

When poll is active, respond at [pollev.com/cse333](https://pollev.com/cse333)

# W Favourite editor for coding?

Vim

Emacs

VS Code

Atom or Sublime

Intellij

MS Paint

Eclipse

Notepad

`echo "my code" > main.c`

MS Word or Google Docs

# Memory and Arrays

## CSE 333 Spring 2021

**Instructors:** Travis McGaha, Justin Hsia

**Teaching Assistants:**

Atharva Deodhar

Callum Walker

Cosmo Wang

Dylan Hartono

Elizabeth Haker

Kyrie Dowling

Leo Liao

Markus Schiffer

Neha Nagvekar

Nonthakit Chaiwong

Ramya Challa

# Administrivia (1)

- ❖ Exercise 1 due Friday morning, 11:00 am
- ❖ If you don't have a gradescope account, make a private ed post with your name, student id# and uw email address (xyzz@uw.edu) so we can get you set up
- ❖ Pre-quarter survey & Mini Bio out
  - Survey to collect information about how to best handle things this quarter.
  - Due Friday April 2<sup>nd</sup> at 11:59 pm
  - Is for credit, ~1 point of an exercise

# Administrivia (2)

- ❖ Homework 0 out tonight
  - Logistics and infrastructure for projects
    - Gitlab email sent later today when repos created – no action needed
  - Please make a private ed post if you don't have a repo and/or the hw0 files on Thursday
  
- ❖ Reference system for grading is CSE labs/attu/current VM
  - For both exercises and homework (project) code
  - It's your job to be sure your solution(s) work on them
    - Just because it works on ReallyCoolLinuxDistribution<sup>®</sup> doesn't mean it for sure works on other Linux systems, including ours.

# Administrivia (3)

## ❖ Lecture Groups

- Will be used throughout the quarter for discussions and polls in class. Please fill these out soon.
- More information in the “333 Lectures & Zoom” ed post
  - <https://edstem.org/us/courses/4899/discussion/336253>

# Lecture Outline

- ❖ **Function Definitions vs. Declarations**
- ❖ C's Memory Model (CSE 351 refresher)
- ❖ Pointers (CSE 351 refresher)
- ❖ Arrays



# Function Declaration Style

- ❖ Teaches the compiler arguments and return types; function definitions can then be in a logical order

★ Function comment usually by the *prototype*

sum\_declared.c

(1) Declare functions first

(2) Main function

(3) Define functions later

```
// sum of integers from 1 to max
int32_t sumTo(int32_t); // func prototype

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return EXIT_SUCCESS;
}

int32_t sumTo(int32_t max) {
    int32_t i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

# Function Declaration vs. Definition

- ❖ C/C++ make a careful distinction between these two
- ❖ **Definition:** the thing itself
  - *e.g.* code for function, variable definition that creates storage
  - Must be **exactly one** definition of each thing (no duplicates)
- ❖ **Declaration:** description of a thing
  - *e.g.* function prototype, external variable declaration
    - Often in header files and incorporated via `#include`
    - Should also `#include` declaration in the file with the actual definition to check for consistency
  - Needs to appear in all files that use that thing
    - Should appear before first use

*More on header files & #include  
in lecture 5*

# Multi-file C Programs

C source file 1  
(sumstore.c)

```
void sumstore(int x, int y, int* dest) { <- defined  
    *dest = x + y;  
}
```

C source file 2  
(sumnum.c)

```
#include <stdio.h>  
  
void sumstore(int x, int y, int* dest); <- declared  
  
int main(int argc, char** argv) {  
    int z, x = 351, y = 333;  
    sumstore(x, y, &z); <- used  
    printf("%d + %d = %d\n", x, y, z);  
    return 0;  
}
```

Note: not good  
style. More on  
multiple files in  
lecture 5

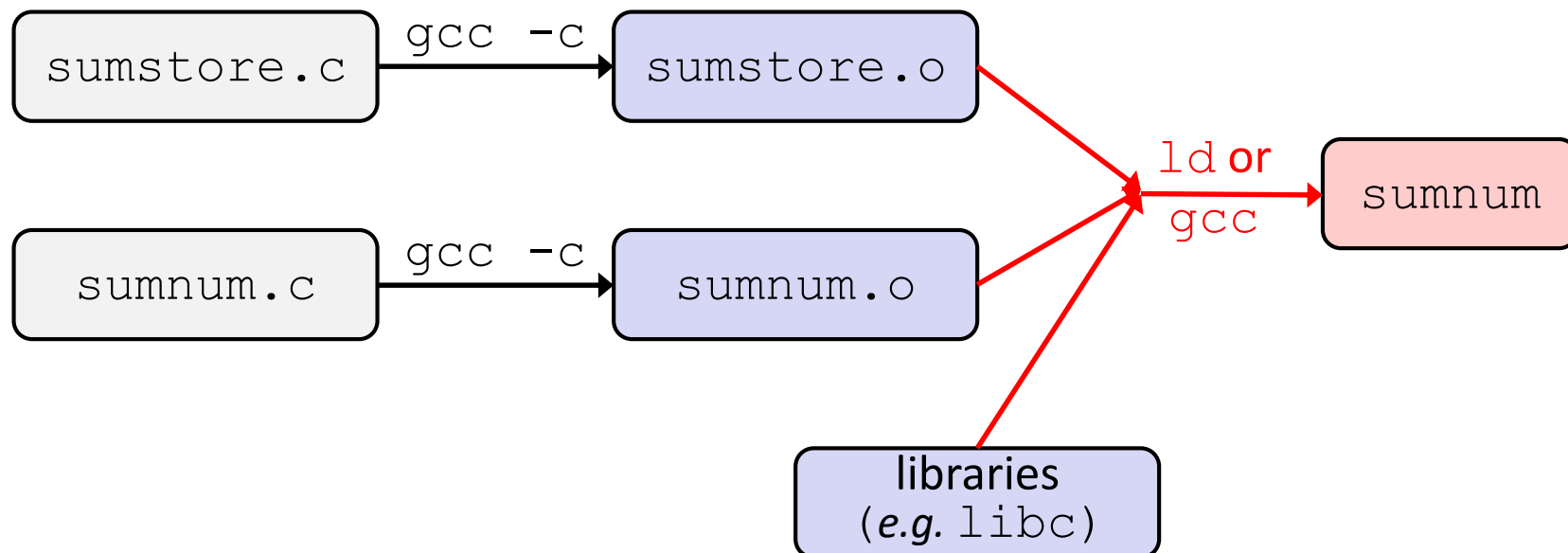
*Both files used in compilation*

Compile together:

