Pointers, Pointers, Pointers
CSE 333 Winter 2024

Instructor: Hal Perkins

Teaching Assistants:

Ann Baturytski	Noa Ferman	Hannah Jiang
Humza Lala	Leanna Nguyen	Varun Pradeep
Justin Tysdal	Deeksha Vatwani	Yiqing Wang
Wei Wu	Jennifer Xu	

Administrivia

❖ Office hours start today! See schedule on web calendar.

❖ Exercise 2 out this morning; due Wednesday 10 am

❖ Homework 0 due tonight(!) 11 pm
  ■ Logistics and infrastructure for projects
    • `cpplint` 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
      – Do *not* just check the gitlab web page – clone the repo and test!
      – If trouble, *throw away* this extra copy and fix things in the original repo, add/commit/push, retag, and repeat
  ■ Reminder: all exercises/hw **must** be done on current Allen School Linux machines (attu/lab/VM)

More Administrivia

❖ HW1 posted and pushed to repos Saturday

- 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

    – Default git merge handling will almost certainly do the right thing

    – If git drops you into vi(m), :q to quit or :wq if you want to save changes

    – To avoid, always do a git pull before any git commit or push

- Please read the assignment and start looking at the code now!

  - For large projects, you want to pace yourself so if something baffling happens, you can let it go for the day and come back to it tomorrow

Yet More Administrivia

- ❖ Exercise grading – Gradescope abuse
  - Score is an overall evaluation: 3/2/1/0 = superior / good / marginal / not sufficient for credit
    - We expect lots of 2's and 3's at first, more 3's on later exercises
  - Then additional ±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 ±0 "score" is just because that's how we have to use Gradescope to handle feedback notes – it does not contribute to "the points"

- ❖ Hoping to have ex0 scores out shortly, but infrastructure…

Lecture Outline

- ❖ **Pointers & Pointer Arithmetic**

- ❖ Pointers as Parameters

- ❖ Pointers and Arrays

- ❖ Function Pointers

Box-and-Arrow Diagrams

boxarrow.c

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

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

  return EXIT_SUCCESS;
}
```

address | **name** | value

Box-and-Arrow Diagrams

boxarrow.c

```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

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

stack frame for main()

7

Box-and-Arrow Diagrams

boxarrow.c

```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

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

8

Box-and-Arrow Diagrams

boxarrow.c

```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…78	**arr[2]**	4
0x7fff…74	**arr[1]**	3
0x7fff…70	**arr[0]**	2
0x7fff…68	**p**	0x7fff…74
0x7fff…64	**x**	1

9

Pointer Arithmetic

- ❖ Pointers are *typed*
  - ▪ Tells the compiler the size of the data you are pointing to
  - ▪ <u>Exception</u>: `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 and a pointer
  - ▪ Subtract two pointers (within same stack frame or malloc block)
  - ▪ Compare pointers (<, <=, ==, !=, >, >=), including `NULL`
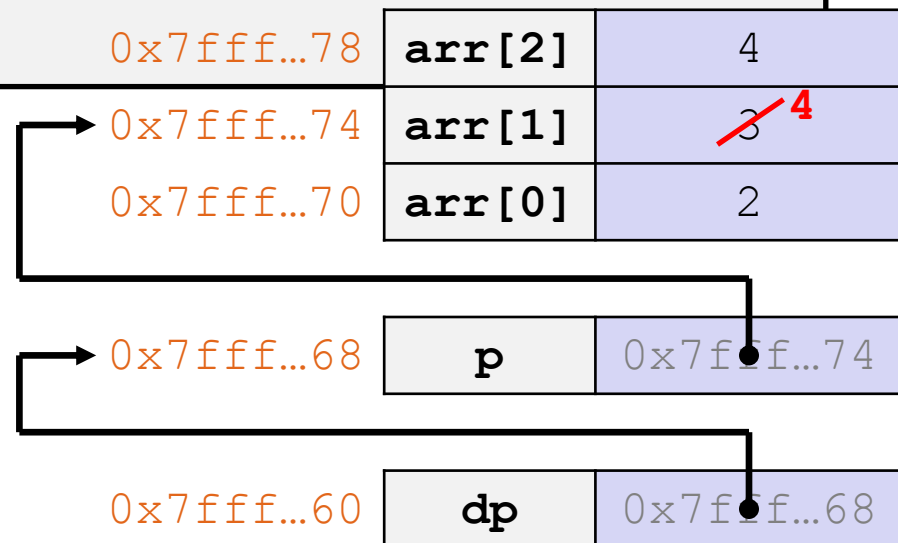
Practice Question

boxarrow2.c

```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;    At this point in the code, what values are
                  stored in arr[]?
  return EXIT_SUCCESS;
}
```

address | **name** | value

0x7fff…78	**arr[2]**	4
0x7fff…74	**arr[1]**	3
0x7fff…70	**arr[0]**	2

0x7fff…68	**p**	0x7fff…74

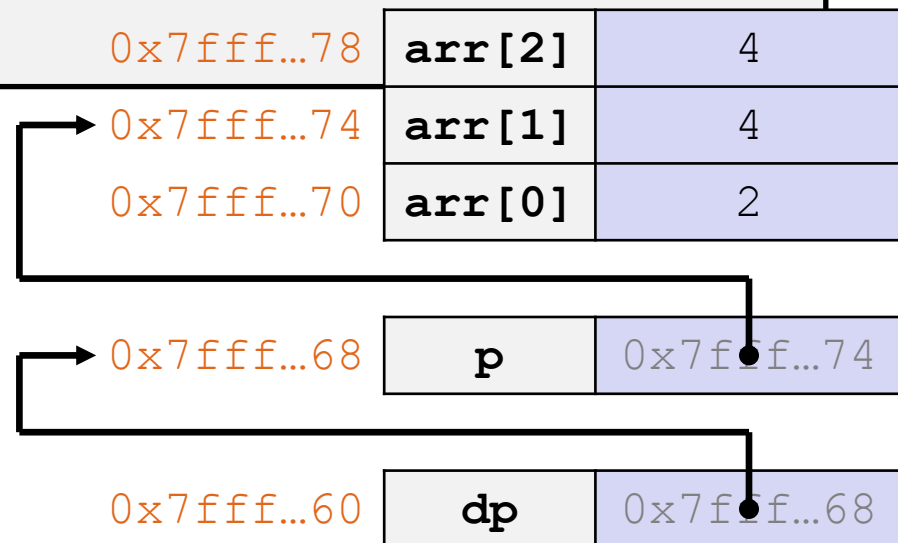|		
0x7fff…60	**dp**	0x7fff…68

Practice Solution

boxarrow2.c

```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]**	~~3~~ **4**
0x7fff…70	**arr[0]**	2

0x7fff…68	**p**	0x7fff…74

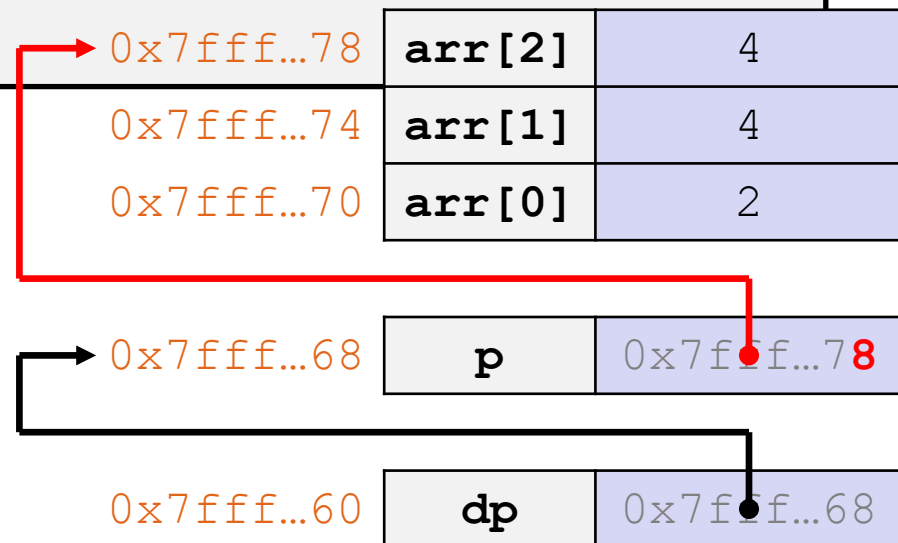| 0x7fff…60	**dp**	0x7fff…68

Practice Solution

boxarrow2.c

