# CS 238P Operating Systems Discussion 1

Based on slides from Saehanseul Yi (Hans)

# About us

Professor:

Anton Burtsev

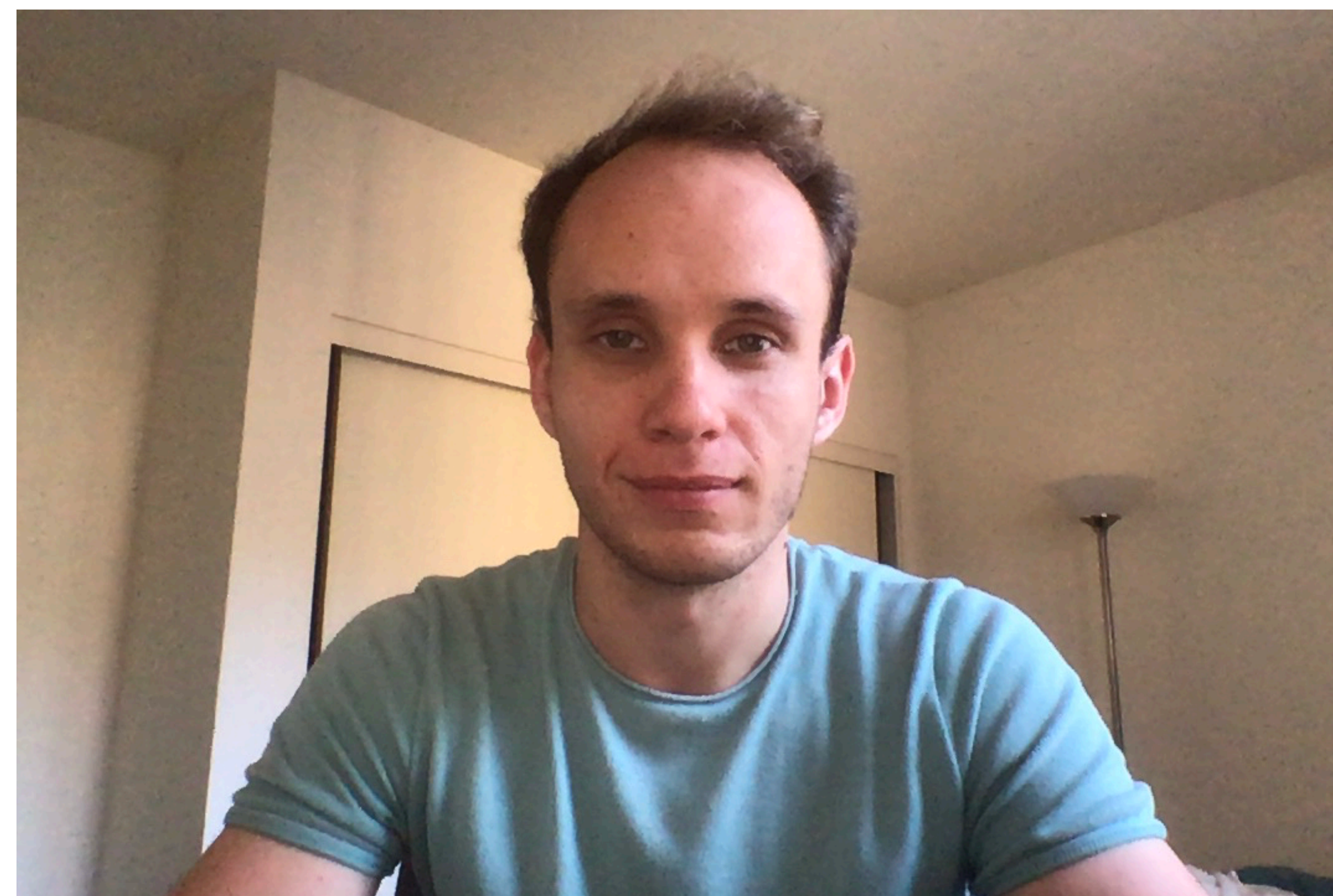
aburtsev@uci.edu

TAs:

Fedor Zaitsev

(Fred)

fzaytsev@uci.edu



Harishankar Vishwanathan

(Hari)

hvishwan@uci.edu

# Today’s agenda

- Introduction to the class
- Intro to Linux
- Intro to pointer
- Basic GDB

# Where I can find materials?

- For lecture videos, discussion videos, assignments: https://www.ics.uci.edu/~aburtsev/238P/
- For questions: Piazza https://piazza.com/uci/spring2020/cs238p/home
- Virtual office hours (starting next week): TBD via zoom (check prof. website)

# Regrades

- Grading for quizes would be done on Canvas. For anything else - Gradescope
- Regrade relating quizes/exams: please submit regrade request on Gradescope
- Regrade relating for programming homework: please create a private post on Piazza and explain why you believe you deserve better grade

