CSE 333
Section 1

C, Pointers, and Gitlab
Logistics

● Exercise 1:
  ○ Due **Monday @ 10 PM (10/2)**

● Homework 0:
  ○ Due **Tuesday @ 10 PM (10/3)**
  ○ Meant to acquaint you to your repo and project logistics
  ○ Must be done individually (some future HWs allow partners)
TA Intro: Leanna Nguyen (Mi)

- Junior in CS, minoring in Vietnamese Language & Culture!
- 3rd time TA, 2nd time for 333
- Interests: cybersecurity, education
- Hobbies: running, listening to music (spotify ftw), crocheting
TA Intro: Humza Lala

- Senior in CS
- 4th-time TA, 3rd-time 333 TA
- Hobbies: Hiking, kayaking, languages, photography, and genealogy
Icebreaker!

In groups of ~3, please share:

● Name and Year
● Favorite hobby, sport, and class taken so far.
● Find one thing in common
● Choose 1 person to share out your names and your group’s common ~thing~ with the section
Pointer Review
Pointers

- **Data type** that stores the address of (the lowest byte of) a datum
  - **Tip!** Can draw an arrow in memory diagrams from pointer to pointed to data, particularly if actual value (stored address) is unknown

- Common uses:
  - Reference to data allocated elsewhere (*e.g.*, `malloc`, literals, files)
  - Iterators (*e.g.*, data structure traversal)
  - Data abstraction (*e.g.*, head of linked list, function pointers)
Pointers: Syntax and Semantics

- Declared as `type* name;` or `type *name;`
  - Doesn’t matter, just be consistent
- “Address-of” operator `&` gets a variable’s address
- “Dereference” operator `*` refers to the pointed-to datum

- Example code:

```c
int* ar = (int*) malloc(3*sizeof(int));  // reference
int* p = &ar[1];  // iterator
*p = 3;
```

- Example diagram:
Output Parameters
Output Parameters

- Recall: the `return` statement in a function passes a single value back through the `%rax` register.
- An **output parameter** is a C idiom that emulates “returning values” through parameters:
  - An output parameter is a pointer (i.e., the address of a location in memory).
  - The function with this parameter must dereference it to change the value stored at that location.
  - The new value is “returned” by persisting after the function returns.
- Output parameters are the only way in C to achieve returning *multiple values*.
Exercise 1
Exercise 1

- Which parameters are output parameters?
  quotient and remainder

- What should go in the division blanks?
  &quot and &rem

- What should go in the printf blanks?
  quot and rem

```c
void division(int numerator, int denominator, int* quotient, int* remainder) {
  *quotient = numerator / denominator;
  *remainder = numerator % denominator;
}

int main(int argc, char* argv[]) {
  int quot, rem;
  division(22, 5, _____, _____);
  printf("%d rem %d\n", _____, _____);
  return EXIT_SUCCESS;
}
```
Exercise 1

- Which parameters are output parameters?
- What should go in the `division` blanks?
- What should go in the `printf` blanks?

```c
void division(int numerator, int denominator, int* quotient, int* remainder) {
    *quotient = numerator / denominator;
    *remainder = numerator % denominator;
}

int main(int argc, char* argv[]) {
    int quot, rem;
    division(22, 5, ______, ______);
    printf("%d rem %d\n", ______, ______);
    return EXIT_SUCCESS;
}
```
Exercise 1

- Draw out a memory diagram of the beginning of this call to `division`.

```c
void division(int numerator, int denominator, int* quotient, int* remainder) {
    *quotient = numerator / denominator;
    *remainder = numerator % denominator;
}

int main(int argc, char* argv[]) {
    int quot, rem;
    division(22, 5, ____ , ____);
    printf("%d rem %d\n", ____ , ____);
    return EXIT_SUCCESS;
}
```
Exercise 1

- Draw out a memory diagram of the beginning of this call to `division`.

```c
void division(int numerator,  
              int denominator,  
              int* quotient,  
              int* remainder) {  
    *quotient = numerator / denominator;  
    *remainder = numerator % denominator;  
}

int main(int argc, char* argv[]) {  
    int quot, rem;  
    division(22, 5, _____, _____);  
    printf("%d rem %d\n", _____, _____);  
    return EXIT_SUCCESS;  
}
```
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 allocating space for a string, remember to add an extra element for the null character
Initialization Examples

- **Code:**

```c
// list initialization
cchar str1[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

// string literal initialization
cchar str2[6] = "Hello";
```

- **Memory:**

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>'H'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

- **Notes:**
  - Both initialize the array *in the declaration scope* (e.g., on the stack if a local var), though the latter can be thought of copying the contents from the string literal into the array.
  - The size 6 is *optional*, as it can be inferred from the initialization.
Common String Literal Error

- **Code:**
  ```c
  // pointer instead of an array
  char* str3 = "Hello";
  ```

- **Memory:**
  ```
  str3  0x402037
  index 0 1 2 3 4 5
  value 'H' 'e' 'l' 'l' 'o' '\0'
  ```

- **Notes:**
  - By default, using a string literal will allocate and initialize the character array in *read-only* memory (Literals)
Common String Literal Error

- **Code:**
  ```c
  // pointer instead of an array
  char* str3 = "Hello";
  ```