```
$ gcc -o sumnum sumnum.c sumstore.c
```

# Compiling Multi-file Programs

- ❖ The **linker** combines multiple object files plus statically-linked libraries to produce an executable
  - Includes many standard libraries (*e.g.* `libc`, `crt1`)
    - A *library* is just a pre-assembled collection of `.o` files



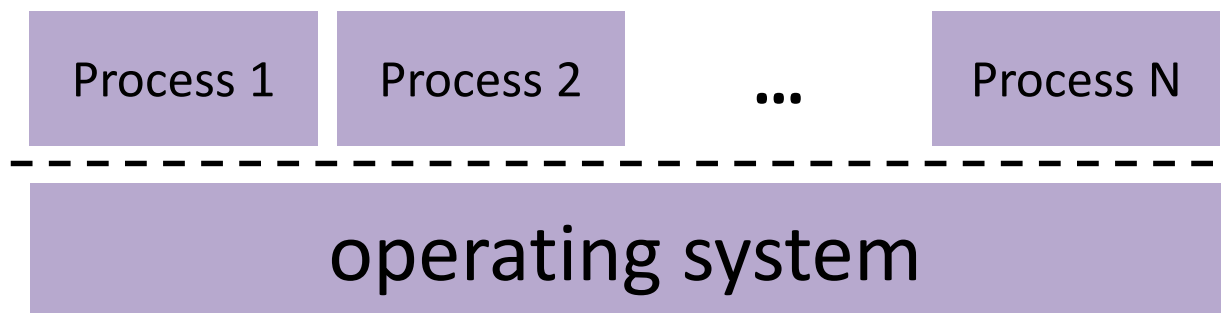
# Lecture Outline

- ❖ Function Definitions vs. Declarations
- ❖ **C's Memory Model** (CSE 351 refresher)
- ❖ Pointers (CSE 351 refresher)
- ❖ Arrays

# OS and Processes

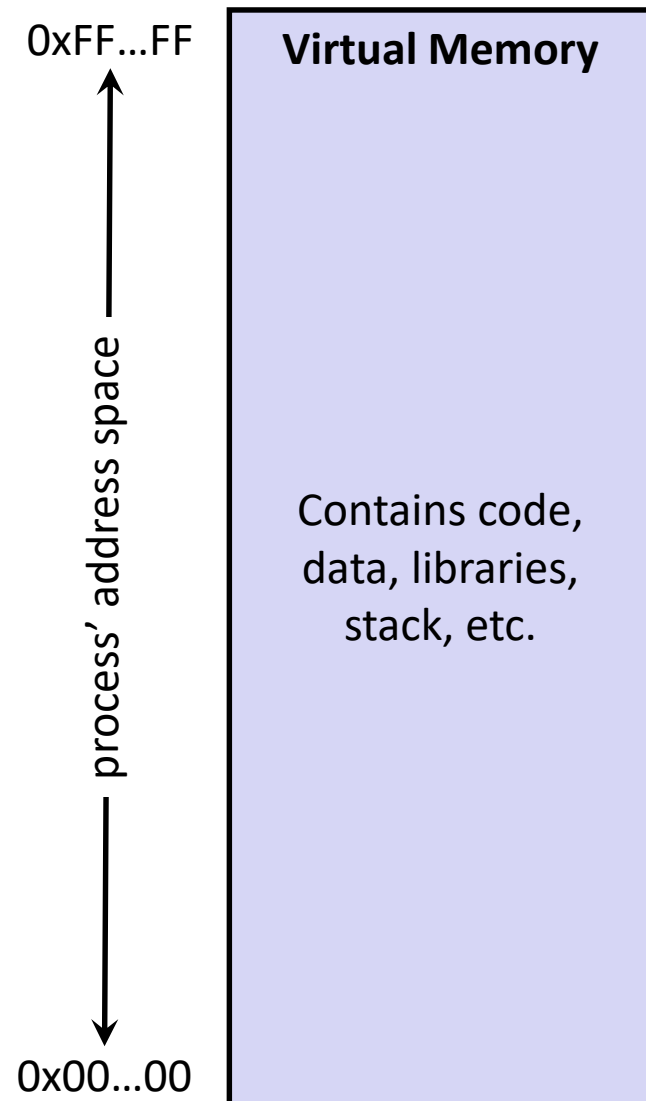
- ❖ The OS lets you run multiple applications at once
  - An application runs within an OS “process”
  - The OS timeslices each CPU between runnable processes
    - This happens *very quickly*: ~100 times per second

Context switching



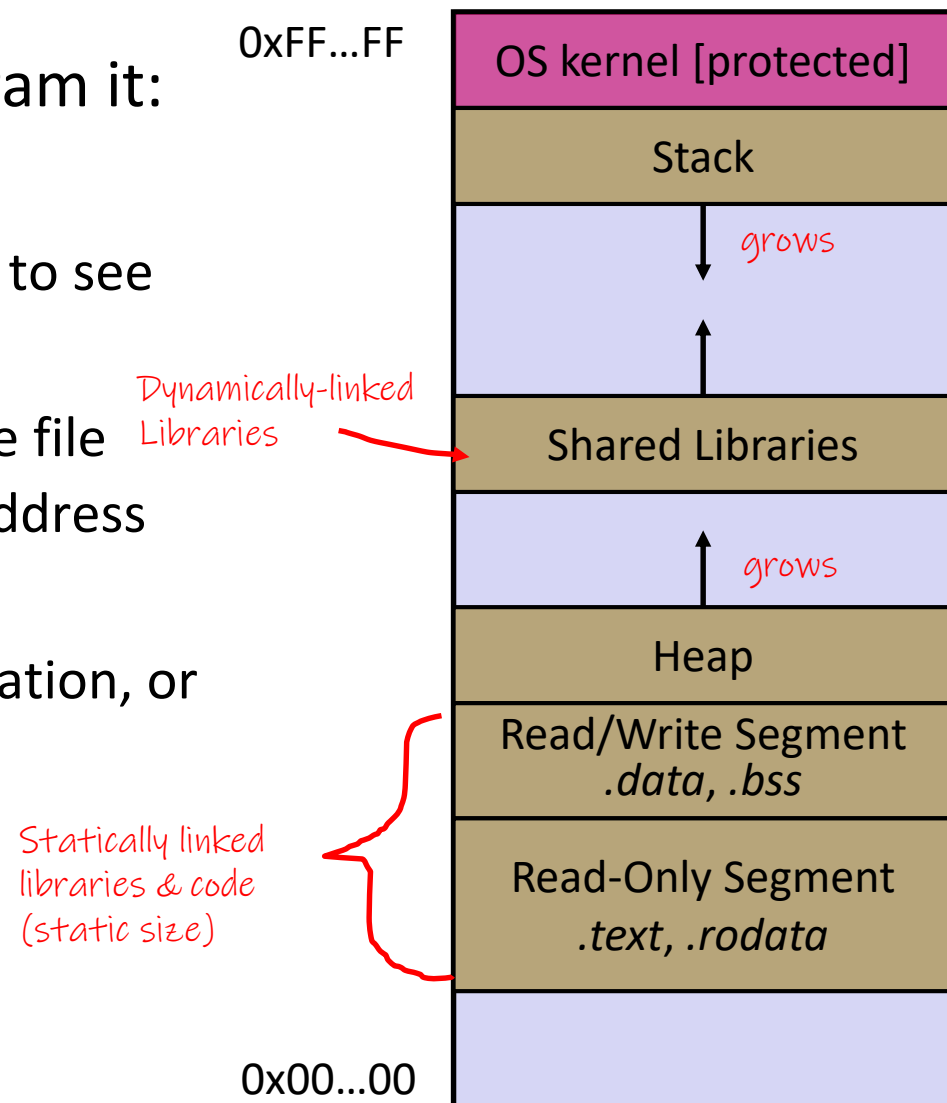
# Processes and Virtual Memory

- ❖ The OS gives each process the illusion of its own private memory
  - Called the process' **address space**
  - Contains the process' virtual memory, visible only to it (via translation)
  - $2^{64}$  bytes on a 64-bit machine



