



[pollev.com/cse333](https://pollev.com/cse333)

## 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.**



[pollev.com/cse333](https://pollev.com/cse333)

## About how long did Exercise 1 take you?

- A. [0, 2) hours
- B. [2, 4) hours
- C. [4, 6) hours
- D. [6, 8) hours
- E. 8+ Hours
- F. I didn't submit / I prefer not to say

# C Data, Parameters

## CSE 333 Summer 2023

**Instructor:** Timmy Yang

**Teaching Assistants:**

Jennifer Xu

Leanna Nguyen

Pedro Amarante

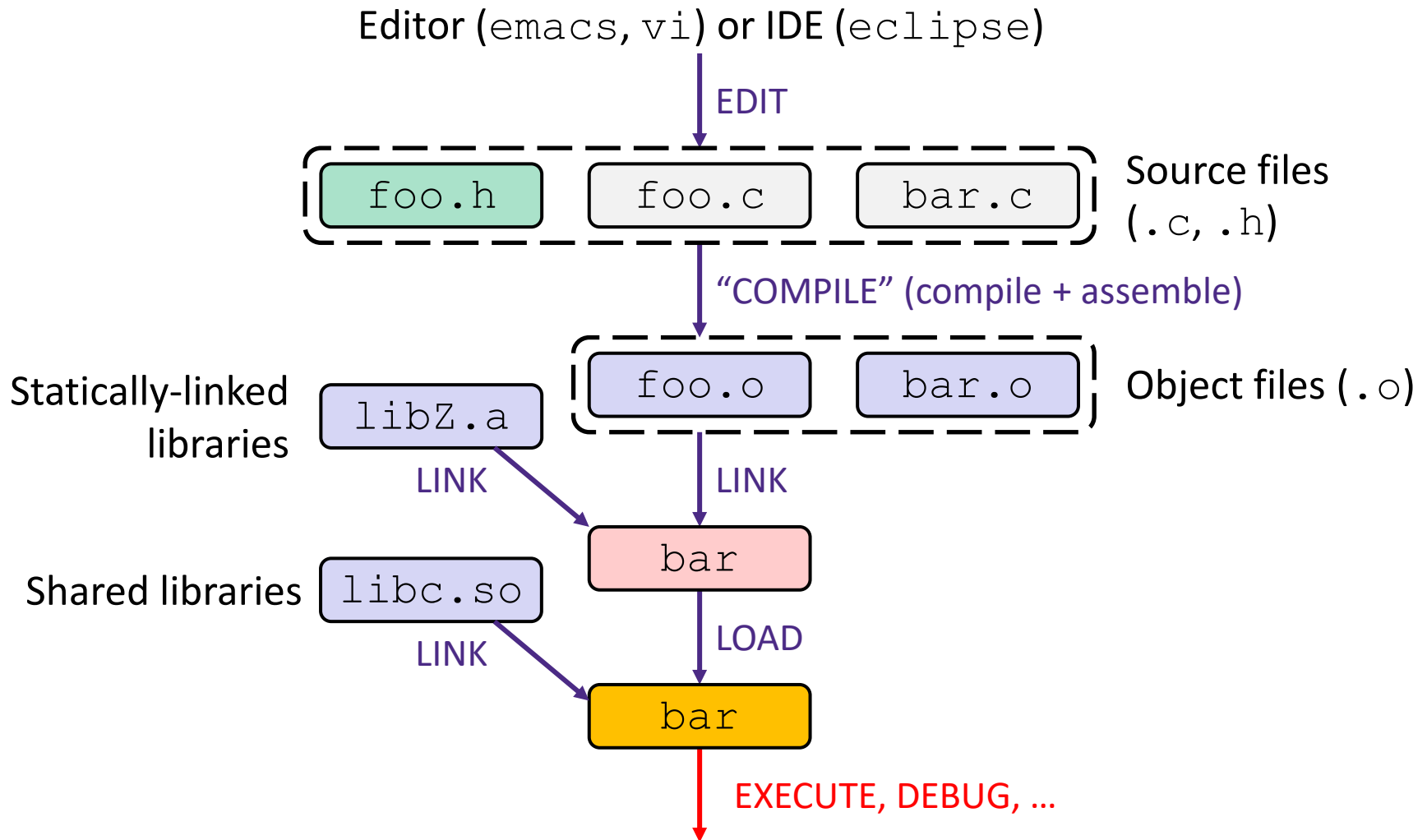
Sara Deutscher

Tanmay Shah

# Relevant Course Information

- ❖ Pre-quarter survey due tonight, 11:59 pm (Canvas)
- ❖ Exercise 1 due earlier today, 1:00pm
  - Submission via Gradescope (contact us if you don't have access)
  - Make sure that you are testing on the CSE Linux environment
  - Sample solution will be posted tomorrow afternoon