```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;
}
```

	name	value
address		

0x7fff…78	**arr[2]**	4
0x7fff…74	**arr[1]**	4
0x7fff…70	**arr[0]**	2

0x7fff…68	**p**	0x7fff…74

0x7fff…60	**dp**	0x7fff…68

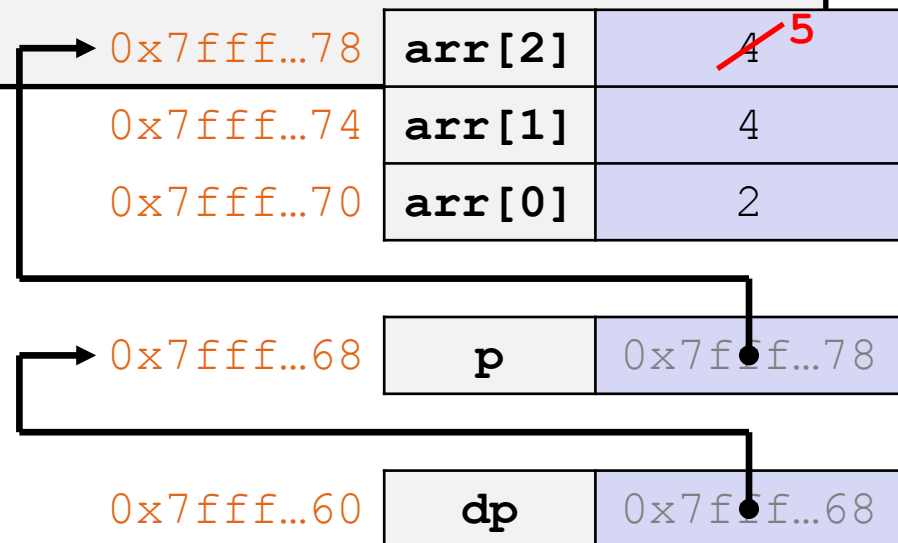13

Practice Solution

boxarrow2.c

```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;
}
```
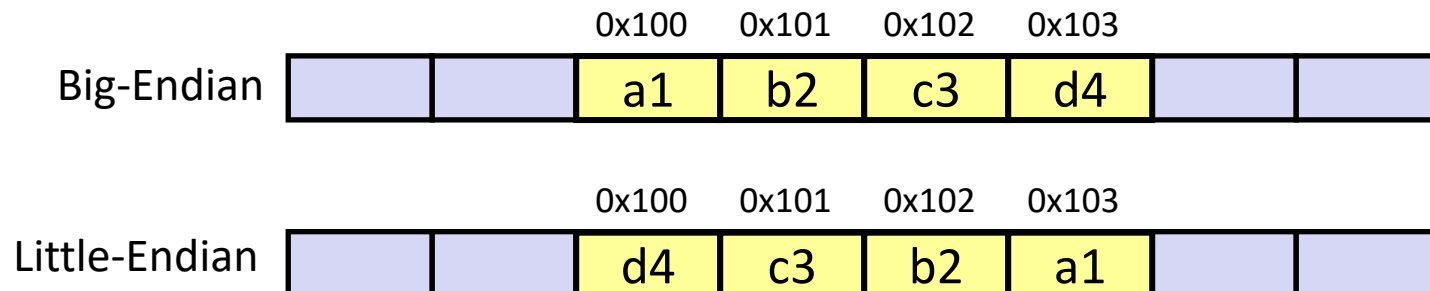
address	name	value
0x7fff…78	**arr[2]**	4
0x7fff…74	**arr[1]**	4
0x7fff…70	**arr[0]**	2
0x7fff…68	**p**	0x7fff…78
0x7fff…60	**dp**	0x7fff…68

14

Practice Solution

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

boxarrow2.c

```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 5
0x7fff…74	**arr[1]**	4
0x7fff…70	**arr[0]**	2

0x7fff…68	**p**	0x7fff…78

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* address
- Little-endian: Least significant byte has *lowest* address

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

	0x100	0x101	0x102	0x103		
Big-Endian			a1	b2	c3	d4

	0x100	0x101	0x102	0x103		
Little-Endian			d4	c3	b2	a1

Pointer Arithmetic Example

Note: Arrow points to *next* instruction.
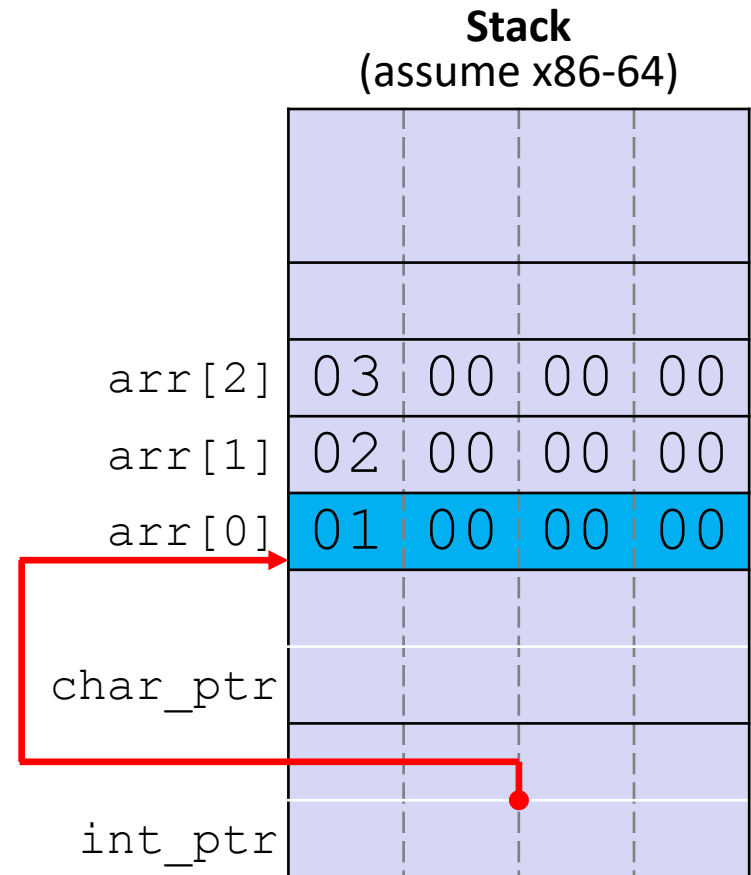
```c
int main(int argc, char** argv) {
  int arr[3] = {1, 2, 3};
  int* int_ptr = &arr[0];
  char* char_ptr = (char*) int_ptr;

  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]

arr[1]

arr[0]

char_ptr

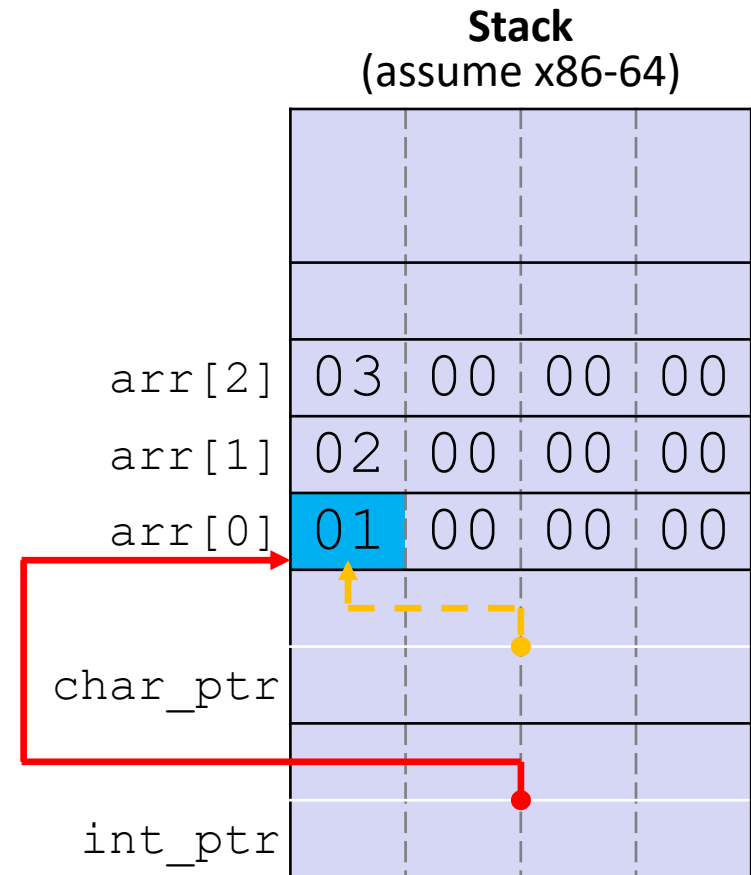int_ptr

Pointer Arithmetic Example

