

Pointers, Pointers, Pointers...

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano

Ian Hsiao

Ramya Challa

Allen Jung

Kyrie Dowling

Sylvia Wang



Poll Everywhere

pollev.com/cse33320su

About how long did Exercise 1 take?

- A. 0-1 Hours
- B. 1-2 Hours
- C. 2-3 Hours
- D. 3-4 Hours
- E. 4+ Hours
- F. I didn't submit / I prefer not to say

Administrivia: Chat Window

- ❖ We're going to keep this enabled, please stick to asking questions here.
 - Your instructor (Travis) will try to wait until appropriate pauses in the class and recognize people to ask questions then
 - I am not ignoring you, I just want to avoid breaking up the 'flow' of lecture too much
 - Please keep asking questions, I don't mean to discourage them with this!

Administrivia

- ❖ Exercise 2 out today and due Monday morning 10:30 am
- ❖ Exercise grading
 - We will do our best to keep up
 - Things to watch for:
 - Input sanity check
 - No functional abstraction (single blob of code)
 - Formatting funnies (*e.g.* tabs instead of spaces)
 - Regrade requests:
 - Will open 24 hours after grades are published and will be open for 24 hours.

Administrivia

- ❖ Pre-Quarter survey up on canvas. Due Tonight @11:59 pm
 - Answers are anonymous. Will help us figure out how to make course as great as possible
- ❖ Homework 0 due Monday
 - Logistics and infrastructure for projects
 - clint 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
- ❖ Homework 1 out today or tomorrow, due July 9th
 - 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

Administrivia

- ❖ Documentation:
 - man pages, books
 - Reference websites: cplusplus.org, man7.org, gcc.gnu.org, etc.
- ❖ Folklore:
 - Google-ing, Stack Overflow, that rando in lab
- ❖ Tradeoffs? Relative strengths & weaknesses?

Lecture Outline

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

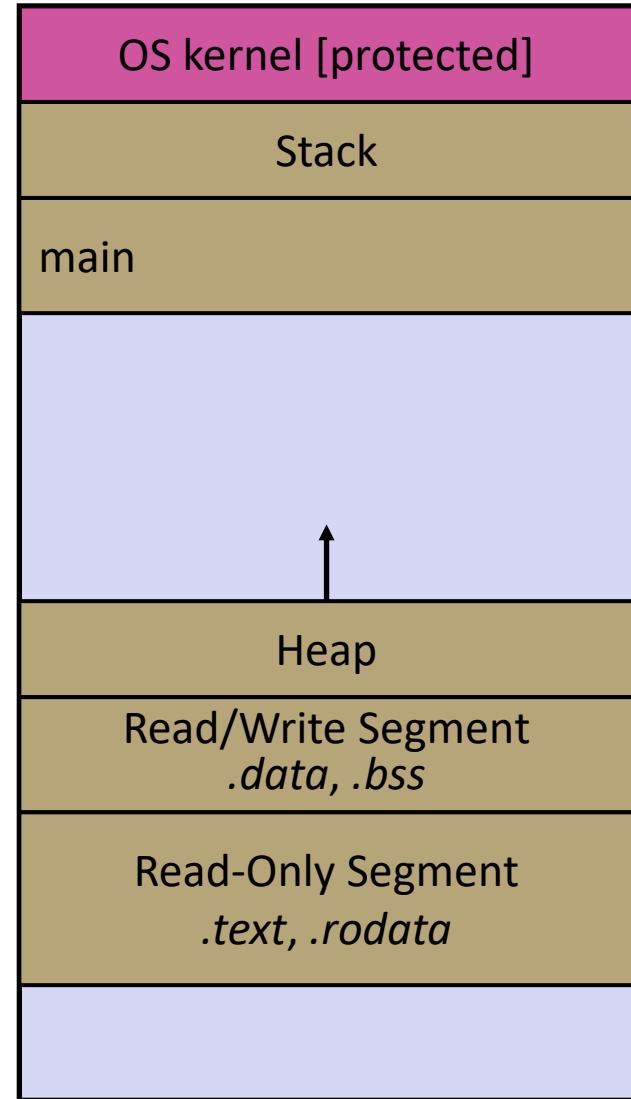
```
void swap(int a, int b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main(int argc, char** argv) {  
    int a = 42, b = -7;  
    swap(a, b);  
    ...  
}
```

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);  
    ...  
}
```

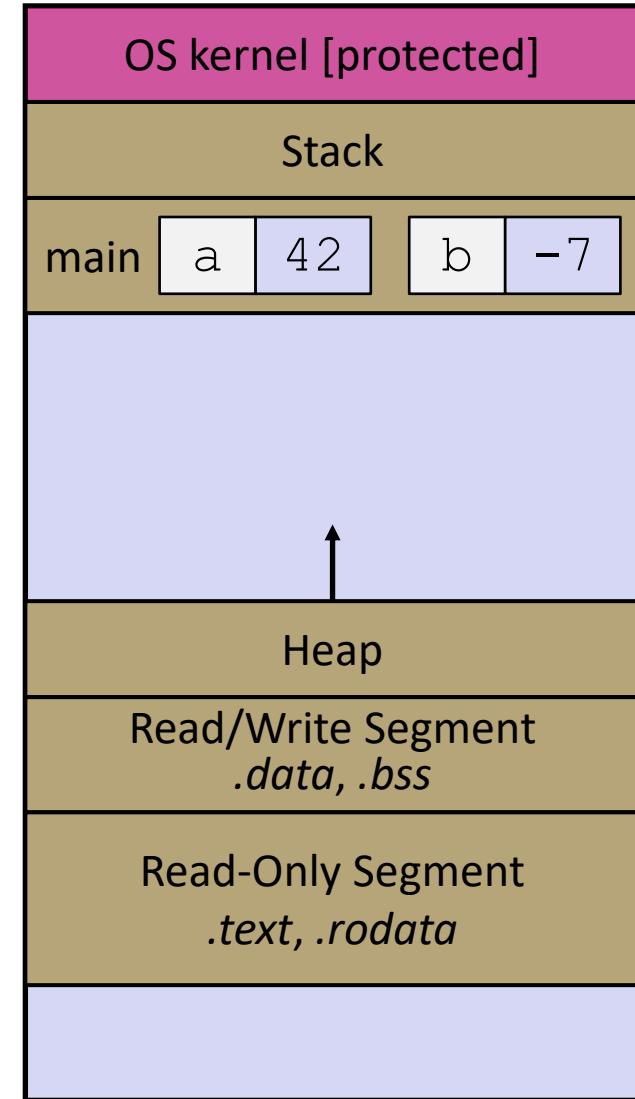


Broken Swap

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);
    ...
}
```

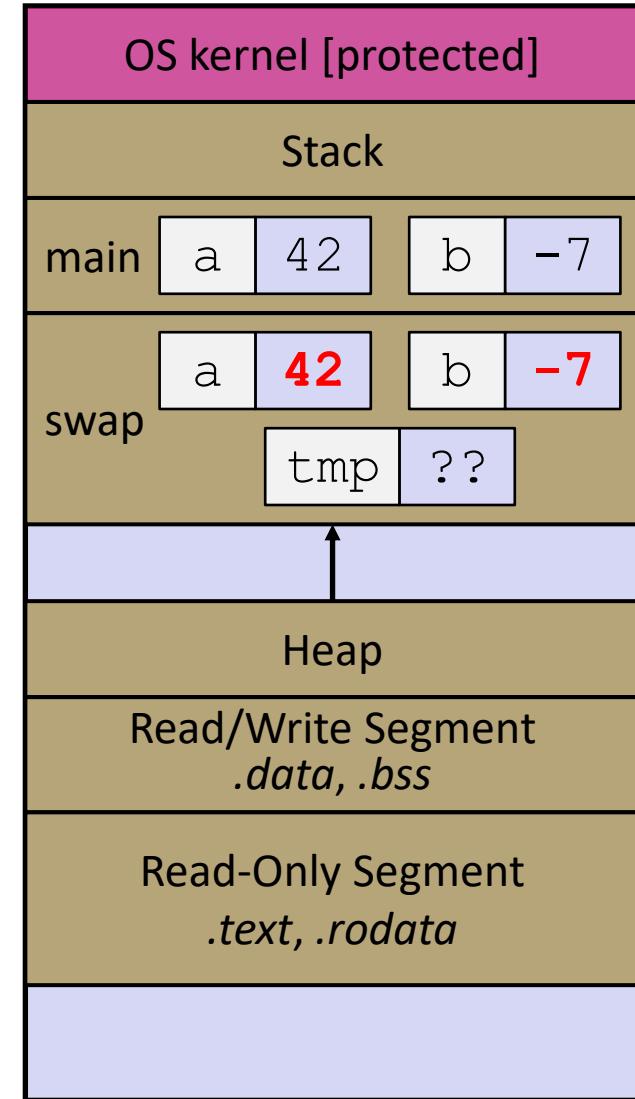


Broken Swap

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);
    ...
}
```

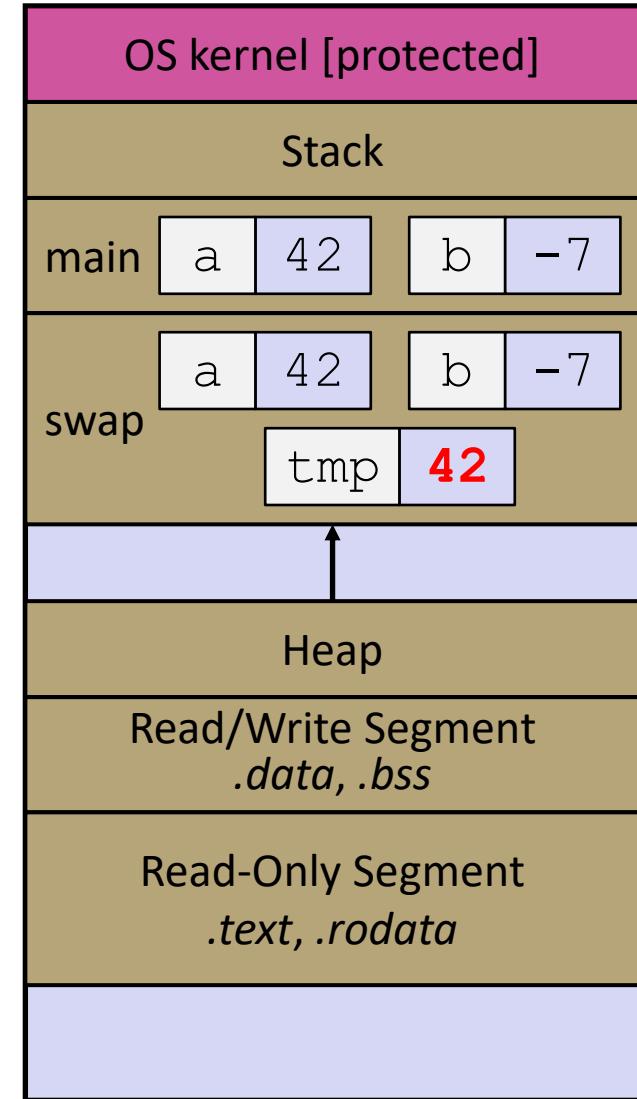


Broken Swap

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);
    ...
}
```

