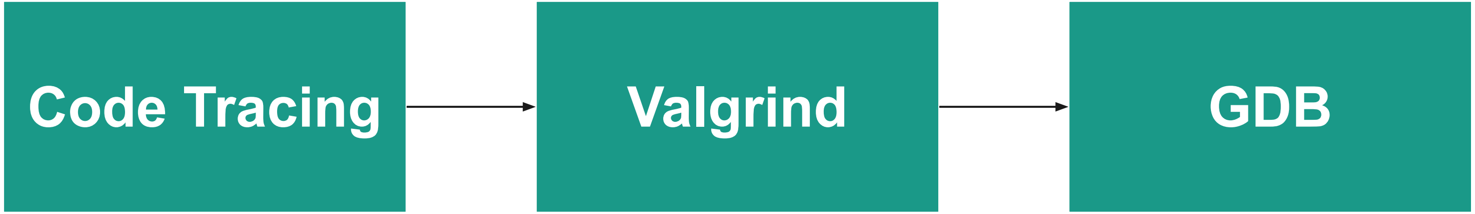
Introduction to GDB and Debugging

15-213/18-213/15-513/14-513/18-613: Introduction to Computer Systems

Big Questions

- How can code be debugged?
  - What is code tracing?
  - What is valgrind?
  - What is GDB?
- How do you use GDB?

Tools for Debugging



Code Tracing → Valgrind → GDB

printf()

malloc/free bug

Everything else

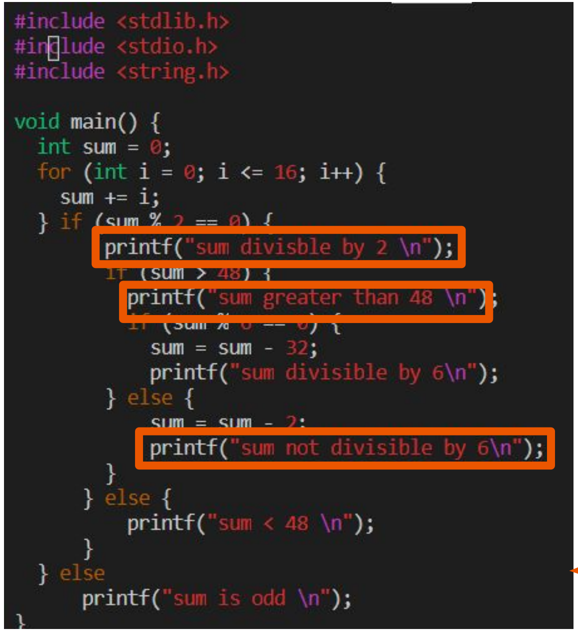
Debugging Basics: Code Tracing

Code Tracing

- Use print statements to determine variable values at different points in code
  - Insert print statements after sections of code
    - Keep track of values
  - Can also print out several values at a time to see how values change
  - Think through the actual vs expected outputs

Why use code tracing?	
When to Use	When Not to Use
● Easy and relatively simple code ● Tracing conditional paths in an if statement	● Messy and complicated programs ● Typically prints out variable values regardless of if the value has changed ○ “Tidal wave of output”

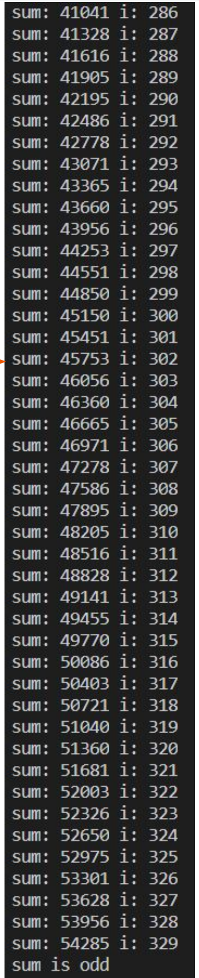
Code Tracing Example



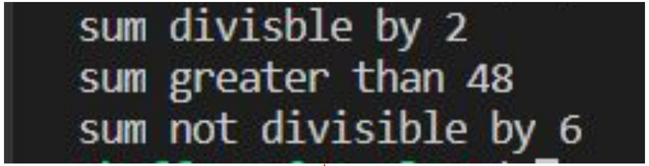
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main() {
  int sum = 0;
  for (int i = 0; i <= 16; i++) {
    sum += i;
  } if (sum % 2 == 0) {
      printf("sum divisble by 2 \n");
      if (sum > 48) {
        printf("sum greater than 48 \n");
        if (sum % 6 == 0) {
          sum = sum - 32;
          printf("sum divisible by 6\n");
        } else {
          sum = sum - 2;
          printf("sum not divisible by 6\n");
        }
      } else {
        printf("sum < 48 \n");
      }
  } else
     printf("sum is odd \n");
}
```

BAD

- Prints out series of unhelpful information



