# More Pointers, The Heap

## CSE 333 Winter 2020

**Instructor:** Justin Hsia

**Teaching Assistants:**

Andrew Hu
Cheng Ni
Guramrit Singh
Rehaan Bhimar
Zachary Keyes

Austin Chan
Cosmo Wang
Mengqi Chen
Renshu Gu

Brennan Stein
Diya Joy
Pat Kosakanchit
Travis McGaha

# Administrivia

- Exercise 2 out today and due Monday morning

- 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)
  - Grades:
    - Autograder [0 to 3], Style [-2 to +0.1] – Overall [0, 1, 2, 3, 3.1]

# Administrivia

- 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 later today, due in 2 weeks (Thu 1/23)
  - 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

- **Pointer Arithmetic**
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
- Heap-allocated Memory

# 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/justinh

A. {2, 3, 4}

B. {3, 4, 5}

C. {2, 6, 4}

D. {2, 4, 5}

E. We’re lost…

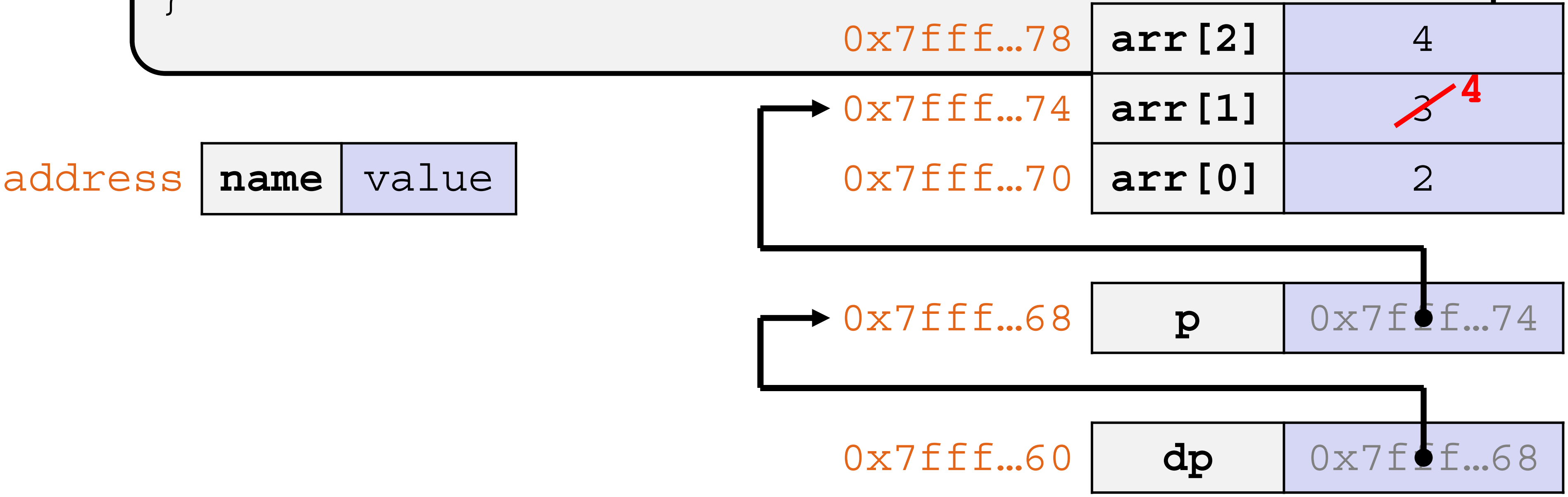
| | | |
|---|---|---|
| 0x7fff…78 | **arr[2]** | 4 |
| 0x7fff…74 | **arr[1]** | 3 |
| 0x7fff…70 | **arr[0]** | 2 |
| 0x7fff…68 | **p** | 0x7fff…74 |
| 0x7fff…60 | **dp** | 0x7fff…68 |