# Are discussions mandatory?

No

In next discussions we are planning to spend time mostly on solving homework

# How I can get machine to solve homework?

Openlab

- Available only on campus or via VPN (https://www.oit.uci.edu/help/vpn/)
- 48 machines (circinus-1.ics.uci.edu ~ circinus-48.ics.uci.edu)

# Openlab: How to access

- For windows 8 and below: download PuTTY
- For Windows 10, MacOS, Linux: just open terminal
- ssh UCInetID@openlab.ics.uci.edu
- Example: ssh panteater@openlab.ics.uci.edu
- It would ask you for password. Just type it

# Openlab: Troubleshooting

- What if I have some weird problems with access/server?
  - Use another server directly: circinus-1.ics.uci.edu to circinus-48.ics.uci.edu. Example: ssh panteater@circinus-13.ics.uci.edu
  - Or visit: https://www.ics.uci.edu/~lab/students/

# Welcome to Linux!

```
/
├── auto
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── extra
├── home
├── lib -> usr/lib
├── lib64 -> usr/lib64
├── lv_scratch
├── media
├── mnt
├── net
├── opt
├── pkg
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── scratch -> lv_scratch/scratch
├── srv
├── sys
├── tmp -> lv_scratch/tmp
├── usr
└── var
```

```
/home
├── klefstad
├── saehansy
├── sana
├── siyuew1
└── wayne
```

```
$ pwd
/home/saehansy
```

- /: **root** directory
  The "path" always starts with /
- In a path, directories are separated with /
- After login, you will be at your home directory
  /home/UCNetID
- First command:
  pwd
  **P**rint **W**orking **D**irectory

# Welcome to Linux!

- man <command>: **manual** for the command
- E.g. man pwd

```
PWD(1)                        User Commands                        PWD(1)

NAME
       pwd - print name of current/working directory

SYNOPSIS
       pwd [OPTION]...

DESCRIPTION
       Print the full filename of the current working directory.

       -L, --logical
              use PWD from environment, even if it contains symlinks

       -P, --physical
              avoid all symlinks

       --help display this help and exit

       --version
              output version information and exit

       NOTE: your shell may have its own version of pwd, which usually supersedes the ver-
```

# Linux commands: Navigation

| Command | Short for … | Description |
|---|---|---|
| pwd | Print Working Directory | Current working directory |
| ls | List | List files and directories |
| cd | Change directory | go to home directory |
| cd .. | | go out to parent directory |
| cd <directory_name> | | go inside the directory |

**./ (dot followed by a slash):** means the current directory (**relative path**).
An **absolute path** is the path starts from the root directory. i.e. /home/UCNetID

Exercise) Go to the root directory and then come back to your home directory

1) cd ..; cd ..; cd ./home; cd ./UCNetID
2) cd /; cd
3) cd /; cd /home/UCNetID

# Linux commands: File handling

| Command | Short for … | Description |
|---|---|---|
| mkdir <dir_name> | Make directory | |
| touch <file_name> | | Create an empty file(0 in size) |
| mv <source> <dest> | move | move files(dirs.) or rename |
| cp <source> <dest> | copy | copy files(dirs.) + rename |
| rm <file_name> | remove | remove file |
| rm –r <dir_name> | remove recursively | remove directories |

**Note: *rm* is not reversible; no way to recover the files! Be careful**

Exercise) create an empty file and check if it exists using ‘ls’. Delete that file after.

Sol) touch empty; ls; rm empty

Examples)
1) mv ./empty ../ : move the file ‘empty’ to the parent directory
2) mv ./empty ./empty2: rename the file ‘empty’ to ‘empty2’

# Linux commands: File read & write

| Command | Short for … | Description |
|---|---|---|
| cat <file_name> | concatenate | prints the file content |
| vi <file_name> | visual | opens a text editor |
| emacs <file_name> | | C-x C-c to quit |
| nano <file_name> | | C-x to quit |
| | | |
| | | |