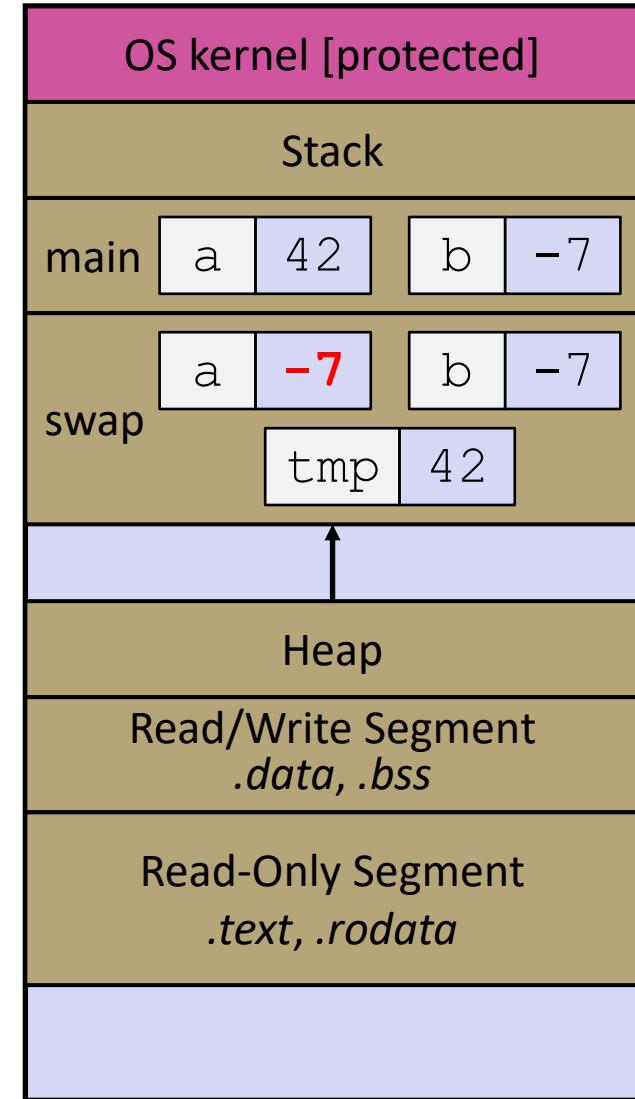


Broken Swap

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);
    ...
}
```

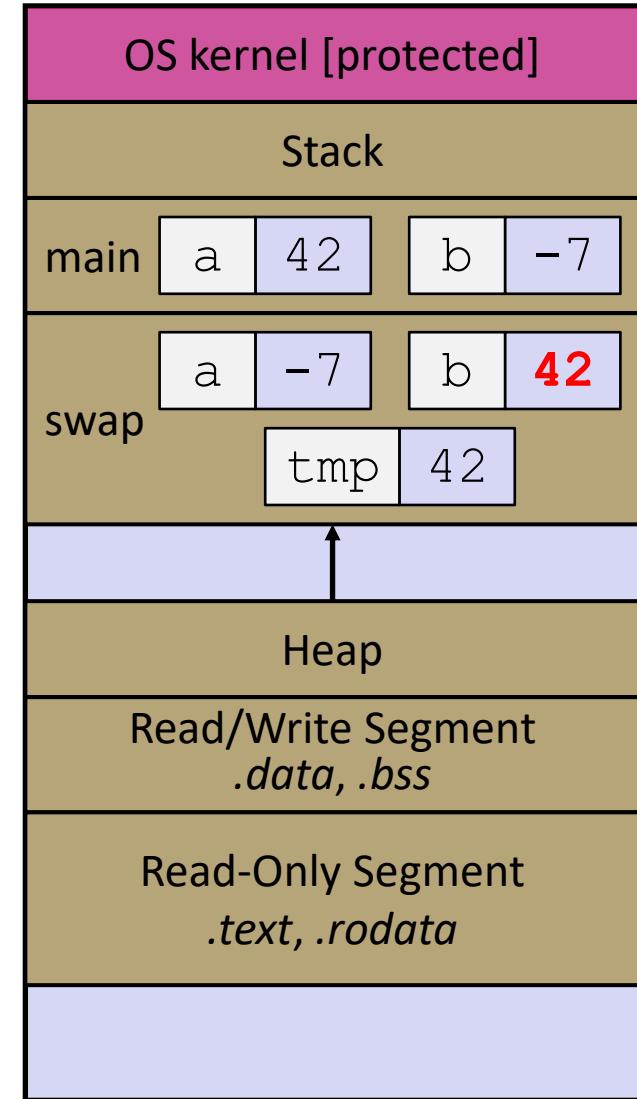


Broken Swap

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);
    ...
}
```

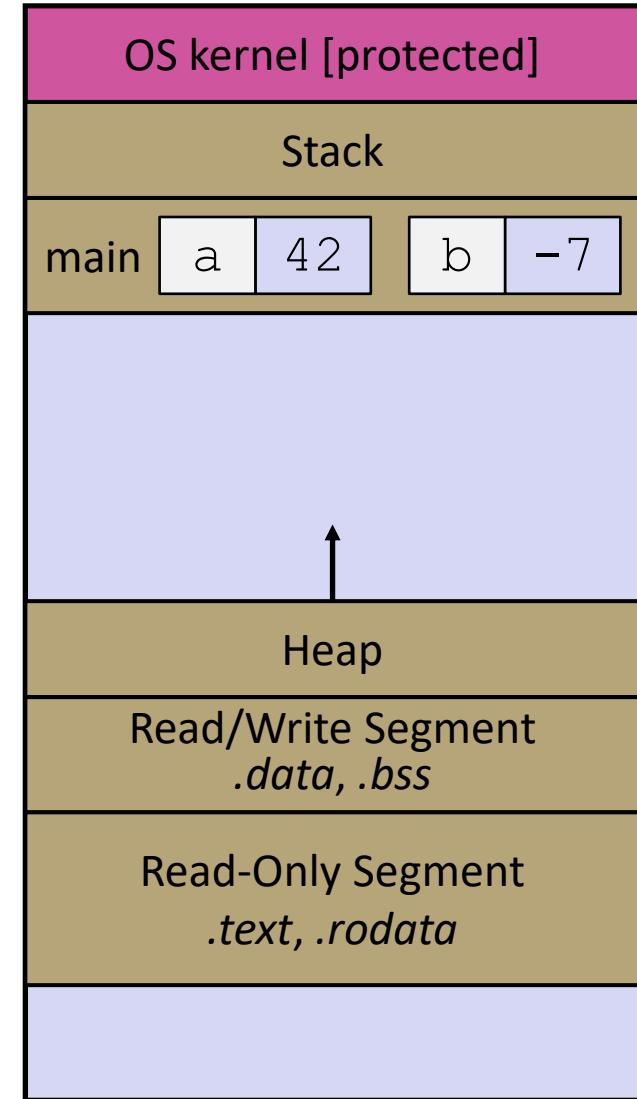


Broken Swap

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);  
    ...  
}
```

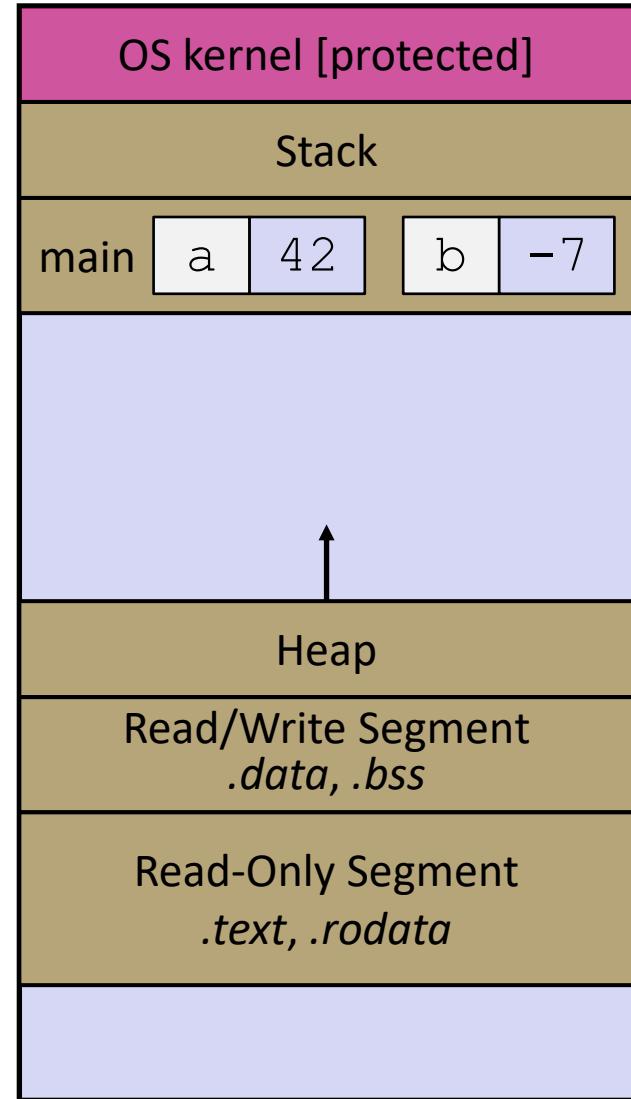
Fixed Swap

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);
    ...
}
```