# 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;
}
```



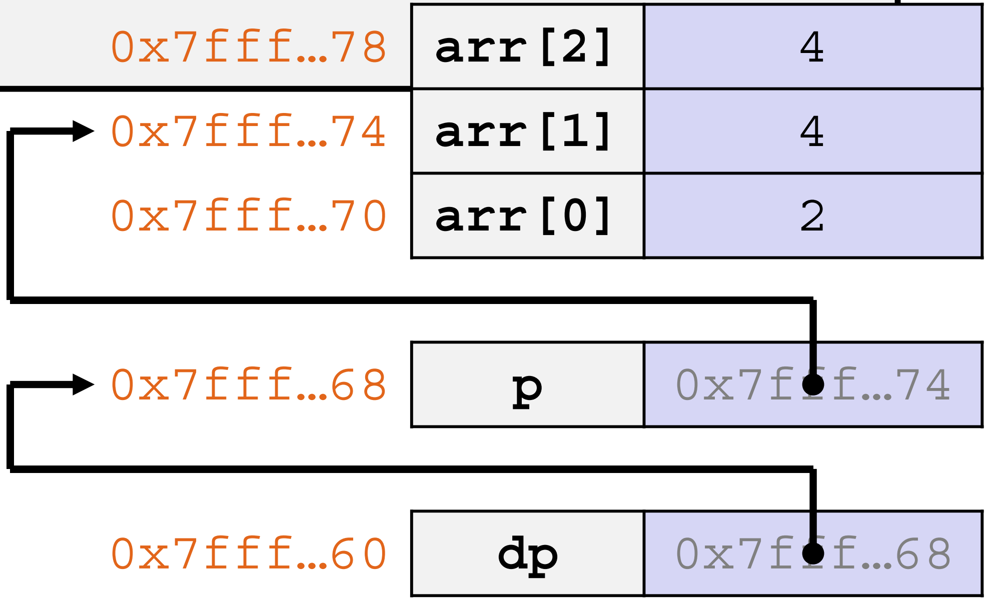
| address | name | value |
|---|---|---|
| 0x7fff…78 | **arr[2]** | 4 |
| 0x7fff…74 | **arr[1]** | ~~3~~ 4 |
| 0x7fff…70 | **arr[0]** | 2 |
| 0x7fff…68 | **p** | 0x7fff…74 |
| 0x7fff…60 | **dp** | 0x7fff…68 |

# Practice Solution

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…74 |
| 0x7fff…60 | dp | 0x7fff…68 |

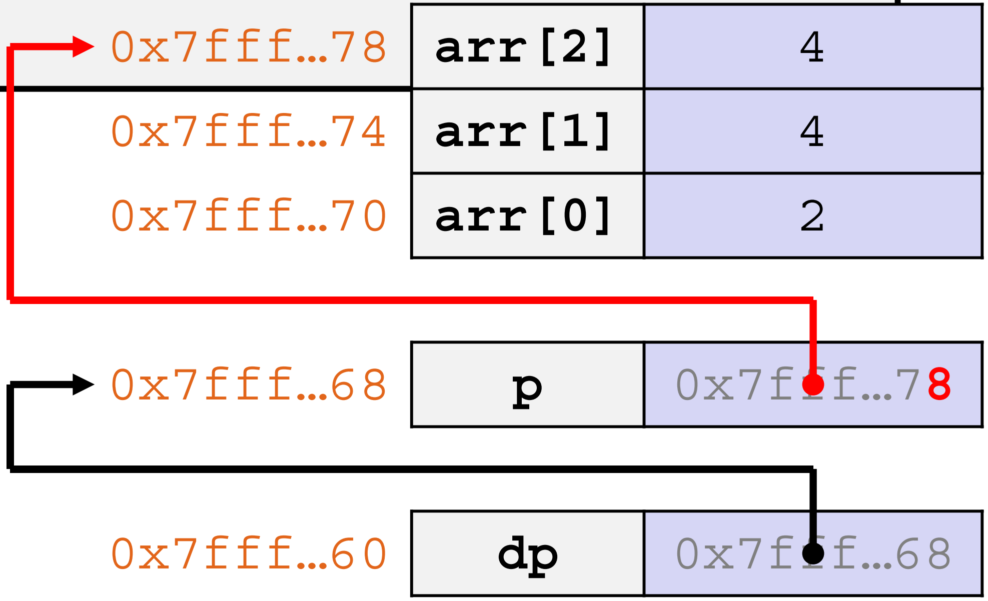
# Practice Solution

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 |
|---|---|---|



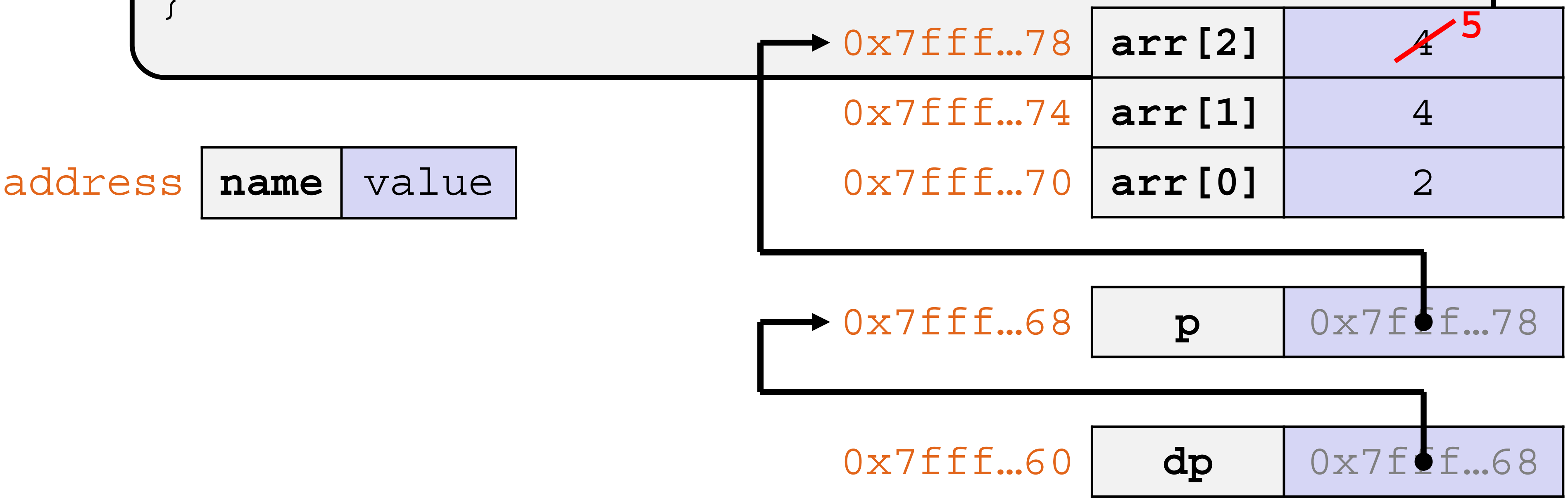
| | | |
|---|---|---|
| 0x7fff…78 | **arr[2]** | 4 |
| 0x7fff…74 | **arr[1]** | 4 |
| 0x7fff…70 | **arr[0]** | 2 |
| 0x7fff…68 | **p** | 0x7fff…7**8** |
| 0x7fff…60 | **dp** | 0x7fff…68 |

# 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;
}
```

