CSE 333 Section 1

C, Pointers, and Gitlab

Logistics

Due Friday:

Exercise 1 @ 10 am

Due Monday:

Exercise 2 @ 10 am

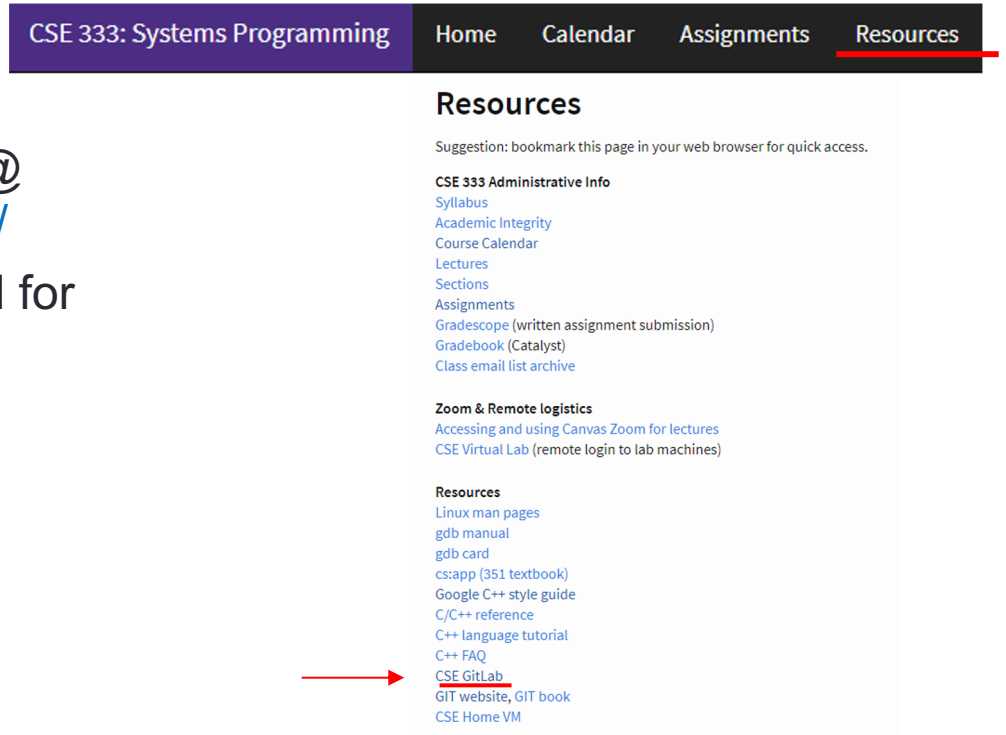
HW0 (setup Gitlab ASAP) @11:00 pm

Icebreaker!

Gitlab Demo

Accessing Gitlab

- Sign In using your **CSE NetID** @ https://gitlab.cs.washington.edu/
- You should have a repo created for you titled: cse333-20sp-<netid>



SSH Key Generation

Step 1a: Check if you have a key

- Run **cat ~/.ssh/id_rsa.pub**
- If you see a long string starting with ssh-rsa or ssh-dsa go to Step 2

Step 1b: Generate a new SSH key if necessary

- Run **ssh-keygen -t rsa -C "<netid>@cs.washington.edu"** to generate a new key
- Click enter to skip creating a password
  - git docs suggest creating a password, but it's overkill for 333 and complicates operations

SSH Key Generation

Step 2: Copy SSH key

- run **cat ~/.ssh/id_rsa.pub**
- Copy the complete key key starting with ssh- and ending with your username and host

Step 3: Add SSH key to gitlab

- Navigate to your ssh-keys page (click on your avatar in the upper-right, then “Settings,” then “SSH Keys” in the left-side menu)
- Paste into the “Key” text box and give a “Title” to identify what machine the key is for
- Click the green “Add key” button below “Title”

First Commit

1) **git clone <repo url from project page>**
   - Clones your repo
2) **touch README.md**
   - Creates an empty file called README.md
3) **git status**
   - Prints out the status of the repo: you should see 1 new file README.md
4) **git add README.md**
   - Stages a new file/updated file for commit. git status: README.md staged for commit
5) **git commit -m "First Commit"**
   - Commits all staged files with the provided comment/message.
     git status: Your branch is ahead by 1 commit.
6) **git push**
   - Publishes the changes to the central repo. You should now see these changes in the web interface (may need to refresh).
   - Might need **git push -u origin master** on first commit (only)

