# CSE 333 Section 1

C, Pointers, and Gitlab

C isn't that hard:

`void (*(*f[])())()` defines `f` as an array of unspecified size, of pointers to functions that return `void`.

W UNIVERSITY of WASHINGTON

# Logistics

- Exercise 0:
  - Due **Friday (tomorrow!) @ 10 AM (01/05) – no late exercises accepted**
- Homework 0:
  - Due **Monday @ 11:00 PM (01/08)**
  - Meant to acquaint you to your repo and project logistics
  - Must be done individually

# Icebreaker!

Please turn to the people next to you and share:

- Name, pronouns, year
- What are you excited for this summer? Any fun travel plans?
- What are you excited to learn in CSE 333?

# Pointer Review

# Pointers

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

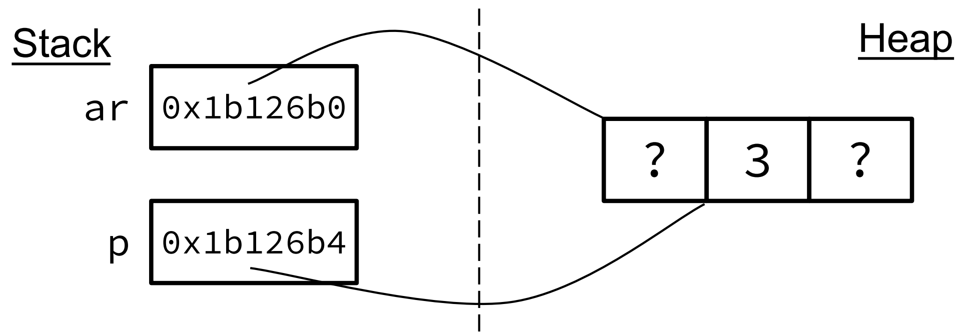
- Common uses:
  - Reference to data allocated elsewhere (*e.g.*, `malloc`, literals, files)
  - Iterators (*e.g.*, data structure traversal)
  - Data abstraction (*e.g.*, head of linked list, function pointers)

# Pointer 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:

```
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?

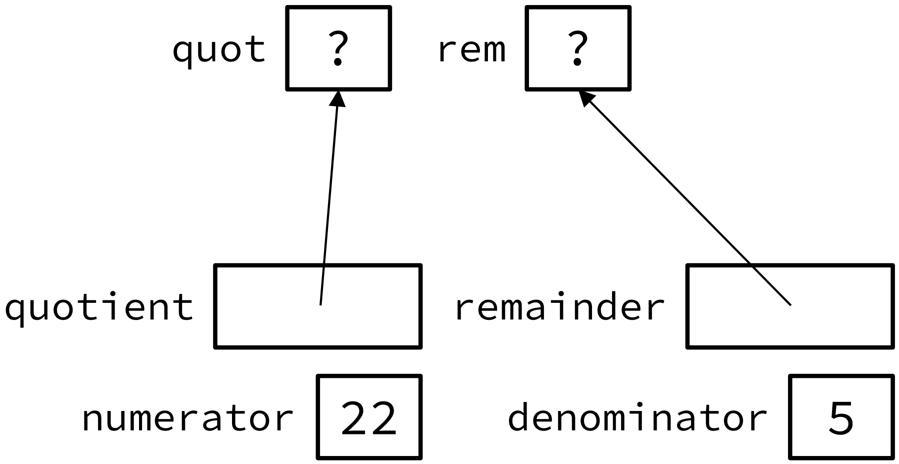
  quot and rem

