CMPSC 447
C Debugging Review

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

C Program Flaws

PENNSTATE 1855

- A lot of unintended behaviors in C programs cause bugs or flaws
  - These may crash the program (seg fault)
  - Or cause the program to behave incorrectly
- How do you find and repair flaws in C programs quickly?
  - Not an easy task given the complex semantics of C concepts, especially without type/memory safety

Printf Debugging

PENNSTATE 1855

- Find where you think there is a problem and print the relevant variable values using printf
  - If you have a segmentation fault, which values do you print?
    - Segmentation fault refers to a pointer referencing an illegal memory location
      - How do you print pointer values?
  - There may be several causes
    - Initialization (null pointer)
    - Error from another pointer access that modified this pointer
      - Any other one?

Printf Debugging

PENNSTATE 1855

- Find where you think there is a problem and print the relevant variable values
  - If you have an erroneous data value, which values do you print?
    - You could print that variable, but it may have been modified at any time by a stray pointer
  - That could be a lot of printf statements
    - All statements that impact the value normally
    - And any other statement where a pointer operation may have modified the variable value or some input to the variable

Debuggers

PENNSTATE 1855

- Programs that track the execution of another target program
- You run the program "in" the debugger
  - The debugger can then read the memory of the target
  - If you compiled the target with "debugger symbols" these are used as a guide to help display the state of the target
  - You can use the debugger to run the target incrementally to "breakpoints" where you can inspect the state
- Debuggers are super useful

Debugger Options

PENNSTATE 1855

- Debuggers are tied to compilers
  - gcc compiler: gdb
  - clang compiler: lldb
  - Pretty similar
- Code compiled in clang can be debugged using either debugger
  - Command map: https://lldb.llvm.org/use/map.html
- We will take a walk through lldb today

lldb Debugging

PENNSTATE 1855

- Program code: gdb_demo.c
- Compile for lldb with "-g"
  - clang –g gdb_demo.c –O0 –o gdb_demo
- Run in the debugger: lldb <executable>
  - lldb gdb_demo
- Run the program in the debugger
  - r (for "run")
    - EXC_BAD_ACCESS (type of segmentation fault)
    - Stopped at line 63, column 26

lldb Debugging

- Program code: gdb_demo.c
- Seg fault debugging
  - What happened?
- Let's find out where we are in the target's execution
  - bt (for "backtrace")
    - Displays a sequence of functions from crash (#0) back up the call stack – usually to main
    - Recursive calls to tree_size

lldb Debugging

PENNSTATE 1855

- Program code: gdb_demo.c
- Seg fault debugging
  - What are the variable values?
- Print the variable/expression value "p"
  - Super useful!
  - p (for "print") size (name of variable/expression)
    - Response: (int) $0 = 1
    - "(int)" is the type, "$0" is an identifier to reuse value, "1" is the value
    - Note: p $0+$0 = 2

lldb Debugging

- Program code: gdb_demo.c
- Seg fault debugging (more)
  - What are the variable values?
- Print the variable/expression value "p" for the tree
  - p t
    - Response: (tree_t *) $1 = 0x0
    - "(tree_t *)" is the type, "$1" is an identifier to reuse value, "0x0" is the value

lldb Debugging

- Program code: gdb_demo.c
- Seg fault debugging
  - What happened in the tree_size function?
- Let's find out what the variable values are problematic
  - See "t=0x0" in frame #0
  - l (for "list") tree_size (function to list)
    - See line 63
    - Do "list" repeatedly to see the next part of the code

lldb Debugging

- Program code: gdb_demo.c
- Look at other functions
  - Switch to calling function
    - f (for "frame") 1 (index in backtrace)
      - Note the movement of the asterisk in the backtrace to frame 1
  - Can print variable/expression values in each frame
    - "p t" – not null and size is still "1" in frame 1
    - Or can use "up" or "down" to traverse frames
  - Print "t→left" and "t→right" in frame 1

lldb GUI

- Program code: gdb_demo.c
- Graphical debugger
  - "gui" starts it
- Shows the code and variables with values
  - Right to expand and left to retract
  - Up and down keys to scan the code
  - NOTE: Need to run the program before activating
- However, to run commands need to exit (escape)
  - May like "gdb –tui instead"

lldb Debugging – Part 2

- Program code: gdb_demo.c
- Let's look at another example
  - Uncomment lines in main
- Recompile and run in lldb again
  - How?
- What happens?

