Recitation 9/18

Welcome back!

Today’s Topics

- GDB
- Structs
- Valgrind
- Makefiles
- Q&A if time

GDB Quick Facts

- C cannot be debugged like Java can with the IDE debugger(think IntelliJ from 1200)
- Instead we use: gdb (GNU Debugger) for debugging
  - Very useful in tracking undefined behavior and state of variables

Segmentation Fault?

- Segmentation Fault
    - C doesn’t tell you much when it crashes, usually just prints: “Segmentation fault (Core Dumped)”
- Causes:
    - Dereferencing an uninitialized pointer
    - Dereferencing NULL
    - Using a previously freed pointer
    - Writing beyond the bounds of an array
    - Literally anything
    - …
- GDB is incredibly useful for debugging a segmentation fault

Running GDB on a Program

- Open terminal in the folder the executable is in
- Run “gdb ./[executable]”
- Enter “l” (lowercase L) to see the code, or use “tui enable” to get a nice GUI
  - Tui = text user interface, shows a scrollable code page and your break points
- Enter “break [line number]” to stop the executable before that line number
- Type “run [command args]” to run the program
- Use “next” to pass over the next line(will pass over function calls)
- Use “step” to go to the next line, will go inside a function of the line you are on
- Use “continue” to run to the next break point
- Use “print [variable]” or “p [variable]” to see the value of a variable

GDB “Cheat Sheet”

- run <command_line_args>
  - Runs the program with specified command line arguments
- backtrace
  - Prints out the “trace” of where functions were invoked to get to the current spot in the program
- up/down
  - Can be used to look at the function who called us/we are calling
- print <expression>
  - Prints out a value so that we can examine it
- quit
  - Quit the program

GDB “Cheat Sheet” Part 2

- tui enable
  - Used to enable the Text User Interface
- step
  - Move forward a line, steps into a function if we call one
- next
  - Moves forward a line, doesn’t step into a function if called
- continue
  - Run until we crash, hit a breakpoint, or program finishes
- breakpoints
  - Next slide

gdb breakpoints

- Usage:
  - break <function_name>
  - break <filename:line#>
    - Example: `break main.c:20`
  - info break
    - Prints out information of all breakpoints
  - del <id>
    - Deletes the breakpoint with specified num.
    - Get breakpoint num with `info break`

One last thing: printing arrays

- We saw print <expression>, which works for basic variables, but it can also be used for arrays
- Given an array named “my_array” and length = len:
    - print *my_array@len
    - Very helpful for printing out an array that is represented as a pointer

Makefiles!

This is mostly about **writing** Makefile btw

Makefiles - First of All, Why?

- Not needed if your project is one C file, just put command in terminal
- But what if your project is big, with many modules that depend off each other?

Example: PennPals from CIS 1200… but in C?

- PennPals = a server tracking multiple chat rooms
- Users, admins, server backend, protocols, chat rooms, and main are individual components of the project that can be split into different files for organization
- Chat room management involves both users and admins
- Server is composed of multiple chat rooms and protocols

Makefiles - First of All, Why?

- Not needed if your project is one C file, just put command in terminal
- But what if your project is big, with many modules that depend off each other?

Example: PennPals from CIS 1200… but in C?

- PennPals = a server tracking multiple chat rooms
- Users, admins, server backend, protocols, chat rooms, and main are individual components of the project that can be split into different files for organization
- **Chat room management involves both users and admins**
- Server is composed of multiple chat rooms and protocols

Makefiles - Why???

- If you’re debugging only **chatroom.c**, do you need to recompile all 5+ files to update the project?
- In this same scenario (**chatroom.c**), do you only need to recompile **chatroom.c**?

Makefiles - just get to the point already!

Makefiles track each file’s dependencies

- If one file changes, all other files that use that file also need to be recompiled
- Makefile keeps track of that so we don’t have to remember

Components of a Makefile

Makefile is made of **rules**

Rules look something like this:

```
target: prerequisites
    command
    command
    command
```

`target:` the file we are making using this rule

`dependencies:` Makefile needs to make sure these files are up to date before it can compile target

`command:` Makefile will run these in order to get/compile the target

In this example, `dependencies` could be just one file or a list of files separated by spaces (ie: `prereq-0 prereq-1 prereq-2`)

How do I know what rules to add?

- Generally you want a **clean** rule, you will tell it to remove files that are listed as `target` in the Makefile
    - This is so when you run “make” again, it will recompile everything

```
clean:

    rm *.o, executible_1, executible_2, executible_3
```

- In this week’s homework, we specify which rules you should add
    - If target `x` depends on `a`, `b`, and `c,` make sure you include 3 additional rules where `a`, `b`, and `c` are each the target

Dependencies: which files do I choose? (.c, .h, .o, executable)

1. Most obvious dependency is: “where is your file being compiled from?”
    - **chatroom.c** compiles to **chatroom.o**, which compiles to **chatroom** (executable file)
    - Therefore, **file.o** depends on **file.c**, and **file** (executable) depends on **file.o**
2. In partial compilation, **file.o** also depends on **file.h**
3. Non-system **`#include`** statements on top of the .c file corresponding to target

Example: if **chatroom.c** contains **`#include “user.h”`** and **`#include “admin.h”`**, then:

- **chatroom.o** also depends on **user.h** and **admin.h**
- **chatroom** (the executable) also depends on **user.o** and **admin.o**
- If **chatroom.c** also contains **`#include <stdlib.h>`**, it’s not a dependency you need to include

Makefile **commands**

In your single-file compilation command, you do both compilation and linking in one step: **`clang-15 -g3 -gdwarf-4 -Wall -o file file.c`**

- Went straight from **file.c** to **file** (executable) without explicitly calling **file.o**

When multiple files (and dependencies) are involved, you split compiling and linking into 2 separate commands

Makefile **commands** Example

Compile **chatroom.c** into **chatroom**. **chatroom.c** uses methods and structs defined in **user.c** and **admin.c**

(Partial) Compiling:

```
chatroom.o: chatroom.c chatroom.h user.h admin.h
    clang-15 -g3 -gdwarf-4 -Wall -c chatroom.c
```

Linking:

```
chatroom: chatroom.o user.o admin.o
    clang-15 -g3 -gdwarf-4 -Wall -o chatroom chatroom.o user.o admin.o
```

Makefile tips

- Ensure your indents are all tabs and not spaces, otherwise Makefile won’t compile
- Draw a dependency DAG! The file containing main method will be the source, and arrows will be drawn from **`target`** to **`dependency`**