CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Debugging
  - Theory
  - `gdb`
  - `valgrind`

Your Goals

- Homework 3
- Check out the debugging assessment info

Debugging basics

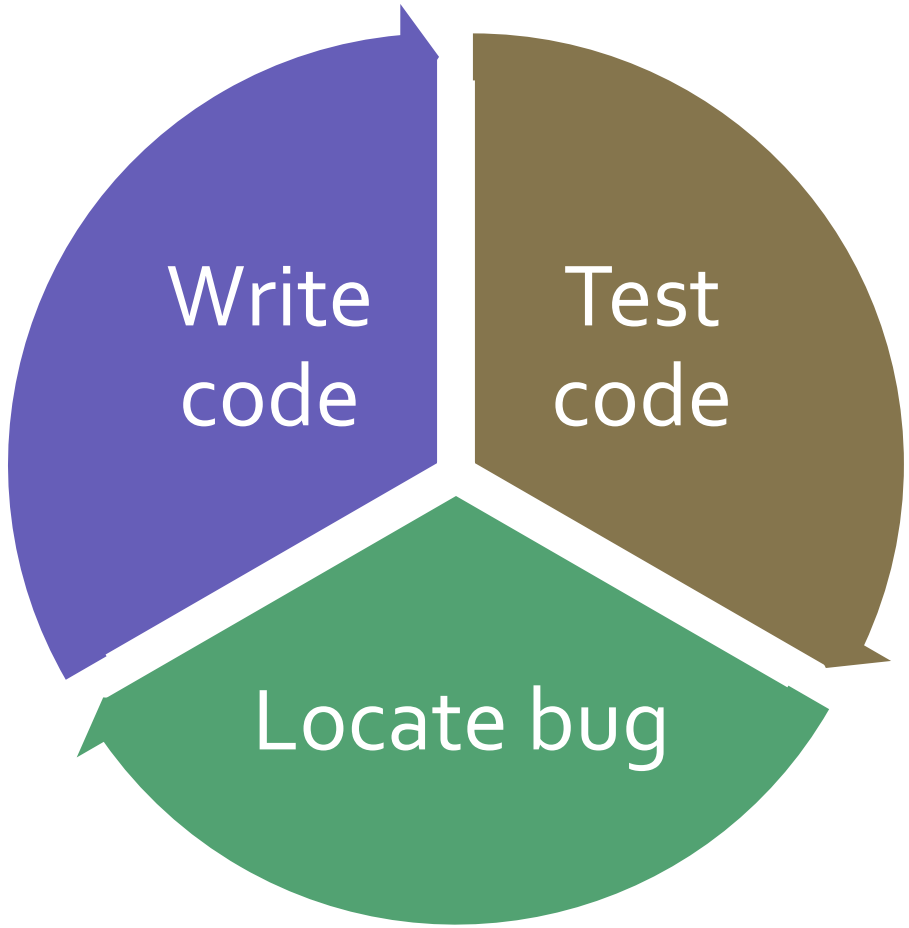
- A bug is a difference between the design of a program and its implementation
  - Definition based on Ko & Meyers (2004)
- Examples of bugs
  - Expected factorial(5) to be 120, but it returned 0
  - Expected program to finish successfully, but crashed and printed "segmentation fault"
  - Expected normal output to be printed, but instead printed strange symbols
- Therefore – 'debugging' is just making your program behave as designed.

