

pollev.com/cse333a

How is course setup going for you?

Vote for *each* of CSE Linux environment, text editor, and Gitlab/git.

- A. **Done! Went (relatively) smoothly.**
- B. **Done! Was tough to set up.**
- C. **Still working on it.**
- D. **Haven't tried to set it up yet.**

Systems Programming

Memory, Data, Parameters

Instructors:

Amber Hu

Justin Hsia

Teaching Assistants:

Ally Tribble

Blake Diaz

Connor Olson

Grace Zhou

Jackson Kent

Janani Raghavan

Jen Xu

Jessie Sun

Jonathan Nister

Mendel Carroll

Rose Maresh

Violet Monserate

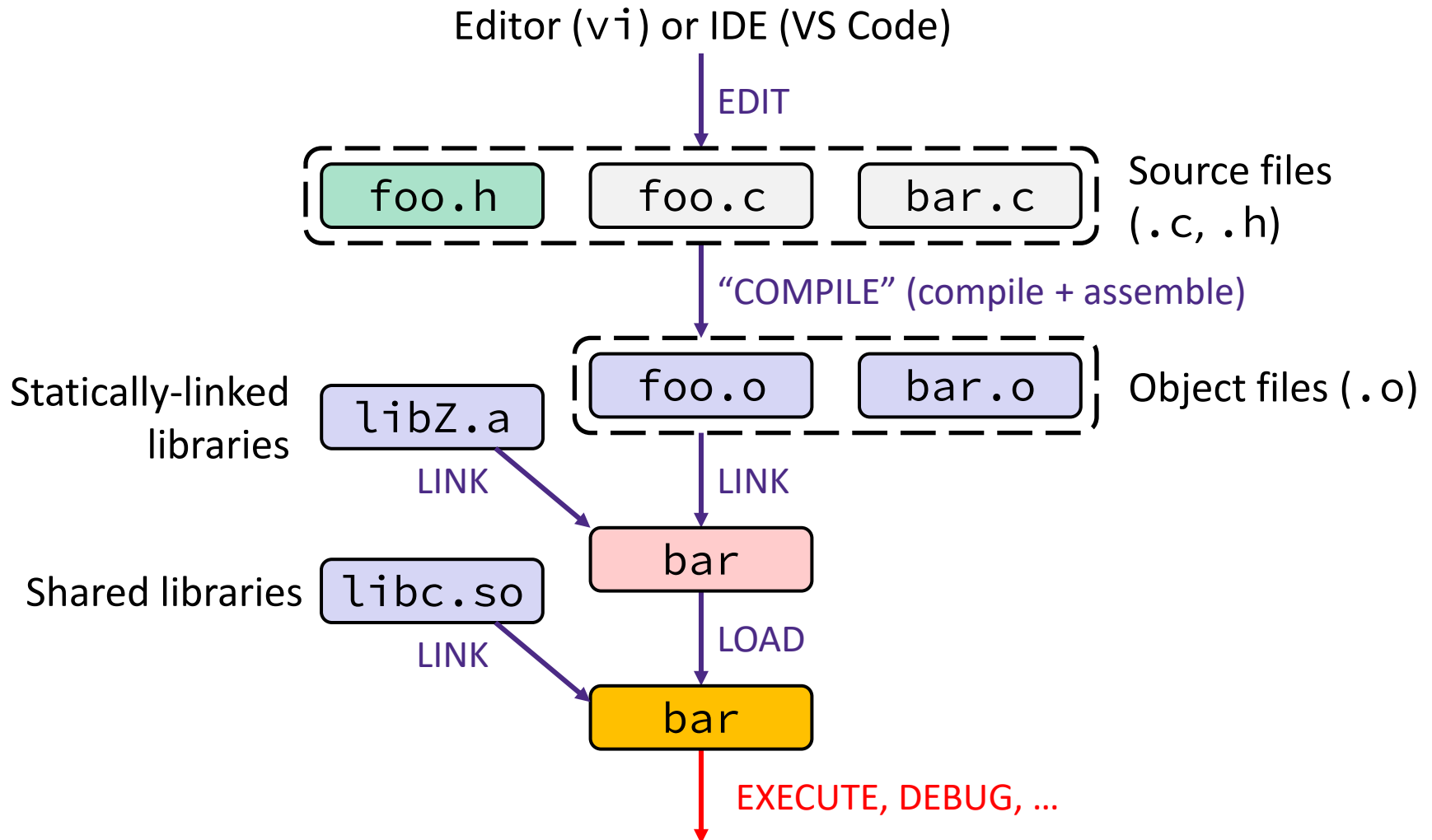
Relevant Course Information (1/2)

- ❖ Pre-quarter survey due Friday, 11:59 pm (Canvas)
- ❖ Exercise 0 was due at 11:00 am today
 - No late days for exercises
 - Solutions have been released
 - Style issues Ed post will be released soon
 - Ex 0 grades will be released before Ex 2 is due
- ❖ Exercise 1 due Friday morning, 11:00 am
 - Submission via Gradescope (contact us if you don't have access)
 - Make sure your solution compiles & runs properly on the CSE Linux environment before submitting

Relevant Course Information (2/2)

- ❖ Starting with Exercise 2 (released Friday), all submissions will be done via Gitlab tagging
 - We will submit to Gradescope from your repo for you
 - Gitlab email sent when repos created – no action needed
 - Make a private Ed post if you don't have your repos yet
 - Set up an SSH key and clone repos ASAP
 - Clone your repo before section tomorrow!