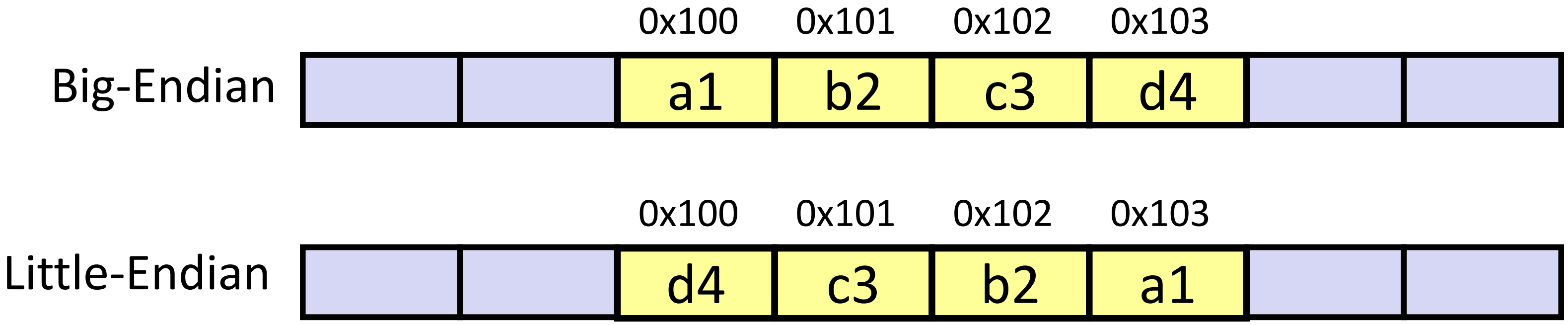


| address | name | value |
|---|---|---|
| 0x7fff…78 | arr[2] | ~~4~~ 5 |
| 0x7fff…74 | arr[1] | 4 |
| 0x7fff…70 | arr[0] | 2 |
| 0x7fff…68 | p | 0x7fff…78 |
| 0x7fff…60 | dp | 0x7fff…68 |