- ❖ Homework 0 due Monday night, 11:59 pm
  - Logistics and infrastructure for projects
    - `cpplint` and `valgrind` are useful for exercises, too
  - Should have set up an SSH key and cloned GitLab repo by now
    - Do this ASAP so we have time to fix things if necessary
  - We will submit to Gradescope from your repo for you

# 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)

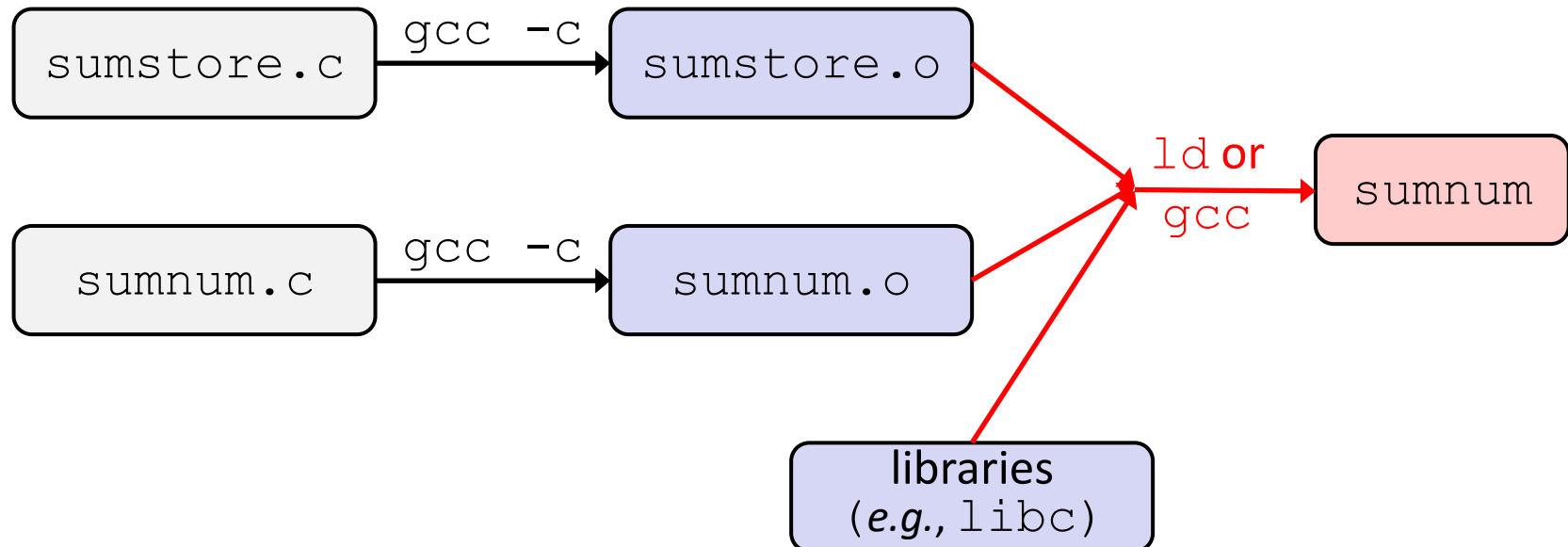
```
#include <stdio.h>  
#include <stdlib.h>  
  
void SumStore(int x, int y, int* dest);  
  
int main(int argc, char** argv) {  
    int z, x = 351, y = 333;  
    SumStore(x, y, &z);  
    printf("%d + %d = %d\n", x, y, z);  
    return EXIT_SUCCESS;  
}
```

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





# Poll Everywhere

[pollev.com/cse333](https://pollev.com/cse333)