```c
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			

18

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

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				

19

Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```c
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;
}
```
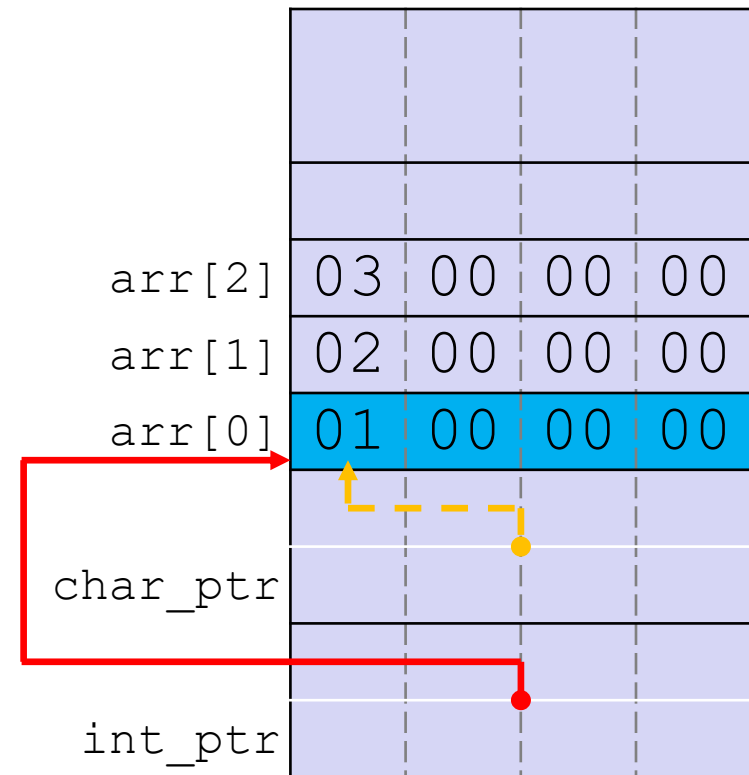
pointerarithmetic.c

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

20

Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```c
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;
}
```
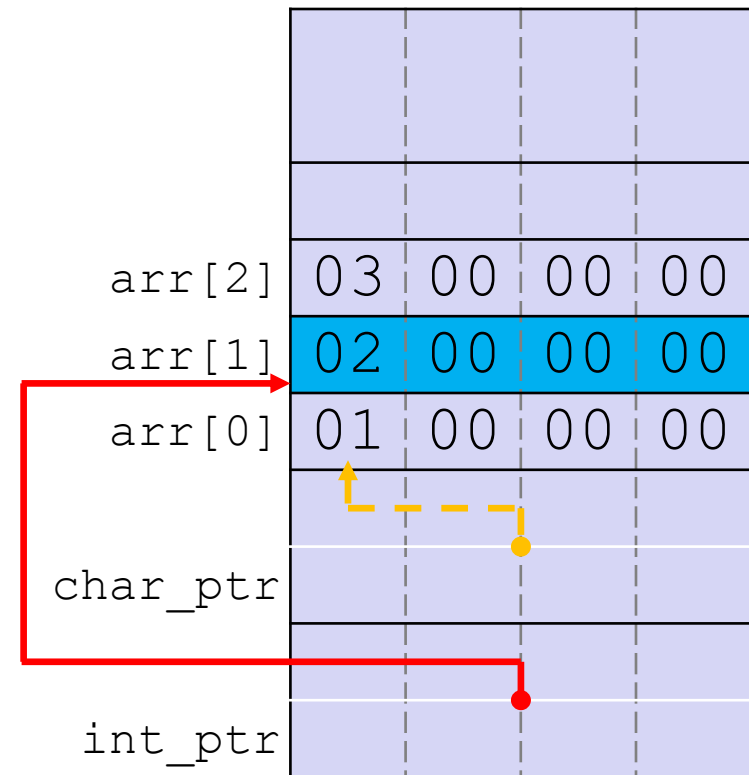
pointerarithmetic.c

int_ptr: 0x0x7fffffde010
***int_ptr:**   1

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.

```c
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;
}
```
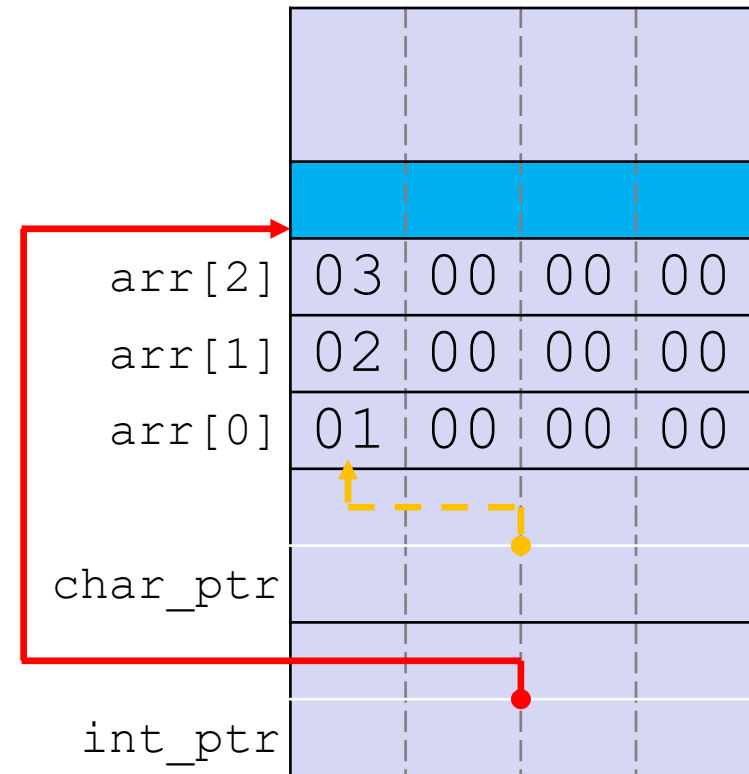
pointerarithmetic.c

int_ptr: 0x0x7fffffde01**4**
*int_ptr: **2**

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

```
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;
}
```
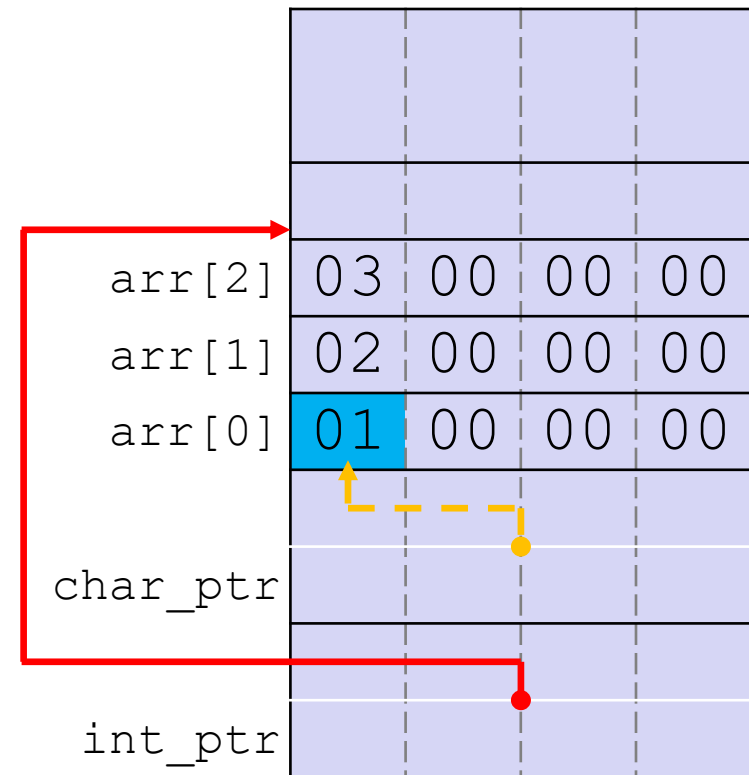
pointerarithmetic.c