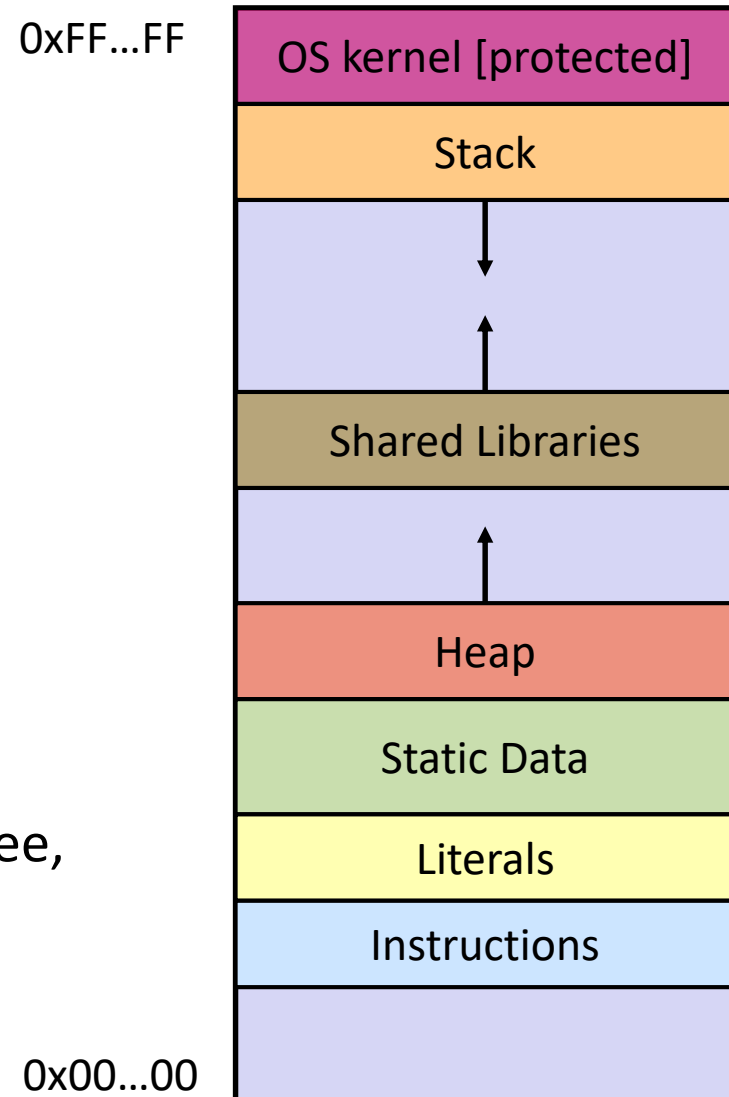
# Loading

- ❖ When the OS loads a program it:
  - 1) Creates an address space
  - 2) Inspects the executable file to see what's in it
  - 3) (Lazily) copies regions of the file into the right place in the address space
  - 4) Does any final linking, relocation, or other needed preparation



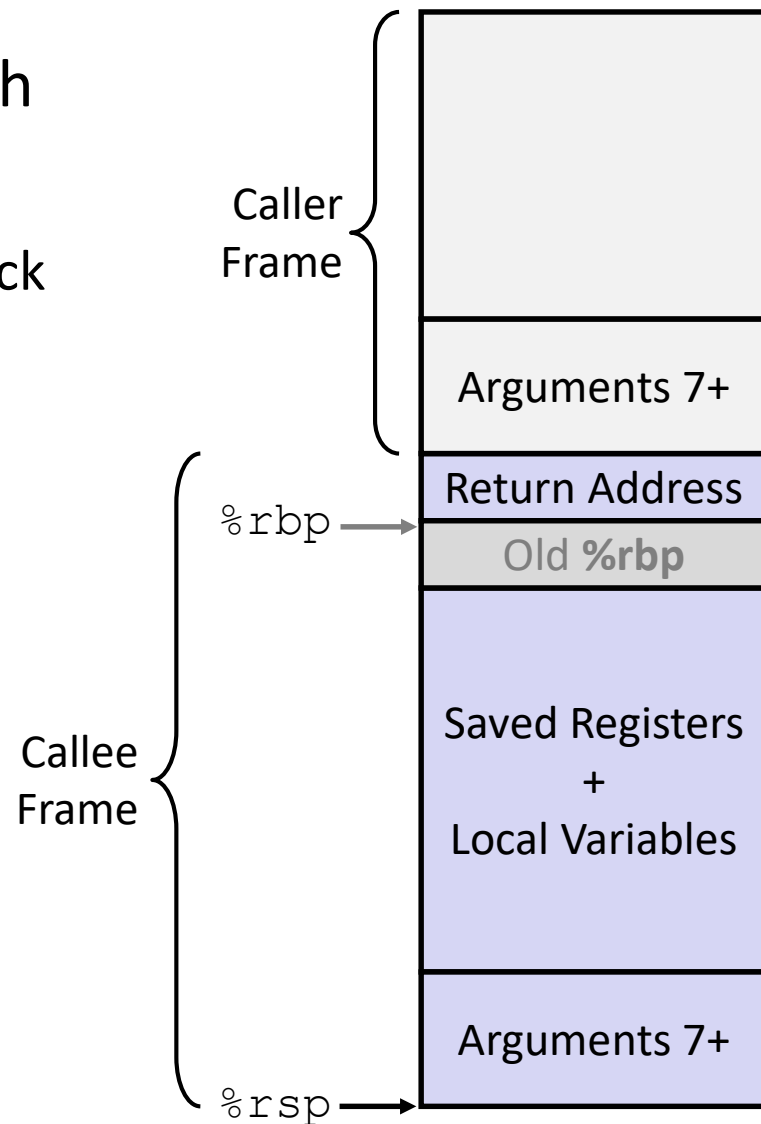
# Memory Management

- ❖ *Local* variables on the Stack
  - Allocated and freed via calling conventions (`push`, `pop`, `mov`)
- ❖ *Global* and *static* variables in Data
  - Allocated/freed when the process starts/exits
- ❖ *Dynamically-allocated* data on the Heap
  - `malloc()` to request; `free()` to free, otherwise **memory leak**
  - More about this on Monday



# Review: The Stack

- ❖ Used to store data associated with function calls
  - Compiler-inserted code manages stack frames for you
- ❖ Stack frame (x86-64) includes:
  - Address to return to
  - Saved registers
    - Based on calling conventions
  - Local variables
  - Argument build
    - Only if > 6 used



# Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

```

#include <stdlib.h>

int f(int, int);
int g(int);

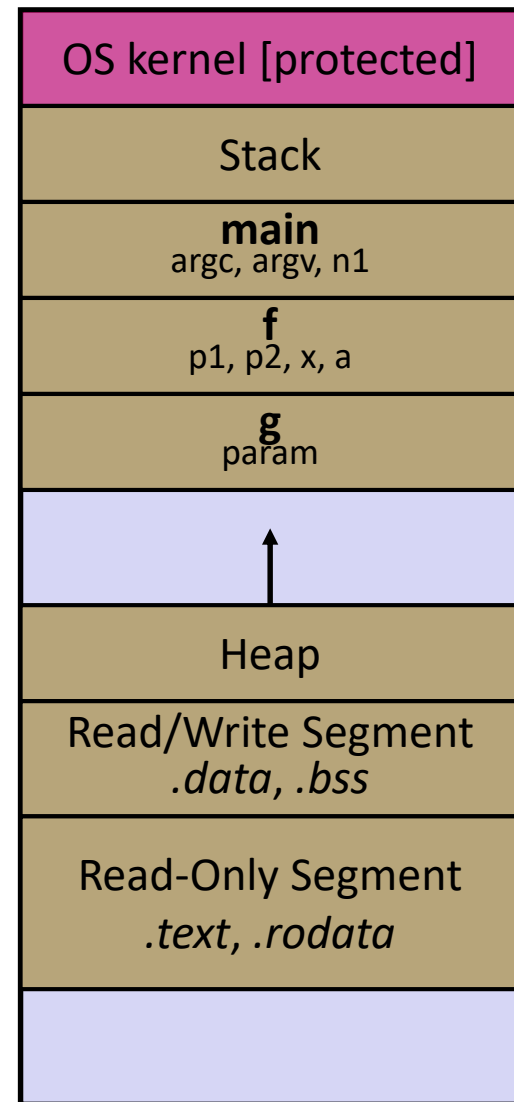
int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
    
```

*Note the use of function declarations above. Definitions below main().*

*Parameters are local data too!*



# Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

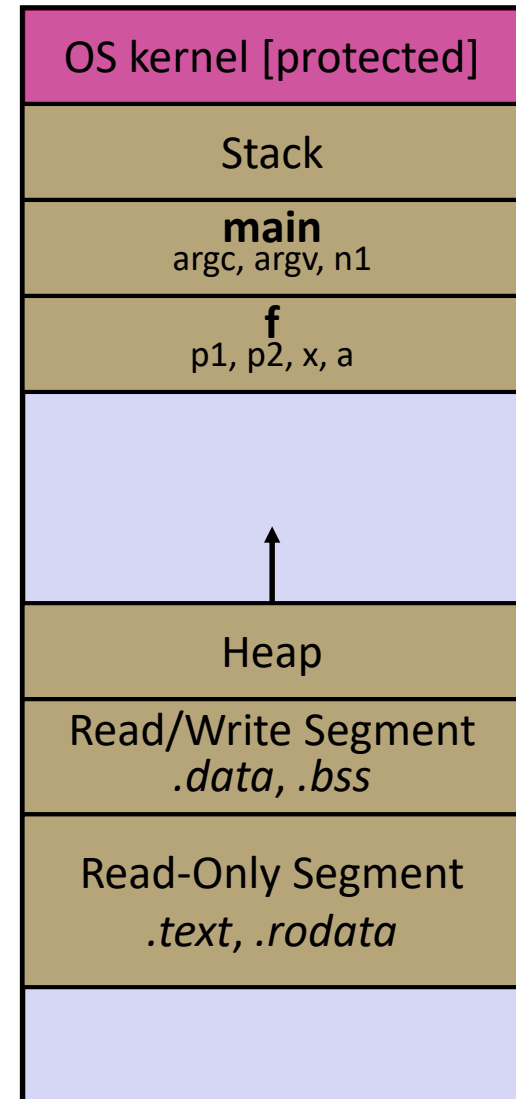
```
#include <stdlib.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```



# Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

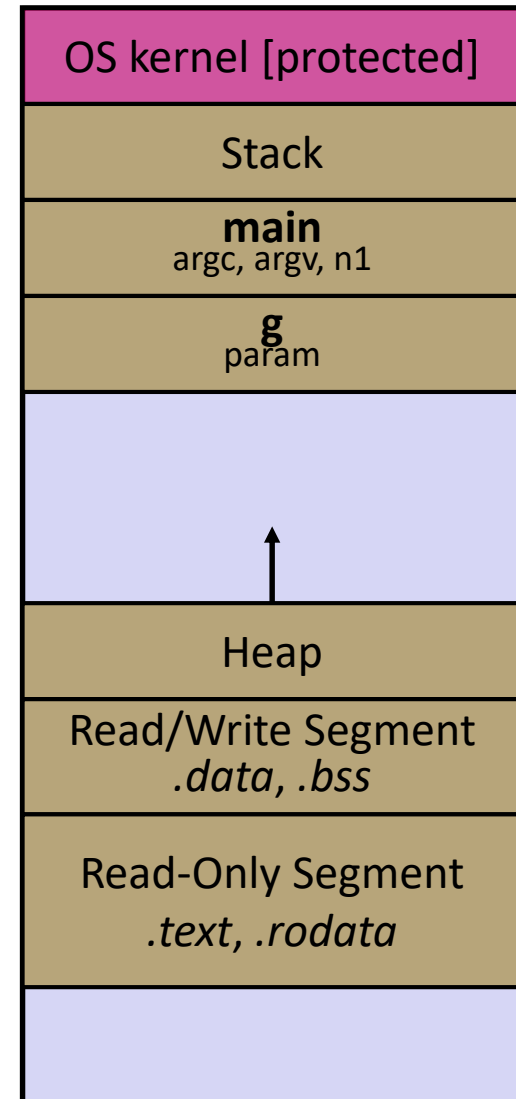
```
#include <stdlib.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```



# Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

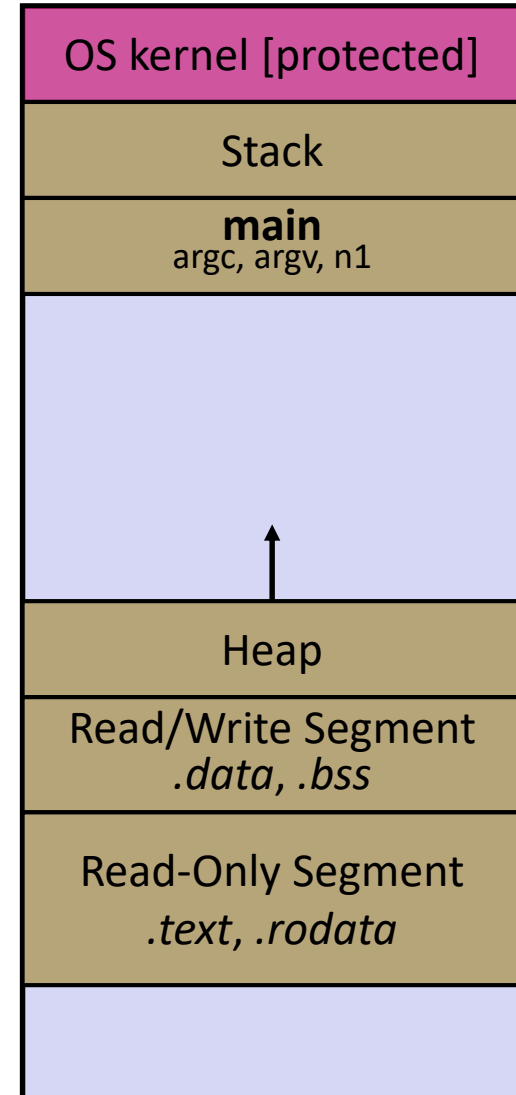
```
#include <stdlib.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```



# Lecture Outline

- ❖ Function Definitions vs. Declarations
- ❖ C's Memory Model (CSE 351 refresher)
- ❖ **Pointers** (CSE 351 refresher)
- ❖ Arrays

# Pointers

## ❖ Variables that store addresses

- It points to somewhere in the process' virtual address space
- `&foo` produces the virtual address of `foo`

## ❖ Generic definition: `type* name;` or `type *name;`

Use either, but be consistent

equivalent

- Recommended: do not define multiple pointers on same line:

Same issue with `int* p1, p2`

`int *p1, p2;` <sup>ptr int</sup>

not the same as

`int *p1, *p2;` <sup>ptr ptr</sup>

- Instead, use:

`int *p1;`

`int *p2;`

## ❖ Dereference a pointer using the unary `*` operator

- Access the memory referred to by a pointer

# Pointer Example

## pointy.c

```
#include <stdio.h>


int main(int argc, char** argv) {
    int x = 351;
    int* p;    // p is a pointer to a int

    p = &x;    // p now contains the addr of x
    printf("&x is %p\n", &x);
    printf(" p is %p\n", p);
    printf(" x is %d\n", x);
    printf("*p is %d\n", *p);

    *p = 333;  // change value of x
    printf(" x is %d\n", x);

    return 0;
}
```

<b>x</b>	333
<b>p</b>	&x



# Something Curious

- ❖ What happens if we run `pointy.c` several times?

```
bash$ gcc -Wall -std=c17 -o pointy pointy.c
```

Run 1:

```
bash$ ./pointy
&x is 0x7ffff9e28524
p is 0x7ffff9e28524
x is 351
x is 333
```

Run 2:

```
bash$ ./pointy
&x is 0x7fffe847be34
p is 0x7fffe847be34
x is 351
x is 333
```

Run 3:

```
bash$ ./pointy
&x is 0x7fffe7b14644
p is 0x7fffe7b14644
x is 351
x is 333
```

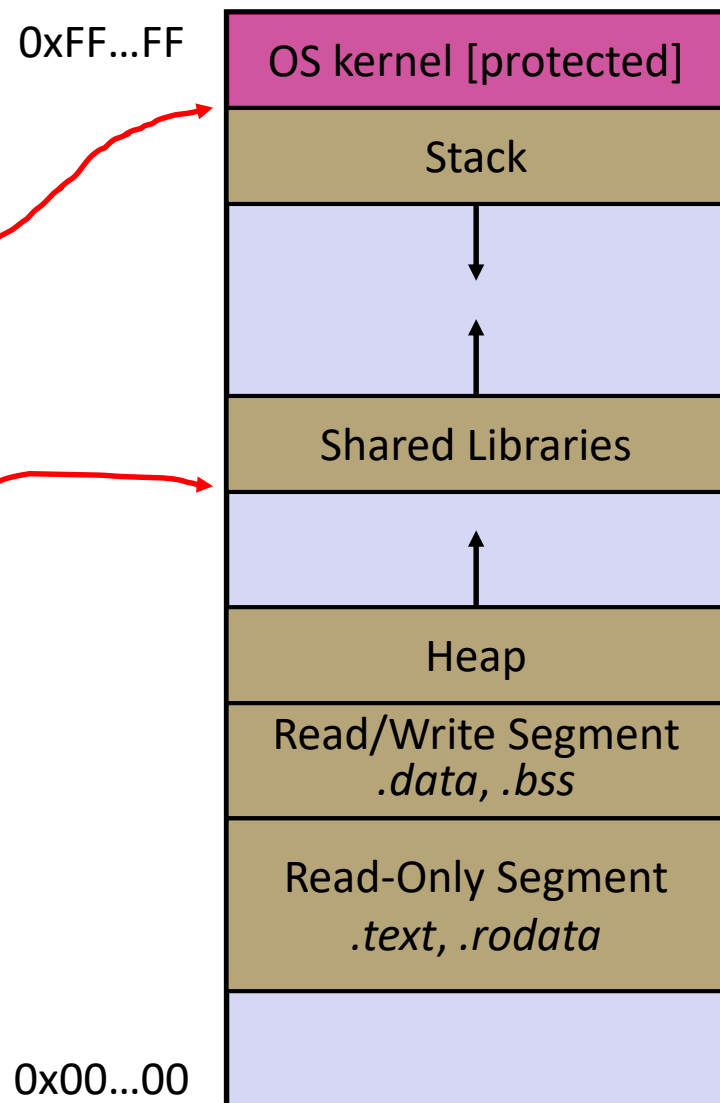
Run 4:

```
bash$ ./pointy
&x is 0x7fffff0dfe54
p is 0x7fffff0dfe54
x is 351
x is 333
```

# Address Space Layout Randomization

❖ Linux uses *address space layout randomization* (ASLR) for added security

- Randomizes:
  - Base of stack
  - Shared library (`mmap`) location
- Makes Stack-based buffer overflow attacks tougher 😊
- Makes debugging tougher ☹️
- Can be disabled (`gdb` does this by default); Google if curious



 **Poll Everywhere**[pollev.com/cse333](https://pollev.com/cse333)

❖ When run, what does this code print?

- A. 2
- B. 333
- C. 999
- D. A return address
- E. Undefined Behavior
- F. We're Lost...

```
int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}
```

# Answer

## local\_addr.c

```
#include <stdint.h>
#include <stdio.h>

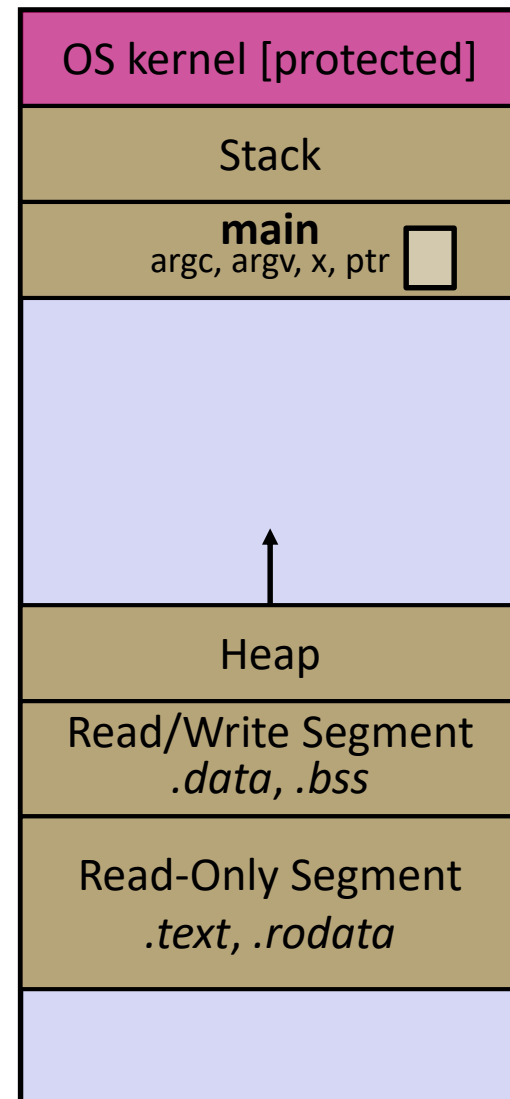
int64_t foo();
int64_t bar(int64_t);

int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}
```

## Answer: Undefined Behaviour



# Answer

## local\_addr.c

```

#include <stdint.h>
#include <stdio.h>

int64_t foo();
int64_t bar(int64_t);

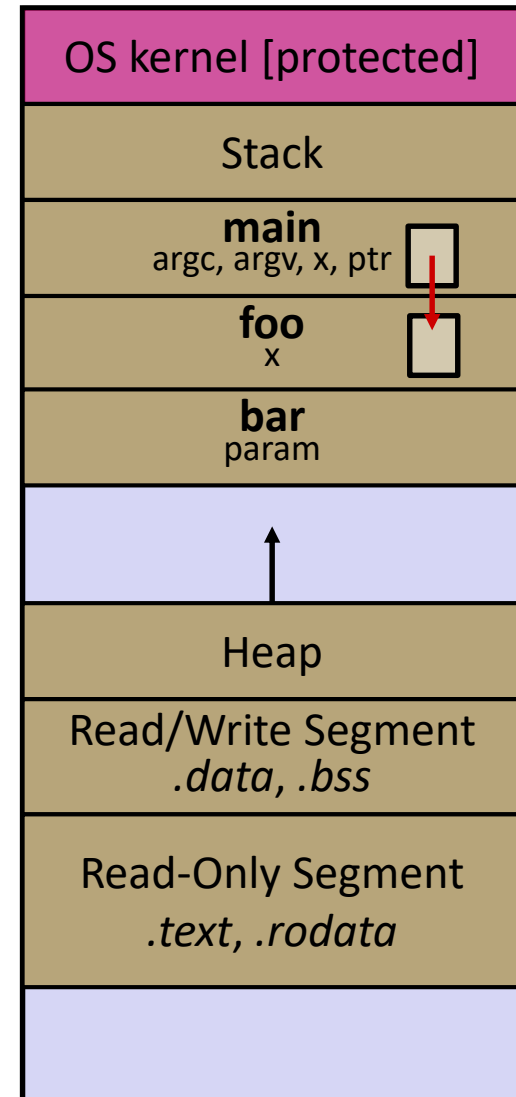
int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}

```

## Answer: Undefined Behaviour



# Answer

## local\_addr.c

```
#include <stdint.h>
#include <stdio.h>

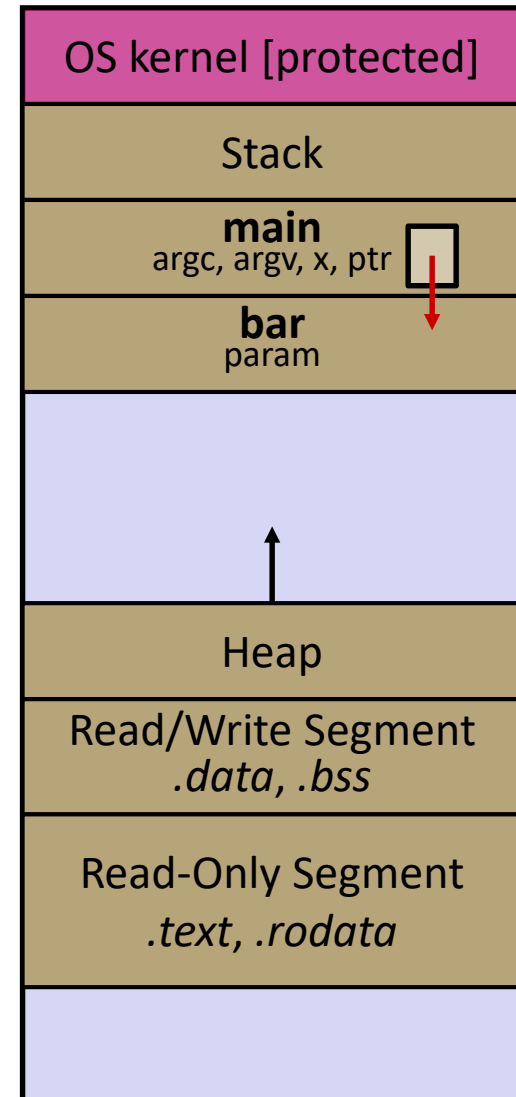
int64_t foo();
int64_t bar(int64_t);

int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}
```

## Answer: Undefined Behaviour



# Answer

## local\_addr.c

```
#include <stdint.h>
#include <stdio.h>

int64_t foo();
int64_t bar(int64_t);

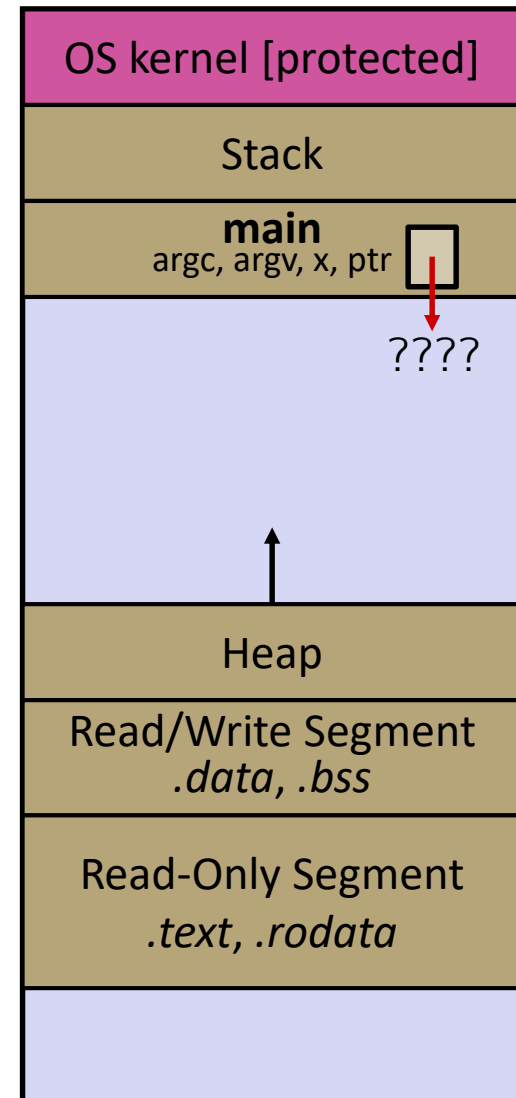
int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}
```



## Answer: Undefined Behaviour



# Lecture Outline

- ❖ Function Definitions vs. Declarations
- ❖ C's Memory Model (CSE 351 refresher)
- ❖ Pointers (CSE 351 refresher)
- ❖ **Arrays**

# Arrays

- ❖ Definition: `type name [size]`
  - Allocates `size * sizeof (type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (e.g. `int scores [175];`)
  - **Initially, array values are “garbage”**
- ❖ Size of an array
  - Not stored anywhere – array does not know its own size!
    - `sizeof (array)` only works in variable scope of array definition
  - Recent versions of C allow for variable-length arrays
    - Uncommon and can be considered bad practice [*we won't use*]

```
int n = 175;  
int scores[n]; // OK in C99
```

# Using Arrays

Optional when initializing

❖ Initialization: `type name [size] = {val0, ..., valN};`

- `{ }` initialization can *only* be used at time of definition
- If no `size` supplied, infers from length of array initializer

❖ Array name used as identifier for “collection of data”

- `name [index]` specifies an element of the array and can be used as an assignment target or as a value in an expression

✳ Array name (by itself) produces the address of the start of the array

- Cannot be assigned to / changed

```
int primes[6] = {2, 3, 5, 6, 11, 13};  
primes[3] = 7;  
primes[100] = 0; // memory smash!
```

No IndexOutOfBounds  
Hope for segfault

# Multi-dimensional Arrays

## ❖ Generic 2D format:

```
type name[rows][cols] = {{values}, ..., {values}};
```

- Still allocates a single, contiguous chunk of memory
- C is *row-major*

```
// a 2-row, 3-column array of doubles
double grid[2][3];

// a 3-row, 5-column array of ints
int matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
};
```

- 2-D arrays normally only useful if size known in advance. Otherwise use dynamically-allocated data and pointers (later)

# Arrays as Parameters

## ❖ It's tricky to use arrays as parameters

- What happens when you use an array name as an argument?
- Arrays do not know their own size

*Gets address of start of array*

```
int sumAll(int a[]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers);
    return 0;
}

int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < ...???)
}
```

# Solution 1: Declare Array Size

```
int sumAll(int a[5]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++) {
        sum += a[i];
    }
    return sum;
}
```

- ❖ Problem: loss of generality/flexibility

# Solution 2: Pass Size as Parameter

```
int sumAll(int a[], int size); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers, 5);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[], int size) {
    int i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += a[i];
    }
    return sum;
}
```

arraysum.c

- Standard idiom in C programs

# Parameters: reference vs. value

- ❖ There are two fundamental parameter-passing schemes in programming languages
- ❖ **Call-by-value**
  - Parameter is a local variable initialized with a copy of the calling argument when the function is called; manipulating the parameter only changes the copy, *not* the calling argument
  - **C, Java, C++** (most things)
- ❖ **Call-by-reference**
  - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
  - C++ references (we'll see these later)

# So what's the story for arrays?

- ❖ Is it call-by-value or call-by-reference?
- ❖ Technical answer: a  $T[]$  array parameter is “promoted” to a pointer of type  $T^*$ , and the *pointer* is passed by value
  - So it acts like a call-by-reference array (if callee changes the array parameter elements it changes the caller's array)
  - But it's really a call-by-value pointer (the callee can change the pointer parameter to point to something else(!))
    - This is because  $T[i]$  is really  $*(T+i)$ . We aren't changing  $T$ !

```
void copyArray(int src[], int dst[], int size) {
    int i;
    dst = src;    // Doesn't copy the array, copies address!
    for (i = 0; i < size; i++) {
        dst[i] = src[i];    // copies source array to itself!
    }
}
```

# Returning an Array

- ❖ Local variables, including arrays, are allocated on the Stack
  - They “disappear” when a function returns!
  - Can't safely return local arrays from functions
    - Can't return an array as a return value – why not?

```
int* copyArray(int src[], int size) {  
    int i, dst[size];    // OK in C99  
  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
  
    return dst;    // no compiler error, but wrong!  
}
```

*returns address of start of local array on stack*

buggy\_copyarray.c

# Solution: Output Parameter

- ❖ Create the “returned” array in the caller
  - Pass it as an **output parameter** to `copyarray()`
    - A pointer parameter that allows the called function to store values that the caller can use
  - Works because arrays are “passed” as pointers
    - “Feels” like call-by-reference, but technically it’s not

```
void copyArray(int src[], int dst[], int size) {  
    int i;  
  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
    dst[i] = *(dst + i);  
}
```

*Output parameter used to “return” data*

*Data stored by dereferencing pointer*

copyarray.c

# Output Parameters

❖ Output parameters are common in library functions

- `long int strtol(char* str, char** endptr, int base);`  
*Output parameters* (pointing to `endptr`)
- `int sscanf(char* str, char* format, ...);`  
*Output parameters* (pointing to `...`)