```
sum: 41041 i: 286
sum: 41328 i: 287
sum: 41616 i: 288
sum: 41905 i: 289
sum: 42195 i: 290
sum: 42486 i: 291
sum: 42778 i: 292
sum: 43071 i: 293
sum: 43365 i: 294
sum: 43660 i: 295
sum: 43956 i: 296
sum: 44253 i: 297
sum: 44551 i: 298
sum: 44850 i: 299
sum: 45150 i: 300
sum: 45451 i: 301
sum: 45753 i: 302
sum: 46056 i: 303
sum: 46360 i: 304
sum: 46665 i: 305
sum: 46971 i: 306
sum: 47278 i: 307
sum: 47586 i: 308
sum: 47895 i: 309
sum: 48205 i: 310
sum: 48516 i: 311
sum: 48828 i: 312
sum: 49141 i: 313
sum: 49455 i: 314
sum: 49770 i: 315
sum: 50086 i: 316
sum: 50403 i: 317
sum: 50721 i: 318
sum: 51040 i: 319
sum: 51360 i: 320
sum: 51681 i: 321
sum: 52003 i: 322
sum: 52326 i: 323
sum: 52650 i: 324
sum: 52975 i: 325
sum: 53301 i: 326
sum: 53628 i: 327
sum: 53956 i: 328
sum: 54285 i: 329
sum is odd
```



```
sum divisble by 2
sum greater than 48
sum not divisible by 6
```

GOOD

- Not super complicated code
- Trace through if/else chain
- RISK: bug in trace code!

Debugging Memory: Valgrind

Valgrind

- Tool for debugging, memory leak detection, and profiling
- Valgrind flags errors that don't appear without valgrind

Using valgrind (Make sure Valgrind is installed):

```
$ valgrind ./a.out
...
HEAP SUMMARY:
==41495== in use at exit: 0 bytes in 0 blocks
==41495== total heap usage: 1 allocs, 1 frees, 8 bytes allocated
==41495==
==41495== All heap blocks were freed --- no leaks are possible
...
```

Why use Valgrind?	
When to Use	When Not to Use
● Dealing with memory (especially dynamic memory allocation) ● Whenever bugs occur. Get instant feedback about what the bug is, where it occurred, and why.	● Program contains no invalid reads and writes and no leaked memory ● If the test case is inherently slow, then this is not a good choice

Running Valgrind

Recommended Valgrind Options:

