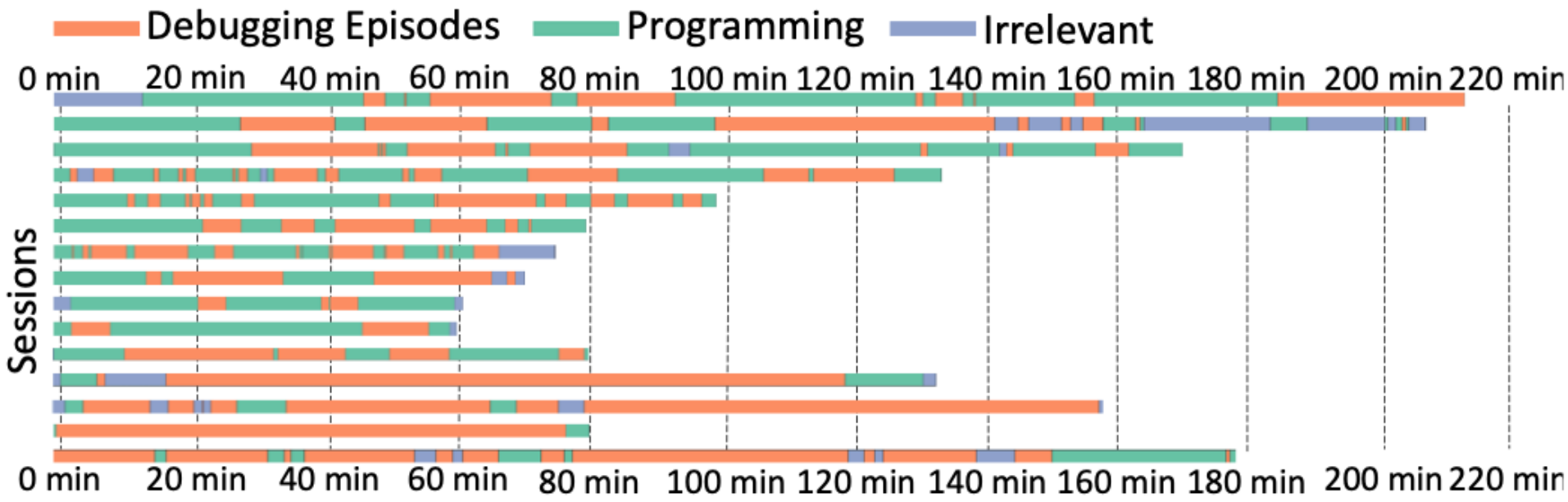
Lab3 GDB Basic

Jason Zixu Zhou

No one writes good code the first time



Aladoubi, et al. “An Exploratory Study of Debugging Episodes”. arXiv 2021.

You will spend a lot of your time debugging!

Debugging Process

1. Reproduce the error
2. Identify Problematic Region (coarse or fine)
    1. Incremental testing
    2. Run with valgrind
3. Use print statements
4. Use GDB

What is GDB

- **Definition**: GDB (GNU Debugger) is a powerful debugging tool used for C/C++ programs.
- **Key Features**:
  - Set breakpoints
  - Step through code
  - Inspect variables and memory
  - Analyze core dumps
- **Supported Languages**: C, C++, Fortran, Go, etc.

Why use GDB

- Helps locate and fix bugs by:
  - Stopping the program at breakpoints
  - Inspecting the state of variables
  - Following the flow of execution
- Debugging improves code reliability and performance.

Compiling with Debugging Information

- Use -g flag when compiling to include debugging information in the binary

```
gcc -g your_program.c -o your_program
```

- This enables GDB to map the binary back to source code.

