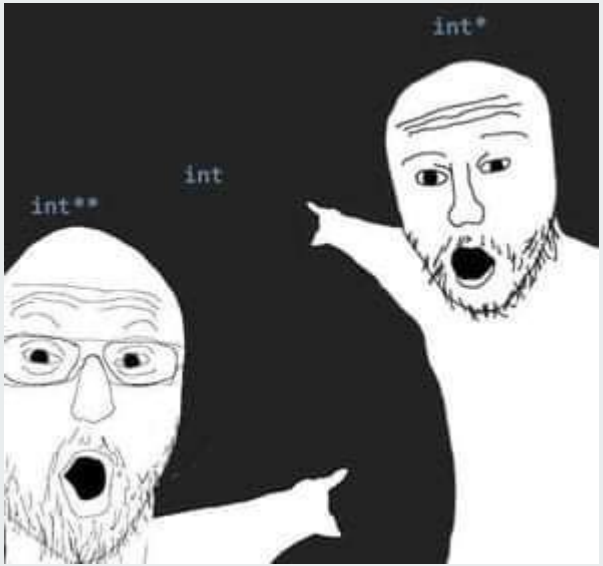
CSE 333 Section 1

C, Pointers, and Gitlab



Logistics

Due Friday (1/7):

Exercise 1 @ 11 am

Pre-Quarter Survey @ 11:59 pm

Due Monday (1/10):

HW0 (setup Gitlab ASAP) @11:59 pm

Due Wednesday (1/12):

Exercise 2 @ 11 am (released on Friday)

Icebreaker!

TODO: For TA's to add something here :) – Refer to Section Notes

Pointer Review

Pointers

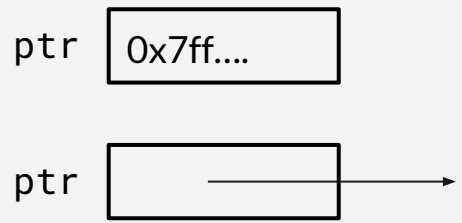
Pointers are just another primitive data type.

An integer can hold an index into an array.

If memory is a giant array of bytes, then a pointer just holds an index into that array.

```
type* name;
```

```
int32_t* ptr;
```



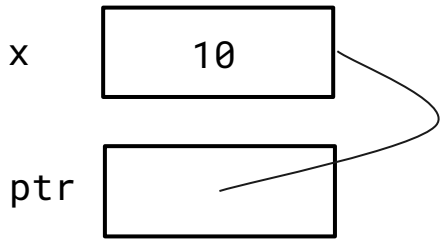
Pointer Syntax

& “Address of”

***** “Value at”

```
int32_t x;
int32_t* ptr;

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

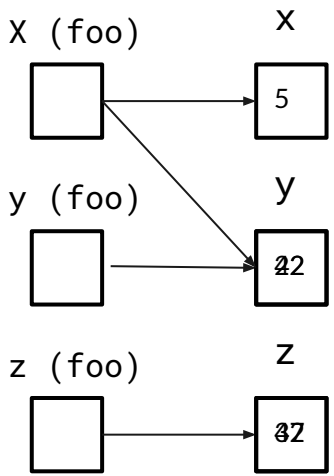


Exercise 1a

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 allocating space for a string, remember to add an extra element for the null character.

Initialization Examples

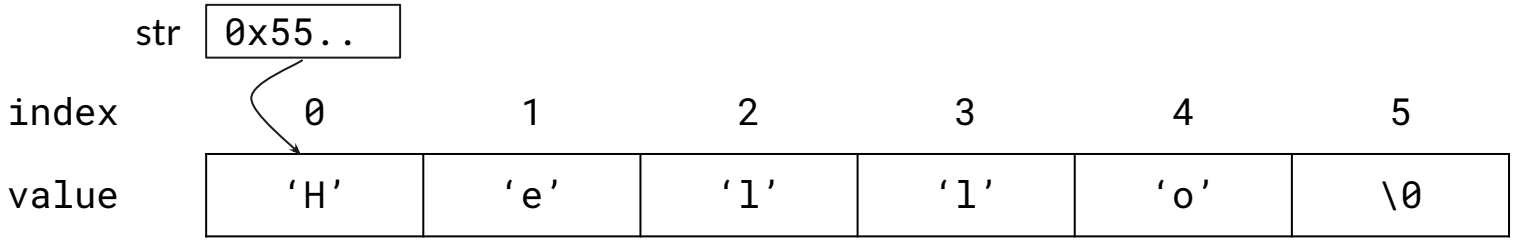
```
char str[6] = {'H','e','l','l','o','\0'};  // list initialization
char str[6] = "Hello";                     // string literal initialization
```

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

- 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.
    - The size 6 is optional, as it can be inferred from the initialization.

String Literal Example

