

Unit 3c – Debugging Strategies

Mark Redekopp

(Part 2 in a few weeks)

DEBUGGING – PART 1

Bugs

- The original "bug"

9/9

0800 Antan started

1000 . stopped - antan ✓

1300 (033) MP-MC $\left. \begin{matrix} 1.2700 & 9.037847025 \\ 1.582147000 & 9.037846995 \text{ correct} \\ 2.130476415 & 4.615925059(-2) \end{matrix} \right\}$

(033) PRO 2 2.130476415


correct 2.130676415

Relays 6-2 in 033 failed special speed test in relay

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multi-Adder Test.

1545  Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

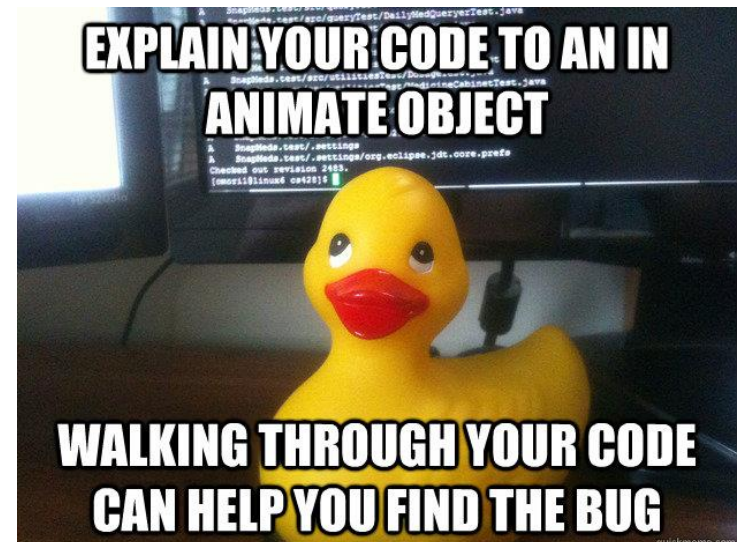
1630 Antan started.

1700 closed down.

Relay 2142
bug 3370

Step 1: Review your Own Code

- **Rubber Duck Debugging:** Reference from an anecdote from a book, "The Pragmatic Programmer", that has become popular
- **Idea:** Explain your code line by line to yourself or some other "object"
 - Note: Commenting your code is a way to do this



Step 2: Test Cases & Expected Outputs

- Determine input test case(s) that will exercise various parts of your code
 - Each if/else block
 - When the loop executes 0, 1, or more times
- Determine the expected output
 - You cannot effectively debug without an expectation of the ***right*** output so you know when the program is working

Step 3: Tracing

- Use one of your input scenarios that is not working and trace the execution of your code by hand
 - Make a table of variables and walk the code line by line

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    int n;
    cout << "Enter an integer:" endl;
    cin >> n;
    bool isPrime;

    isPrime = true;
    for(int i=2; i < sqrt(n); i++){
        if(n % i == 0){
            isPrime = false;
            break;
        }
    }
    if(isPrime){
        cout << "Prime" << endl;
    }
    else {
        cout << "Not prime" << endl;
    }
    return 0;
}
```

i	n	isPr

Step 4: Print Statements / Narration

- Let the computer "trace" for you by using print statements
- Now that you know what to expect, the most common and easy way to find the error is to add print statements that will "narrate" where you are and what the variable values are
- Be a **detective** by narrowing down where the error is
 - Put a print statement in each 'for', 'while', 'if' or 'else' block...this will make sure you are getting to the expected areas of your code
 - Then print variable values so you can see what data your program is producing



Tips

- Don't write the entire program all at once
- Write a small portion, compile and test it
 - Write the code to get the input values, add some couts to print out what you got from the user, and make sure it is what you expect
 - Write a single loop and test it before doing nested loops
- Once one part works, add another part and test it

CODIO DEBUGGER

Debuggers

- Allows you to:
 - Set a breakpoint (the code will run and then stop when it reaches the certain line of code)
 - Step through your code line by line so that you can see where the flow of the program goes
 - Show variable values when you have stopped at a certain line of code.

Starting the Debugger

- In codio, Use the “Debug Current File” on the far right of the top menu bar to launch the debugger targeting the file your cursor is in.
- When you run the program , It highlights the exact line where the code breaks.
- Thus, whenever the code does not run as you expect or want, one of your first steps should be : **Run the debugger**



Breakpoints

- Allows you to specify lines of code where you want the flow to stop and analyse the values of the variables/function calls/etc.
- In codio, you can set it up by clicking on the margin of the code and see a red dot appear at the line of the code.

```
35  
36 int main() {  
37 ● int vals[10] = {9, 7, 14, 20, 2, 8,  
38     cout << "Enter a number: ";  
39     int num;  
40     cin >> num;  
41
```

Commands



Runs the program (until breakpoint). *Note: it will pause temporarily when at a cin statement and wait for you to type the input.*



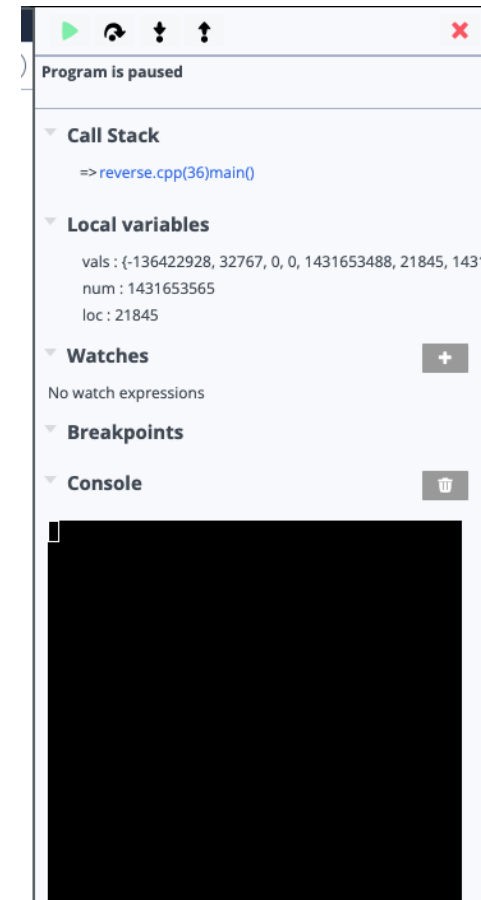
Step Over (Next) : Run the current line and pause at the next line of code.



Finishes (step out of) the current function's code (or hit the breakpoint), and then pause at the next line of the previous function on the stack



Step into : If current line is function, it steps into the function pausing at the first line of the function body. If current line is not a function, it executes the line (like step over)



Showing Values

- We can view the value of local variables which will update as we step through the program

Local variables

```
vals : {-136422928, 32767, 0, 0, 1431653488, 21845, 1431653565}
num  : 1431653565
loc  : 21845
```

Function Call Stack (Not Covered)

- Allows you to see the functions being called and what the system stack looks like.
- For example here, the “main()” function calls “reverseAndFind()”
- We can view the value of local variables in each function by clicking that function in the stack area.

Call Stack

```
=> reverse.cpp(15)reverseAndFind()
reverse.cpp(42)main()
```

Local variables

```
vals : {-136422928, 32767, 0, 0, 1431653488, 21845, 1431653488}
num : 1431653565
loc : 21845
```