C Compilation Workflow



Multi-file C Programs

Note: This example has poor style for code split. More on multiple files in Lecture 5.

C source file 1
(sumstore.c)

```
void SumStore(int x, int y, int* dest) {  
    *dest = x + y;  
}
```

C source file 2
(sumnum.c)

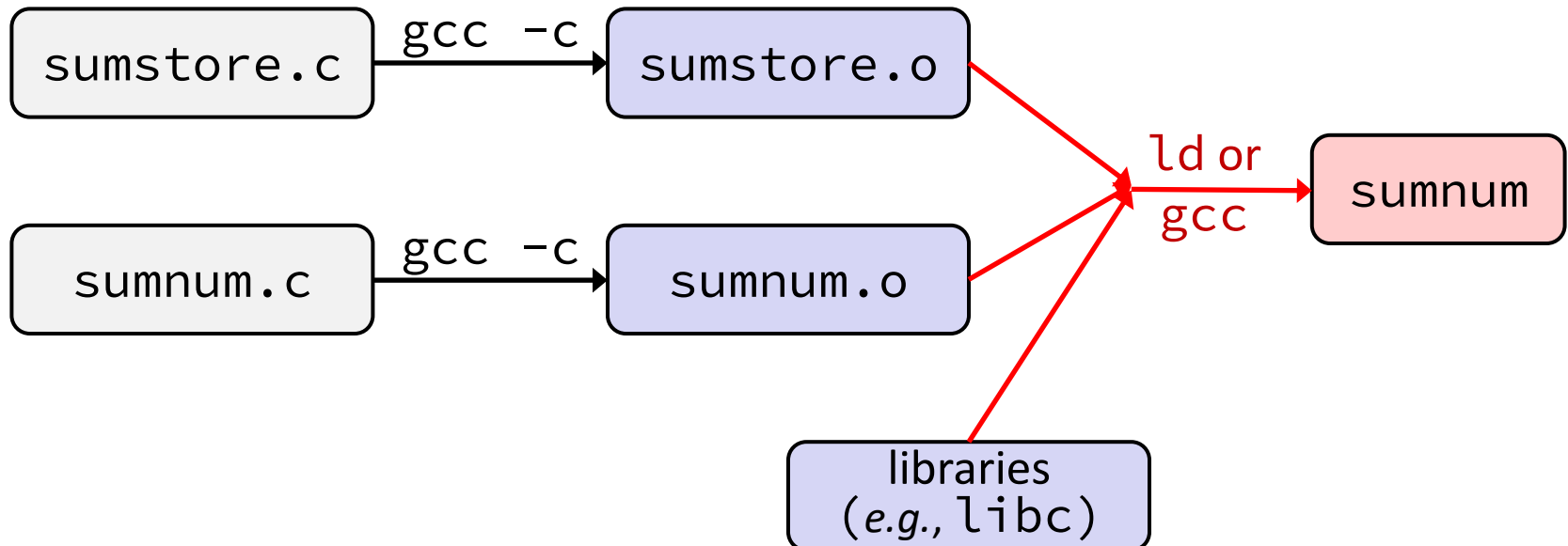
```
#include <stdio.h>  
#include <stdlib.h>  
  
void SumStore(int x, int y, int* dest);  
  
int main(int argc, char** argv) {  
    int z, x = 351, y = 333;  
    SumStore(x, y, &z);  
    printf("%d + %d = %d\n", x, y, z);  
    return EXIT_SUCCESS;  
}
```

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



pollev.com/cse333a

Which of the following statements are FALSE?

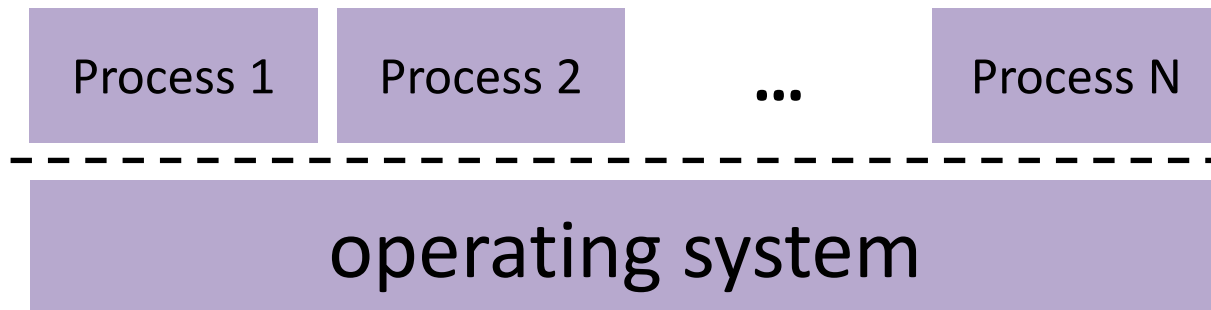
- A. With the standard `main` syntax, it is always safe to use `argv[0]`
- B. Your program's returned status code is unimportant
- C. Using function declarations is beneficial to both single- and multi-file C programs
- D. Defined error constants need to be looked up in function documentation, man pages, or header files like `errno.h`
- E. We're lost...

