

# Pointers, Pointers, Pointers

CSE 333 Autumn 2020

**Instructor:** Hal Perkins

**Teaching Assistants:**

Rehaan Bhimani

Ramya Challa

Eric Chan

Mengqi Chen

Ian Hsiao

Pat Kosakanchit

Arjun Singh

Guramrit Singh

Sylvia Wang

Yifan Xu

Robin Yang

Velocity Yu

# Administrivia

- ❖ Exercise 2 out this morning; due Wed. 10 am
  
- ❖ Homework 1 out now, due a week from this Thursday
  - 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 repo has unpushed changes
      - If git drops you into vi(m), `:q` to quit or `:wq` if you want to save changes
  - Start early and make steady progress

# More Administrivia

- ❖ Homework 0 due tonight, 11 pm
  - Logistics and infrastructure for projects
    - `clint` and `valgrind` are useful for exercises, too
  - Git: `add/commit/push`, then tag with `hw0-final`, then push tag
    - Then clone your repo somewhere totally different and do `git checkout hw0-final` and verify that all is well
      - Leave yourself enough time before 11 pm to do this and fix any problems
  - All exercises/hw **must** be done on Allen School Centos 8 / gcc 9 machines
    - Not the VM from last year or summer!

# Yet More Administrivia

- ❖ Exercise grading – Gradescope abuse
  - Score is an overall evaluation: 3/2/1/0 = superior / good / marginal / not sufficient for credit
  - Then additional  $\pm 0$  rubric items as needed
    - These are a quick way of communicating “why” – reasons for deductions or comments about your solution
    - Allows us to be more consistent in feedback
    - The  $\pm 0$  “score” is just because that’s how we have to use Gradescope to handle feedback notes – it does not contribute to “the points”
  
- ❖ We should have ex0 graded in a day or two, then ex1 shortly after that, then hope to stay current after that

# Lecture Outline

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

# 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 0;
}
```

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 0;
}
```

address	name	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
&x	<b>x</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 0;
}
```

address

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

&arr[2]	<b>arr[2]</b>	4
&arr[1]	<b>arr[1]</b>	3
&arr[0]	<b>arr[0]</b>	2
&p	<b>p</b>	&arr[1]
&x	<b>x</b>	1



# 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 0;
}
```

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

0x7fff...78	<b>arr[2]</b>	4
0x7fff...74	<b>arr[1]</b>	3
0x7fff...70	<b>arr[0]</b>	2
0x7fff...68	<b>p</b>	0x7fff...74
0x7fff...64	<b>x</b>	1

# 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!
- ❖ Valid pointer arithmetic:
  - Add/subtract an integer to a pointer
  - Subtract two pointers (within stack frame or malloc block)
  - Compare pointers (`<`, `<=`, `==`, `!=`, `>`, `>=`), including `NULL`

# Practice 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 0;
}
```

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



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

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.

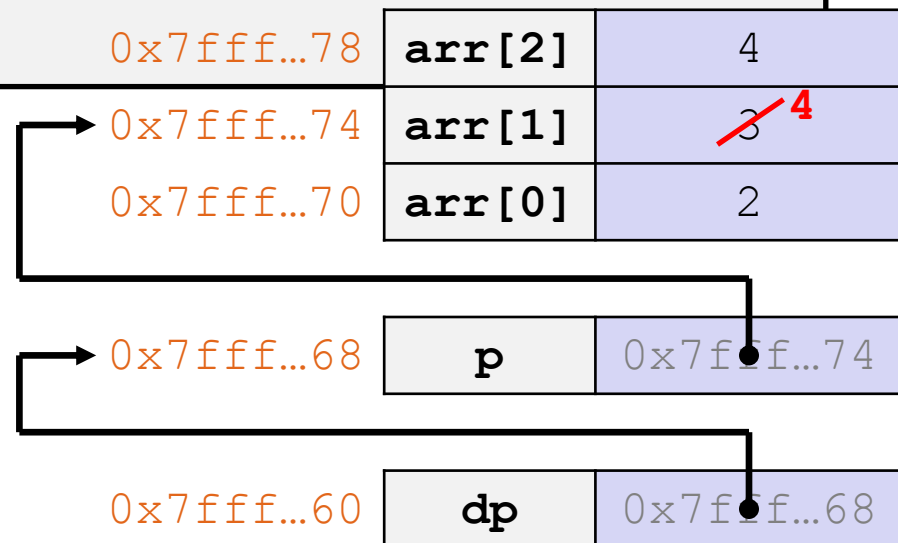
boxarrow2.c

```
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 0;
}
```

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



# Practice Solution

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

boxarrow2.c

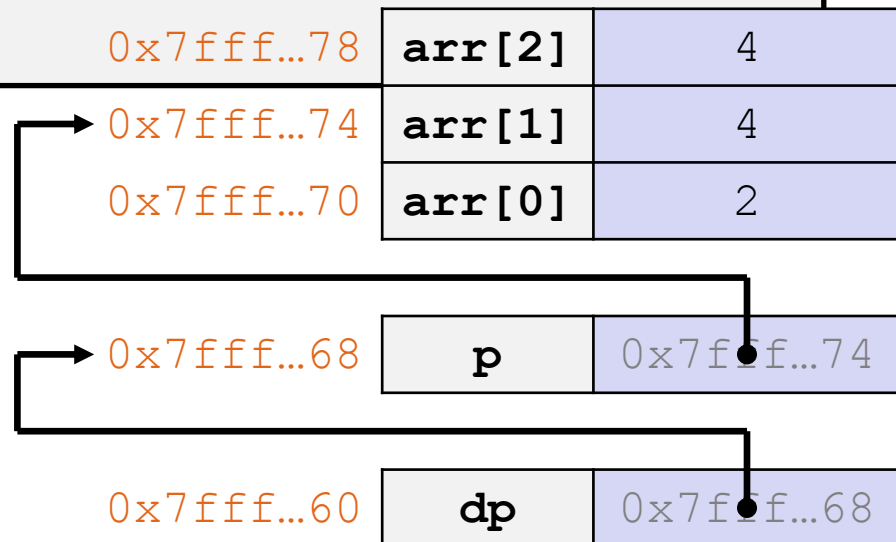
```
int main(int argc, char** argv) {
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];
    int** dp = &p; // pointer to a pointer

    *(*dp) += 1;
    p += 1;
    *(*dp) += 1;

    return 0;
}
```



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



# Practice Solution

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

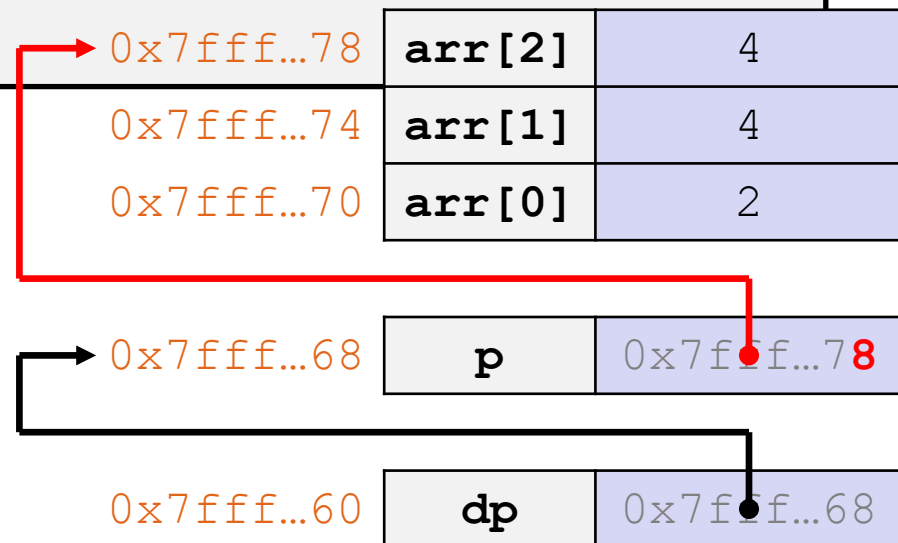
boxarrow2.c