# Source code editor: vi

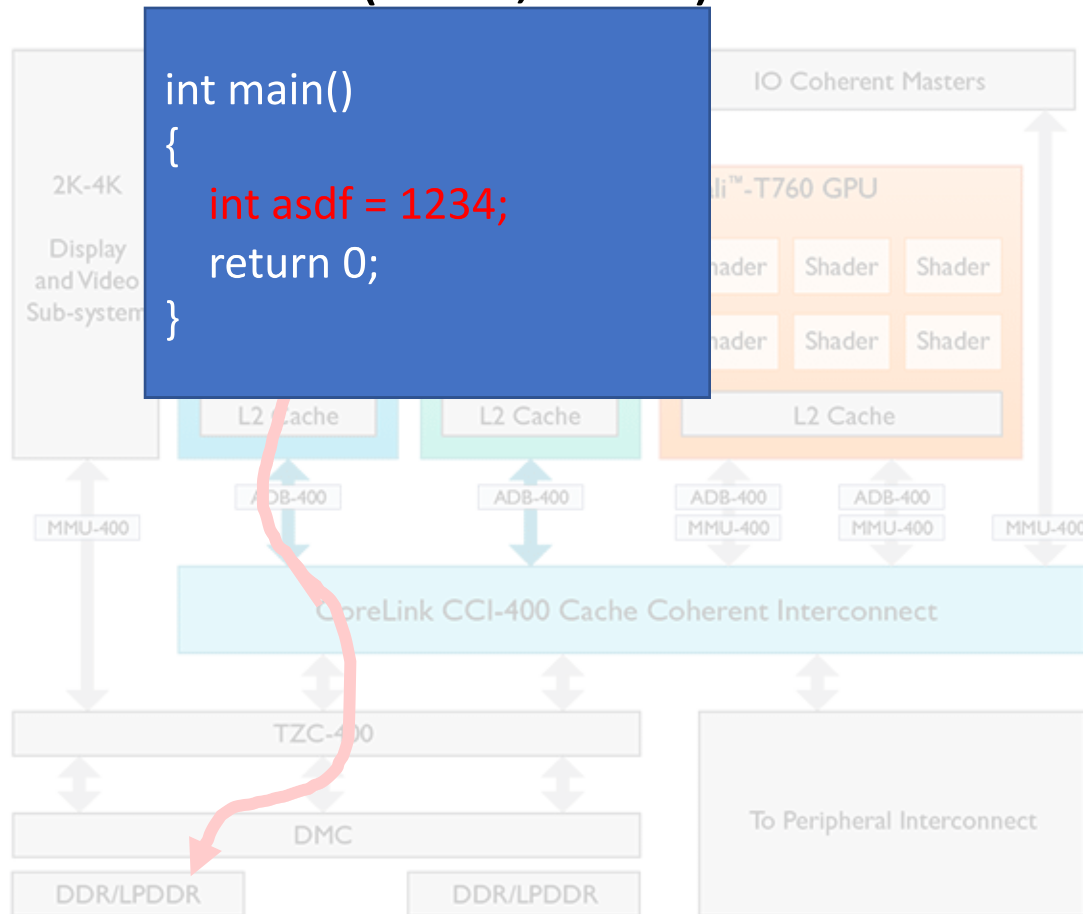
- Two modes: **insert** and **normal**(read only & navigation)
- When you first launch *vi*, it's in **normal** mode where you can press many hot keys
- if you press 'i' in normal mode, it goes into **insert** mode where you can type any text
- Press ESC will make it go back to **normal** mode
- For now, use 'i', ESC, and arrow keys to edit a file
- To quit type '**:**' and then '**q!**'(quit w/o saving) or '**wq**'(write-and-quit) in **normal** mode
- For more info, https://openvim.com/

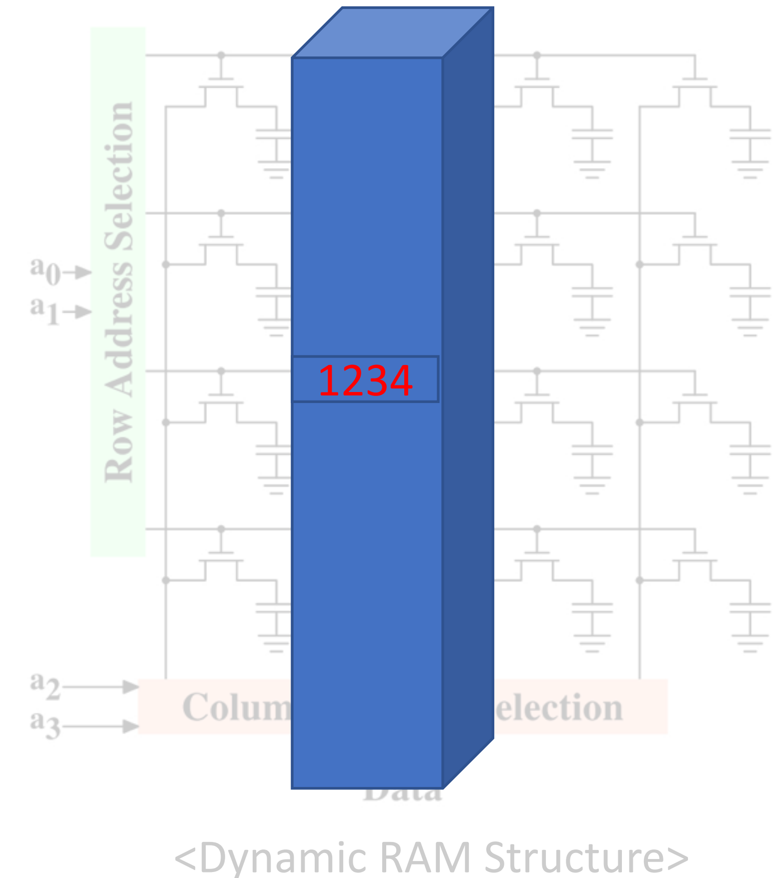
# Source code editor: vi

- Exercise
  - Open a file vi_practice.txt
  - Type your name and UCINetID
  - Save and quit (:w and then :q or :wq)
  - View the file contents
  - Delete the file