Stack
(assume x86-64)

int_ptr: 0x0x7fffffde01**C**
*int_ptr: **???**

23

Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

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

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

  char_ptr += 1;
  char_ptr += 2;

  return EXIT_SUCCESS;
}
```
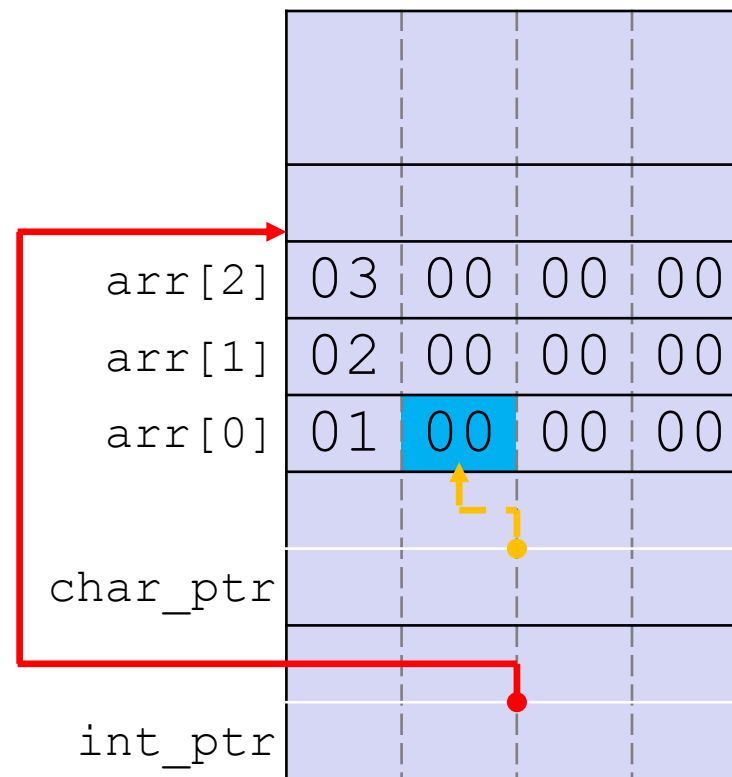
pointerarithmetic.c

Stack
(assume x86-64)

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

char_ptr: 0x0x7fffffde010
*char_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;
}
```
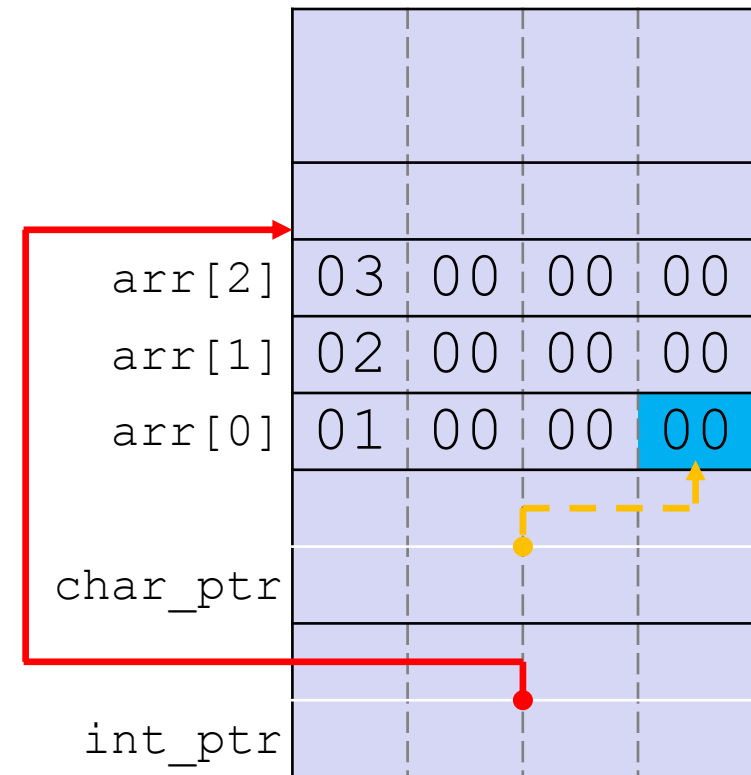
pointerarithmetic.c

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

char_ptr: 0x0x7fffffde01**1**
*char_ptr: 0

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

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

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

Lecture Outline

* Pointers & Pointer Arithmetic

* **Pointers as Parameters**

* Pointers and Arrays

* Function Pointers

C parameters are 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

```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

Note: Arrow points to *next* instruction.

brokenswap.c

```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);
  ...
```
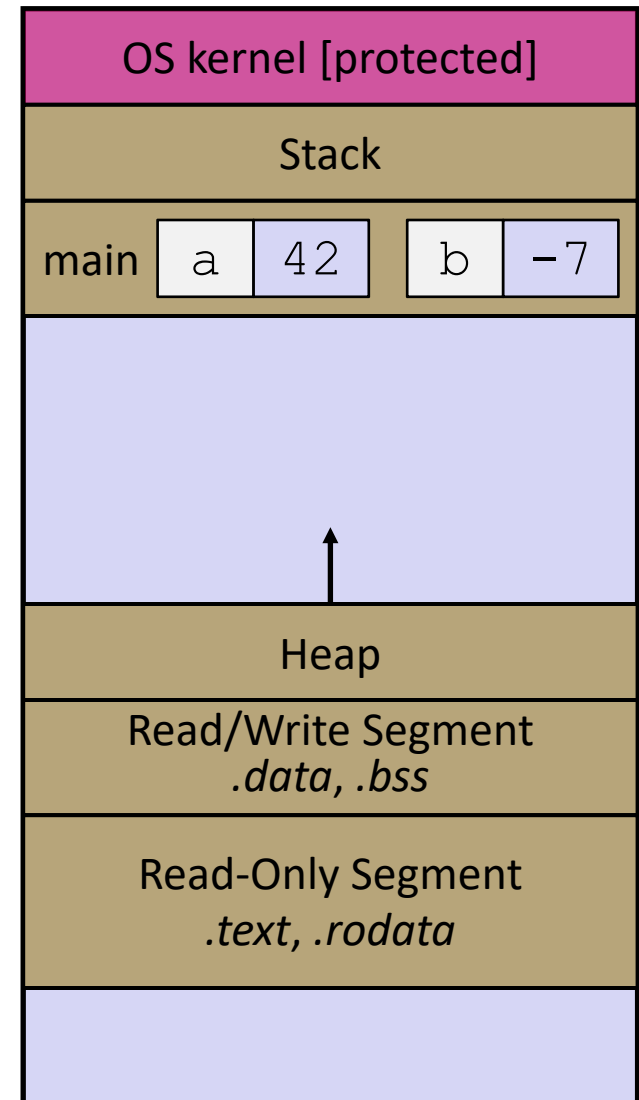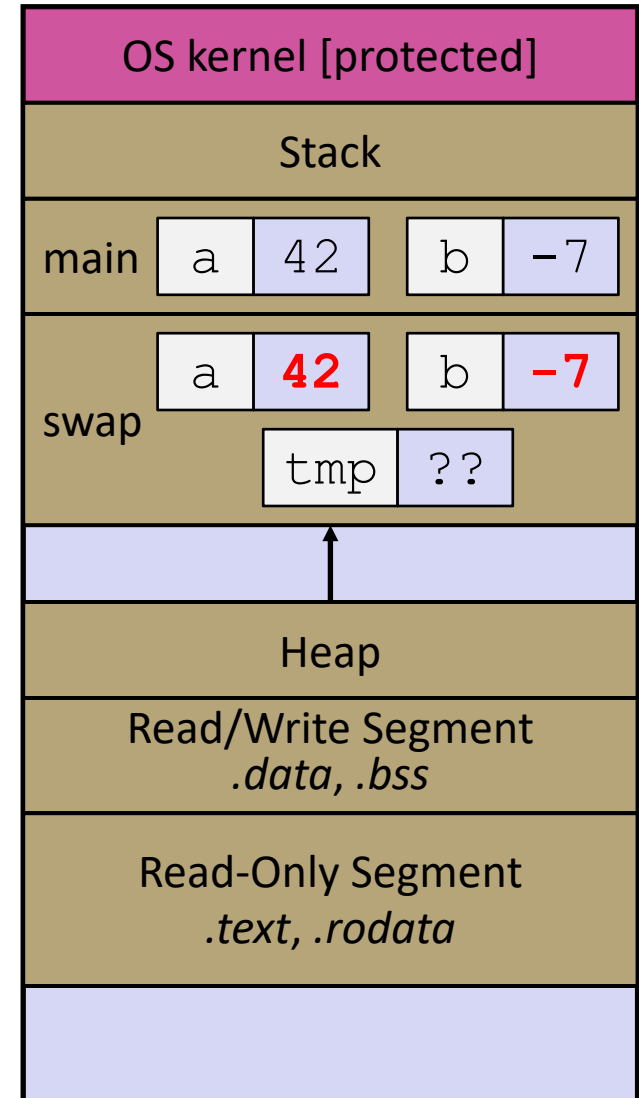
OS kernel [protected]
Stack
main
Heap
Read/Write Segment *.data*, *.bss*
Read-Only Segment *.text*, *.rodata*

29

Broken Swap

brokenswap.c