```
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 0;
}
```

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



# Practice Solution

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

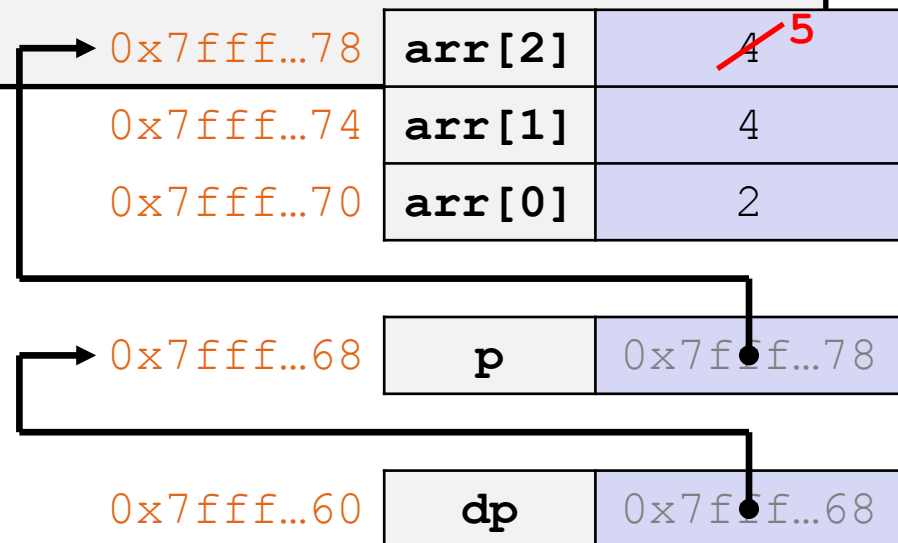
boxarrow2.c

```
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;

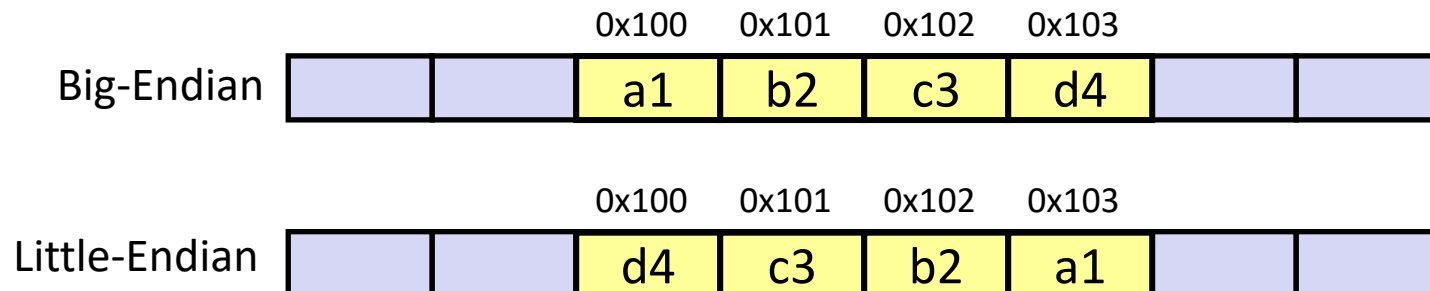
    return 0;
}
```

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;

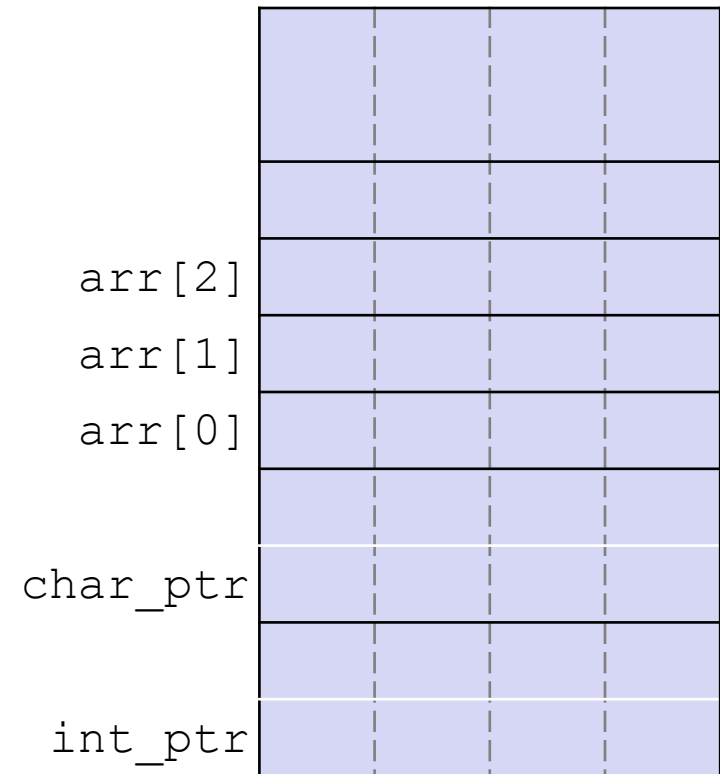
    int_ptr += 1;
    int_ptr += 2; // uh oh

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
    
```

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 0;
}
```

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.

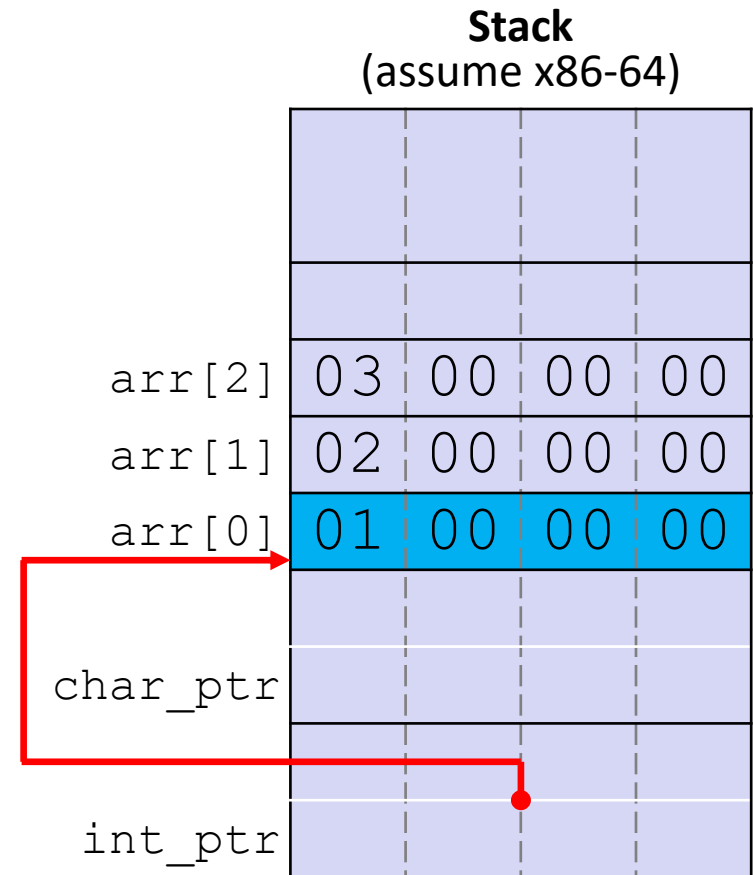
```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