# C pointers

- How data move(read,write) between memory and processors?




<Dynamic RAM Structure>

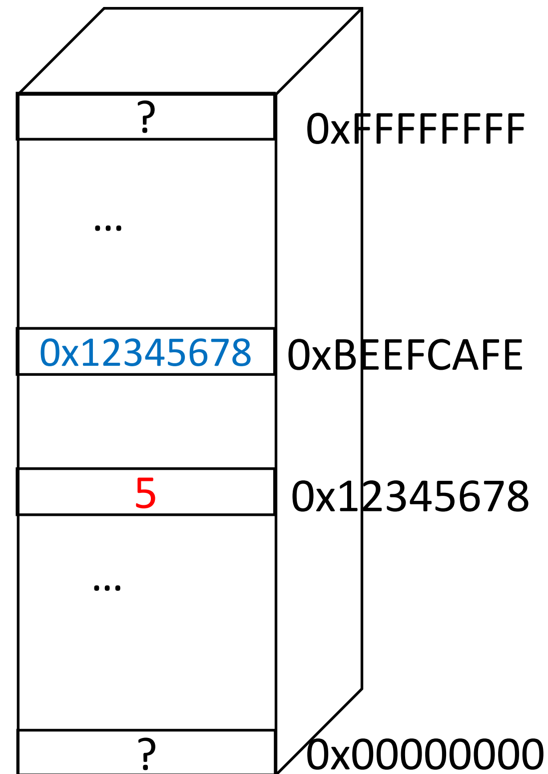
# C pointers

- Pointer: a way to access memory(?) programmatically
- Why?
  - A full control of your computer
  - Optimize program to run faster or use less memory
- Computers these days are becoming more powerful
  - Modern programmers do not need such fine control
  - Only few languages provide pointers: C, C++, and C#
- Operating Systems need such fine control
  - Every program on a computer shares the functionality defined in OS
  - OS bridges the hardware and software (uses memory addresses)

# C pointers

```
#include <stdio.h>

int main()
{
   int temp = 5;
   int *pTemp = &temp;
}
```




- The program sees memory as **1D array!**
- This system is 32-bit because the address is 4 bytes
  - 2 hex digits = 1 byte
  - For 64-bit system, memory address is 8 bytes

# C pointers

- **Declaration** (* after the type)
  - TYPE *VAR_NAME = INIT_VAL;
  - int * pTemp = NULL;
  - int* pTemp = NULL;
  - **int *pTemp = NULL;**
- **Reference operator (&):** returns the address of the operand(variable)
  - pTemp = &temp;
- **Dereference operator (* before the variable name):** returns the value
  - int new_val = *pTemp
  - printf (“%d”, *pTemp) → 5
  - printf (“%d”, new_val) → 5
  - printf (“%p”, pTemp) → 0x12345678

# C pointers

- Pointer has a type

```
int temp = 5;
float *fp_temp = &temp; (error)
void *vp_temp = (void *) temp; (ok)
char *cp_temp = 'a';
```

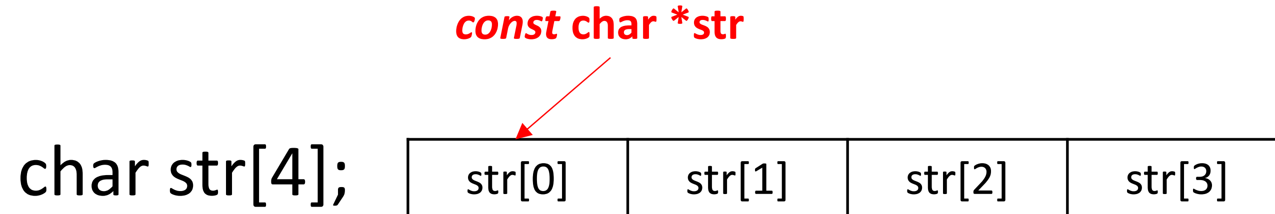
- Increment & decrement

```
int *ip_temp = 0x10001000;
ip_temp+1 ?
printf("%p", ip_temp + 1) → 0x10001004 (not 0x10001001)

char *cp_temp = 0x10001000;
cp_temp+1 ?
printf("%p", ip_temp + 1) → 0x10001001 (not 0x10001004)
```

# C pointers

- Pointer == array?