```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);
  ...
```
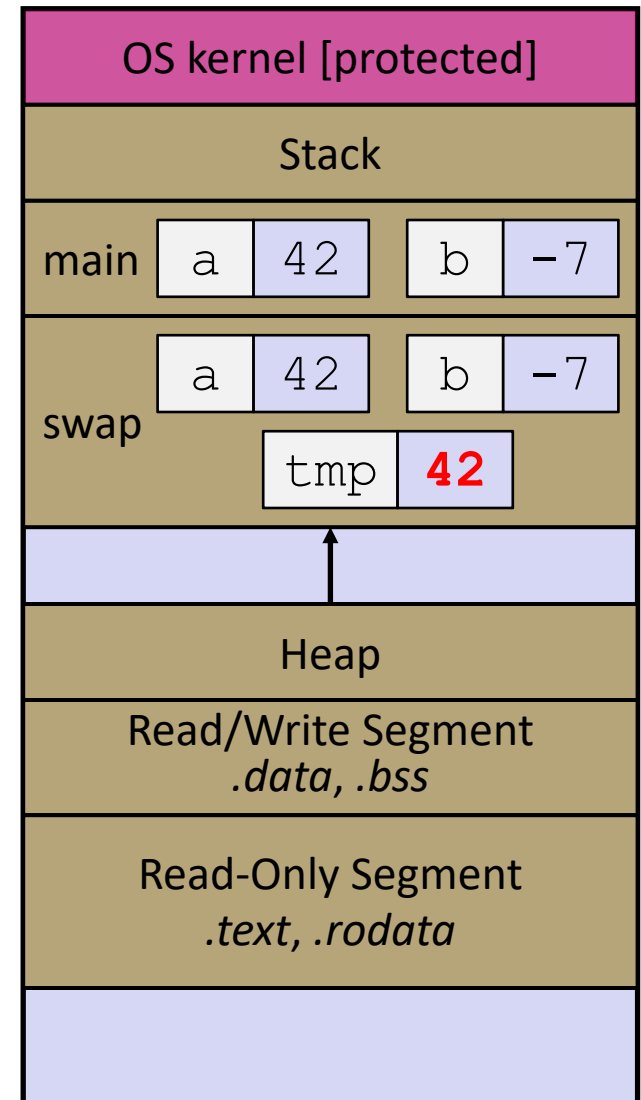
OS kernel [protected]
Stack

main | a | 42 | b | -7 |

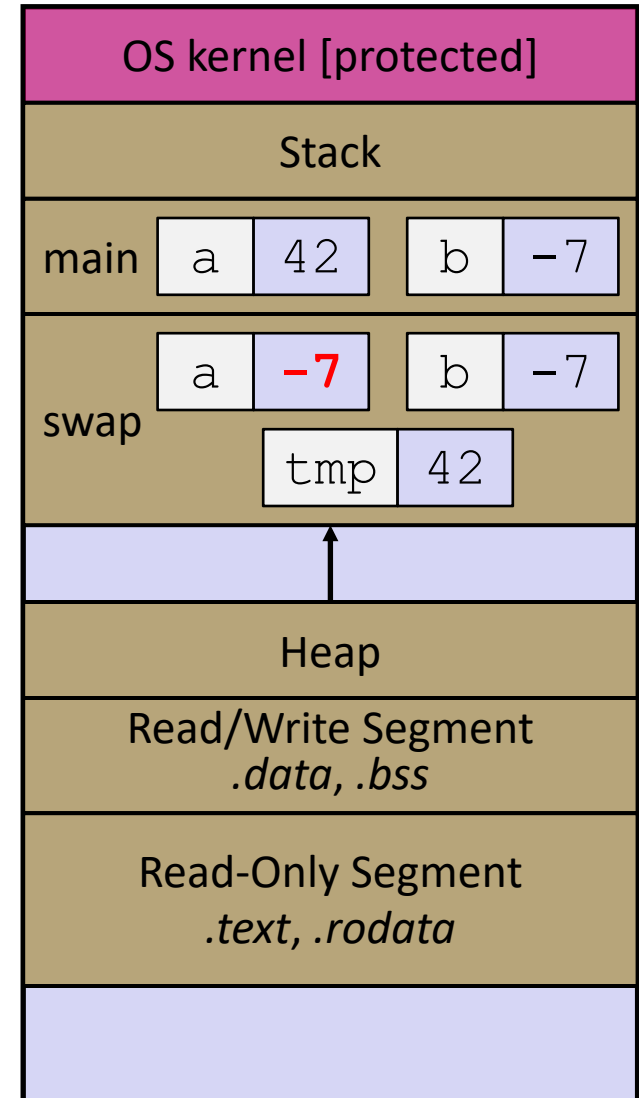| |
|---|
| Heap |
| Read/Write Segment
.data, .bss |
| Read-Only Segment
.text, .rodata |

Broken Swap

brokenswap.c

```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);
  ...
```

OS kernel [protected]
Stack

main | a | 42 | | b | -7

swap | a | **42** | | b | **-7**
| | tmp | ?? |

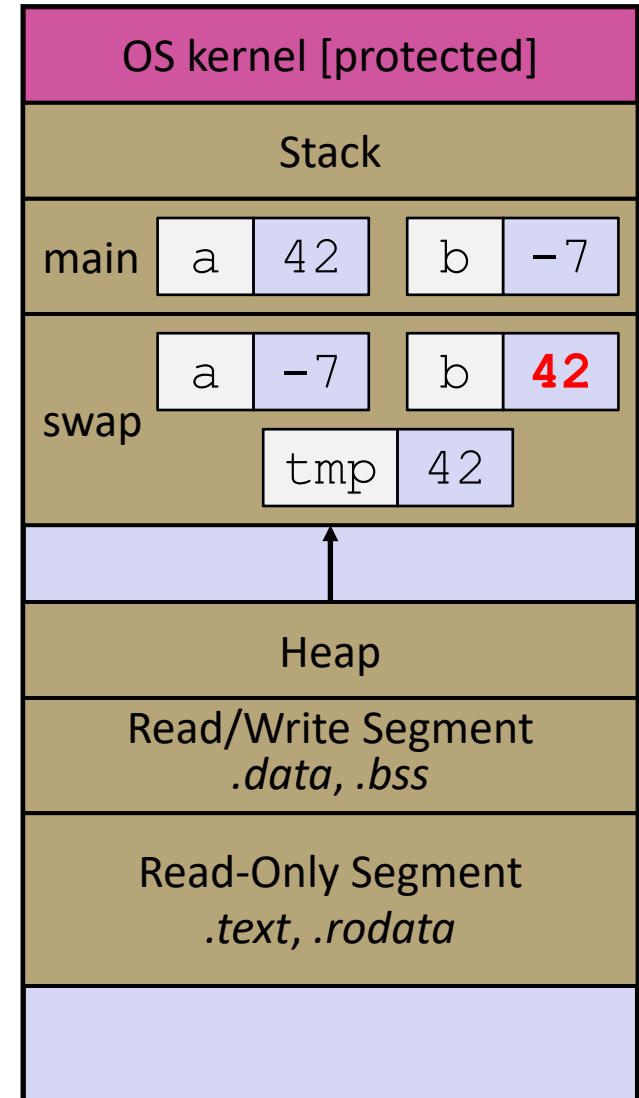| Heap |
| --- |
| Read/Write Segment .data, .bss |
| Read-Only Segment .text, .rodata |

Broken Swap

brokenswap.c

```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);
    ...
```

OS kernel [protected]
Stack

main | a | 42 | | b | -7 |

swap | a | 42 | | b | -7 |
       | tmp | **42** |

Heap
Read/Write Segment *.data*, *.bss*
Read-Only Segment *.text*, *.rodata*

Broken Swap

brokenswap.c

```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);
  ...
```

OS kernel [protected]
Stack

main | a | 42 | | b | -7 |

swap | a | **-7** | | b | -7 |
tmp | 42 |

Heap
Read/Write Segment *.data*, *.bss*
Read-Only Segment *.text*, *.rodata*

33

Broken Swap

brokenswap.c

```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);
  ...
```

OS kernel [protected]
Stack

main | a | 42 | | b | -7 |

swap | a | -7 | | b | **42** |
tmp | 42 |

| Heap |
| Read/Write Segment
.data, .bss |
| Read-Only Segment
.text, .rodata |

Broken Swap

brokenswap.c