# 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 | | |

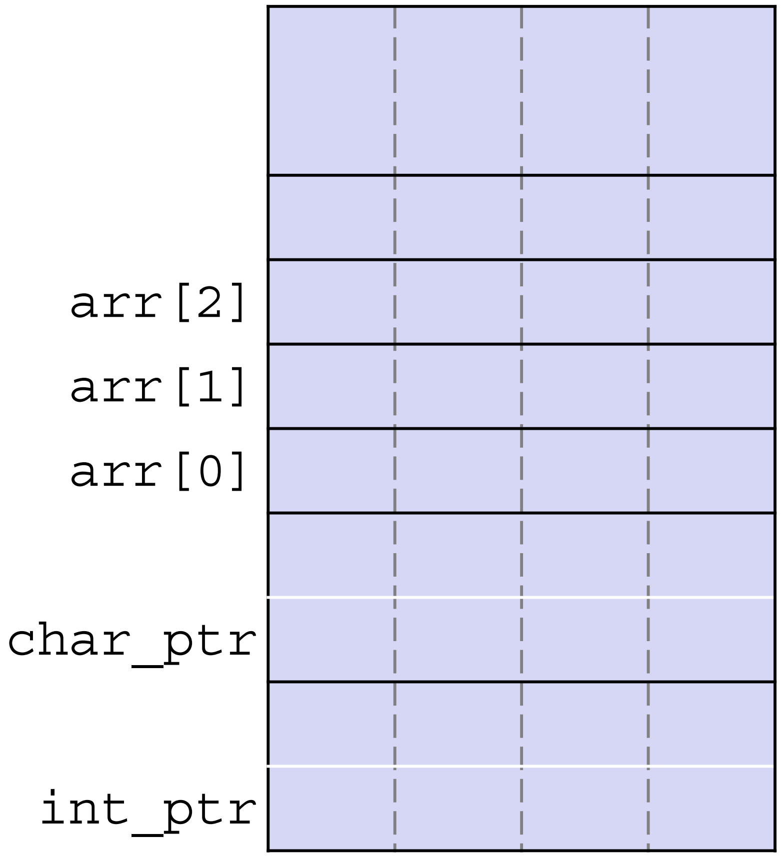
# Pointer Arithmetic Example

Note: Arrow points to *next* instruction.

```
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)



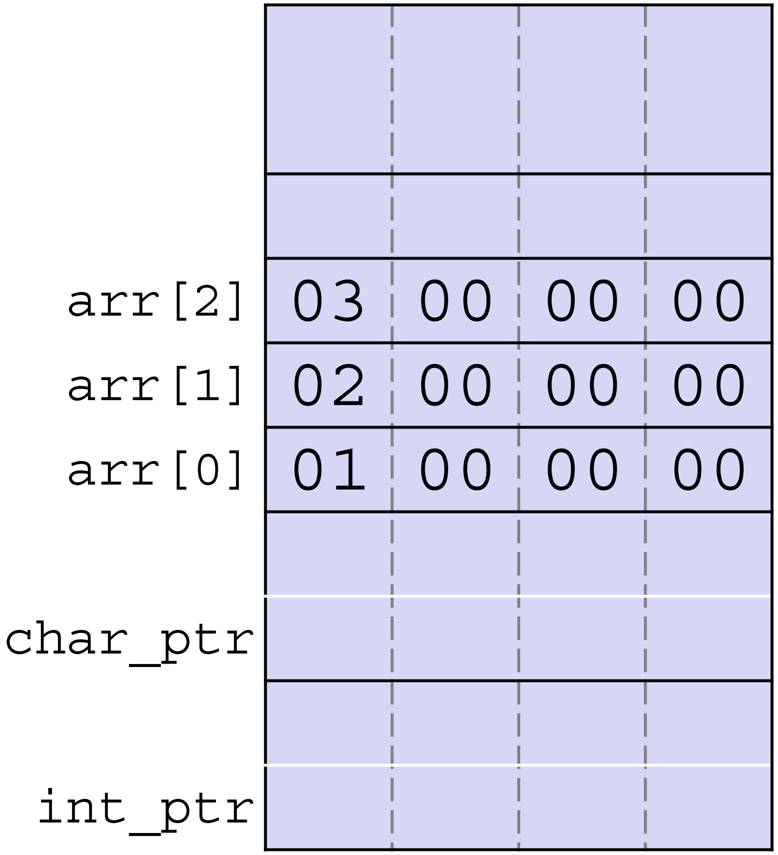
# Pointer Arithmetic Example

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 | | | | |

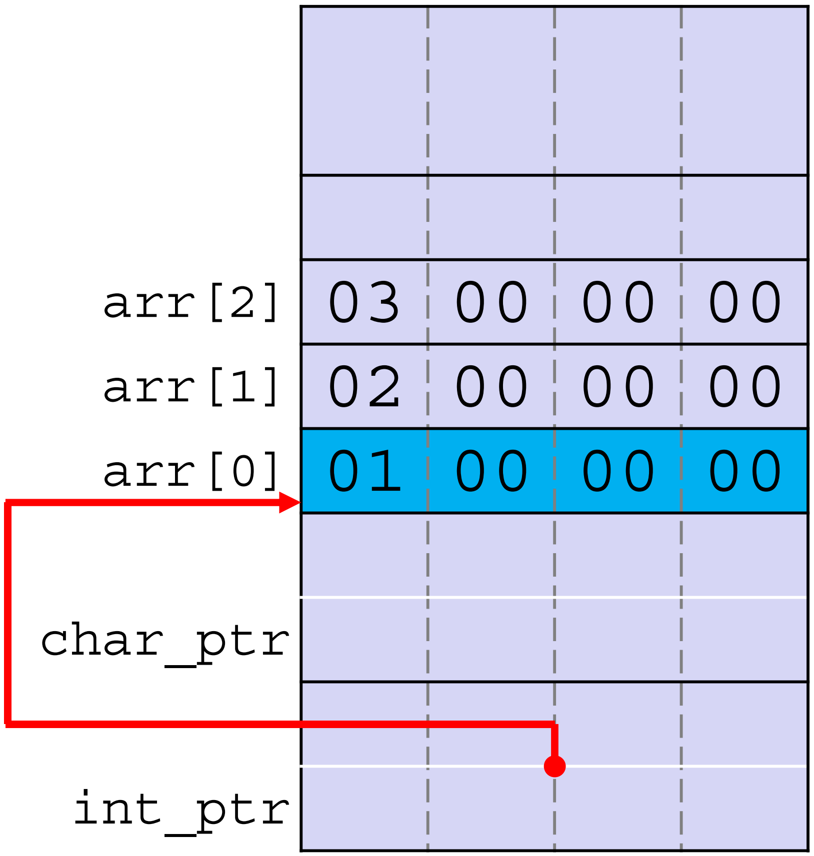
# Pointer Arithmetic Example

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)

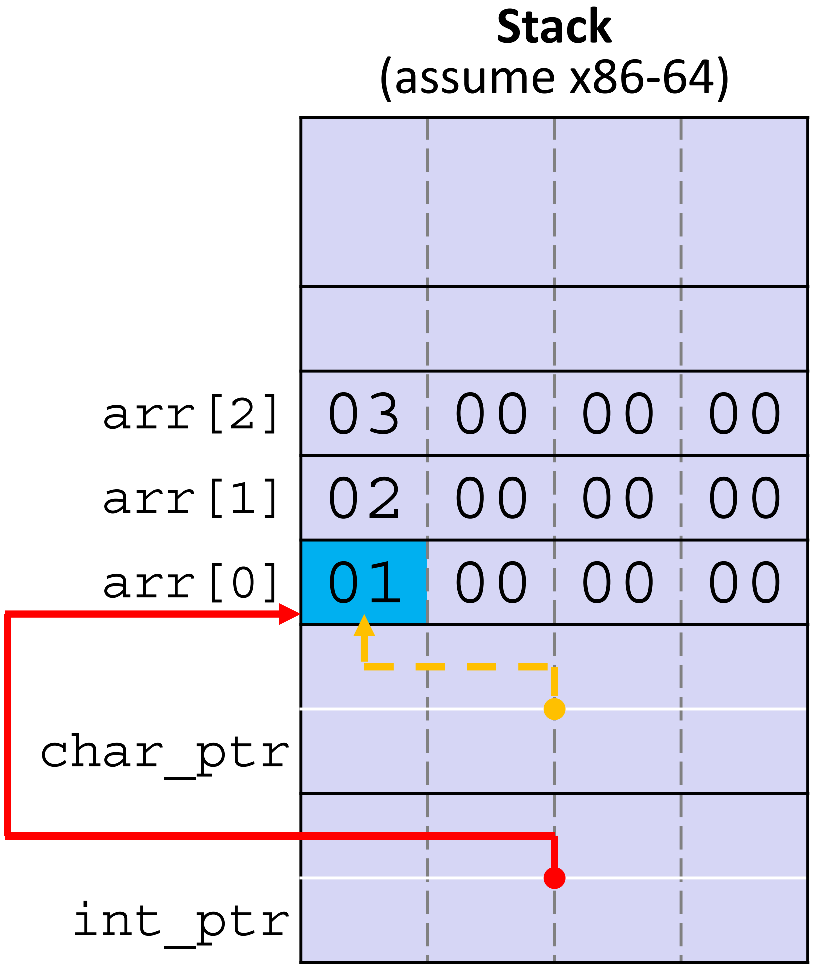