Git Repo Usage

Try to use the command line interface (not Gitlab's web interface)

Only push files used to build your code to the repo

- No executables, object files, etc.
- Don't always use `<git add .>` to add all your local files

Commit and push when an individual chunk of work is tested and done

- Don't push after every edit
- Don't only push once when everything is done

Git References

- **SSH Key generation:**
  https://gitlab.cs.washington.edu/help/ssh/README.md

- **Basic Git Tutorial:**
  https://courses.cs.washington.edu/courses/cse333/18su/hw/git.html

Pointer Review

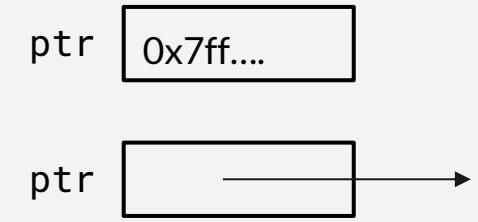
Pointers

Pointers are just another primitive data type.

The difference between an int and a pointer:
pointers hold an address

```
type *name;
```

```
int32_t *ptr;
```



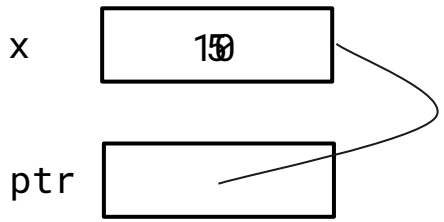
Pointer Syntax

& “Address of”

***** “Value at”

```
int32_t x;
int32_t *ptr;

ptr = &x;
x = 5;
*ptr = 10;
```

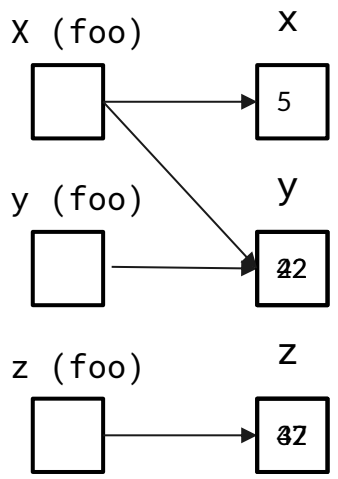


Exercise 1

Draw a memory diagram like the one above for the following code and determine what the output will be.

```
void foo(int32_t *x, int32_t *y, int32_t *z) {
  x = y;
  *x = *z;
  *z = 37;
}

int main(int argc, char *argv[]) {
  int32_t x = 5, y = 22, z = 42;
  foo(&x, &y, &z);
  printf("%d, %d, %d\n", x, y, z);
  return EXIT_SUCCESS;
}
```



So, the code will output 5, 42, 37.

C-Strings

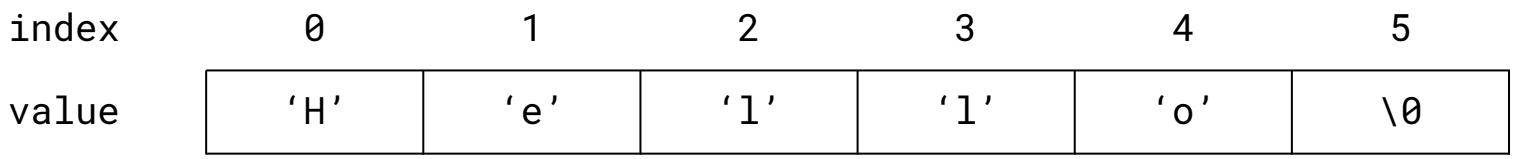
C-Strings

```
char str_name[size];
```

- A string in C is declared as an array of characters that is terminated by a null character ‘\0’.
- When declaring a string, remember to add an extra space for the null terminator.