```
$ valgrind --leak-resolution=high --leak-check=full
--show-reachable=yes --track-fds=yes ./myProgram arg1 arg2
```

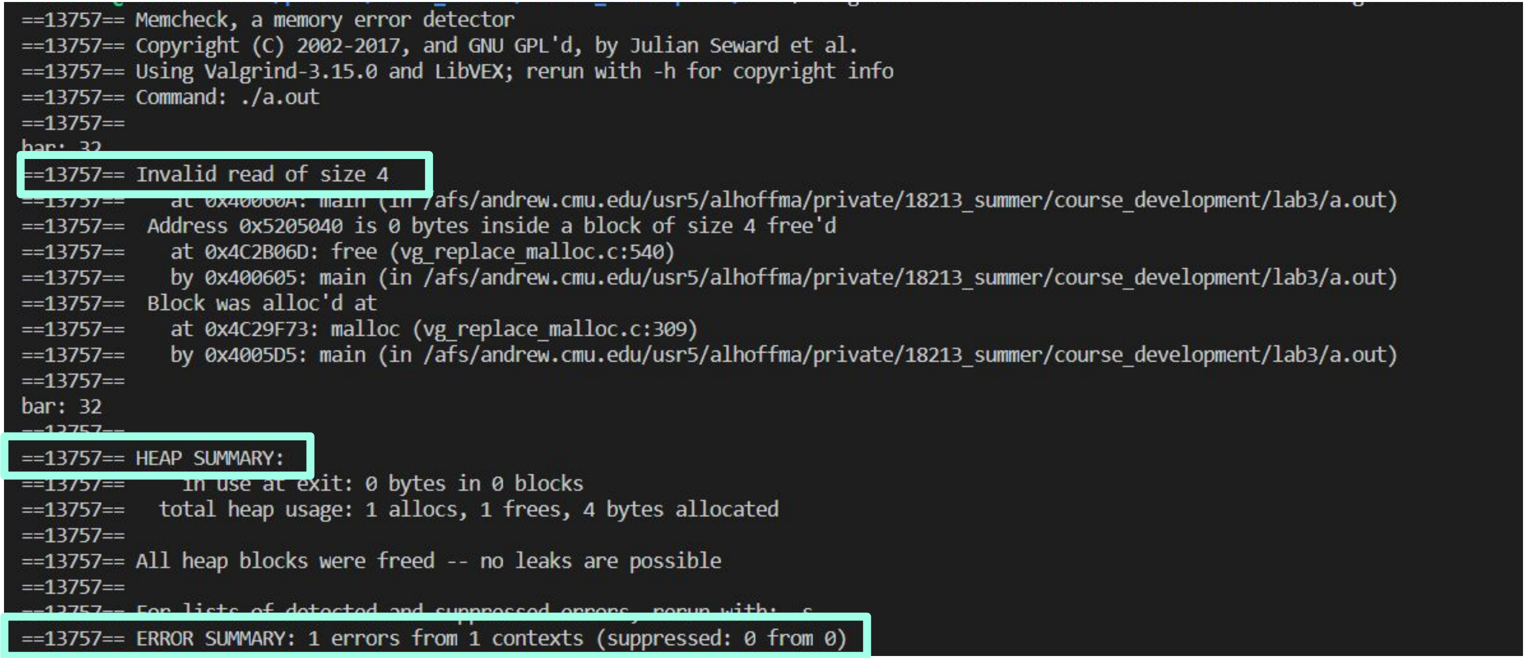
Feel free to look through **`$ man valgrind`** and play around with options

Invalid Reads and Writes

- Reading freed variables
- Reading uninitialized variables
- Writing to uninitialized memory
  - Caused by writing too much data to allocated memory

```
int foo( int y) {
    int *bar = malloc(sizeof(int));
    *bar = y;
    free(bar);
    printf(“bar: %d \n”, *bar);
    return y;
}
```

Invalid Reads and Writes Sample Output



```
==13757== Memcheck, a memory error detector
==13757== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13757== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==13757== Command: ./a.out
==13757==
bar: 32
==13757== Invalid read of size 4
==13757==    at 0x40060A: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==13757==  Address 0x5205040 is 0 bytes inside a block of size 4 free'd
==13757==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==13757==    by 0x400605: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==13757==  Block was alloc'd at
==13757==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==13757==    by 0x4005D5: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==13757==
bar: 32
==13757==
==13757== HEAP SUMMARY:
==13757==     in use at exit: 0 bytes in 0 blocks
==13757==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==13757==
==13757== All heap blocks were freed -- no leaks are possible
==13757==
==13757== For lists of detected and suppressed errors, rerun with: -s
==13757== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Memory Leaks

- Forgetting to free memory after using it
  - Sometimes, there is overhead memory that is never freed
    - Memory that is allocated by a programmer should **always** be freed

```
int foo( int y) {
     int *bar = malloc(sizeof(int));
     *bar = y;
     printf(“bar: %d \n”, *bar);
     return y;
}
```

Types of Memory Leaks

Still Reachable

- Block is still pointed at, programmer could go back and free it before exiting

Definitely Lost

- No pointer to the block can be found

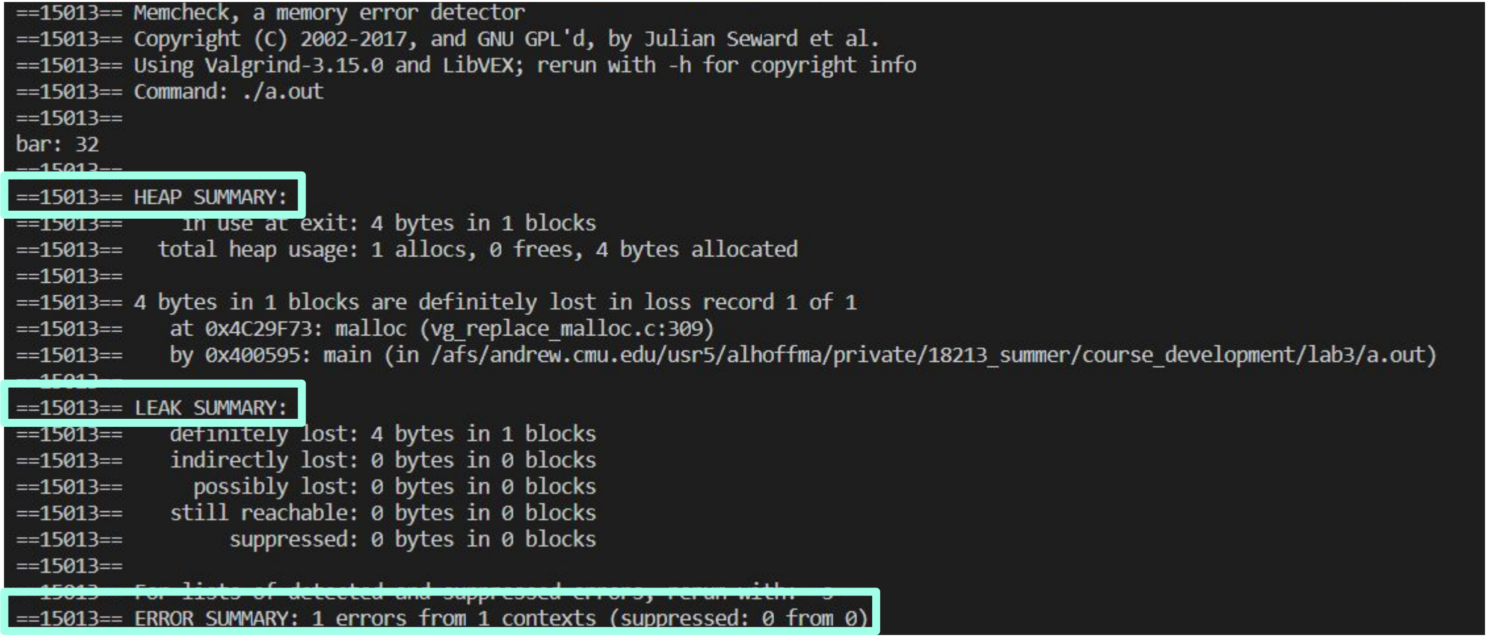
Indirectly Lost

- Block is “lost” because the blocks that point to it are themselves lost

Possibly Lost

- Pointer exists but it points to an internal part of the memory block

Memory Leaks Sample Output



```
==15013== Memcheck, a memory error detector
==15013== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15013== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==15013== Command: ./a.out
==15013==
bar: 32
==15013==
==15013== HEAP SUMMARY:
==15013==     in use at exit: 4 bytes in 1 blocks
==15013==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==15013==
==15013== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==15013==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==15013==    by 0x400595: main (in /afs/andrew.cmu.edu/usr5/alhoffma/private/18213_summer/course_development/lab3/a.out)
==15013==
==15013== LEAK SUMMARY:
==15013==    definitely lost: 4 bytes in 1 blocks
==15013==    indirectly lost: 0 bytes in 0 blocks
==15013==      possibly lost: 0 bytes in 0 blocks
==15013==    still reachable: 0 bytes in 0 blocks
==15013==         suppressed: 0 bytes in 0 blocks
==15013==
==15013== For lists of detected and suppressed errors, rerun with: -s
==15013== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Debugging Everything: GDB

What is GDB?

- GNU Debugger
- Powerful debugger that lets you inspect your program as it's executing
- Allows you to see what is going on 'inside' another program
- Breaks abstraction between program and machine

Why use GDB?	
When to Use	When Not to Use
● Complicated code that you need to step through ● Need to find values at specific points ● Valgrind was not helpful ● To inspect machine state	NOTE: This is intentionally left blank (Often Super Useful!)

GDB Takeaways

- GDB is a powerful debugger that has the capabilities to
  - **Set breakpoints** stop at line of code
  - **Set watchpoints** stop when variable changes
  - **Print values**
  - **Step through execution**
  - **Backtrace** see previous function calls
- These capabilities will be useful for debugging general code in 213
  - GDB has many functionalities beyond these slides, check out this link for more features
    - https://sourceware.org/gdb/current/onlinedocs/gdb/

Starting GDB

Helpful Resource:
https://sourceware.org/gdb/current/onlinedocs/gdb/

- You can open gdb by typing into the shell:
  ```
  $ gdb
  (gdb) run 15213  // run program
  ```
- Type gdb and then a binary to specify which program to run
  ```
  $ gdb <binary> ($ gdb ./a.out)
  ```
- You can optionally have gdb pass any arguments after the executable file using --args
  ```
  $ gdb --args gcc -O2 -c foo.c
  ```
- Quitting GDB:
  ```
  (gdb) quit [expression]
  (gdb) q
  ```
  - or type an end-of-file character (usually Ctrl-d)
- More GDB options and help:
  ```
  $ gdb -help  OR $ gdb -h
  ```

GDB Commands

Controlled Program Execution

- `(gdb) CTRL + c`: stops execution
- `(gdb) next (n)`: run next line of program and does NOT step into functions
  - `(gdb) next  X (n X)`: run next X lines of function
  - `(gdb) nexti`: run next line of assembly code and does NOT step into functions
- `(gdb) step (s)`: run next line of program AND step into functions
  - `(gdb) step X (s X)`: step through next X lines of function
  - `(gdb) stepi`: step through next line of assembly code
- `(gdb) continue (c)`: continue running code until next breakpoint or error
- `(gdb) finish (f)`:run code until current function is finished

Connecting Execution with Code

- **`(gdb) disassemble (disas)`** : disassemble source code into assembly code
  - **NOT dis:** dis == disable breakpoints
- **`(gdb) list (l)`** : list 10 lines of source code from current line
  - **`(gdb) list X (l X)`** : list 10 lines of source code from line number X
  - **`(gdb) list fnName (l fnName)`** : list 10 lines of source code from fnName function

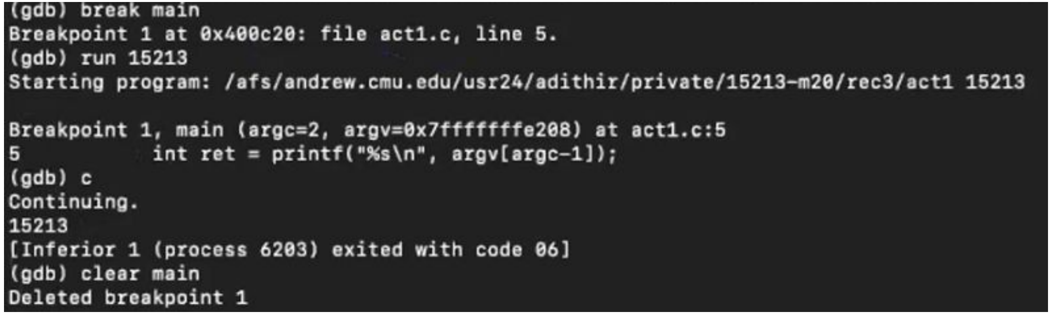
Breakpoints

- A breakpoint makes your program stop whenever a certain point in the program is reached
- `(gdb) break function_name`: breaks once you call a specific function. (`break` abbreviated `b`)
- `(gdb) break *0x...`: breaks when you execute instruction at a certain address
- `(gdb) info b`: displays information about all breakpoints currently set
- `(gdb) disable #`: disables breakpoint with ID equal to # **(`$disa` is short form not `$disas`!!!)**
- `(gdb) clear [location]`: delete breakpoints according to where they are in your program.

Setting breakpoint →

Breakpoint hit →

Breakpoint deleted →



```
(gdb) break main
Breakpoint 1 at 0x400c20: file act1.c, line 5.
(gdb) run 15213
Starting program: /afs/andrew.cmu.edu/usr24/adithir/private/15213-m20/rec3/act1 15213

Breakpoint 1, main (argc=2, argv=0x7fffffffe208) at act1.c:5
5           int ret = printf("%s\n", argv[argc-1]);
(gdb) c
Continuing.
15213
[Inferior 1 (process 6203) exited with code 06]
(gdb) clear main
Deleted breakpoint 1
```

Watchpoints

- A special breakpoint that stops your program when the value of an expression changes
  - The expression may be a value of a variable, or involve values combined by operators
- Enable, disable, and delete both breakpoints and watchpoints
- `(gdb)delete [watchpoint]`:delete individual breakpoints/ watchpoints by specifying breakpoint numbers
  - If no argument is specified, delete all breakpoints , `(gdb)d`

Examples:

- `(gdb)watch foo:` watch the value of a single variable
- `(gdb)watch *(int *)0x600850:` watch for a change in a numerically entered address

(output)
```
Watchpoint 1: *(int *)6293584
```

Printing Values & Inspecting Memory

Printing Values

- **`(gdb) print (p) [any valid expression]`**
  - Print local variables or memory locations
  - Be sure to cast to the right data type
    - (e.g. p *(long*)ptr)
  - **`(gdb) print (p) *pntr`**: prints value of pointer
  - **`(gdb) print (p) *(struct_t*) tmp`**: casts tmp to struct_t* and prints internal values
- **`(gdb) print (p) expr`**: prints value of data type

Inspecting Memory

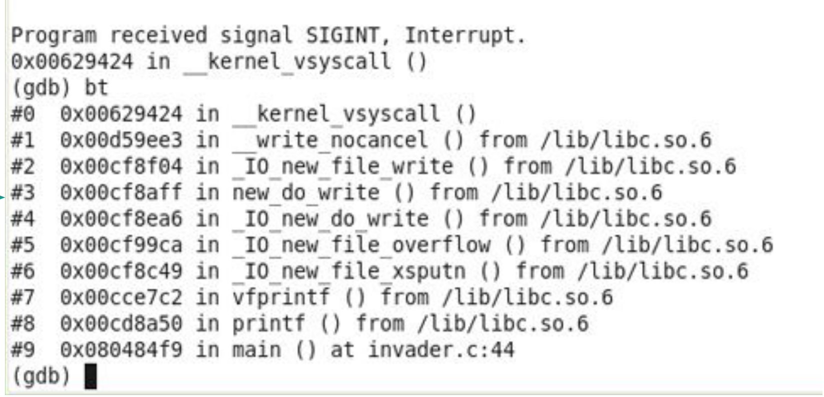
- **`(gdb) x/nfu [memory address]`**: equivalent to **`(gdb) print *(addr)`**
  - **`n`**: inspect next n units of memory
  - **`f (format)`**: can be represented as:
    - d (decimal), x (hexadecimal), s (string)
  - **`u (unit)`**: can be represented as:
    - b (bytes), w (words/ 4 bytes)

These are just some common ways to inspect memory and print values, check the resources links for more uses

Backtrace

- `(gdb) backtrace (bt)`: prints a summary of how program got where it is
  - Print sequence of function calls that led to this point
  - Helpful to use when programs crash
- `(gdb) up N (u N)`: go up N function calls
- `(gdb) down N (d N)`: go down N function calls

Previous
“frames”



```
Program received signal SIGINT, Interrupt.
0x00629424 in __kernel_vsyscall ()
(gdb) bt
#0  0x00629424 in __kernel_vsyscall ()
#1  0x00d59ee3 in __write_nocancel () from /lib/libc.so.6
#2  0x00cf8f04 in _IO_new_file_write () from /lib/libc.so.6
#3  0x00cf8aff in new_do_write () from /lib/libc.so.6
#4  0x00cf8ea6 in _IO_new_do_write () from /lib/libc.so.6
#5  0x00cf99ca in _IO_new_file_overflow () from /lib/libc.so.6
#6  0x00cf8c49 in _IO_new_file_xsputn () from /lib/libc.so.6
#7  0x00cce7c2 in vfprintf () from /lib/libc.so.6
#8  0x00cd8a50 in printf () from /lib/libc.so.6
#9  0x080484f9 in main () at invader.c:44
(gdb) 
```

Calling Functions & Changing Values

Calling your program's functions

- Examples:
- `(gdb) call` ***`expr`***`:` Evaluate the expression expr without displaying void returned values.

Changing values:

- **`(gdb) set [variable] expression:`** change the value associated with a variable, memory address, or expression
  - Evaluates the specified expression. If the expression includes the assignment operator ("="), that operator will be evaluated and the assignment will be done.
- The only difference between the **`set`** variable and the **`print`** commands is printing the value

→ Will be useful later...

Lab Time!

https://tinyurl.com/y6ca8kea

Feedback:

https://tinyurl.com/213bootcamp3

Resources

https://www.tutorialspoint.com/gnu_debugger/index.htm

https://sourceware.org/gdb/current/onlinedocs/gdb/ [scroll down for more information]