## Which of the following statements is 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...



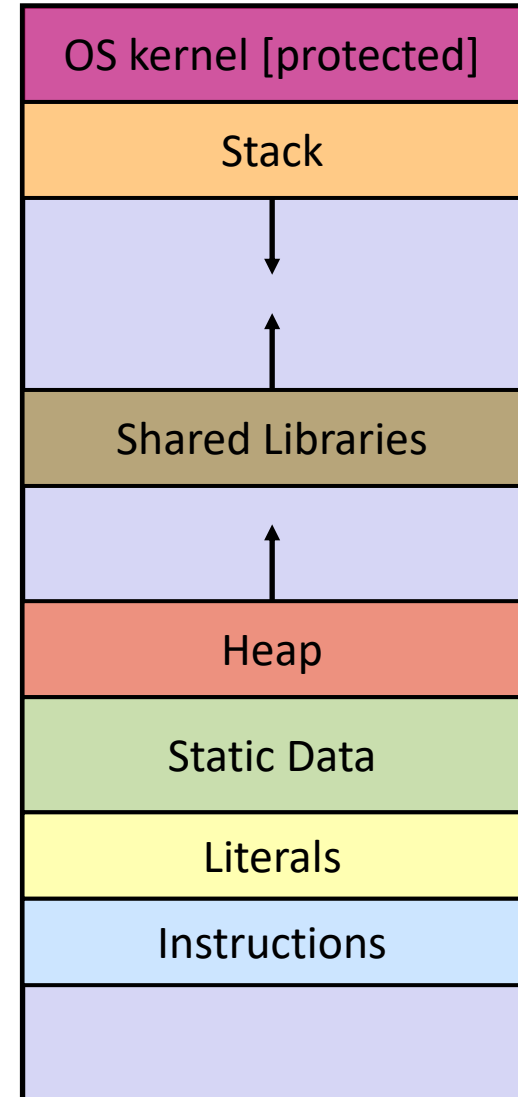
# Lecture Outline

- ❖ **C Data Considerations**
  - **Memory, Integers**
  - **Arrays and Pointers Review**
- ❖ **C Parameters**
  - **Arrays and Pointers as Parameters**

# 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**

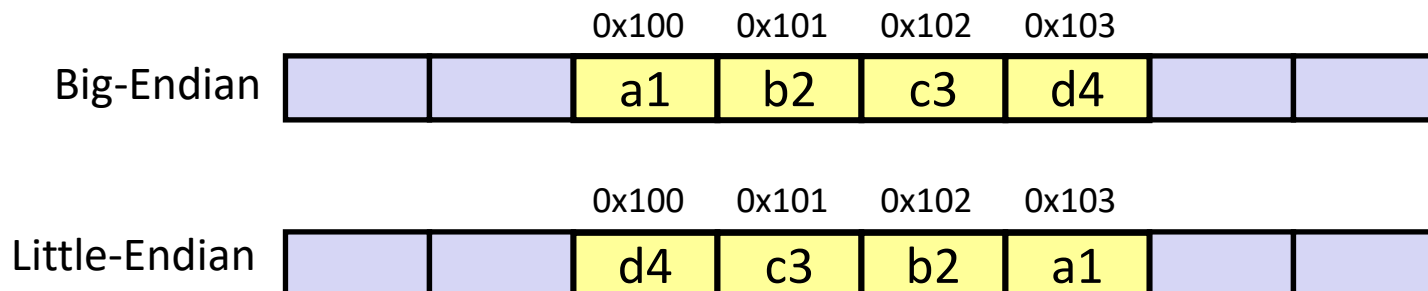
0xFF...FF



0x00...00

# Endianness

- ❖ Memory is byte-addressed, so endianness determines what ordering that multi-byte data gets read and stored *in memory*
  - **Big-endian**: Least significant byte has *highest* address
  - **Little-endian**: Least significant byte has *lowest* address
- ❖ **Example**: 4-byte data 0xa1b2c3d4 at address 0x100