- The variable ‘str’ points to the start of the array (*const* means it cannot point anywhere else; it’s fixed)

```
char str[4] = {0,};
printf (“%p\n”, &str[0]);
printf (“%p\n”, str);
```

- str[0] == *(str)
  str[1] == *(str + 1)

# C pointers

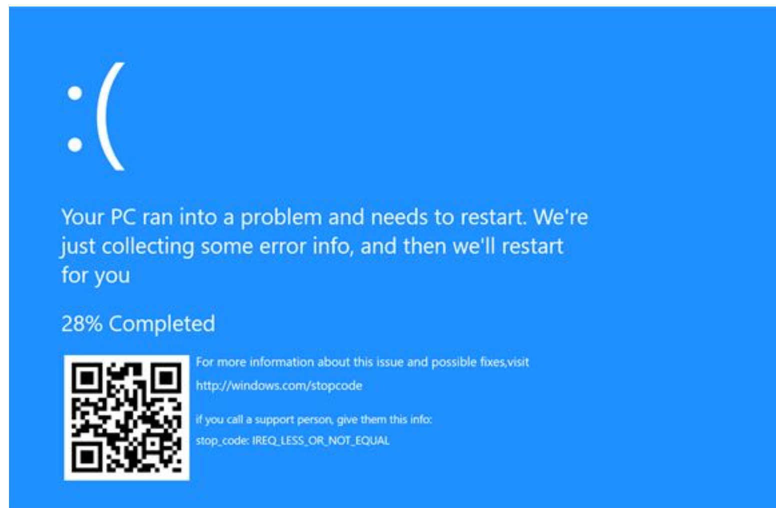
- 2D pointer

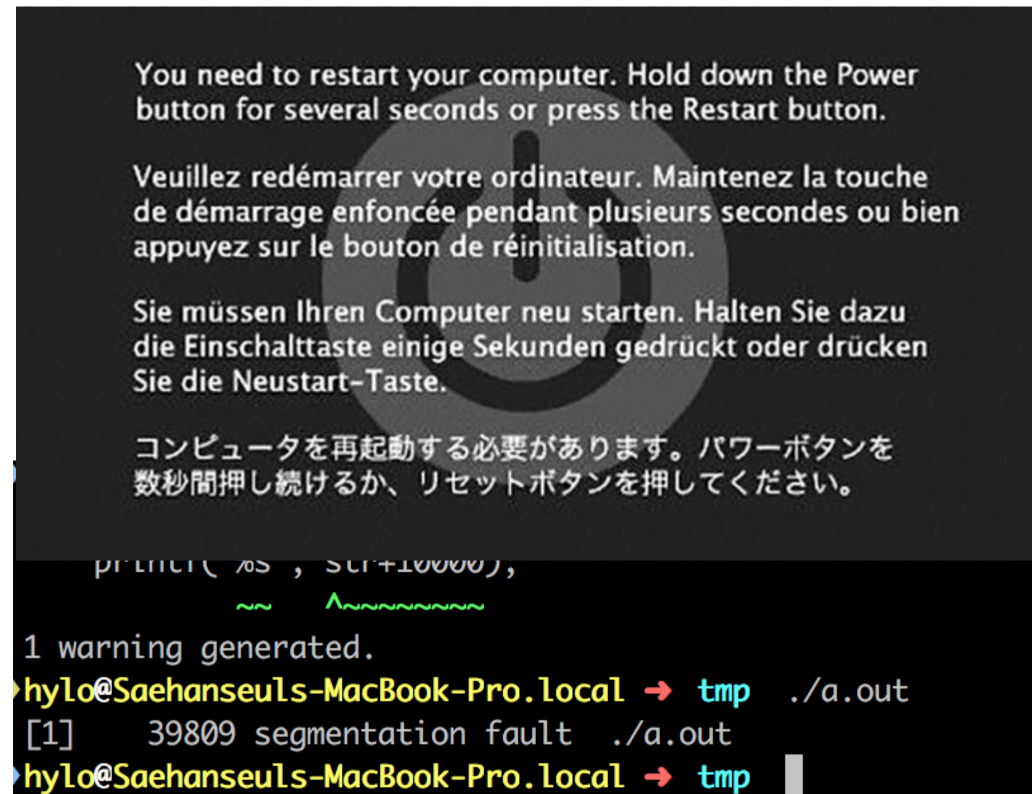
```
char str[2][3] = {0,};                0xef21 e8e6
printf ("%p\n", &str[0]);             0xef21 e8e6
printf ("%p\n", &str[1]);             0xef21 e8e9
printf ("%p\n", &str[1][0]);          0xef21 e8e9
printf ("%p\n", &str[2]);             0xef21 e8ec
printf ("%p\n", &str[2][0]);          0xef21 e8ec
printf ("%p\n", &str[2][1]);          0xef21 e8ed
```

- 3D pointer?