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

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

pointerarithmetic.c



# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

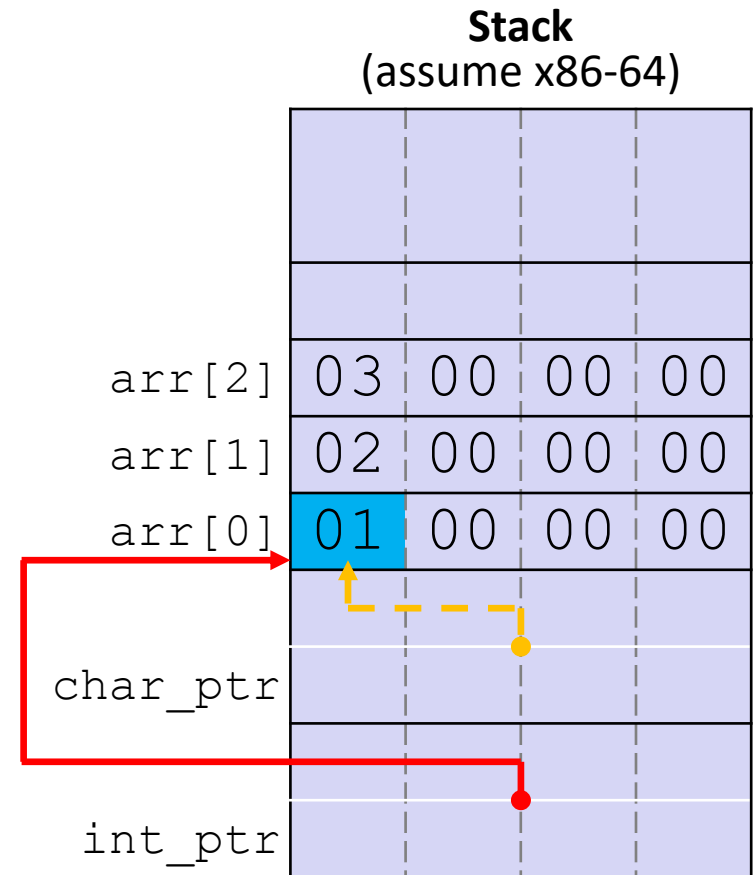
```
int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3};
    int* int_ptr = &arr[0];
    char* char_ptr = (char*) int_ptr;

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

    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

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

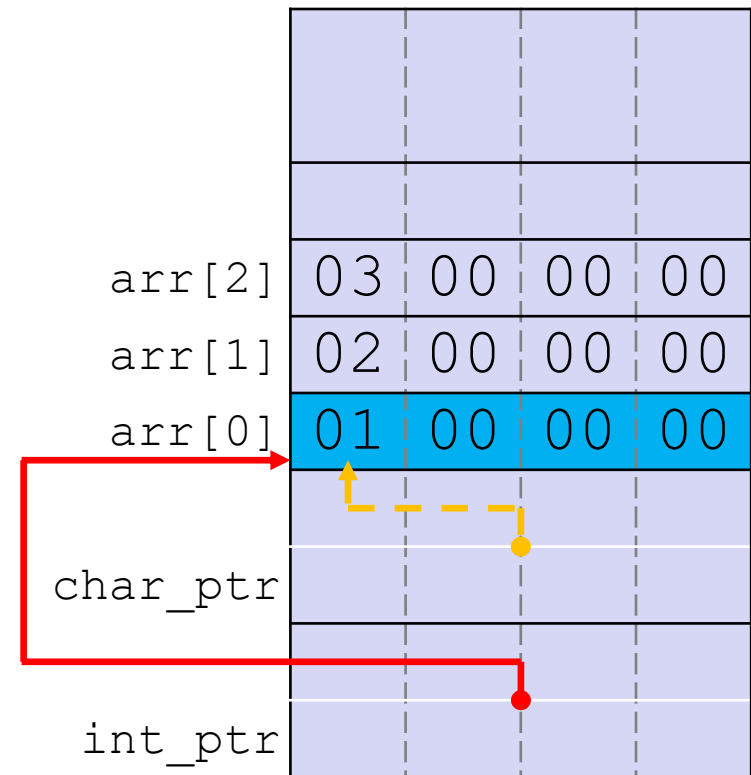
    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

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

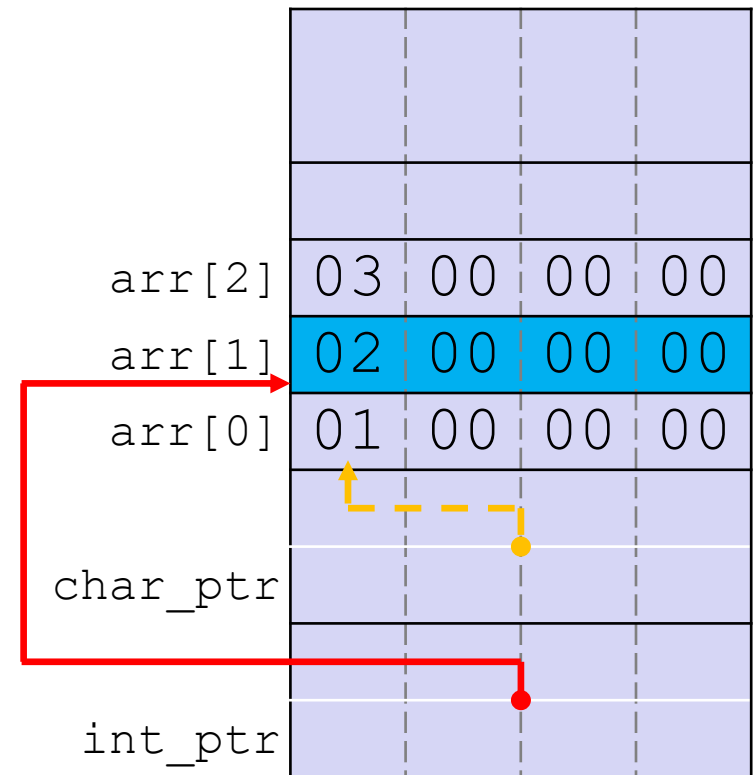
    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

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

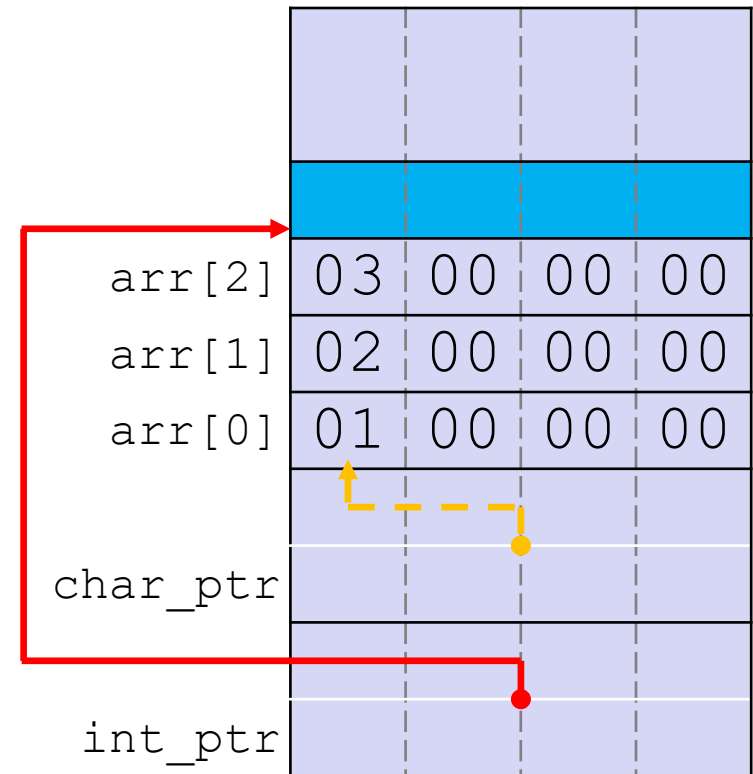
    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

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

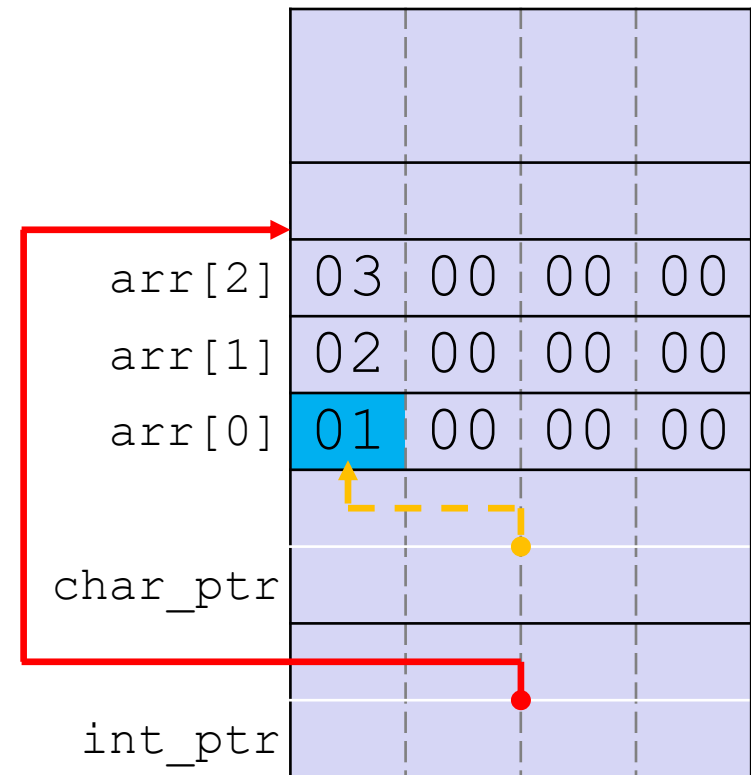
    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

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

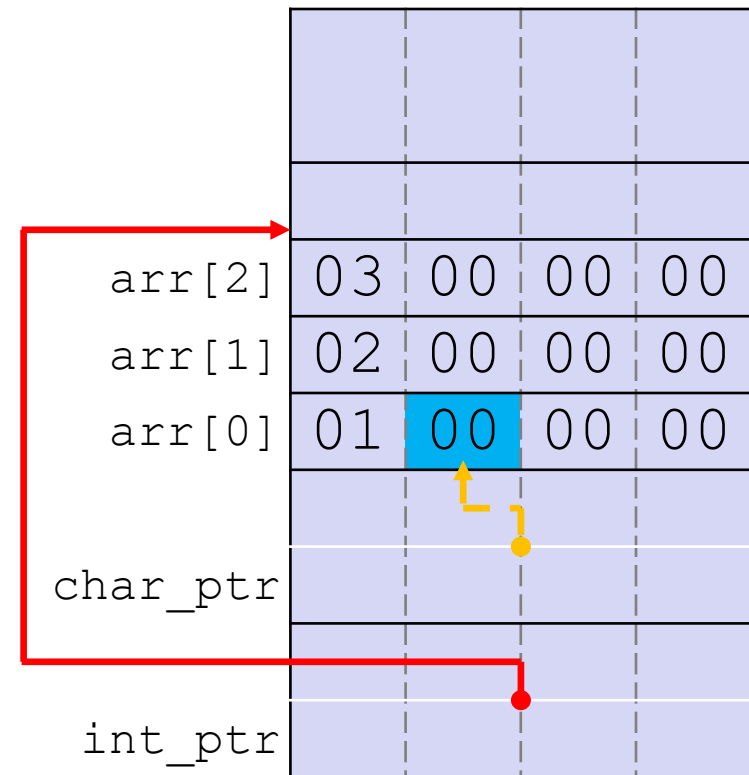
    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

pointerarithmetic.c

char\_ptr: 0x0x7fffffffde01**1**  
 \*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

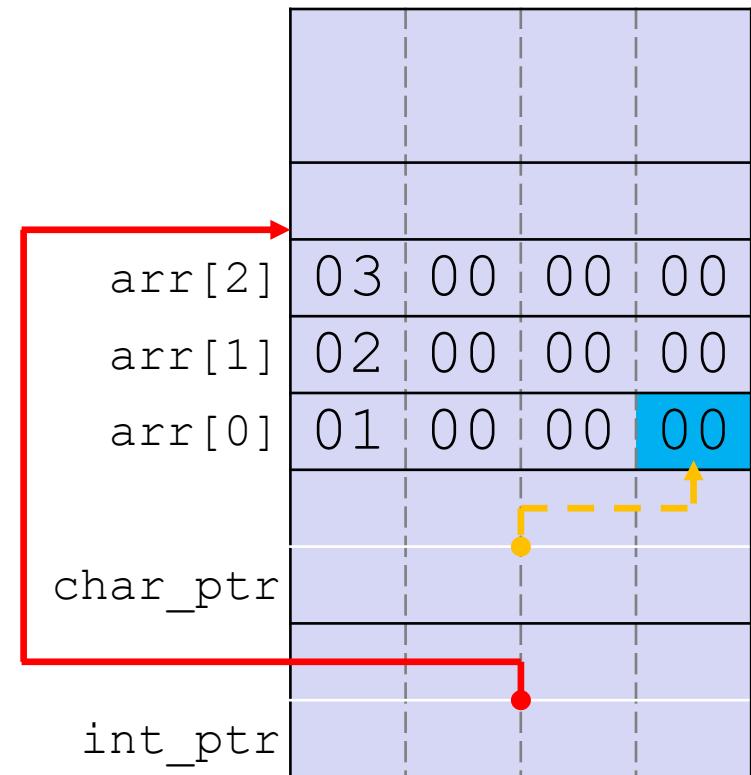
    char_ptr += 1;
    char_ptr += 2;

    return 0;
}
```

