Pointers and Pointer Applications

Lecture Outline

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

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[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
  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

```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[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
  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

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

stack frame for `main()`

Box-and-Arrow Diagrams

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[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
  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

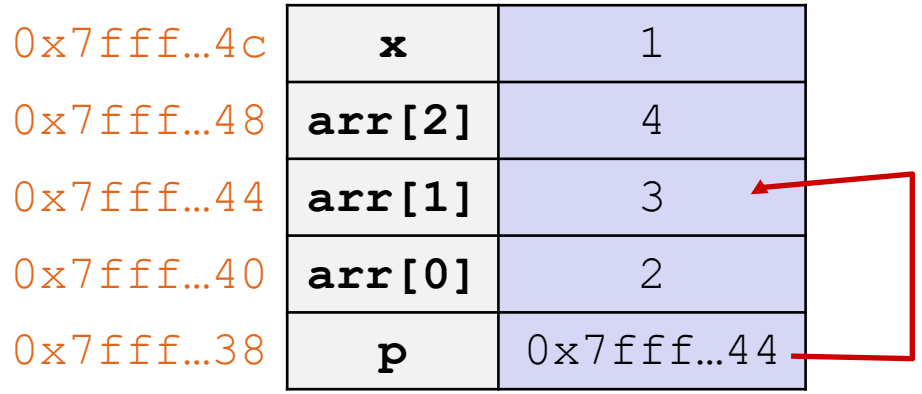
&x	**x**	1
&arr[2]	**arr[2]**	4
&arr[1]	**arr[1]**	3
&arr[0]	**arr[0]**	2
&p	**p**	&arr[1]

Box-and-Arrow Diagrams

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[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
  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



Test runs in 32 bits and 64 bits systems

```
cs257@cs257-VirtualBox:~/Desktop/programs$ ./boxarrow
&x:      0xbfd7e398;  x:      1
&arr[0]: 0xbfd7e3a0;  arr[0]: 2
&arr[1]: 0xbfd7e3a4;  arr[1]: 3
&arr[2]: 0xbfd7e3a8;  arr[2]: 4
&p: 0xbfd7e39c;  p: 0xbfd7e3a4; *p: 3
cs257@cs257-VirtualBox:~/Desktop/programs$ ./boxarrow
&x:      0xbfed03a8;  x:      1
&arr[0]: 0xbfed03b0;  arr[0]: 2
&arr[1]: 0xbfed03b4;  arr[1]: 3
&arr[2]: 0xbfed03b8;  arr[2]: 4
&p: 0xbfed03ac;  p: 0xbfed03b4; *p: 3
cs257@cs257-VirtualBox:~/Desktop/prog
```

32 bits (VM)


```
[sonmeza@cmsc257 code]$ ./boxarrow
&x:       0x7ffe40bc76ac;  x:       1
&arr[0]: 0x7ffe40bc76a0;  arr[0]: 2
&arr[1]: 0x7ffe40bc76a4;  arr[1]: 3
&arr[2]: 0x7ffe40bc76a8;  arr[2]: 4
&p: 0x7ffe40bc7698; p: 0x7ffe40bc76a4; *p: 3
[sonmeza@cmsc257 code]$ ./boxarrow
&x:       0x7ffd8538886c;  x:       1
&arr[0]: 0x7ffd85388860;  arr[0]: 2
&arr[1]: 0x7ffd85388864;  arr[1]: 3
&arr[2]: 0x7ffd85388868;  arr[2]: 4
&p: 0x7ffd85388858; p: 0x7ffd85388864; *p: 3
[sonmeza@cmsc257 code]$
```

64 bits (server)

Test runs in 32 bits and 64 bits systems

```
cs257@cs257-
&x:      0xb
&arr[0]: 0xb
&arr[1]: 0xb
&arr[2]: 0xb
&p: 0xbfd7e3
cs257@cs257-
&x:      0xb
&arr[0]: 0xb
&arr[1]: 0xb
&arr[2]: 0xb
&p: 0xbfed03
cs257@cs257-
```

C (gcc 4.8, C11)
EXPERIMENTAL! known limitations

```
1  int main(int argc, char** argv) {
2    int x = 1;
3    int arr[3] = {2, 3, 4};
4    int* p = &arr[1];
5
6    printf("&x: %p;  x: %d\n", &x, x);
7    printf("&arr[0]: %p;  arr[0]: %d\n", &arr[0], arr[0]);
8    printf("&arr[1]: %p;  arr[1]: %d\n", &arr[1], arr[1]);
9    printf("&arr[2]: %p;  arr[2]: %d\n", &arr[2], arr[2]);
10   printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);
11
12   return 0;
13 }
```

Edit this code

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Print output (drag lower right corner to resize)

Stack Heap

main
- argc: int ?
- argv: pointer ?
- x: int ?
- arr: array [0] int ? | [1] int ? | [2] int ?
- p: pointer ?

```
&p: 0x7ffd85388858; p: 0x7ffd85388864; *p: 3
[sonmeza@cmsc257 code]$
```

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)`
  - Returns size of data that is 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`

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[]`?

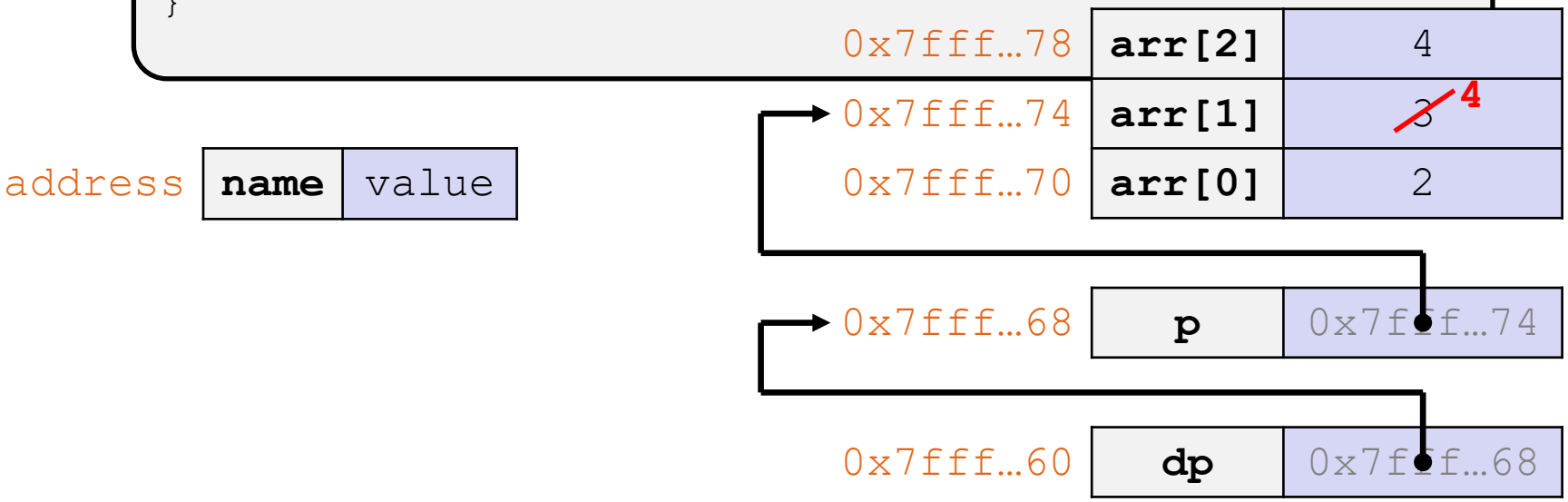
address	name	value
0x7fff…78	arr[2]	4
0x7fff…74	arr[1]	3
0x7fff…70	arr[0]	2
0x7fff…68	p	0x7fff…74
0x7fff…60	dp	0x7fff…68

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;
}
```

