



[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

# Pointers, Pointers, Pointers...

## CSE 333 Spring 2021

**Instructor:** Chris Thachuk

**Teaching Assistants:**

Arpad (John) Depaszthory

Ian Hsiao

Logan Gnanapragasam

Mengqi (Frank) Chen

Angela Xu

Khang Vinh Phan

Maruchi Kim

Cosmo Wang

# Administrivia (1/2)

- ❖ Exercise 3 out and due Wednesday (Oct 6) @ 10am
  
- ❖ Exercise grading
  - Autograder scores visible immediately after deadline; sample solutions released same day as deadline
  - Grades (out of 7):
    - Autograder: Compilation (1), Correctness (3), Linter (1), Valgrind (1)
    - Manual: Other Style (1)
  - Style things to watch for:
    - FOLLOW THE SPEC (especially the Style Guide section)
    - Check the Google C++ Style Guide
    - Make a judgment call and document
  - Keep style tips in mind, as you will need to use them in hw

# Administrivia (2/2)

- ❖ Homework 0 due tonight @ 11pm
  - Logistics and infrastructure for projects
    - `clint` and `valgrind` are useful for exercises, too
  
- ❖ Homework 1 out later today (due Thu, Oct 14)
  - Linked list and hash table implementations in C
  - Get starter code using `git pull` in your course repo
    - Might have “merge conflict” if your local copy has unpushed changes
      - If git drops you into `vi(m)`, `:q` to quit or `:wq` if you want to save changes
  
- ❖ Working on a better solution for live questions in class

# Box-and-Arrow Diagrams

boxarrow.c

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

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address

name	value
------	-------

# Box-and-Arrow Diagrams

boxarrow.c

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

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address	<b>name</b>	value
---------	-------------	-------

&x	<b>x</b>	value
&arr[2]	<b>arr[2]</b>	value
&arr[1]	<b>arr[1]</b>	value
&arr[0]	<b>arr[0]</b>	value
&p	<b>p</b>	value

stack frame for main()

# Box-and-Arrow Diagrams

boxarrow.c

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

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address	name	value
	<b>x</b>	1
<b>&amp;arr[2]</b>	<b>arr[2]</b>	4
<b>&amp;arr[1]</b>	<b>arr[1]</b>	3
<b>&amp;arr[0]</b>	<b>arr[0]</b>	2
<b>&amp;p</b>	<b>p</b>	<b>&amp;arr[1]</b>

# Box-and-Arrow Diagrams

boxarrow.c

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

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address	name	value
0x7fff...4c	<b>x</b>	1
0x7fff...48	<b>arr[2]</b>	4
0x7fff...44	<b>arr[1]</b>	3
0x7fff...40	<b>arr[0]</b>	2
0x7fff...38	<b>p</b>	0x7fff...44

# Lecture Outline

- ❖ **Pointer Arithmetic**
- ❖ Pointers as Parameters
- ❖ Pointers and Arrays
- ❖ Function Pointers

# 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;
}

```

boxarrow2.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
----	-------------

# Practice Solution

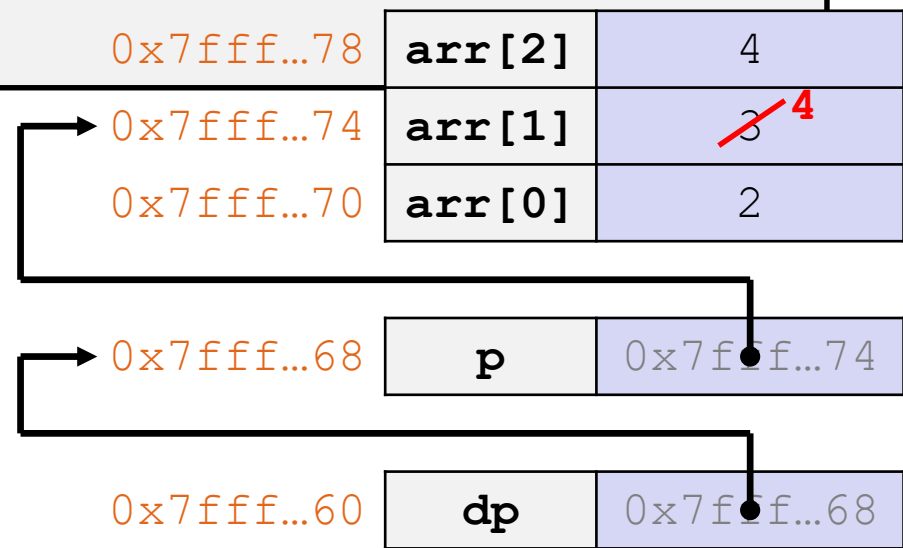
Note: arrow points to *next* instruction to be executed.  
 boxarrow2.c

```
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;
}
```

address	name	value
---------	------	-------



# Practice Solution

Note: arrow points to *next* instruction to be executed.

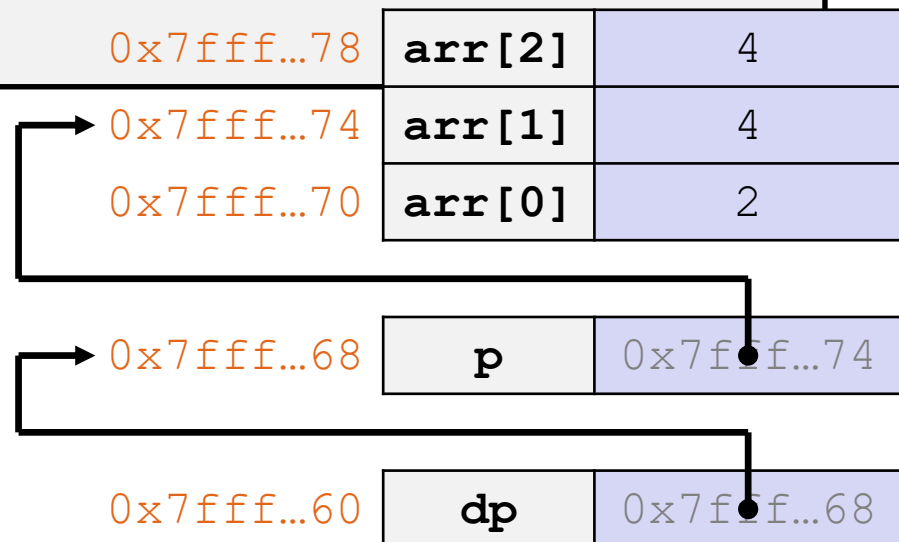
boxarrow2.c

```
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;
}
```

address	name	value
---------	------	-------



# Practice Solution

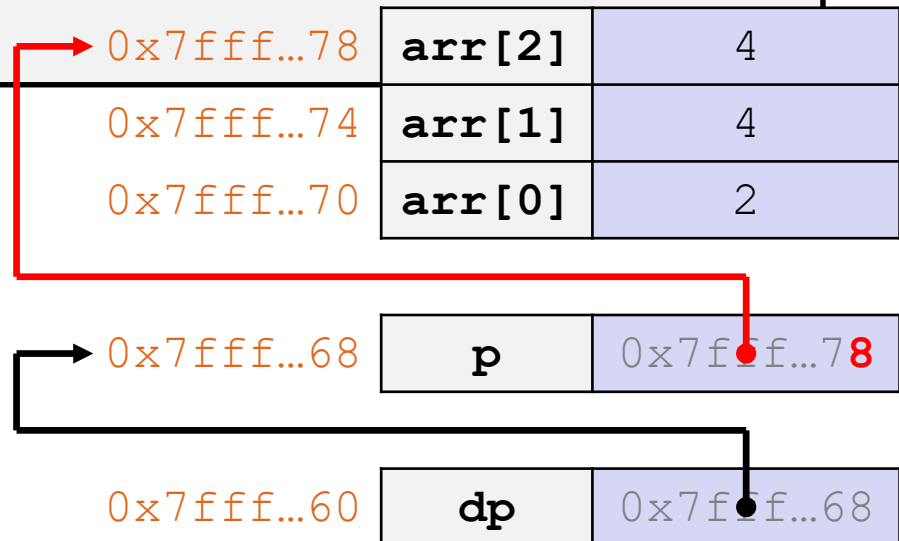
Note: arrow points to *next* instruction to be executed.  
 boxarrow2.c