Basic GDB Commands

1. **Starting GDB**

   Start the GDB debugger and load the executable you want to debug: gdb ./your_program

2. **Running the Program**

   Run the program until it hits a breakpoint or finishes: run

3. **Setting Breakpoints**

   Set a breakpoint at a specific line number: break <line_number>

4. **Continuing Execution**

   Continue running the program until the next breakpoint or the program ends: continue

Inspecting Variables

1. **Printing Variables**
   Print the current value of a variable: print variable_name
2. **Viewing Local Variables**
   Display all local variables in the current function: info locals
3. **Changing Variable Values**
   Change the value of a variable while debugging: set variable variable_name = value

Stepping Through Code

1. **Next Line (Without Entering Functions)**

   Execute the next line of code without stepping into functions: next

2. **Step Into Functions**

   Step into the function calls in the current line: step

3. **Continue Until a Specific Line**

   Continue execution until a specified line number is reached: until <line_number>

Backtrace and Call Stack

1. **View the Call Stack**

   Display the current call stack to see the chain of function calls: backtrace

2. **Switching Frames**

   Switch to a specific frame in the call stack to inspect it: frame <number>

Managing Breakpoints

1. **Conditional Breakpoints**

   Set a breakpoint that triggers only when a specific condition is met: break <line_number> if <condition>

2. **Listing Breakpoints**

   View all currently set breakpoints: info breakpoints

3. **Removing Breakpoints**

   Delete a specific breakpoint by its number: delete <breakpoint_number>

How to use GDB?

- Compile for gdb with `gcc -g`
- Run file with gdb `gdb ./a.out`
- Set break points
  - At break points – analyse
- Run `(gdb) r`
- Quit `(gdb) q`

GDB Cheat Sheet

Basics	
`$ gcc -g ...`	create an executable that can be debugged using GDB
`$ gdb progName`	start debugging `progName`
`$ gdb --args progName args`	start debugging `progName`, using command-line arguments `args`
`(gdb) q`	quit GDB
`(gdb) help command`	display information about `command`, incl. its syntax
`(gbd) run`	start running program
`(gbd) kill`	terminate currently running program

Examining Data

`print expr`	show current value of expression `expr`
`print var->attr`	show current value of attribute `attr` of struct `var`
`print *arr@len`	show current value of first `len` elements of array `arr`
`print/format expr`	show current value of expression `expr` in format `format`
`print/x expr`	show current value of `expr` in hexadecimal
`print/t expr`	show current value of `expr` in binary
`print/c expr`	show current value of `expr` as an integer and its character representation
`print/f expr`	show current value of `expr` in floating point syntax
`print/s expr`	show current value of `expr` as a string, if possible
`display expr`	automatically print value of expression `expr` at each halt in execution
`undisplay disp#`	stop displaying expression with display number `disp#`
`watch expr`	set a watchpoint on expression `expr` (break whenever value of `expr` changes)
`info args`	show value of all arguments to current function
`info locals`	show current value of all local variables
`x addr`	show current word in memory at address `addr`, in hexademical
`x/units|format|size addr`	show current value of memory of size `units` x `size` at address `addr`, in format `format`
`x/3tb addr`	show current value of 3 bytes of memory at address `addr`, in binary

Examining the Stack

`backtrace`	display the current call stack (can be used after a runtime error, eg. segfault)

Breakpoints	
`break point`	create a breakpoint at `point`

`break 5`
`break func`
`break foo.c:5`
	create a breakpoint at line 5 of current source file
	create a breakpoint at body of function `func`
	create a breakpoint at line 5 of source file `foo.c`