Example

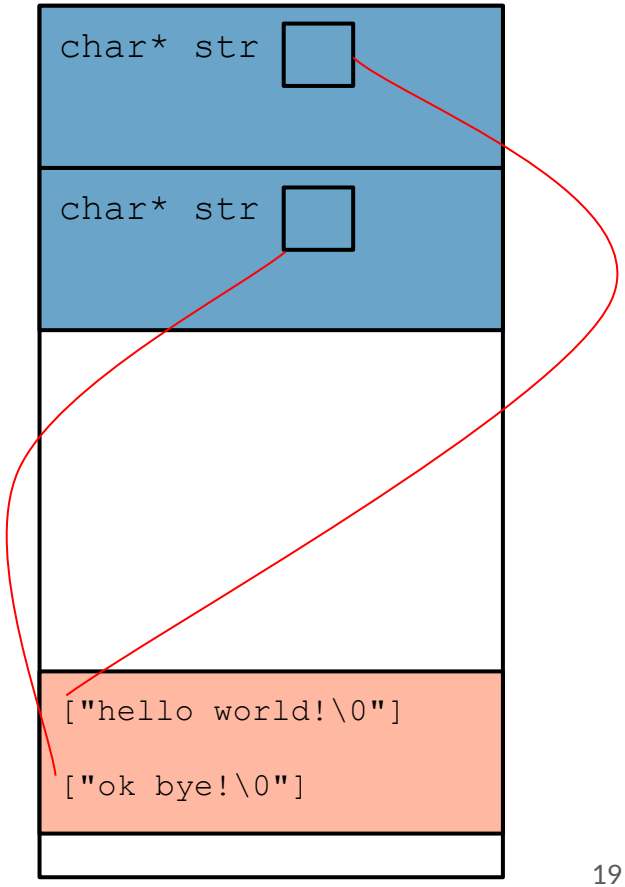
```
char str[6] = "Hello";
```



The following code has a bug. What's the problem, and how would you fix it?

```
void bar(char *str) {
  str = "ok bye!";
}

int main(int argc, char *argv[]) {
  char *str = "hello world!";
  bar(str);
  printf("%s\n", str);  // should print "ok bye!"
  return EXIT_SUCCESS;
}
```



Modifying the argument `str` in `bar` will not effect `str` in `main` because arguments in C are always passed by value.

In order to modify `str` in `main`, we need to pass a pointer to a pointer (`char **`) into `bar` and then dereference it:

```
void bar_fixed(char **str_ptr) {
  *str_ptr = "ok bye!";
}
```

Output Parameters

Output Parameters

Definition: a pointer parameter used to store output in a location specified by the caller.

Useful for returning multiple items :)

Output Parameter example

Consider the following function:

```
void getFive(int ret){
    ret = 5;
}
```

Will the user get the value '5'?

No! You need to use a pointer so that the caller can see the change

```
void getFive(int* ret){
    *ret = 5;
}
```

Exercise 2

```
char *strcpy(char *dest, char *src) {
    char *ret_value = dest;
    while (*src != '\0') {
      *dest = *src;
      src++;
      dest++;
    }
    *dest = '\0';  // don't forget the null terminator!
    return ret_value;
}
```

Why do we need an output parameter? Why can't we just return an array?

If we allocate an array inside strcpy, it will be allocated on the stack. Thus, we have no control over this memory after strcpy returns, which means we can't safely use the array whose address we've returned.

We could've also returned a pointer to a `malloc`'d block of memory. How might this be better or worse than using an output parameter?