```
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;
}
```

address	name	value
---------	------	-------



# Practice Solution

Note: arrow points to *next* instruction to be executed.

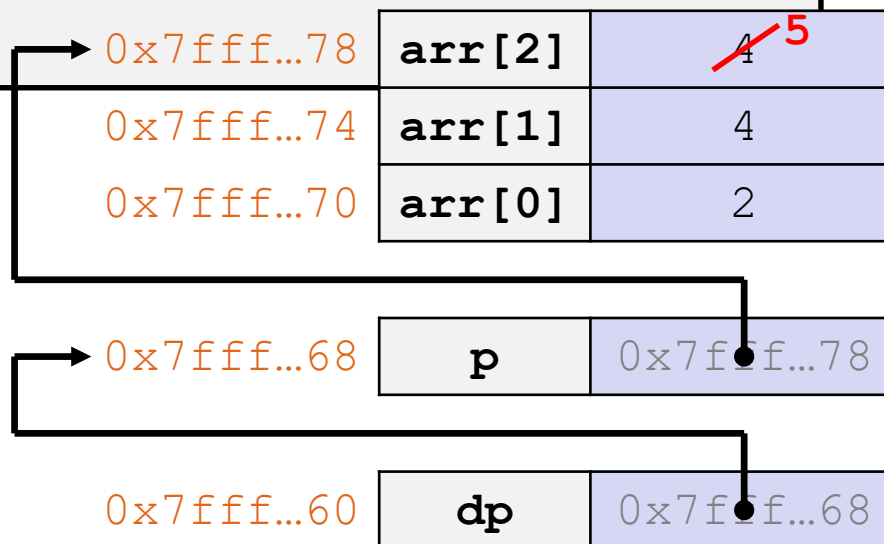
boxarrow2.c

```
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;

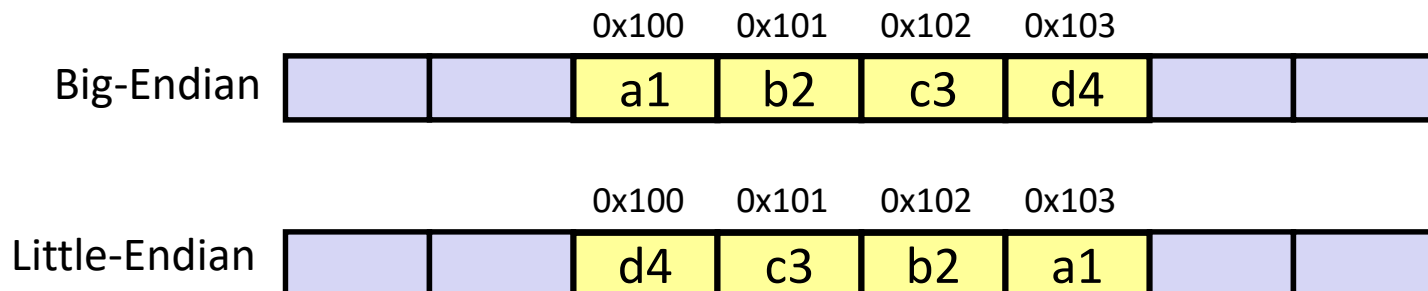
    return EXIT_SUCCESS;
}
```

address	name	value
---------	------	-------



# 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



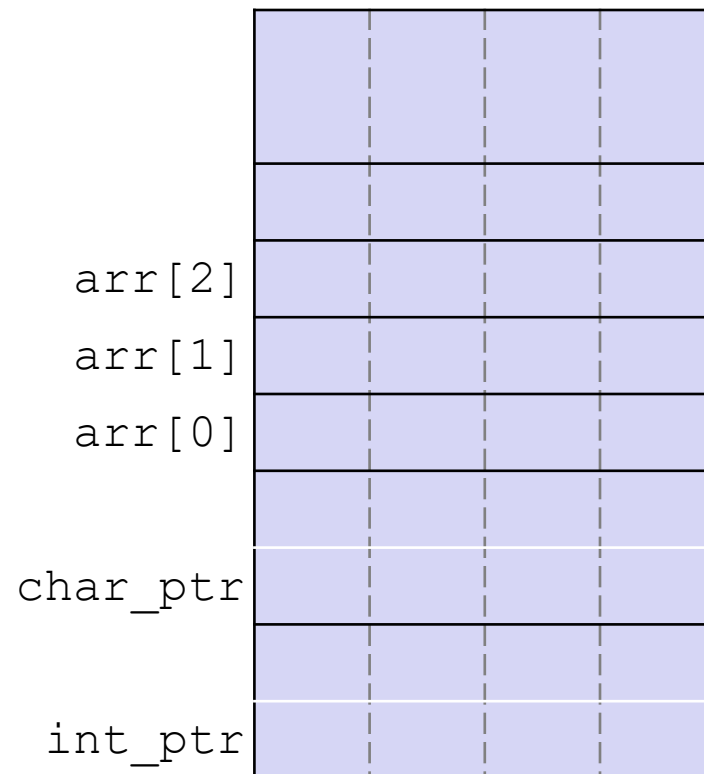
# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {  
→ int arr[3] = {1, 2, 3};  
  int* int_ptr = &arr[0];  
  char* char_ptr = (char*) int_ptr;  
  
  int_ptr += 1;  
  int_ptr += 2; // uh oh  
  
  char_ptr += 1;  
  char_ptr += 2;  
  
  return EXIT_SUCCESS;  
}
```

pointerarithmetic.c

**Stack**  
(assume x86-64)



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

**Stack**  
(assume x86-64)

arr[2]	03	00	00	00
arr[1]	02	00	00	00
arr[0]	01	00	00	00
char_ptr				
int_ptr				

# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

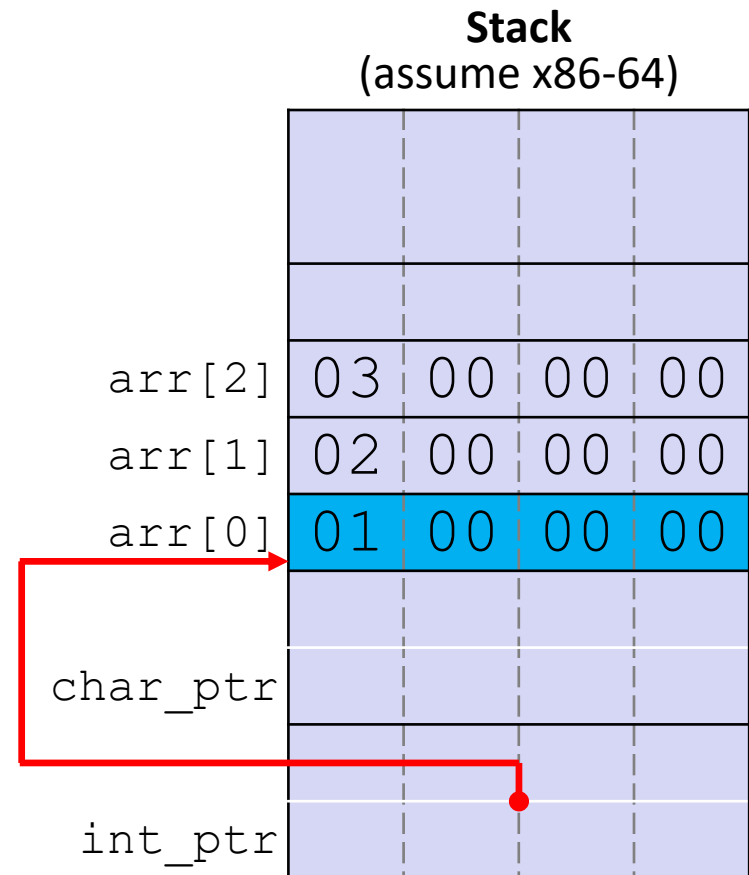
```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

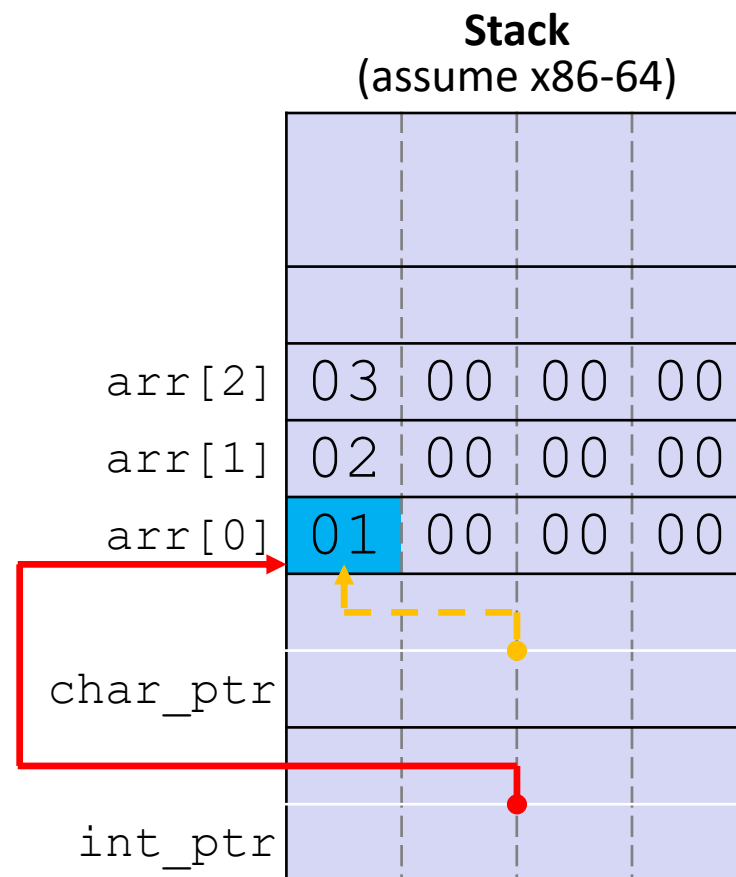


# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {  
    int arr[3] = {1, 2, 3};  
    int* int_ptr = &arr[0];  
    char* char_ptr = (char*) int_ptr;  
  
    int_ptr += 1;  
    int_ptr += 2; // uh oh  
  
    char_ptr += 1;  
    char_ptr += 2;  
  
    return EXIT_SUCCESS;  
}
```

