

pollev.com/cse333j

About how long did Exercises 0 and 1 take you? (two polls)

- 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

Systems Programming

Pointers

Instructors:

Justin Hsia

Amber Hu

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 (1/3)

- ❖ Exercise grading (see [Ed post #53](#))
 - Autograder scores visible immediately after deadline; sample solutions released same day as deadline
 - Grades (out of 8):
 - Correctness: Output (3)
 - Tools: Compilation (1), Valgrind (1)
 - Style: Linter (1), Other Style [manual] (2)
 - 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

Relevant Course Information (2/3)

- ❖ Pre-quarter survey (Canvas quiz) due tonight
- ❖ Exercise 2 out today and due Monday (1/12) morning
 - First exercise submitted via Gitlab tagging (`ex2-submit`)
 - **Make a private Ed post if you don't have your exercise repo yet**
- ❖ Homework 1 out tonight, due in 2 weeks (Thu 1/22)
 - Linked list and hash table implementations in C
 - Get starter code by cloning your new *homework* repo

Relevant Course Information (3/3)

❖ Documentation:

- man pages, books
- Reference websites: `cplusplus.org`, `man7.org`, `gcc.gnu.org`, etc.

❖ Folklore:

- Google-ing, Stack Overflow, generative AI, that rando in Discord

❖ Tradeoffs? Relative strengths & weaknesses?

Lecture Outline (1/4)

- ❖ **Pointer Basics**
- ❖ Pointer Arithmetic
- ❖ Pointers as Parameters
- ❖ Function Pointers

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

Box-and-Arrow Diagrams (1/4)

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 (2/4)

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
------	-------

&x	x	value
&arr[2]	arr[2]	value
&arr[1]	arr[1]	value
&arr[0]	arr[0]	value
&p	p	value

stack frame for main()

Box-and-Arrow Diagrams (3/4)

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
------	-------

&x	x	1
&arr[2]	arr[2]	4
&arr[1]	arr[1]	3
&arr[0]	arr[0]	2
&p	p	&arr[1]

Box-and-Arrow Diagrams (4/4)

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	x	1
0x7fff...48	arr[2]	4
0x7fff...44	arr[1]	3
0x7fff...40	arr[0]	2
0x7fff...38	p	0x7fff...44

Lecture Outline (2/4)

- ❖ Pointer Basics
- ❖ **Pointer Arithmetic**
- ❖ Pointers as Parameters
- ❖ 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

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
```


pollev.com/cse333j


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
arr[1]	3
arr[0]	2

0x7fff...74

0x7fff...70

0x7fff...68

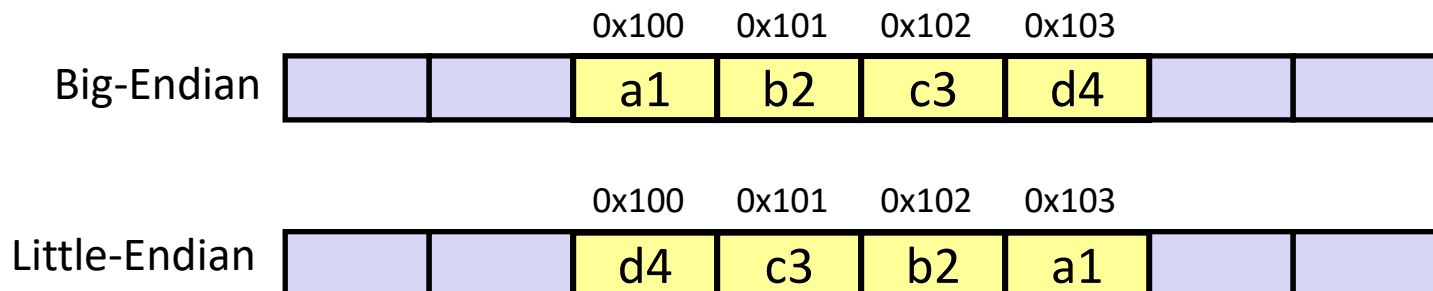
p	0x7fff...74
---	-------------

0x7fff...60

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

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/biggest* address
 - **Little-endian:** Least significant byte has *lowest/littlest* address
- ❖ **Example:** 4-byte data 0xa1b2c3d4 at address 0x100



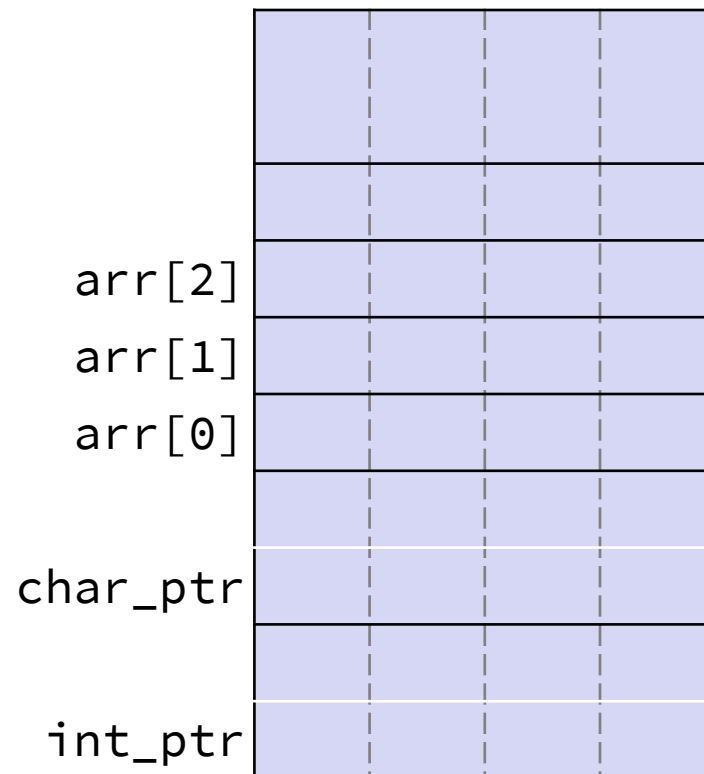
Pointer Arithmetic Example (1/10)

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 (2/10)

```
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)

