching for bugs. Try and break it!

4. Lab credit

- Complete above steps in the files. OK to skip any “optional” sections.
- Submit only the corrected & updated last file (`lab9-debugTestScores.cpp`) to CourSys. Ensure you:
 1. correct the bugs, and
 2. in the code, type your answers to section 3’s Understanding questions.
- **You do not need to submit any work from sections 1 or 2.**