pointerarithmetic.c



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

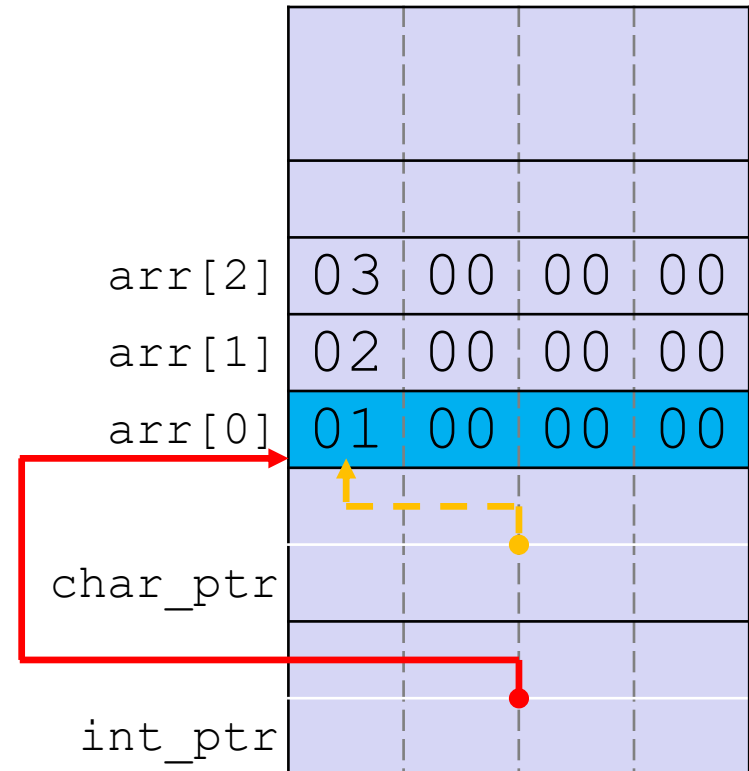
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

```
int_ptr: 0x0x7fffffffde010
*int_ptr: 1
```

Stack  
(assume x86-64)



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

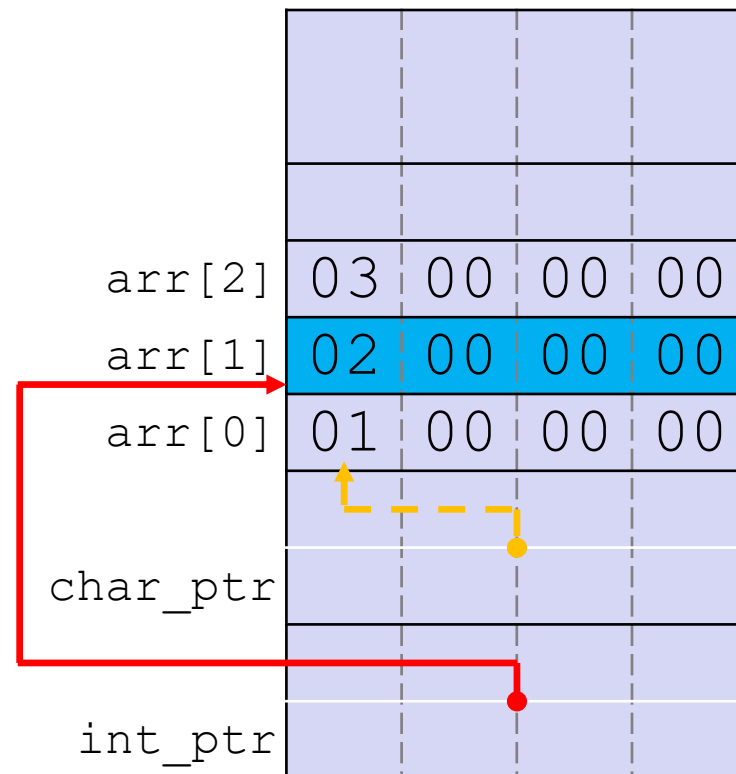
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

```
int_ptr: 0x0x7fffffffde014
*int_ptr: 2
```

Stack  
(assume x86-64)



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

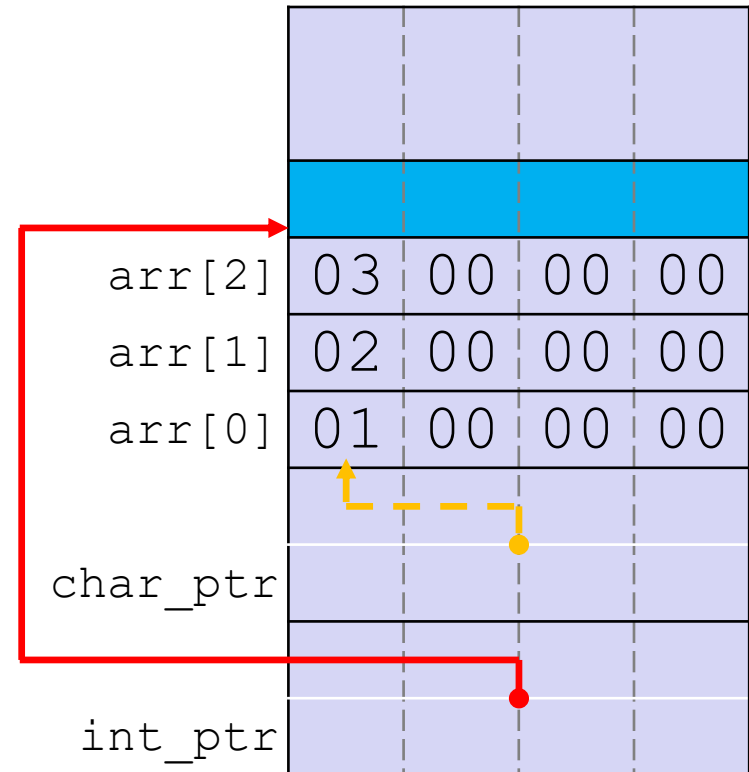
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

```
int_ptr: 0x0x7fffffffde01c
*int_ptr: ???
```

