CSE 374 Lecture 11

Debugging and *GDB*

Guest Lecturer: Andrew Hu

Agenda

- What is a bug?
- How should we debug?
- How can we debug in C?
    - GDB demo

What is a Bug?

- A bug is a difference between the design of a program and its implementation
  - Definition based on Ko & Meyers (2004)
- Basically, we expected something different from what is happening
- 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

How Should We Debug?

- Debugging is the process of finding and fixing bugs
- Debugging strategies look like:
    1. Describe a difference between expected and actual behavior
    2. Hypothesize possible causes
    3. Investigate possible causes (return to step 2 if cause not found)
    4. Fix the code which was causing the bug
- Vast majority of the time spent in steps 2 & 3

Why Should You Believe Me?

- CS Education research
  - Conducted by academic researchers
- Literature review
  - Reviewed around 15 papers
  - Mostly from ACM sponsored publications
- Expert opinion
  - Amy Ko, world expert on debugging
  - Code & Cognition Lab

Describe

We can describe the problem without even looking at the code

Which of these bug descriptions do you think 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

Hypothesize

Now, let's look at the code for factorial()

Select all the places where the error *could* be coming from

- ❏ The if statement's "then" branch
- ❏ The if statement's "else" branch
- ❏ Somewhere else

```
int factorial(int x) {
  if (x == 0) {
    // ignore for now
  } else {
    // ignore for now
  }
}
```

Investigate

For now, let's just investigate the base case and recursive case

The base case is the "if then" branch

The recursive case is the "else" branch

```
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
  }
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	`factorial(0)`	`0! = 1`	`1`	`???`
Recursive	`factorial(1)`	`1! = 1`	`1`	`???`
Recursive	`factorial(2)`	`2! = 1 * 2`	`2`	`???`
Recursive	`factorial(3)`	`3! = 1 * 2 * 3`	`6`	`???`

Demo: Testing

Investigate - Testing

One way to investigate is to write code to test different inputs

If we do this, we find that the base case has a problem

```
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
  }
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	`factorial(0)`	`0! = 1`	`1`	`0`
Recursive	`factorial(1)`	`1! = 1`	`1`	`0`
Recursive	`factorial(2)`	`2! = 1 * 2`	`2`	`0`
Recursive	`factorial(3)`	`3! = 1 * 2 * 3`	`6`	`0`

Fix

```
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
  }
}
```

```
int factorial(int x) {
  if (x == 0) {
    return 1;
  } else {
    return x * factorial(x-1);
  }
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	`factorial(0)`	`0! = 1`	1	1
Recursive	`factorial(1)`	`1! = 1`	1	1
Recursive	`factorial(2)`	`2! = 1 * 2`	2	2
Recursive	`factorial(3)`	`3! = 1 * 2 * 3`	6	6

Testing

Pros

- Prevent bugs by testing early and often
- Forces you to think about edge cases (i.e. test driven development)

Cons

- Time consuming to write
- Only as good as the number of tests you write
- Doesn't give you a lot of detail

Demo: Segmentation fault

Review: Debugging Segmentation Fault

If we get a segmentation fault:

1. Compile with debugging symbols using `gcc -g -o myexecutable file.c`
2. `gdb ./myexecutable`
3. Type "`run`" into GDB
4. When you get a segmentation fault, type "`backtrace`"
5. Start from the top of the backtrace and investigate the line numbers

Demo: Inspect values at runtime

The Problem with `reverse.c`

Input

h	e	l	l	o	\n	\0

Output

h	e	l	l	o	\n	\0

The Problem with `reverse.c`

Input

h	e	l	l	o	\n	\0

Output

~~h~~\0	~~e~~\n	~~l~~ o	~~l~~ l	~~o~~ l	~~\n~~ e	~~\0~~h

The Problem with `reverse.c`

Input

h	e	l	l	o	\n	\0

Output

~~h~~\0	~~e~~\n	~~l~~ o	~~l~~ l	~~o~~ l	~~\n~~ e	~~\0~~h

Output is an empty C string. Zero characters followed by a null terminator

Why Should We Use a Debugger?

- Want to know which line we crashed at (backtrace)
- Inspect variables during run time
- Want to know which functions were called to get to this point (backtrace)

GDB Most Important 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

Breakouts: Using Breakpoints

Breakouts: Using Breakpoints

Do This

- Log onto klaatu or the VM
- `wget -O mysterynum.c tinyurl.com/374mystery`
- Have one person share their screen
- Try using GDB to find out the value that is computed