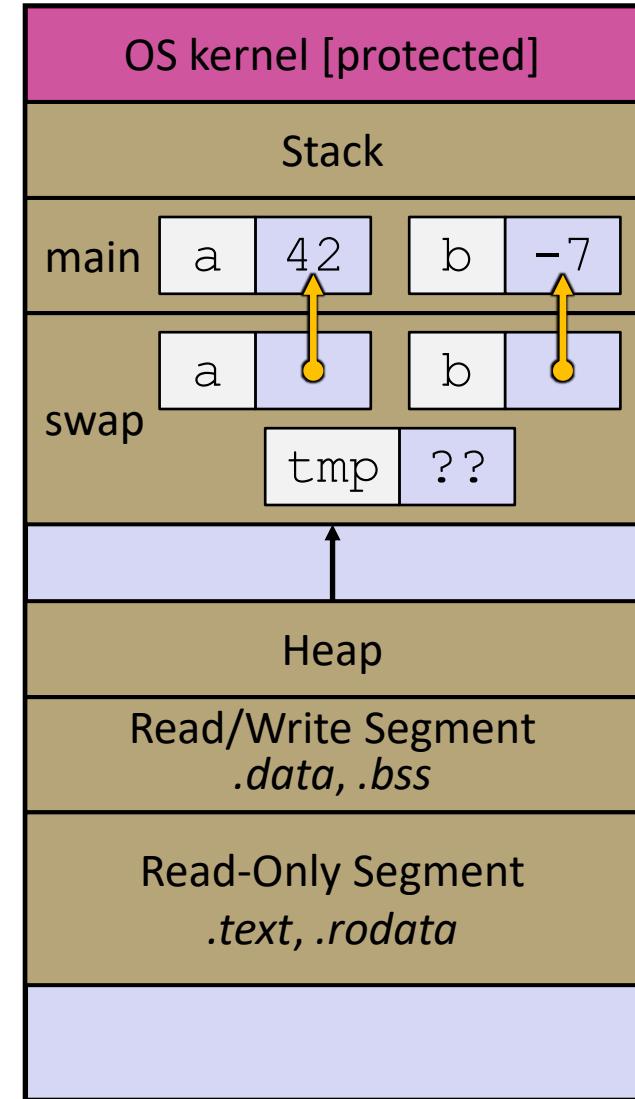


Fixed Swap

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);
    ...
}
```

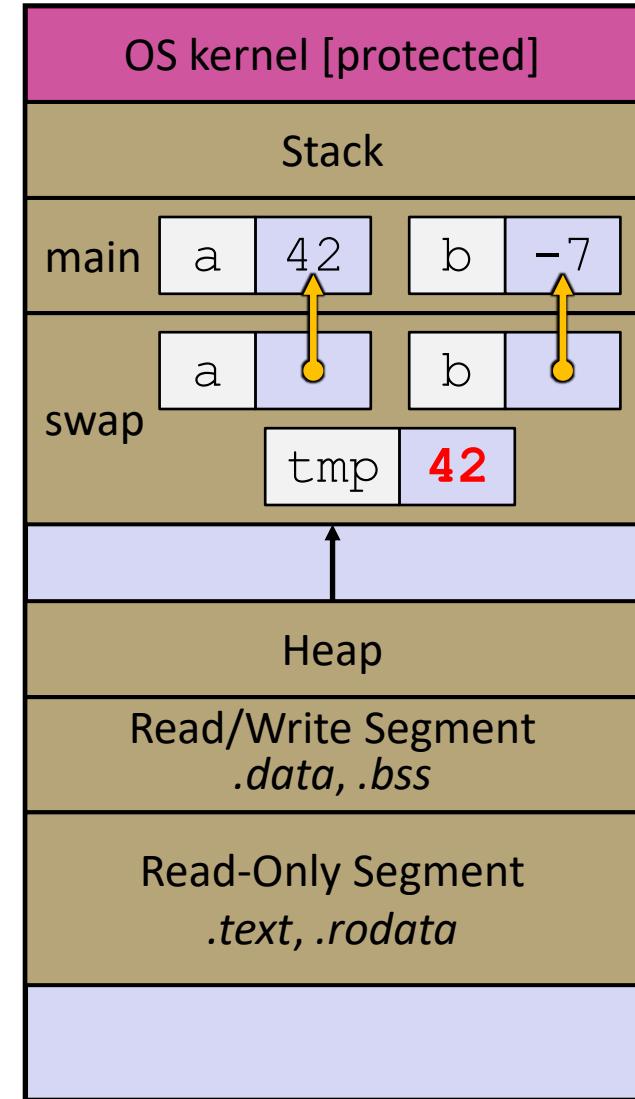


Fixed Swap

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);
    ...
}
```

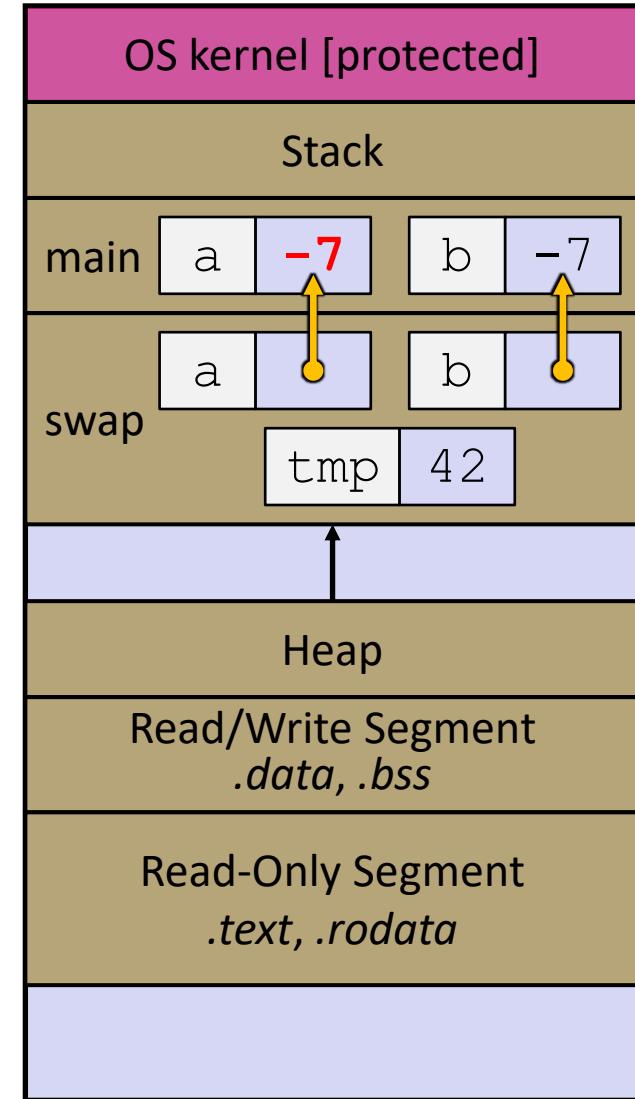


Fixed Swap

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);
    ...
}
```

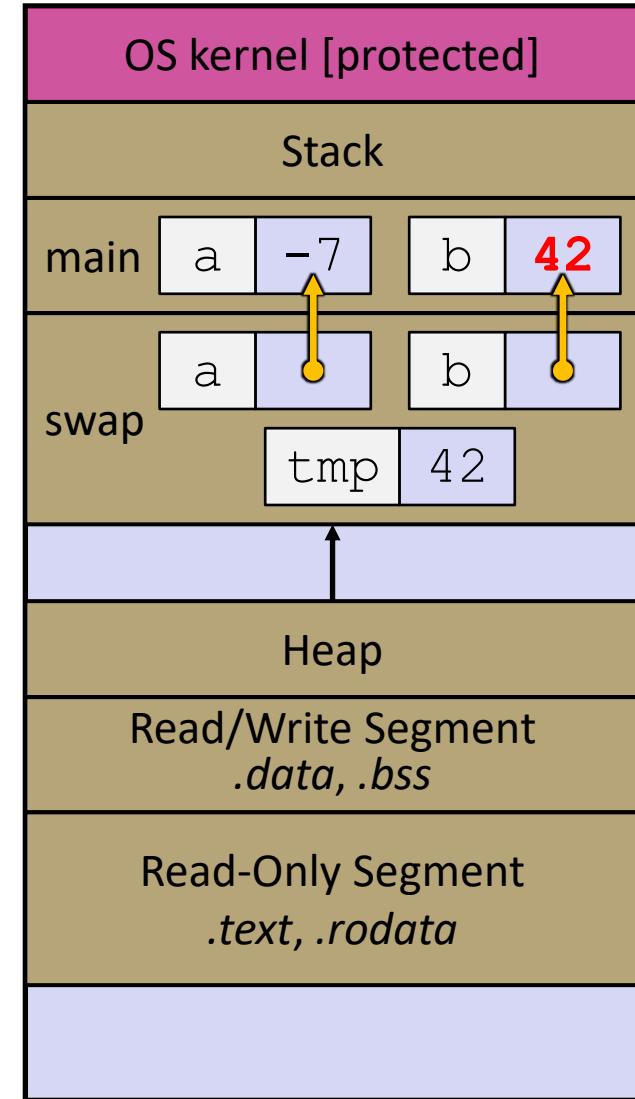


Fixed Swap

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);
    ...
}
```