Pros of Malloc

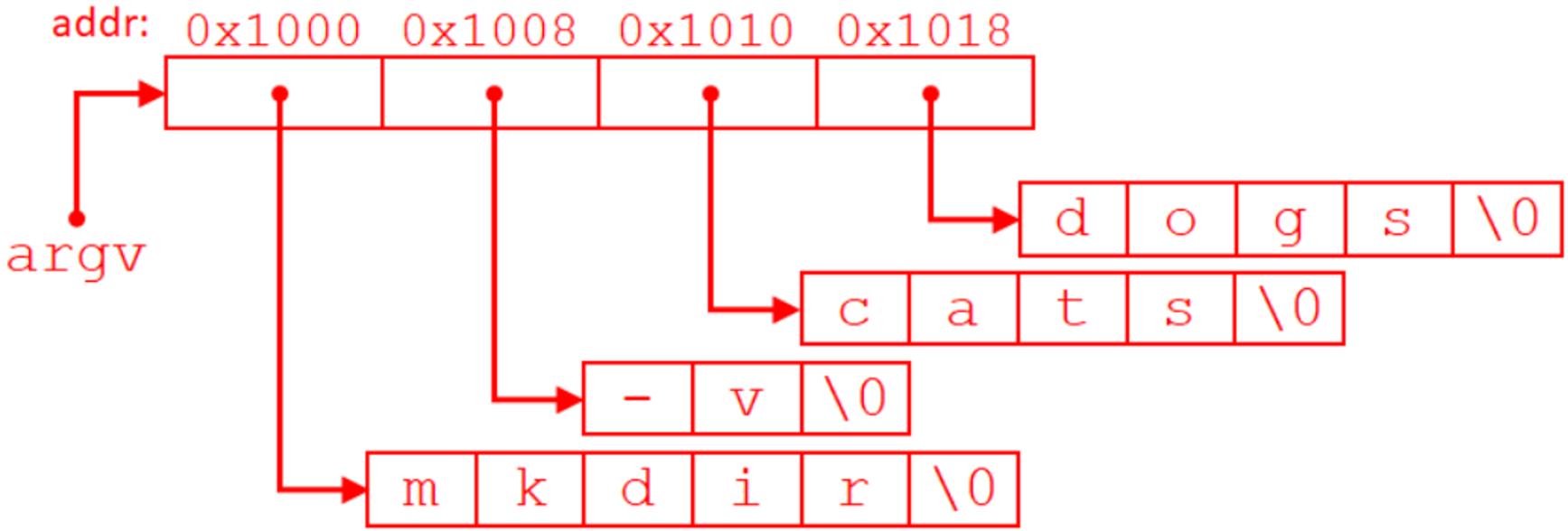
- We don't have to worry about an output parameter not having enough storage space for the copy.

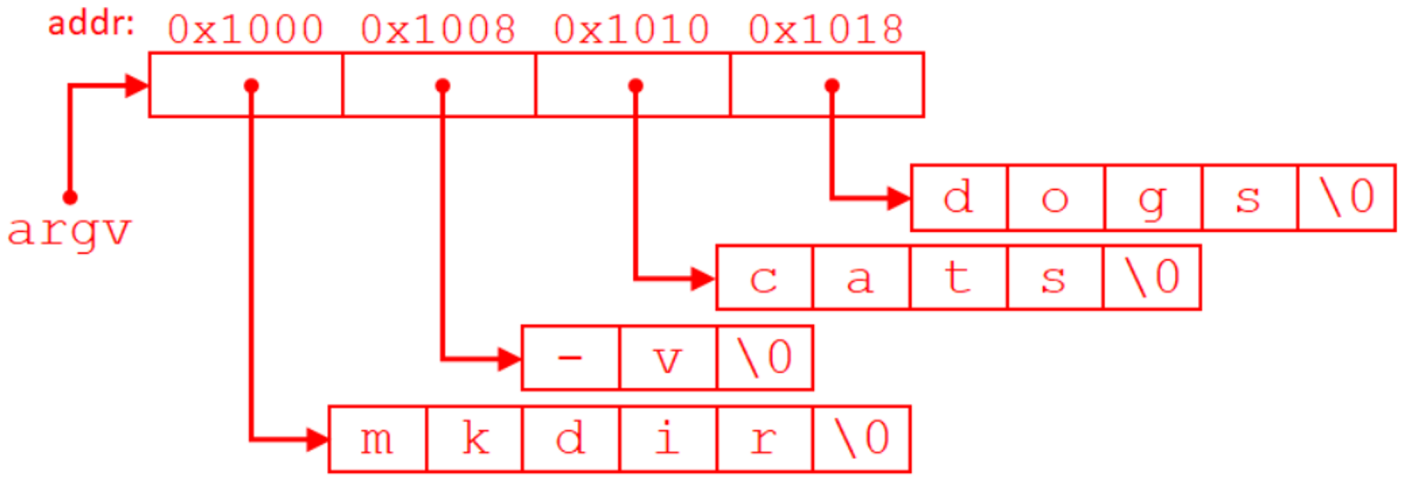
Pros of Output Params

- We don't have to handle the case where malloc fails.

Exercise 3 (Bonus)

Given the following command: "`mkdir -v cats dogs`" and `argv = 0x1000`, draw a box-and-arrow memory diagram of `argv` and its contents for when `mkdir` executes.





1) argv[0]

2) argv + 1

3) *(argv[1] + 1)

4) argv[0] + 1

5) argv[0][3]