# Pointer Arithmetic Example

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



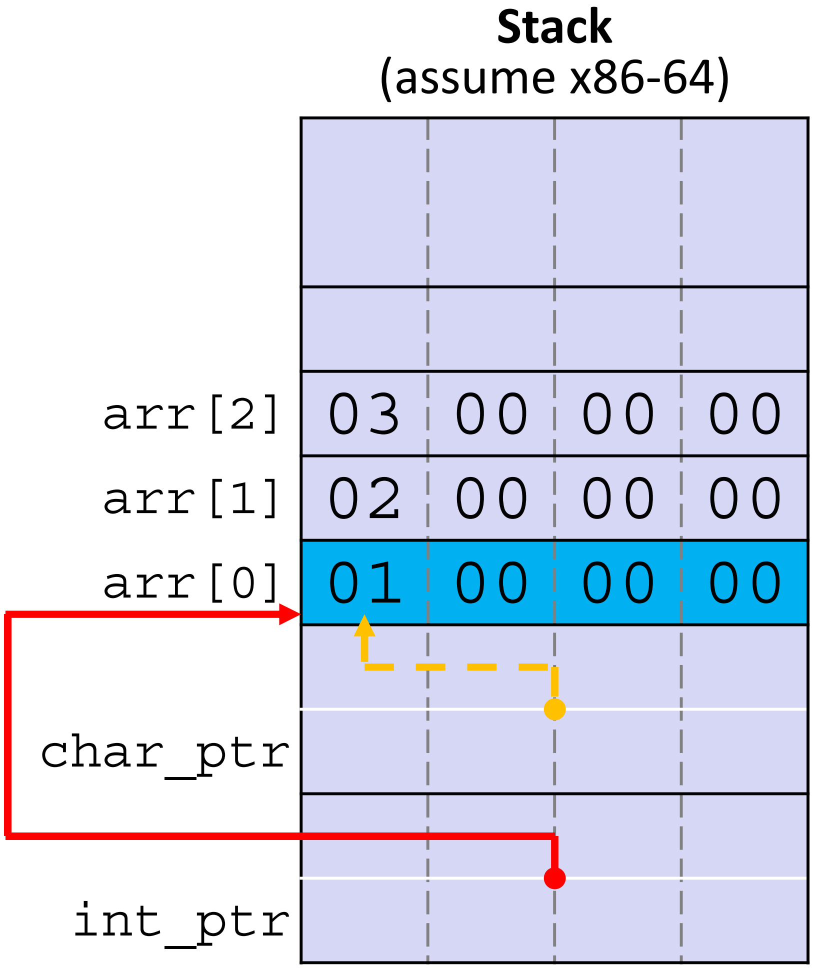
**Stack**
(assume x86-64)

# Pointer Arithmetic Example

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

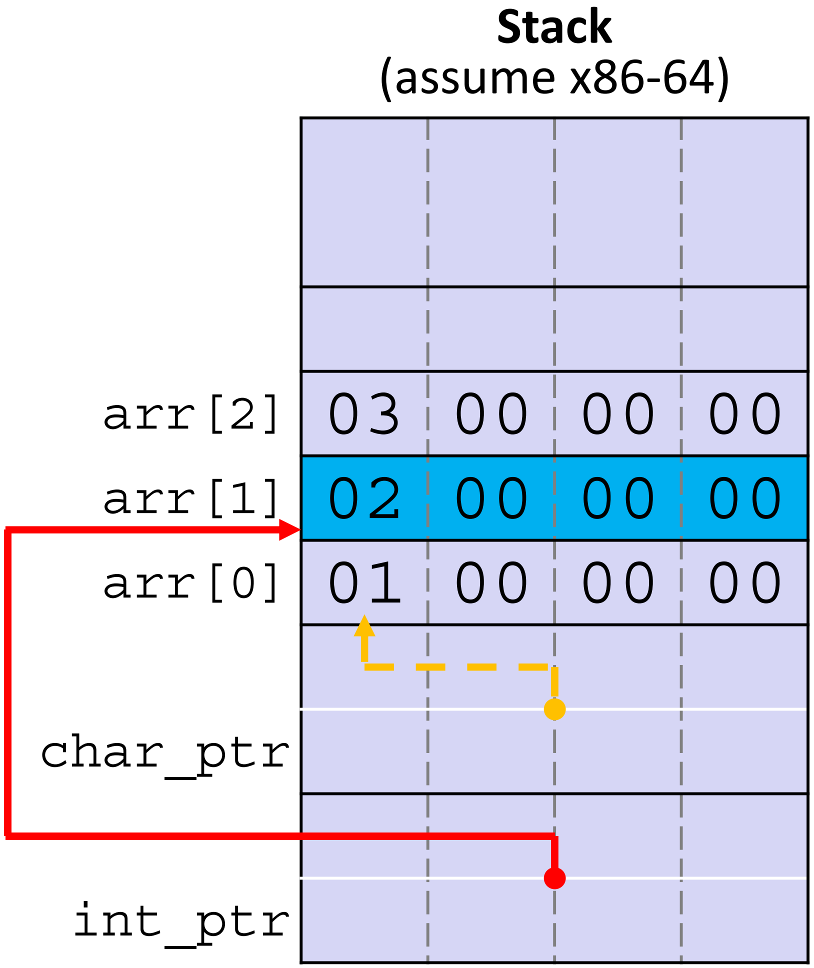
```
 int_ptr:   0x0x7ffffffde010
*int_ptr:   1
```

# 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



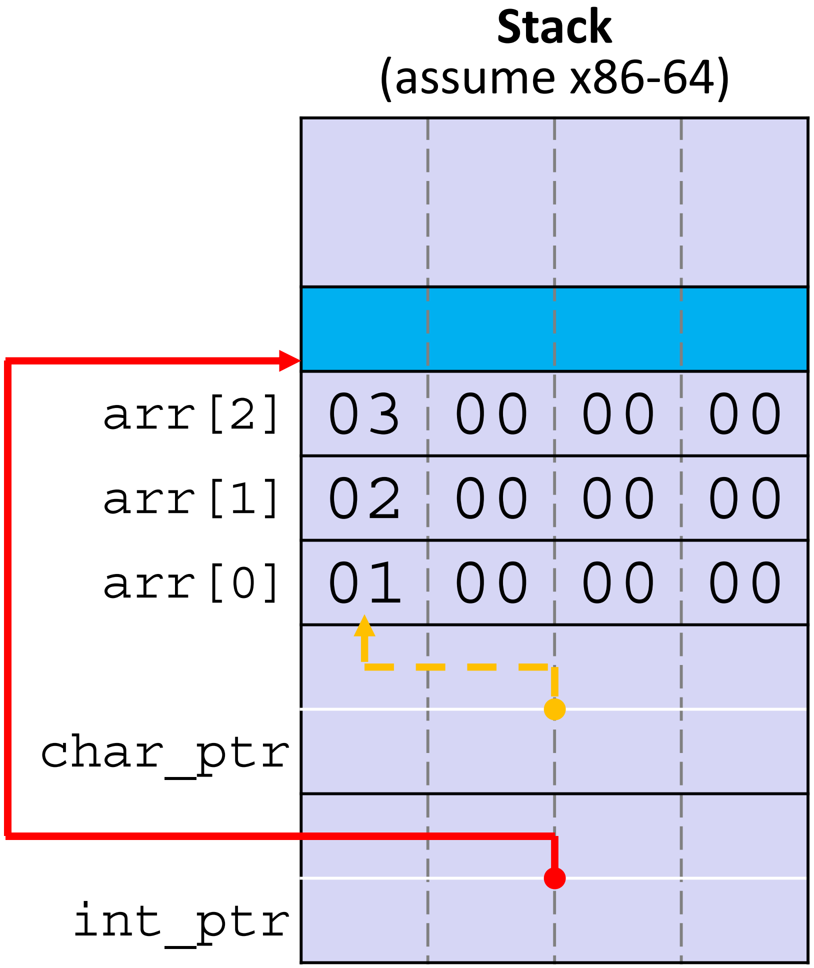
```
 int_ptr:   0x0x7fffffde014
*int_ptr:   2
```

# 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 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 | | | | |