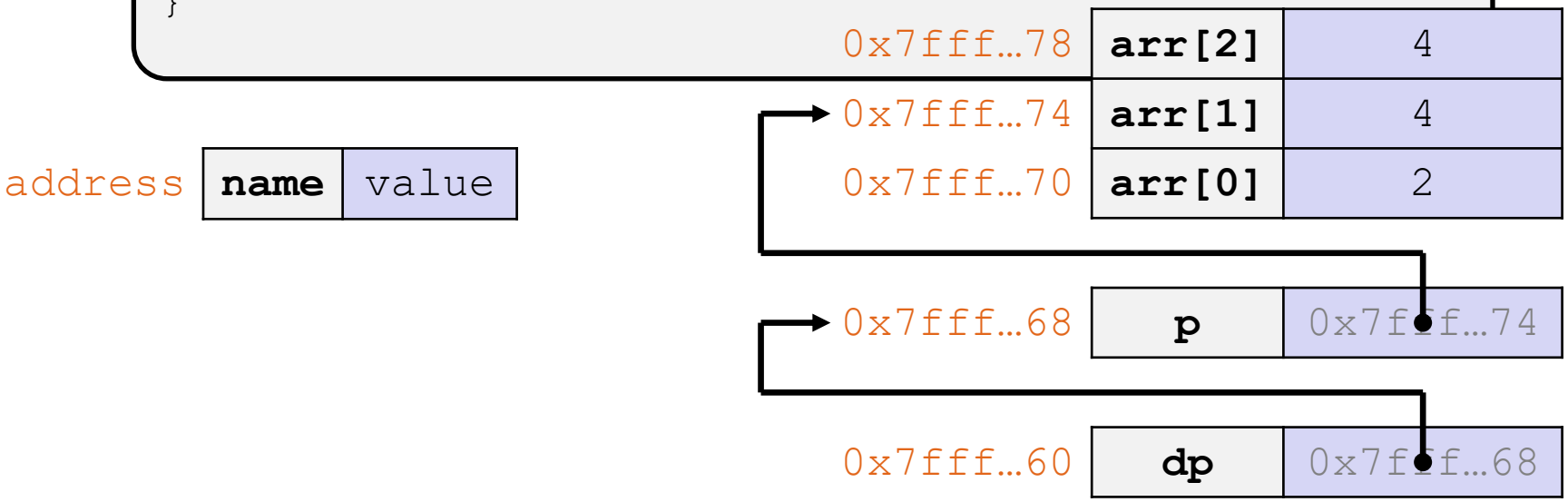


Practice Solution

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;
}
```

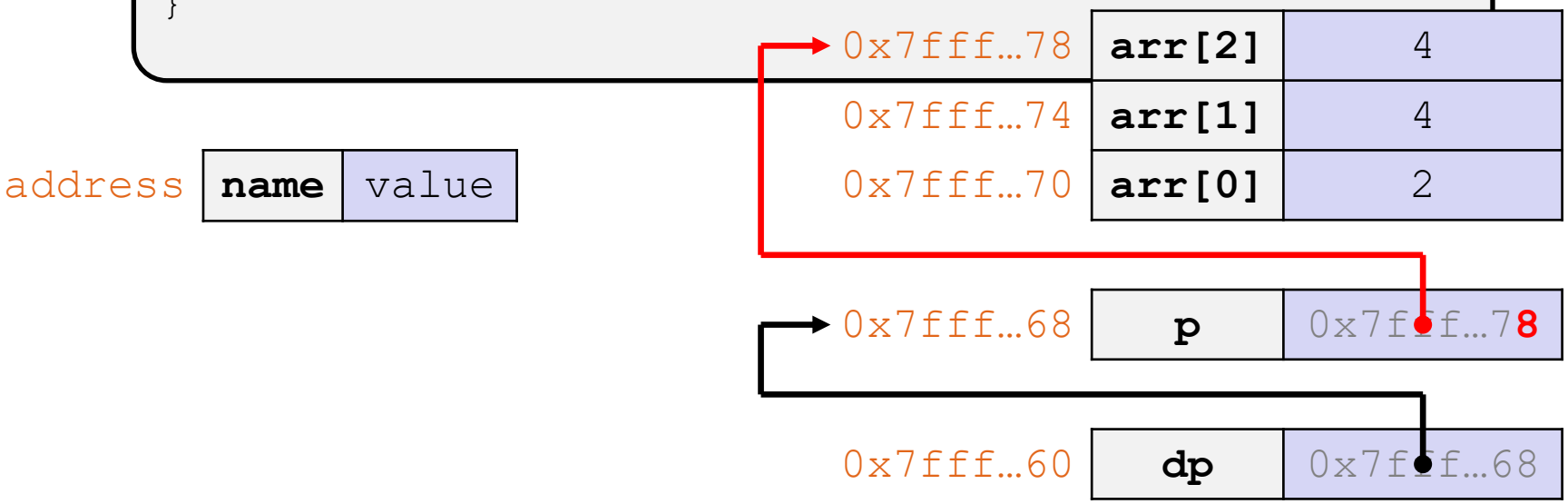


Practice Solution

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;
}
```

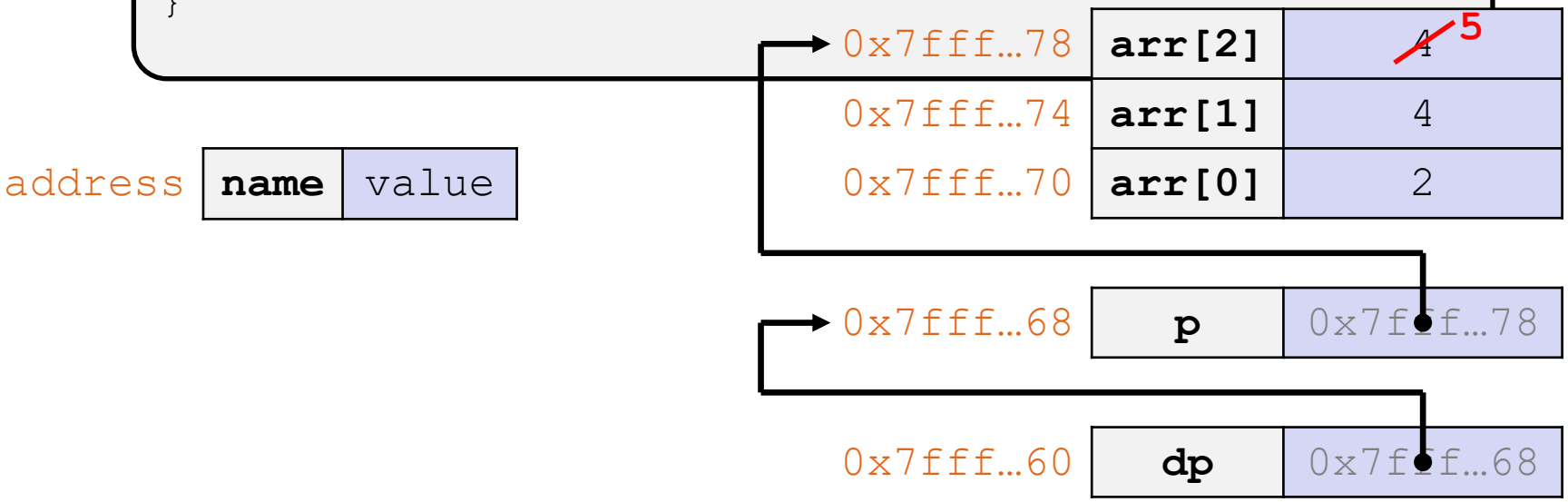


Practice Solution

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

boxarrow2.c

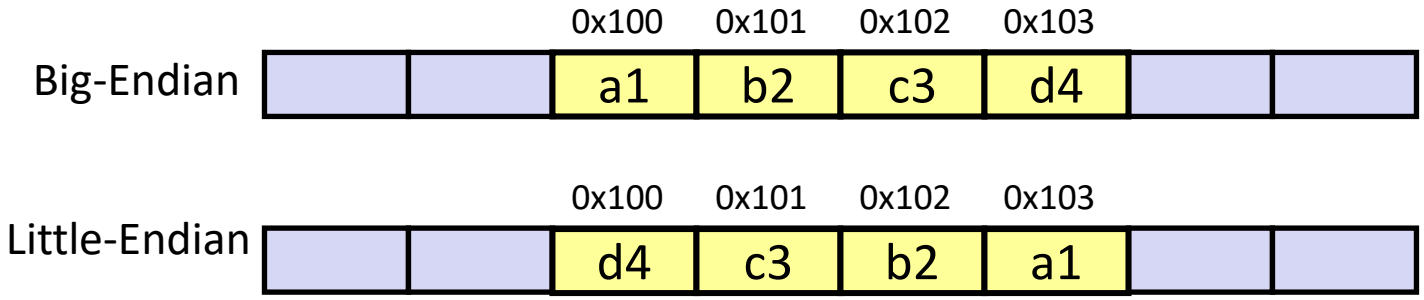
```
int main(int argc, char** argv) {
  int arr[3] = {2, 3, 4};
  int* p = &arr[1];
  int** dp = &p;  // pointer to a pointer

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

  return 0;
}
```



Endianness

- Memory is byte-addressed, 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