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Nothing much is happening – let's see in debugger
  - Ctrl-C to stop the execution
  - Then what?
- Next – rerun last command and up for history
  - Print variable values as before
- Print what you need from the debugger– no need to recompile
  - Print *new - get value at memory location of "new"

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Now that we have narrowed down the problem area, want to run the program directly to there
- Set a breakpoint
  - b (break) tree_remove_root – at a function
  - b (break) gdb_demo.c:84 or b 84 – at a line in a file
  - "b" lists all breakpoints
- How to run inside that function?
  - "Next" runs the next instruction in the same function

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Answer: Use "step" to follow a call into the callee
- Follow control flow in the debugger
  - Next (n): run the next line in the same function
  - Step (s): run the next line in the same function unless a function call
    - Run into the callee
  - Continue (c): run to the next breakpoint
- Don't forget to use list to see rest of the code

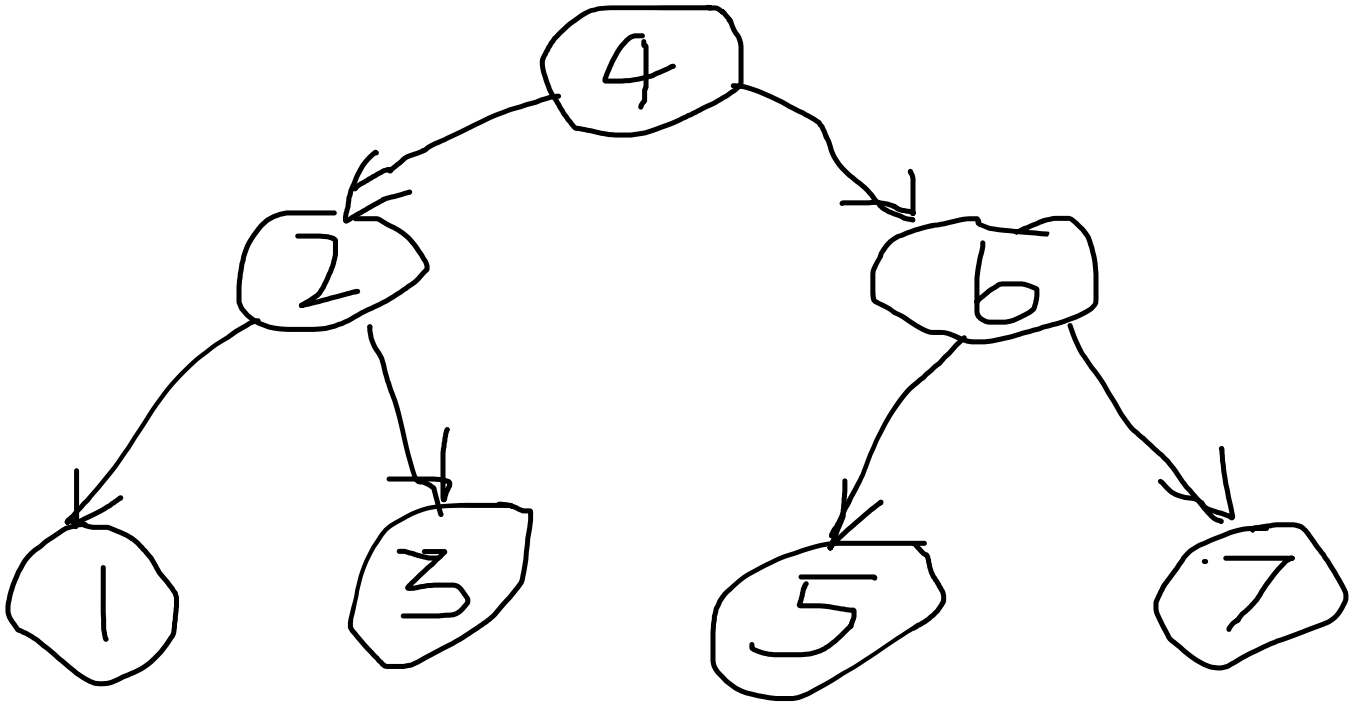
lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Find the cause
  - Divide and conquer
    - Debug from beginning to *tree_size* at function level
    - Then drill down
  - Program stops running in second *tree_remove_root*
- What does the tree data structure look like
  - Print root, *root, root->left, root->right
    - Draw the tree to see it

