Lecture 15: Debugging in C

Lecture Participation Poll #15

Log onto pollev.com/cse374
Or
Text CSE374 to 22333
Reminder: Midpoint Deadline Friday November 6th at 9pm PST
- Will post grades to canvas sometime the week after
What is a Bug?

- A bug is a difference between the design of a program and its implementation
  - Definition based on Ko & Meyers (2004)

- 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
Debugging techniques

- Comment out (or delete) code
  - tests to determine whether removed code was source of problem

- Test one function at a time

- Add print statements
  - Check if certain code is reachable
  - check current state of variables

- test the edges
  - code often breaks at the beginning or end of the loop, the entry or exit of a function <- double check logic here
  - double check your logic in the odd/ rare exceptional case
Debugging Basics

Debugging strategies look like:

▪ Describe a difference between expected and actual behavior
▪ Hypothesize possible causes
▪ Investigate possible causes (if not found, go to step 2)
▪ Fix the code which was causing the bug
▪ Vast majority of the time spent in steps 2 & 3
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

```c
int factorial(int x) {
    if (x == 0) {
        return x;
    } else {
        return x * factorial(x-1);
    }
}
```
Let's investigate the base case and recursive case
- Base case is the "if then" branch
- Recursive case is the "else" branch

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

<table>
<thead>
<tr>
<th>Case</th>
<th>Input</th>
<th>Math Equivalent</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>factorial(0)</td>
<td>0! = 1</td>
<td>1</td>
<td>???</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(1)</td>
<td>1! = 1</td>
<td>1</td>
<td>???</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(2)</td>
<td>2! = 1 * 2</td>
<td>2</td>
<td>???</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(3)</td>
<td>3! = 1 * 2 * 3</td>
<td>6</td>
<td>???</td>
</tr>
</tbody>
</table>
Investigate

- 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.

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

<table>
<thead>
<tr>
<th>Case</th>
<th>Input</th>
<th>Math Equivalent</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>factorial(0)</td>
<td>0! = 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(1)</td>
<td>1! = 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(2)</td>
<td>2! = 1 * 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(3)</td>
<td>3! = 1 * 2 * 3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
Fix

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

<table>
<thead>
<tr>
<th>Case</th>
<th>Input</th>
<th>Math Equivalent</th>
<th>Expected</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>factorial(0)</td>
<td>0! = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(1)</td>
<td>1! = 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(2)</td>
<td>2! = 1 * 2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Recursive</td>
<td>factorial(3)</td>
<td>3! = 1 * 2 * 3</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
C Debugger

▪ A debugger is a tool that lets you stop running programs, inspect values etc…
  - instead of relying on changing code (commenting out, printf) interactively examine variable values, pause and progress set-by-step
  - don’t expect the debugger to do the work, use it as a tool to test theories
  - Most modern IDEs have built in debugging functionality

▪ ‘gdb’ -> gnu debugger, standard part of linux development, supports many languages
  - techniques are the same as in most debugging tools
  - can examine a running file
  - can also examine core files of previous crashed programs

▪ 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)
Meet gdb

- Compile code with ‘-g’ flag (saves human readable info)
- Open program with gdb <executable file>
- start or restart the program: run <program args>
  - quit the program: kill
  - quit gdb: quit
- Reference information: help
  - Most commands have short abbreviations
    - bt = backtrace
    - n = next
    - s = step
    - q = quit
  - <return> often repeats the last command
**GDB QUICK REFERENCE**

**Essential Commands**
- `gdb [core]`  |  debug program [using coredump core]
- `b [file:]func`  |  set breakpoint at function [in file]
- `run [arglist]`  |  start your program [with arglist]
- `bt`  |  backtrace: display program stack
- `p expr`  |  display the value of an expression
- `c`  |  continue running your program
- `n`  |  next line, stepping over function calls
- `s`  |  next line, stepping into function calls

**Starting GDB**
- `gdb`  |  start GDB, with no debugging files
- `gdb program`  |  begin debugging program
- `gdb program core`  |  debug core dump produced by program
- `gdb --help`  |  describe command line options

**Stopping GDB**
- `quit`  |  exit GDB; also q or EOF (eg C-c) terminate current command, or send to running process
- `INTERRUPT`  |  (eg C-c) terminate current command, or send to running process

**Getting Help**
- `help`  |  list classes of commands
- `help class`  |  one-line descriptions for commands in class
- `help command`  |  describe command

**Executing your Program**
- `run arglist`  |  start your program with arglist
- `run`  |  start your program with current argument list
- `run ... <inf> outf`  |  start your program with input, output redirected
- `kill`  |  kill running program