pointerarithmetic.c

char\_ptr: 0x0x7fffffffde013  
 \*char\_ptr: 0

Stack  
 (assume x86-64)



# Lecture Outline

- ❖ Pointers & 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

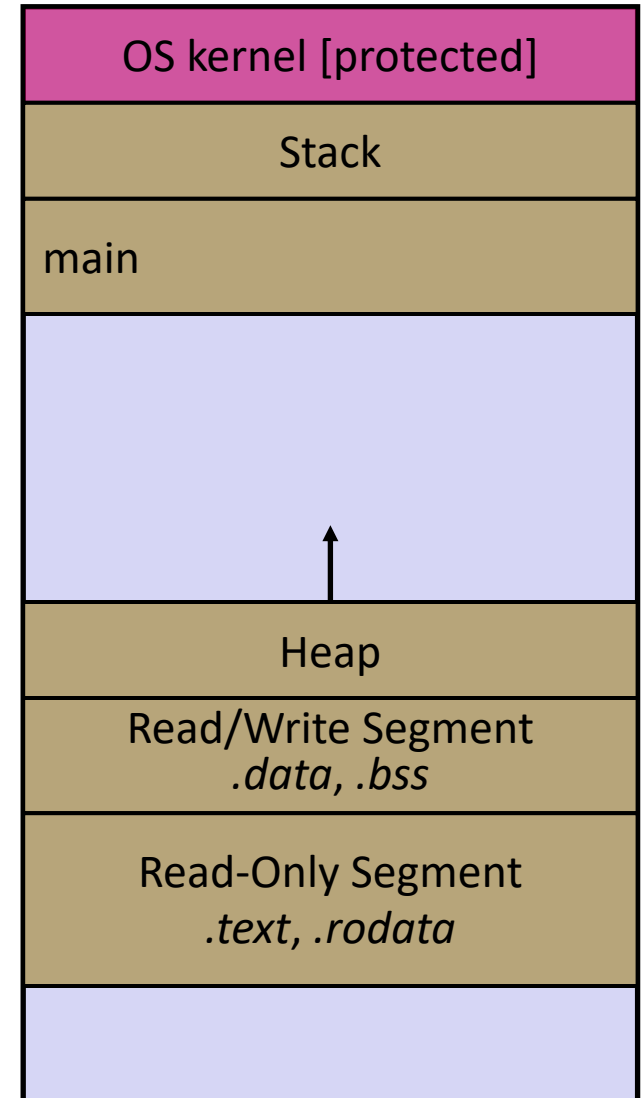
Note: Arrow points to *next* instruction.

## brokenswap.c

```

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

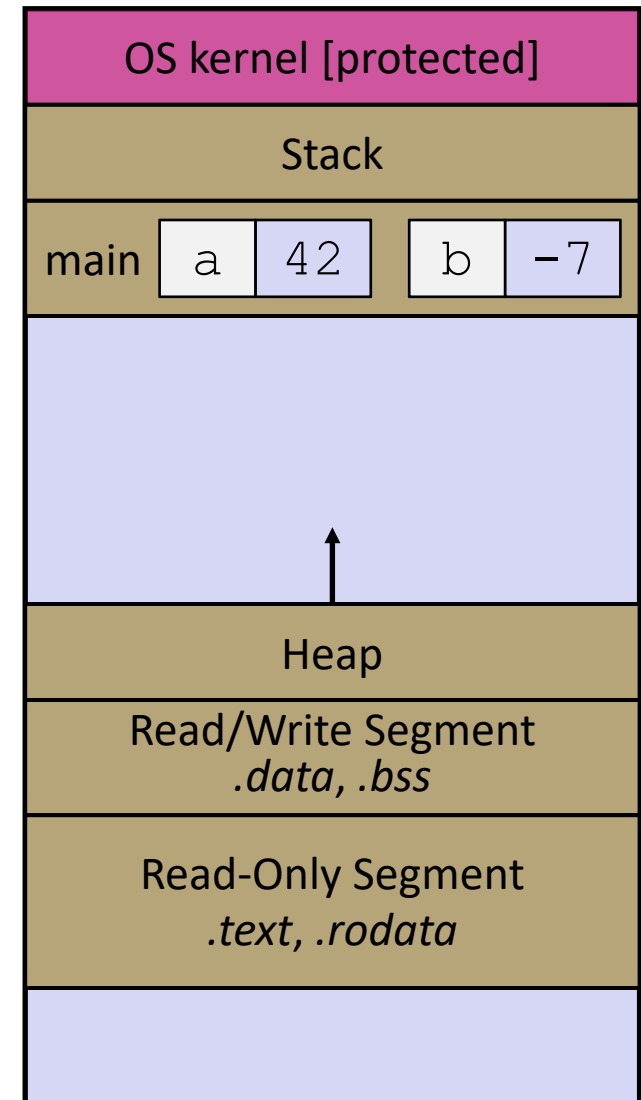
int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
    
```



# 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);  
    ...  
}
```



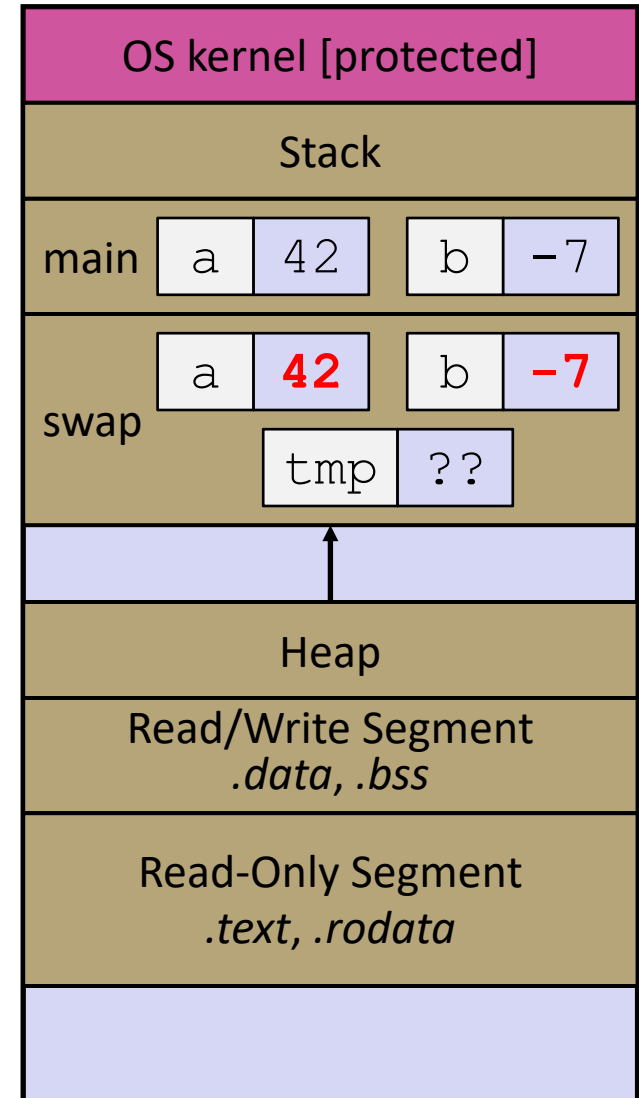
# Broken Swap

## 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);
    ...
    
```