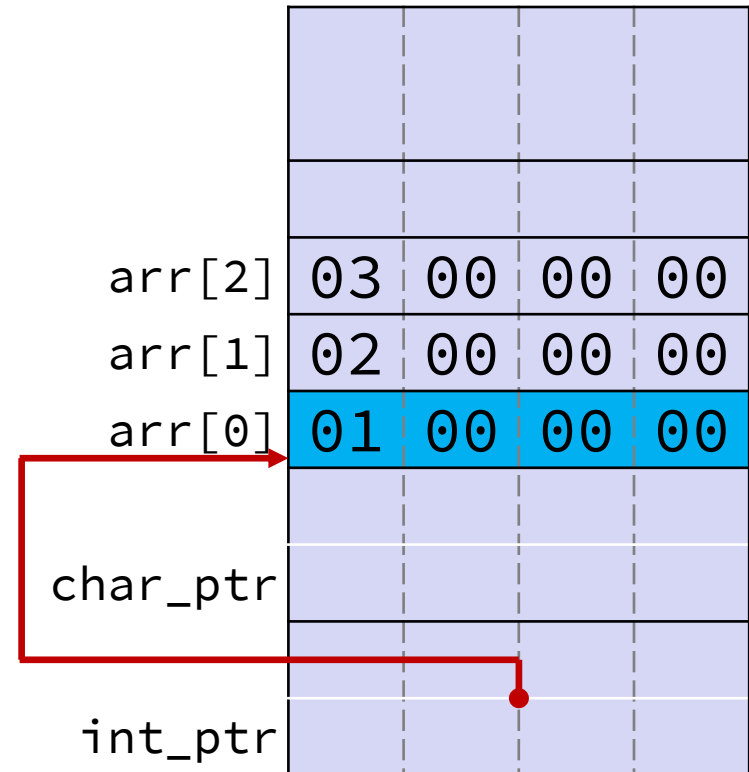
arr[2]	03	00	00	00
arr[1]	02	00	00	00
arr[0]	01	00	00	00
char_ptr				
int_ptr				

Pointer Arithmetic Example (3/10)

```
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)

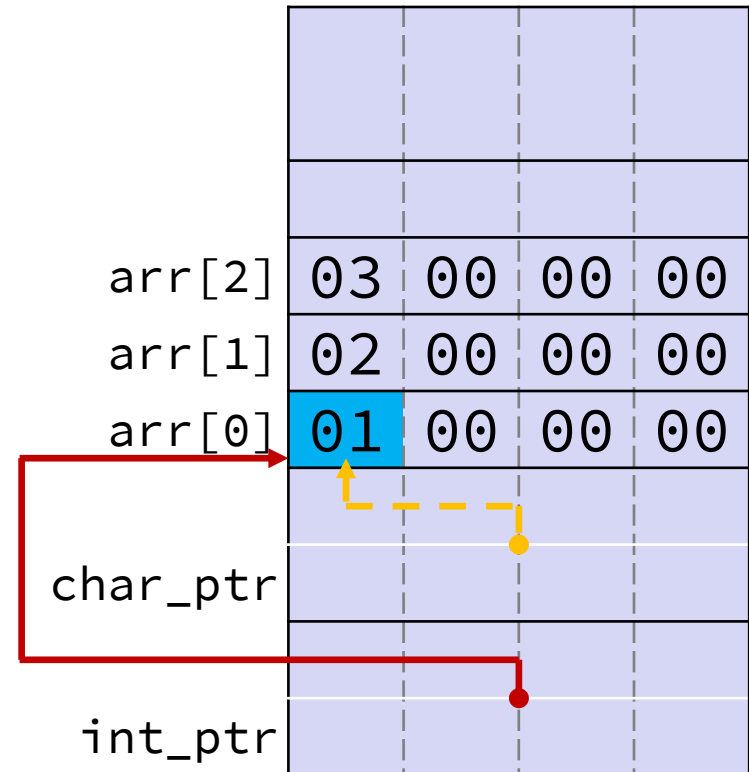


Pointer Arithmetic Example (4/10)

```
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)



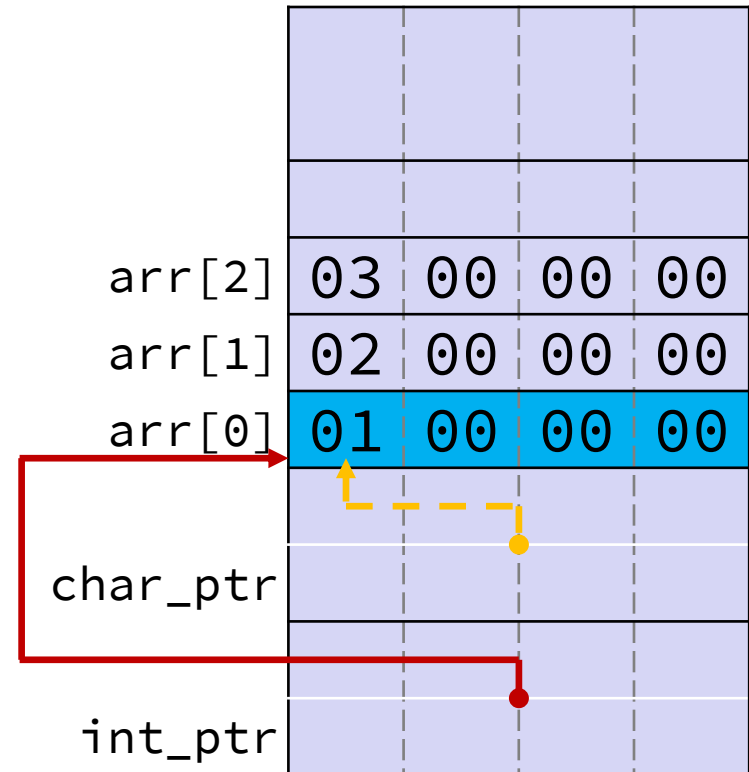
Pointer Arithmetic Example (5/10)

```
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: 0x7fffffffde010
***int_ptr:** 1

Stack
(assume x86-64)

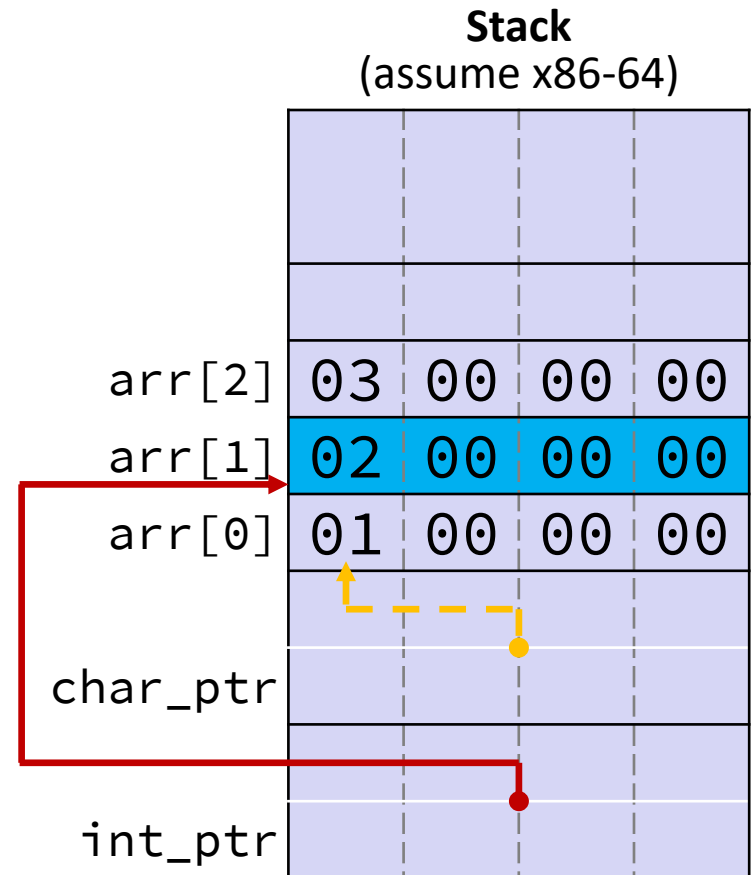


Pointer Arithmetic Example (6/10)

