Lab 10 - Debugging

Directions

- While completing these labs, you are encouraged to help your classmates and receive as much help as you like. Assignments, however, are individual work. **You may not work on assignments during your lab time.**

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 mealCost.cpp, and add it to your lab project. It has a bug. Can you find it?
   - Start your testing with a few inputs which are easy to check the answers.
     - **Hint:** 0 is my favourite number, 1 is my second favourite number. Why?

   - **Understanding:** How to carefully test a program is correct.

2. Integrated Debugger

   One benefit of using an integrated development environment (IDE) such as CLion 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 integrateddebugger.cpp and add it to your project. It contains two functions, each of which have 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 on line 7 and then going to “Run” --> “Toggle Line Breakpoint”.
      - You’ll then see a red circle beside the line.
      - Alternatively, you could click in the margin to the left of the code to set the breakpoint.

2. “Debug” the project from the menu “Run” → “Debug ‘Lab 10’”.
   You can also use Shift-F9 (debug option on menu will show your project name).
3. You'll now see a blue bar at line 9 of your code. This shows you the line of the program which will be the next line to execute:

4. You can “step” (or “single step”) your program to execute one line of code at a time:
   **Step Over (F8):** Executes the line you are pointing to. If that line calls a function, just execute the function (without interactively single-stepping through it).
   **Step Into (F7):** 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.
5. Step over the current line by pressing F8.
   Note that the blue line changes to the next line.
6. Step-over a few times to watch what lines are executed.
7. **Resume Program** (F9) let the program run until it either finishes or hits a break point.
8. Notice the Variables window at the bottom. It shows the value of some local variables.
9. Resume the program (F9) a couple times. 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.
10. **Stop** debugging by either clicking the red square to the left of the variables window (bottom), or Run --> Stop ‘Build All’.
    - This will end your current debugging run.
11. Re-run the debugger (Shift-F9). It will encounter your breakpoint in `factorial()`.
12. 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 (F8) step over lines of code while watching variables.*
13. 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. **Understanding:**
   1. How do you set a breakpoint on a line of code?
   2. How do you step a program, one line at a time, to watch it execute?
   3. How can you watch 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 in a much bigger program, such as your most recent assignment? Why?
### 3. Good test values

Many people debug their programs as fast as possible in order to show that there are no bugs. **Debugging is trying to find bugs, not trying to hide from them.**

From the course website, download the file `debugging.cpp` and add it to your project. 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 2\textsuperscript{nd} row for the name makes the program quit).

<table>
<thead>
<tr>
<th>Test</th>
<th>Name</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Expected Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>80</td>
<td>80</td>
<td>80.0 Program quits.</td>
</tr>
<tr>
<td>2</td>
<td>Bill</td>
<td>70</td>
<td>80</td>
<td>75.0 Program quits.</td>
</tr>
<tr>
<td>3</td>
<td>Tom</td>
<td>80</td>
<td>90</td>
<td>85.0 Program quits.</td>
</tr>
<tr>
<td>4</td>
<td>Sam</td>
<td>-1, then 1</td>
<td>999 then 99</td>
<td>50.0 Program quits.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Name</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Purpose</th>
<th>Expected Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mary</td>
<td>80</td>
<td>80</td>
<td>Handles whole number results (80.0), and decimal results (85.5)</td>
<td>80.0</td>
</tr>
<tr>
<td></td>
<td>Bill</td>
<td>70</td>
<td>80</td>
<td>And, works with multiple students.</td>
<td>75.0</td>
</tr>
<tr>
<td></td>
<td>Tom</td>
<td>80</td>
<td>91</td>
<td></td>
<td>85.5</td>
</tr>
<tr>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td>Program quits.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test 2</th>
<th>Name</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Purpose</th>
<th>Expected Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sam</td>
<td>-1, then 1</td>
<td>101 then 99</td>
<td>Handles one and more than one bad input. Catches numbers just invalid (101), and way out of range (500).</td>
<td>50.0</td>
</tr>
<tr>
<td></td>
<td>Ted</td>
<td>-1 then -2 then 1</td>
<td>200 then 500 then 99</td>
<td></td>
<td>50.0</td>
</tr>
<tr>
<td></td>
<td>Q</td>
<td></td>
<td></td>
<td>Program quits.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test 3</th>
<th>Name</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Purpose</th>
<th>Expected Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bob</td>
<td>0</td>
<td>100</td>
<td>Handles values at extreme of valid range.</td>
<td>50.0</td>
</tr>
<tr>
<td></td>
<td>q</td>
<td></td>
<td></td>
<td>Program quits.</td>
<td></td>
</tr>
</tbody>
</table>

**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**

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!