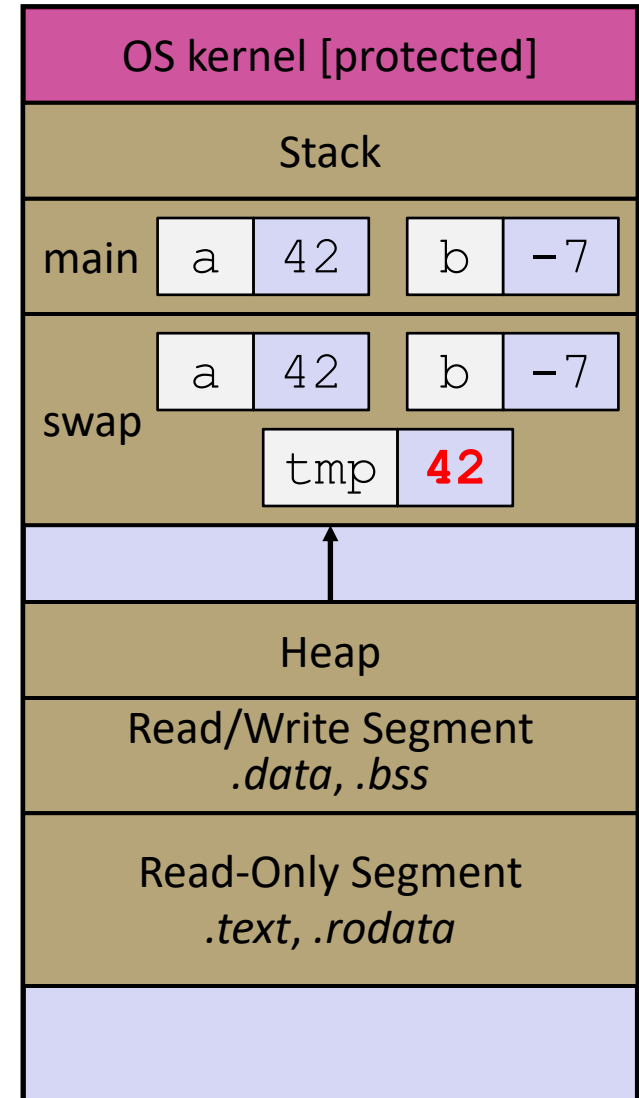
# Broken Swap

## 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);
    ...
}
    
```





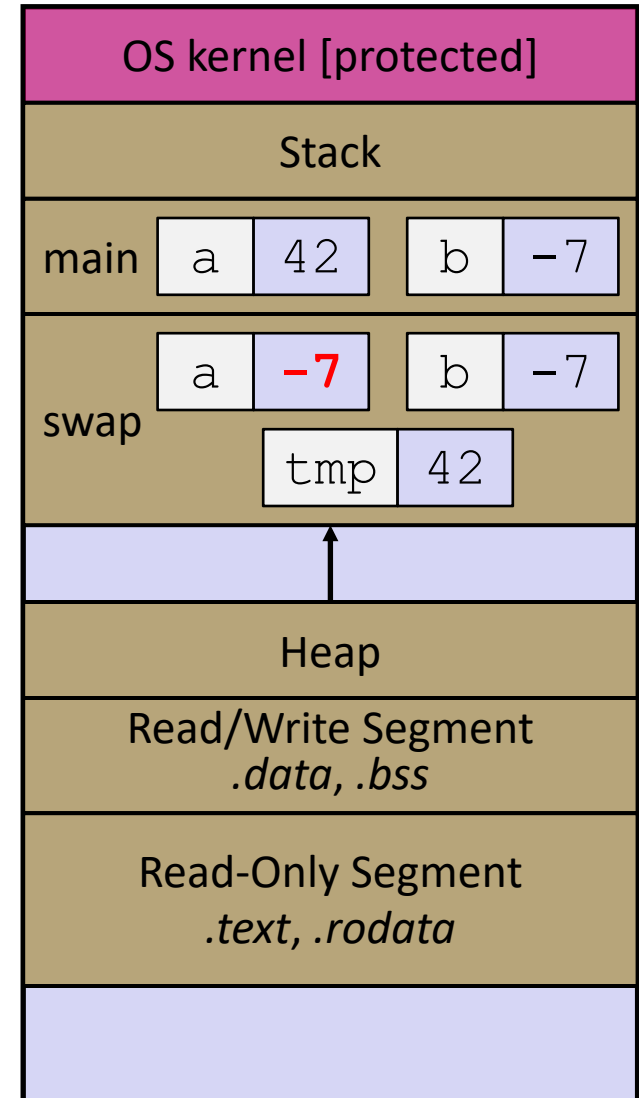
# Broken Swap

## 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);
    ...
}
    
```