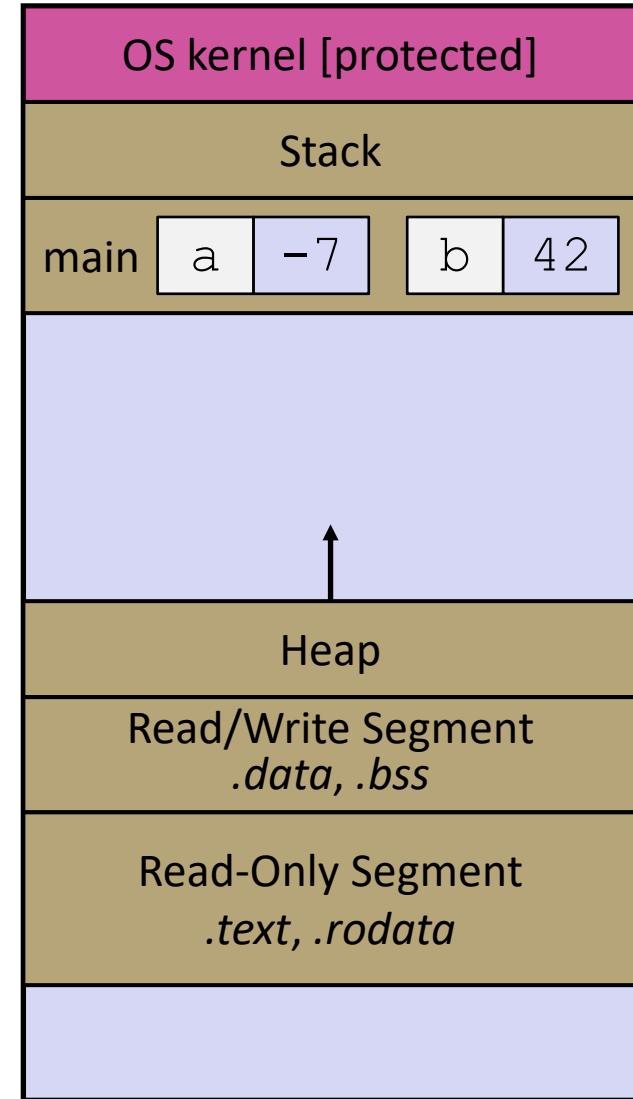


Fixed Swap

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);
    ...
}
```



Lecture Outline

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

Output Parameters

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

- ❖ Output parameter
 - a parameter that we use to store an output of a function.
 - can modify something in the caller's scope by dereferencing the pointer parameter.
 - Useful if you want to have multiple returns
- ❖ Caller passes in a pointer to the type they want.
output2.c
 - If caller wants an **int**,
 - Declare an int **variable**
 - Pass in **&variable**
- ❖ Function dereferences the pointer parameter, and sets output.

```
void get333(int* output) {  
    *output = 333;  
}  
  
int main(int argc, char** argv) {  
    int num;  
    get333(&num);  
}
```



pollev.com/cse33320su

What would be the best way to invoke generateString()?

genstr.c

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

int main(int argc, char** argv) {
    /* ??? */
    return EXIT_SUCCESS;
}
// ...
```

A.

```
char** result;
generateString(result);
printf(*result);
```

C.

```
char* str;
char** result = &str;
generateString(result);
printf(str);
```

B.

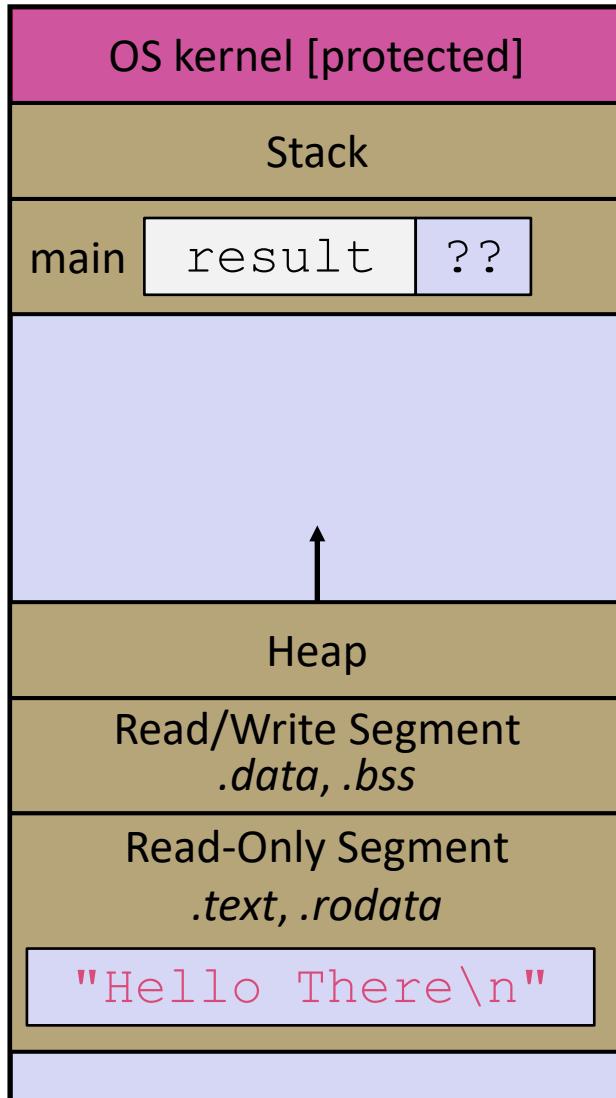
```
char* result[1] = {NULL};
generateString(result);
printf(result[0]);
```

D.

```
char* result;
generateString(&result);
printf(result);
```

E. We're lost...

Answer



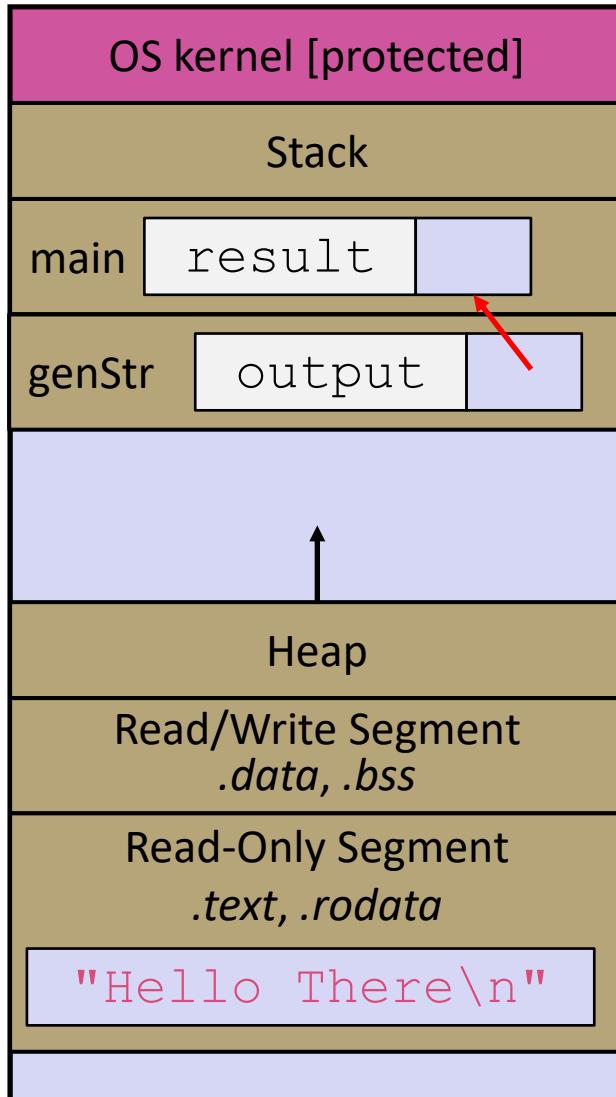
D.

genstr.c

```
void generateString(char**  
output);  
  
int main(int argc, char** argv) {  
    char* result;  
    generateString(&result);  
    printf(result);  
  
    return EXIT_SUCCESS;  
}  
// ...
```

- ❖ Works correctly
- ❖ Minimizes memory usage
- ❖ Short and simple

Answer



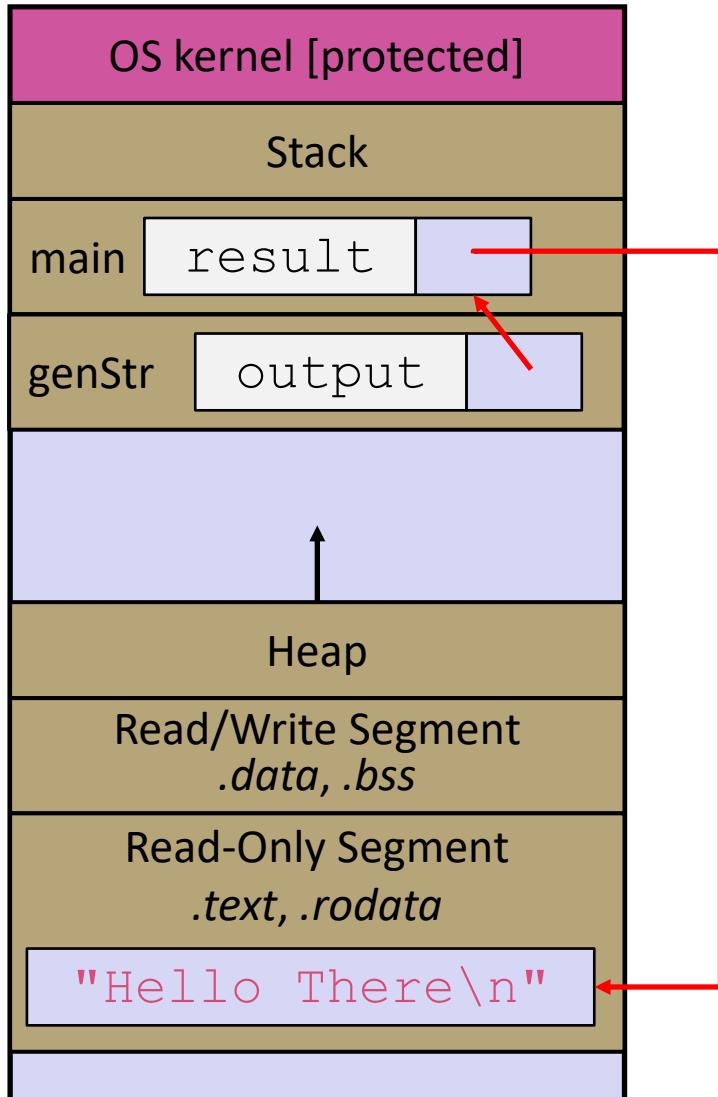
D.

