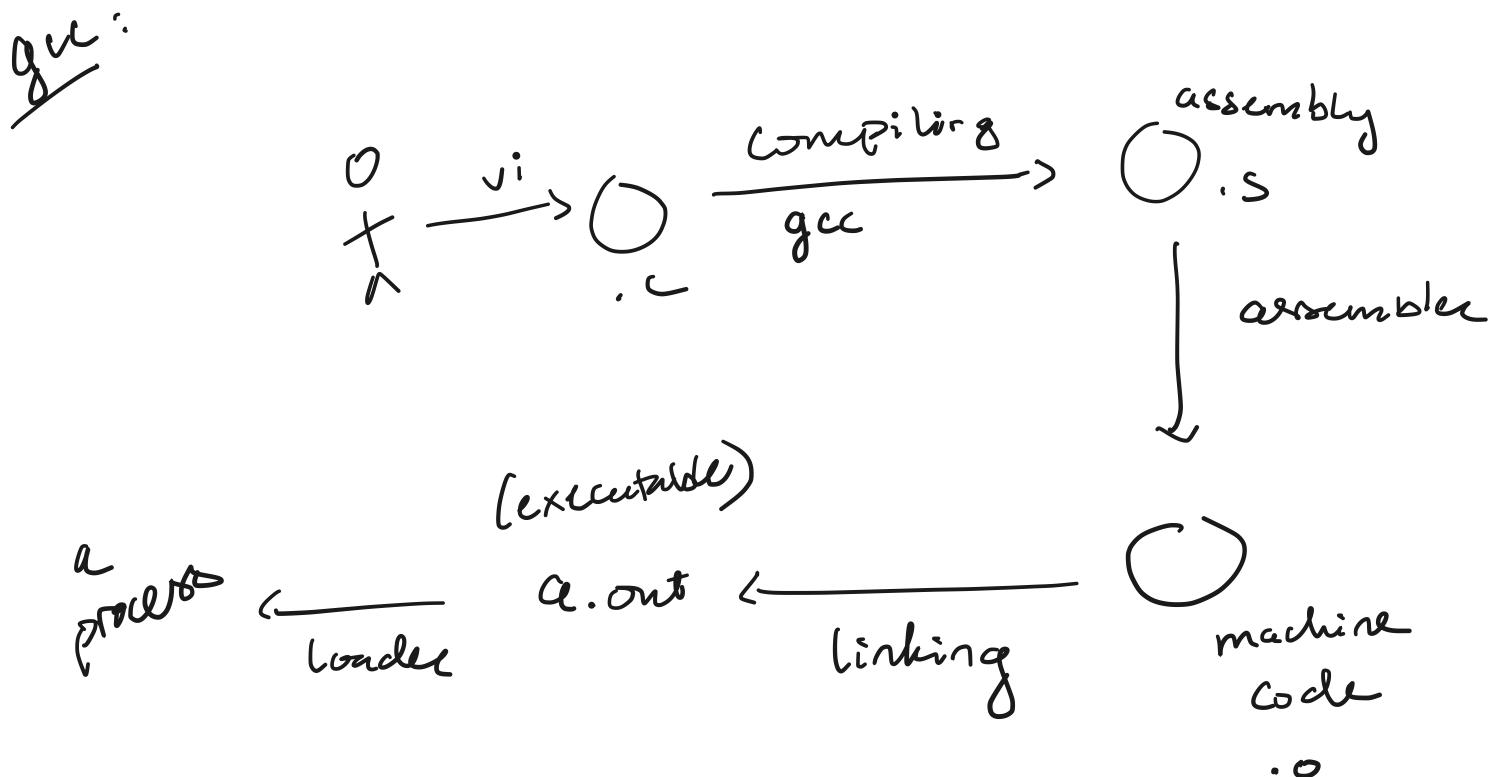


CS202 - 001: OS

Review Session 1

- ✓ 0. Record + Attendance
- ✓ 1. Introduction
- ✓ 2. Logistics
- ✓ 3. Motivation + tips
- 4. Compilation Process
 Makefile
- 5. Git Introduction
- 6. Lab Overview + C Review
- 7. Debugging



Makefile :

special file w/ shell commands
"Makefile"

Goal :

- Efficient in compiling code . Avoid mixing up commands.
- Only compile files that change