Lecture Outline (1/3)

- ❖ **Memory Management** (351 refresher)
- ❖ C Data Considerations
- ❖ C Parameters
 - Arrays and Pointers as Parameters

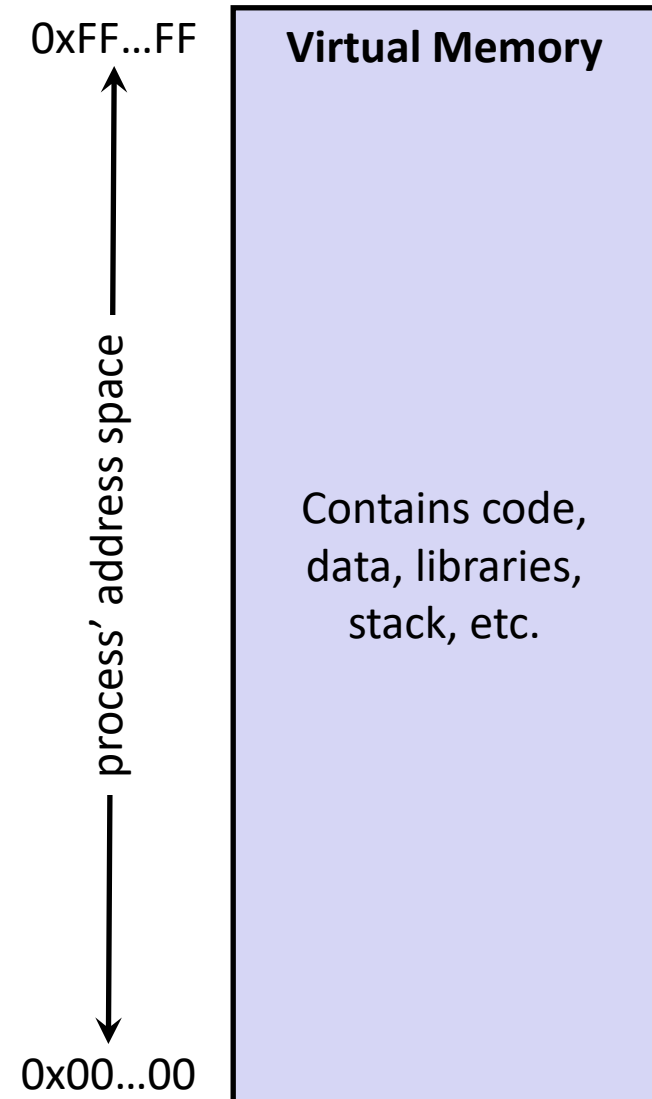
OS and Processes

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



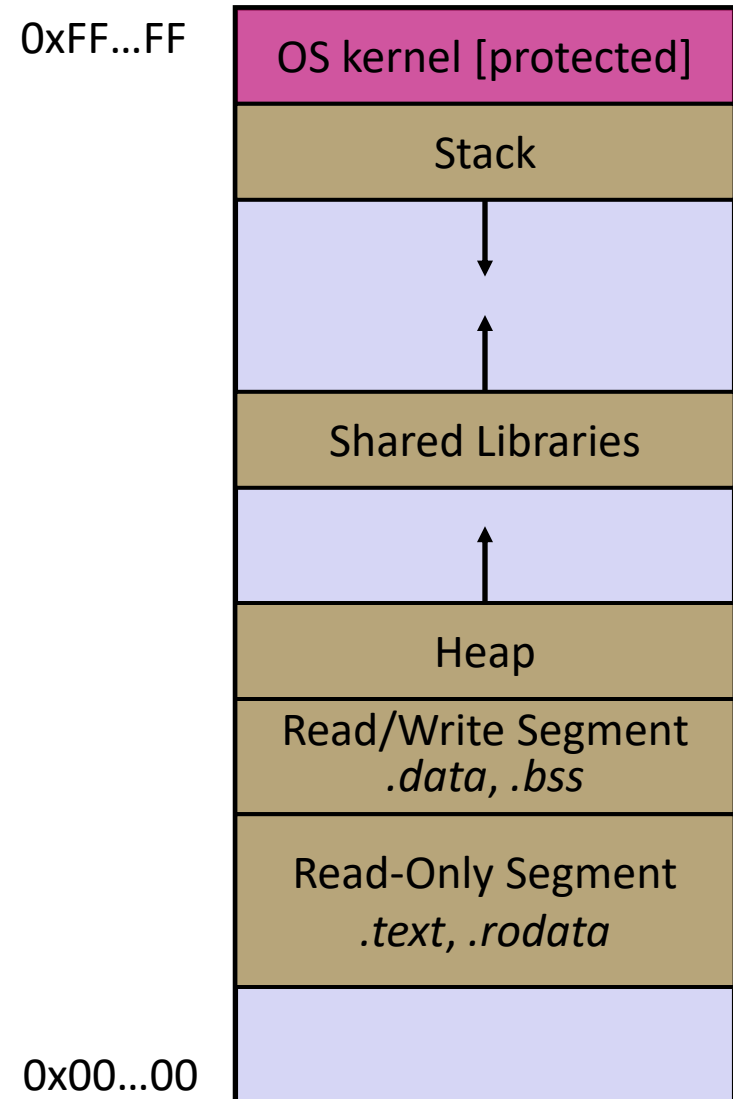
Processes and Virtual Memory

- ❖ 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

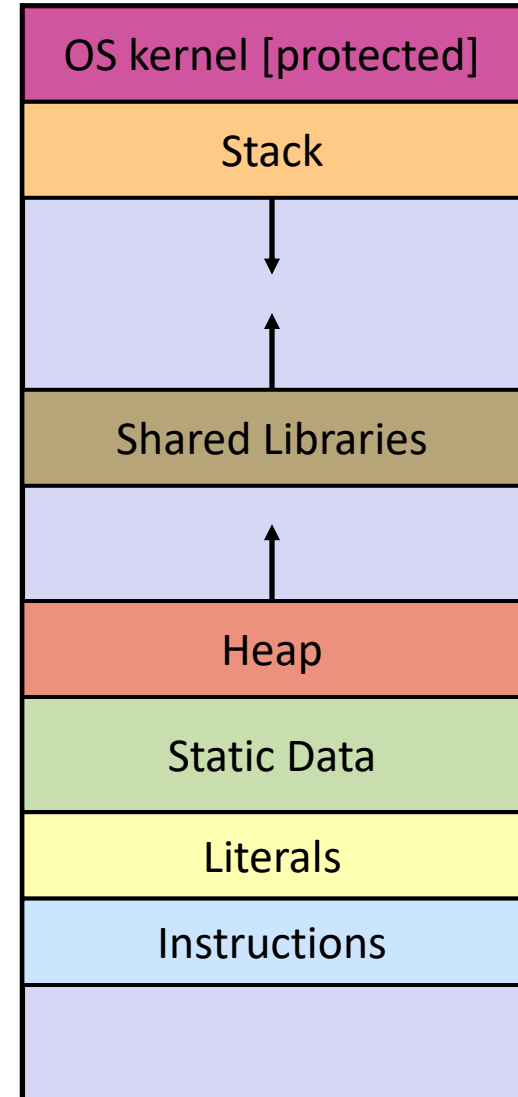


Memory Management

- ❖ *Local* variables on the Stack