```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);
  ...
```
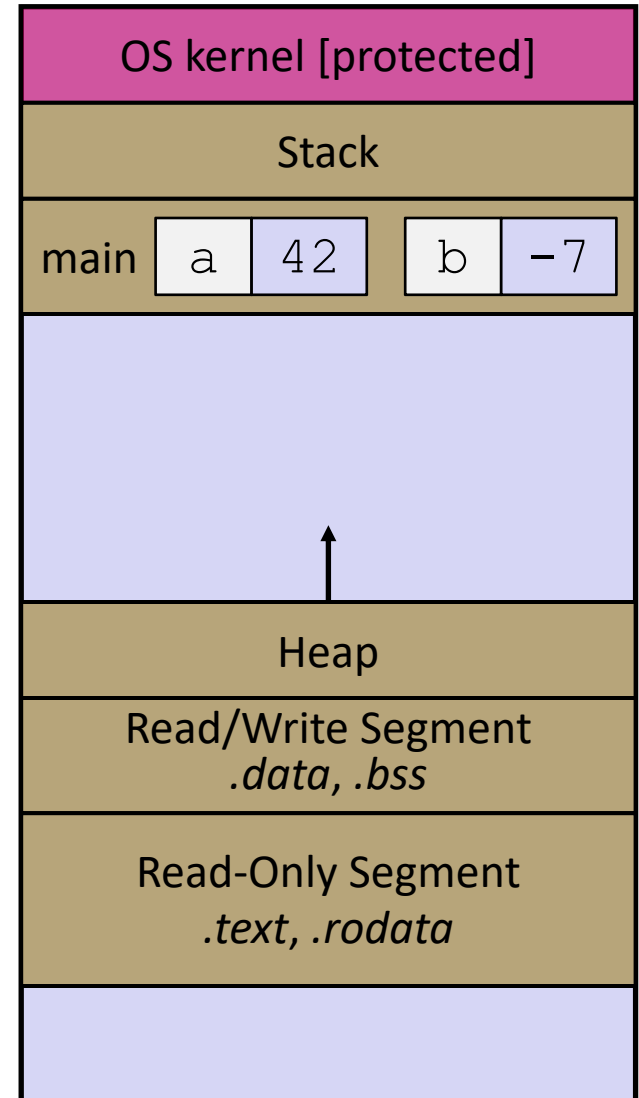
OS kernel [protected]

Stack

main a 42   b -7

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata

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

```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);
  ...
```
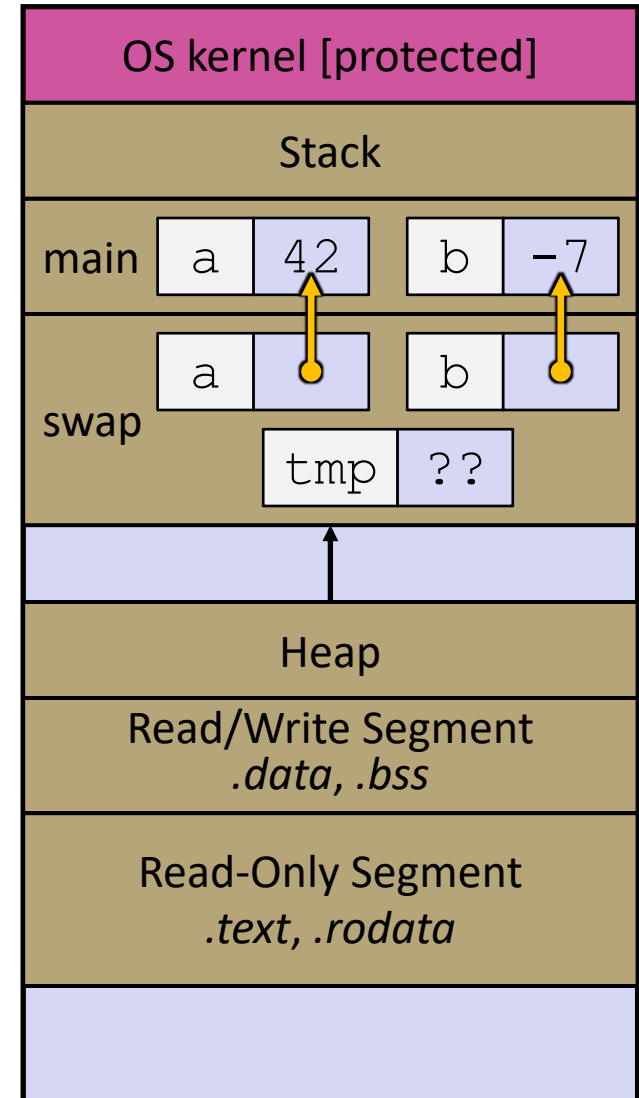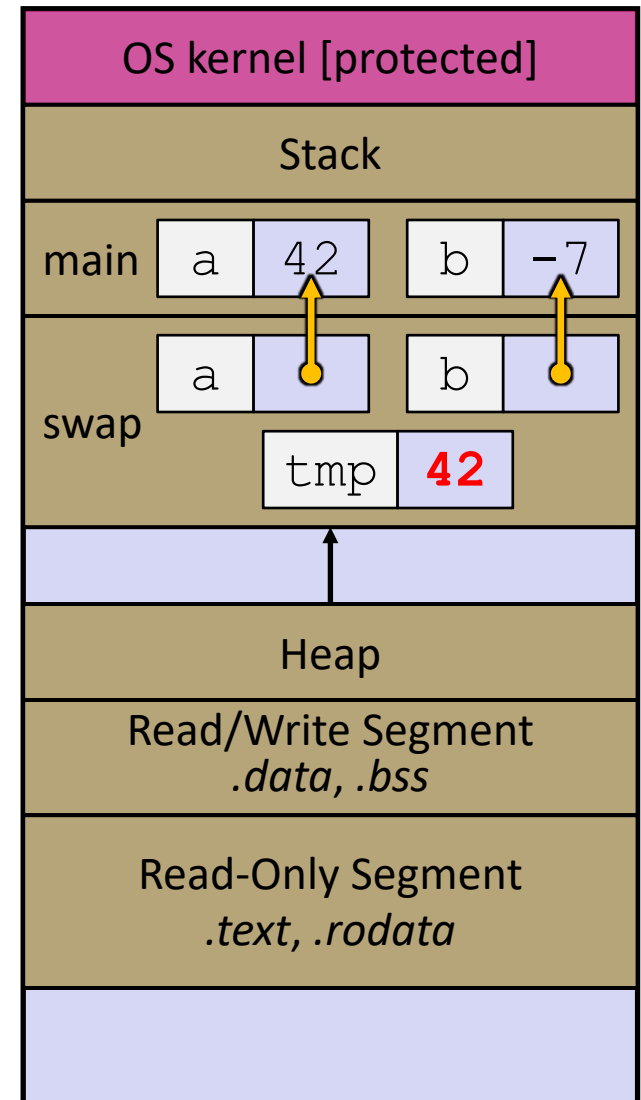
Fixed Swap

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);
  ...
```

OS kernel [protected]

Stack

main   a 42   b -7

Heap

Read/Write Segment
.data, *.bss*
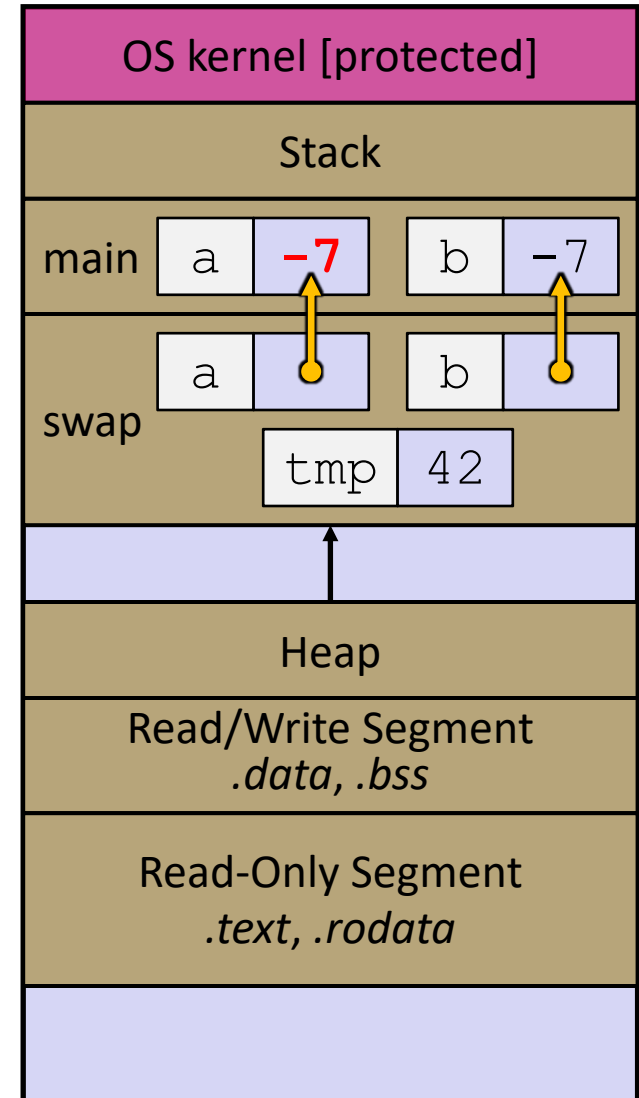
Read-Only Segment
.text, *.rodata*