```
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: 0x7fffffffde014
***int_ptr:** 2



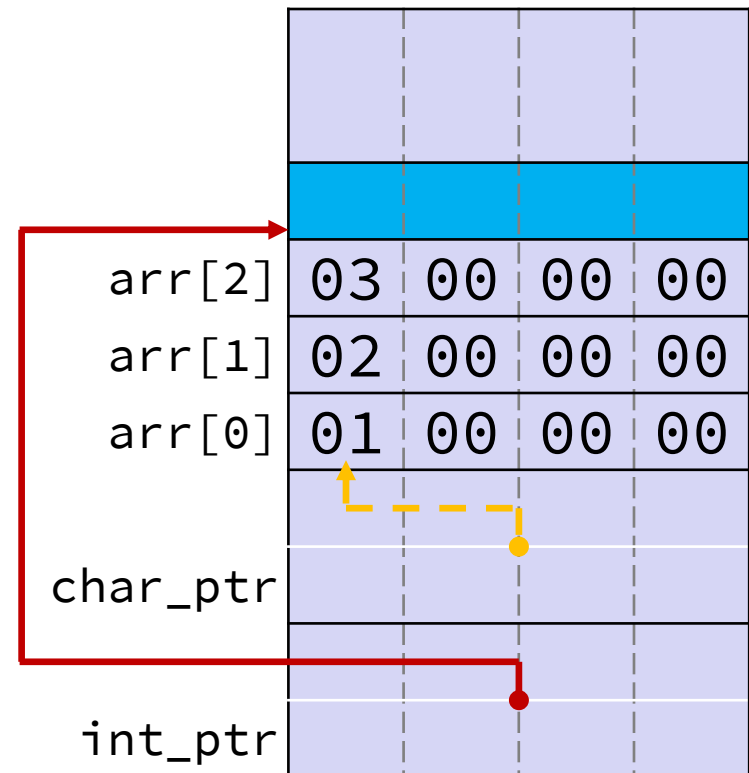
Pointer Arithmetic Example (7/10)

```
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: 0x7fffffffde01**c**
***int_ptr:** ???

Stack
(assume x86-64)



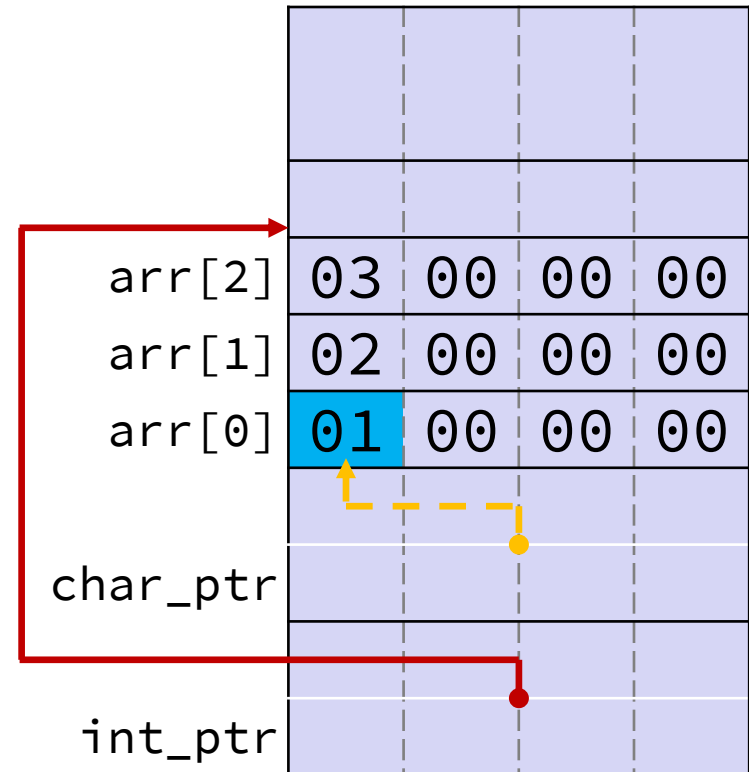
Pointer Arithmetic Example (8/10)

```
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: 0x7fffffffde010
***char_ptr:** 1

Stack
(assume x86-64)