Avoiding debugging

Avoid writing code

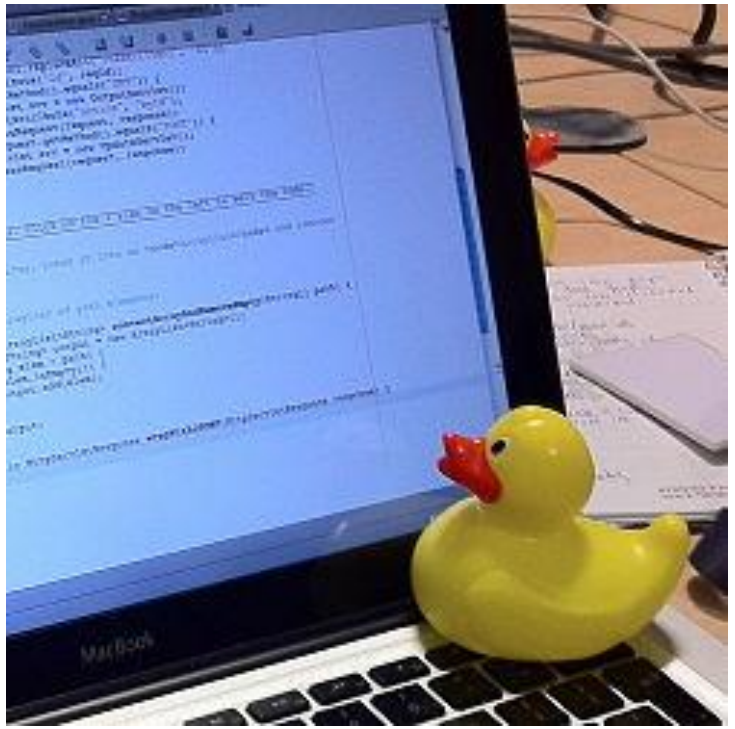
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian Kernighan -- Wrote THE BOOK on C (our book!)



Debugging skills

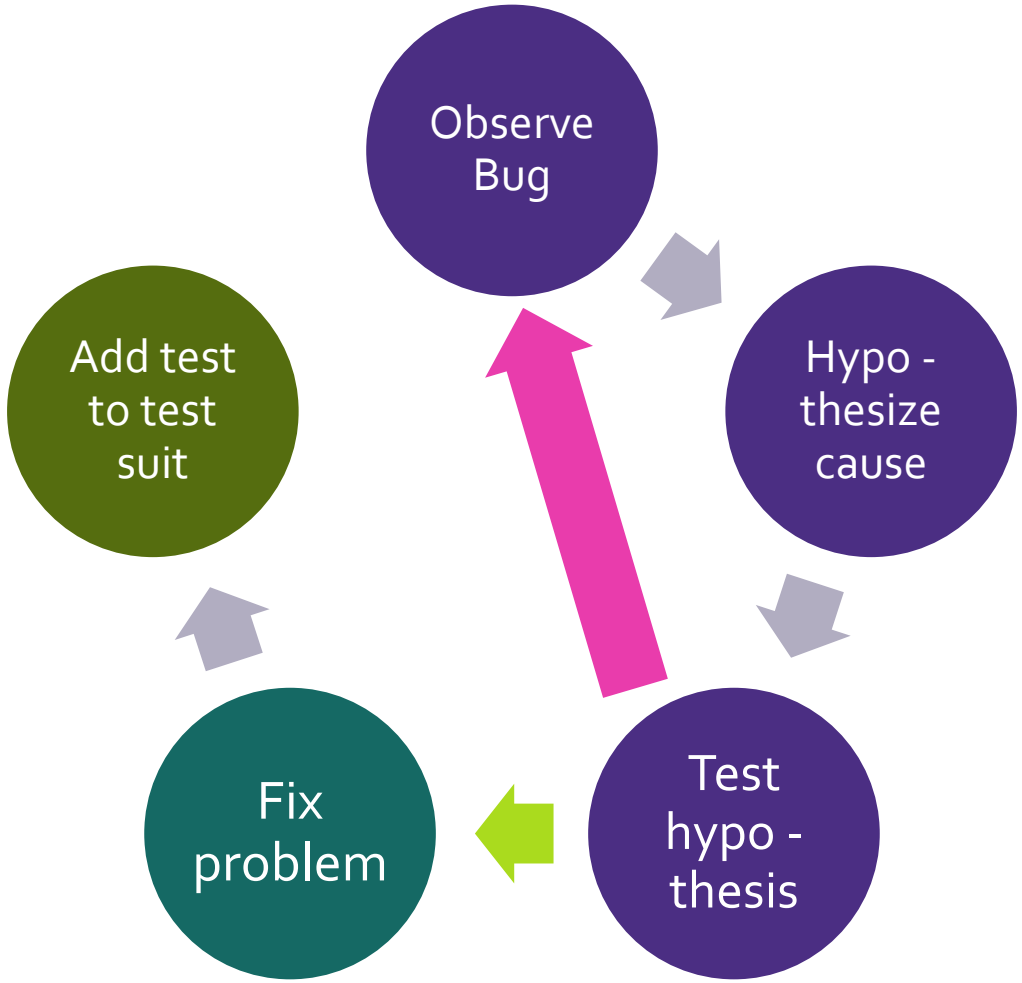
- Don't Panic
- Be systematic
- Create theories
  - Describe the problem
  - Hypothesize causes
- Test theories
  - Rule out hypotheses
  - Narrow area of bug
- Practice
- **Test early and often**