```
char* str = "Hello";
```



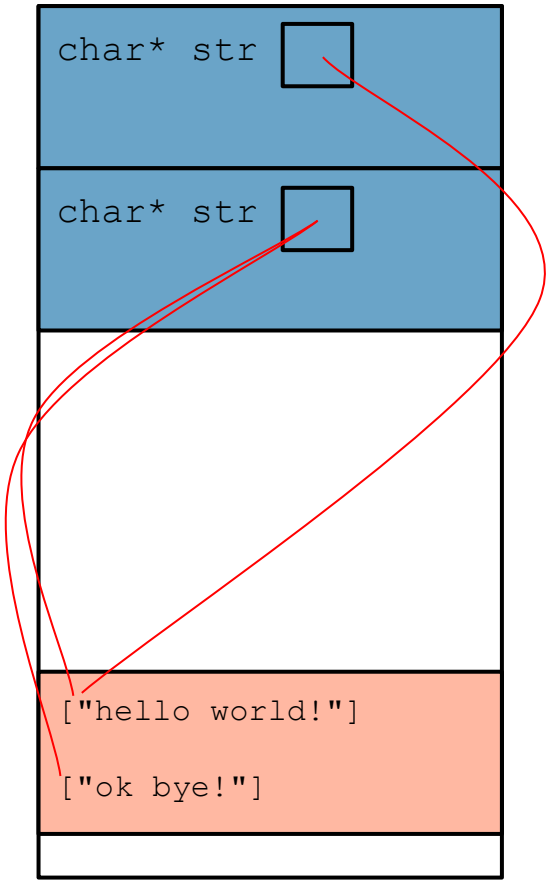
- By default, using a string literal will allocate and initialize the character array in *read-only* memory and the expression will return the *address of the array*, which can be stored in a pointer.

Exercise 1 b

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!";
}
```

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

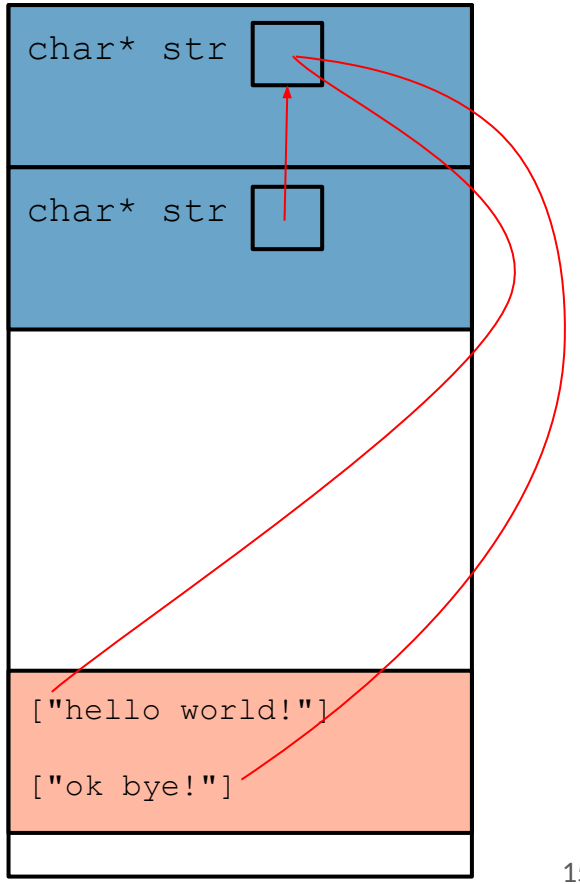
```
void bar(char** str) {
  *str = "ok bye!";
}

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;
}
```

How is the caller able to see the changes in `dest` if C is pass-by-value?