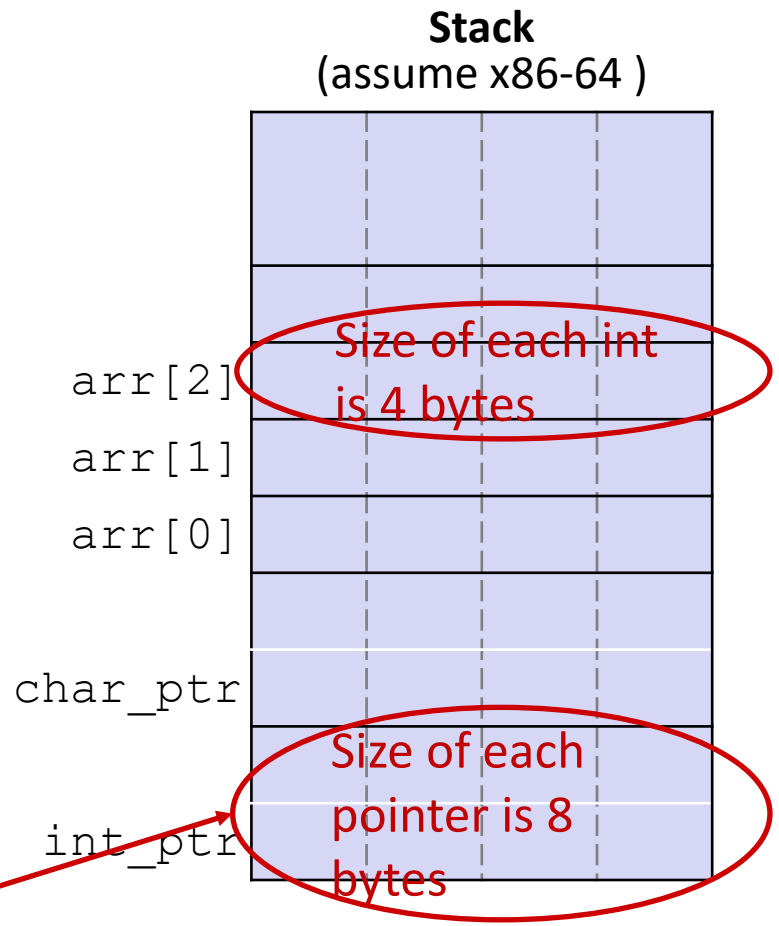
- For more: https://www.geeksforgeeks.org/little-and-big-endian-mystery/

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



This is a 64 bit system

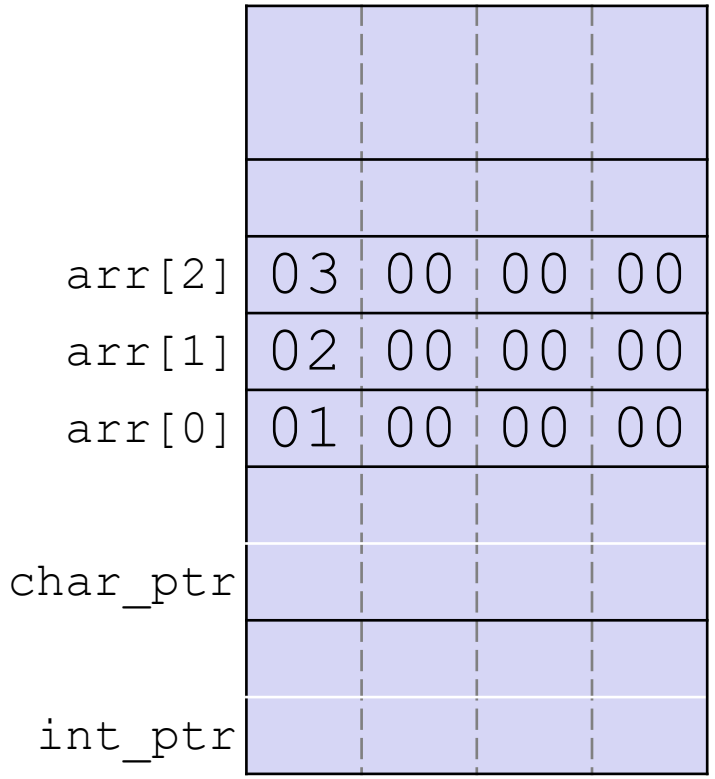
Pointer Arithmetic Example

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)