https://en.wikipedia.org/wiki/Rubber_duck_debugging

No more guess work

Use testable hypotheses to direct your fixes



Testable Hypotheses

Describe the problem with as much specificity as possible

- When does it happen?
- What actually happens?

Where could the bug be to cause that problem?

What might the bug look like?

Testable Hypothesis: Observe

Code 'factorial' does not produce the correct outputs

Which description is best?

A. `factorial()` does not return correct output
B. `factorial()` always returns `0`
C. `factorial(5)` does not return correct number
D. `factorial(5)` returns `0`

```
  printf("%d factorial is %d\n",
  input, output);
}
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
} }

[mh75@calgary cse374]$ ./factorial 0
0 factorial is 0
[mh75@calgary cse374]$ ./factorial 1
1 factorial is 0
[mh75@calgary cse374]$ ./factorial 5
5 factorial is 0
[mh75@calgary cse374]$
```

Testable Hypothesis: create theories

A. `factorial()` does not return correct output
B. `factorial()` always returns `0`
C. `factorial(5)` does not return correct number
D. `factorial(5)` returns `0`

Where may the problem be?

A. The `if` branch?
B. The `else` branch?
C. Somewhere else?

```
  printf("%d factorial is %d\n",
  input, output);
}
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
} }

[mh75@calgary cse374]$ ./factorial 0
0 factorial is 0
[mh75@calgary cse374]$ ./factorial 1
1 factorial is 0
[mh75@calgary cse374]$ ./factorial 5
5 factorial is 0
[mh75@calgary cse374]$
```

Testable Hypothesis: Tests

Create a series of tests that evaluates possible problem spots:

Case	Input	Math Equivalent	Expected	Actual
Base (`if` clause)	`factorial(0)`	`0! = 1`	1	???
Recursive (`else` clause)	`factorial(1)`	`1! = 1`	1	???
Recursive	`factorial(5)`	`5! = 1*2*3*4*5`	120	???

```
  printf("%d factorial is %d\n",
  input, output);
}
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
} }

[mh75@calgary cse374]$ ./factorial 0
0 factorial is 0
[mh75@calgary cse374]$ ./factorial 1
1 factorial is 0
[mh75@calgary cse374]$ ./factorial 5
5 factorial is 0
[mh75@calgary cse374]$
```

Testable Hypothesis: Execute Tests

Create a series of tests that evaluates possible problem spots:

Case	Input	Math Equivalent	Expected	Actual
Base (`if` clause)	`factorial(0)`	`0! = 1`	1	0
Recursive (`else` clause)	`factorial(1)`	`1! = 1`	1	0
Recursive	`factorial(5)`	`5! = 1*2*3*4*5`	120	0

```
  printf("%d factorial is %d\n",
  input, output);
}
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
} }
```

```
[mh75@calgary cse374]$ ./factorial 0
0 factorial is 0
[mh75@calgary cse374]$ ./factorial 1
1 factorial is 0
[mh75@calgary cse374]$ ./factorial 5
5 factorial is 0
[mh75@calgary cse374]$
```