4. Git (Good)

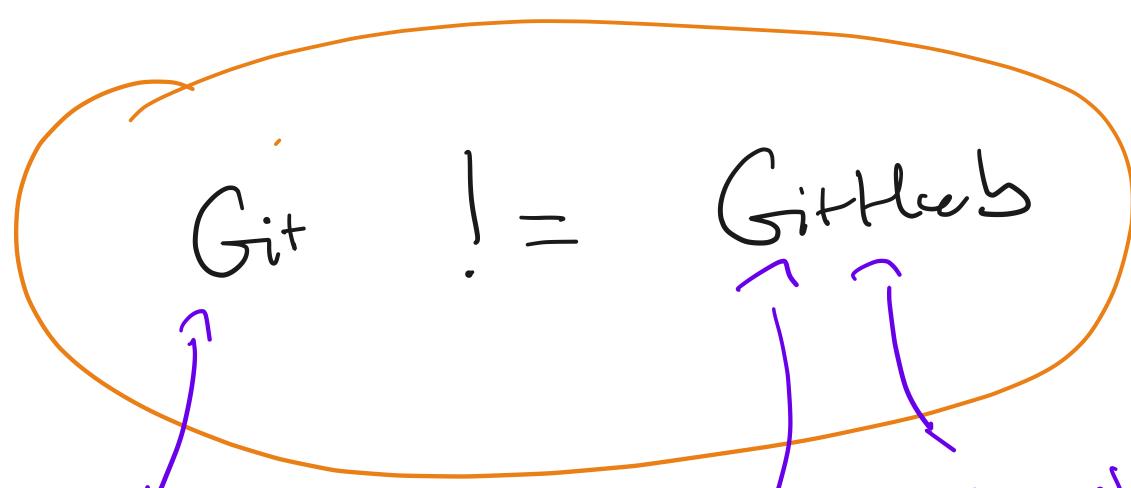
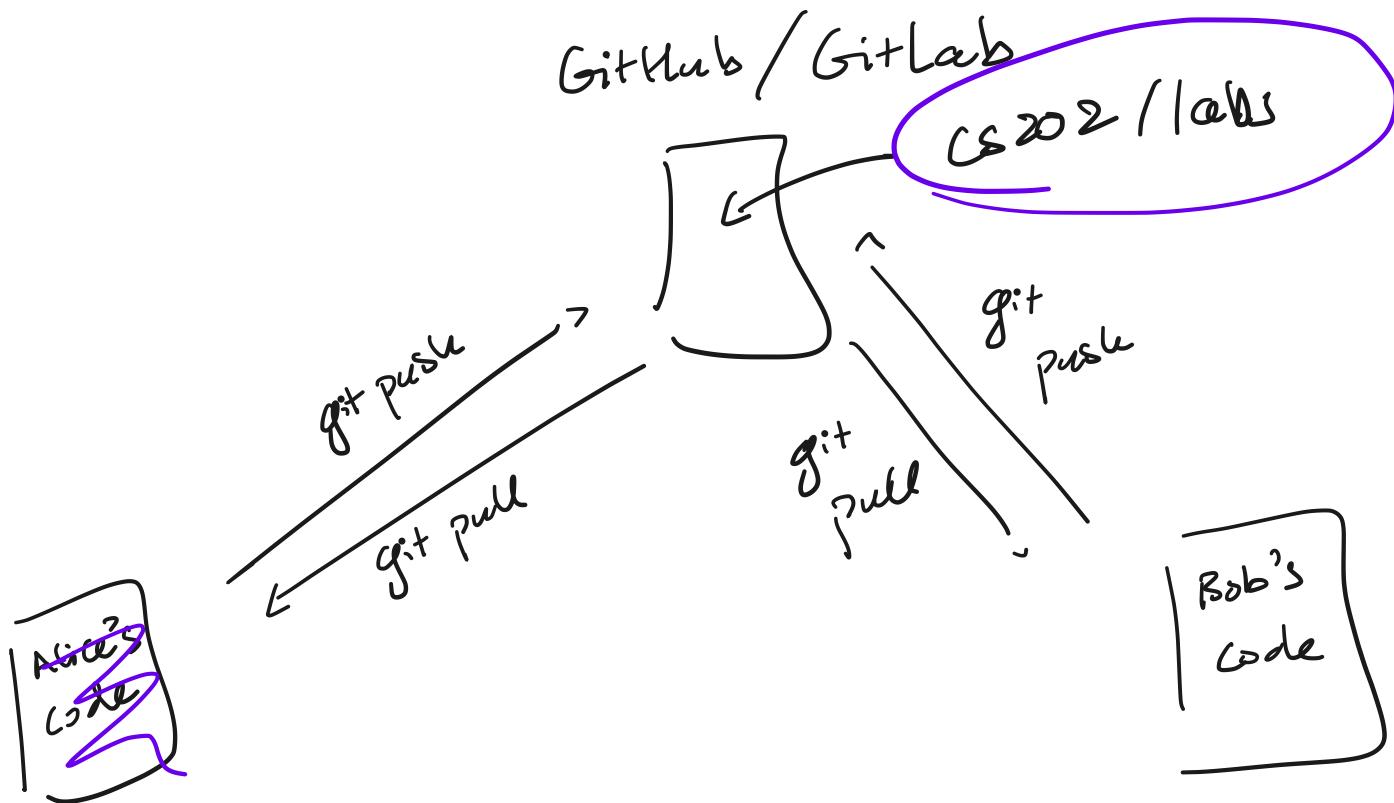
↳ built by Linus