# C pointers

- Pointers are dangerous…
- BSoD(Blue screen of death) == Kernel panic ← Segmentation Fault




- How to debug? **GDB**

# Simple C Program

```
#include <stdio.h>

int main()
{
    char str[2][3] = {0,};

    printf ("%p\n", str);
    printf ("%p\n", &str[0]);
    printf ("%p\n", &str[1]);
    printf ("%p\n", &str[1][0]);
    printf ("%p\n", &str[2]);
    printf ("%p\n", &str[2][0]);
    printf ("%p\n", &str[2][1]);

    return 0;
}
```

```
gcc –g test.c -o test.exe
./test.exe
gdb test.exe
```

# GNU Debugger(GDB)

- Control the execution flow of the program (stop/resume)
- View/modify the system status (register, memory contents, …)
- Run the target(inferior) inside gdb or *attach* to the running process
- Even remote debugging is possible (through network)

# GNU Debugger(GDB)

- Check debug information
  - l (or list)

```
list
list <filename>:<function>
list <filename>:<line_number>
```

```
(gdb) l
1       #include <stdio.h>
2
3       int main()
4       {
5       char str[2][3] = {0,};
6       printf ("%p\n", str);
7       printf ("%p\n", &str[0]);
8       printf ("%p\n", &str[1]);
9       printf ("%p\n", &str[1][0]);
10      printf ("%p\n", &str[2]);
(gdb) list
11      printf ("%p\n", &str[2][0]);
12      printf ("%p\n", &str[2][1]);
13      return 0;
14      }
15
16
```

# GNU Debugger(GDB)

- breakpoint: stop the program at certain point
  - where?
    - a line of the source code
    - or at specific memory address
- info b: list breakpoints
- delete <num>

```
(gdb) break 5
Breakpoint 1 at 0x400525: file test.c, line 5.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400525 in main at test.c:5
(gdb) delete 1
(gdb) info b
No breakpoints or watchpoints.
(gdb) break 5
Breakpoint 2 at 0x400525: file test.c, line 5.
(gdb) run
Starting program: /home/saehansy/Workspace/ics143a/FQ19/test.exe

Breakpoint 2, main () at test.c:5
5       char str[2][3] = {0,};
```

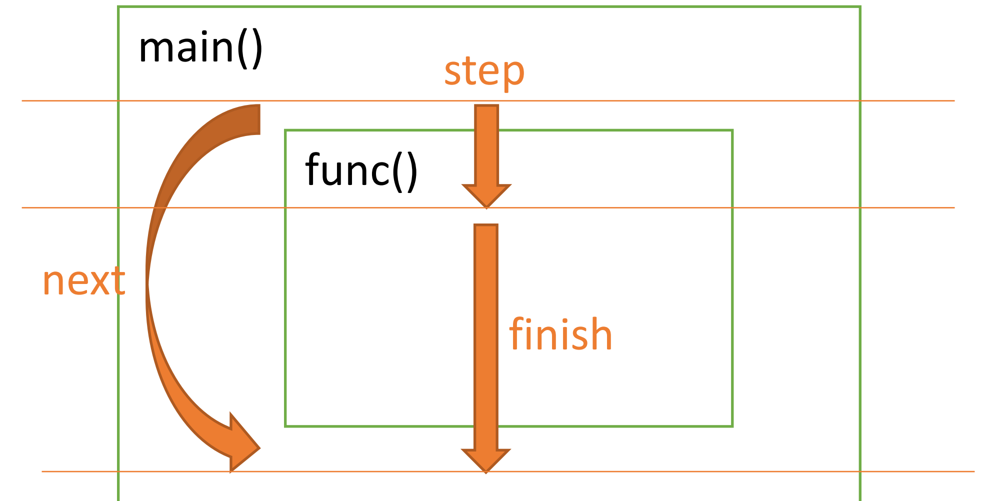
# GNU Debugger(GDB)

- run & continue
  - **run**: run the program. If there’s no breakpoint, the program will run until the end as if there is no gdb
  - **continue**: when program stopped at some breakpoint, *continue* will make the program run until the next breakpoint; otherwise, no further breakpoint, it run until the end

# GNU Debugger(GDB)

- next, step in & out
  - step over: execute one line (gdb command: next)
  - step in: execute one line & go inside the function (gdb command: step)
  - step out: skip the rest of the current function (gdb command: finish)

```
(gdb) step
step        stepi       stepping
(gdb) stepi
0x000000000040052c      5       char str[2][3] = {0,};
(gdb)
6       printf ("%p\n", str);
(gdb)
0x0000000000400536      6       printf ("%p\n", str);
(gdb)
0x0000000000400539      6       printf ("%p\n", str);
```