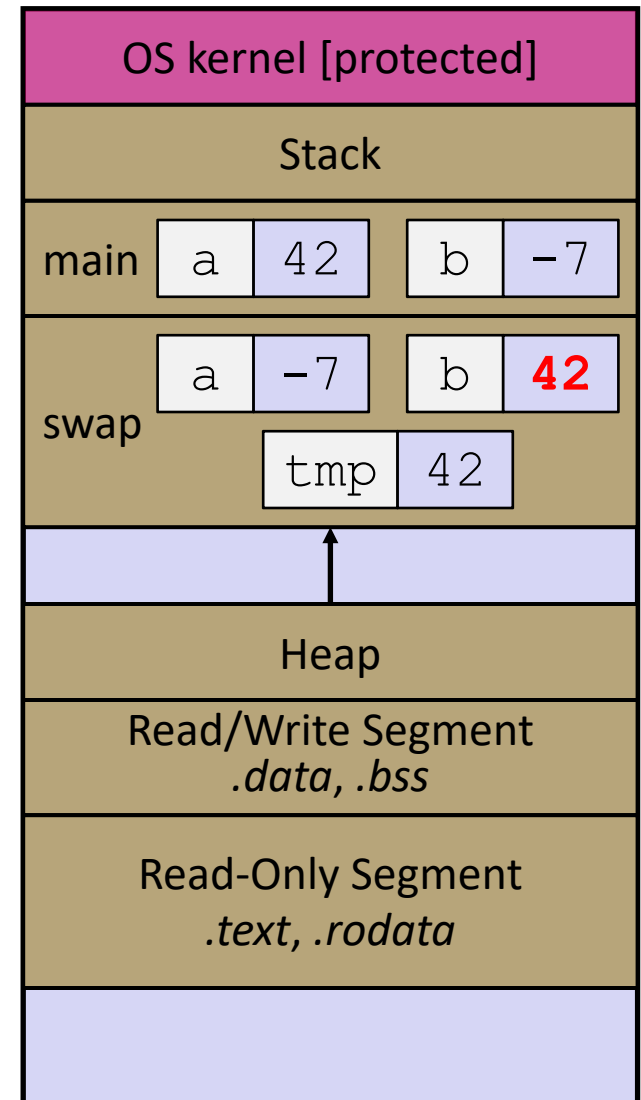
# Broken Swap

## brokenswap.c

```

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

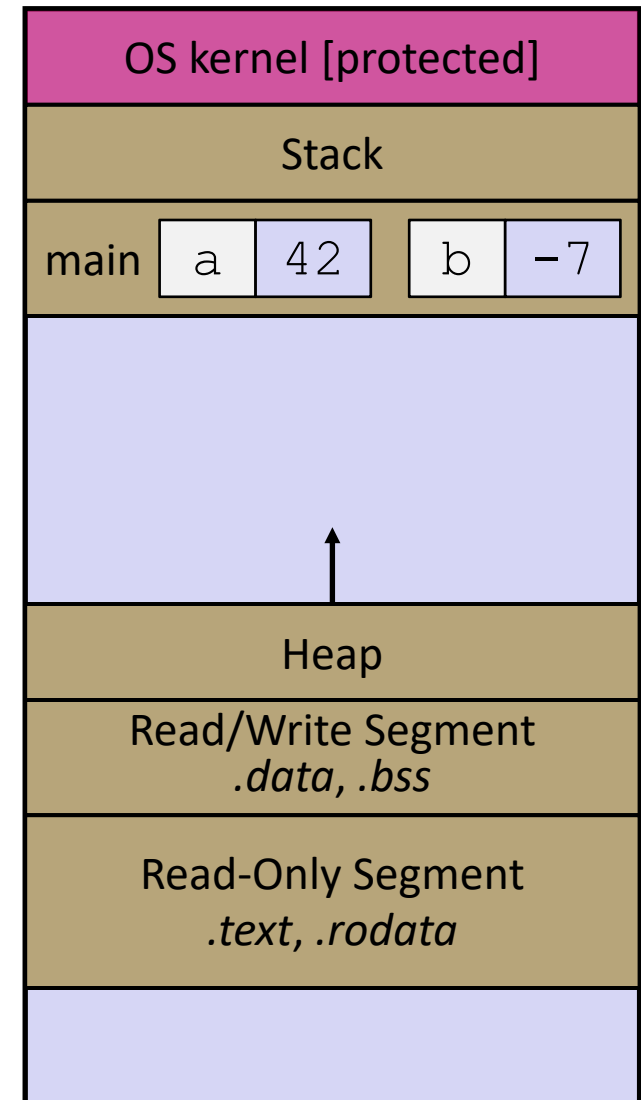
int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(a, b);
    ...
    
```



# 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

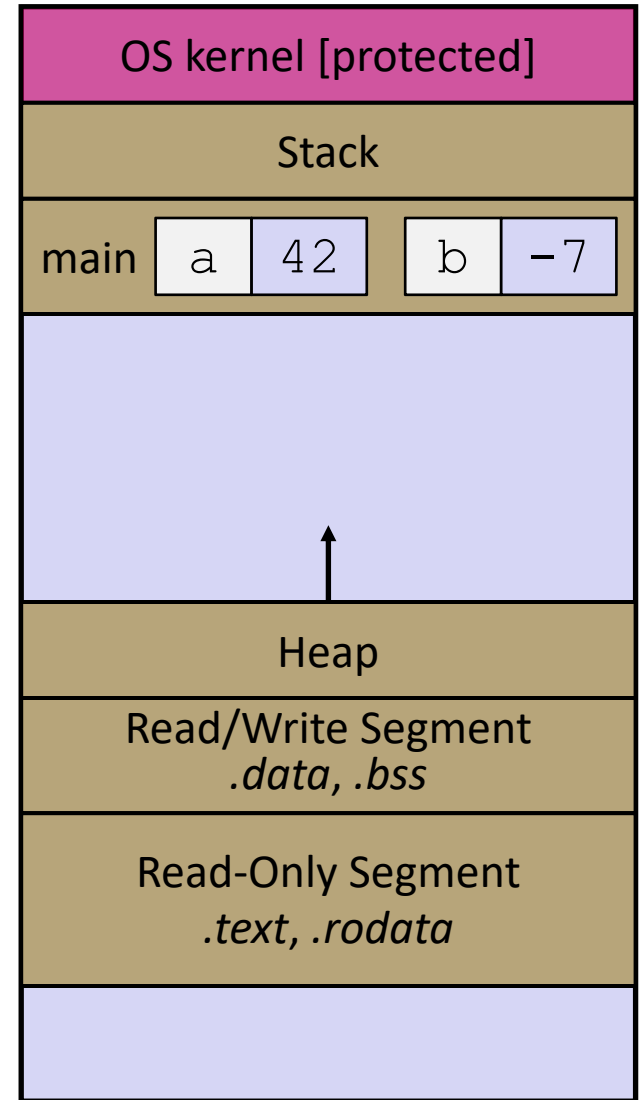
Note: Arrow points to *next* instruction.

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);
    ...
    
```



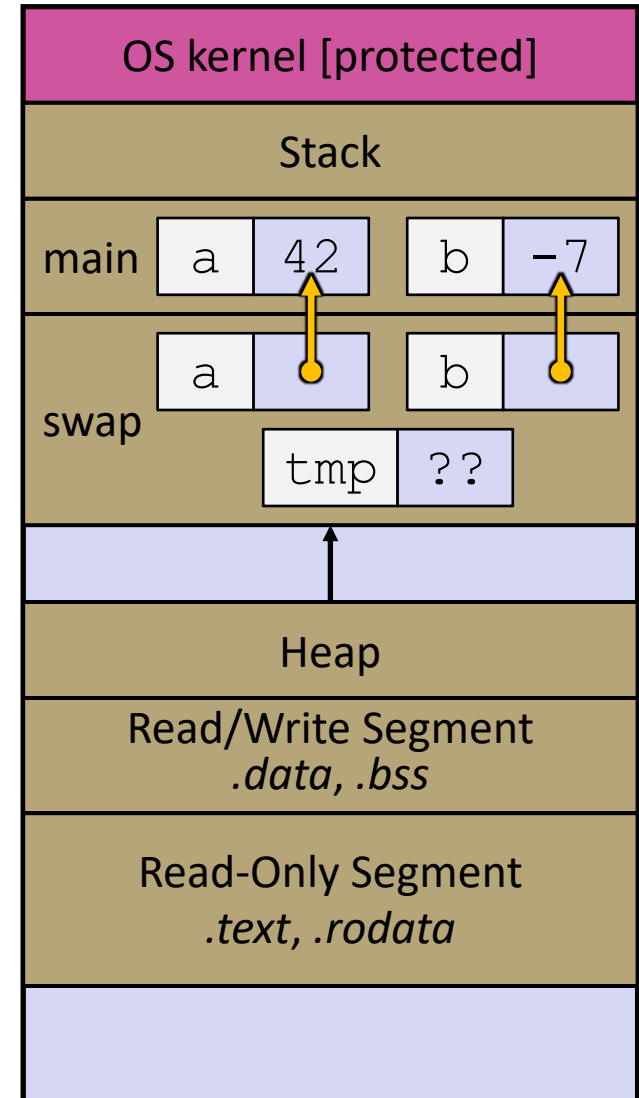
# Fixed Swap

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);
    ...
}
    
```