```
 int_ptr:   0x0x7ffffffde01C
*int_ptr:   ???
```

# Pointer Arithmetic Example

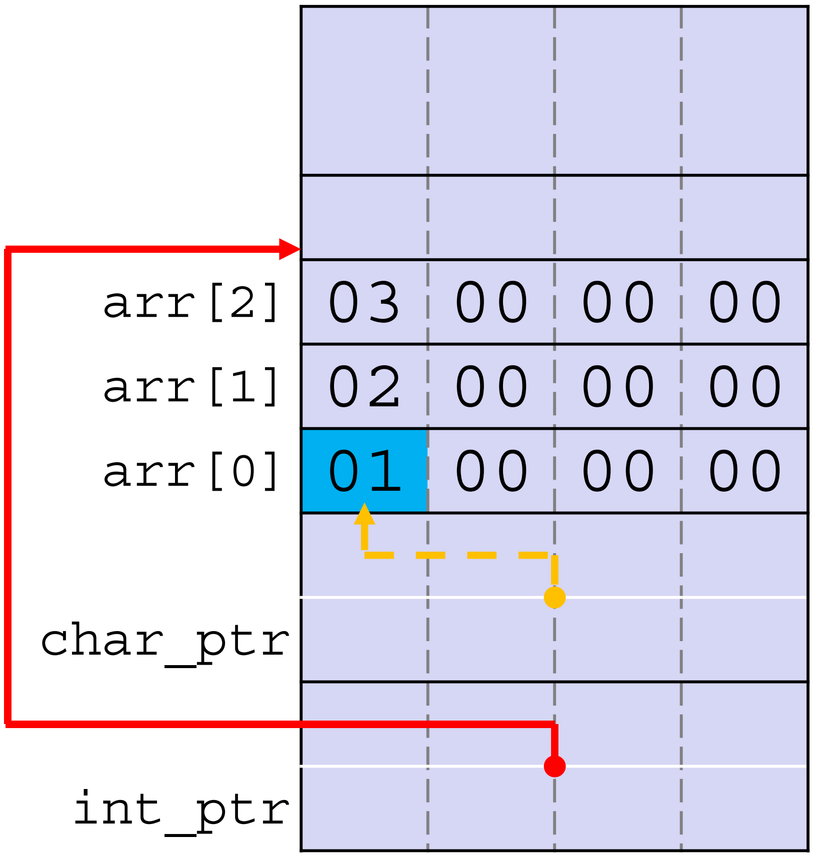
Note: Arrow points to *next* instruction.

```
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

```
 char_ptr:   0x0x7ffffffde010
*char_ptr:   1
```

**Stack**
(assume x86-64)



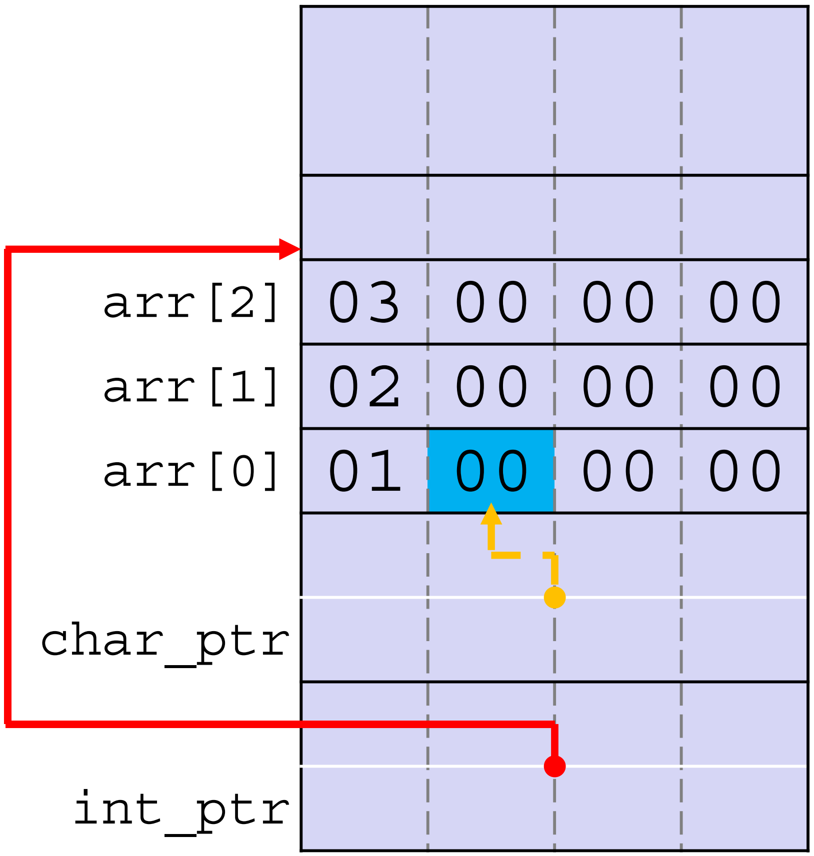
# Pointer Arithmetic Example

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 EXIT_SUCCESS;
}
```

pointerarithmetic.c

**Stack**
(assume x86-64)



```
 char_ptr:   0x0x7ffffffde011
*char_ptr:   0
```

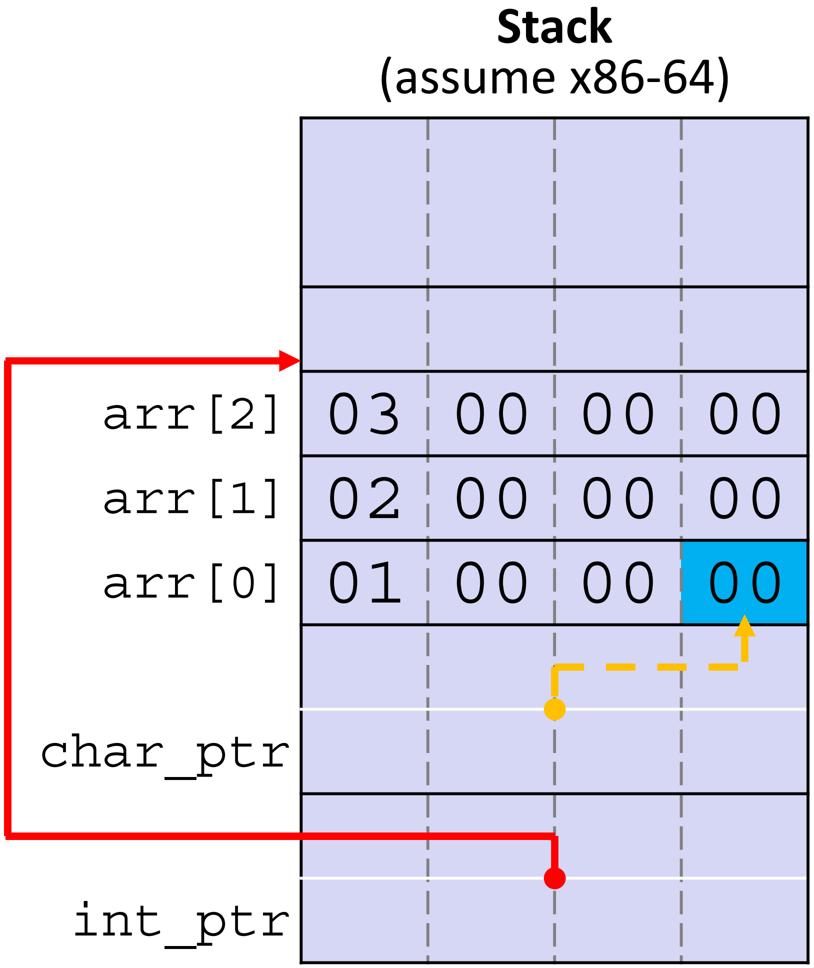
# Pointer Arithmetic Example

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