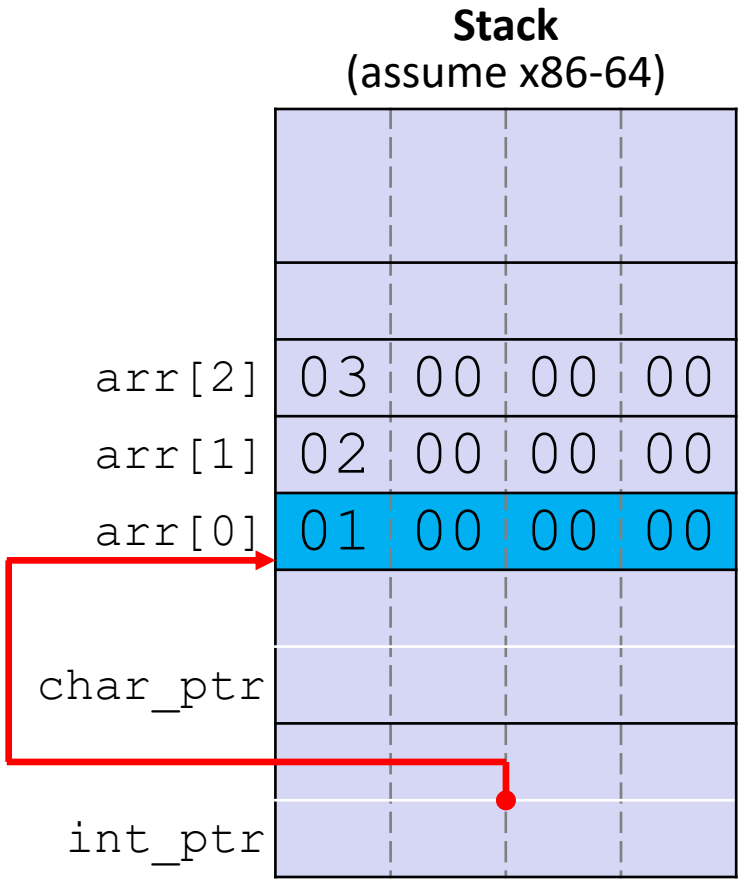


Pointer Arithmetic Example

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

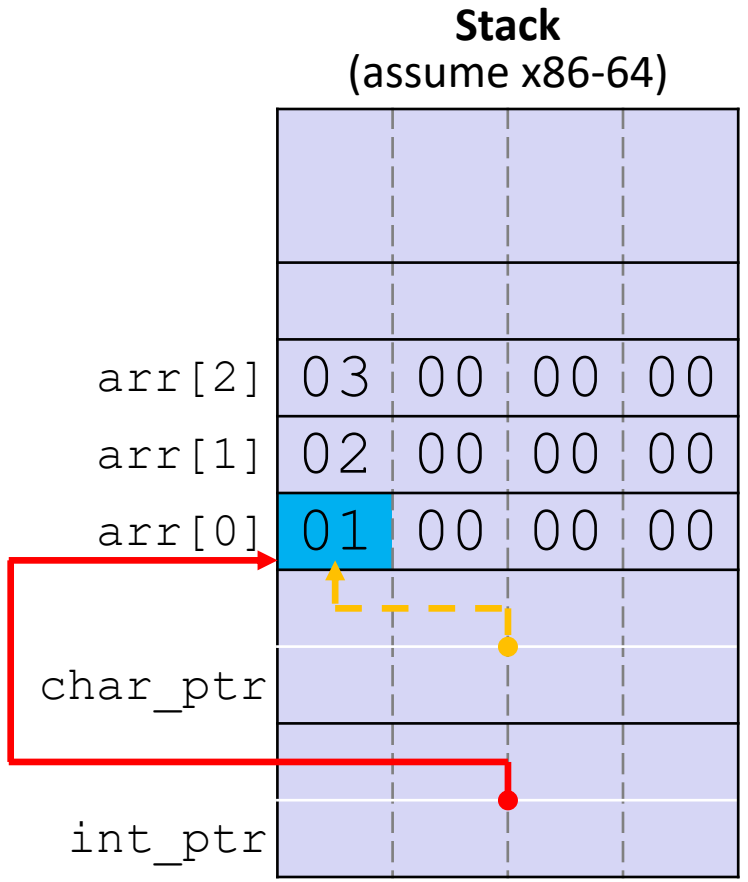


Pointer Arithmetic Example

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

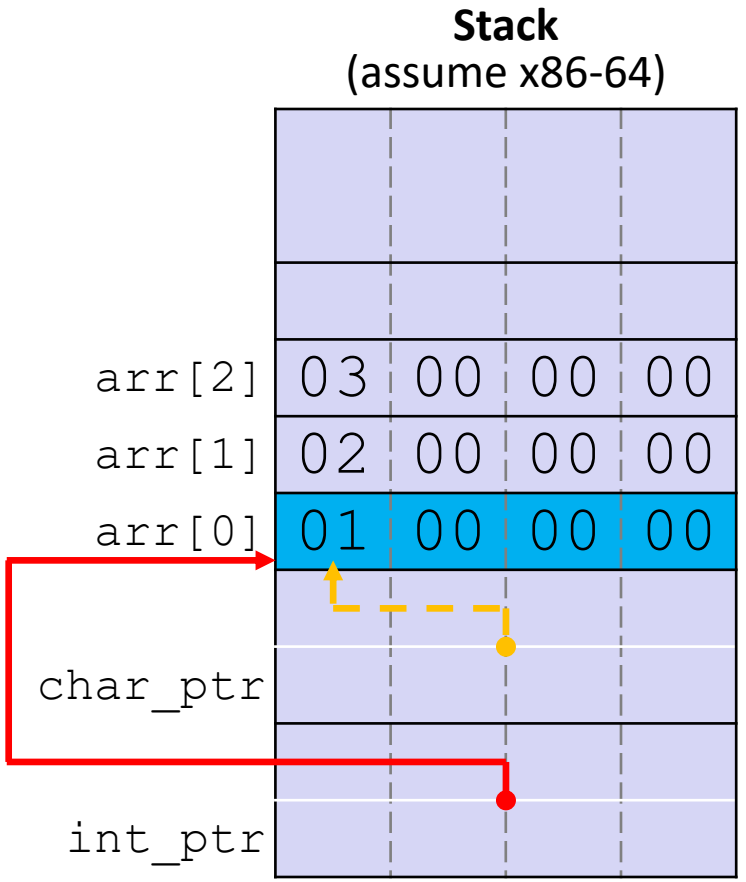


Pointer Arithmetic Example

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



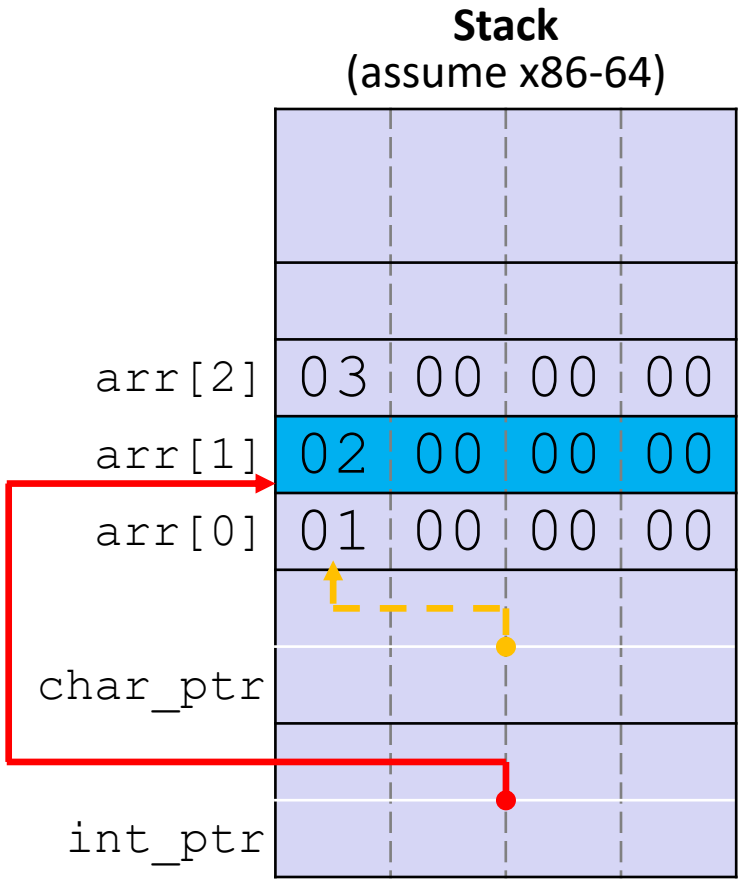
int_ptr: 0x0x7fffffde010
***int_ptr:** 1