genstr.c

```
void generateString(char**  
output);  
  
int main(int argc, char** argv) {  
    char* result;  
    generateString(&result);  
    printf(result);  
  
    return EXIT_SUCCESS;  
}  
// ...
```

- ❖ Works correctly
- ❖ Minimizes memory usage
- ❖ Short and simple

Answer



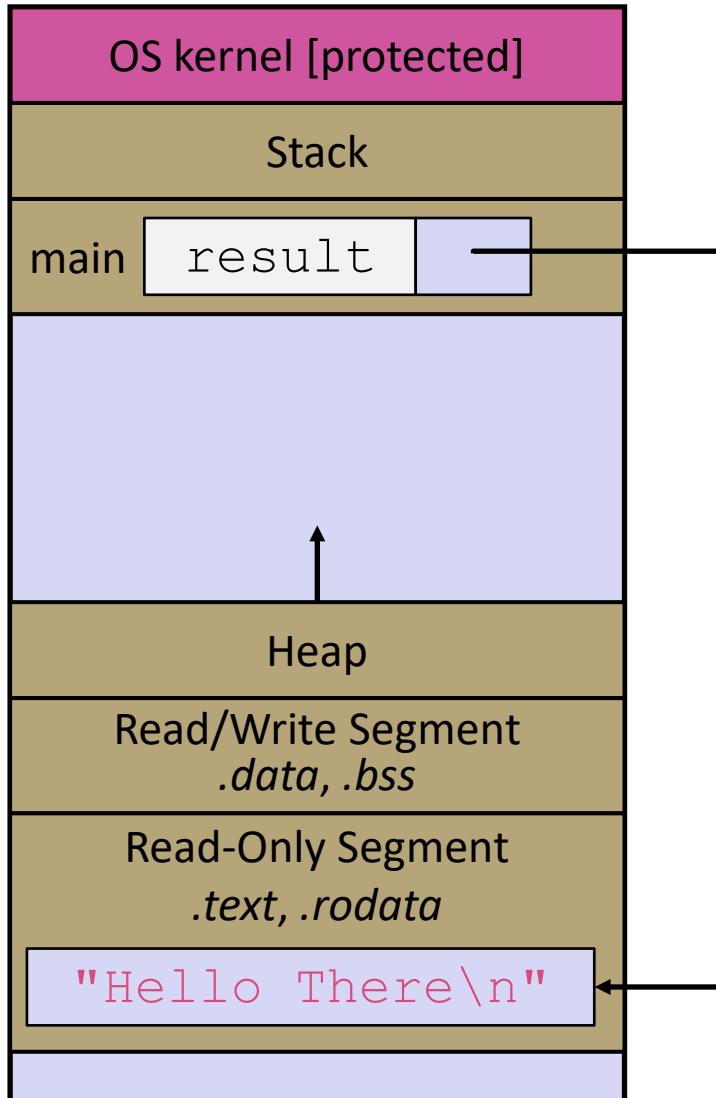
D.

genstr.c

```
void generateString(char**  
output);  
  
int main(int argc, char** argv) {  
    char* result;  
    generateString(&result);  
    printf(result);  
  
    return EXIT_SUCCESS;  
}  
// ...
```

- ❖ Works correctly
- ❖ Minimizes memory usage
- ❖ Short and simple

Answer



D.

genstr.c

```
void generateString(char**  
output);  
  
int main(int argc, char** argv) {  
    char* result;  
    generateString(&result);  
    printf(result);  
  
    return EXIT_SUCCESS;  
}  
// ...
```

- ❖ Works correctly
- ❖ Minimizes memory usage
- ❖ Short and simple

Lecture Outline

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

Polling Question

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;  
}
```

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

- Vote at
<http://PollEv.com/cse33320su>

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

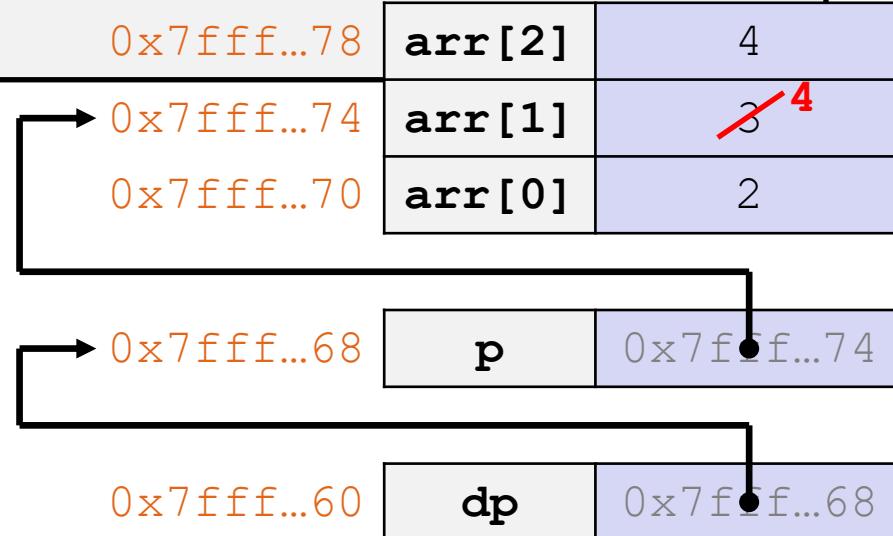
0x7fff...60	dp	0x7fff...68
-------------	-----------	-------------

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

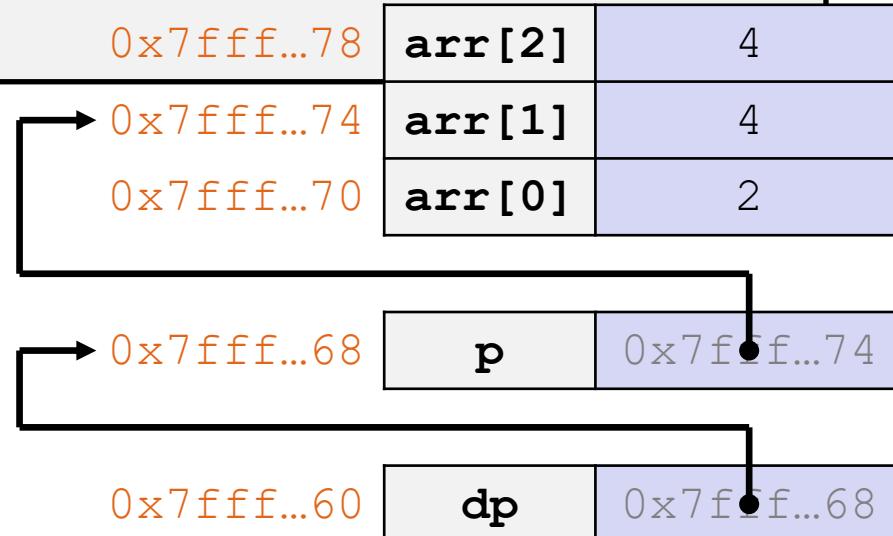


Practice Solution

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

0x7fff...78	arr[2]	4
0x7fff...74	arr[1]	4
0x7fff...70	arr[0]	2

0x7fff...68	p	0x7fff...78
-------------	----------	-------------

0x7fff...60	dp	0x7fff...68
-------------	-----------	-------------

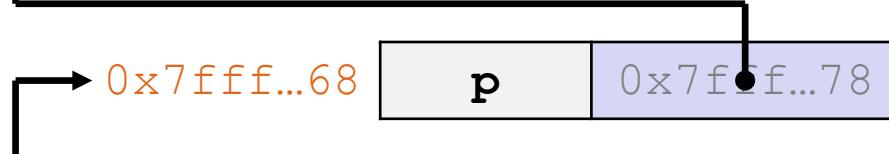
Practice Solution

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;  
}
```