# 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  |
|--------------------|--------|--------|---------|
| <b>char</b>        | 1      | 1      | %c      |
| short int          | 2      | 2      | %hd     |
| unsigned short int | 2      | 2      | %hu     |
| <b>int</b>         | 4      | 4      | %d / %i |
| unsigned int       | 4      | 4      | %u      |
| long int           | 4      | 8      | %ld     |
| long long int      | 8      | 8      | %lld    |
| <b>float</b>       | 4      | 4      | %f      |
| <b>double</b>      | 8      | 8      | %lf     |
| long double        | 12     | 16     | %Lf     |
| <b>pointer</b>     | 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) {
```

# 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;`

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

`int *p1, p2;` not the same as `int *p1, *p2;`

- Instead, use:

```
int *p1;
int *p2;
```

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

- Access the memory referred to by a pointer

# Pointer Arithmetic

- ❖ Pointers are *typed*
  - Tells the compiler the size of the data you are pointing to
  - Exception: `void*` is a generic pointer (*i.e.*, a placeholder)
- ❖ Pointer arithmetic is scaled by `sizeof (*p)`
  - Works nicely for arrays
  - Does not work on `void*`, since `void` doesn't have a size!
    - Not allowed, though confusingly GCC allows it as an extension 😞
- ❖ Valid pointer arithmetic:
  - Add/subtract an integer to/from a pointer
  - Subtract two pointers (within stack frame or malloc block)
  - Compare pointers (`<`, `<=`, `==`, `!=`, `>`, `>=`), including `NULL`
  - ... but plenty of valid-but-inadvisable operations, too



# Poll Everywhere

[pollev.com/cse333](http://pollev.com/cse333)

At **this point** in the code, what values are stored in `arr[]`?

```

int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p; // pointer to a pointer
    *(*dp) += 1;
    p += 1;
    *(*dp) += 1;
    return EXIT_SUCCESS;
}

```

ptr\_poll.c

A. {2, 3, 4}

B. {3, 4, 5}

C. {2, 6, 4}

D. {2, 4, 5}

E. We're lost...

0x7fff...78

|        |   |
|--------|---|
| arr[2] | 4 |
|--------|---|

0x7fff...74

|        |   |
|--------|---|
| arr[1] | 3 |
|--------|---|

0x7fff...70

|        |   |
|--------|---|
| arr[0] | 2 |
|--------|---|

0x7fff...68

|   |             |
|---|-------------|
| p | 0x7fff...74 |
|---|-------------|

0x7fff...60

|    |             |
|----|-------------|
| dp | 0x7fff...68 |
|----|-------------|



# Arrays

- ❖ Definition: `type name [size]` allocates  $size * sizeof(type)$  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

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

# Pointers and Arrays

- ❖ A pointer can point to an array element
  - You can use array indexing notation on pointers
    - `ptr[i]` is `*(ptr+i)` with pointer arithmetic – reference the data `i` elements forward from `ptr`
  - An array name's value is the beginning address of the array
    - *Like* a pointer to the first element of array, but can't change

```
int a[] = {10, 20, 30, 40, 50};
int* p1 = &a[3]; // refers to a's 4th element
int* p2 = &a[0]; // refers to a's 1st element
int* p3 = a;    // refers to a's 1st element