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 0;
}
```

pointerarithmetic.c



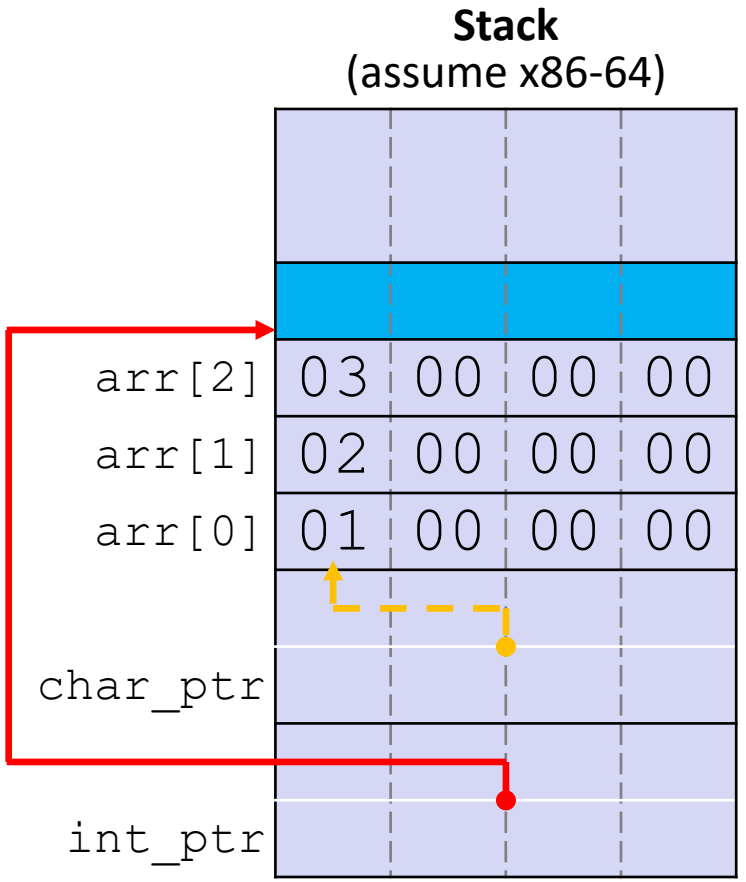
int_ptr: 0x0x7fffffde014
***int_ptr:** 2

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: 0x0x7fffffde01C

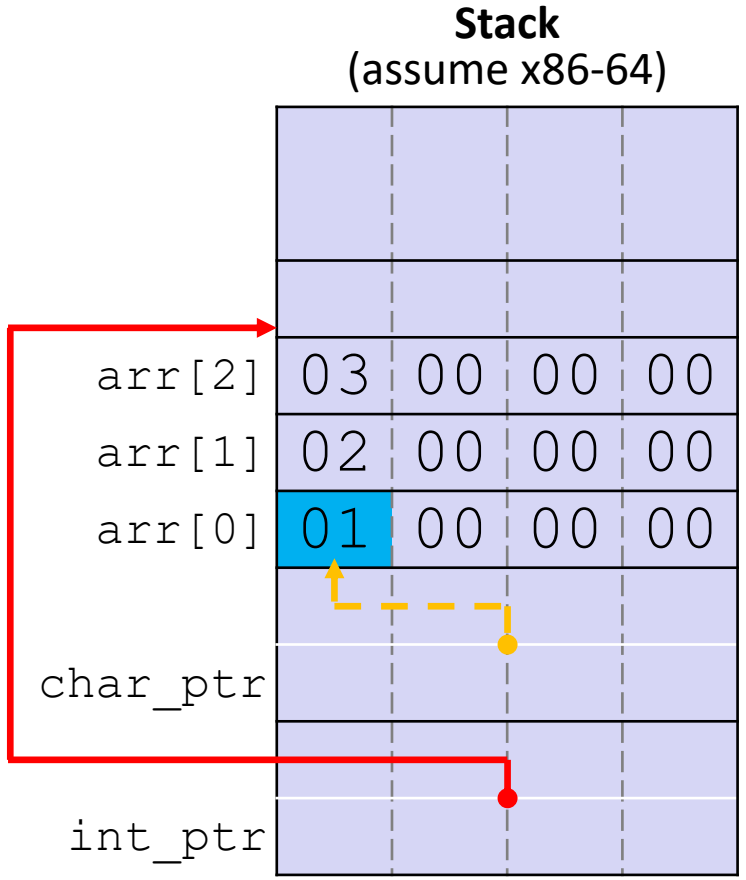
***int_ptr:** ???

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



char_ptr: 0x0x7fffffde010
***char_ptr:** 1 (SOH-start of heading character in ASCII)

ASCII Character Set

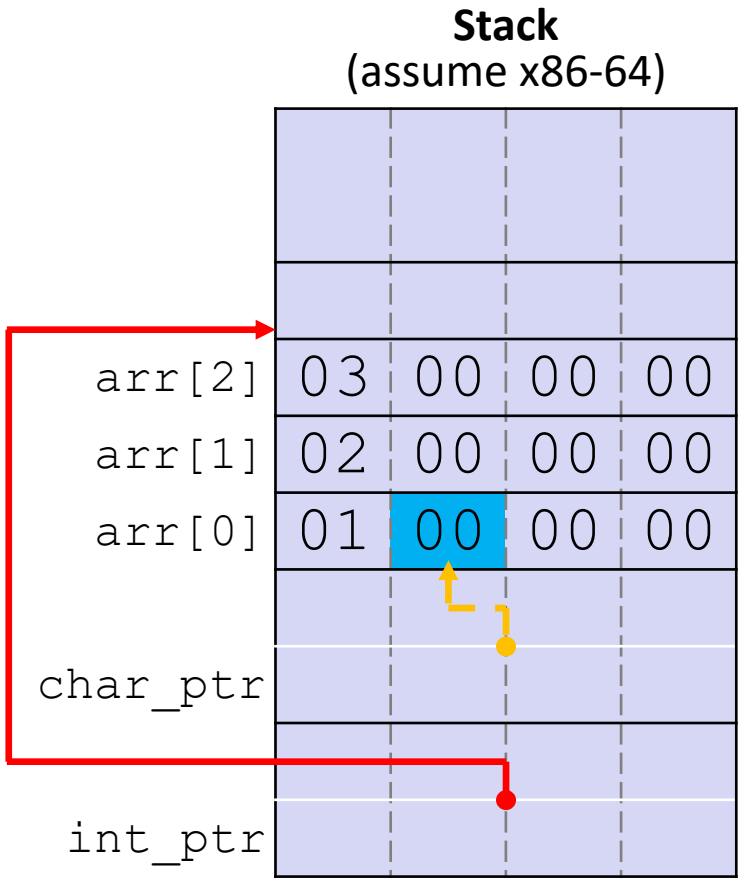
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B

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: 0x0x7fffffde011

***char_ptr:** **null**

ASCII Character Set

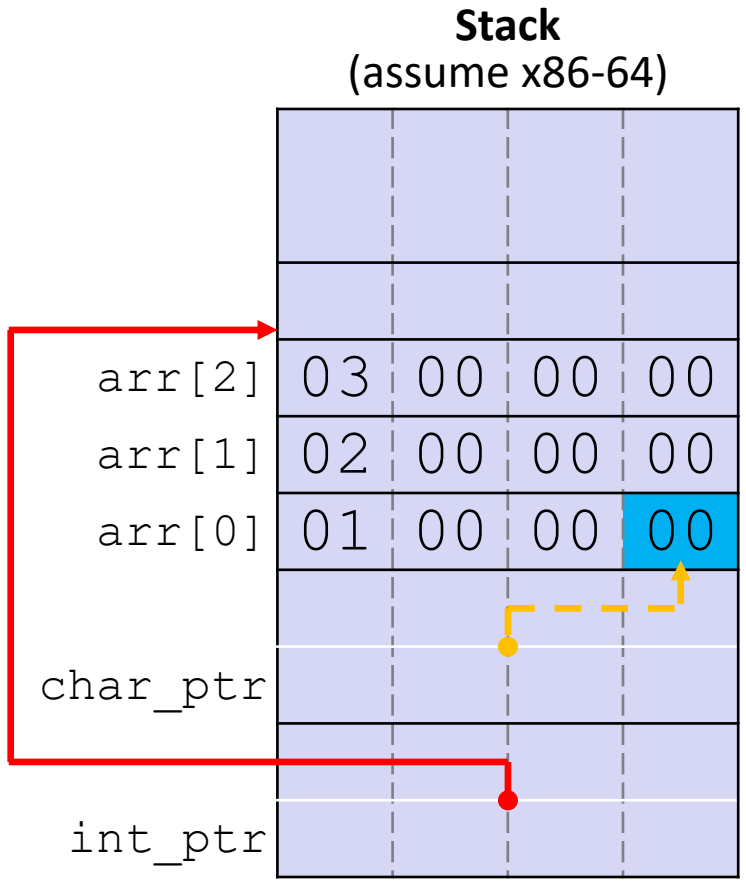
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B

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: 0x0x7fffffde013

***char_ptr:** **null**

Lecture Outline

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

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