Fixed Swap

swap.c

```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);
  ...
```
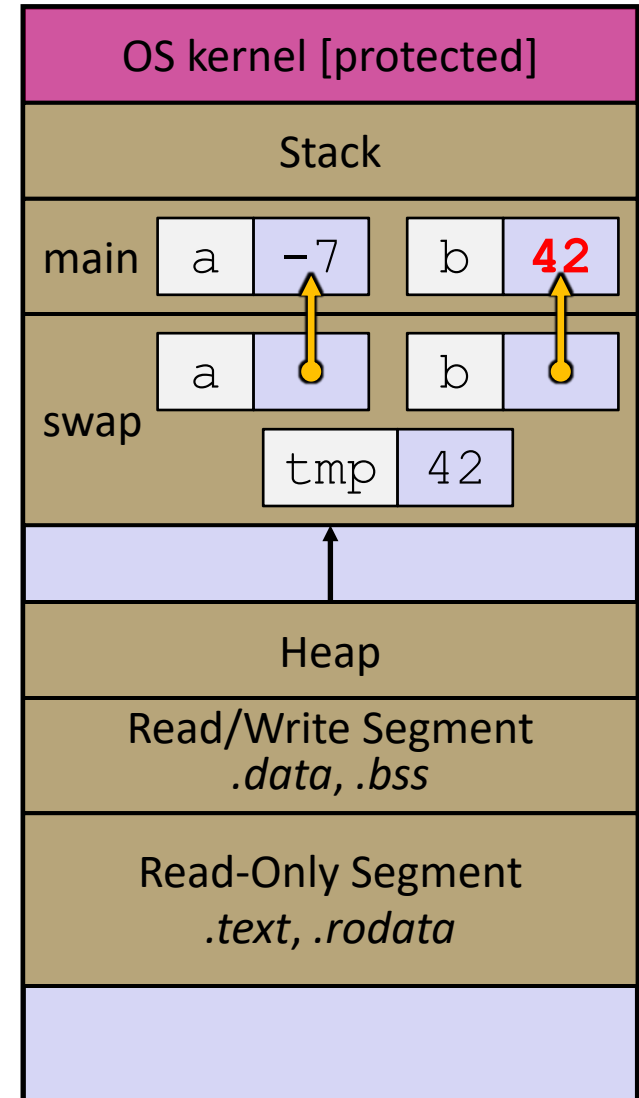
Fixed Swap

swap.c

```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);
  ...
```

OS kernel [protected]

Stack

main   a 42   b -7

swap   a   b

        tmp **42**

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata

Fixed Swap

swap.c

```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);
  ...
```
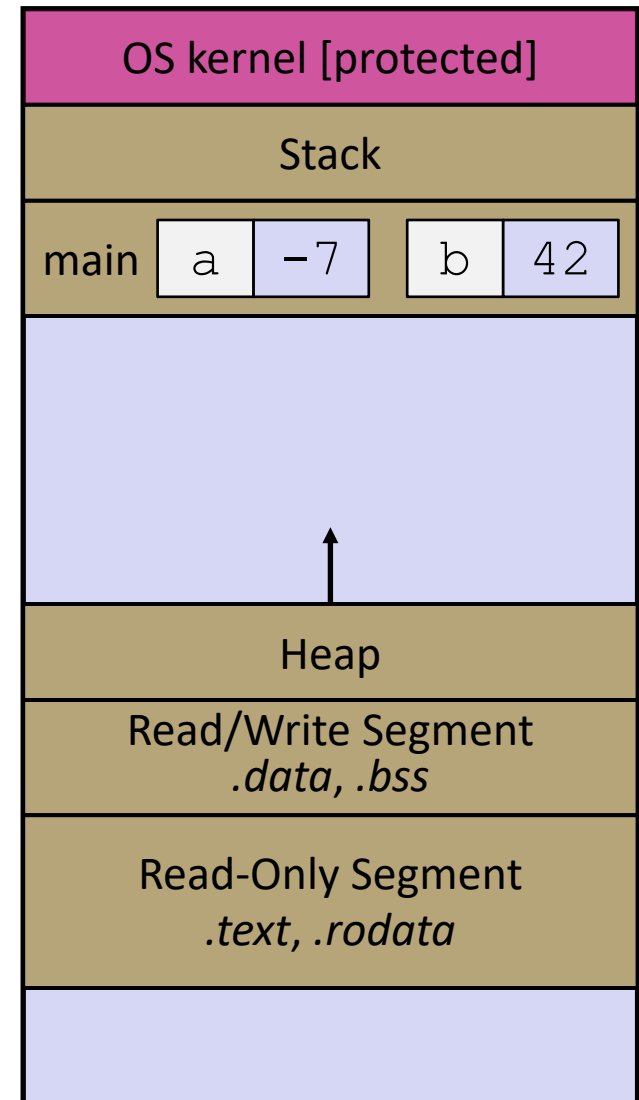

40

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);
  ...
```

| OS kernel [protected] |
| Stack |

main   a   -7   b   **42**

swap   a   b

tmp   42

| Heap |
| Read/Write Segment
.data, *.bss* |
| Read-Only Segment
.text, *.rodata* |

Fixed Swap

swap.c

```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);
  ...
```

OS kernel [protected]

Stack

main   a   -7   b   42

Heap

Read/Write Segment
.data, *.bss*

Read-Only Segment
.text, *.rodata*

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 value *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 a pointer to that space to Callee (*e.g.,* `&var`)
  3) Callee has an output parameter (*e.g.,* `type* outparam`)
  4) Callee uses parameter to store data in space provided by caller (*e.g.,* `*outparam = value;`)
  5) Caller accesses output via modified data (*e.g.,* `var`)

43

Lecture Outline

- ❖ Pointers & Pointer Arithmetic

- ❖ Pointers as Parameters
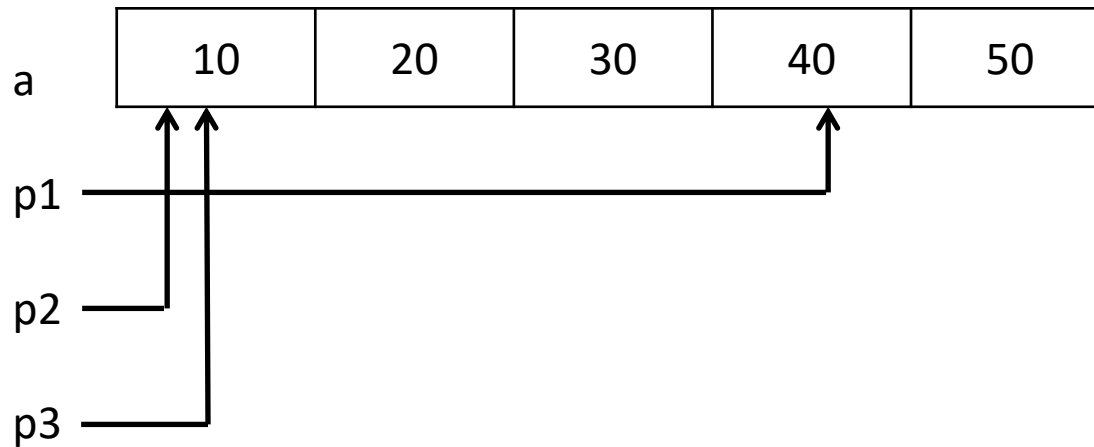
- ❖ **Pointers and Arrays**

- ❖ 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)` using 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