Stack  
(assume x86-64)



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

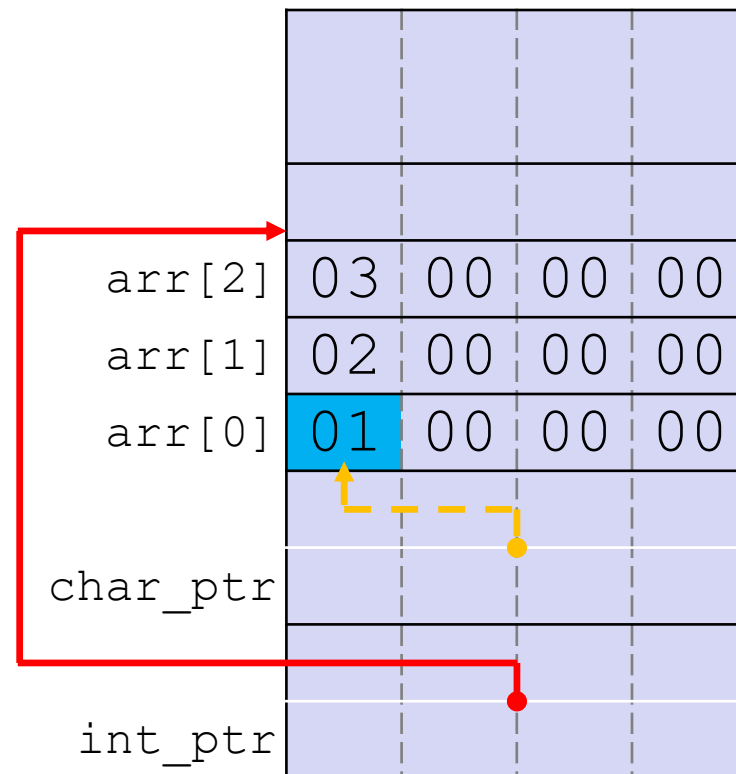
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

```
char_ptr: 0x0x7fffffffde010
*char_ptr: 1
```

Stack  
(assume x86-64)



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

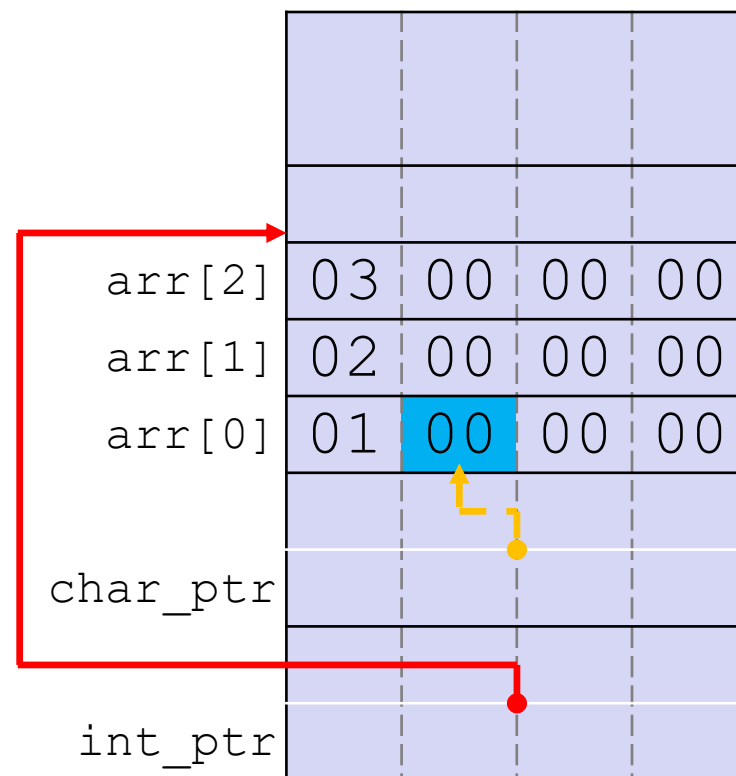
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

```
char_ptr: 0x0x7fffffffde011
*char_ptr: 0
```

Stack  
(assume x86-64)



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

    int_ptr += 1;
    int_ptr += 2; // uh oh

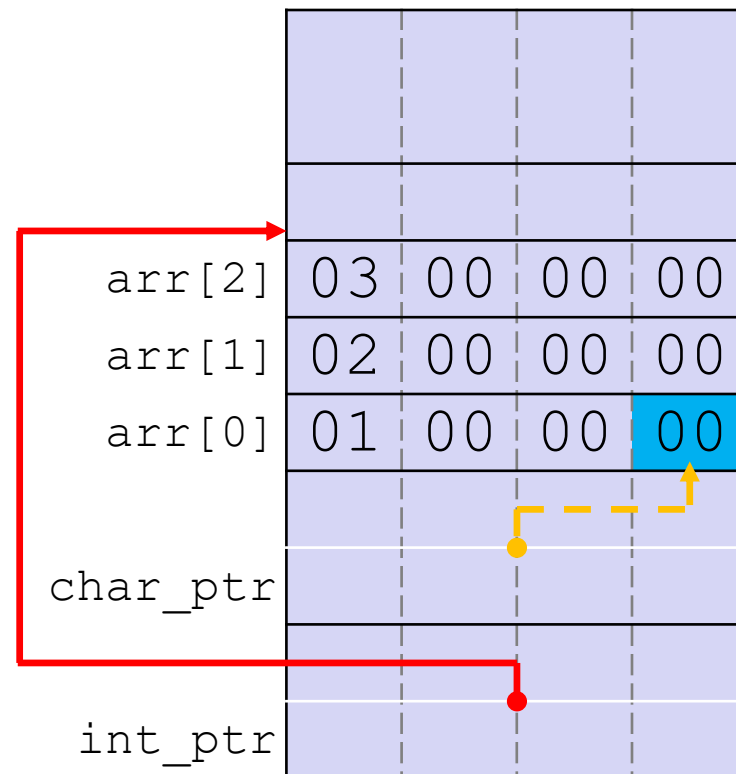
    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

```
char_ptr: 0x0x7fffffffde013
*char_ptr: 0
```

Stack  
(assume x86-64)



# Lecture Outline

- ❖ Pointer Arithmetic
- ❖ **Pointers as Parameters**
- ❖ Pointers and Arrays
- ❖ Function Pointers

# C is Call-By-Value

- ❖ C (and Java) pass arguments by *value*
  - Callee receives a **local copy** of the argument
    - Register or Stack
  - If the callee modifies a parameter, the caller's copy *isn't* modified

```
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);  
    ...  
}
```

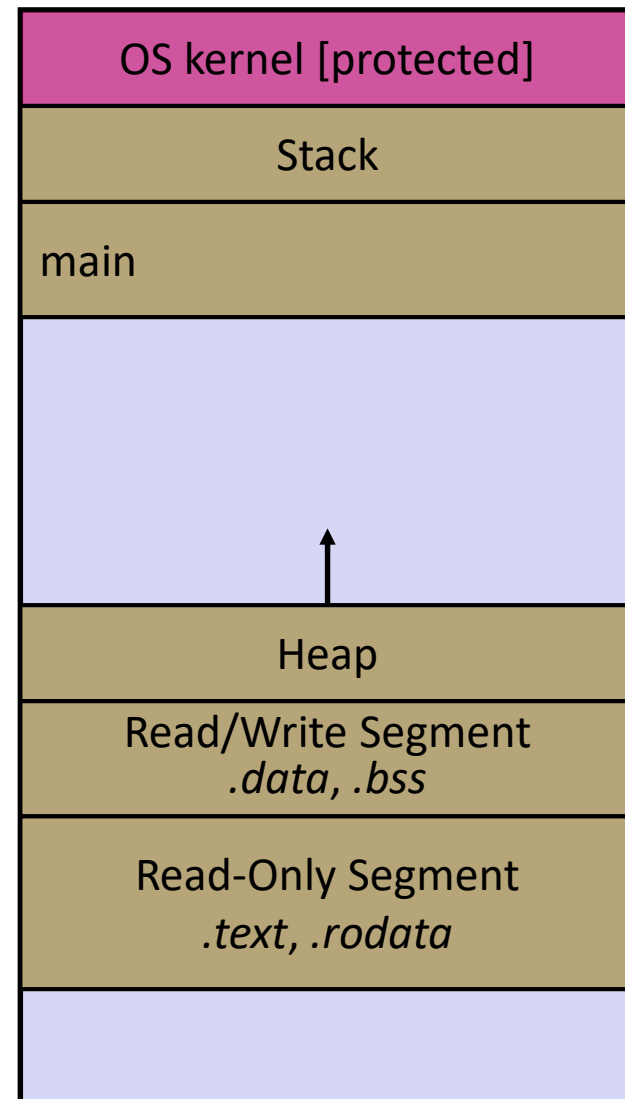
# Broken Swap

Note: Arrow points to *next* instruction.