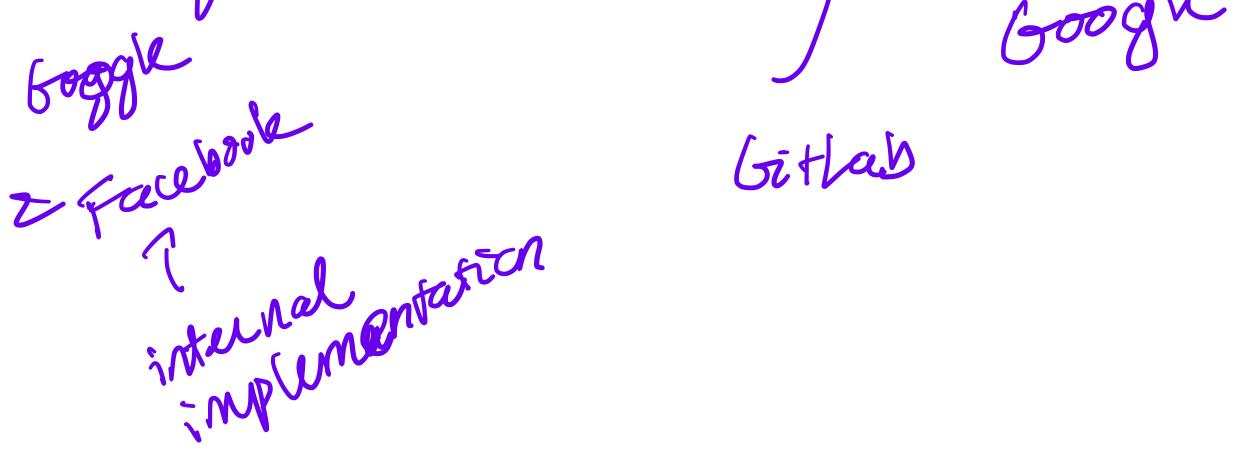
git add — — —

git commit

git push

git pull == git fetch & git merge origin/main





Be familiar w/ Unix / terminal

5. Lab Overview + C Review

• Declaration & Initialization

`int x;` → undefined value
`x = 8;`

• Pointers:

- An address within memory
- variable whose value is a memory address.

type * var_name

E.g.: `int * p;`

`*p`

Dereferencing / Access the pointers: `*p`

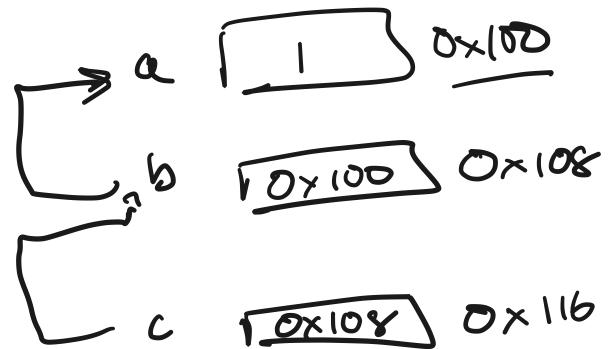
8: get me the address in memory of what follows.

int $x = \&j;$ y has the address of x
int * $y = \&x;$ ←

Mental model

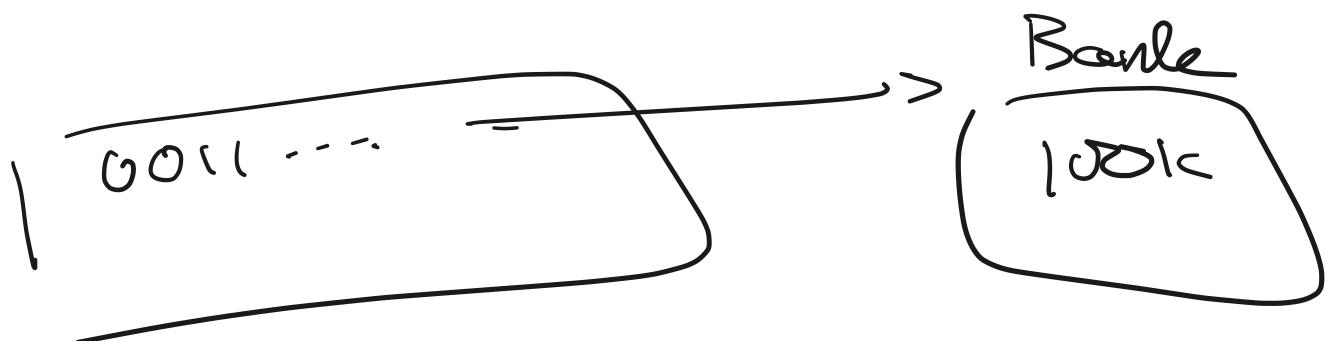
int $a = 1;$
int * $b = \&a;$
int ** $c = \&b;$