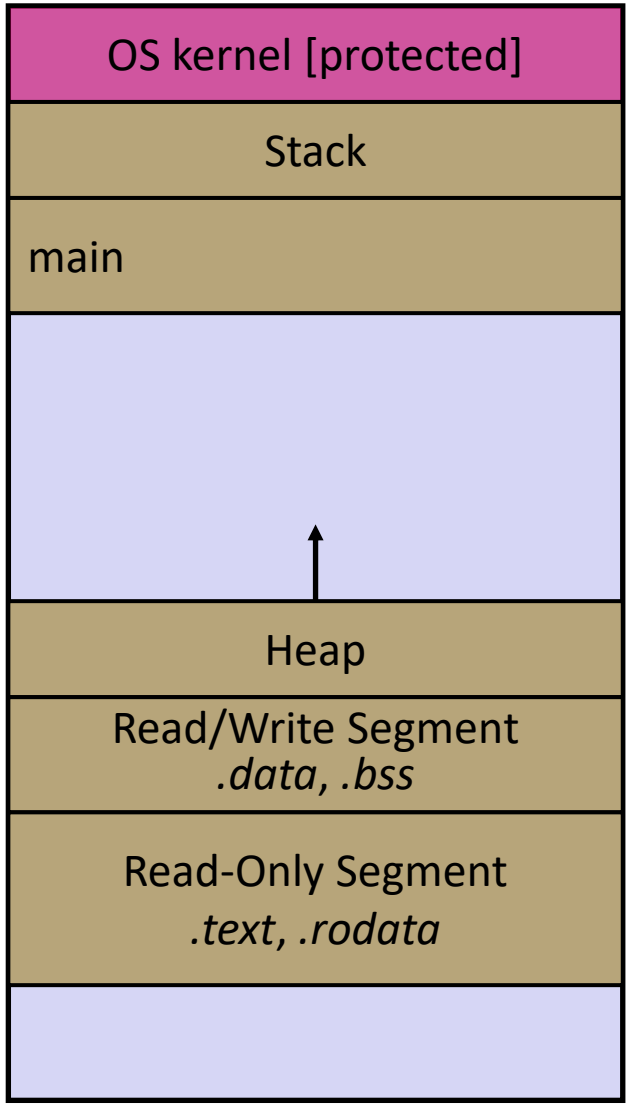
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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

Broken Swap

Note: Arrow points to *next* instruction.

brokenswap.c

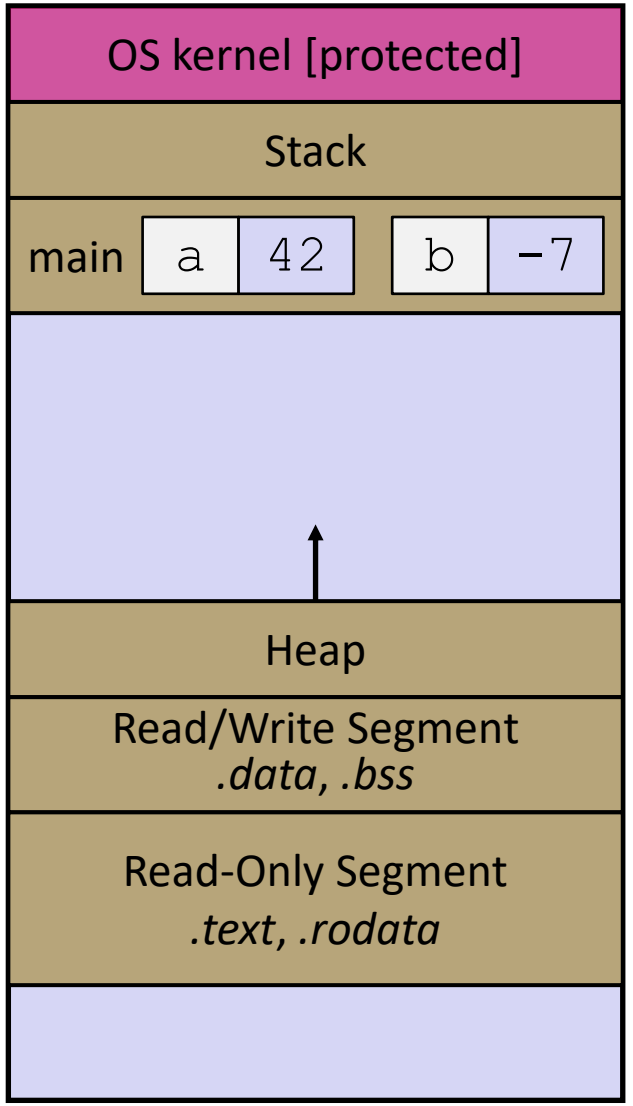
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



Broken Swap

brokenswap.c

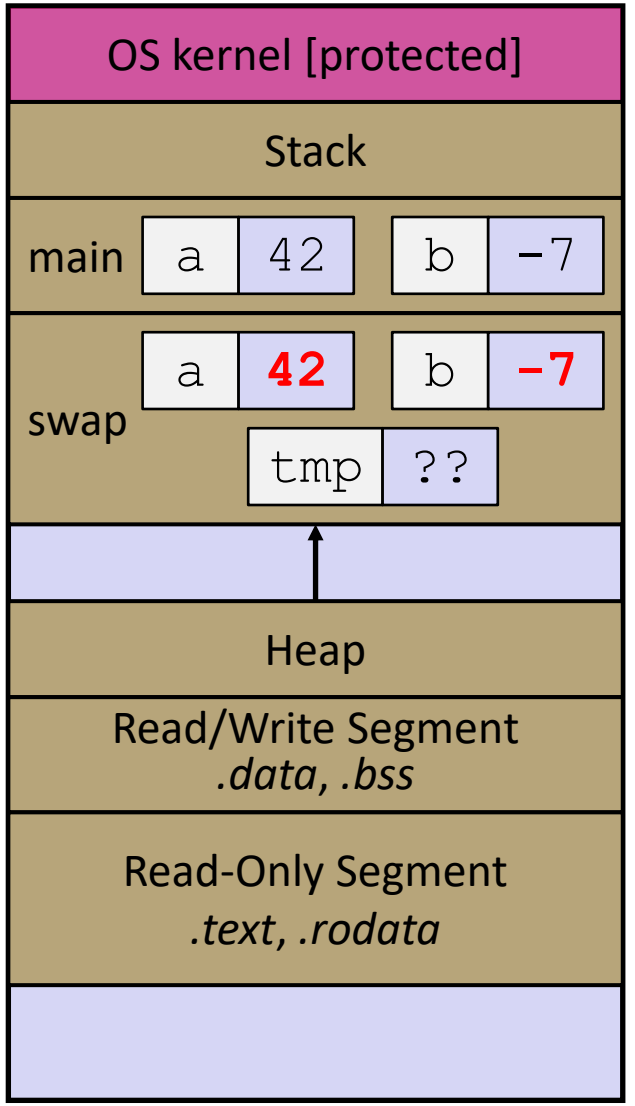
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



Broken Swap

brokenswap.c

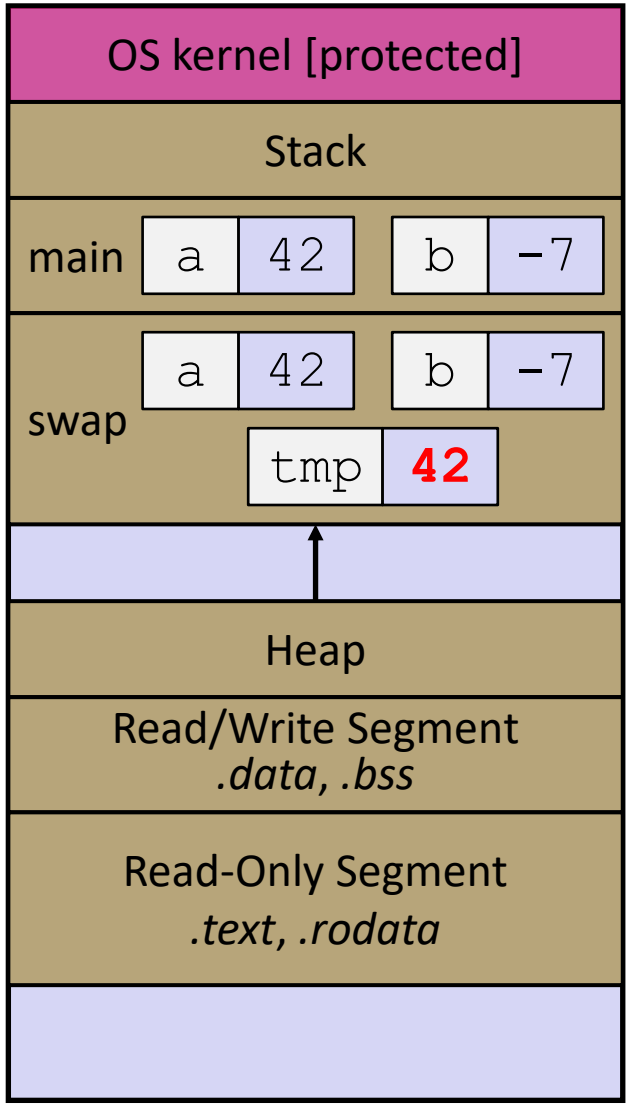
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



Broken Swap

brokenswap.c

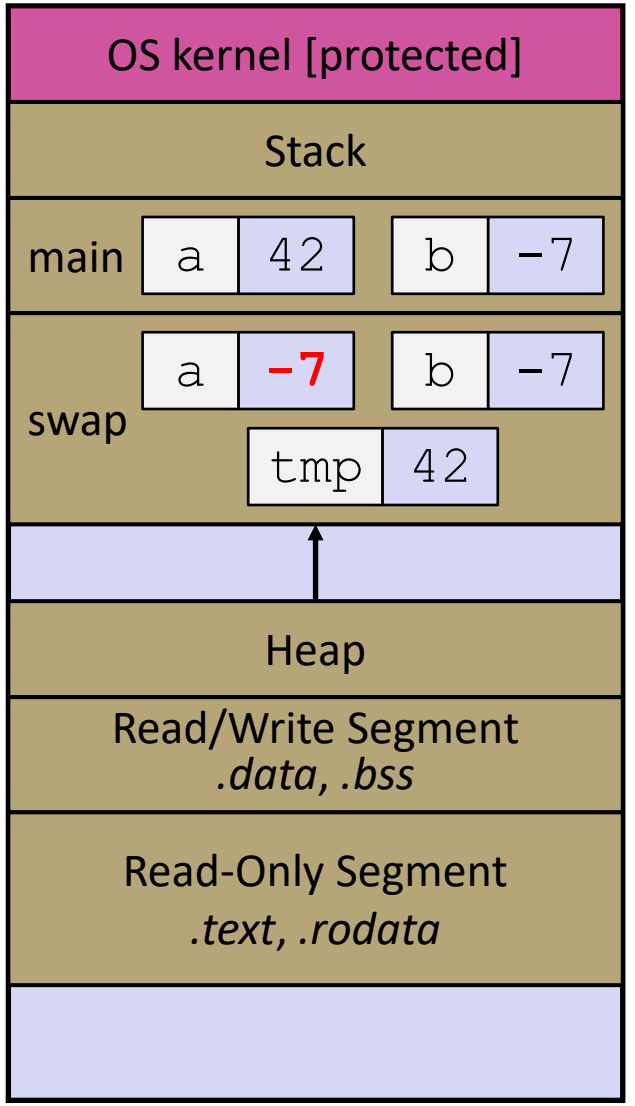
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



Broken Swap

brokenswap.c

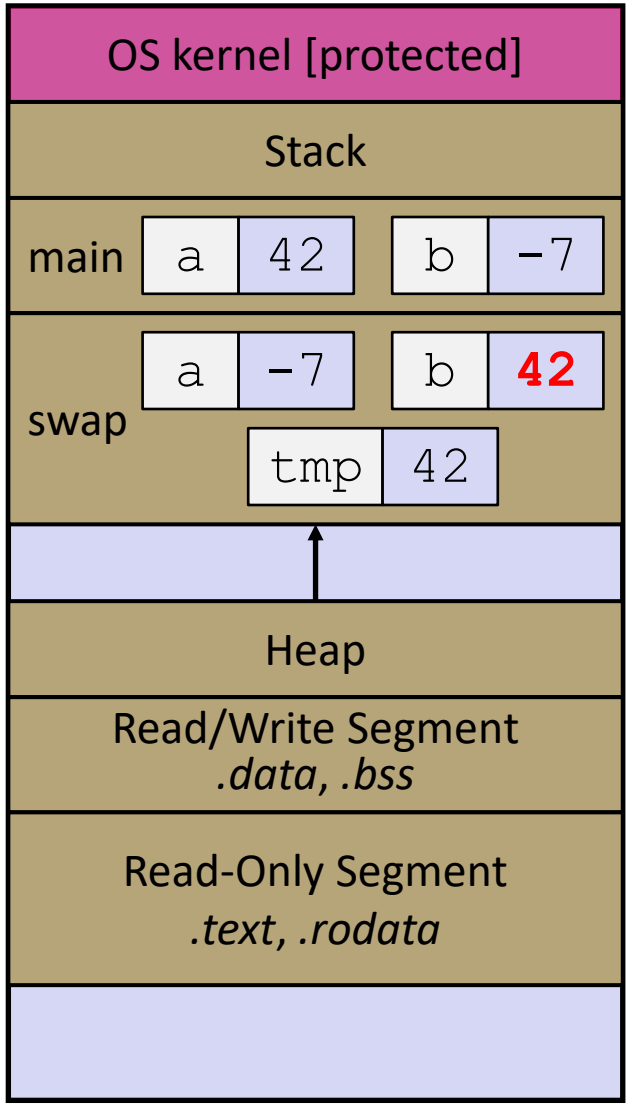
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



Broken Swap

brokenswap.c

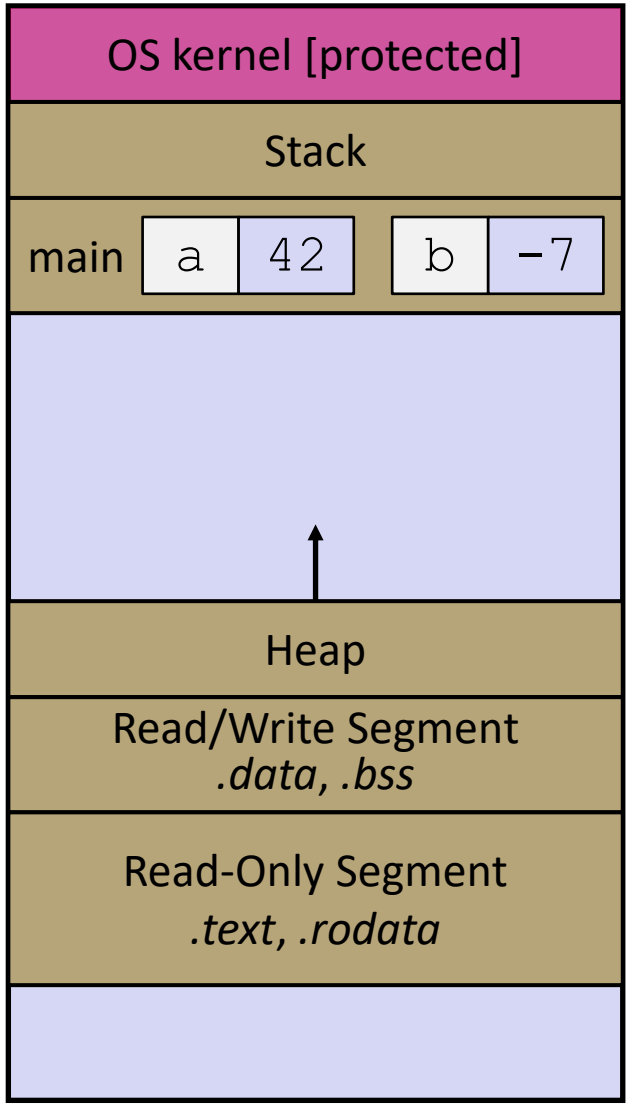
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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

