CSE 333 Section

Section 2
Office Hours

• Professor Zahorjan:
  – 12:40 pm – 1:30 pm on Fridays
  – By appointment
• Renshu
  – 1:30 pm – 2:30 pm on Tuesdays
• Matthew
  – 12:00 pm – 1:00 pm on Thursdays
• Ricardo
  – 2:00 pm – 3:00 pm on Thursdays
• Zhitao (Reid)
  – 3:30 pm – 4:30 pm on Wednesdays

Do these work for you?
A (Incomplete) Recap Since Last Time

• Build Process (Preprocessor, Compiler, Linker)
• Header Files (and why we use them)
• Variables
  – Declaration vs. Definition
  – Where they live and when this is known
• Types
• C call and return style
• Static keyword on functions
• Structs (more tomorrow)
Any Questions?

• Course Material?
• Exercises?
• Homework?
Upcoming Due Dates

• Exercise 2 due @ 11:15 am on Friday, Oct. 9th
• Homework 0 due @ 11:59 pm on Monday, Oct. 12th

• Homework 1 out soon!
How Do You Debug?

• Print statements?
• Compare against known ‘good’ output?
• Debugging tool?
gdb—It’s like 351 again, but optional

• A common debugger for C/C++ (and many other languages)

• Let’s go through a quick gdb refresher
gdb—Getting Started

- Starting gdb: `gdb`
- Running gdb: `run <arg1> <arg2> <arg3> ...`
- Stopping gdb: `kill`
- Exiting gdb: `quit`
gdb – Breakpoints

• Setting a breakpoint : b(reak)
  – b <function name>
  – b <line #>
  – b <memory address>

• Delete a particular breakpoint : d(lete) <breakpoint #>

• Delete all breakpoints: d(lete)
gdb – More Breakpoints

• Show breakpoints: info break
• Conditional Breakpoints:
  condition <breakpoint #> <boolean condition>
  * This doesn’t work on attu = ( *
• If you are working with multiple files you can specify the file to break in by prefixing the line number with ‘<file name>.c:’
gdb – Moving through code

• s(tep) : Execute one line of code; fall into functions
  – Optionally provide a # of lines to skip

• n(ext) : Execute one line of code; skip over functions
  – Optionally provide a # of lines to skip
gdb – Inspecting Code

• `p(rint) <what>`: Prints what
  – Can print:
    • Primitives
    • Structs
    • Pointers
    • ...

• `disp(lay) <what>`: Prints what after every pause
  – Use ‘info display’ to see what is being displayed
  – Use `undisp(lay) <display #>` to stop displaying

Note: There are a number of display modifiers (I.E. decimal vs. hex)
gdb – Other Useful Commands

• bt (backtrace) – Current address and stack backtrace
• set variable <name> <value> - Allows you to change the value of a variable during execution
• call <function name>(arg0, arg1, ...) - Allows you to call a function while debugging
gdb – Other Useful Commands

• define <macro name> : Allows you define a macro during debugging
gdb – Useful Facts

• gdb does not require that you recompile your code after a change.
• Use `gdb –tui` for visual component of gdb
Valgrind

• Check for memory utilization
  – During execution
  – At termination

• Invoking Valgrind: valgrind <program name>
  – Useful options:
    • --leak-check=full : For each block check to see if it is reachable by using registers or pointers in-memory
    • --show-reachable=yes : Show all kinds of leaks
Understanding Valgrind Output