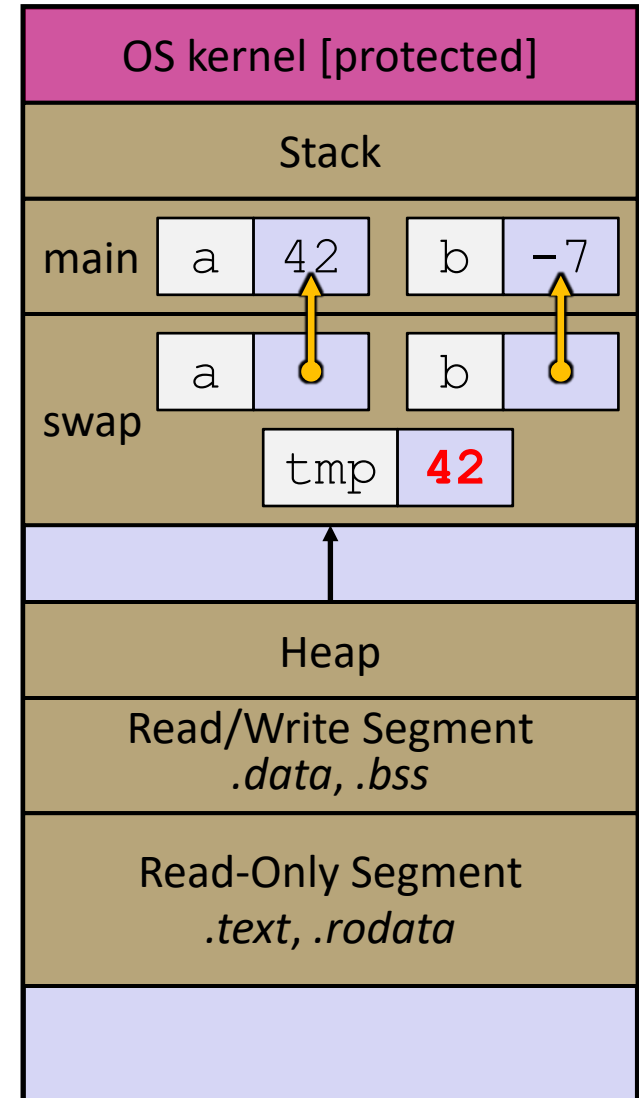
# Fixed Swap

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);
    ...
}
    
```



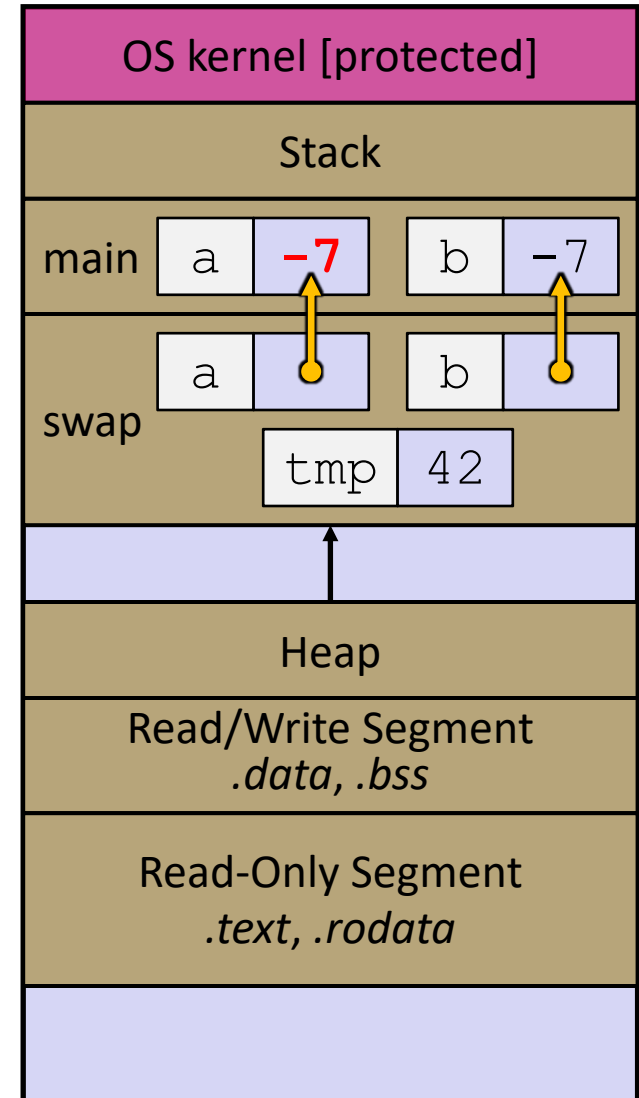
# Fixed Swap

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);
    ...
}
    
```





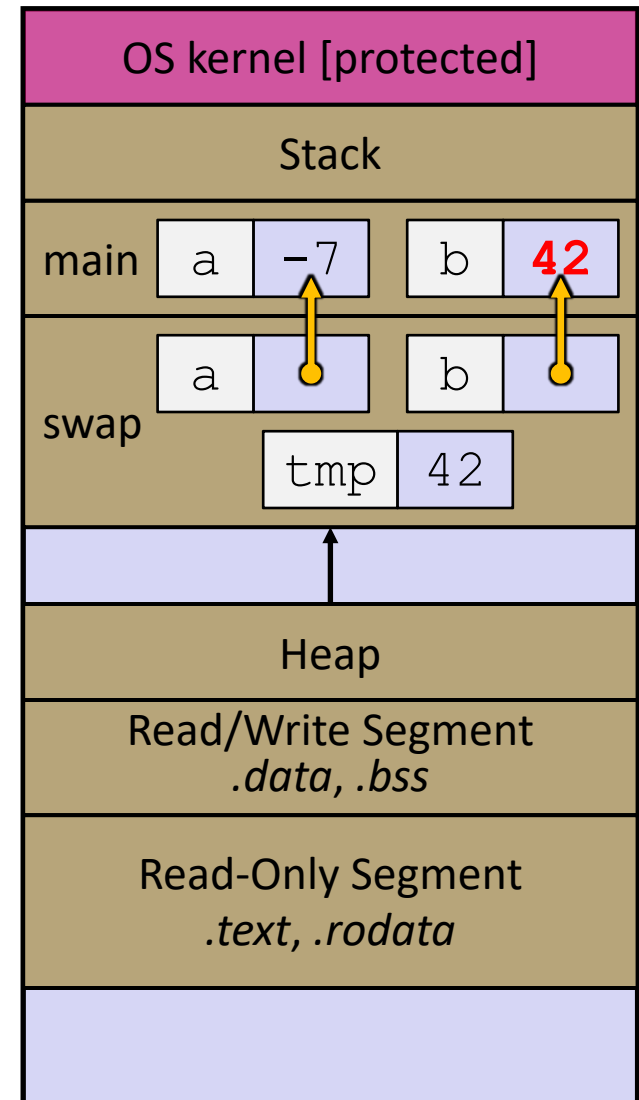
# Fixed Swap

swap.c

```

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

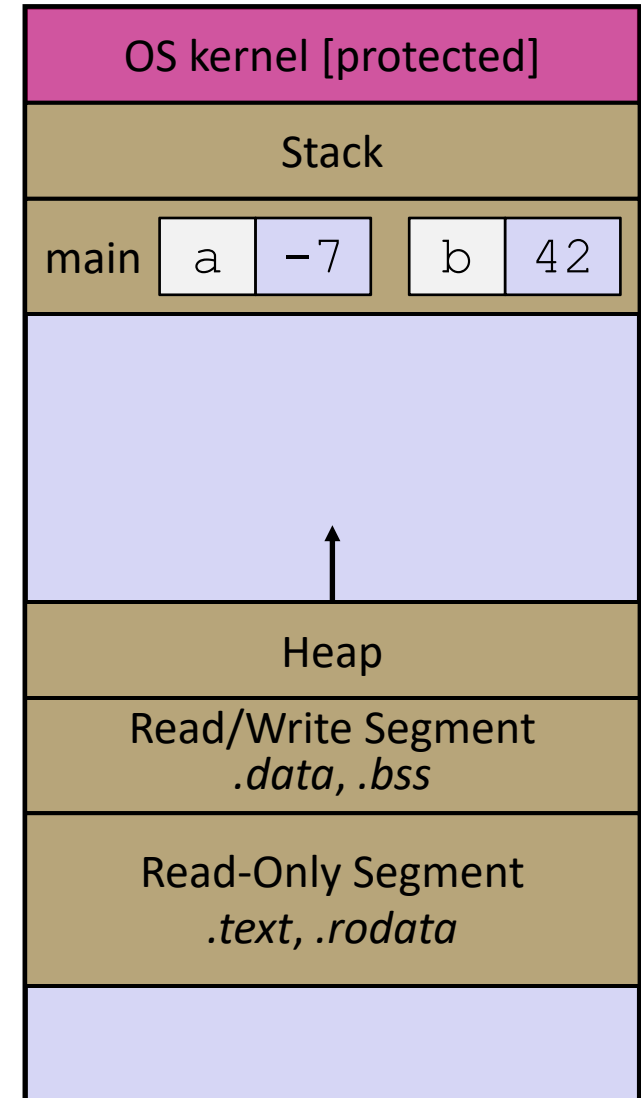
int main(int argc, char** argv) {
    int a = 42, b = -7;
    swap(&a, &b);
    ...
    
```



# 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);  
    ...  
}
```



# Lecture Outline

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

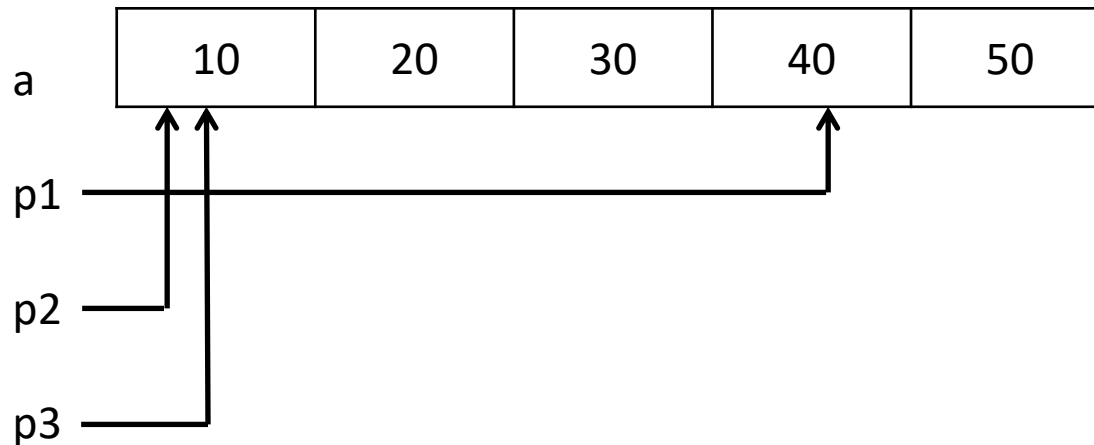
# Pointers and Arrays

- ❖ 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
```