- **Memory:**
  ```c
  str3: 0x402037
  ```

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'H'</td>
</tr>
<tr>
<td>1</td>
<td>'e'</td>
</tr>
<tr>
<td>2</td>
<td>'l'</td>
</tr>
<tr>
<td>3</td>
<td>'l'</td>
</tr>
<tr>
<td>4</td>
<td>'o'</td>
</tr>
<tr>
<td>5</td>
<td>\0</td>
</tr>
</tbody>
</table>

- **Notes:**
  - By default, using a string literal will allocate and initialize the character array in *read-only* memory (Literals)
  - What would happen if we executed `str3[0] = 'J';`? Segfault!
Exercise 2
The following code has a bug. What's the problem, and how would you fix it?

```c
void bar(char ch) {
    ch = '3';
}

int main(int argc, char* argv[]) {
    char fav_class[] = "CSE331";
    bar(fav_class[5]);
    bar(fav_class[5]);
    printf("%s\n", fav_class);  // should print "CSE333"
    return EXIT_SUCCESS;
}
```
The following code has a bug. What’s the problem, and how would you fix it?

```c
void bar(char ch) {
    ch = '3';
}

int main(int argc, char* argv[]) {
    char fav_class[] = "CSE331";
    bar(fav_class[5]);
    printf("%s\n", fav_class); // should print "CSE333"
    return EXIT_SUCCESS;
}
```

Modifying the argument `ch` in `bar` will not affect `fav_class` in `main()` because arguments in C are always passed by value.

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

```c
void bar_fixed(char* ch) {
    *ch = '3';
}
```
The following code has a bug. What's the problem, and how would you fix it?

```c
void bar_fixed(char* ch) {
    *ch = '3';
}

int main(int argc, char* argv[]) {
    char fav_class[] = "CSE331";
    bar(&fav_class[5]);
    printf("%s\n", fav_class);  // should print "CSE333"
    return EXIT_SUCCESS;
}
```

Modifying the argument `ch` in `bar` will not affect `fav_class` in `main()` because arguments in C are always passed by value.

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

```c
void bar_fixed(char* ch) {
    *ch = '3';
}
```
Setting Up git
gcc 11

- CSE Lab machines and the attu cluster have been updated to use gcc 11.
- As such we’ll be using gcc 11 this quarter
- To verify that you’re using gcc 11 run:
  - `gcc -v` or
  - `gcc --version`
- If you use the CSE Linux home VM, you need to use the newer version even if you have an older one installed (i.e., use 22au or later).
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
We have a page that details how to (1) set up Gitlab and (2) use git to manage your repo (solo or with a partner):


We asked you to attempt your Gitlab setup ahead of time:

- If you didn’t, please do so now on your CSE Linux environment setup
- If you did and ran into issues, we’ll walk around to help you now
Accessing Gitlab

- Sign-in using your CSE NetID @ https://gitlab.cs.washington.edu/
- There should be a repo created for you titled: cse333-23au-<netid>
- Please let us know if you don’t have one!
SSH Key Generation

**Step 1a)** See if you have an existing SSH 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 you don’t have an existing SSH key, you’ll need to create one
- Run `ssh-keygen -t rsa -C "<netid>@cs.washington.edu"` to generate a new key
- Hit enter to skip creating a password
  - git docs suggest creating a password, but it’s overkill for CSE333
Adding your SSH key to Gitlab

Step 2) Copy your SSH key

- Run `cat ~/.ssh/id_rsa.pub`
- Copy the complete key starting with `ssh-` and ending with your username and host
  (i.e. `<netid>@cs.washington.edu`)

Step 3) Add your SSH key to Gitlab
Adding your SSH key to Gitlab

Step 3) Add your SSH key to Gitlab

- Navigate to your ssh-keys page (click on your avatar in the upper-right, then “Preferences,” 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”
Setting up git

- The git command looks for a file named `.gitconfig` in your home directory. Some commands like `commit` and `push` expect certain options to be set and will produce verbose messages if not.
- If you have not already configured git, enter the following commands (once) in a terminal window to set these values:

```bash
git config --global user.name "<your name>"

git config --global user.email <your netid>@cs.washington.edu

git config --global push.default simple
```
First Commit

1. `git clone <repo url from project page>`
   a. Clones your repo

2. `touch README.md`
   a. Creates an empty file called `README.md`

3. `git status`
   a. Prints out the status of the repo: you should see 1 new file `README.md`

4. `git add README.md` (or: `git stage README.md`)
   a. Stages a new file/updated file for commit.
      
      `git status`: README.md staged for commit

5. `git commit -m "First Commit"`
   a. Commits all staged files with the provided comment/message.
      
      `git status`: Your branch is ahead by 1 commit.

6. `git push`
   a. Publishes the changes to the central repo.
      
      You should now see these changes in the web interface (may need to refresh).

7. Might need `git push -u origin master` on first commit (only), but would be unusual for this to happen.
Function Pointers
Function Pointers

- Pointers can store addresses of functions
  - Functions are just instructions in read-only memory, their names are pointers to this memory.
- Used when performing operations for a function to use
  - Like a comparator for a sorter to use in Java
  - Reduces redundancy

```c
int one() { return 1; }
int two() { return 2; }
int three() { return 3; }

int get(int (*func_name)()) {  
    return func_name();
}

int main(int argc, char* argv[]) {  
    int res1 = get(one);
    int res2 = get(two);
    int res3 = get(three);
    printf("%d, %d, %d\n", res1, res2, res3);
    return EXIT_SUCCESS;
}
```