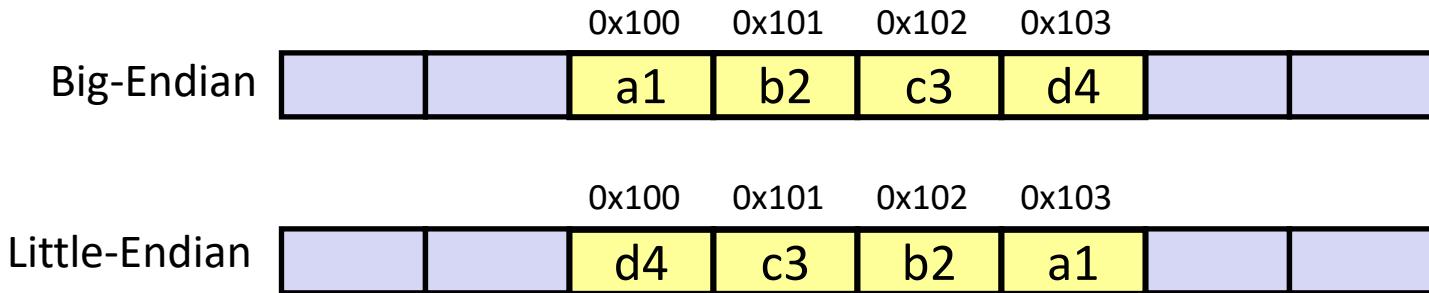
address name value

0x7fff...78	arr[2]	5
0x7fff...74	arr[1]	4
0x7fff...70	arr[0]	2



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



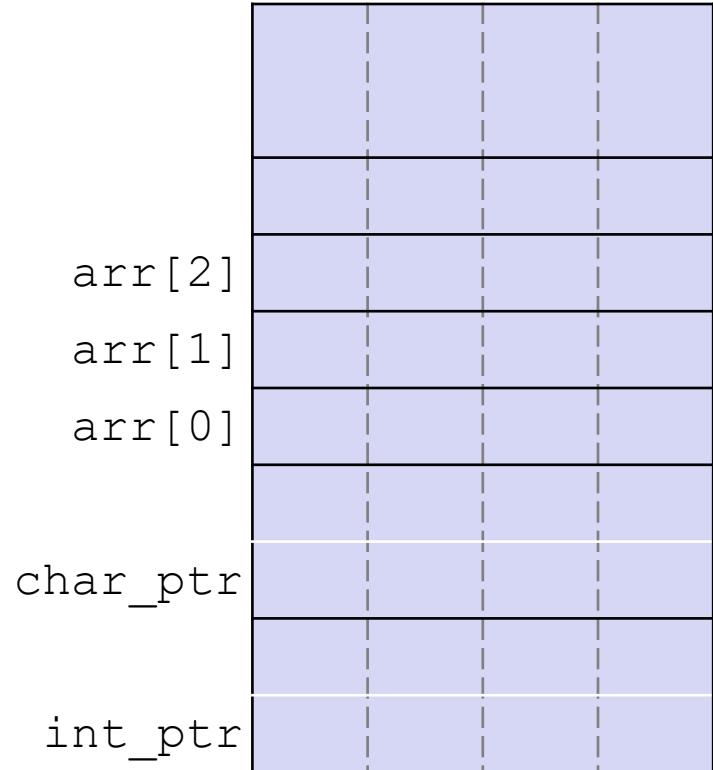
Pointer Arithmetic Example

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

```
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

Note: Arrow points to *next* instruction.

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;

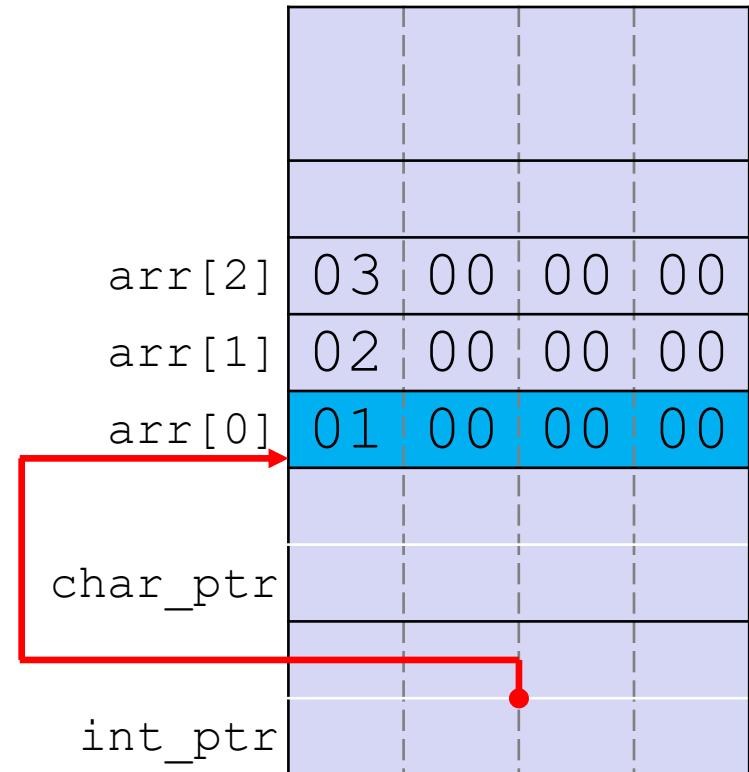
    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

Stack
(assume x86-64)



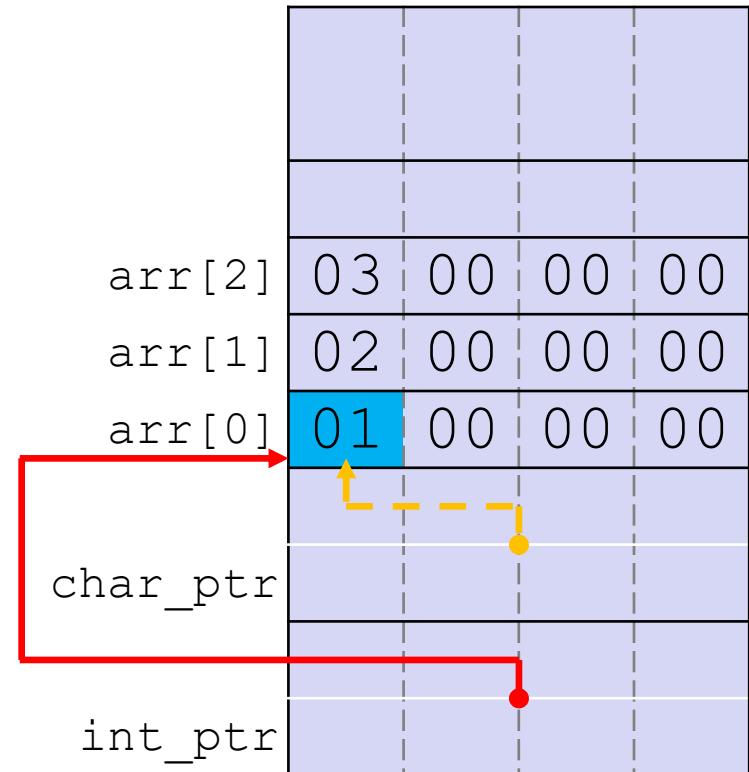
Pointer Arithmetic Example

```
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

Note: Arrow points to *next* instruction.

Stack
(assume x86-64)



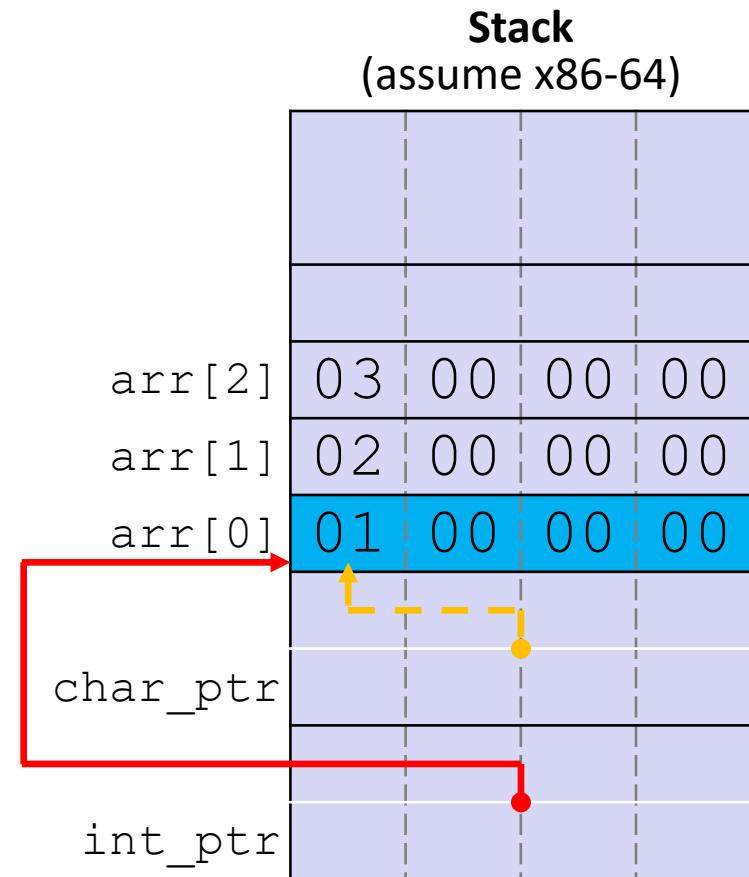
Pointer Arithmetic Example

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: 0x0x7fffffffde010  
*int_ptr: 1
```



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;

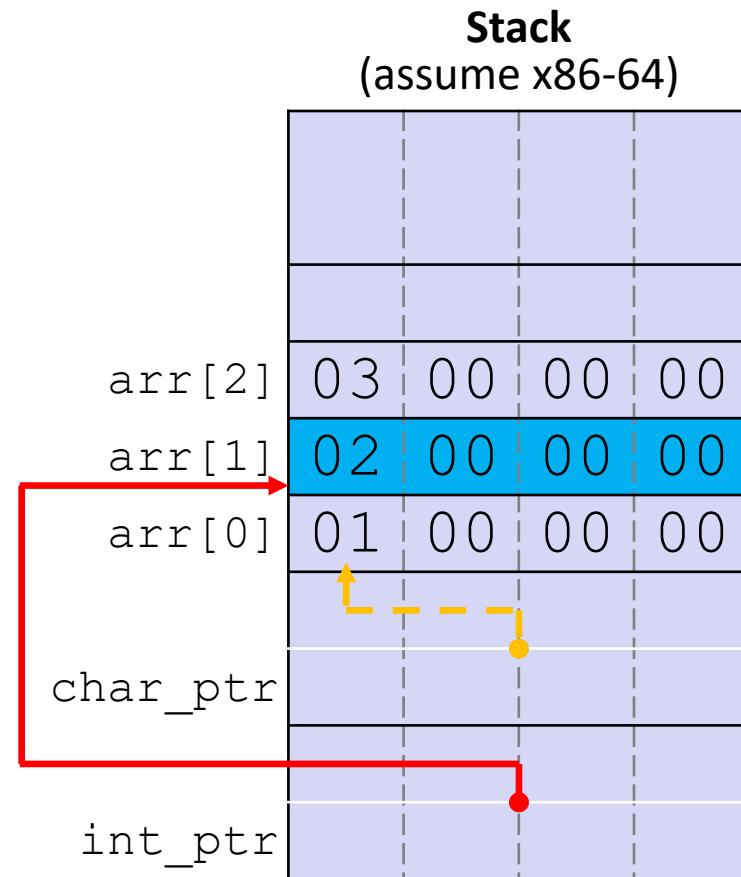
    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return EXIT_SUCCESS;
}
```