## brokenswap.c

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

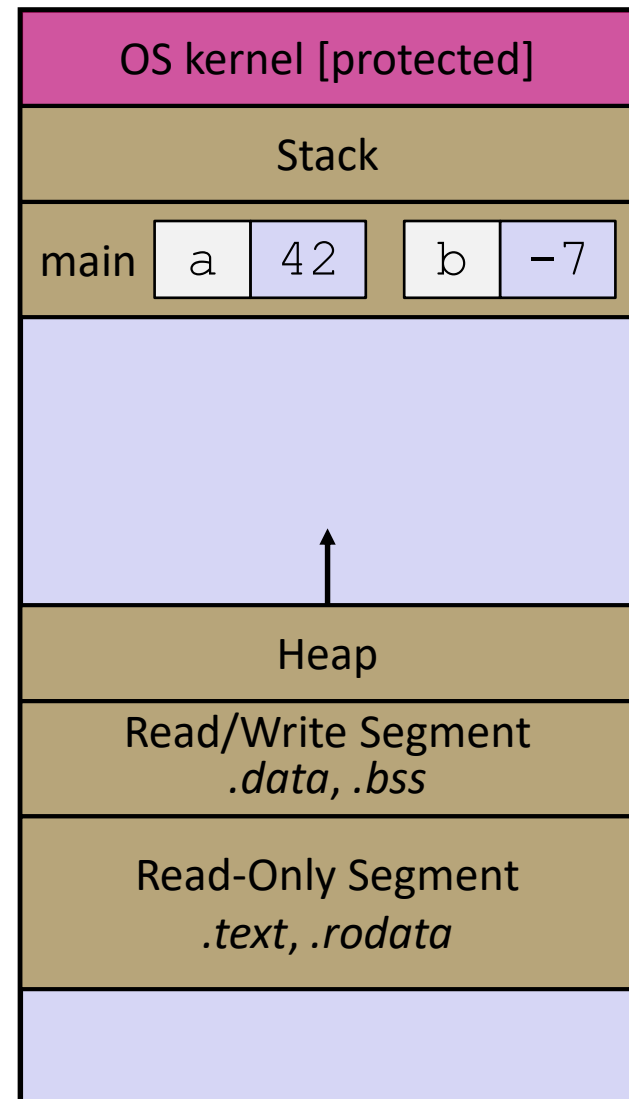
int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
}
```



# Broken Swap

## brokenswap.c

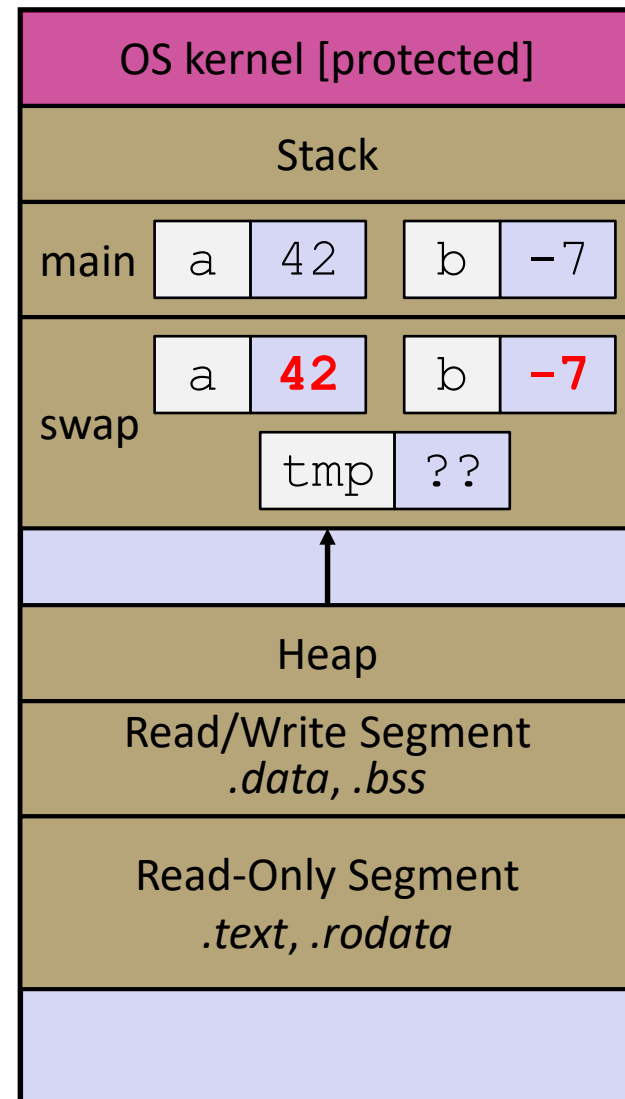
```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(a, b);  
    ...  
}
```



# Broken Swap

## brokenswap.c

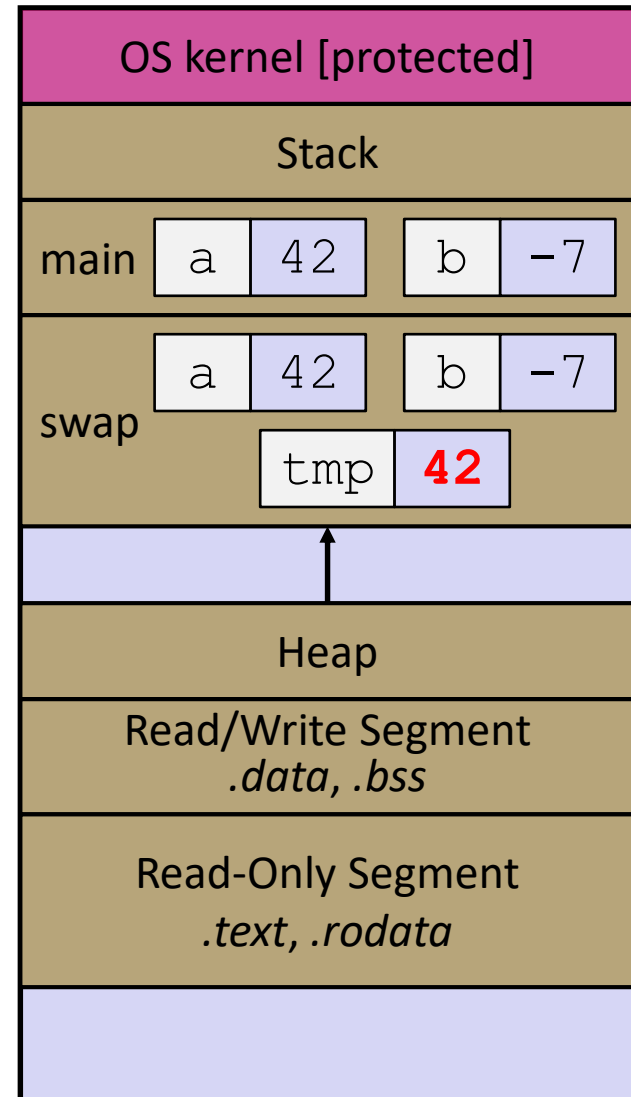