**Breakpoints and Watchpoints**
- `break [file:]line`  |  set breakpoint at line number [in file]
- `b [file:]line`  |  eg: `break main.c:37`
- `break [file:]func`  |  set breakpoint at func [in file]
- `break *offset`  |  set break at offset lines from current stop
- `break -offset`  |  set break at current offset
- `break *addr`  |  set break at address addr
- `break ... if expr`  |  break conditionally on nonzero expr
- `cond n [expr]`  |  new conditional expression on breakpoint n; make unconditional if no expr
- `tbreak ...`  |  temporary break; disable when reached
- `rbreak [file:]regex`  |  break on all functions matching regex [in file]
- `watch expr`  |  set a watchpoint for expression expr
- `catch event`  |  break at event, which may be catch, throw, exec, fork, vfork, load, or unload
- `info break`  |  show defined breakpoints
- `info watch`  |  show defined watchpoints
- `clear`  |  delete breakpoints at next instruction
- `clear [file:]func`  |  delete breakpoints at entry to func()
- `clear [file:]line`  |  delete breakpoints on source line
- `delete [n]`  |  delete breakpoints [or breakpoint n]
- `disable [n]`  |  disable breakpoints [or breakpoint n]
- `enable [n]`  |  enable breakpoints [or breakpoint n]
- `enable once [n]`  |  enable breakpoints [or breakpoint n]; disable again when reached
- `enable del [n]`  |  enable breakpoints [or breakpoint n]; delete when reached
- `ignore n count`  |  ignore breakpoint n, count times
- `commands n`  |  execute GDB command-list every time breakpoint n is reached.
- `command-list [silent] [silent]`  |  suppresses default display

**Execution Control**
- `continue [count]`  |  continue running; if count specified, ignore this breakpoint next count times
- `c [count]`  |  continue running; if count specified, ignore this breakpoint next count times
- `step [count]`  |  execute until another line reached; repeat times if specified
- `s [count]`  |  step by machine instructions rather than source lines
- `stepi [count]`  |  execute next line, including any function calls
- `n [count]`  |  next line machine instruction rather than source line
- `next [count]`  |  next line machine instruction rather than source line
- `ni [count]`  |  next line machine instruction rather than source line
- `until [location]`  |  run until next instruction (or location)
- `finish`  |  run until selected stack frame returns
- `return [expr]`  |  pop selected stack frame without executing [setting return value]
- `signal num`  |  resume execution with signal s (none if 0)
- `jump line`  |  resume execution at specified line number
- `jump *address`  |  or address
- `set var=expr`  |  evaluate expr without displaying it; use for altering program variables

**Display**
- `print [f] [expr]`  |  show value of expr [or last value $] according to format f:
- `p [f] [expr]`  |  show value of expr [or last value $] according to format f:
- `x`  |  hexdecimal
- `d`  |  signed decimal
- `u`  |  unsigned decimal
- `o`  |  octal
- `t`  |  binary
- `a`  |  address, absolute and relative
- `c`  |  character
- `f`  |  floating point
- `call [f] [expr]`  |  like print but does not display void
- `x [N &] [expr]`  |  examine memory at address expr; optional format spec follows slash
- `N`  |  count of how many units to display

https://courses.cs.washington.edu/courses/cse374/19sp/refcard.pdf
Useful GDB Commands

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

If we get a segmentation fault:
1. `gdb ./myprogram`
2. Type "run" into GDB
3. When you get a segfault, type "backtrace" or "bt"
4. Look at the line numbers from the backtrace, starting from the top
Breakpoints

temporarily stop program running at given points
- look at values in variables
- test conditions
- break function (or line-number)
- conditional breakpoints
  - to skip a bunch of iterations
  - to do assertion checking

- break – sets breakpoint
  - break <function name> | <line number> | <file>:<line number>

- info break – print table of currently set breakpoints

- clear – remove breakpoints

- disable/enable temporarily turn breakpoints off/on

- continue – resume execution to next breakpoint or end of program

- step – execute next source line

- next – execute next source line, but treat function calls as a single statement and don’t “step in”

- finish – execute to the conclusion of the current function
  - how to recover if you meant “next” instead of “step”
gdb demo
Output is an empty C string, zero characters followed by a null terminator.
Testing

Computers don’t make mistakes- people do!

“I’m almost done, I just need to make sure it works”
– Naive 14Xers

**Software Test:** a separate piece of code that exercises the code you are assessing by providing input to your code and finishes with an assertion of what the result should be.

1. Isolate
2. Break your code into small modules
3. Build in increments
4. Make a plan from simplest to most complex cases
5. Test as you go
6. As your code grows, so should your tests
Types of Tests

▪ Black Box
  - Behavior only – ADT requirements
  - From an outside point of view
  - Does your code uphold its contracts with its users?
  - Performance/efficiency

▪ White Box
  - Includes an understanding of the implementation
  - Written by the author as they develop their code
  - Break apart requirements into smaller steps
  - “unit tests” break implementation into single assertions
What to test?

Expected behavior
- The main use case scenario
- Does your code do what it should given friendly conditions?

Forbidden Input
- What are all the ways the user can mess up?

Empty/Null
- Protect yourself!
- How do things get started?

Boundary/Edge Cases
- First
- last

Scale
- Is there a difference between 10, 100, 1000, 10000 items?
Tips for testing

▪ You cannot test every possible input, parameter value, etc.
  - Think of a limited set of tests likely to expose bugs.

▪ Think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size

▪ Think about empty cases and error cases
  - 0, -1, null; an empty list or array

▪ Test behavior in combination
  - Maybe add usually works, but fails after you call remove
  - Make multiple calls; maybe size fails the second time only
Appendix