pointerarithmetic.c

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



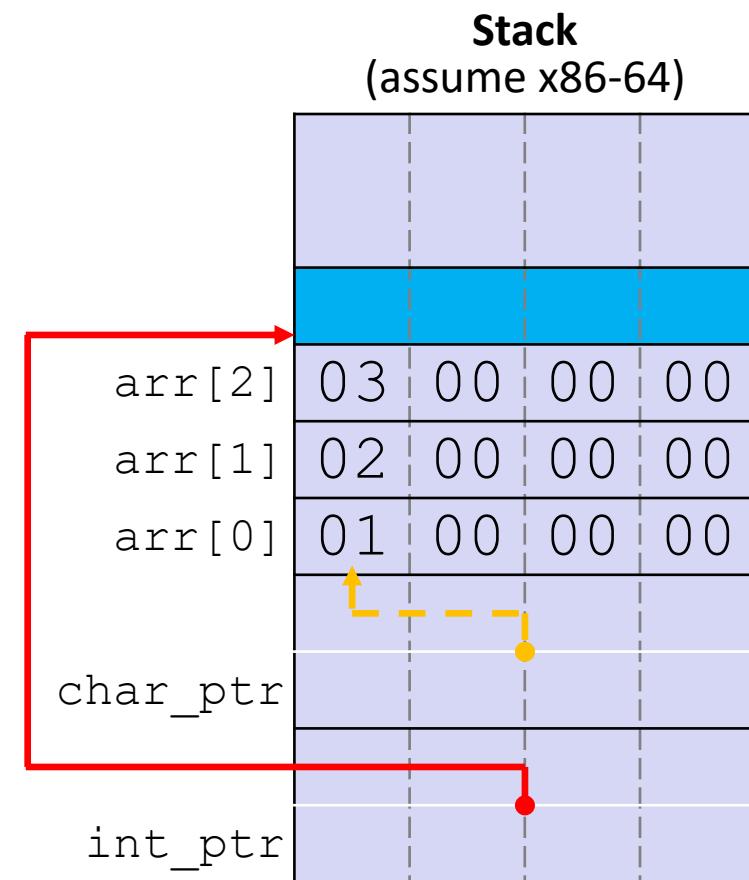
Pointer Arithmetic Example

```
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: ???

Note: Arrow points to *next* instruction.



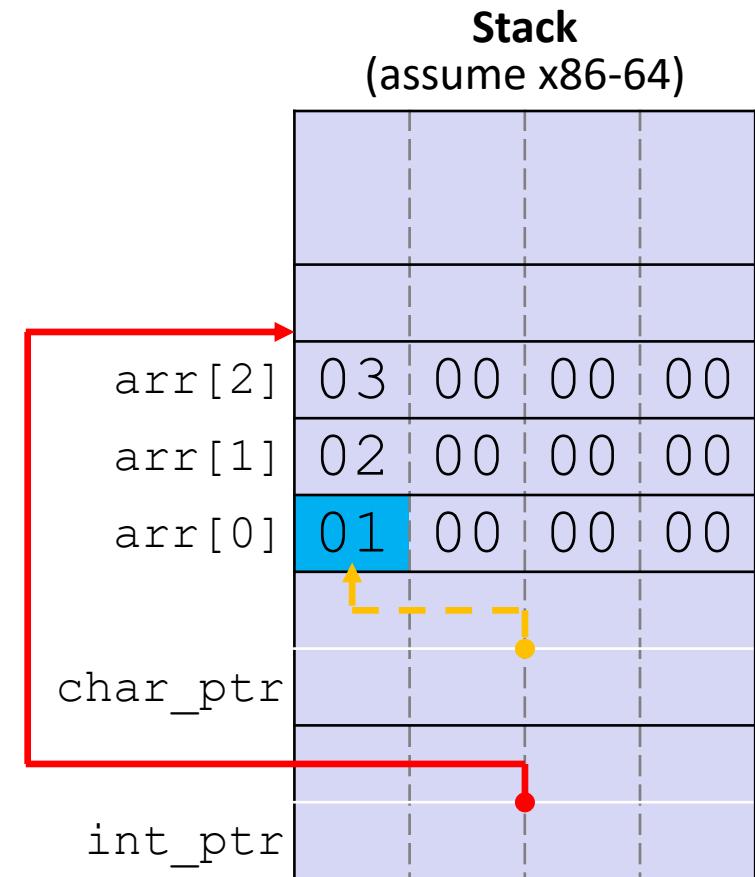
Pointer Arithmetic Example

```
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
```

Note: Arrow points to *next* instruction.



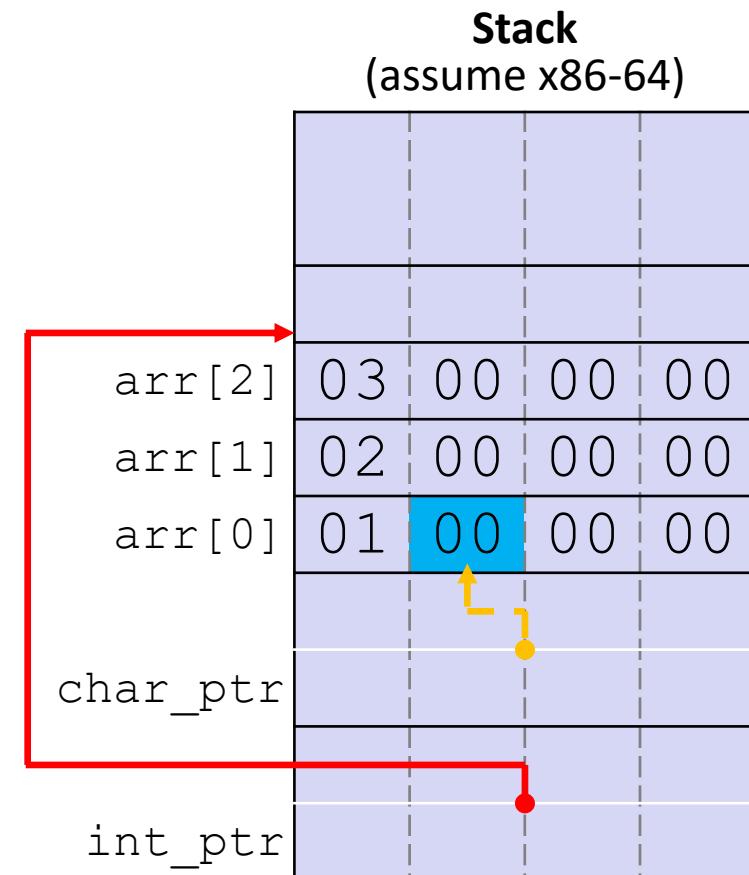
Pointer Arithmetic Example

```
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: 0x0x7fffffffde01**1**
***char_ptr:** 0

Note: Arrow points to *next* instruction.



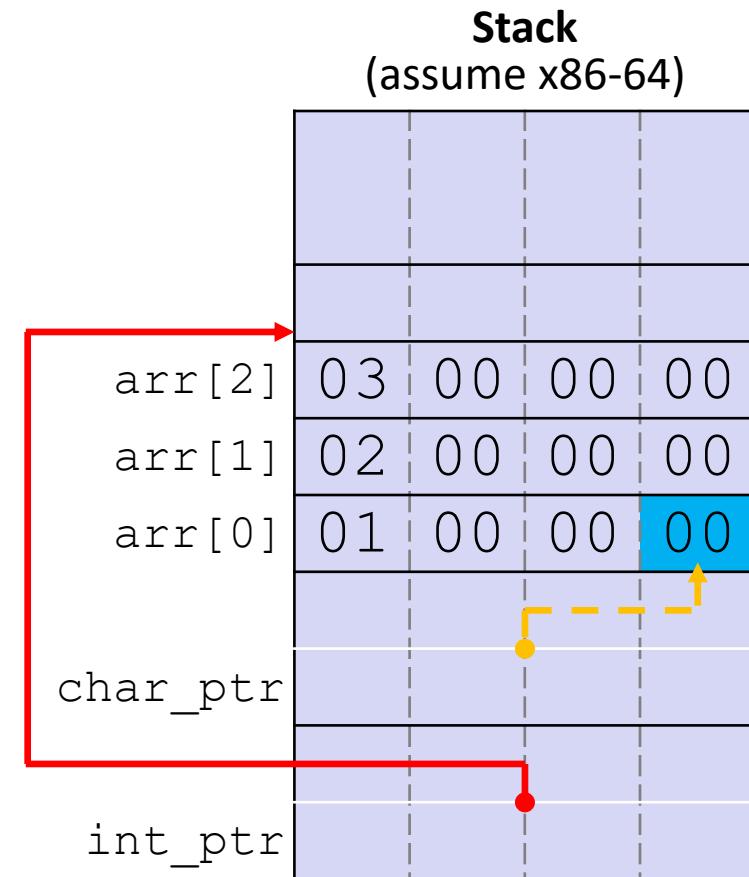
Pointer Arithmetic Example

```
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: 0x0x7fffffffde01**3**
***char_ptr:** 0

Note: Arrow points to *next* instruction.



Lecture Outline

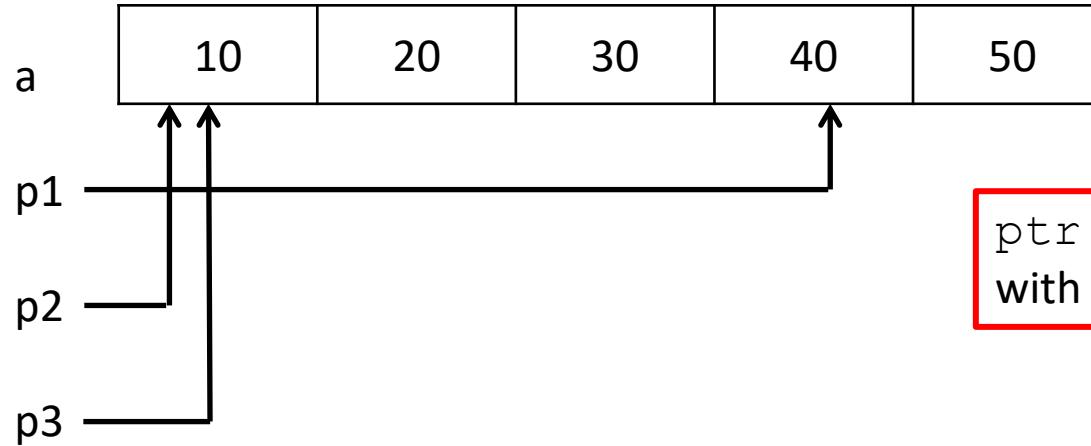
- ❖ Pointers as Parameters
 - Pointers as Output Parameters
- ❖ Pointer Arithmetic
- ❖ **Pointers and Arrays**
- ❖ Function Pointers

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
```