 - **Automatically**-allocated and deallocated via calling conventions (push, pop, mov)
- ❖ *Global* and *static* variables in Data
 - **Statically**-allocated when the process starts and deallocated when it exits
- ❖ `malloc`-ed data on the Heap
 - **Dynamically**-allocated by process
 - Must call `free()` to free, otherwise a **memory leak**

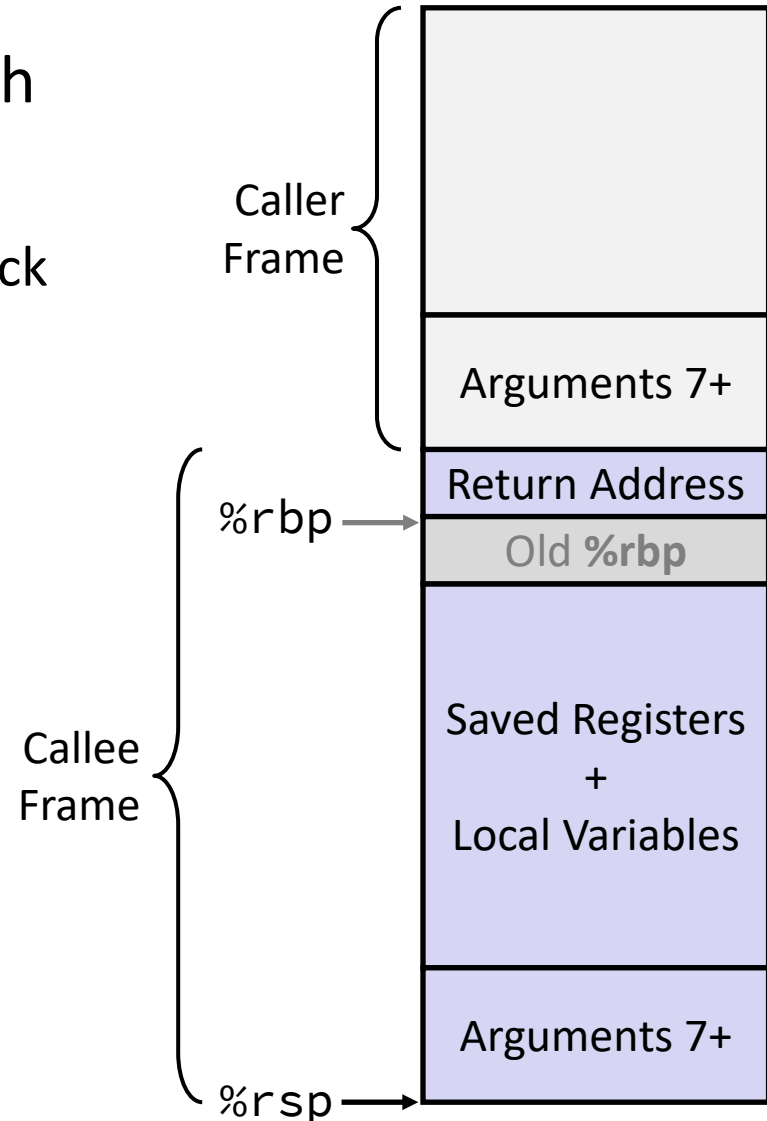
0xFF...FF

0x00...00



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 (1/4)