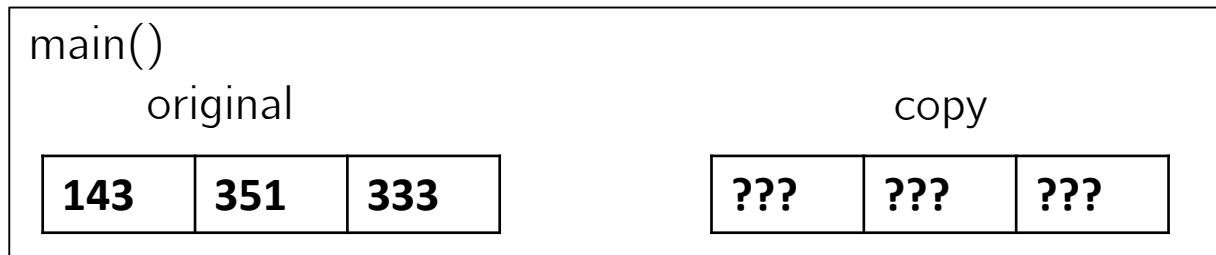
```
int    num, i;
char*  pEnd, str1 = "333 rocks";
char   str2[10];

// converts "333 rocks" into long -- pEnd is conversion end
num = (int) strtol(str1, &pEnd, 10);
    ↖ "returns" data in 2 ways ↗

// reads string into arguments based on format string
num = sscanf("3 blind mice", "%d %s", &i, str2);
```

outparam.c

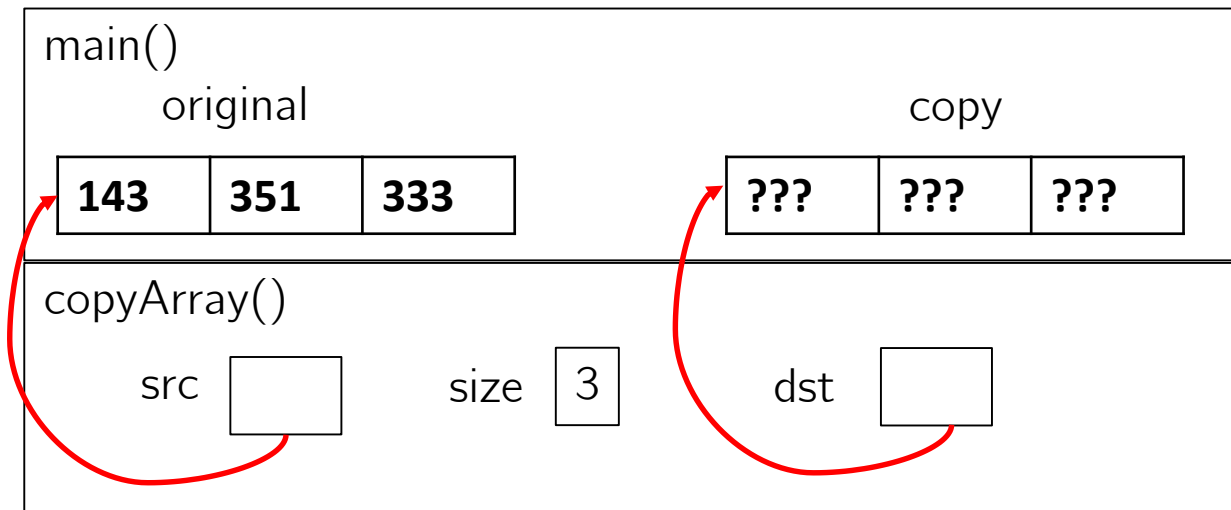
# Array Memory Diagram



```
int main() {
    int original[] = {143, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

# Array Memory Diagram

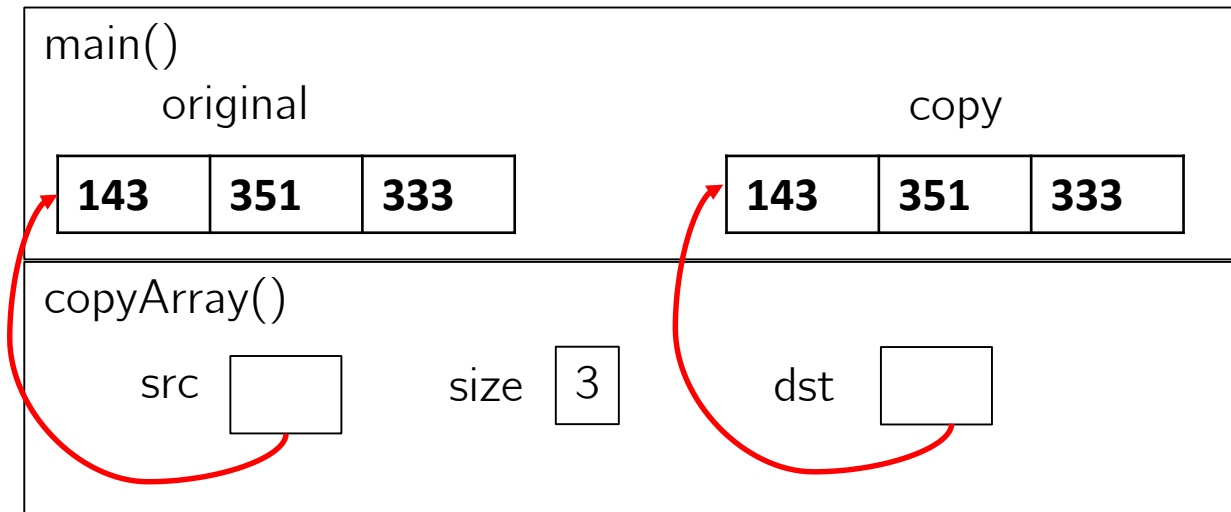


```
int main() {
    int original[] = {143, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

`dst[i]` is really  
`*(dst+i)`. We  
aren't changing `dst`!

# Array Memory Diagram



```
int main() {
    int original[] = {143, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

`dst[i]` is really `*(dst+i)`. We aren't changing `dst`!

# Extra Exercises

- ❖ Some lectures contain “Extra Exercise” slides
  - Extra practice for you to do on your own without the pressure of being graded
  - You may use libraries and helper functions as needed
    - Early ones may require reviewing 351 material or looking at documentation for things we haven’t discussed in 333 yet
  - Always good to provide test cases in `main()`
- ❖ Solutions for these exercises will be posted on the course website
  - You will get the most benefit from implementing your own solution before looking at the provided one

# Extra Exercise #1

- ❖ Write a function that:
  - Accepts an array of 32-bit unsigned integers and a length
  - Reverses the elements of the array in place
  - Returns nothing (`void`)

# Extra Exercise #2

- ❖ Write a function that:
  - Accepts a string as a parameter
  - Returns:
    - The first white-space separated word in the string as a newly-allocated string
    - AND the size of that word
  - (probably need to wait until we look at malloc/free later)