```c
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
```
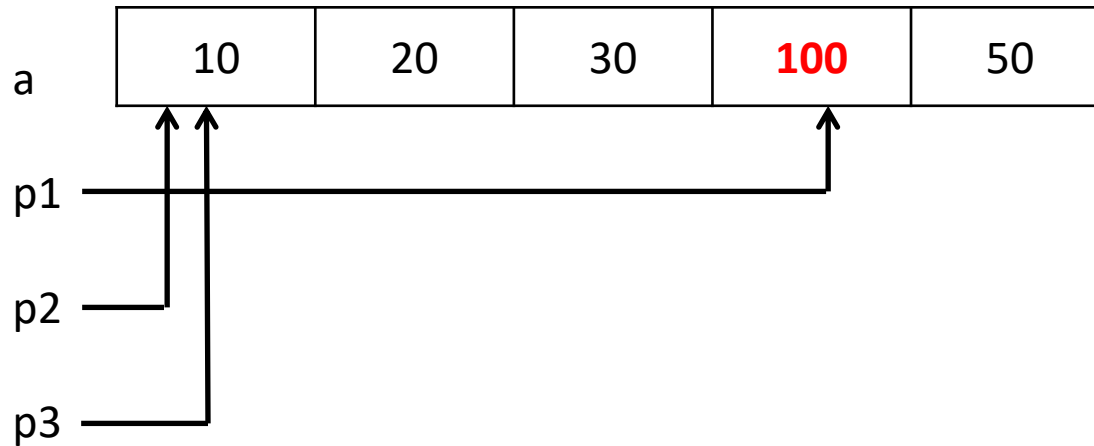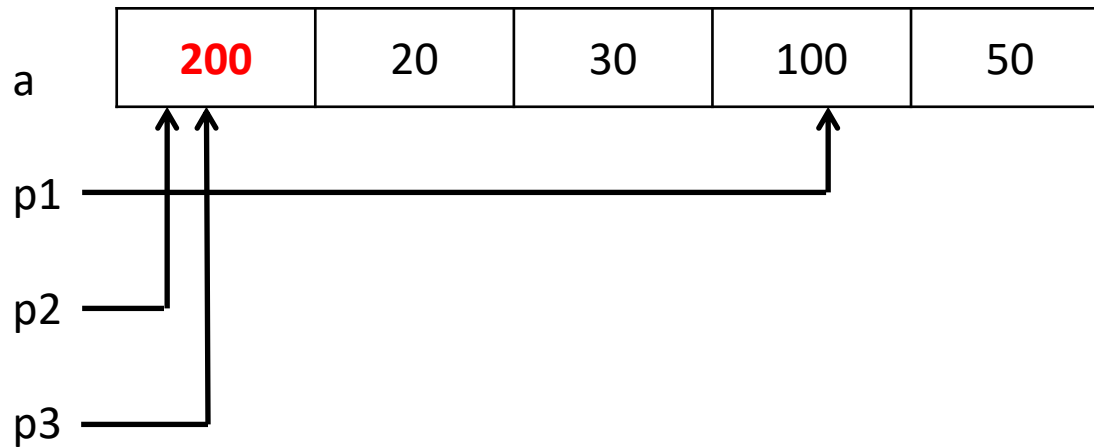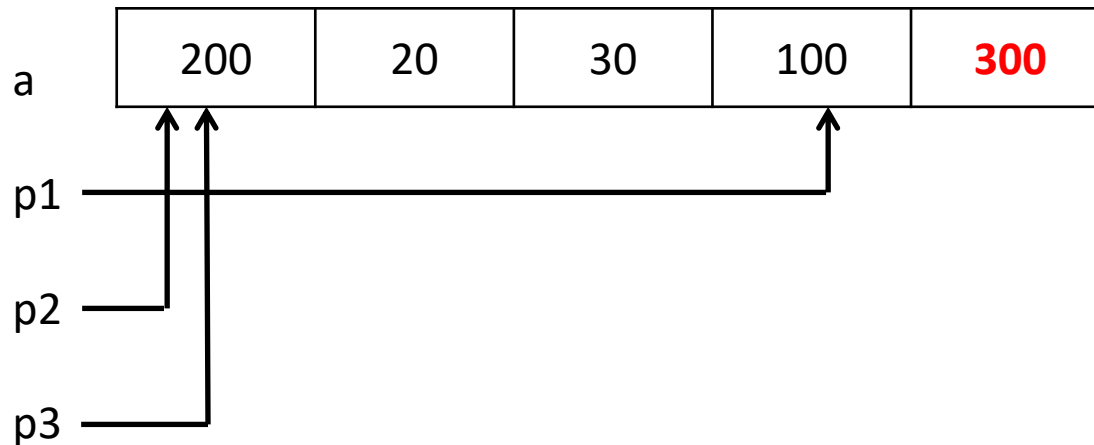
Pointers and Arrays - Trace


```
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

a	10	20	30	**100**	50

p1

p2

p3

```
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

a	**200**	20	30	100	50

p1

p2

p3

```
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

200	20	30	100	**300**

a

p1

p2

p3

```
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
```
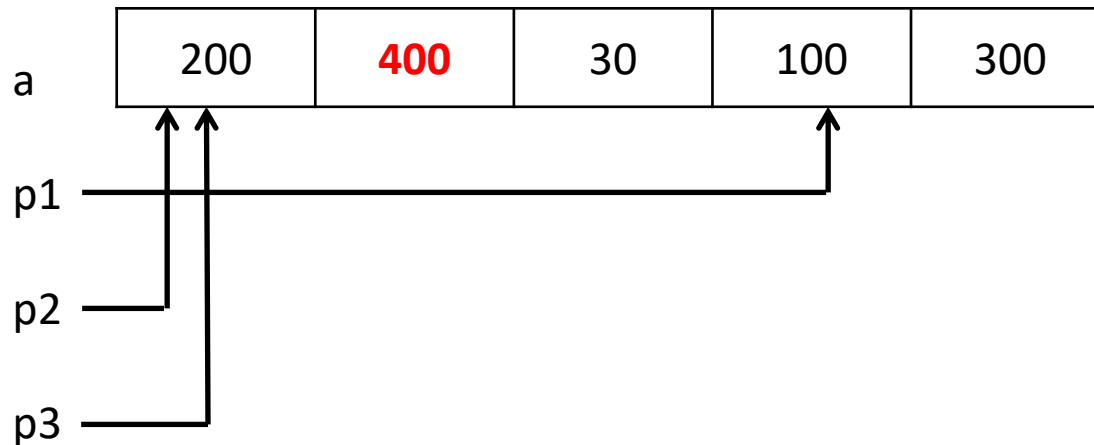
Pointers and Arrays - Trace

a	200	**400**	30	100	300

p1

p2

p3

```
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
```
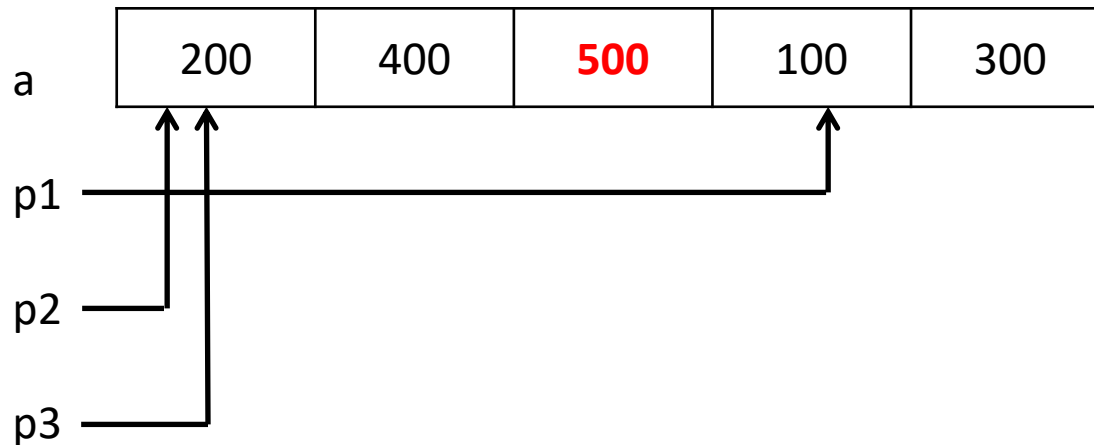
Pointers and Arrays - Trace

a	200	400	**500**	100	300

p1

p2

p3

```
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

```c
#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

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:

```c
#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 EXIT_SUCCESS;
}
```

Extra Exercise #2

❖ Write a program that determines and prints out whether the computer it is running on is little-endian or big-endian.

  ▪ <u>Hint</u>: `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