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

stack.c

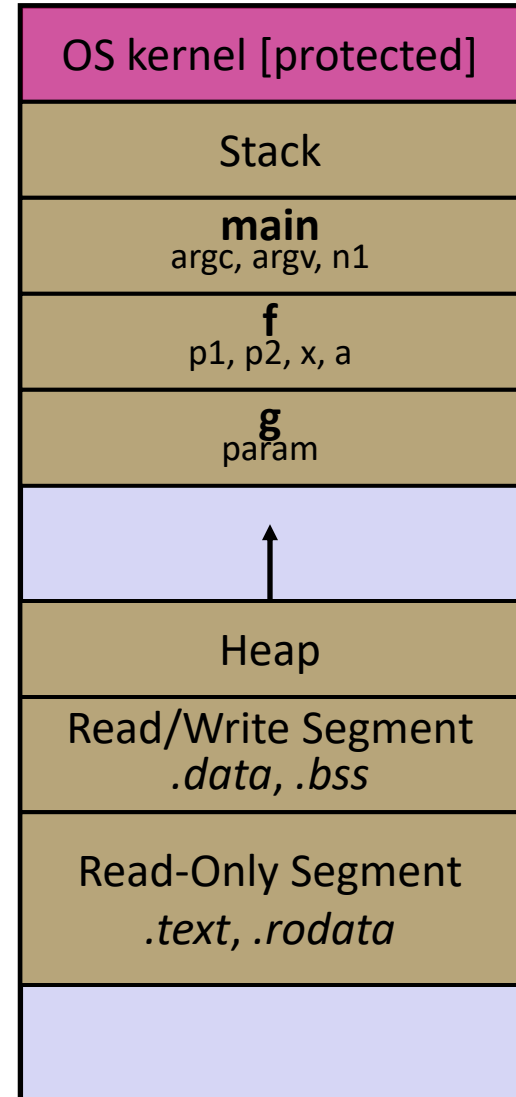
```
#include <stdlib.h>

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

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

→ 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 (2/4)

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

stack.c

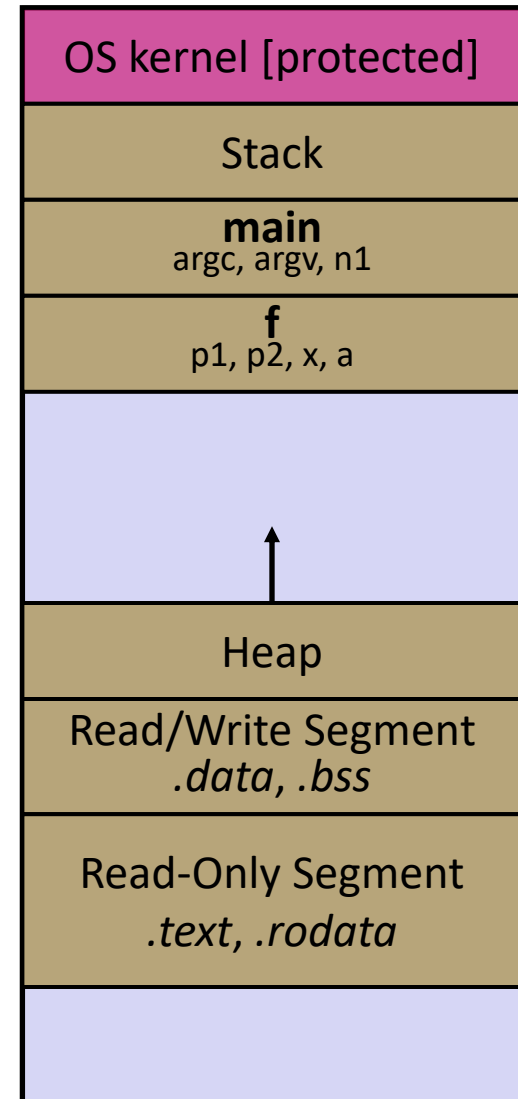
```
#include <stdlib.h>

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

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

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 (3/4)