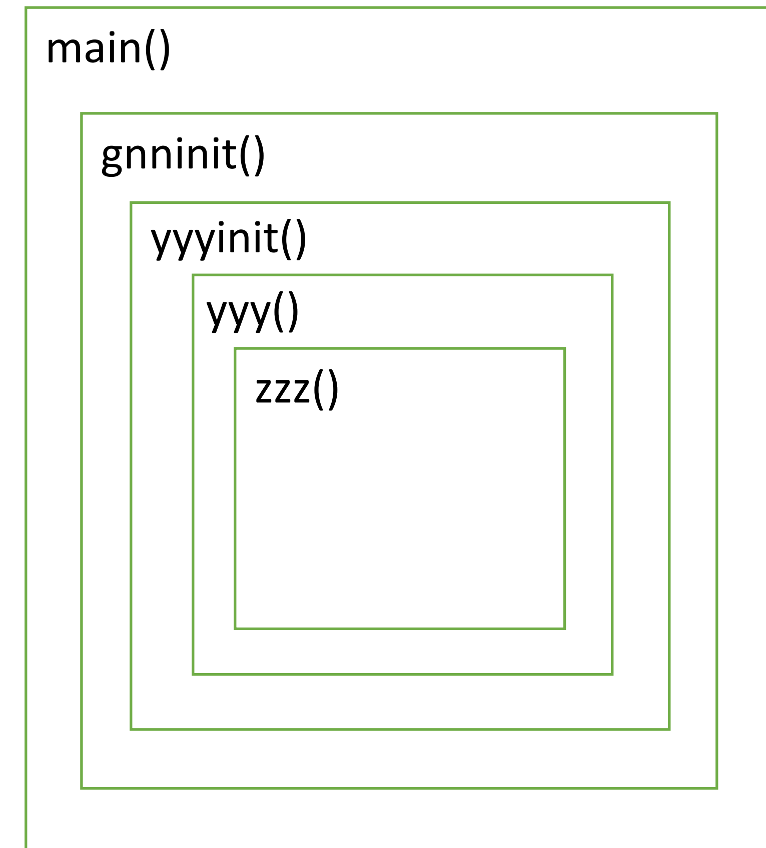



- execute one instruction: step**i**, next**i**

# GNU Debugger(GDB)

- bt (or backtrace): shows the *call stack*

```
(gdb) bt
#0 zzz () at zzz.c:96
#1 0xf7d39cba in yyy (arg=arg@entry=0x0) at
yyy.c:542
#2 0xf7d3a4f6 in yyyinit () at yyy.c:590
#3 0x0804ac0c in gnninit () at gnn.c:374
#4 main (argc=1, argv=0xffffd5e4) at gnn.c:389
```



# GNU Debugger(GDB)

- info & help
  - **info reg**
  - **info frame**

```
(gdb) info reg
rax            0x7fffffffdbe0   140737488346080
rbx            0x0      0
rcx            0x4005f0 4195824
rdx            0x7fffffffdce8   140737488346344
rsi            0x7fffffffdbe0   140737488346080
rdi            0x1      1
rbp            0x7fffffffdbf0   0x7fffffffdbf0
rsp            0x7fffffffdbe0   0x7fffffffdbe0
r8             0x7ffff7dd5e80   140737351868032
r9             0x0      0
r10            0x7fffffffd880   140737488345216
r11            0x7ffff7a302e0   140737348043488
r12            0x400430 4195376
r13            0x7fffffffdcd0   140737488346320
r14            0x0      0
r15            0x0      0
rip            0x400539 0x400539 <main+28>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

```
(gdb) info
address                 copying                 inferiors
all-registers           dcache                  line
args                    display                 locals
auto-load               extensions              macro
auxv                    files                   macros
bookmarks               float                   mem
breakpoints             frame                   os
checkpoints             frame-filter            pretty-printer
classes                 functions               probes
common                  handle                  proc
```

```
(gdb) help stepping
Specify single-stepping behavior at a tracepoint.
Argument is number of instructions to trace in single-step mode
following the tracepoint.  This command is normally followed by
one or more "collect" commands, to specify what to collect
while single-stepping.
```

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdc00:
 rip = 0x400539 in main (test.c:6); saved rip 0x7ffff7a303d5
 source language c.
 Arglist at 0x7fffffffdbf0, args:
 Locals at 0x7fffffffdbf0, Previous frame's sp is 0x7fffffffdc00
 Saved registers:
  rbp at 0x7fffffffdbf0, rip at 0x7fffffffdbf8
```