↑
double
pointer



void foo(int * p)

int * $x = *c;$
int $z = *(x);$
 ||



• Array & String

type var-name

int arr [5] = { 1, 2, 3, 4, 5 }; or 2 3 4

arr [1]. = 2 ;

int * ptr = arr

ptr

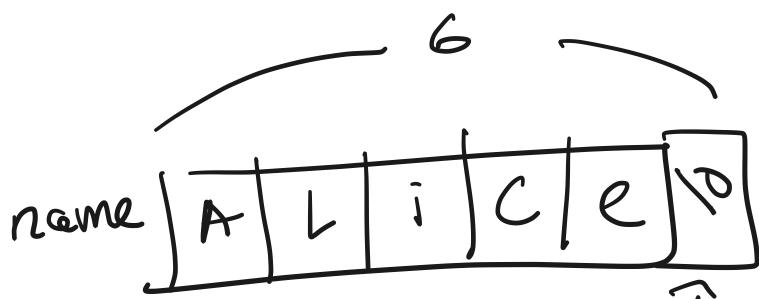
* (ptr + 1) == *(ptr + sizeof (int)*); ==
access
2nd element

char name [] = "Alice" == 6

char * name =

name [1] = 'B'

"Alice"



String

- read only region.

Constants

6. Debugging

7. Q & A.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Source: https://stackoverflow.com/questions/5580761/why-use-double-indirection-or-why-use-pointers-to-pointers
5
6 void p1(int* p) {
7     p = (int *) malloc(sizeof(int));
8     *p = 10;
9 }
10
11 void p2(int** p) {
12     *p = (int *) malloc(sizeof(int));
13     **p = 10;
14 }
15
16 int main()
17 {
18     int* p = NULL;
19     p1(p);
20     printf("%d\n", *p);
21     ...
22     p2(&p);
23     printf("%d\n", *p);
24     ...
25     free(p);
26
27     return 0;
28 }
29

```

The diagram illustrates the state of memory after the execution of the provided C code. It features two vertical columns: 'Stack' on the left and 'heap' on the right, separated by a thick red vertical line.

- Stack:** Labeled 'main' at the top. It contains a pointer variable **P** pointing to a stack-allocated integer value **0x100**. A red arrow points from the label 'main' to the **P** variable.
- heap:** Labeled 'heap' at the top. It contains a heap-allocated integer value **10** (circled). A red arrow points from the label 'heap' to the **10** value.
- Pointers:** Two other pointers, **P1** and **P2**, are shown. **P1** points to the **0x100** location in the stack. **P2** points to the **10** location in the heap.
- Annotations:**
 - A red arrow labeled 'undefined' points from the **0x100** value in the stack to the **10** value in the heap, indicating that reading the stack location results in undefined behavior.
 - A red arrow labeled '10' points from the **10** value in the heap back to the **0x100** value in the stack, suggesting a potential write operation or a pointer swap.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /*
5 Source: Example by Xiangyu Gao
6 */
7
8 void swap(int para1, int para2) {
9     int tmp = para1;
10    para1 = para2;
11    para2 = para1, tmp;
12 }
13
14 void second_swap(int* para1, int* para2) {
15     int tmp = *para1;
16     *para1 = *para2;
17     *para2 = tmp;
18 }
19
20 int main() {
21     int first = 0;
22     int second = 10;
23 }
```

first → 10
second → 0

para1 →
para2 →
tmp → 0

second - swap (&first, &second);

```
1 GCC = gcc  
2 FLAGS = -std=c99 -Werror  
3  
4 all: file1  
5  
6 file1: file1.c  
7 >>${GCC} ${FLAGS} -o file1 file1.c  
8  
9 clean:  
10 >>rm -f file1
```

make all



make file1

remove changes

evaluate to GCC: |

rename executable



gcc -std=c99 -Werror

-o
↑

file1
↑

./file1