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

stack.c

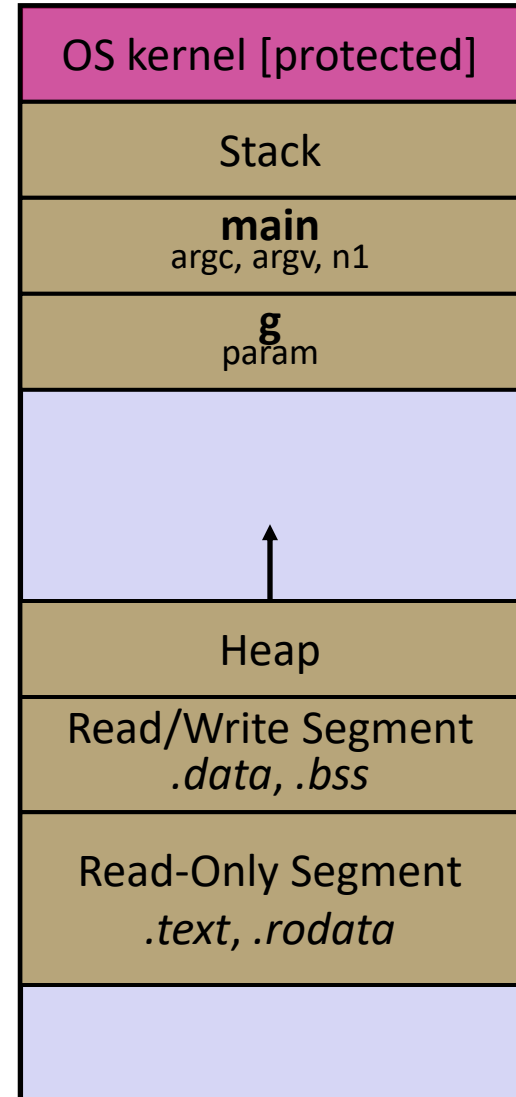
```
#include <stdlib.h>

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

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

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 (4/4)

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

stack.c

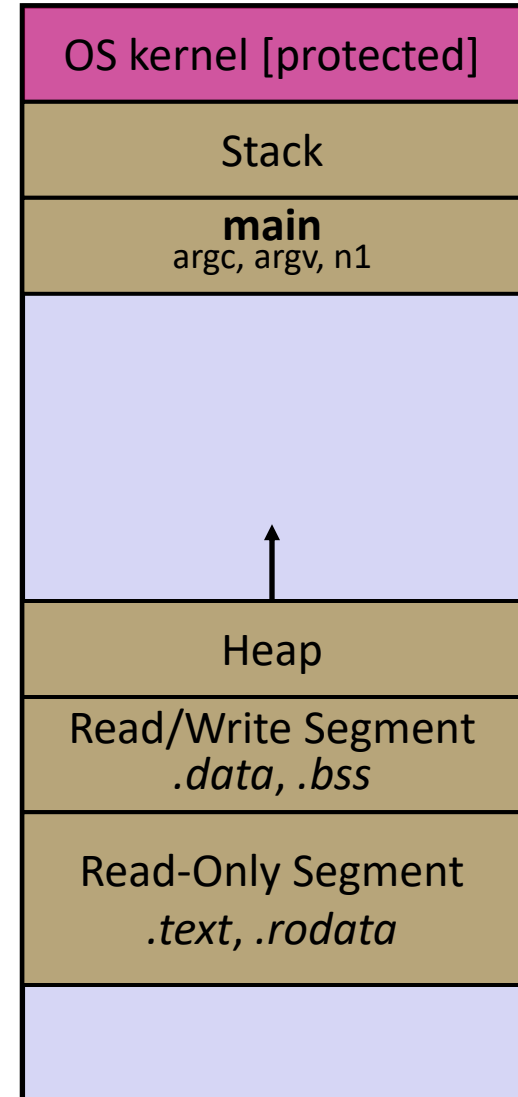
```
#include <stdlib.h>

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

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

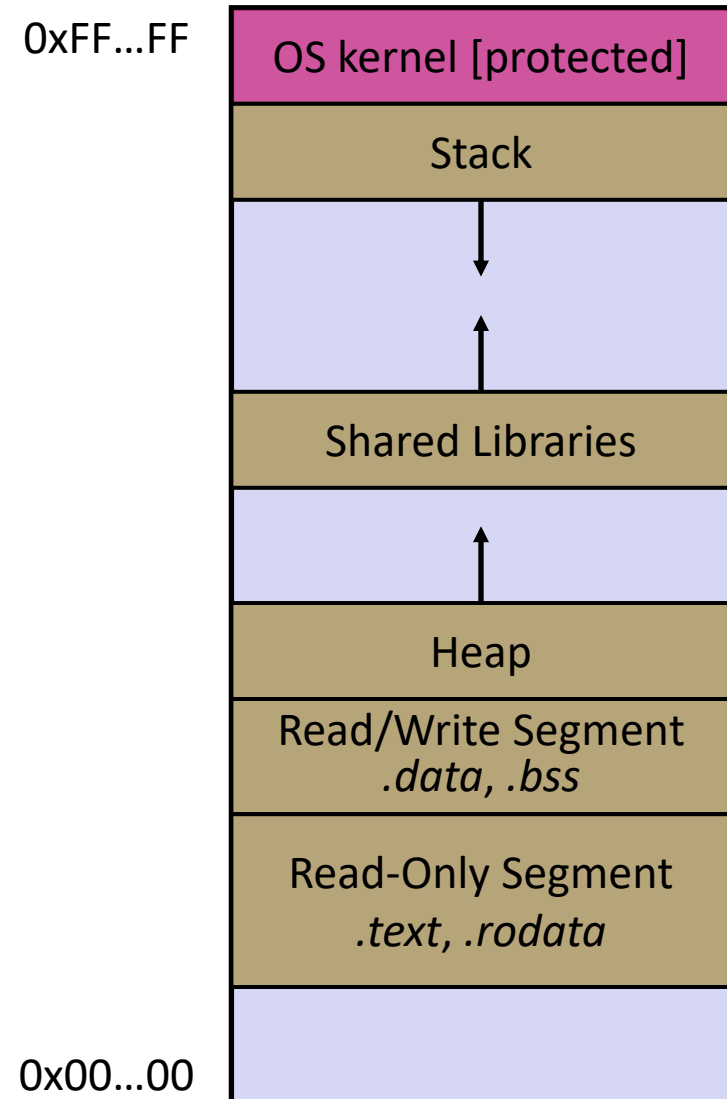
int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