*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;    // final: 200, 400, 500, 100, 300
```

# Lecture Outline

- ❖ C Data Considerations
  - Memory, Integers
  - Arrays and Pointers Review
- ❖ **C Parameters**
  - **Arrays and Pointers as 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)

# Faking Call-By-Reference in C

- ❖ Can use pointers to *approximate* call-by-reference
  - Callee still receives a **copy** of the pointer (*i.e.*, call-by-value), but it can modify something in the caller's scope by dereferencing the pointer parameter

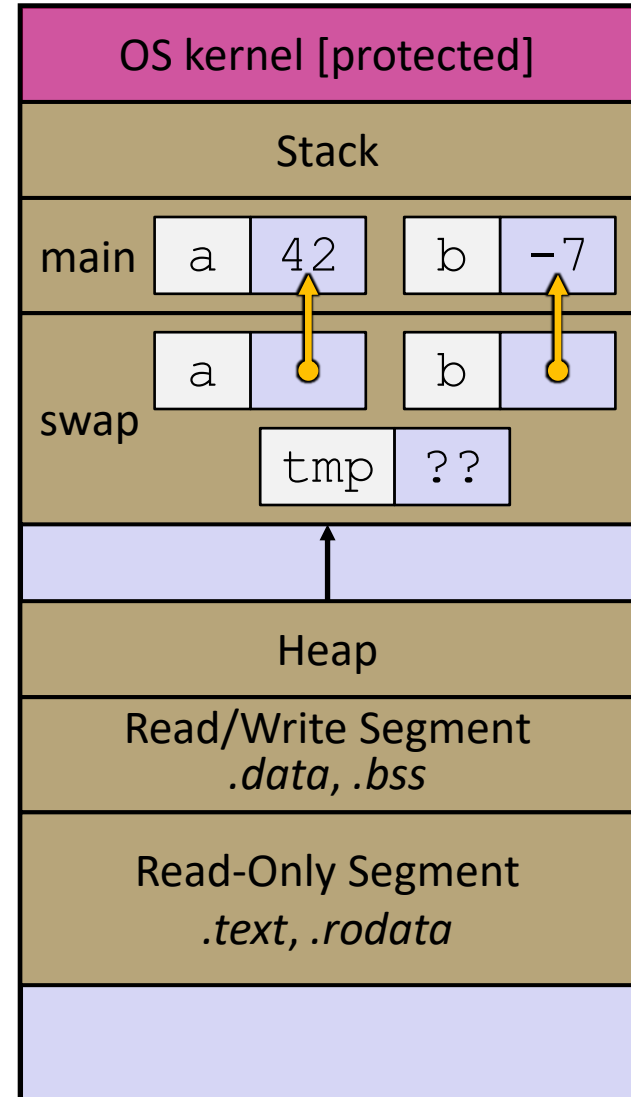
```
void Swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = -7;
    Swap(&a, &b);
    ...
}
```

# Fixed Swap

## swap.c

```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



# 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

```
// sums all elements of the array a
int SumAll(int a[]);

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

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



# Solution 1: Declare Array Size

```
// sums all elements of the array a
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 EXIT_SUCCESS;
}

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

```
// sums all elements of the array a
int SumAll(int a[], int size);

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 EXIT_SUCCESS;
}

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

- ❖ Standard idiom in C programs!

arraysum.c

# 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

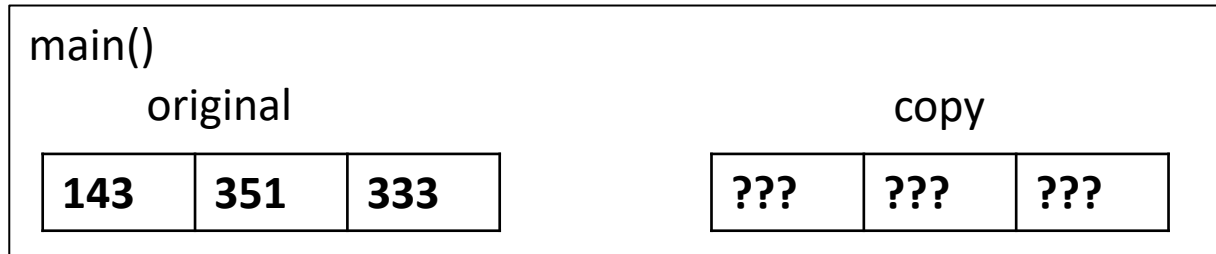
# Solution: Output Parameter

- ❖ 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`

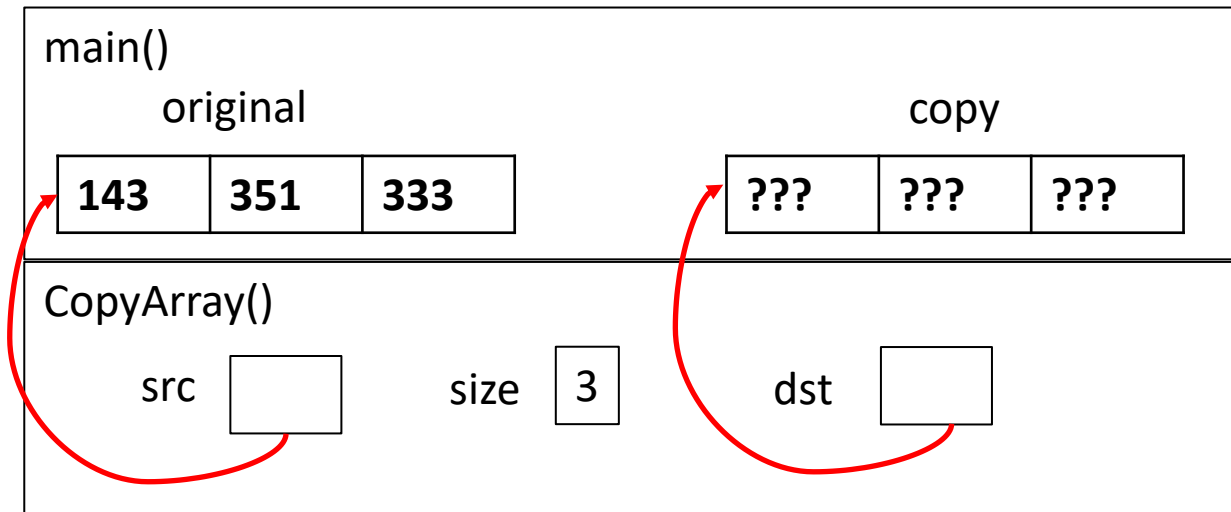
# Array Memory Diagram