Basic Valgrind Usage

**Command**

```
valgrind ./a.out
```

**Example Output**

```
==26428== Memcheck, a memory error detector
==26428== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==26428== Using Valgrind-3.7.0 and LibVEX; rerun with --h for copyright info
==26428== Command: ./a.out
==26428==
==26428== LOTS OF ERRORS
==26428==
==26428== HEAP SUMMARY:
==26428== in use at exit: 528 bytes in 22 blocks
==26428== total heap usage: 22 allocs, 0 frees, 528 bytes allocated
==26428==
==26428== LEAK SUMMARY:
==26428== definitely lost: 408 bytes in 11 blocks
==26428== indirectly lost: 120 bytes in 11 blocks
==26428== possibly lost: 0 bytes in 0 blocks
==26428== still reachable: 0 bytes in 0 blocks
==26428== suppressed: 0 bytes in 0 blocks
==26428== Rerun with --leak-check=full to see details of leaked memory
==26428==
==26428== For counts of detected and suppressed errors, rerun with: -v
==26428== Use --track-origins=yes to see where uninitialised values come from
==26428== ERROR SUMMARY: 65 errors from 16 contexts (suppressed: 3 from 3)
```
Understanding Valgrind Output

• Definitely Lost – The programmer could not possibly have freed the block of memory because no pointer exists.

• Indirectly Lost – A block is lost because all the blocks that have pointers to it are lost.

• Possibly Lost – At least one pointer to the block was found; but at least one is an internal-pointer.

• Still Reachable – A path to the block exists; programmer could free this data.
Understanding Valgrind Output

Illegal reads/writes

Code

```c
#include "stdlib.h"
#include "stdio.h"

int main(int argc, char *argv[]) {
    int *x = (int *)malloc(sizeof(int));
    x += 2; // x now points to invalid memory (some random location).
    printf("%d\n", *x); // XXX Reading to an invalid location of memory.
    *x = 4;
    // XXX Writing to an invalid location of memory.
    free(x - 2);
    printf("%d\n", *((int *)3838338)); // XXX And even worse read.
    return EXIT_SUCCESS;
}
```

==3023== Invalid read of size 4
==3023== at 0x400502: main (error.c:6)
==3023== Address 0x51d2048 is 4 bytes after a block of size 4 alloc'd
==3023== at 0x4c2a93d: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3023== by 0x400584: main (error.c:4)
==3023==
==3023== Invalid write of size 4
==3023== at 0x4005a9: main (error.c:7)
==3023== Address 0x51d2048 is 4 bytes after a block of size 4 alloc'd
==3023== at 0x4c2a93d: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3023== by 0x400584: main (error.c:4)
==3023==
==3023== Invalid read of size 4
==3023== at 0x4005c4: main (error.c:9)
==3023== Address 0x3a9182 is not stack'd, malloc'd or (recently) free'd
Understanding Valgrind Output

Use of uninitialized value

Code

```c
#include "stdlib.h"
int main(int argc, char *argv[]) {
    int *x;
    *x = 4; // XXX Using x before initialized.
    return EXIT_SUCCESS;
}
```

Valgrind Output

```text
==2205== Use of uninitialized value of size 8
==2205==     at 0x4004AB: main (error.c:4)
```
Understanding Valgrind Output

**Illegal frees**

```c
#include "stdlib.h"
int main(int argc, char *argv[]) {
    free((void*) 0xdeadbeef); // XXX free some random address free'd.
    int *x = (int*)malloc(sizeof(int));
    free(x+4); // XXX free outside malloc'd block.
    free(x);
    return EXIT_SUCCESS;
}
```

**Valgrind Output**

```plaintext
==2978== Invalid free() / delete / delete[] / realloc()
==2978==    at 0x4C29A9E: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2978==    by 0x400544: main (error.c:3)
==2978==     Address 0xdeadbeef is not stack'd, malloc'd or (recently) free'd
==2978==
==2978== Invalid free() / delete / delete[] / realloc()
==2978==    at 0x4C29A9E: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2978==    by 0x400562: main (error.c:6)
==2978==     Address 0x51d2050 is 12 bytes after a block of size 4 alloc'd
==2978==    at 0x4C2A93D: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2978==    by 0x40054E: main (error.c:5)
```
Resources

- ‘man gdb’
- ‘info gdb’
- gdb Cheat sheet - [https://courses.cs.washington.edu/courses/cse351/15sp/supp/gdbnotes-x86-64.pdf](https://courses.cs.washington.edu/courses/cse351/15sp/supp/gdbnotes-x86-64.pdf)
Resources