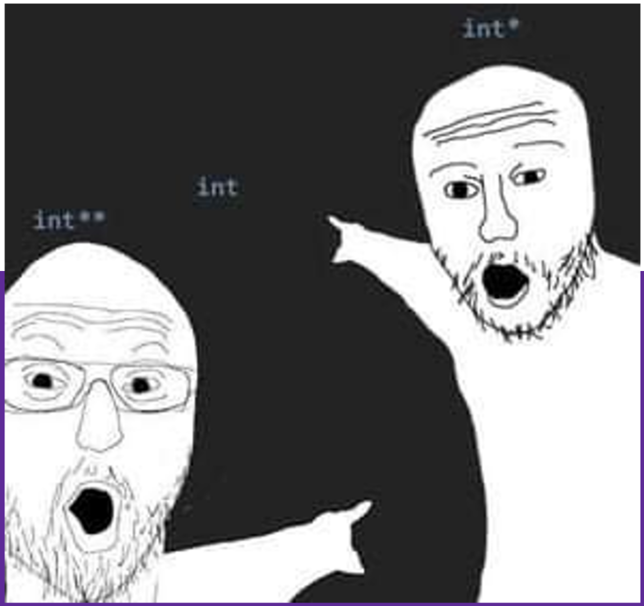
CSE 333 Section 1

C, Pointers, and Gitlab



UNIVERSITY *of* WASHINGTON

Logistics

- Exercise 1:
  - Due **Friday @ 10:00am (4/1)** – April Fools! Not the Exercise though…
- Exercise 2:
  - Due **Monday @ 10:00am (4/4)**
- Homework 0:
  - Due **Monday @ 11:00pm (4/4)**
  - Meant more for acquainting you to your repo

Icebreaker!

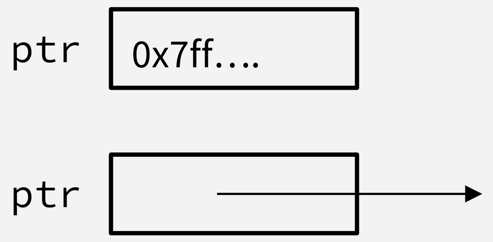
Pointer Review

Pointer Background

- Primitive data type
- Meant to store an address of a value/type (like keeping track of a location in memory)
- Often denoted with an arrow in memory diagrams

```
type* name;
```

```
int32_t* ptr;
```



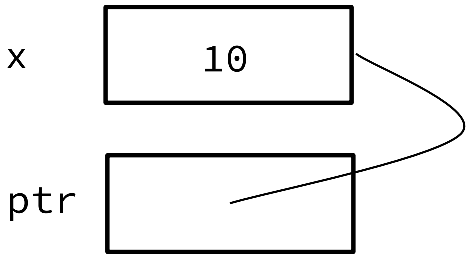
Pointer Syntax and Semantics

- How to get a variable's address (location in memory)?
  - Using the & operator
  - Getting the "address of"

- How to get the associated value of an address?
  - Using the * operator
  - Dereferencing memory

```
int32_t x;
int32_t* ptr;

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

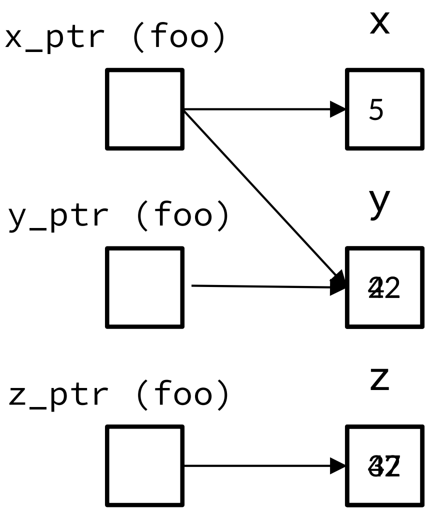


Exercise 1a

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

```
void foo(int32_t* x_ptr, int32_t* y_ptr, int32_t* z_ptr) {
  x_ptr = y_ptr;
  *x_ptr = *z_ptr;
  *z_ptr = 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.

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;
}
```

Output Parameters

Output Parameters

- Idea: Not necessarily returning values through the **return** statement (`%rax` register)
  - Rather it is changing a location in memory to be another value
  - Manipulating the stack

- Output Parameters is an C idiom in order to emulate “returning values” through parameters
  - Call the function with a parameter that takes in a pointer, or an “address of” a variable
  - This will give a location in memory to change inside of the called function
  - The function will dereference that location and change it to give you a “returned” value

- This is particularly helpful for returning **multiple values**

Output Parameter Example

- Which of the following act as returning a value back to main?

  `quotient` and `remainder`

- What gets printed?

  `4, 2`

```c
void division(int32_t num, int32_t den,
                           int32_t* quotient,
                           int32_t* remainder) {
  *quotient = num / den;
  *remainder = num % den;
}


int main(int argc, char* argv[]) {
  int32_t num = 22, den = 5, quot, rem;
  division(num, den, &quot, &rem);
  printf("%d, %d\n", quot, rem);
  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