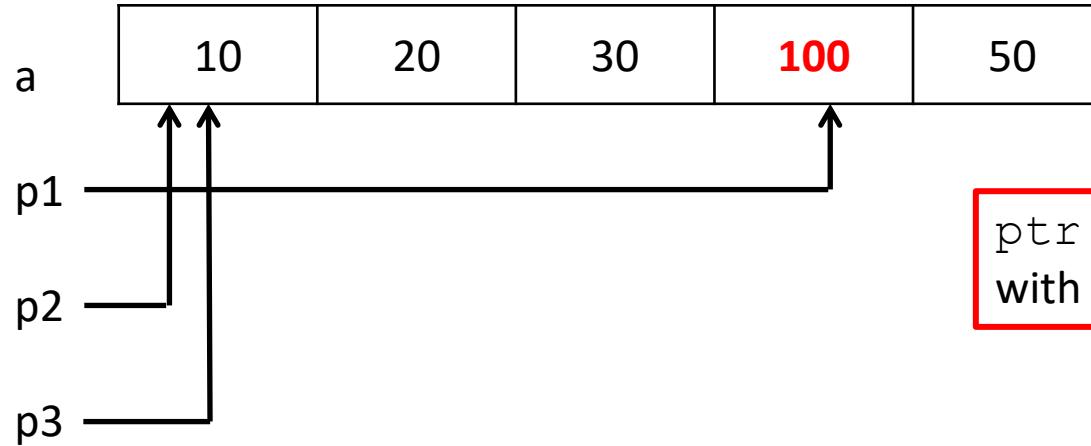
Pointers and Arrays - Trace



```
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

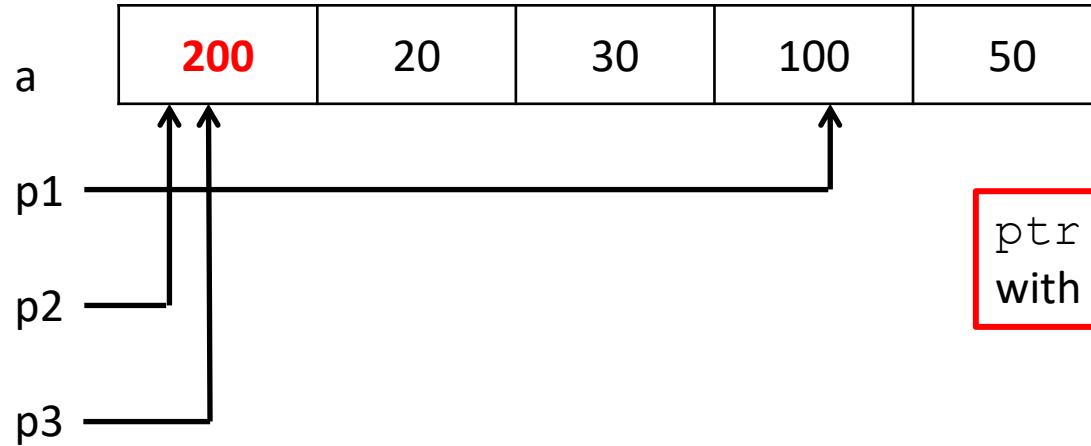
*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;      // final: 200, 400, 500, 100, 300
```

Pointers and Arrays - Trace



```
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
```

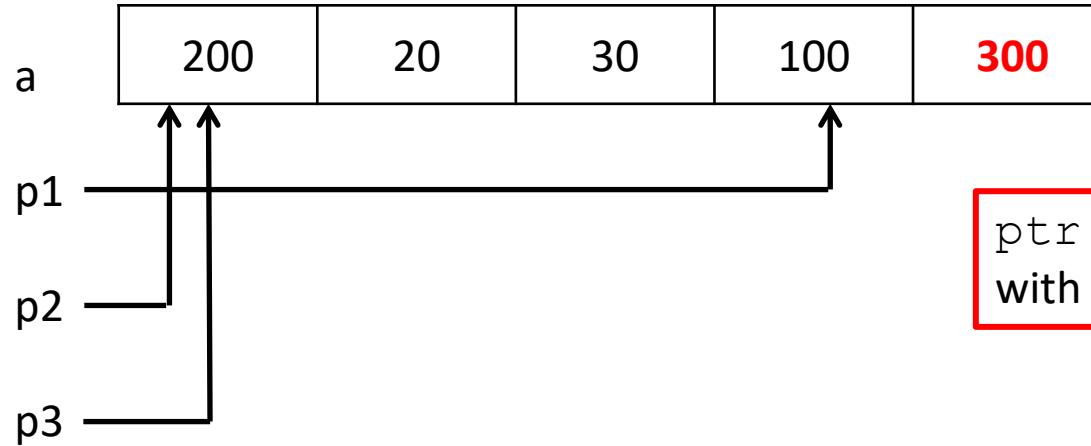
Pointers and Arrays - Trace



```
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
```

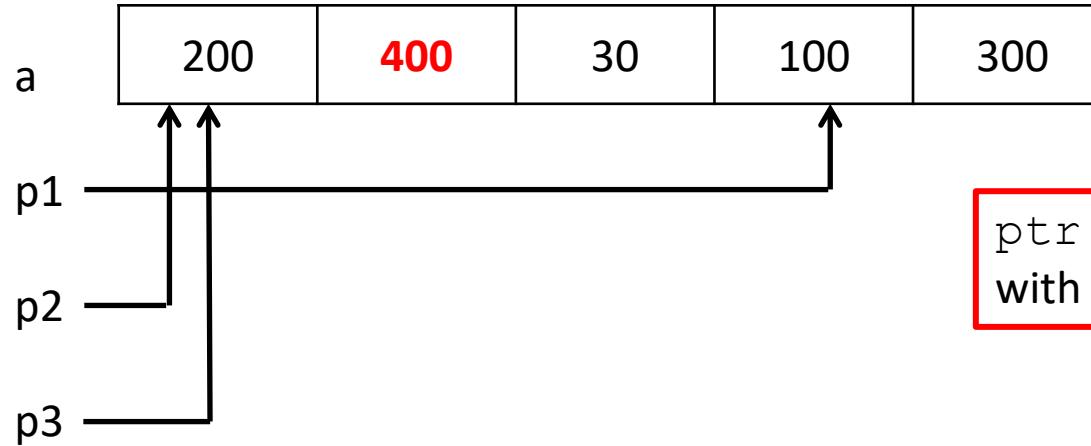
Pointers and Arrays - Trace



ptr[i] is * (ptr + i)
with pointer arithmetic

```
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
```

Pointers and Arrays - Trace

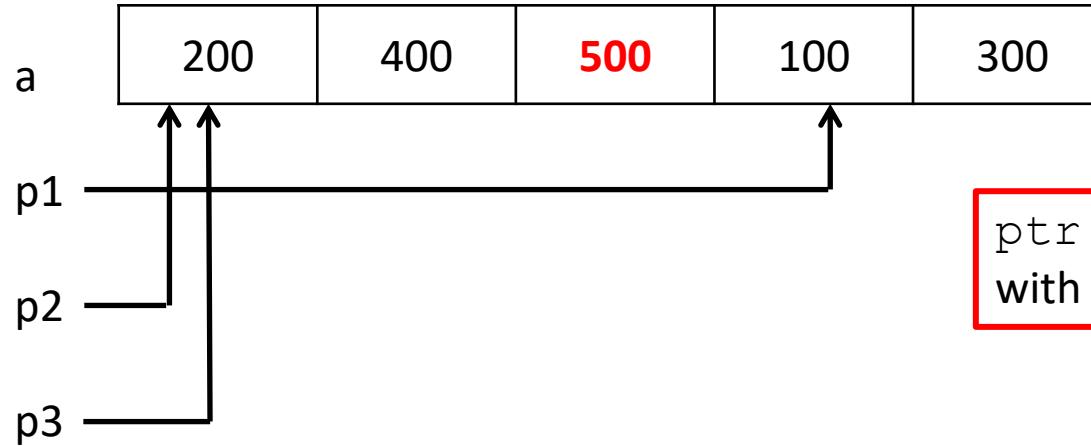


`ptr[i]` is $*(\text{ptr} + i)$
with pointer arithmetic

```
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
```

Pointers and Arrays - Trace



ptr[i] is * (ptr+i)
with pointer arithmetic

```
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
 - OK to use whichever best helps the reader

This code: Promoted to a ptr secretly

```
void f(int a[]);  
  
int main( ... ) {  
    int a[5]; Array declaration  
    ...  
    f(a);  
    return EXIT_SUCCESS;  
}
```

Equivalent to:

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

Lecture Outline

- ❖ Pointers as Parameters
 - Pointers as Output Parameters
- ❖ Pointers & Pointer Arithmetic
- ❖ 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}; funcptr definition
    int (* op)(int n); // function pointer called 'op'
    op = square; // function name returns addr (like array)
    map(arr, LEN, op); funcptr assignment
    ...
}
```

Function Pointer Example

- ❖ C allows you to omit & on a function parameter and omit * when calling pointed-to function; both assumed implicitly.

```
#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