The caller can see the copied over string in `dest` since we are dereferencing `dest`. Note that modifications to dest that do not dereference will not be seen by the caller(such as `dest++`). Also note that if you used array syntax, then `dest[i]` is equivalent to `*(dest+i)`.

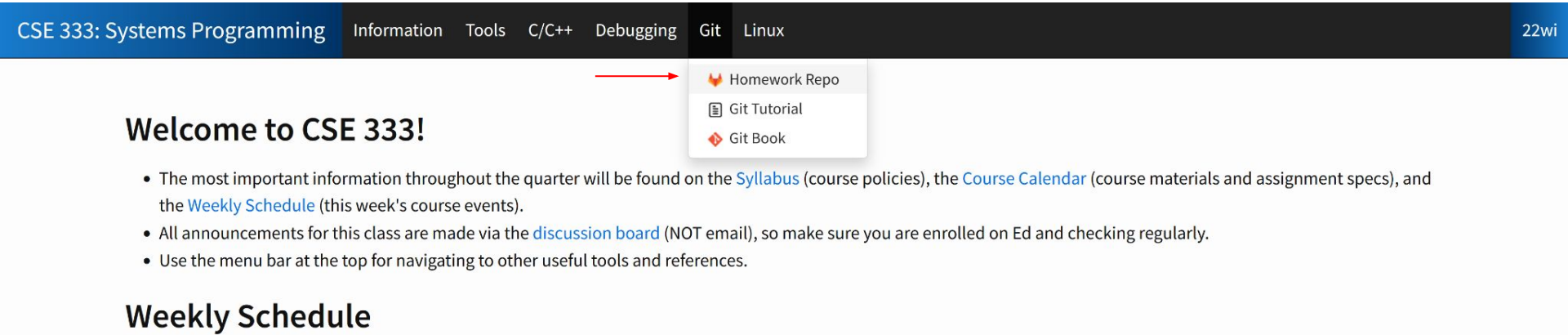
Why do we need an output parameter? Why can't we just return an array we create in strcpy?

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.

Gitlab Demo

Accessing Gitlab

- Sign in using your **UW/CSE NetID** @ https://gitlab.cs.washington.edu/
- You should have a repo created for you titled: cse333-22wi-<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/index.md#generate-an-ssh-key-pair
- **Git Setup:**
  https://courses.cs.washington.edu/courses/cse333/22wi/resources/git_tutorial.html#git-setup
- **Basic Git Workflow:**
  https://courses.cs.washington.edu/courses/cse333/22wi/resources/git_tutorial.html#git-workflow
- **Partner Tips:**