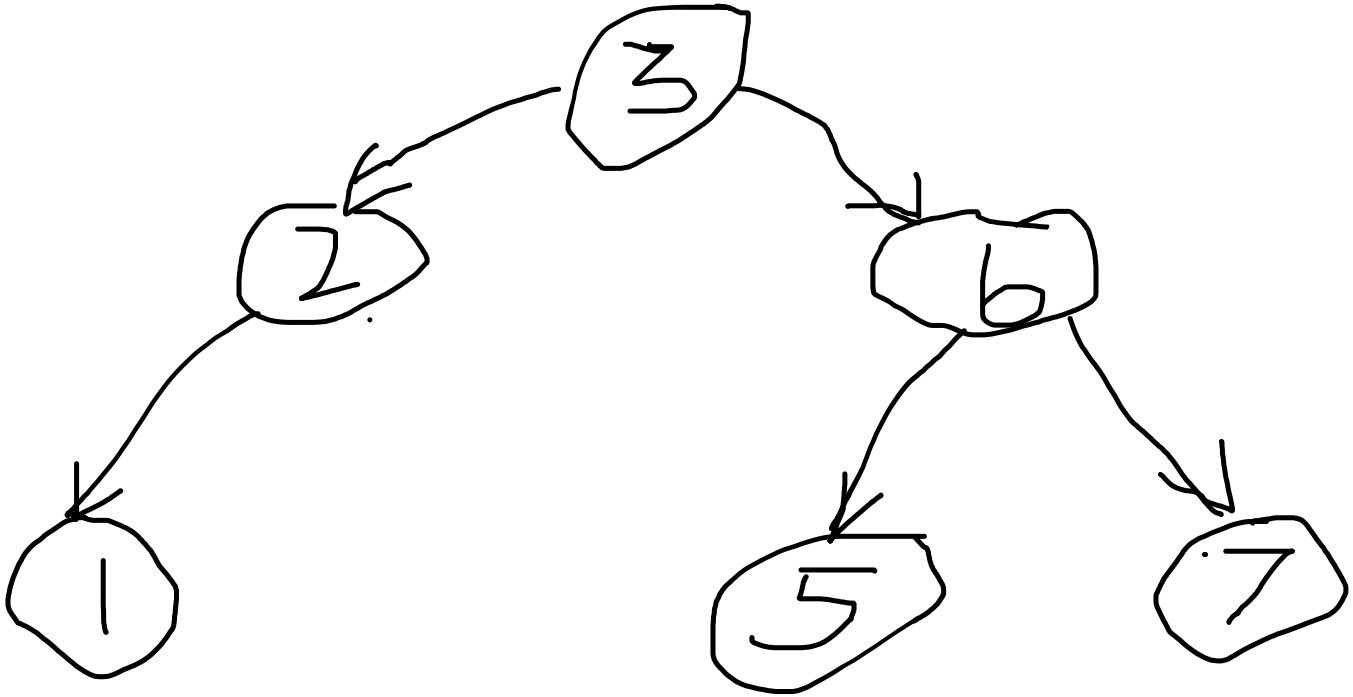
lldb Debugging – Part 2

- Draw the tree



lldb Debugging – Part 2

- Draw the tree



lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- More on breakpoints
  - Disable: br dis #
  - Enable: br en #
- Temporary breakpoints
  - Break at this location once: tb
- Conditional breakpoints – only break when true
  - b 45 if (t->left->right == 0)

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Let's see how tree_remove_root operates
- Step through program states with the debugger
  - See the state of the tree
  - See the relationships among nodes (left and right)
  - See how the code modifies these relationships
- Tree is modified such that
  - Node 3 becomes new root
  - What problem in the code causes the flaw?

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- What problem in the code causes the flaw?
  - Issue - modify node 3's fields (as *new*) before its child (*prev*, as node 2)
- What can you do to assess the impact
  - Can assign variables to new values in the debugger too
  - Using print (p) as a result of an expression
    - p prev->right = 0x0
- Then, can continue the execution - next (n), step (s), or continue (c)

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Continue running after changing assignment
  - Looks good at return of function (via next)
  - Let's try to run to the end
    - Which command to do that?
- Oh, no – another segmentation fault
  - Stops in *tree_remove_root*
    - Remember we are in the second invocation of *tree_remove_root*
      - after removing the first root

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Find the cause of the segmentation fault
  - The variable new is null
    - Why does this create a segmentation fault?
    - What should you do to fix that?

lldb Debugging – Part 2

PENNSTATE 1855

- Program code: gdb_demo.c
- Note that we found the causes, and the basic idea for the fixes of two flaws in one run of the program
  - With the debugger (thanks, debugger)
- With no code modifications (e.g., add printfs) and no recompilation required to find the second flaw
  - Didn't even have to restart the program
  - Only needed to undo the impact of the first flaw and check the state at the fault of the second flaw

Take Away

- Your C programs may contain flaws after they compile successfully
  - Cause the program to crash or give erroneous results
- Flaws may be due to either
  - Erroneous variable values or pointer values
- Debuggers for C are powerful and feature-rich
  - We have just scratched the surface
  - Learn a command a day
- And we will need them later to understand exploits