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



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 is curious



Lecture Outline (2/3)

- ❖ Memory Management (351 refresher)
- ❖ **C Data Considerations**
- ❖ Parameters

C Primitive Types and Memory

Do not memorize, these aren't strict sizes!

❖ Integer types

- `char`, `int`

❖ Floating point

- `float`, `double`

❖ Modifiers

- `short` [`int`]
- `long` [`int`, `double`]
- `signed` [`char`, `int`]
- `unsigned` [`char`, `int`]

C Data Type	32-bit	64-bit	printf
char	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	12	16	%Lf
pointer	4	8	%p



C99 Extended Integer Types

- ❖ Solves the conundrum of “how big is an `long int`?”

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

```
void foo(void) {  
    int8_t  a;  // exactly 8 bits, signed  
    int16_t b;  // exactly 16 bits, signed  
    int32_t c;  // exactly 32 bits, signed  
    int64_t d;  // exactly 64 bits, signed  
    uint8_t w;  // exactly 8 bits, unsigned  
    ...  
}
```

```
void sumstore(int x, int y, int* dest) {
```



```
void sumstore(int32_t x, int32_t y, int32_t* dest) {
```

Arrays

- ❖ Definition: `type name[size]` allocates $\text{size} * \text{sizeof}(\text{type})$ bytes of *contiguous* memory
 - By default, array values are “mystery” data (*i.e.*, uninitialized)
 - Normal usage is a compile-time constant for `size`
e.g., `int scores[175];`
- ❖ Size of an array
 - Not stored anywhere – array does not know its own size!
 - `sizeof(array)` only works in the variable scope of array definition
 - Recent versions of C (but *not* 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

❖ 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”

- Array name produces the address of the start of the array
 - Cannot be assigned to / changed
- `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
 - Is actually `*(name+index)` with pointer arithmetic (Lecture 3)

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

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

Structs

- ❖ The *size* and *layout* of a struct instance is completely determined by (1) the field ordering and (2) alignment requirements
 - Can review 351 if curious
- ❖ In practical terms, wouldn't solve for these by hand; use built-in C functionality instead:
 - `sizeof(type)` returns the size in bytes
 - `offsetof(type, field)` returns offset value in bytes
 - Defined in `stddef.h`
- ❖ We'll talk more about struct usage in Lecture 4

Lecture Outline (3/3)

- ❖ Memory Management (351 refresher)
- ❖ C Data Considerations
- ❖ **Parameters**

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)

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

```
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!

Arrays: Call-by-what?

- ❖ 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* – caller’s array can be changed if callee modifies the array parameter elements
 - But it’s really a *call-by-value pointer* – the callee’s pointer parameter can be changed without affecting the caller’s array
 - 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 the address  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i]; // copies source array to itself  
    }  
}
```



Array Parameters

- ❖ Array parameters are *actually* passed as pointers to the first array element
 - The `[]` syntax for parameter types is just for convenience
 - Use whichever best helps the reader

This code:

```
void f(int a[]);

int main( ... ) {
    int a[5];
    ...
    f(a);
    return EXIT_SUCCESS;
}

void f(int a[]) {
```

Equivalent to:

```
void f(int* a);

int main( ... ) {
    int a[5];
    ...
    f(&a[0]);
    return EXIT_SUCCESS;
}

void f(int* a) {
```