  https://courses.cs.washington.edu/courses/cse333/22wi/resources/partners.html

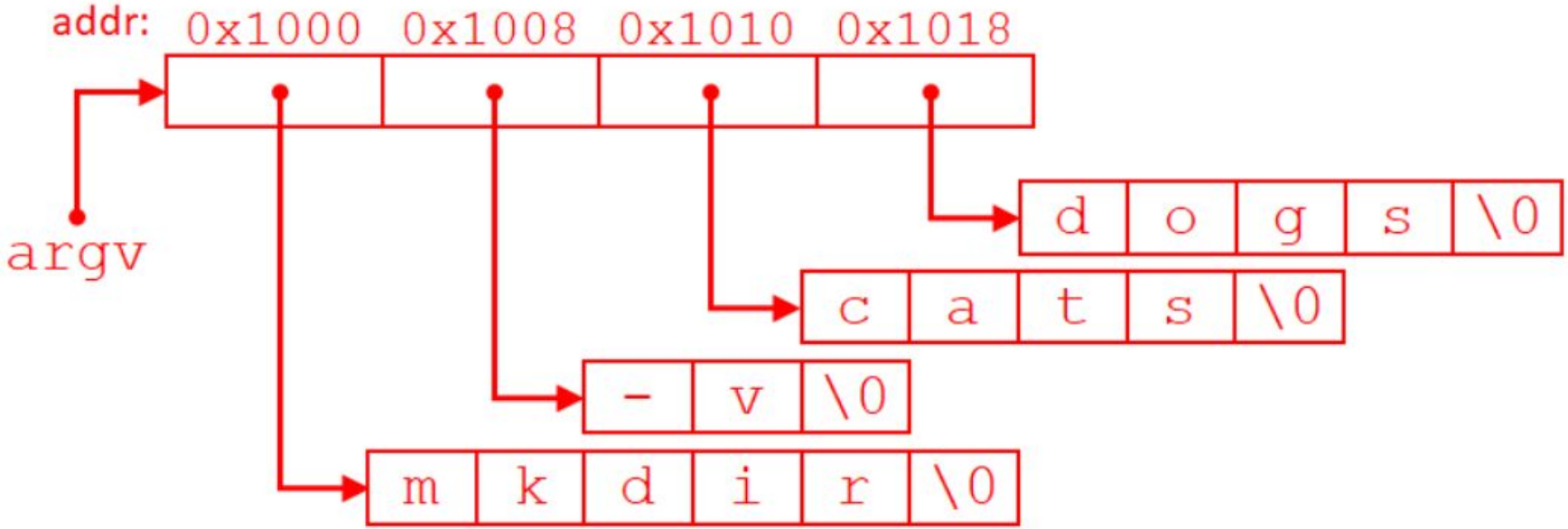
Exercise 3

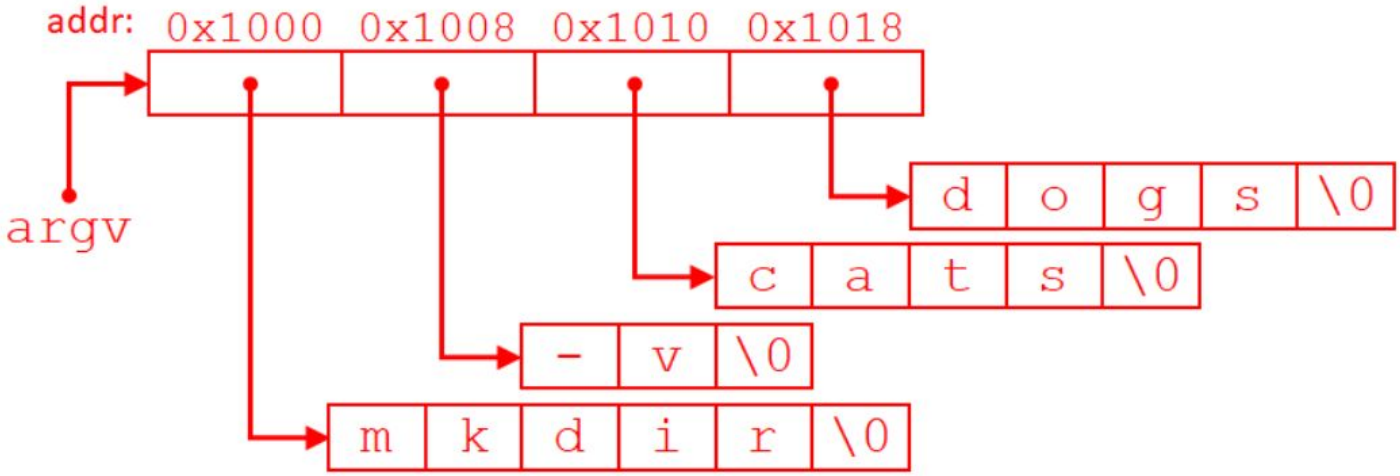
```
void product_and_sum(int* input, int length, int* product, int* sum) {
    int temp_sum = 0;
    int temp_product = 1;
    for (int i = 0; i < length; i++) {
      temp_sum += input[i];
      temp_product *= input[i];
    }
    *sum = temp_sum;
    *product = temp_product;

}
```

Exercise 4 (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]`