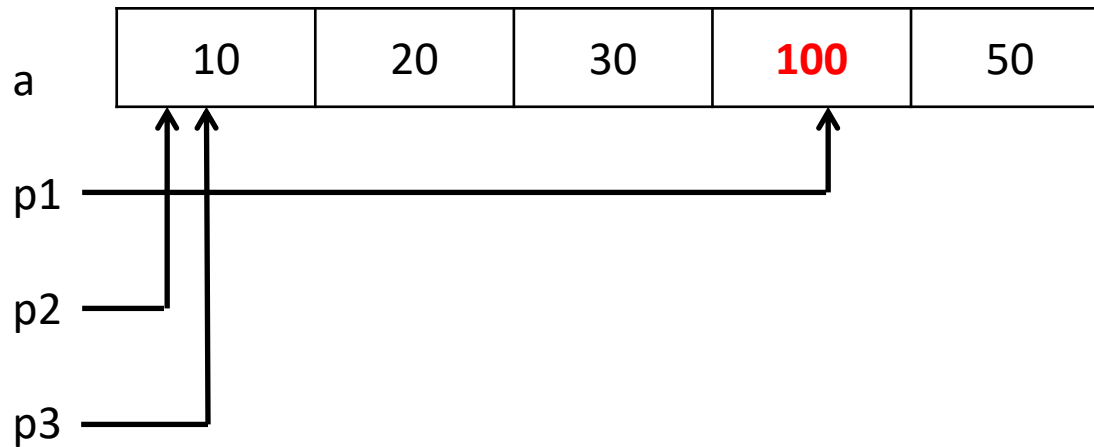
# Pointers and Arrays - Trace



```
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
```

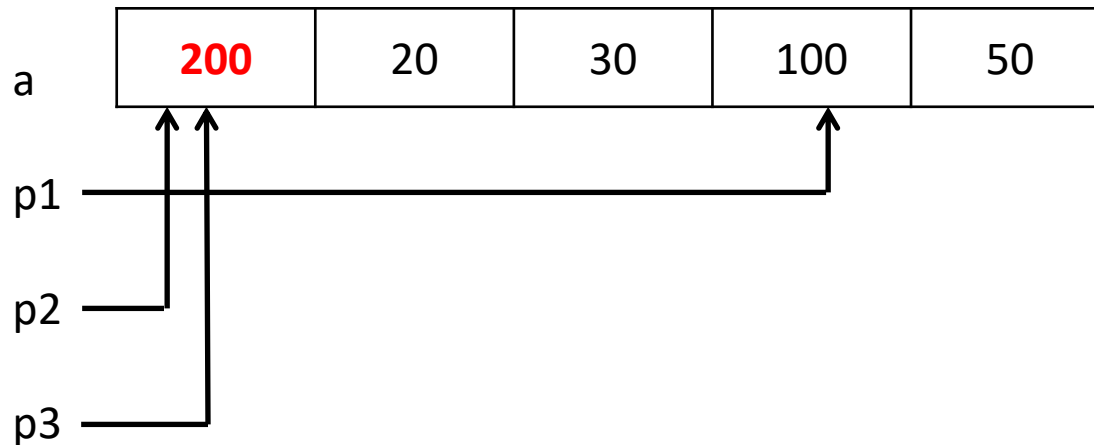
# Pointers and Arrays - Trace



```
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
```

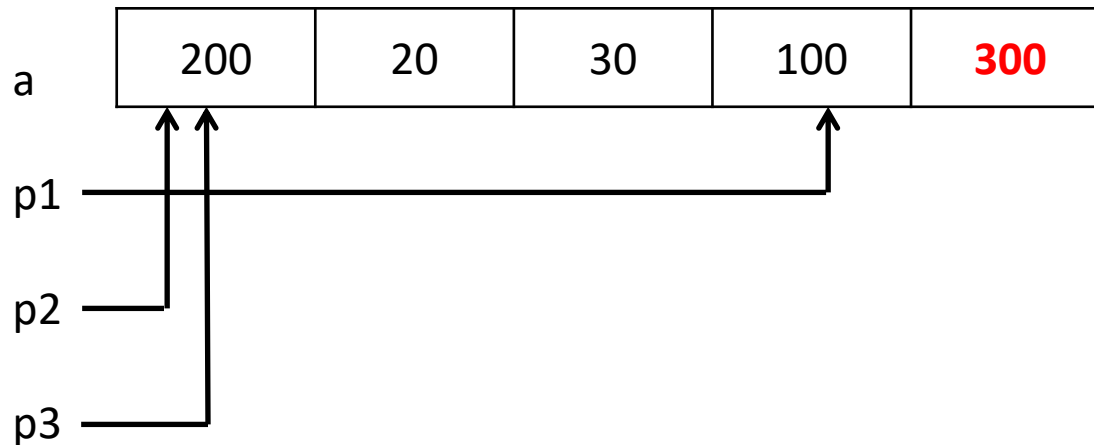
# Pointers and Arrays - Trace



```
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

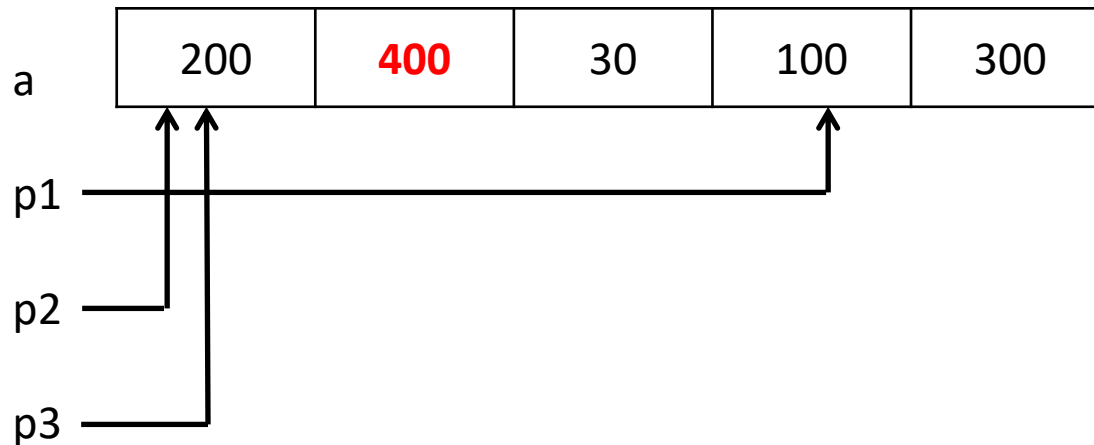


```
int a[] = {10, 20, 30, 40, 50};
int* p1 = &a[3]; // refers to a's 4th element
int* p2 = &a[0]; // refers to a's 1st element
int* p3 = a;    // refers to a's 1st element

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



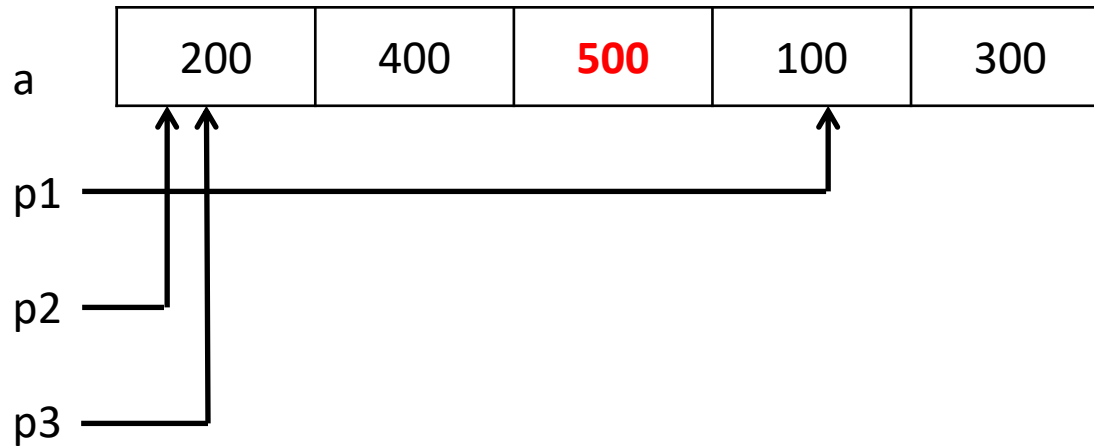
# Pointers and Arrays - Trace



```
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
```

# Array Parameters

- ❖ Array parameters are *actually* passed (by value) 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:

```
void f(int a[]);

int main( ... ) {
    int a[5];
    ...
    f(a);
    return 0;
}

void f(int a[]) {
```

Equivalent to:

```
void f(int* a);

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

void f(int* a) {
```

# Lecture Outline

- ❖ Pointers & 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 (but \* is optional since all you can do is call it!)

# 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;`)

map.c

# 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