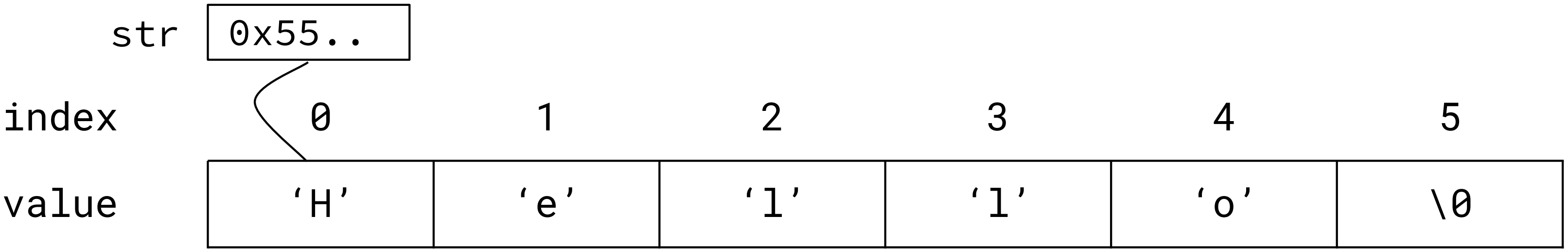
```
char str[6] = {'H','e','l','l','o','\0'};  // list initialization
char str[6] = "Hello";          // string literal initialization
```

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";
```



str 0x55..

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

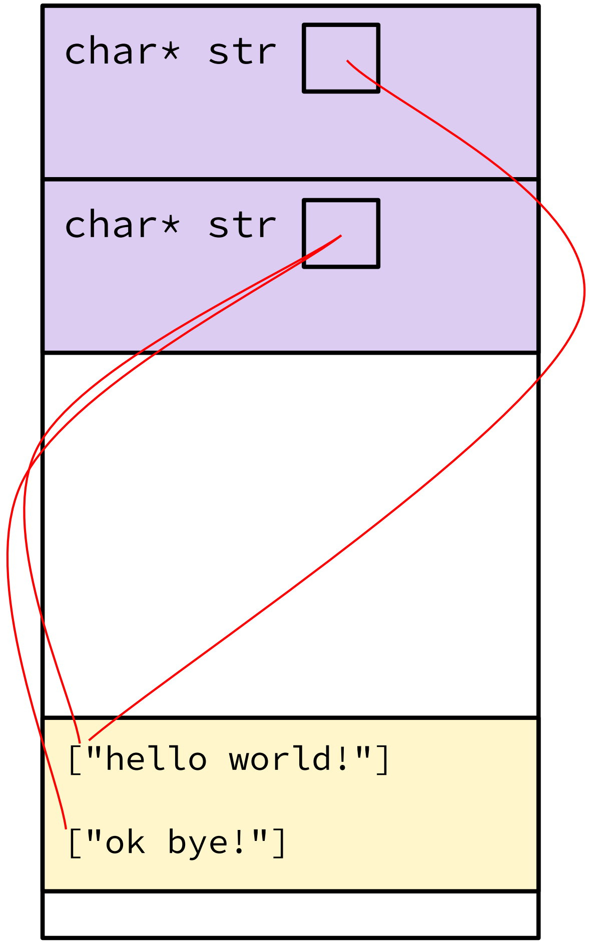
- By default, using a string literal will allocate and initialize the character array in *read-only* memory and the expression will return the *address of the array*, which can be stored in a pointer.

Exercise 1b

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) {
  *str = “ok bye!”;
}
```

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

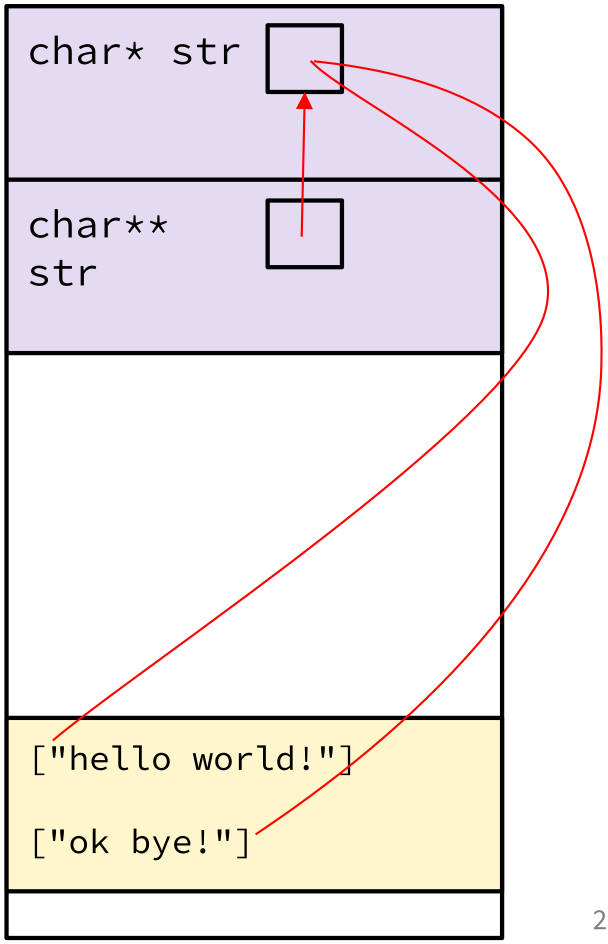
```
void bar_fixed(char** str) {
  *str = "ok bye!";
}

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) {
  *str = “ok bye!”;
}
```



main stack frame

bar stack frame

static data

Gitlab Demo

Git Reference

We have a page detailing the process of setting up git!

https://courses.cs.washington.edu/courses/cse333/22sp/resources/git_tutorial.html

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