```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(a, b);  
    ...  
}
```



# Broken Swap

## brokenswap.c

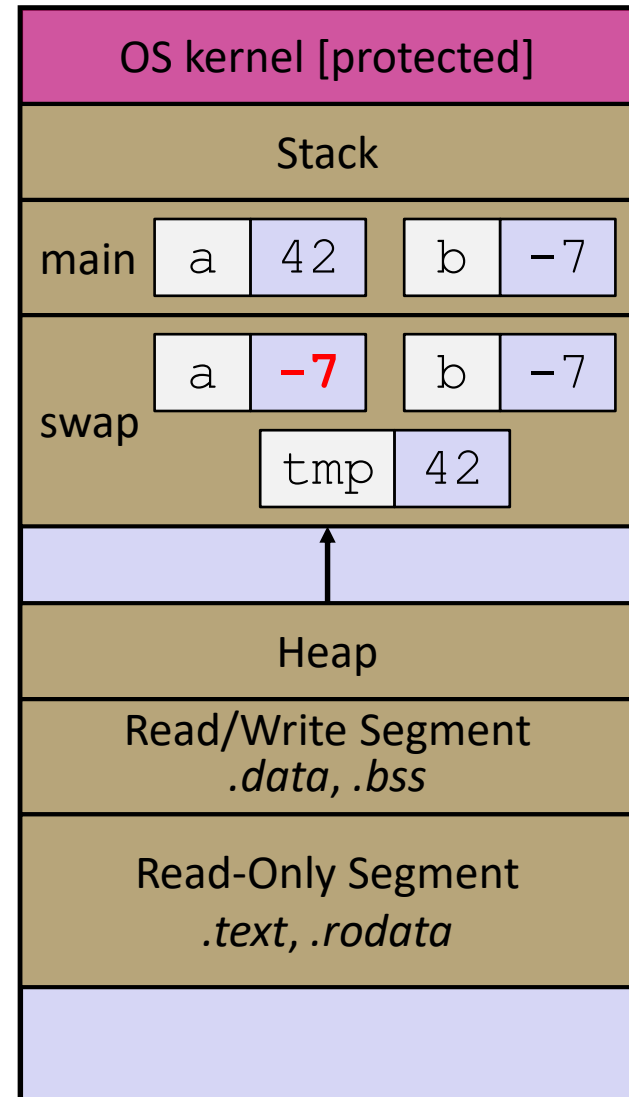
```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(a, b);  
    ...  
}
```



# Broken Swap

## brokenswap.c

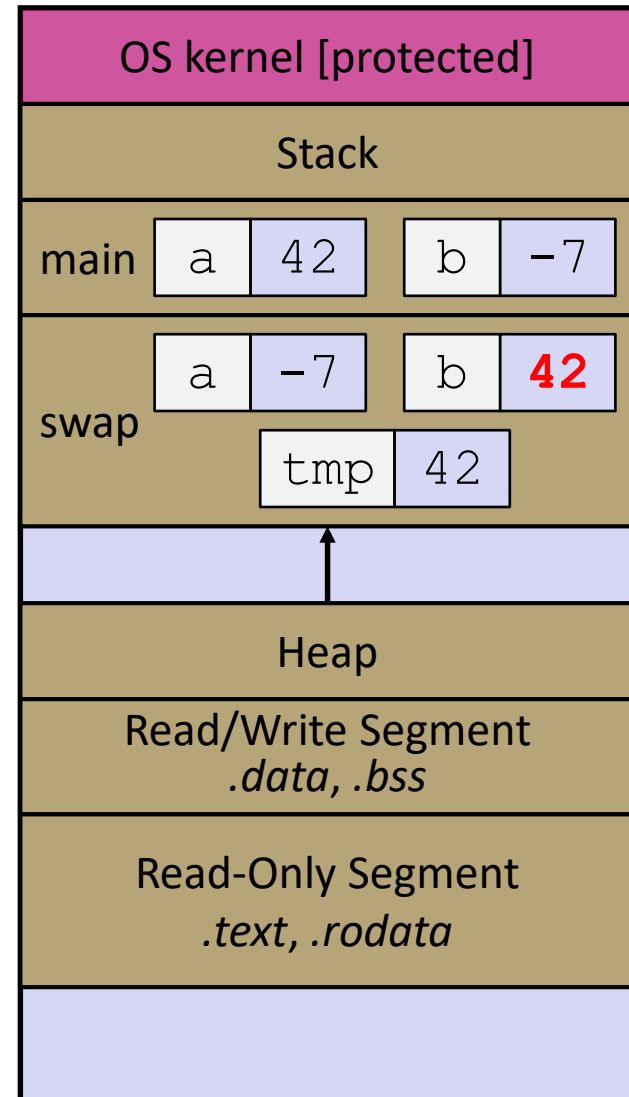
```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(a, b);  
    ...  
}
```



# Broken Swap

## brokenswap.c

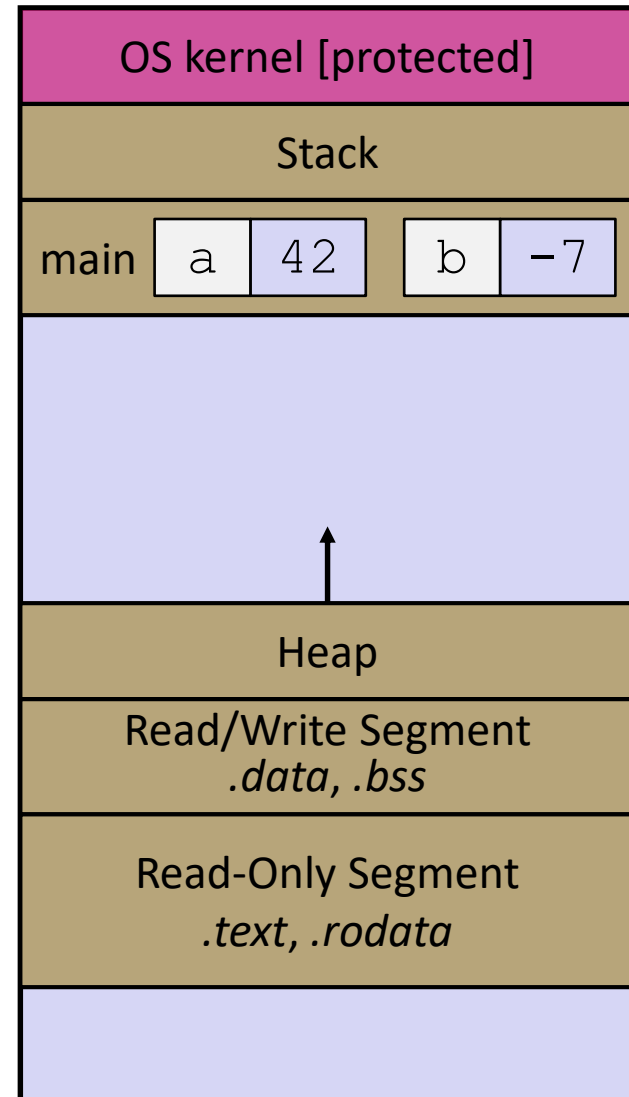