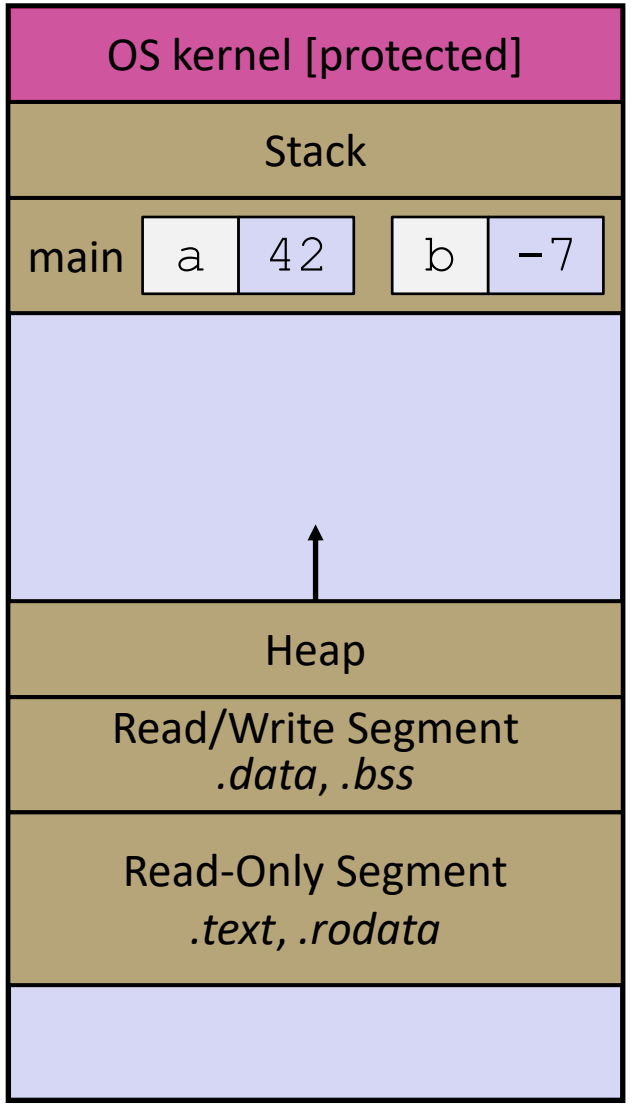
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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

Fixed Swap

Note: Arrow points to *next* instruction.

swap.c

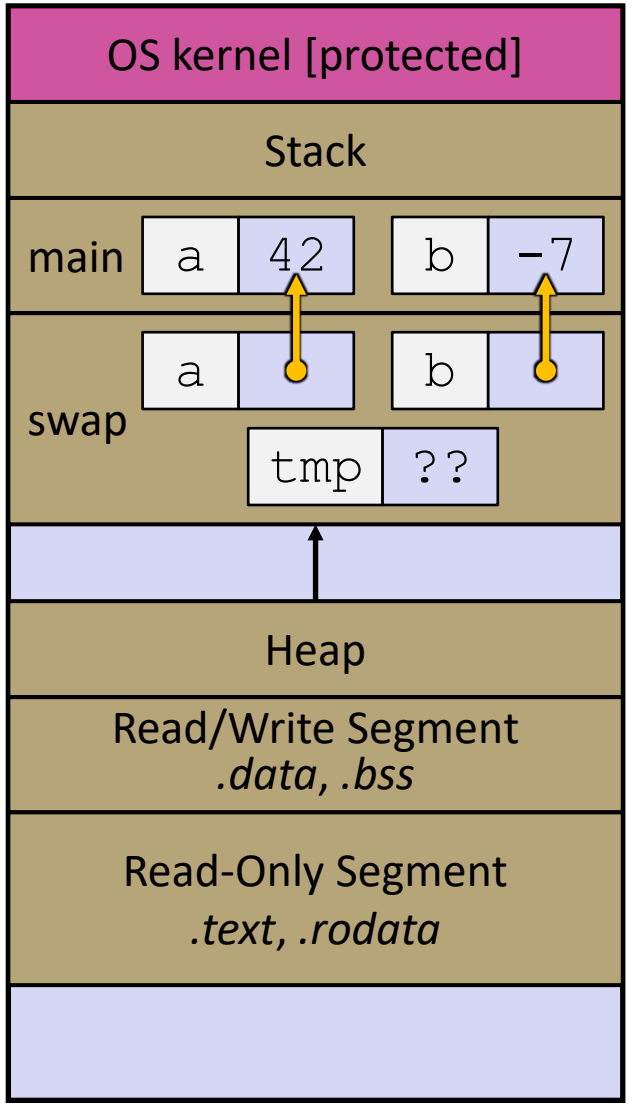
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



Fixed Swap

swap.c

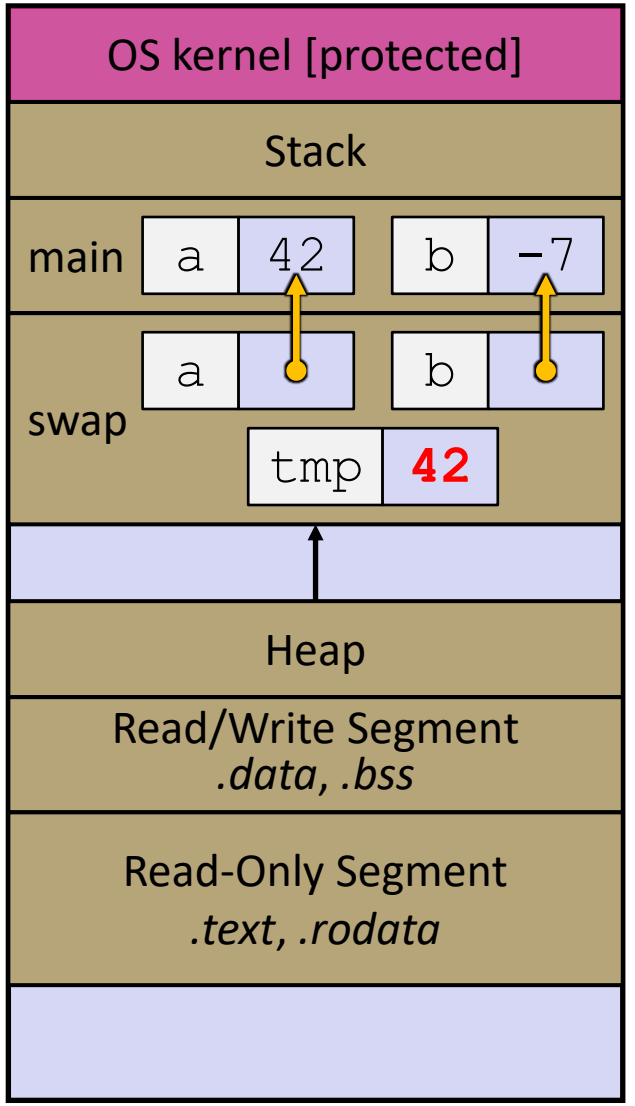
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



Fixed Swap

swap.c

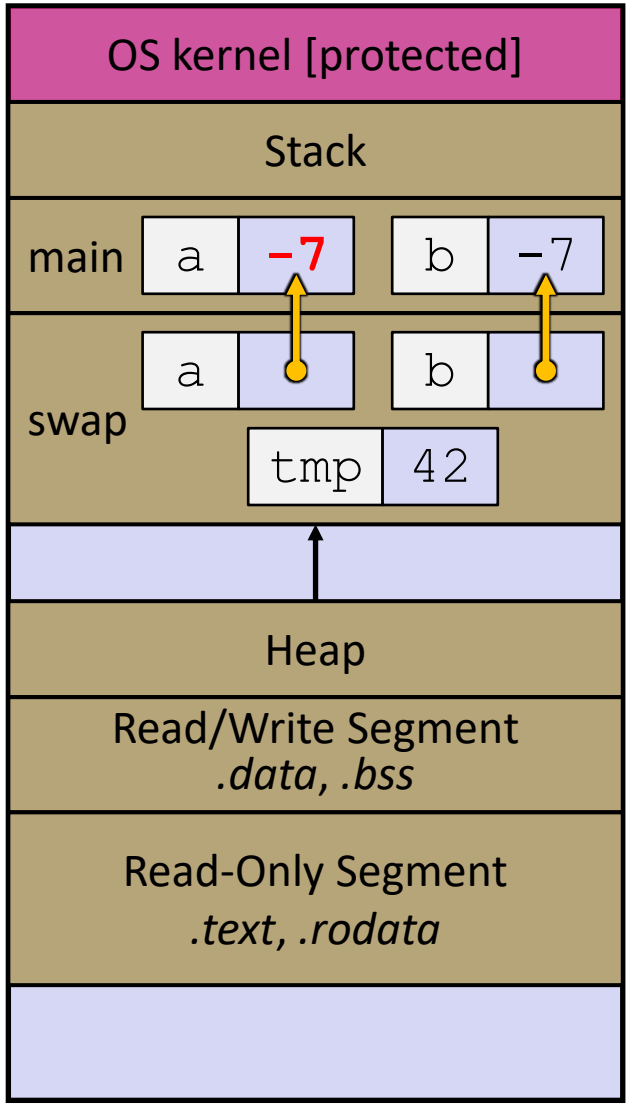
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



Fixed Swap

swap.c

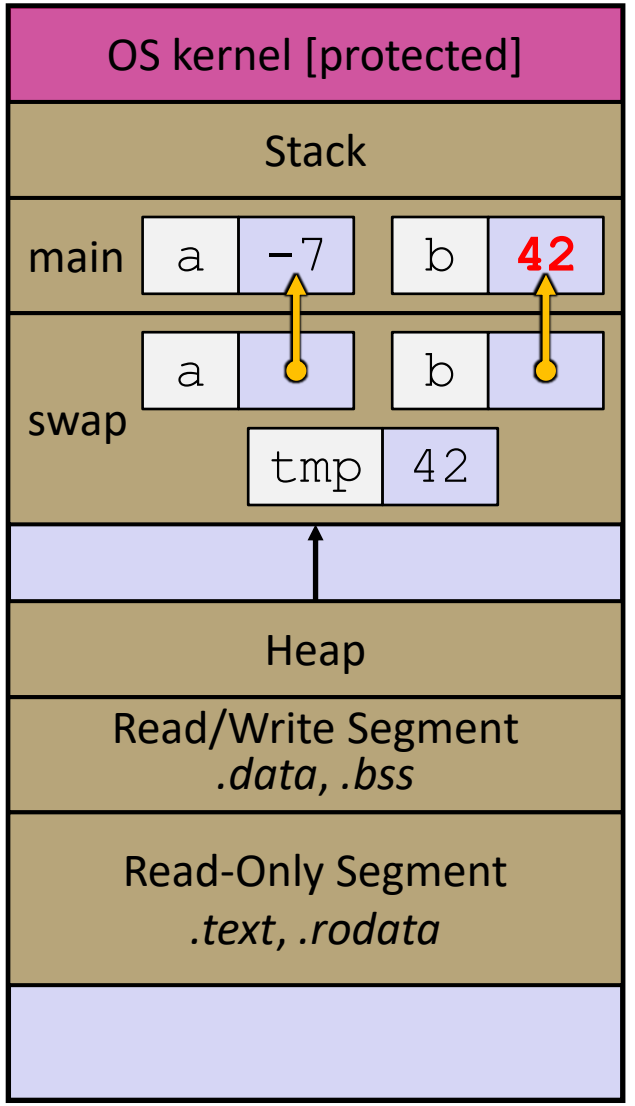
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



Fixed Swap

swap.c

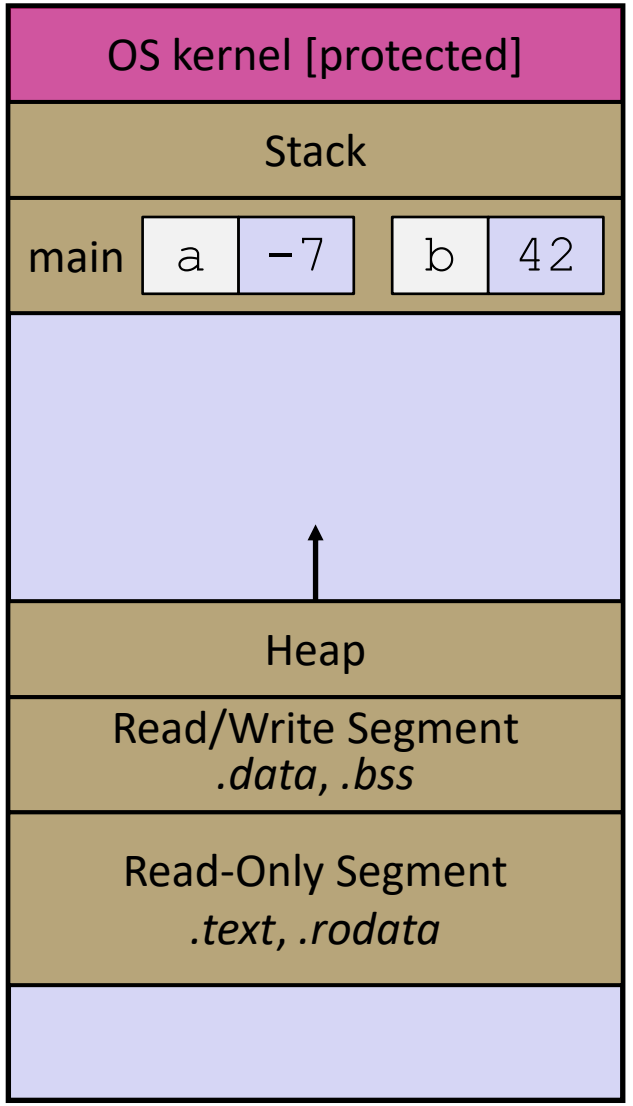
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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**

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

- 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

C (gcc 4.8, C11)
EXPERIMENTAL! known limitations