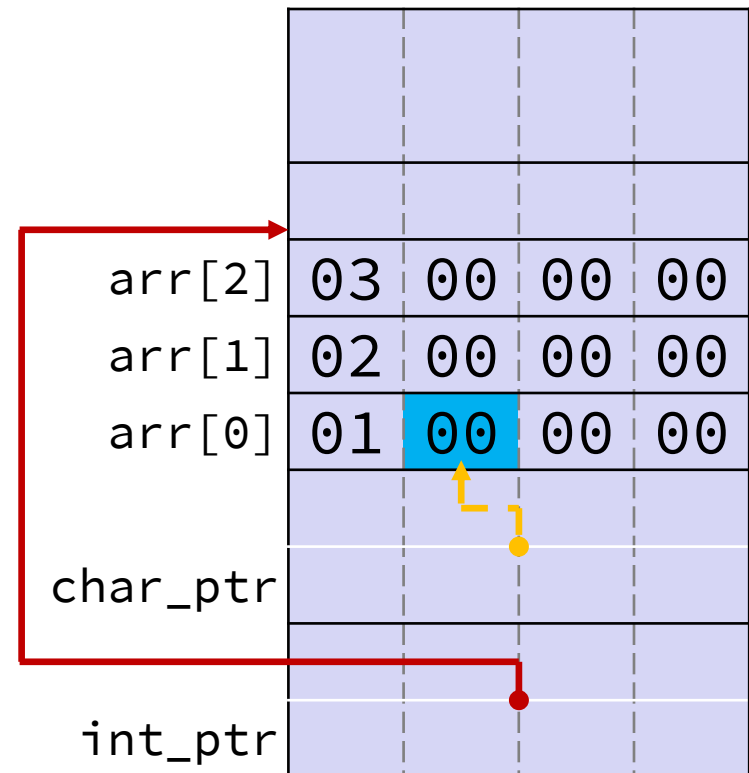
Pointer Arithmetic Example (9/10)

```
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: 0x7fffffffde01**1**
***char_ptr:** **0**

Stack
(assume x86-64)



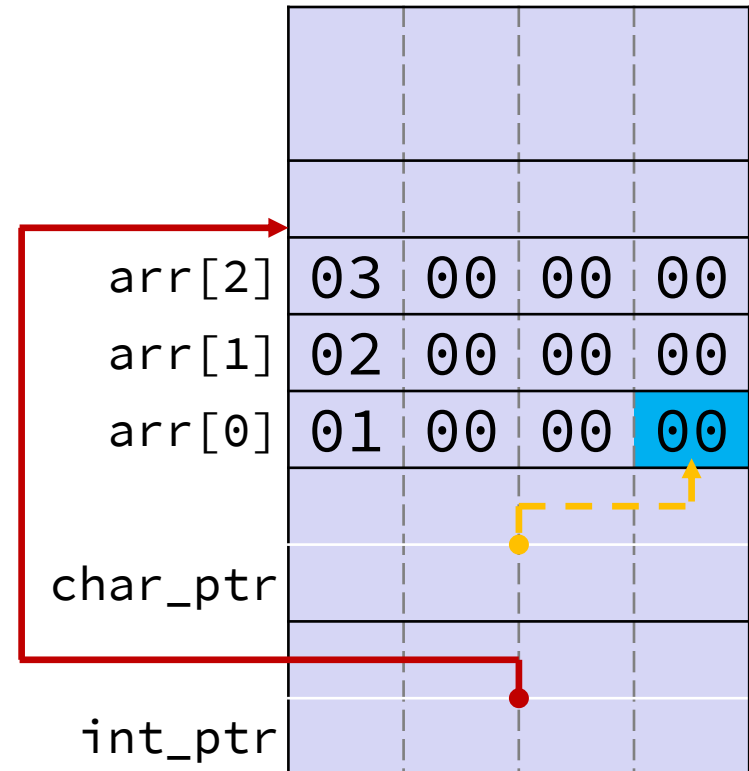
Pointer Arithmetic Example (10/10)

```
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: 0x7fffffffde013
***char_ptr:** 0

Stack
(assume x86-64)



Lecture Outline (3/4)

- ❖ Pointer Basics
- ❖ Pointer Arithmetic
- ❖ **Pointers as Parameters**
- ❖ 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

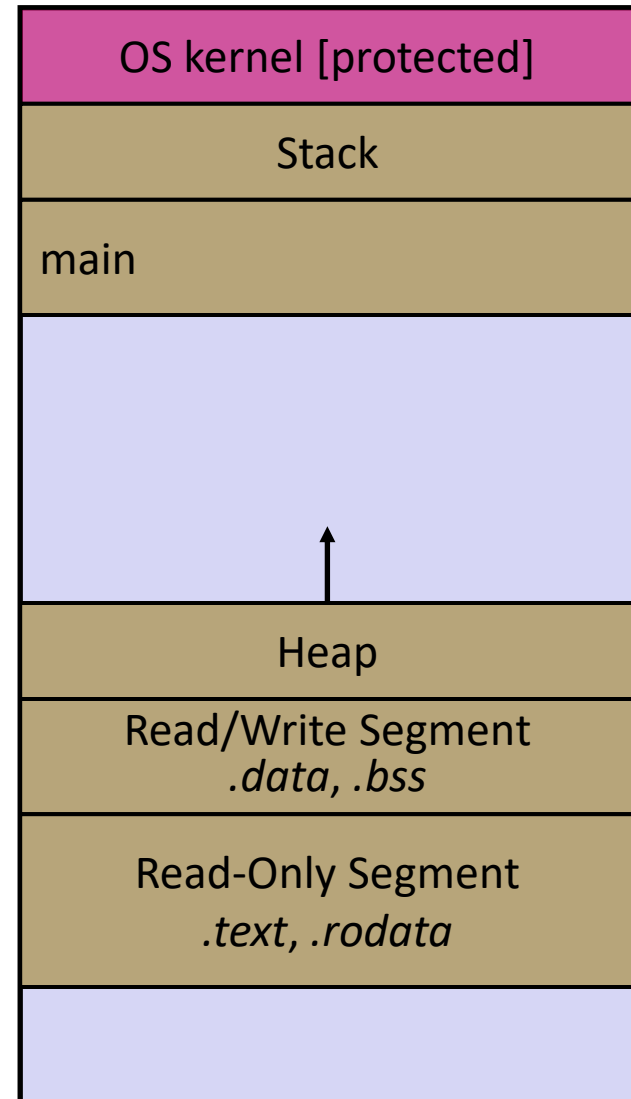
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```

Broken Swap (1/7)

Note: Arrow points to *next* instruction.

breakswap.c

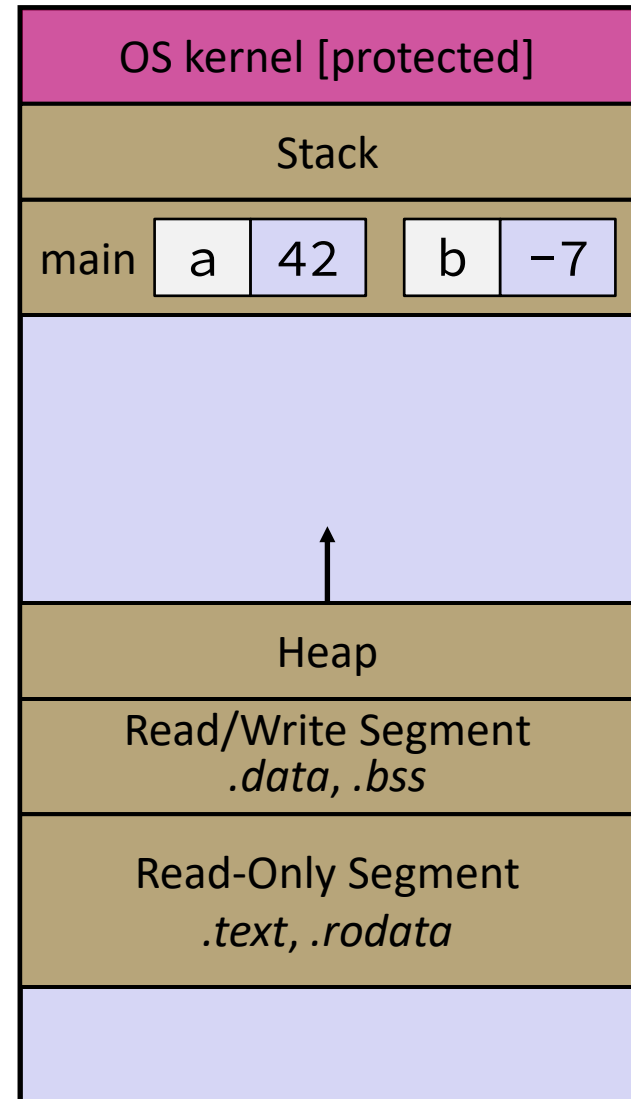
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
→   int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (2/7)