```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(a, b);  
    ...  
}
```



# Broken Swap

## brokenswap.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);  
    ...  
}
```



# 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);
    ...
}
```

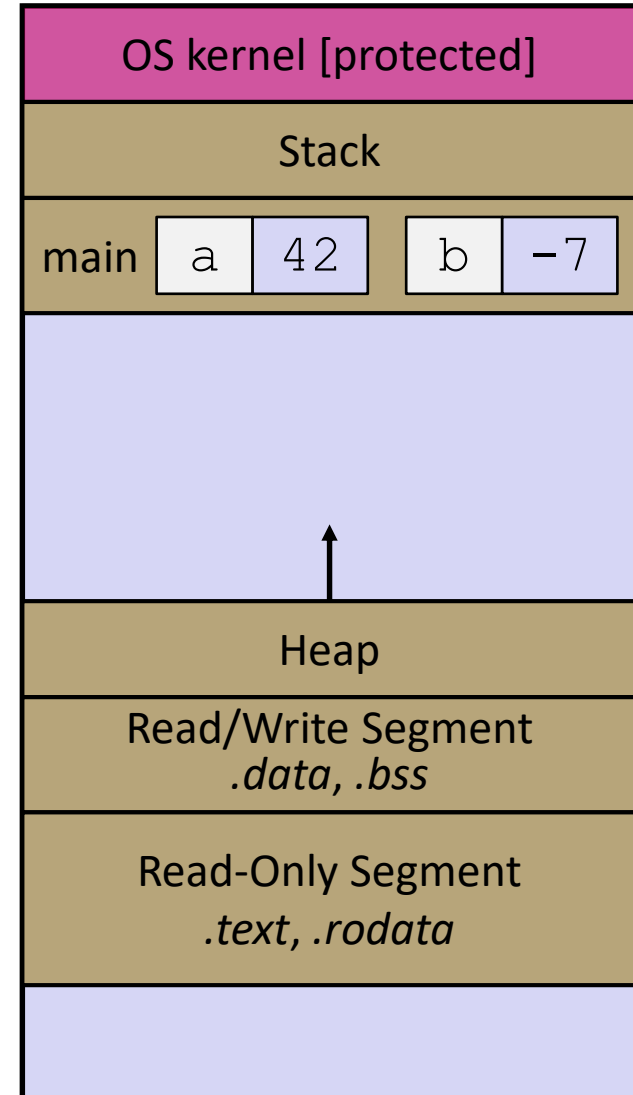
# Fixed Swap

Note: Arrow points to *next* instruction.

swap.c

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

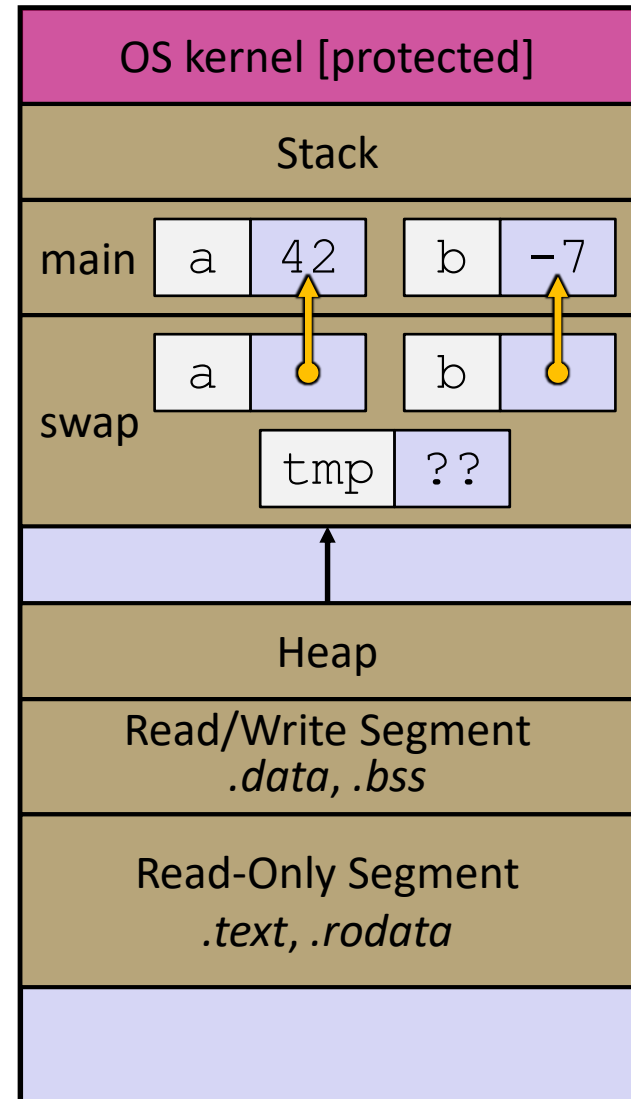
int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...
}
```



# Fixed Swap

## swap.c

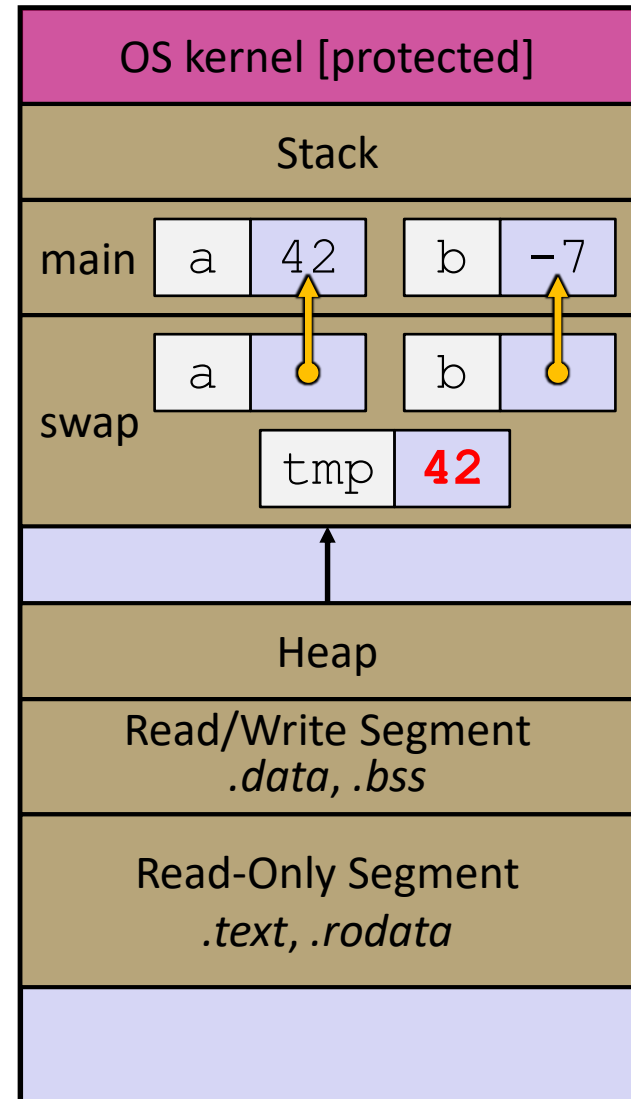
```
void swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(&a, &b);  
    ...  
}
```



# Fixed Swap

## swap.c

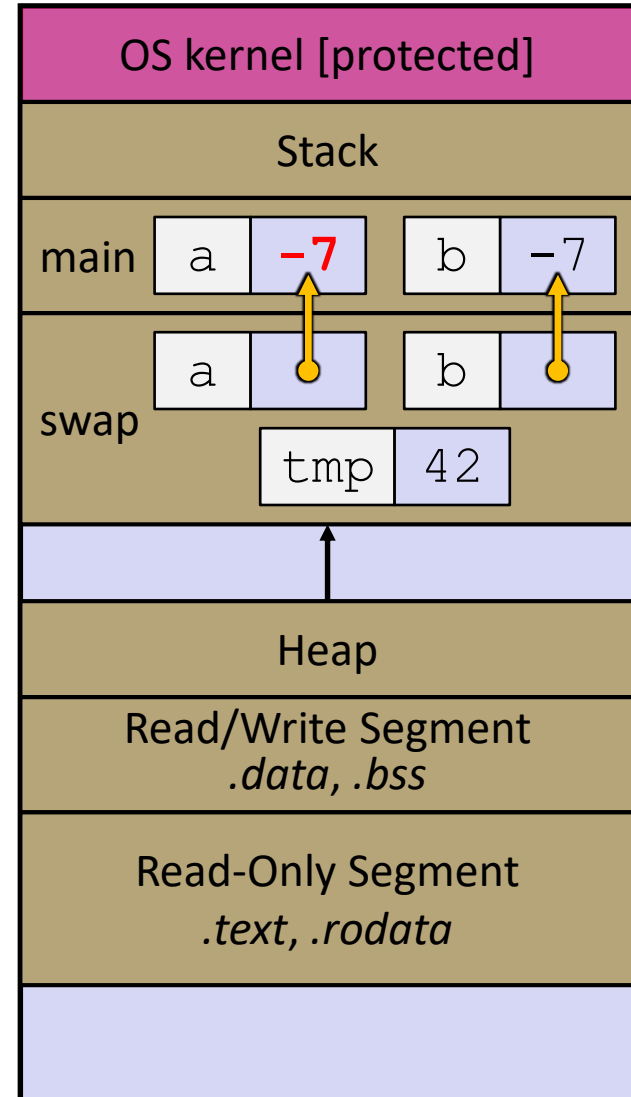
```
void swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(&a, &b);  
    ...  
}
```