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!  
}
```

buggy_copyarray.c

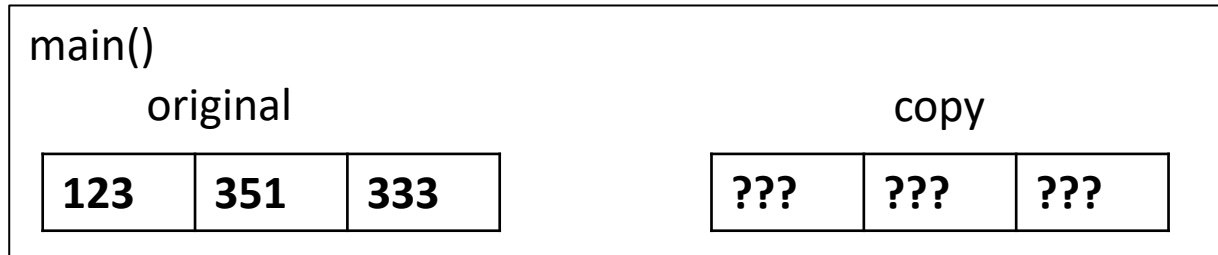
Solution: Output Parameter

- ❖ 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

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

`copyarray.c`

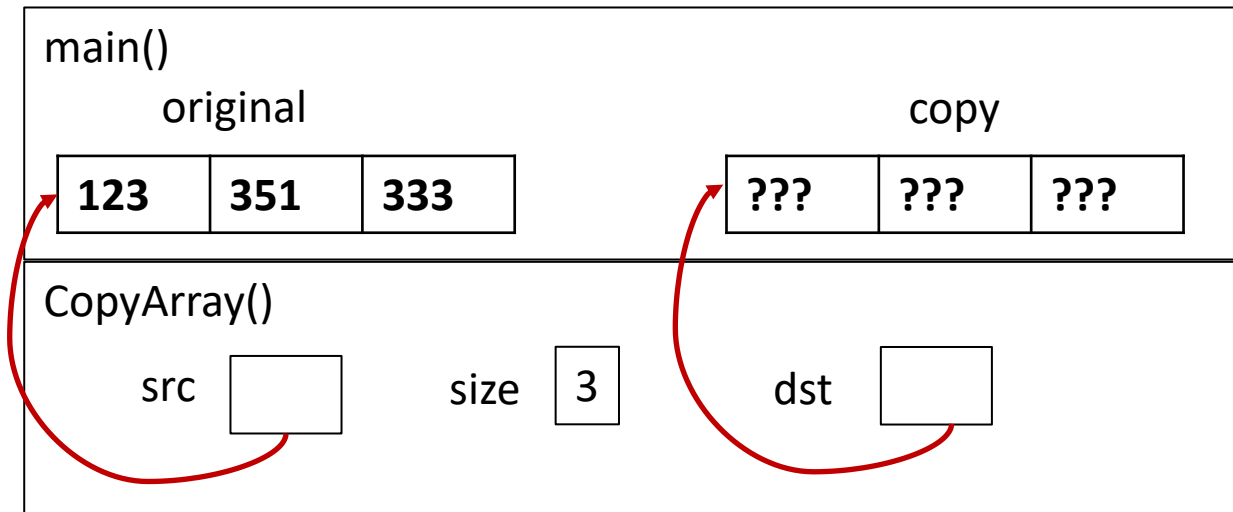
Array Memory Diagram (1/3)



```
int main(){
    int original[] = {123, 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 (2/3)

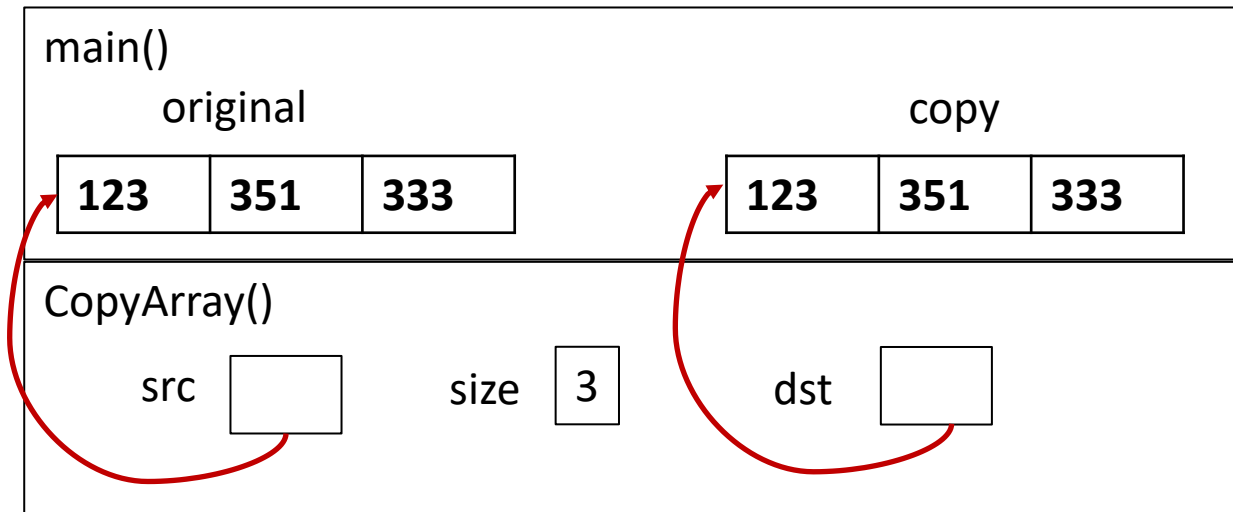


```
int main(){
    int original[] = {123, 351, 333};
    int copy[3];
    CopyArray(original, copy, 3);
}

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 (3/3)



```
int main(){
    int original[] = {123, 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`!

Output Parameters

- ❖ Output parameters are common in library functions
 - `long int strtol(char* str, char** endptr, int base);`
 - `int sscanf(char* str, char* format, ...);`

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

// converts "333 rocks" into long - p_end is conversion end
num = (int) strtol(str1, &p_end, 10);

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

outparam.c

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
- (will need to either wait for Lecture 4 or review malloc/free on your own)