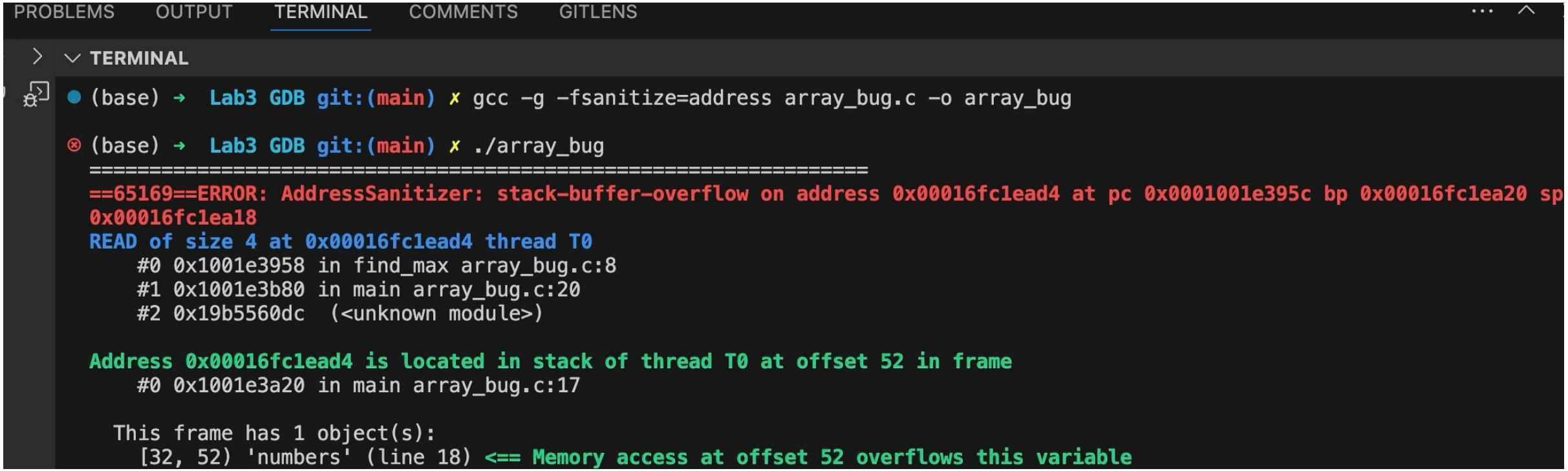
`break point if cond`	create a breakpoint at `point` which triggers if Boolean expression `cond` evaluates to *true*
`info breakpoints`	display information about all current breakpoints
`delete`	remove all breakpoints
`delete breakpoint#`	remove breakpoint with number `breakpoint#`

Continuing and Stepping	
`continue`	continue executing normally
`finish`	continue executing until current function returns
`step`	execute next line of source code
`next`	execute next line of source code, without descending into functions

Altering Execution	
`return expr`	return from current function at this point, with return value `expr`
`set var var=expr`	store value of expression `expr` into program variable `var`
`set var g=4`	store 4 into program variable `g`
`set {type}addr = expr`	store value of expression `expr` (represented as type `type`) into memory at address `addr`
`set {int}0x83040 = 4`	store 4 as an int at address 0x83040
`signal signal`	continue executing and immediately send signal `signal` to the program
`signal SIGINT`	continue executing and immediately send an interrupt signal to the program

Credit: Gabrielle Singh Cadieux
https://gabriellesc.github.io/teaching/resources/GDB-cheat-sheet.pdf

Array_bug



gcc -g -fsanitize=address array_bug.c -o array_bug

Double_free Example

```
(base) ➜  Lab3 GDB git:(main) ✗ lldb ./double_free_bug
(lldb) target create "./double_free_bug"
Current executable set to '/Users/zixuzhou/McGill/TA/ECSE 427/ECSE427-COMP310Lab/Lab3 GDB/double_free_bug' (arm64
(lldb) (lldb) break set -n process_people
error: '(lldb)' is not a valid command.
(lldb)
Current executable set to '/Users/zixuzhou/McGill/TA/ECSE 427/ECSE427-COMP310Lab/Lab3 GDB/double_free_bug' (arm64
(lldb) break set -n process_people
Breakpoint 1: where = double_free_bug`process_people + 24 at double_free_bug.c:39:18, address = 0x0000000100003e5
(lldb) r
Process 67311 launched: '/Users/zixuzhou/McGill/TA/ECSE 427/ECSE427-COMP310Lab/Lab3 GDB/double_free_bug' (arm64)
Process 67311 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100003e54 double_free_bug`process_people at double_free_bug.c:39:18
   36
   37   void process_people()
   38   {
-> 39       Person *p1 = create_person("Alice", 30);
   40       Person *p2 = create_person("Bob", 25);
   41
   42       printf("Processing: %s, %d\n", p1->name, p1->age);
Target 1: (double_free_bug) stopped.
(lldb) next
```