# 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);  
    ...  
}
```



# Fixed Swap

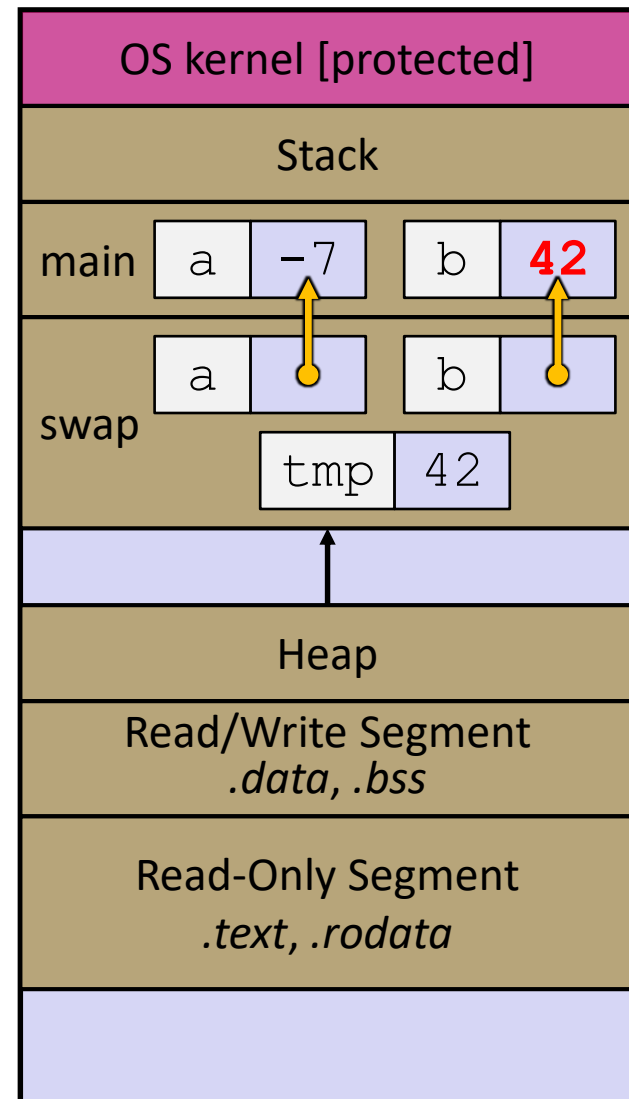
## swap.c

```

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);
    ...

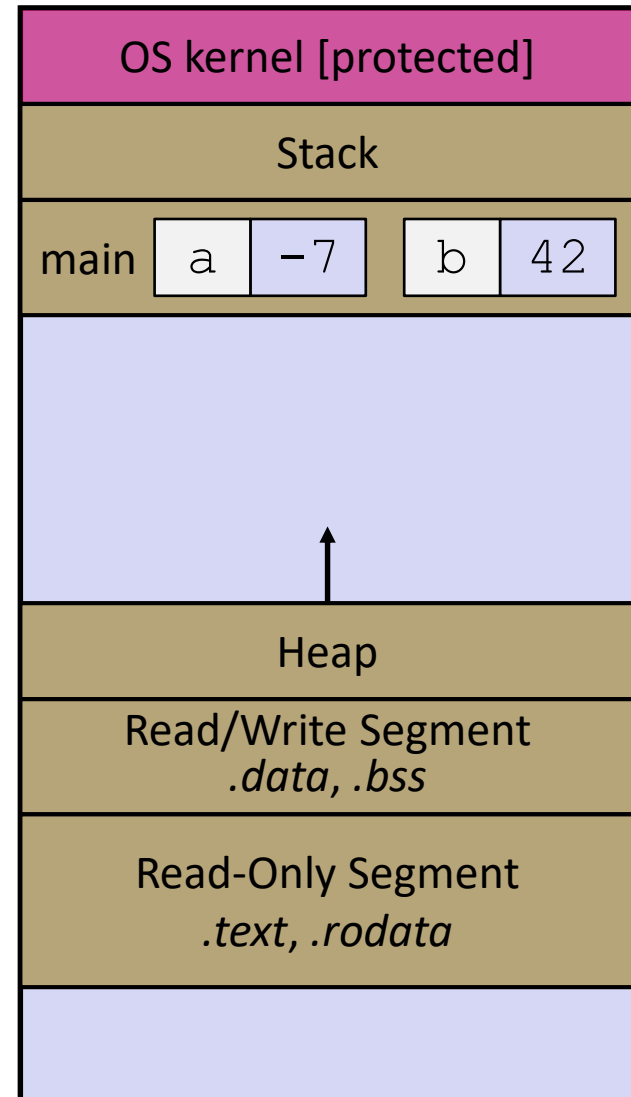
```



# 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);  
    ...  
}
```



# Output Parameters

**Warning:** Misuse of output parameters is *the* largest cause of errors in this course!

## ❖ Output parameter

- A pointer parameter used to store (via dereference) a function output *outside* of the function's stack frame
- Typically points to/modifies something in the **Caller's** scope
- Useful if you want to have multiple return values

## ❖ Setup and usage:

- 1) **Caller** creates space for the data (e.g., `type var;`)
- 2) **Caller** passes in a pointer to **Callee** (e.g., `&var`)
- 3) **Callee** takes in output parameter (e.g., `type* outparam`)
- 4) **Callee** uses parameter to set output (e.g., `*outparam = value;`)
- 5) **Caller** accesses output via modified data (e.g., `var`)

# Lecture Outline

- ❖ Pointer Arithmetic
- ❖ Pointers as Parameters
- ❖ **Pointers and Arrays**
- ❖ Function Pointers

# 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
```



# 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) {
```

# Lecture Outline

- ❖ Pointer Arithmetic
- ❖ Pointers as Parameters
- ❖ Pointers and Arrays
- ❖ **Function Pointers**

# Function Pointers

- ❖ Based on what you know about assembly, what is a function name, really?

- Can use pointers that store addresses of functions!

- ❖ Generic format:

```
returnType (* name) (type1, ..., typeN)
```

- Looks like a function prototype with extra \* in front of name
- Why are parentheses around (\* name) needed?

- ❖ Using the function:

```
(*name) (arg1, ..., argN)
```

- Calls the pointed-to function with the given arguments and return the return value

# Function Pointer Example

- ❖ `map()` performs operation on each element of an array

```
#define LEN 4

int negate(int num) {return -num;}
int square(int num) {return num*num;}

// perform operation pointed to on each array element
void map(int a[], int len, int (*op)(int n)) {
    for (int i = 0; i < len; i++) {
        a[i] = (*op)(a[i]); // dereference function pointer
    }
}

int main(int argc, char** argv) {
    int arr[LEN] = {-1, 0, 1, 2};
    int (*op)(int n); // function pointer called 'op'
    op = square; // function name returns addr (like array)
    map(arr, LEN, op);
    ...
}
```

**funcptr parameter** (points to `int (*op)(int n)`)

**funcptr dereference** (points to `(*op)(a[i])`)

**funcptr definition** (points to `int (*op)(int n);`)

**funcptr assignment** (points to `op = square;`)

# Function Pointer Example

- ❖ C allows you to omit & on a function name (like arrays) and omit \* when calling pointed-to function

```
#define LEN 4

int negate(int num) {return -num;}
int square(int num) {return num*num;}

// perform operation pointed to on each array element
void map(int a[], int len, int (* op)(int n)) {
    for (int i = 0; i < len; i++) {
        a[i] = op(a[i]); // dereference function pointer
    }
}

int main(int argc, char** argv) {
    int arr[LEN] = {-1, 0, 1, 2};
    map(arr, LEN, square);
    ...
}
```

implicit funcptr dereference (no \* needed)

no & needed for func ptr argument

# Extra Exercise #1

- ❖ 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 #2

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

# Extra Exercise #3

- ❖ Write a function that:
  - Arguments: [1] an array of ints and [2] an array length
  - Malloc's an `int*` array of the same element length
  - Initializes each element of the newly-allocated array to point to the corresponding element of the passed-in array
  - Returns a pointer to the newly-allocated array

# Extra Exercise #4

- ❖ Write a function that:
  - Accepts a function pointer and an integer as arguments
  - Invokes the pointed-to function with the integer as its argument