breakswap.c

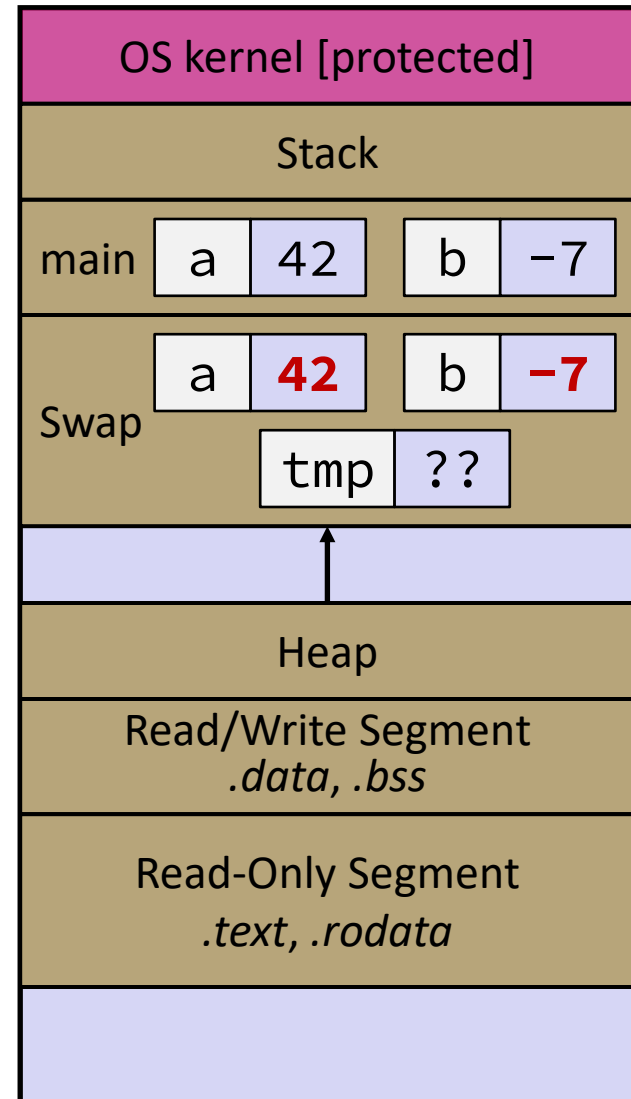
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (3/7)

breakswap.c

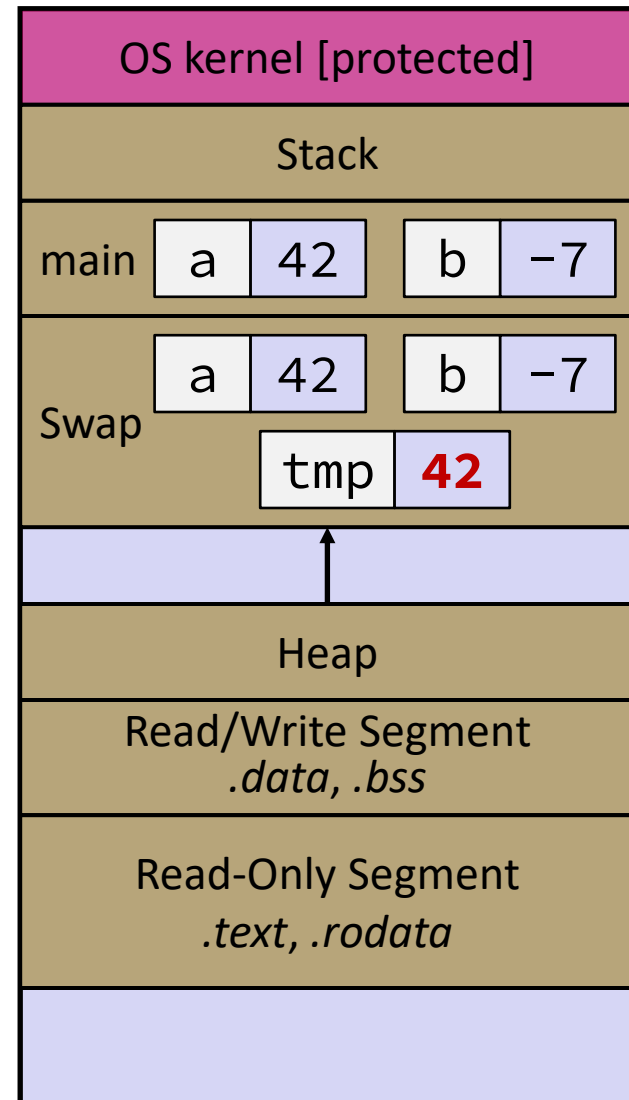
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (4/7)

breakenswap.c

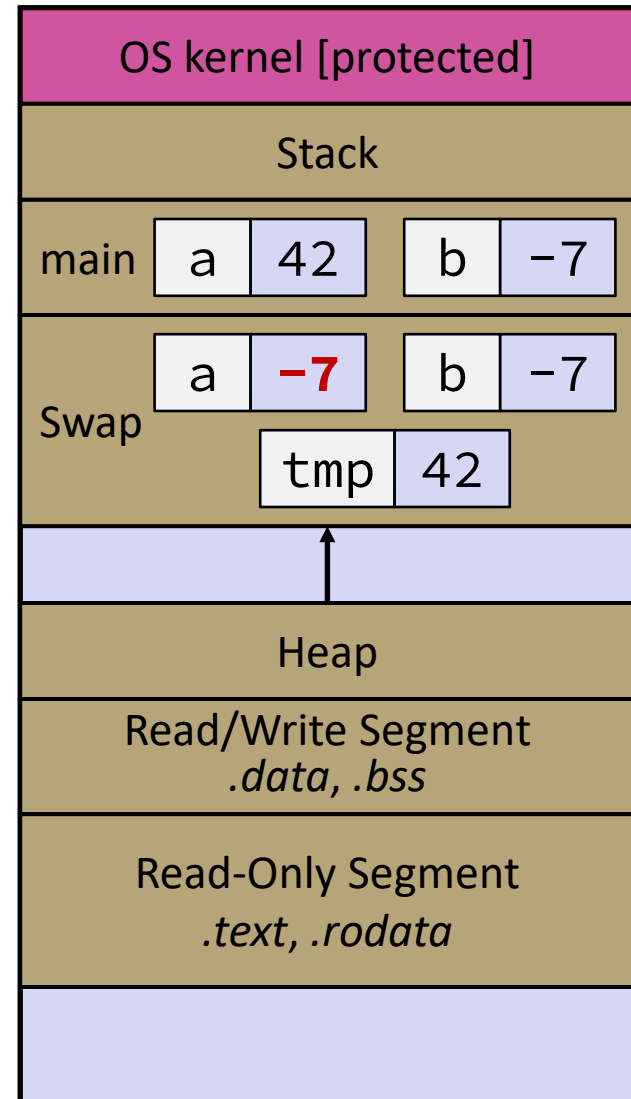
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (5/7)