```
1  int main() {
2
3      int a[] = {10, 20, 30, 40, 50};
4      int* p1 = &a[3];  // refers to a's 4th element
5      int* p2 = &a[0];  // refers to a's 1st element
6      int* p3 = a;      // refers to a's 1st element
7
8      *p1 = 100;
9      *p2 = 200;
10     p1[1] = 300;
11     p2[1] = 400;
12     p3[2] = 500;
13
14   return 0;
15 }
```

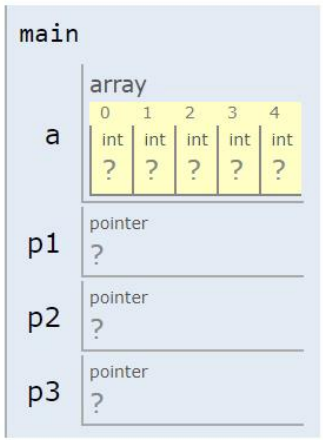
Edit this code

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 10

Stack Heap



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:

```
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) {
```

Function Pointers

- Can use pointers that store addresses of functions!
- Generic format:
  - Looks like a function prototype with extra * in front of name

```
returnType (* name)(type1, …, typeN)
```

- Using the function:
  - Calls the pointed-to function with the given arguments and return the return value

```
(*name)(arg1, …, argN)
```

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};
  map(arr, LEN, square);
  ...
```

funcptr parameter

funcptr dereference

funcptr assignment

map.c https://onlinegdb.com/S1Miv-GMU

Extra Exercise

- 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 0;
}
```