CSE 374 Lecture 12

Debugging and *GDB*

What is debugging?

debugging

/ˌdēˈbəgiNG/

noun

the process of identifying and removing errors from computer hardware or software.
"software debugging"

Definitions from Oxford Languages

What is a Bug?

- 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
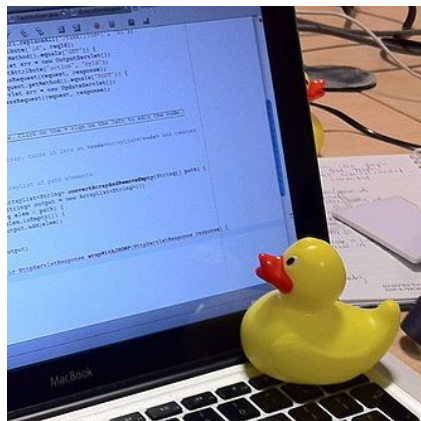
How do you avoid debugging?

Avoid writing code!

Ways to make debugging easier

1. Don't Panic
2. Be systematic
3. Create theories
4. Test theories
5. Practice
6. **Test early and often**

Describe the problem
Hypothesize causes

Rule out causes
Narrow area of the bug

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

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
  }
}
```

Demo: Testing

factorial.c

Aside

You should always do some systematic testing before submitting assignments - test each of the specs... Does it work? Start with the simplest, move to the complex

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	???

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

Basic debugging techniques

If you know something must be true - ex: a pointer can not be Null
Add a test, and print if there is an issue. Later - Asserts

- Add print statements
    - Says 'I got here', or 'my variable value is '2'
- Comment out (or delete) code
    - tests to determine whether removed code was the source of the problem
- Test one function at a time
    - Like you commented out most of the code….
- Test the edges
    - Code often breaks at the beginning or end of a loop, or at the entry or exit of a function; double check your logic in these places
    - Double check your logic in the odd / rare exceptional cases

Debuggers can help

A "debugger" is a tool that lets you stop running programs, inspect (sometimes set) values, and put in stops.

Instead of relying on changing code (commenting out, printf) interactively examine variable values, pause, and progress step-by-step. Eliminates the edit/recompile cycle.

Most modern IDEs have some built in debugging capacity.

Debuggers are just tools; they won't do the work.

GDB

Gdb => gnu debugger (standard part of linux development, supports many languages)
https://courses.cs.washington.edu/courses/cse374/21sp/refcard.pdf

Can examine a running file

Can also examine 'core' files of previous crashed programs…. Neat!

Run GDB

1. Compile code with '-g' flag (saves human readable info)
2. Open the program with: gdb
   a. Start or restart the program: run
   b. Quit the program: kill
   c. 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

Demo: Segmentation fault

arraydynamic.c

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

reverse.c

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

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 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