breakenswap.c

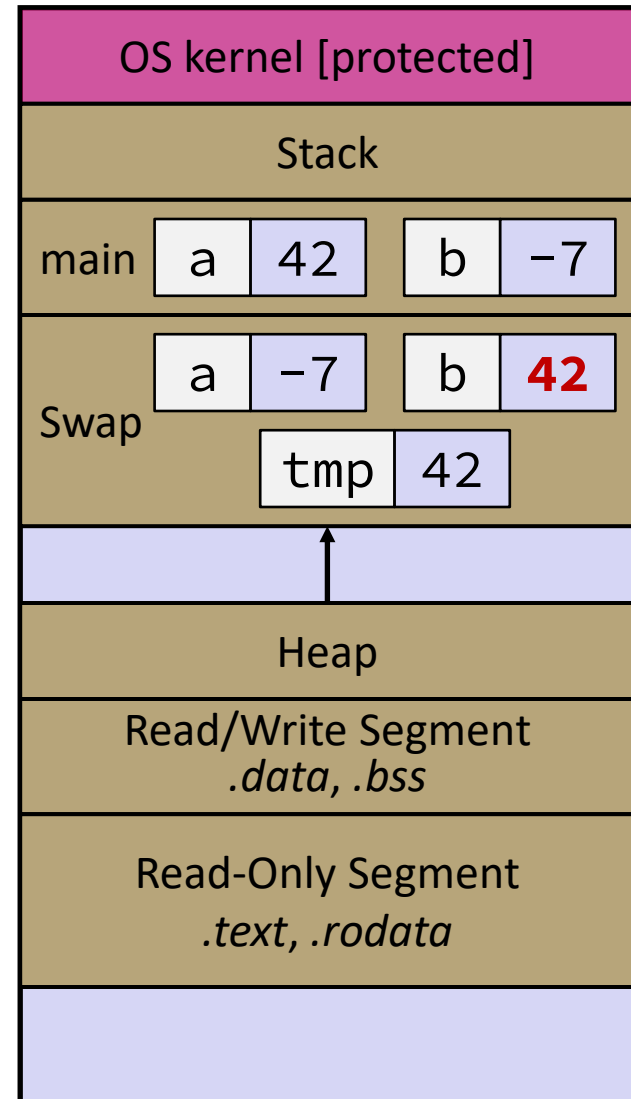
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (6/7)

breakenswap.c

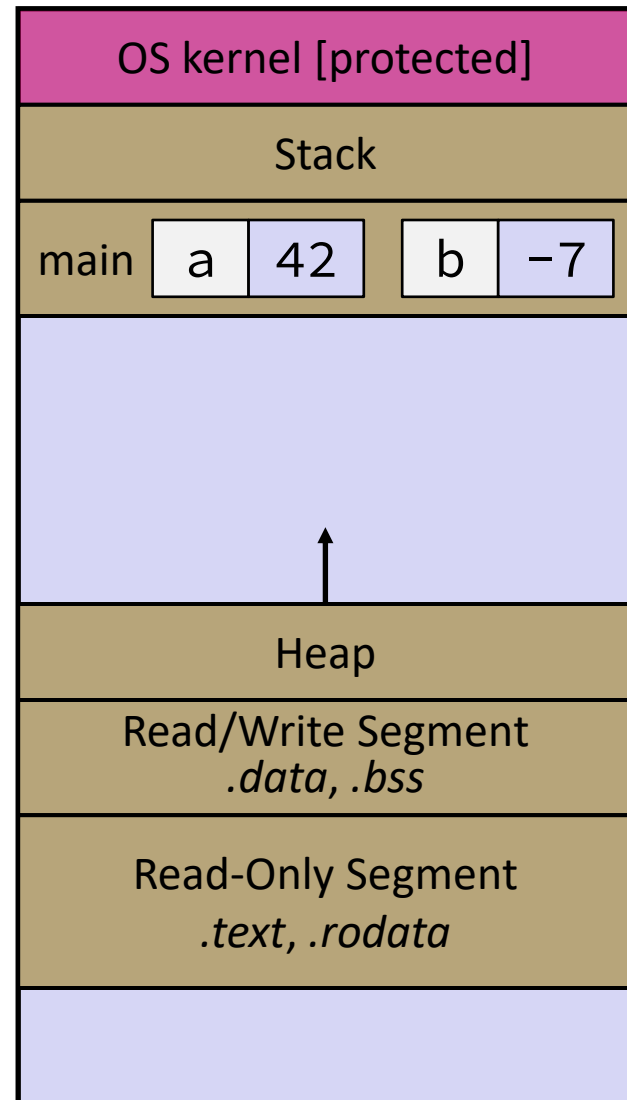
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (7/7)

breakenswap.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

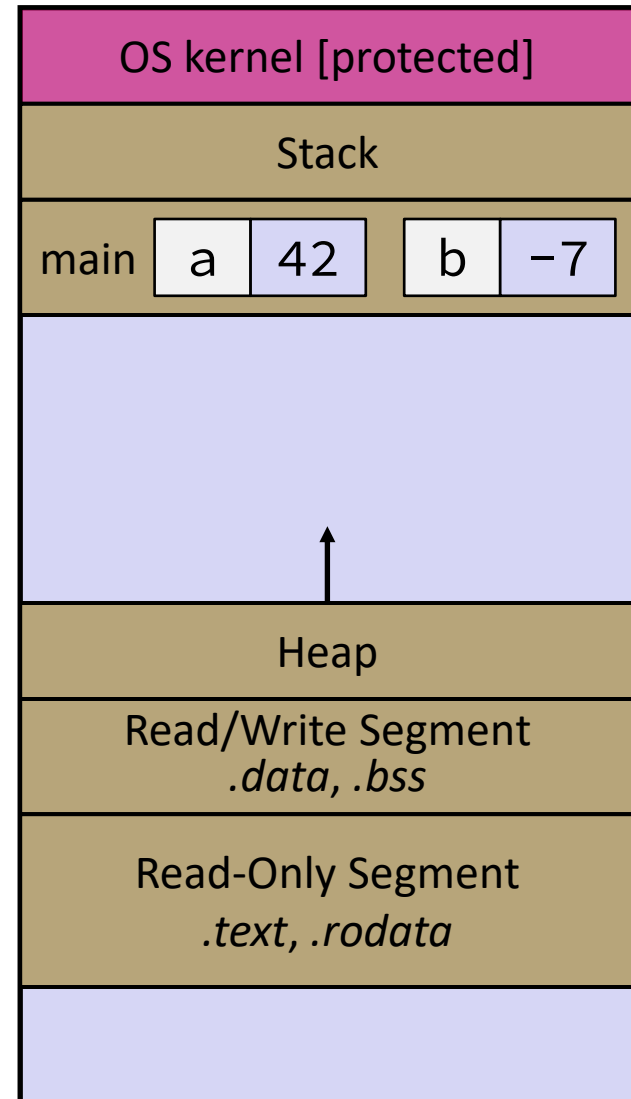
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```