```
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

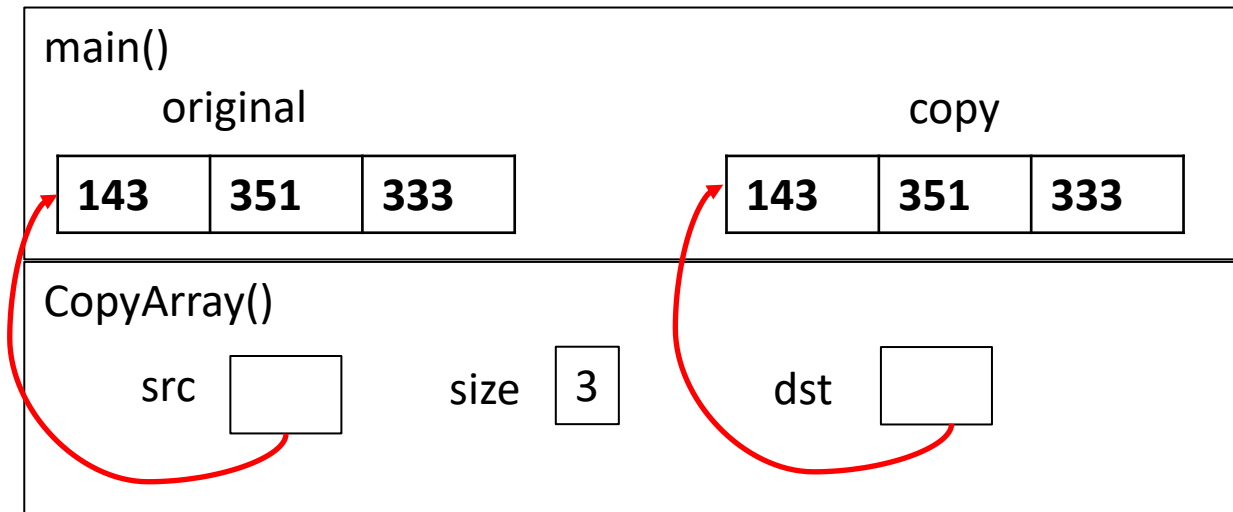


```
int main() {  
    int original[] = {143, 351, 333};  
    int copy[3];  
    CopyArray(original, copy, 3);  
}  
  
void CopyArray(int src[], int dst[], int size) {  
    for (int i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
}
```

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`!

# 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

- ❖ Use a box-and-arrow diagram for the following program and explain what it prints out:

```
#include <stdio.h>

int foo(int* bar, int** baz) {
    *bar = 5;
    *(bar+1) = 6;
    *baz = bar + 2;
    return *((*baz)+1);
}

int main(int argc, char** argv) {
    int arr[4] = {1, 2, 3, 4};
    int* ptr;

    arr[0] = foo(&arr[0], &ptr);
    printf("%d %d %d %d %d\n",
           arr[0], arr[1], arr[2], arr[3], *ptr);
    return 0;
}
```

# Extra Exercise #3

- ❖ Write a program that determines and prints out whether the computer it is running on is little-endian or big-endian.
  - Hint: `show_bytes.c` from 351 Lecture 3