```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

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




```
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:

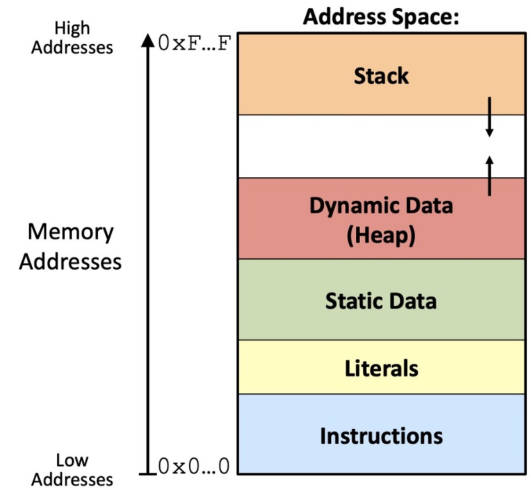
```
// list initialization
char str1[6] = {'H','e','l','l','o','\0'};
// string literal initialization
char str2[6] = "Hello";
```

- Memory:

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| value | 'H' | 'e' | 'l' | 'l' | 'o' | '\0' |

- 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 as copying the contents from the string literal into the array
  - The size 6 is *optional*, as it can be inferred from the initialization

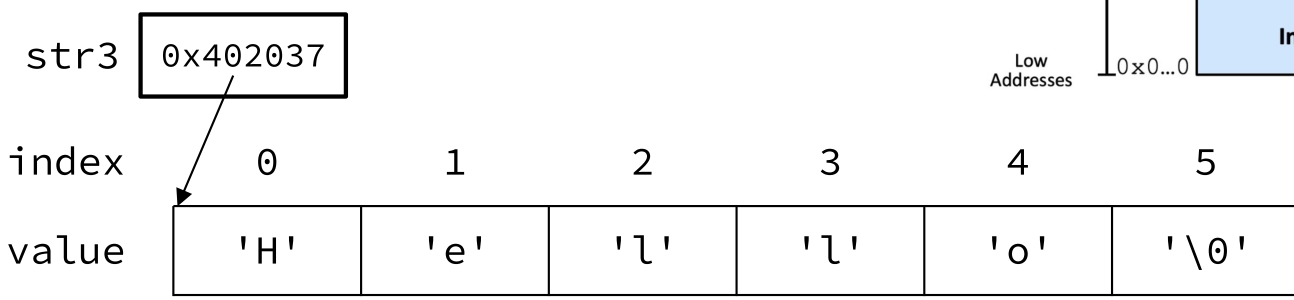
# Common String Literal Error




- Code:

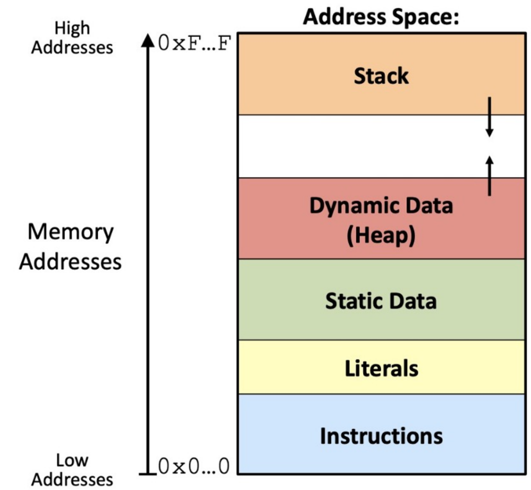
```c
// pointer instead of an array
char* str3 = "Hello";
```

- Memory:




- 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:

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)
  - 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?

```
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;
}
```

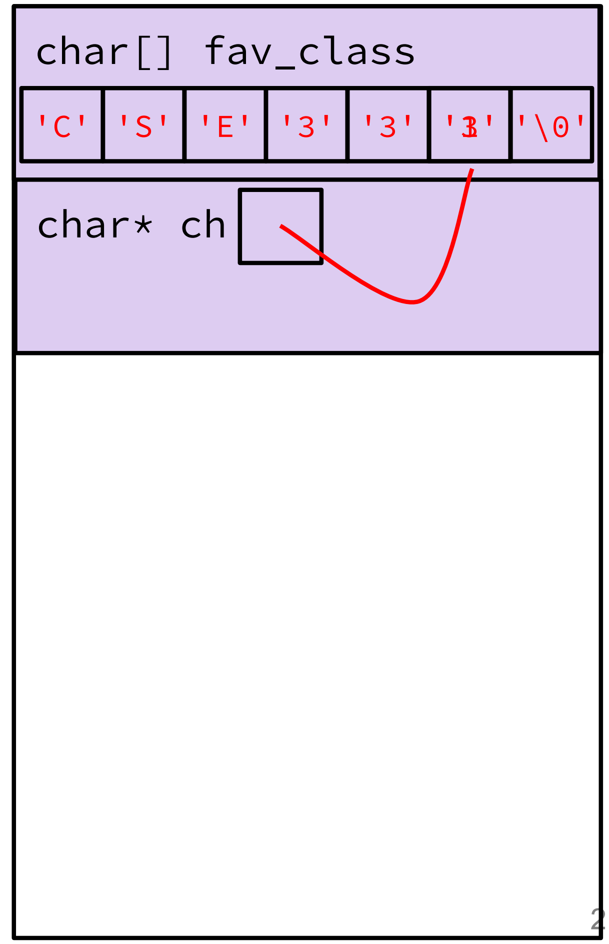
The following code has a bug. What's the problem, and how would you fix it?

```
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;
}
```

main stack frame

bar_fixed stack frame




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:

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

# 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

```
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;
}
```

# Setting Up `git`

# gcc 11

- CSE Lab machines and the attu cluster 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 should use the newer version even if you have an older one installed (*i.e.*, use 24wi).

# 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

# Using VS Code

- Can install an extension that will allow you to directly edit files on a virtual machine (attu!)
- Will also be helpful to install the C/C++ extension for syntax highlighting
- To set up, visit https://courses.cs.washington.edu/courses/cse333/24wi/resources/VSCode.pdf

# git/Gitlab Reference

We have a page that details how to (1) set up Gitlab and (2) use git to manage your repo:

- https://courses.cs.washington.edu/courses/cse333/24wi/resources/git_tutorial.html

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