Fixed Swap (1/6)

Note: Arrow points to *next* instruction.

swap.c

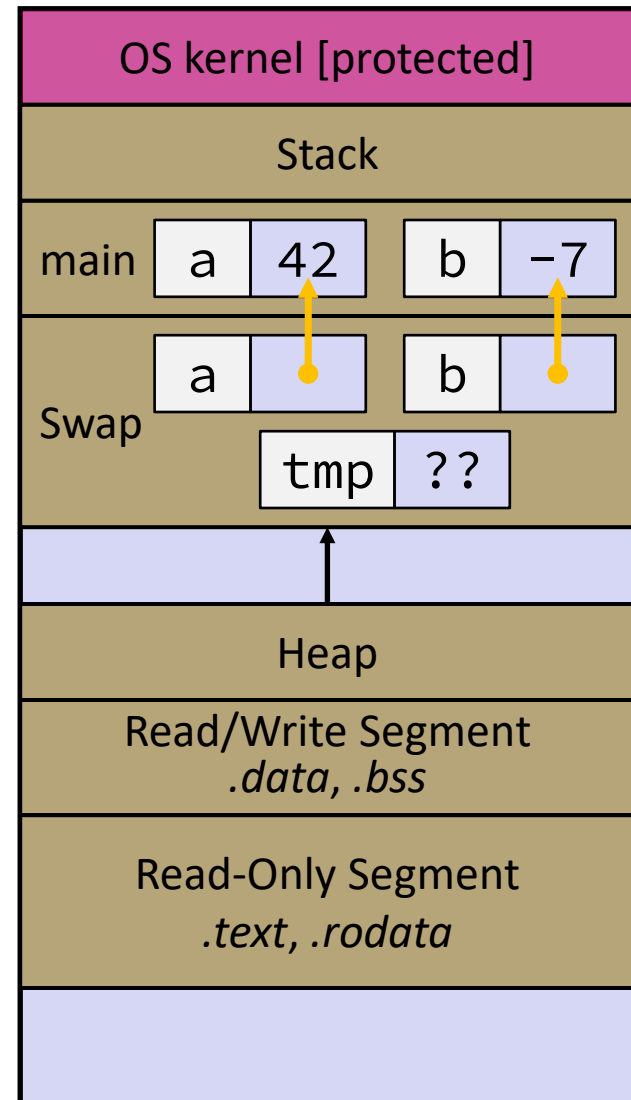
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (2/6)

swap.c

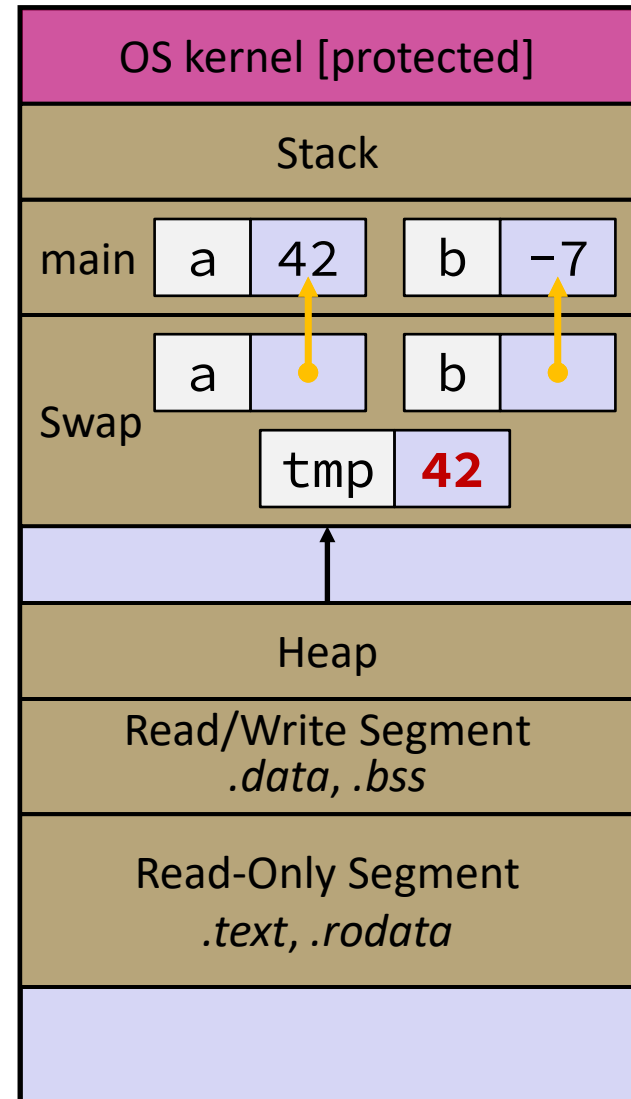
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (3/6)

swap.c

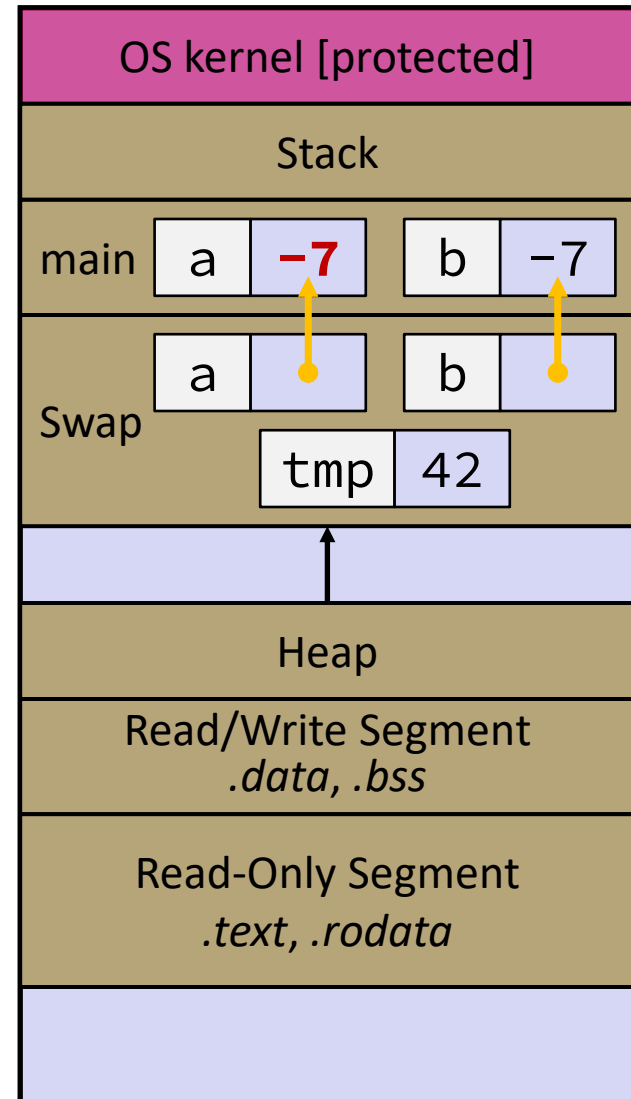
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    → *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (4/6)

swap.c

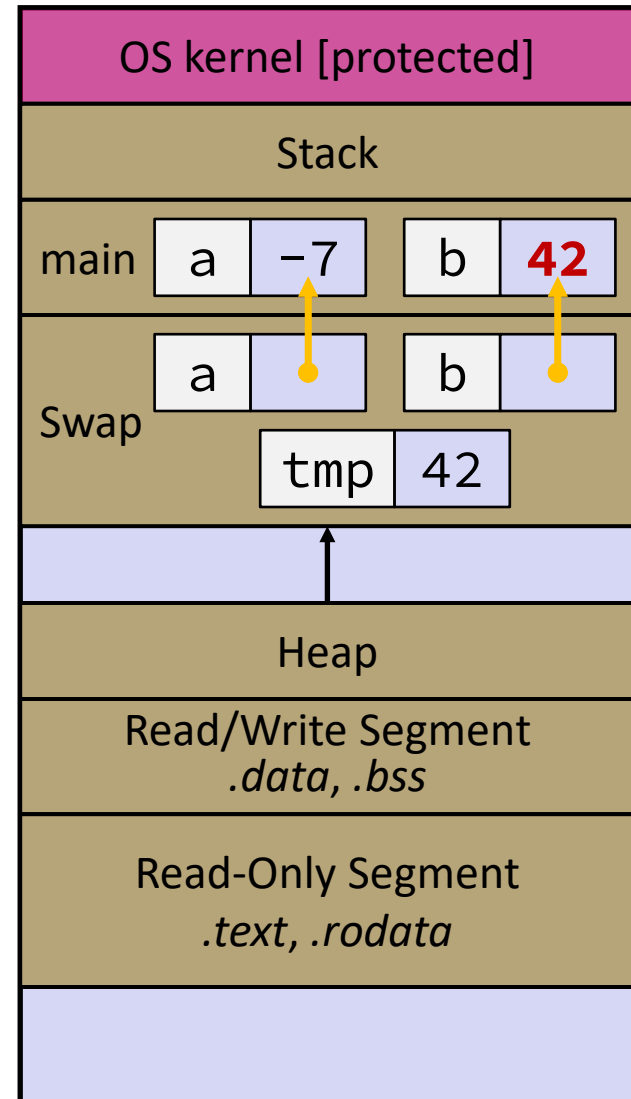
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (5/6)

swap.c

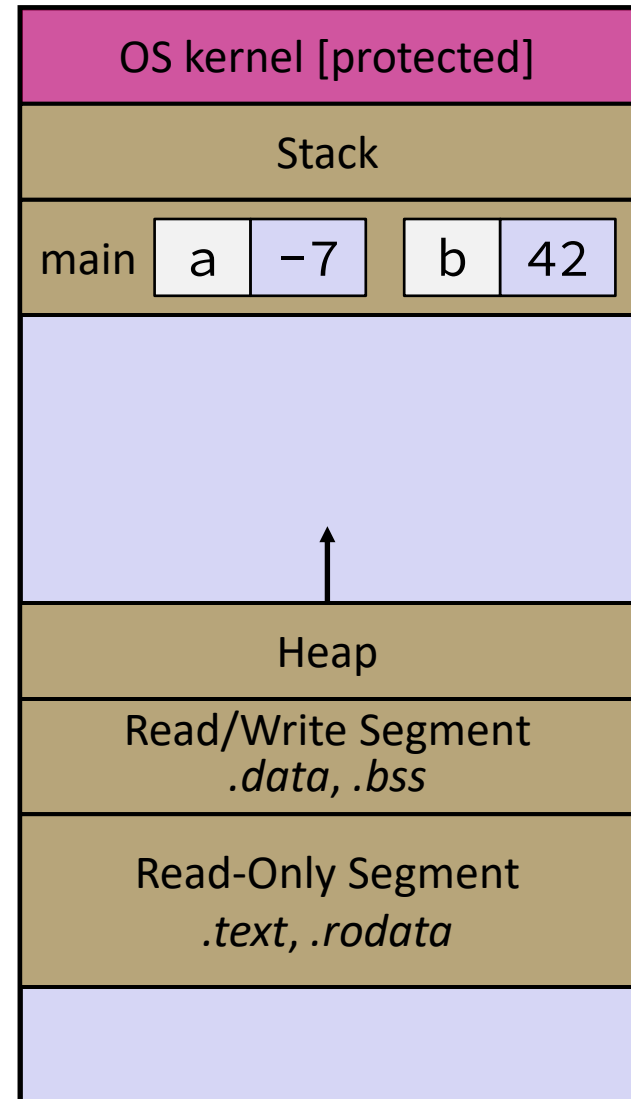
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (6/6)

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`)

pollev.com/cse333j

Which of the following are *invalid* ways to invoke `GenerateString()`?

- Of the working ways, which would be preferred?

```
void GenerateString(char** output) {  
    *output = "Hello there\n";  
}
```

A.

```
char** result;  
GenerateString(result);  
printf("%s", *result);
```

B.

```
char* str;  
char** result = &str;  
GenerateString(result);  
printf("%s", str);
```

C.

```
char* result[1] = {NULL};  
GenerateString(result);  
printf("%s", result[0]);
```

D.

```
char* result;  
GenerateString(&result);  
printf("%s", result);
```

E. We're lost...

Lecture Outline (4/4)

- ❖ Pointer Basics
- ❖ Pointer Arithmetic
- ❖ Pointers as Parameters
- ❖ **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 (1/2)

- ❖ 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

funcptr dereference

funcptr definition

funcptr assignment

Function Pointer Example (2/2)

- ❖ 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