Testable Hypothesis: Fix; re-test

Create a series of tests that evaluates possible problem spots:

Case	Input	Math Equivalent	Expected	Actual
Base (`if` clause)	`factorial(0)`	`0! = 1`	1	**1**
Recursive (`else` clause)	`factorial(1)`	`1! = 1`	1	**1**
Recursive	`factorial(5)`	`5! = 1*2*3*4*5`	120	**120**

```
  printf("%d factorial is %d\n",
  input, output);
}
int factorial(int x) {
  if (x == 0) {
    return 1;
  } else {
    return x * factorial(x-1);
} }

[mh75@calgary cse374]$ ./factorial 0
0 factorial is 1
[mh75@calgary cse374]$ ./factorial 1
1 factorial is 1
[mh75@calgary cse374]$ ./factorial 5
5 factorial is 120
[mh75@calgary cse374]$
```

Basic debugging techniques

- Observe the problem
  - Print statements
  - Display values with debugger
- Narrow down the possibilities
  - Comment out code
  - Replace some code with pre-computed values
  - Test one function at a time
    - Call the function with known values
    - Write test for just that function
- Test the edges
  - Code breaks at the beginning or end of a loop, the entry or exit, the base-case
  - Odd or exceptional cases need extra care

Debuggers can help

- Use the tool to *interactively* observe, change, and test your code
  - Breakpoints
  - Inspect values
  - Trace the stack
  - Identify behavior by line
- Most modern IDEs have some debugging capability
- *Debuggers are still just tools…. You need to do the work.*

GDB – gnu debugger

- Compile code with '-g' flag (saves human readable info)
- Open the program with: gdb
  - Start or restart the program: run
  - Quit the program: kill
  - Quit gdb: quit

- `bt` – stack backtrace
- up, down – change current stack frame
- `list` – display source code (list n, list)
- `print expression` – evaluate and print expression
- `display expression`
  - (re-)evaluate and print expression every time execution pauses.
  - undisplay – remove an expression from this recurring list.
- `info locals` – print all locals (but not parameters)
- `x` (examine) – look at blocks of memory in various formats

Valgrind – memory

- Compile code with '-g' flag (saves human readable info)
- Run **`valgrind`** and pass the program executable in as an argument.

```
valgrind –leak-check=full
./adwrong
```

- Analyze threading bugs
- Profile heap usage
- Generate call graphs
- Tune performance

Demos

- Seg-faults: arrdynamicwrong.c
  - gdb
  - Valgrind
- Inspect values: reverse.c

gdb: breakpoints

- Temporarily stop program running at given points
  - Look at values in variables
  - Test conditions
- break function (or line-number or …)
- conditional breakpoints (break XXX if expr)
  - to skip a bunch of iterations
  - to do assertion checking
- going forward: continue, next, step, finish
  - Some debuggers let you "go backwards" (typically an illusion)
- Also useful for learning program structure (e.g., when is a function called)

- break – set breakpoint.
  - break , break , break :
- info break – print table of current BPs
- clear – remove breakpoints
- disable/enable – temporarily off/on
- continue – resume execution to next BP
- step – execute next source line
- next – execute next source line
  - But treat function calls as a single statement and don't step into them
- finish – execute to the conclusion of the current function
  - How to recover if you meant "next" instead of "step"

gdb: commands

gdb ./myexecutable	Start GDB
run [args] …	Run the program with the given arguments
quit	Quit GDB
backtrace	Print the functions that were called to get here
tui enable/disable	See the code while debugging
break (line number/function name)	Set a breakpoint on a certain line or function
next	Move to the next line, skipping over function calls
step	Move to the next line, going into function calls
List	List the code
help	

AI in the loop