# GNU Debugger(GDB)

- Debugging assembly
  - **objdump –D <exec>:** human-readable dump of instructions of a program
  - **objdump –D exec_file > result.txt; vi result.txt**
- Additional windows(helpful)
  - In some systems, **tui enable** – **layout asm** – **tui disable**
  - or **tui reg general** – **layout asm**
  - To turn it off, C-x a(or C-x C-a, no need to lift the control key up)

```
Register group: general
rax            0x7fffffffdbe0   140737488346080        rbx            0x0      0                      rcx            0x4005f0 4195824
rdx            0x7fffffffdce8   140737488346344        rsi            0x7fffffffdbe0   140737488346080        rdi            0x1      1
rbp            0x7fffffffdbf0   0x7fffffffdbf0         rsp            0x7fffffffdbe0   0x7fffffffdbe0         r8             0x7ffff7dd5e80   140737351868032
r9             0x0      0                              r10            0x7fffffffd880   140737488345216        r11            0x7ffff7a302e0   140737348043488
r12            0x400430 4195376                        r13            0x7fffffffdcd0   140737488346320        r14            0x0      0
r15            0x0      0                              rip            0x400539 0x400539 <main+28>            eflags         0x202    [ IF ]
cs             0x33     51                             ss             0x2b     43                            ds             0x0      0
es             0x0      0                              fs             0x0      0                             gs             0x0      0
```

```
test.c
   2
   3        int main()
   4        {
B+ 5        char str[2][3] = {0,};
 > 6        printf ("%p\n", str);
   7        printf ("%p\n", &str[0]);
   8        printf ("%p\n", &str[1]);
   9        printf ("%p\n", &str[1][0]);
   10       printf ("%p\n", &str[2]);
   11       printf ("%p\n", &str[2][0]);
   12       printf ("%p\n", &str[2][1]);
   13       return 0;
   14       }
   15
```

```
child process 24680 In: main                                        Line: 6    PC: 0x400539
(gdb)
```

Use arrow keys to move around

```
> 0x400539 <main+28>       mov    $0x400680,%edi
  0x40053e <main+33>       mov    $0x0,%eax
  0x400543 <main+38>       callq  0x400400 <printf@plt>
  0x400548 <main+43>       lea    -0x10(%rbp),%rax
  0x40054c <main+47>       mov    %rax,%rsi
  0x40054f <main+50>       mov    $0x400680,%edi
  0x400554 <main+55>       mov    $0x0,%eax
  0x400559 <main+60>       callq  0x400400 <printf@plt>
  0x40055e <main+65>       lea    -0x10(%rbp),%rax
  0x400562 <main+69>       add    $0x3,%rax
  0x400566 <main+73>       mov    %rax,%rsi
  0x400569 <main+76>       mov    $0x400680,%edi
  0x40056e <main+81>       mov    $0x0,%eax
  0x400573 <main+86>       callq  0x400400 <printf@plt>
  0x400578 <main+91>       lea    -0x10(%rbp),%rax
  0x40057c <main+95>       add    $0x3,%rax
  0x400580 <main+99>       mov    %rax,%rsi
  0x400583 <main+102>      mov    $0x400680,%edi
  0x400588 <main+107>      mov    $0x0,%eax
  0x40058d <main+112>      callq  0x400400 <printf@plt>
  0x400592 <main+117>      lea    -0x10(%rbp),%rax
  0x400596 <main+121>      add    $0x6,%rax
  0x40059a <main+125>      mov    %rax,%rsi
  0x40059d <main+128>      mov    $0x400680,%edi
  0x4005a2 <main+133>      mov    $0x0,%eax
  0x4005a7 <main+138>      callq  0x400400 <printf@plt>
  0x4005ac <main+143>      lea    -0x10(%rbp),%rax
  0x4005b0 <main+147>      add    $0x6,%rax
  0x4005b4 <main+151>      mov    %rax,%rsi
child process 24680 In: main                                   Line: 6     PC: 0x400539
(gdb)
```

To turn it off, C-x a(or C-x C-a, no need to lift the control key up)

# GNU Debugger(GDB)

- breakpoints using address
  - b *0x4005b4
  - For addresses, use * in front of it
- Useful print command
  - **p (or print)** <var_name> or *<address> or $registers
  - **x/[NUM][FMT] $sp**: show stack memory; FMT can be x(hex) f(float), …

prints 10 words in hexadecimal above the stack pointer($sp)

```
(gdb) x/10x $sp
0xffeac63c: 0xf7d39cba 0xf7d3c0d8 0xf7d3c21b 0x00000001
0xffeac64c: 0xf78d133f 0xffeac6f4 0xf7a14450 0xffeac678
0xffeac65c: 0x00000000 0xf7d3790e
```

# GNU Debugger(GDB)

- For more information, search for “GDB cheatsheet”
  - https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf