



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

Systems Programming

Pointers

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar

Advaith Vijayakumar

Alice Fu

Andrew Vlahovich

Aviel Wood

Camden Harris

Esha Gore

Hannah Hempstead

Helena Wang

Laura Wang

Megan Frisella

Nina Tharamal

Shrihun Sankepally

Shriya Sateesh

Yusong Yan

Relevant Course Information (1/3)

- ❖ Exercise 1 was due this morning, will be uploaded to Gradscope for you
 - Leniency on submission issues – you should get an email if submission issue, **have until 11:59 PM tonight to correct TAG** (if no email, no resubmission)
- ❖ Exercise 2 released today, due Friday @ 11 AM
- ❖ Meet the Staff activity due next Friday (10/16)
 - Go to office hour, chat with a staff member, fill out Gradescope quiz afterward
- ❖ HW1 due next Thursday (10/15) @ 11:59 PM
 - You **may not** modify interfaces (.h files), but **do** read the interfaces while you're implementing them (!)
 - Suggestion: pace yourself and make steady progress

Relevant Course Information (2/3)

- ❖ Exercise grading (see Ed post for more details)
 - Sample solutions released same day as deadline
 - Autograder scores visible after upload; full grades follow shortly after
 - Style things to watch for:
 - FOLLOW THE SPEC (especially the Style Guide section)
 - Check the CSE 333 Style Guide
 - Make a judgment call and document
 - Keep style tips in mind, as you will need to use them in Homework

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;`

*equivalent just bc
consistent*

❖ *Dereference* a pointer using the unary * operator

- Access the memory referred to by a pointer

Box-and-Arrow Diagrams (1/4)

```
int main(%rdiint argc, %rsichar* 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;  
}
```

need address, must be in stack

boxarrow.c

(address) name value

x 
Stack

p 
Stack

arr 
Stack

Box-and-Arrow Diagrams (2/4)

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

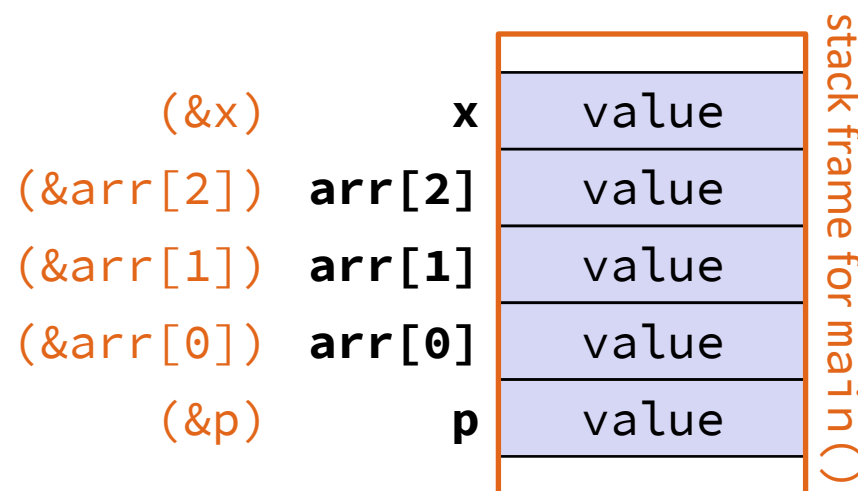
    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

boxarrow.c

(address) name

value



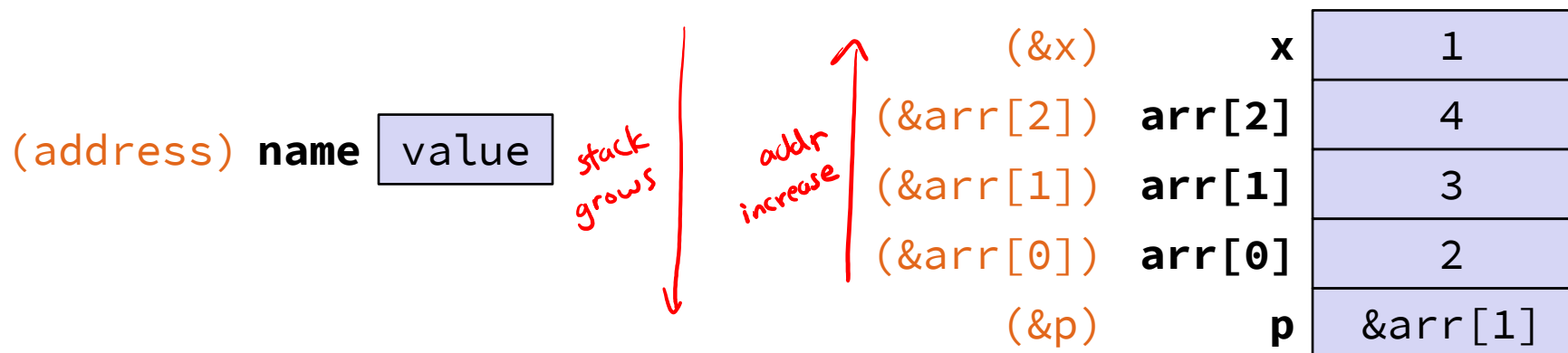
Box-and-Arrow Diagrams (3/4)

```
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;
}
```

boxarrow.c



Box-and-Arrow Diagrams (4/4)

```
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;
}
```

boxarrow.c

(address) name value

p: get addr
**p: get data at addr (follow arrow)*

(0x7f...4c)	x	1
(0x7f...48)	arr[2]	4
(0x7f...44)	arr[1]	3
(0x7f...40)	arr[0]	2
(0x7f...38)	p	0x7f...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 😞

size of the thing
being pointed at

❖ 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`

$ptr[i] \leftrightarrow *(ptr+i) \leftrightarrow *(i+ptr) \leftrightarrow i[ptr]$

■ 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

↑
DONT USE
but all ptrs
"under the
hood"

```
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/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

(0x7f...78)	arr[2]	4
(0x7f...74)	arr[1]	3
(0x7f...70)	arr[0]	2

(0x7f...68)	p	0x7f...74
-------------	---	-----------

(0x7f...60)	dp	0x7f...68
-------------	----	-----------

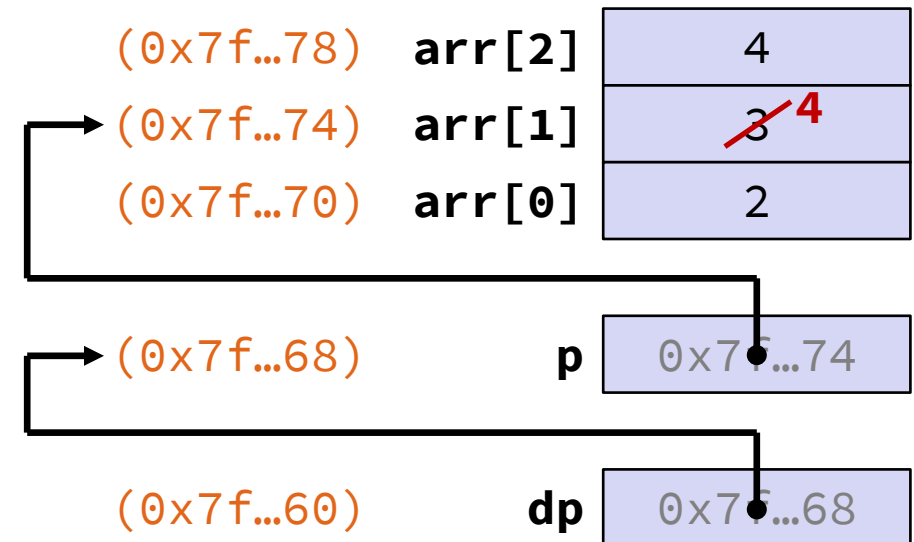
- A. {2, 3, 4} B. {3, 4, 5} C. {2, 6, 4} **D. {2, 4, 5}** E. We're lost...

Practice Solution (1/4)

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

```
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; // same as **dp += 1;  
    p += 1;  
    *(*dp) += 1;  
    return EXIT_SUCCESS;  
}
```

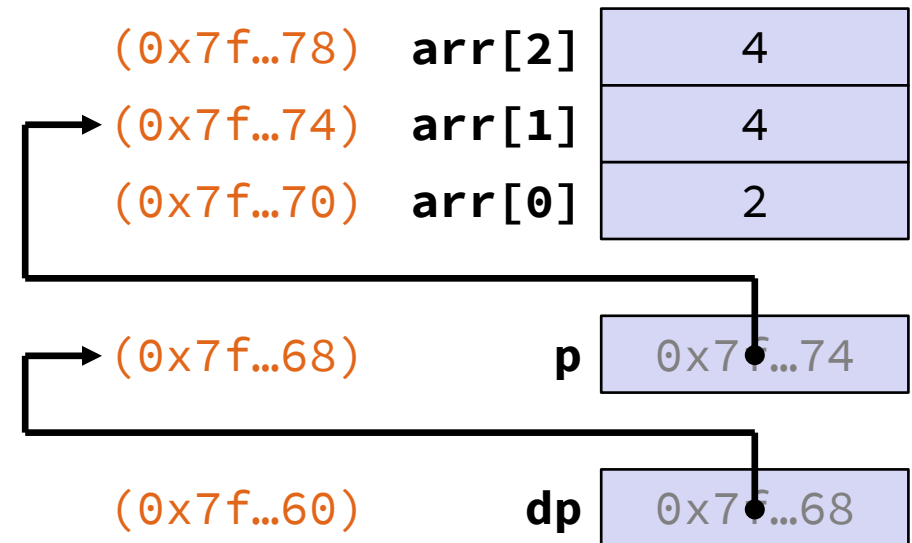
boxarrow2.c



Practice Solution (2/4)

```
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;  
}
```

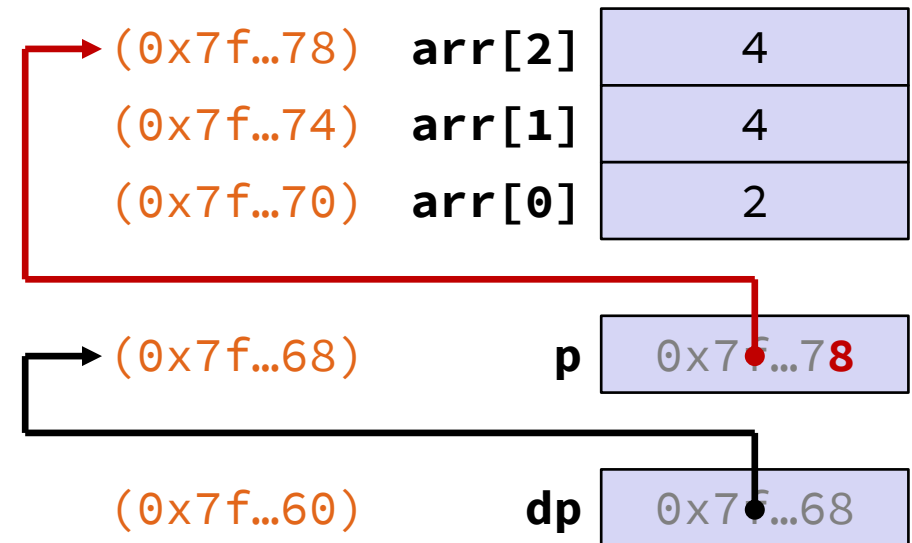
boxarrow2.c



Practice Solution (3/4)

```
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;  
}
```

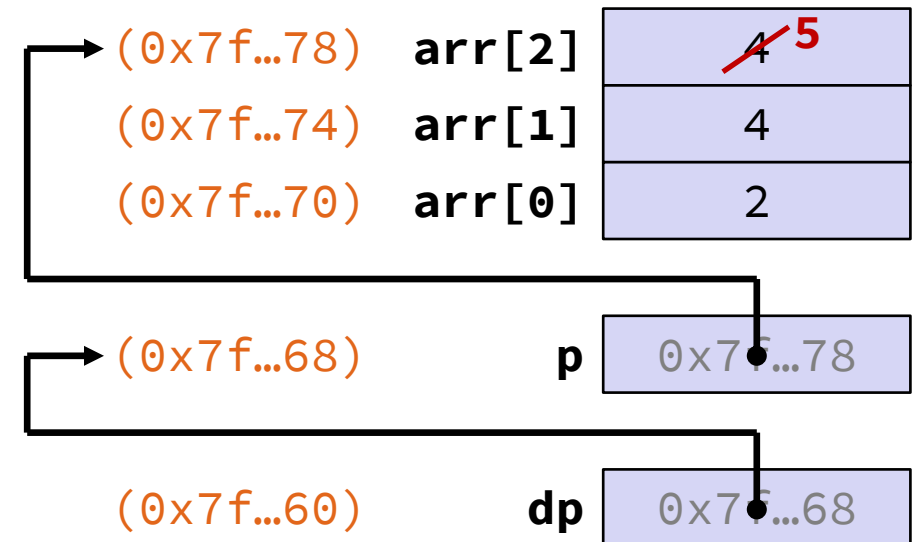
boxarrow2.c



Practice Solution (4/4)

```
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



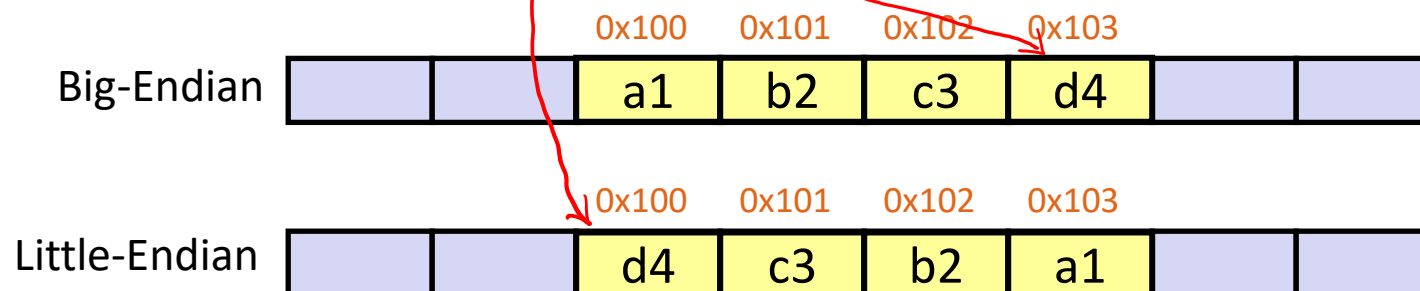
Endianness

- ❖ 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
(x86-64)

- ❖ **Example:** 4-byte data 0xa1b2c3d4 at address 0x100

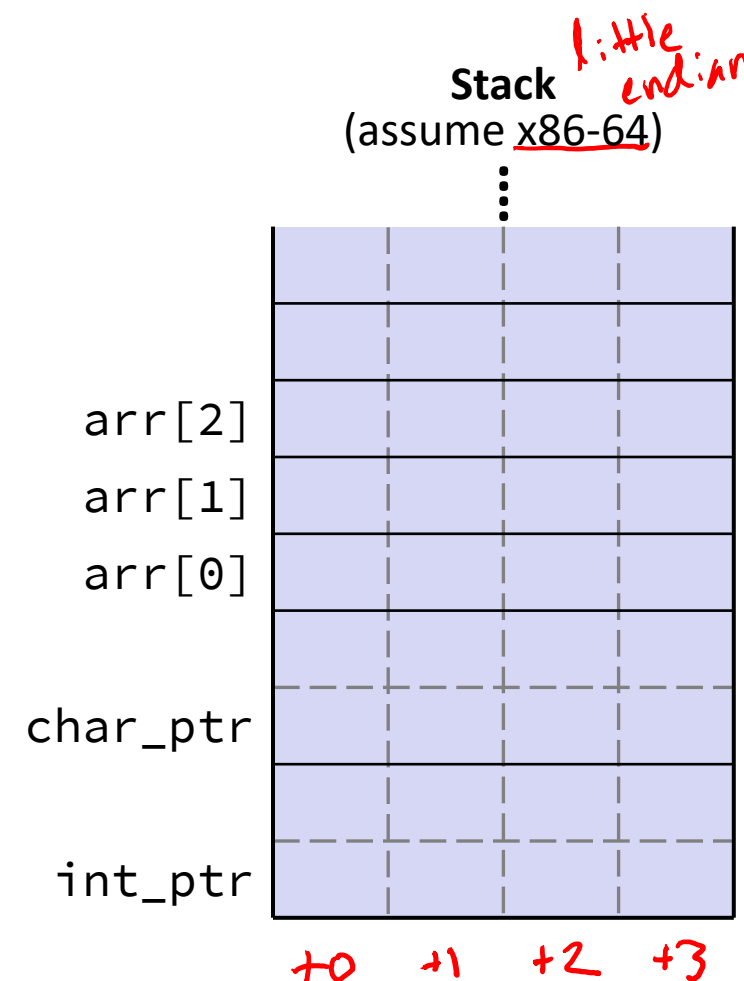


Pointer Arithmetic Example (1/9)

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

pointerarithmetic.c

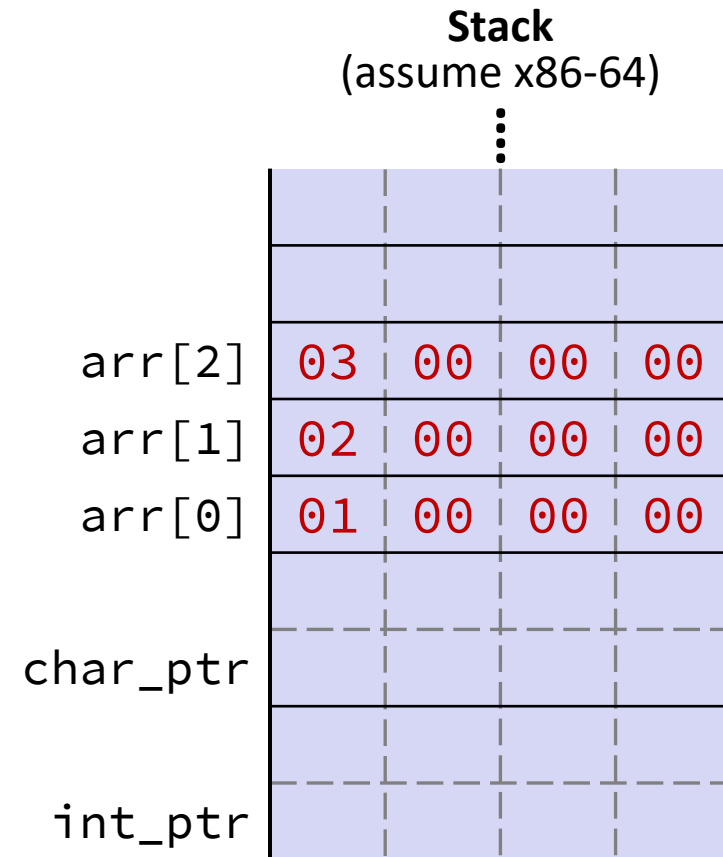
Note: arrow points to *next* instruction to be executed.



Pointer Arithmetic Example (2/9)

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

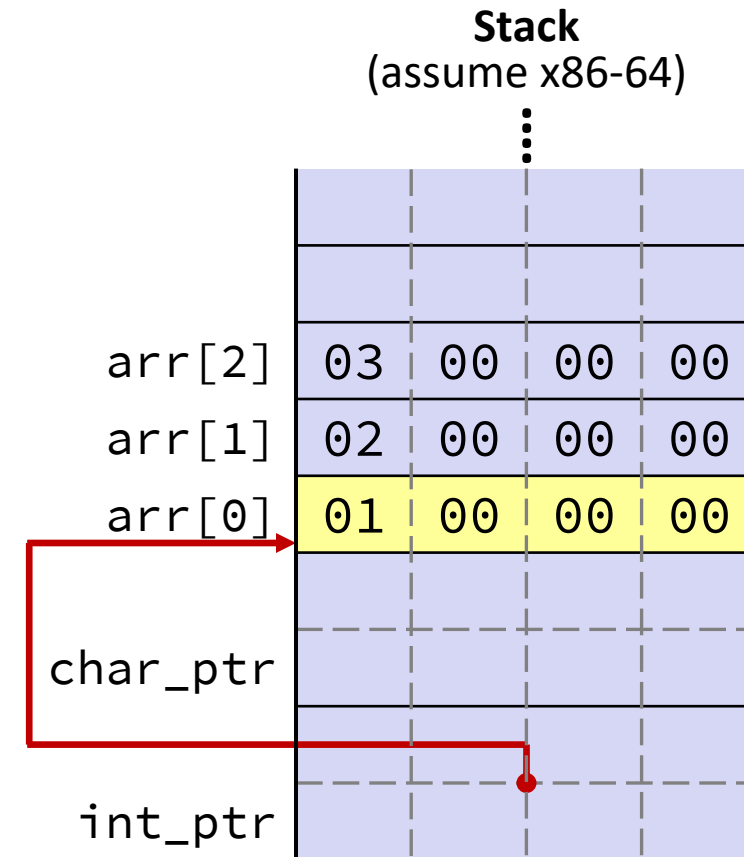
pointerarithmetic.c



Pointer Arithmetic Example (3/9)

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

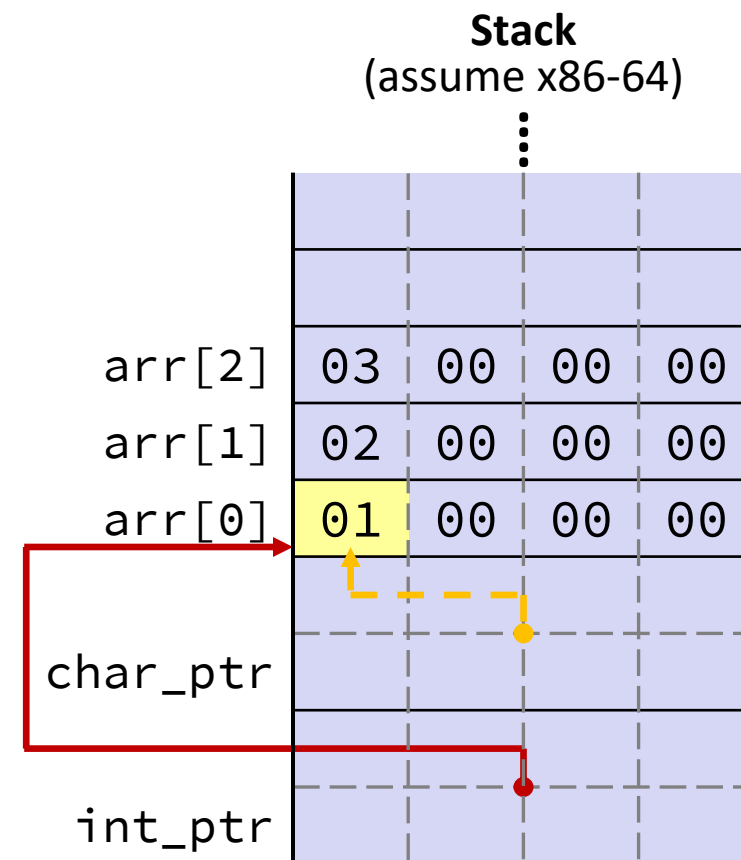
pointerarithmetic.c



Pointer Arithmetic Example (4/9)

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

pointerarithmetic.c

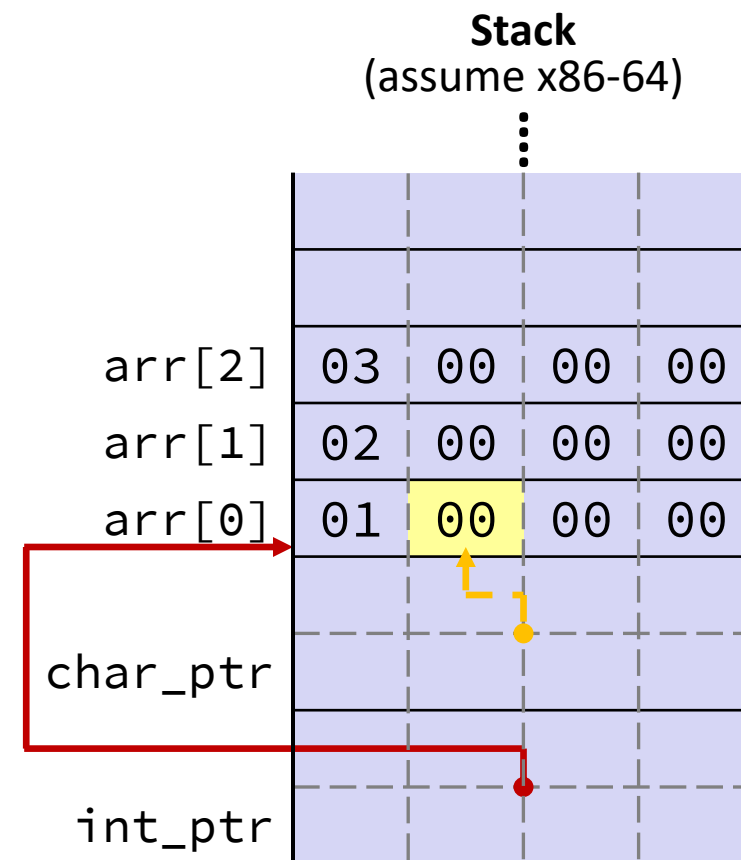


char_ptr: 0x7fffffffde010
*char_ptr: 1

Pointer Arithmetic Example (5/9)

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

pointerarithmetic.c

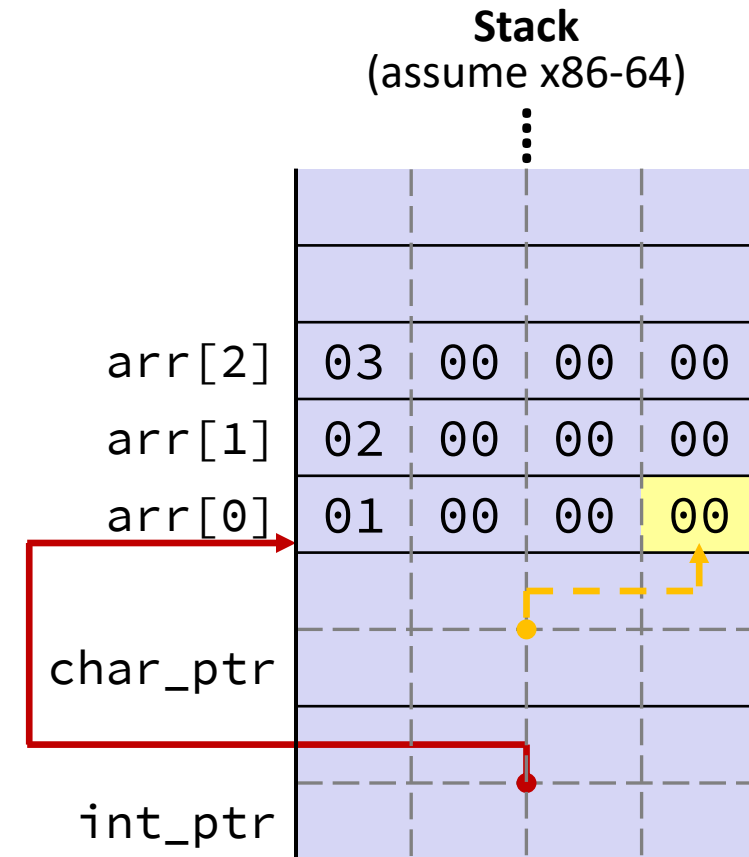


char_ptr: 0x7fffffffde011
*char_ptr: 0

Pointer Arithmetic Example (6/9)

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

pointerarithmetic.c

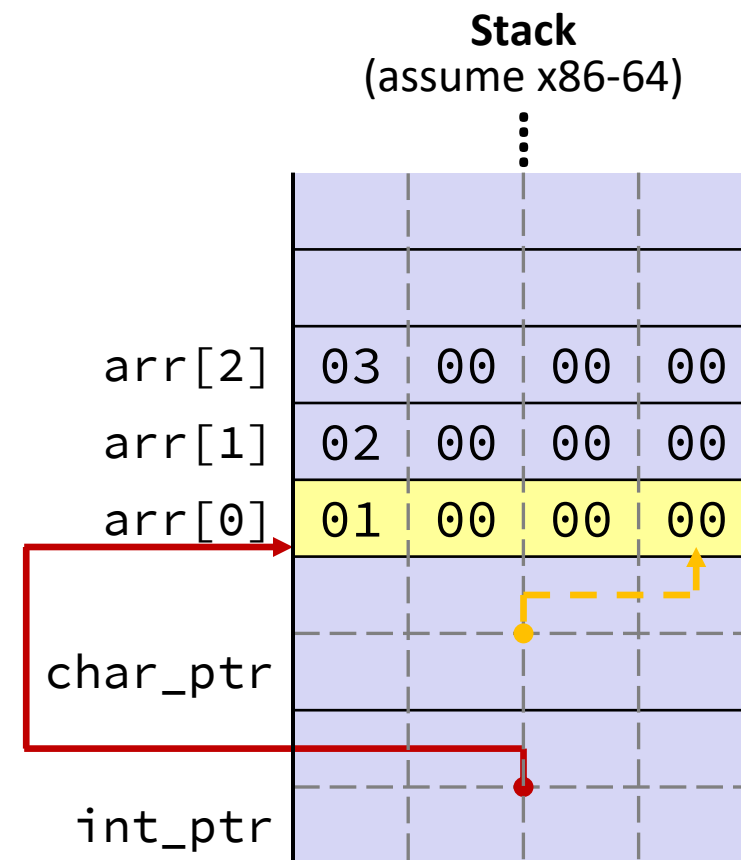


char_ptr: 0x7fffffffde013
*char_ptr: 0

Pointer Arithmetic Example (7/9)

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

pointerarithmetic.c

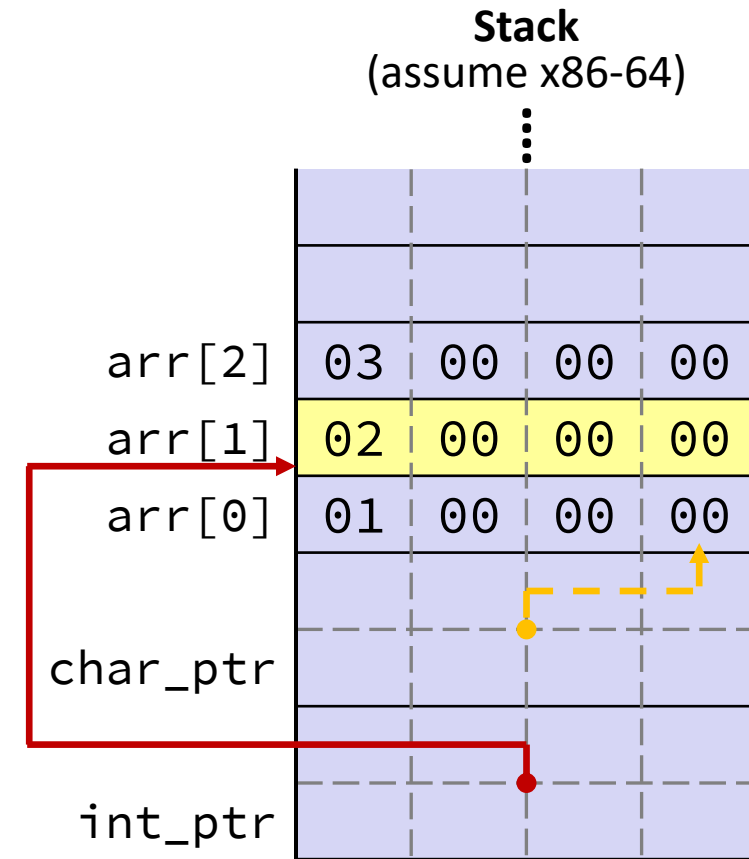


int_ptr: 0x7fffffffde010
***int_ptr:** 1

Pointer Arithmetic Example (8/9)

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

pointerarithmetic.c

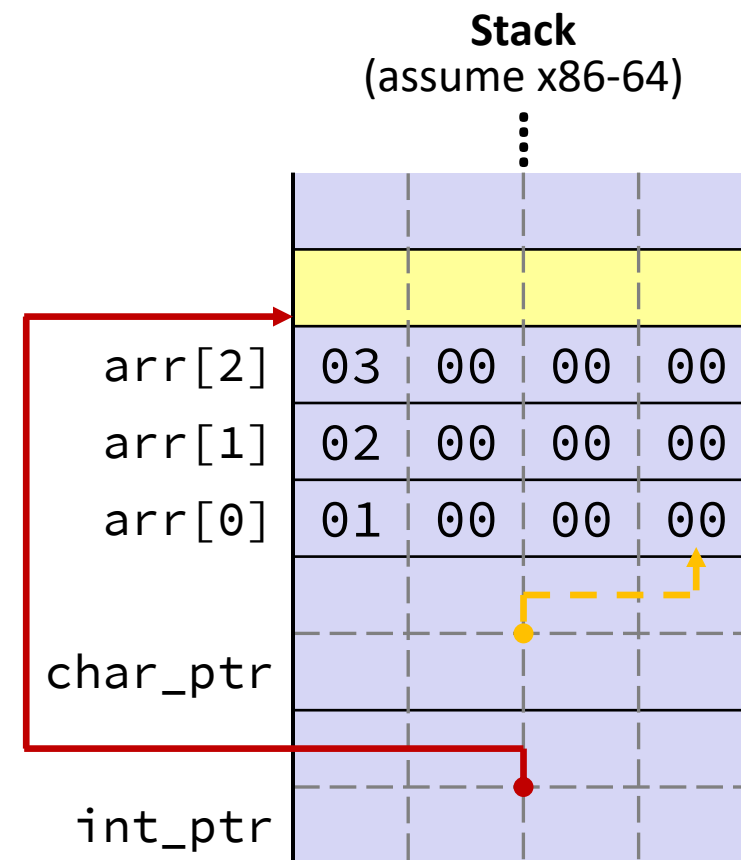


int_ptr: 0x7fffffffde014
*int_ptr: 2

Pointer Arithmetic Example (9/9)

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

pointerarithmetic.c



int_ptr: 0x7fffffffde01c
*int_ptr: ???

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

brokenswap.c

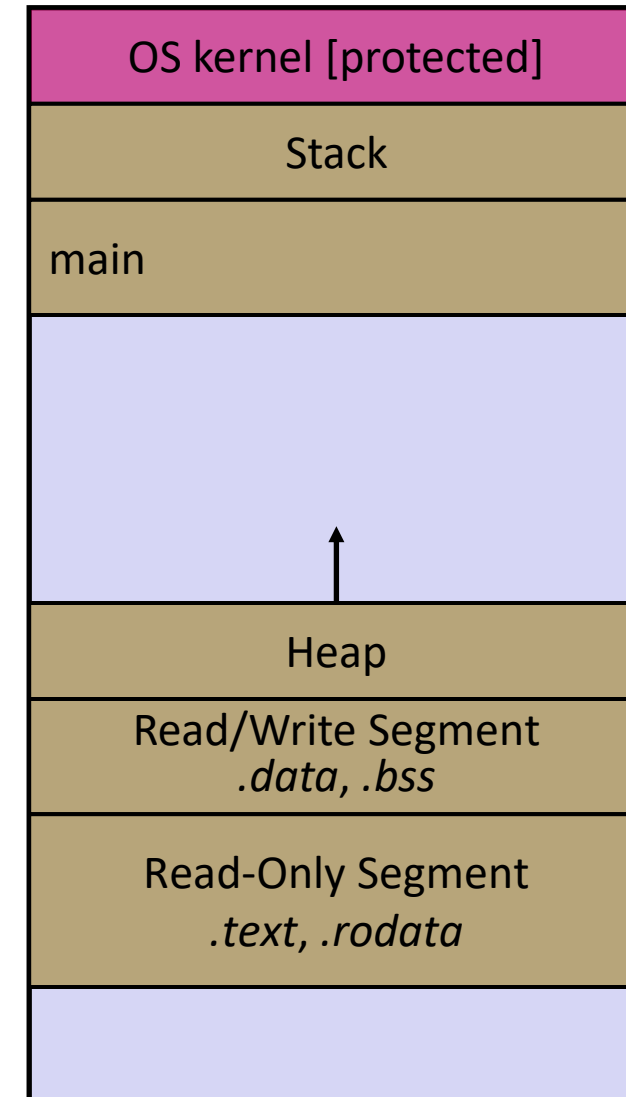
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```

Broken Swap (1/7)

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

breakswap.c

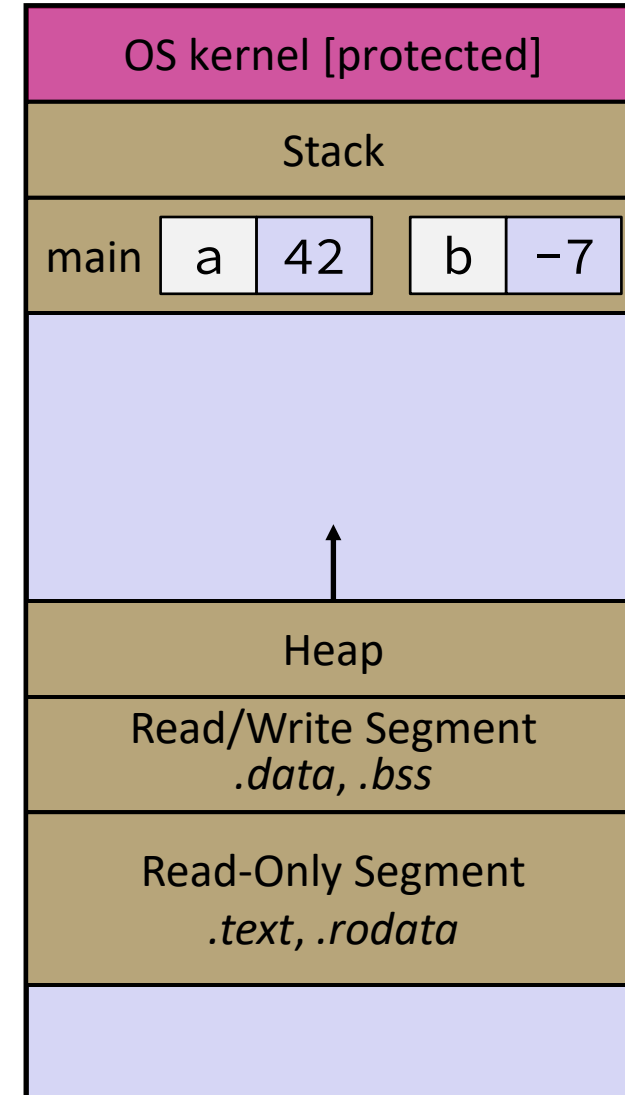
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (2/7)

breakswap.c

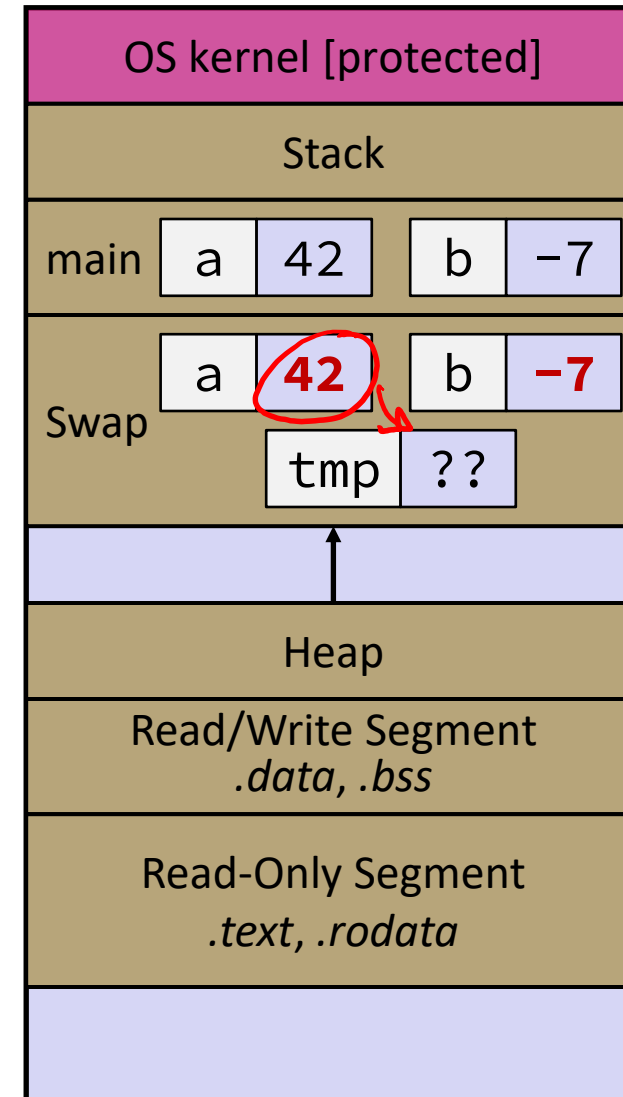
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (3/7)

breakswap.c

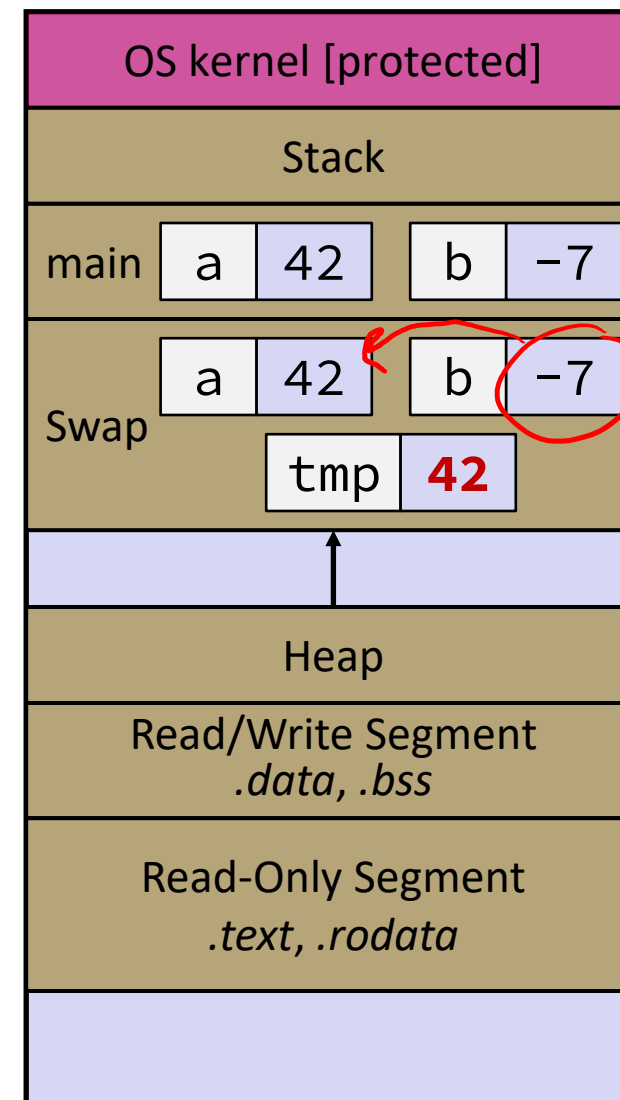
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (4/7)

breakswap.c

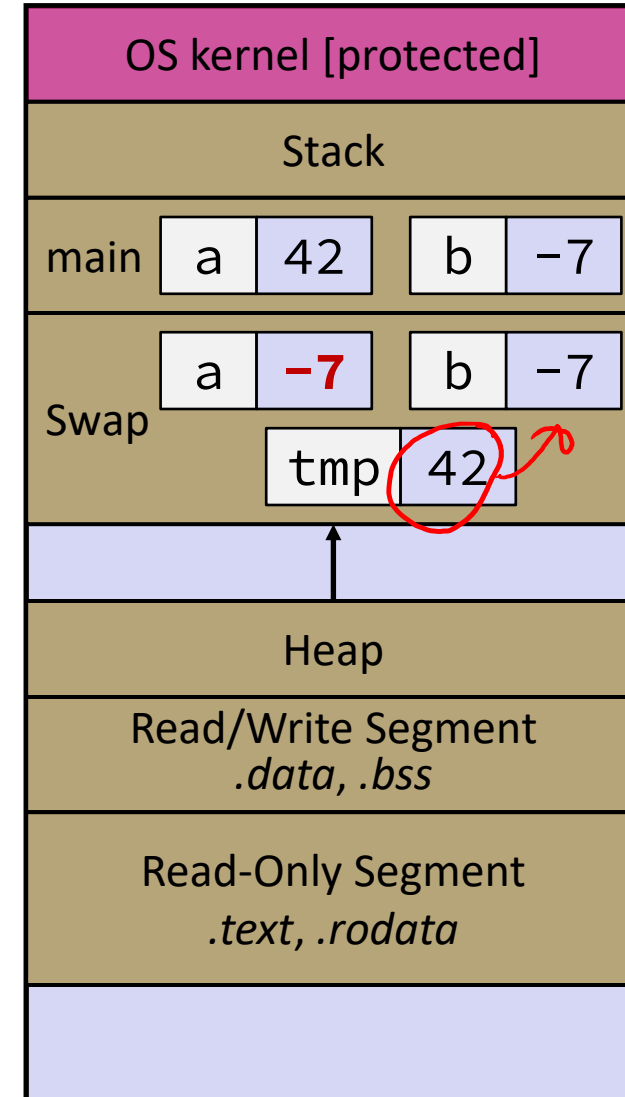
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (5/7)

brokenswap.c

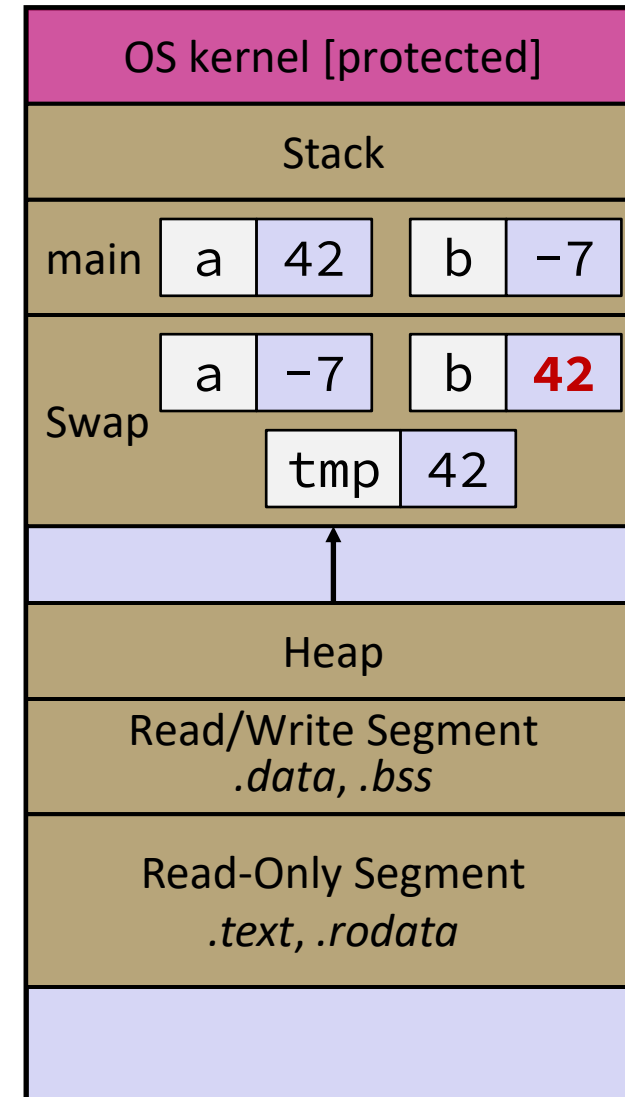
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (6/7)

brokenswap.c

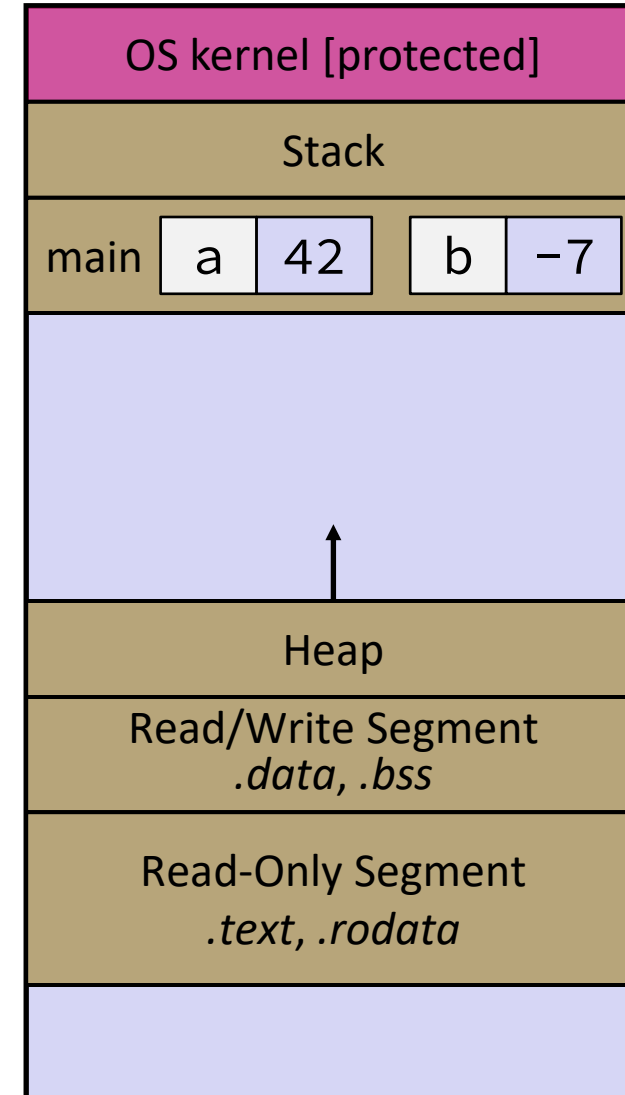
```
void Swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(a, b);  
    ...  
}
```



Broken Swap (7/7)

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);  
    ...  
}
```



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

swap.c a_ptr b_ptr

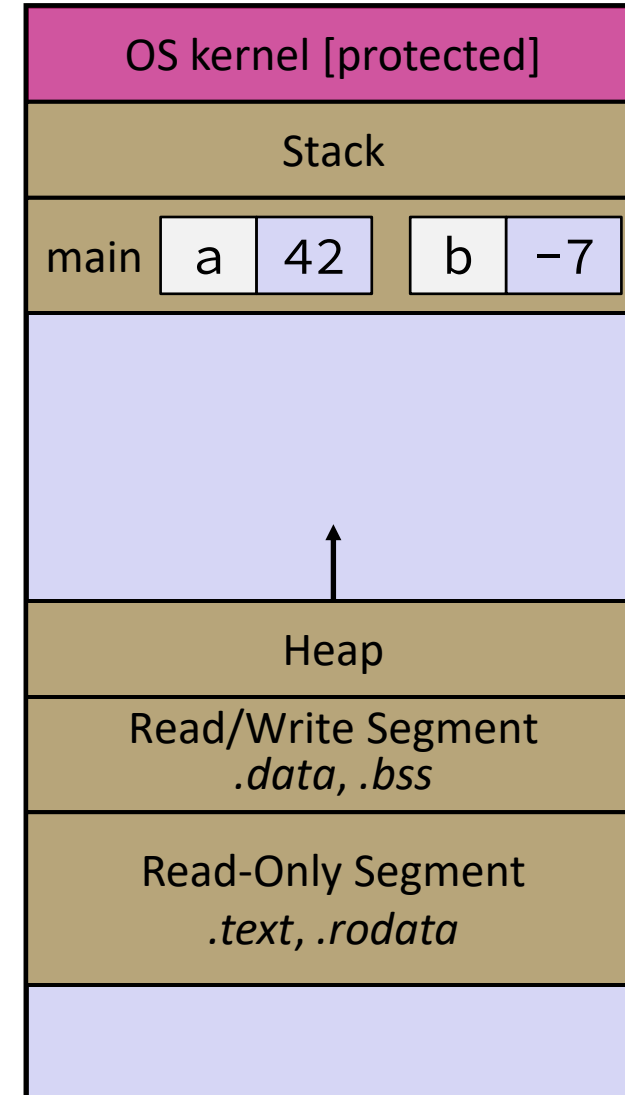
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```

Fixed Swap (1/6)

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

swap.c

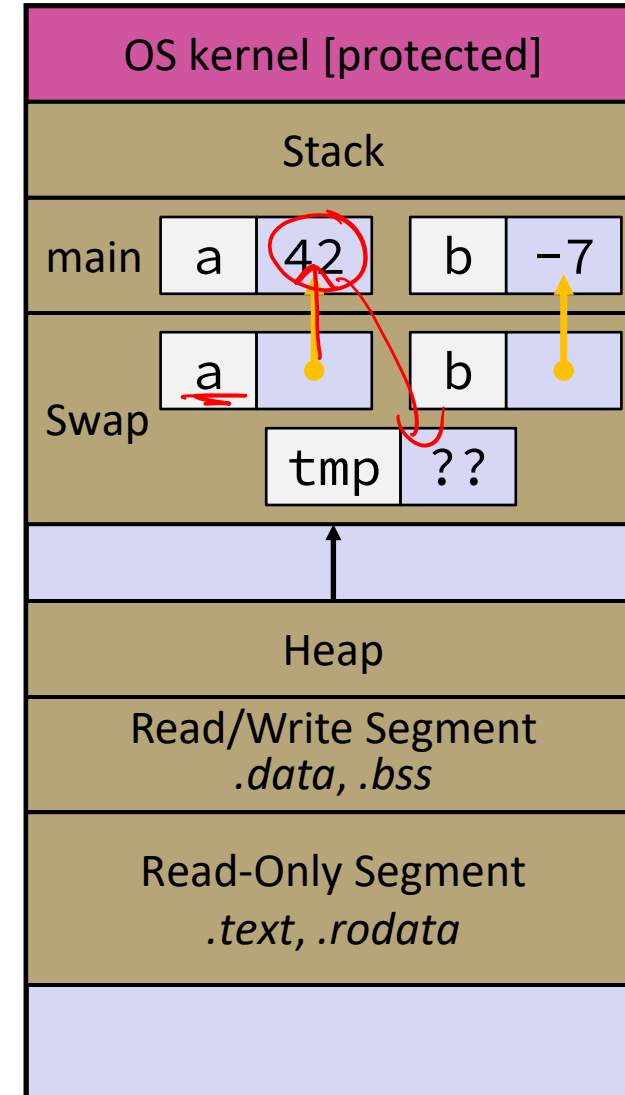
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (2/6)

swap.c

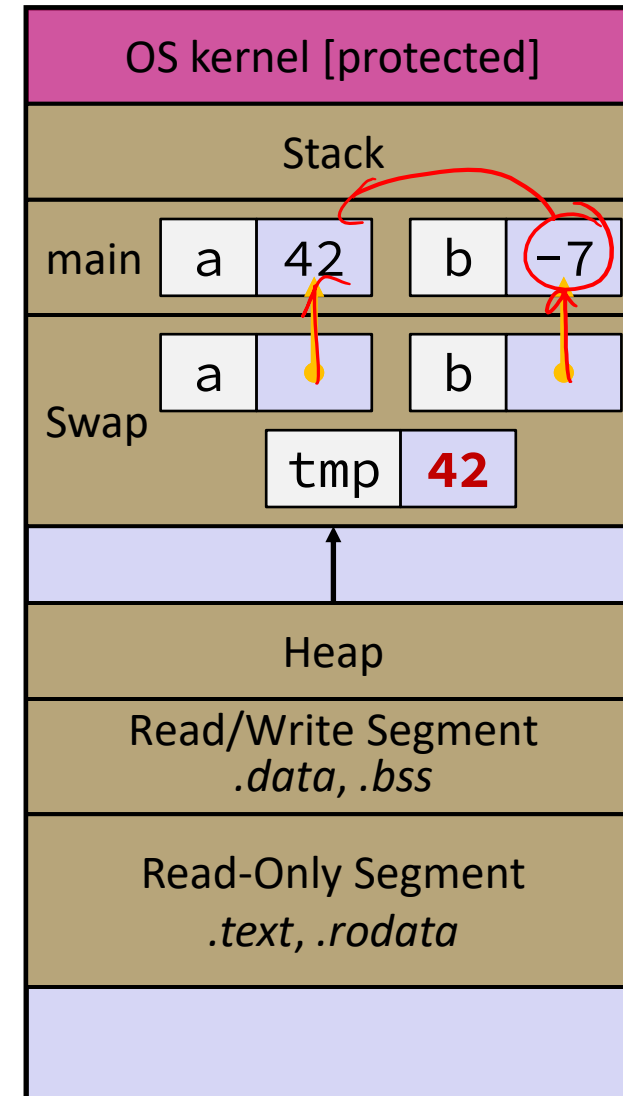
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (3/6)

swap.c

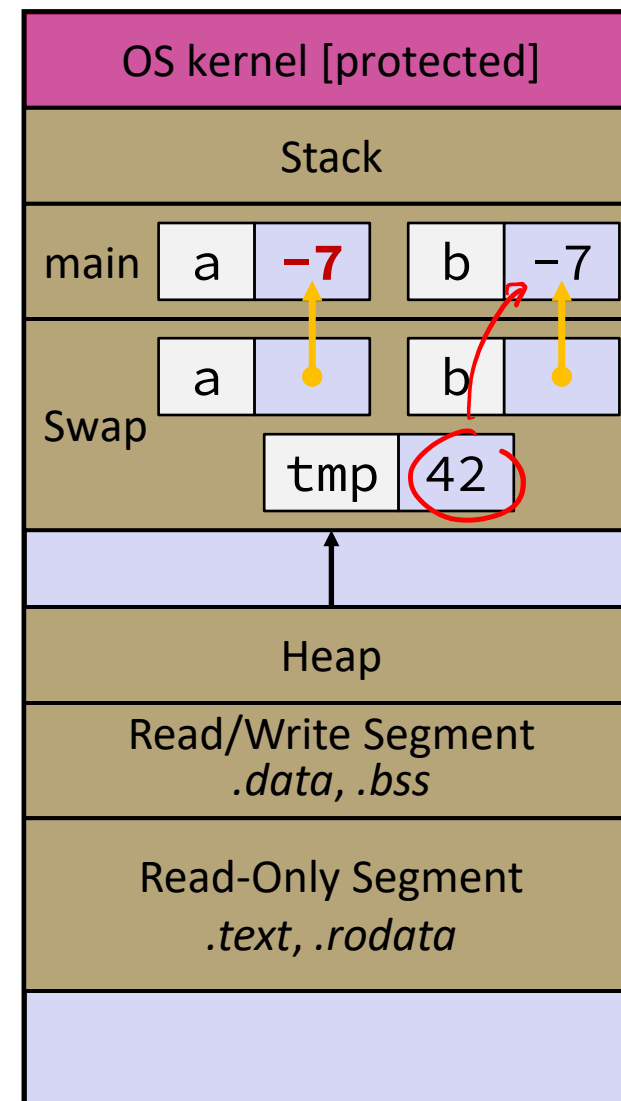
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    → *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (4/6)

swap.c

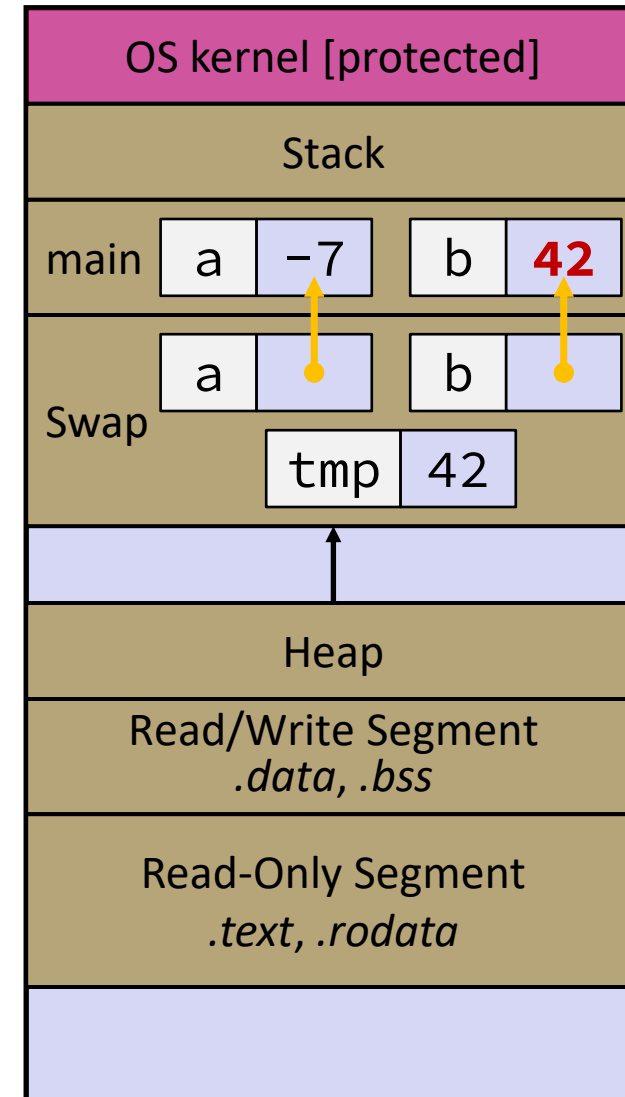
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



Fixed Swap (5/6)

swap.c

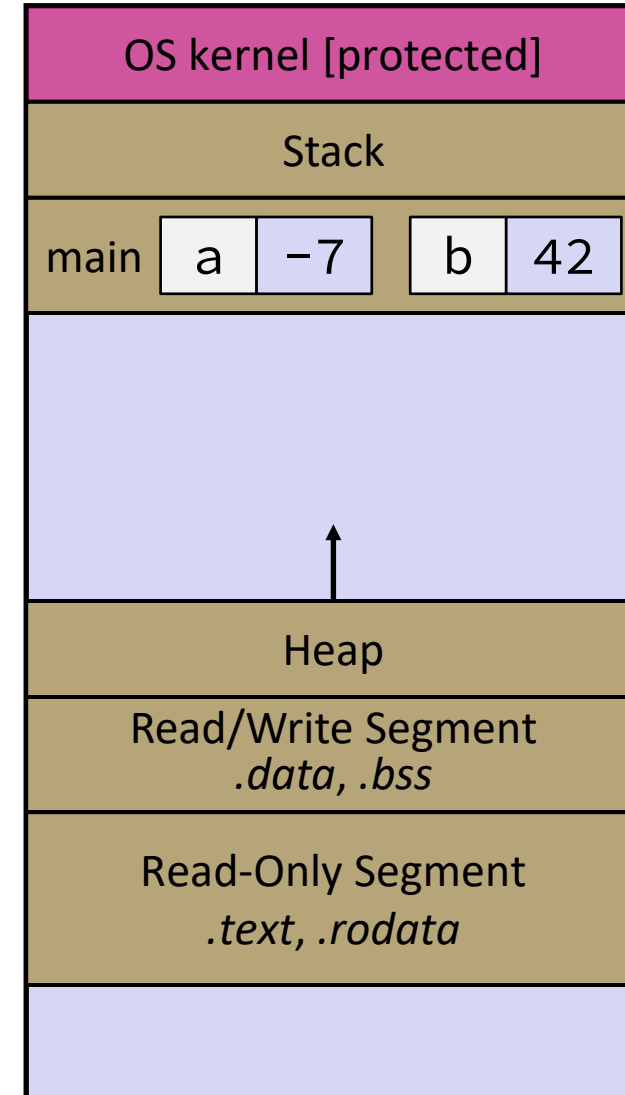
```
void Swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main(int argc, char* argv[]) {  
    int a = 42, b = -7;  
    Swap(&a, &b);  
    ...  
}
```



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/cse333

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

- Of the working ways, which would be preferred?

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

A.

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

C.

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

B.

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

D.

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

E. We're lost...

pollev.com/cse333

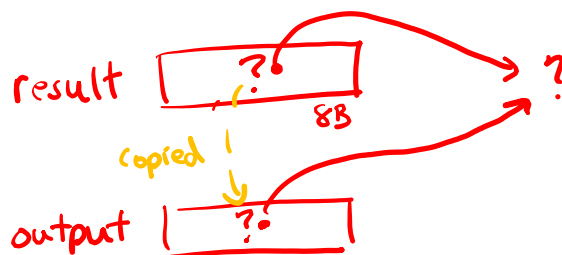
Which of the following are *invalid* ways to invoke `GetGreeting()`?

- Let's investigate:

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

A.

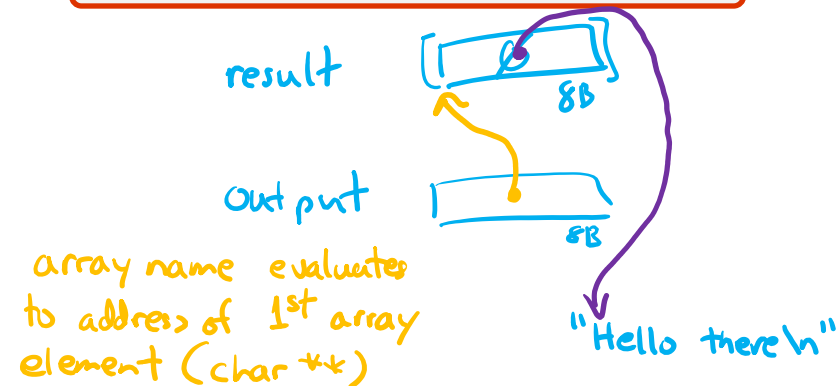
```
char** result; // uninitialized!  
GetGreeting(result);  
printf("%s", *result);
```



dereferencing mystery data
is likely to cause unexpected
behavior (e.g., segfault)

C.

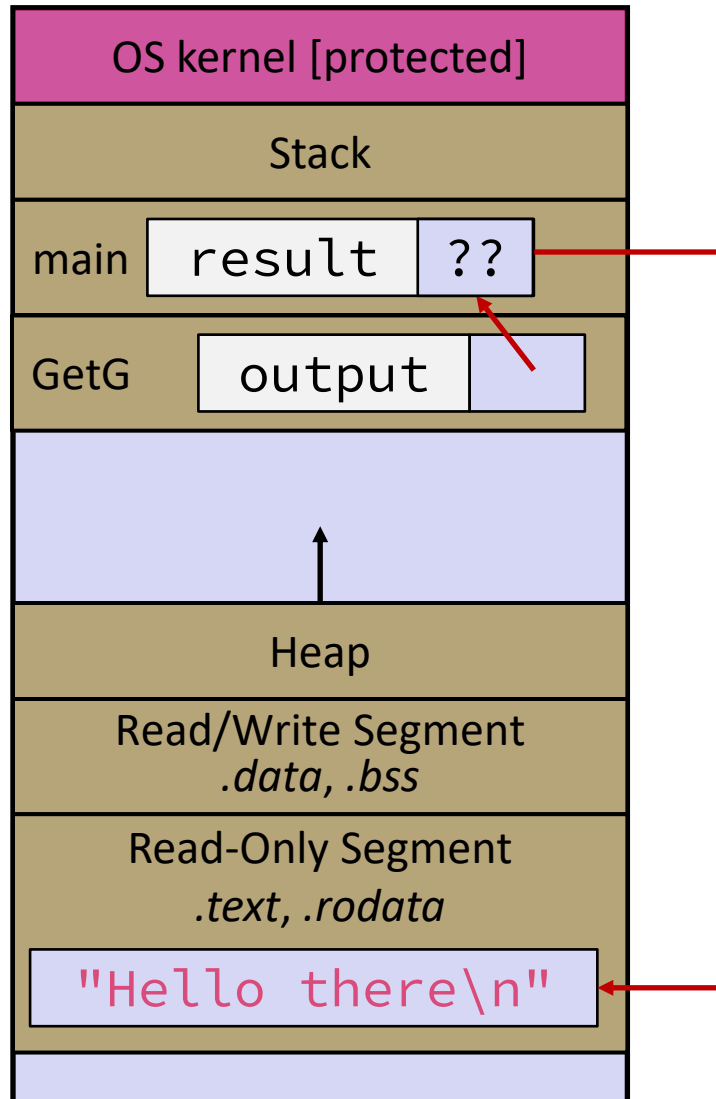
```
char* result[1] = {NULL};  
GetGreeting(result);  
printf("%s", result[0]);
```



dereferencing output lets us
update `result[0]`

Preferred Usage

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



D.

```
void GetGreeting(char** output);

int main(int argc, char* argv[]) {
    char* result;
    GetGreeting(&result);
    printf("%s", result);

    return EXIT_SUCCESS;
}

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

greeting.c

- ✓ Works correctly (unlike A)
- ✓ Minimizes memory usage (unlike B)
- ✓ Intent is clear (unlike C)

Lecture Outline (4/4)

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

Function Pointers

jmp foo → *address* → *PC*

❖ Based on what you know about assembly, what is a function name, really? *label → address*

- Can use pointers that store addresses of functions!

function pointer → *int foo(int);*
function prototype → *int (*fp)(int) = foo;*
*int *fp(int)*

❖ Generic format: `returnType (* name) (type1, ..., typeN)`

- Looks like a function prototype with extra * in front of name

- Why are parentheses around (* name) needed? *to differentiate it from a function prototype*

❖ Using the function: `(dereference *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

// functions that "fit" this function pointer
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

could directly use function name here (square)

map.c

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