```
 char_ptr:  0x0x7fffffde013
*char_ptr:  0
```



# Lecture Outline

- Pointer Arithmetic
- **Pointers as Parameters**
- Pointers and Arrays
- Function Pointers
- Heap-allocated Memory

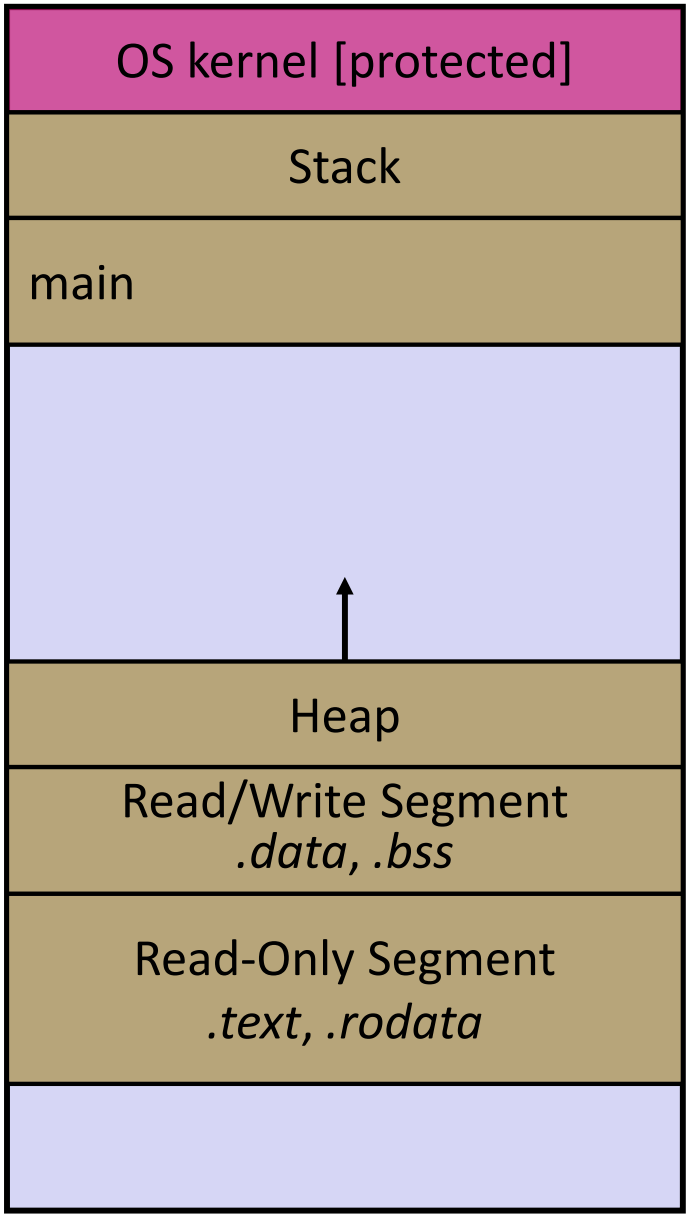
# C is Call-By-Value

- 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.



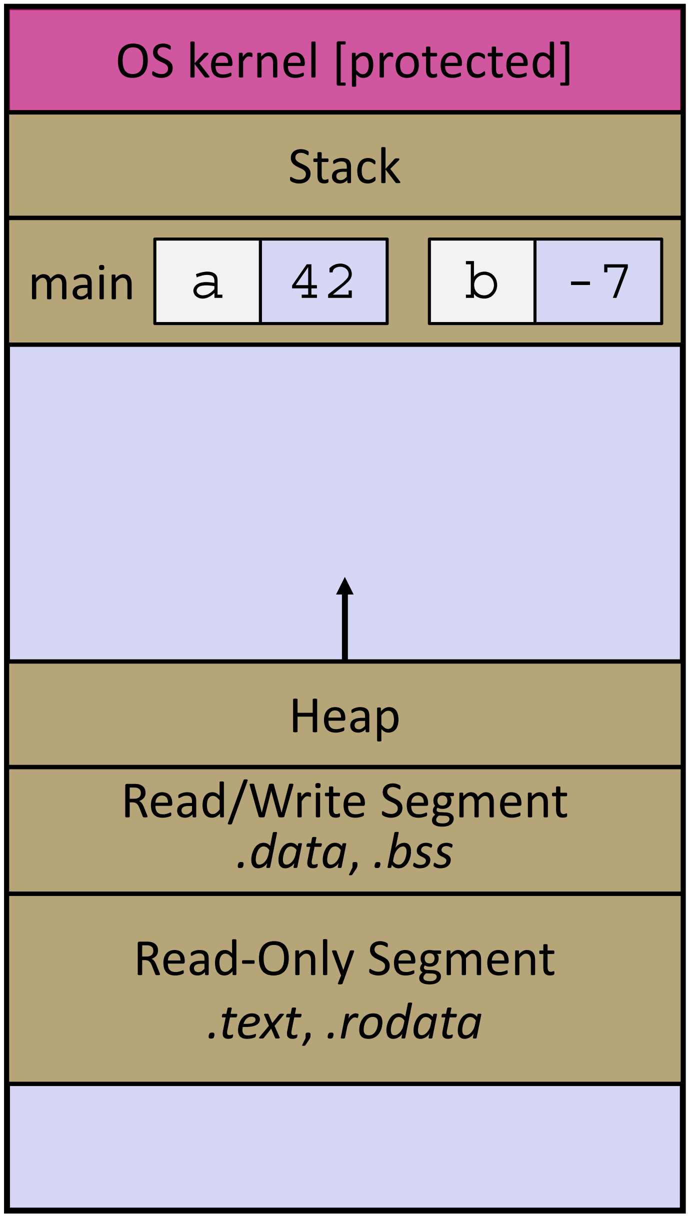
brokenswap.c

```
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);
  ...
```



OS kernel [protected]

Stack

main a 42 b -7

Heap

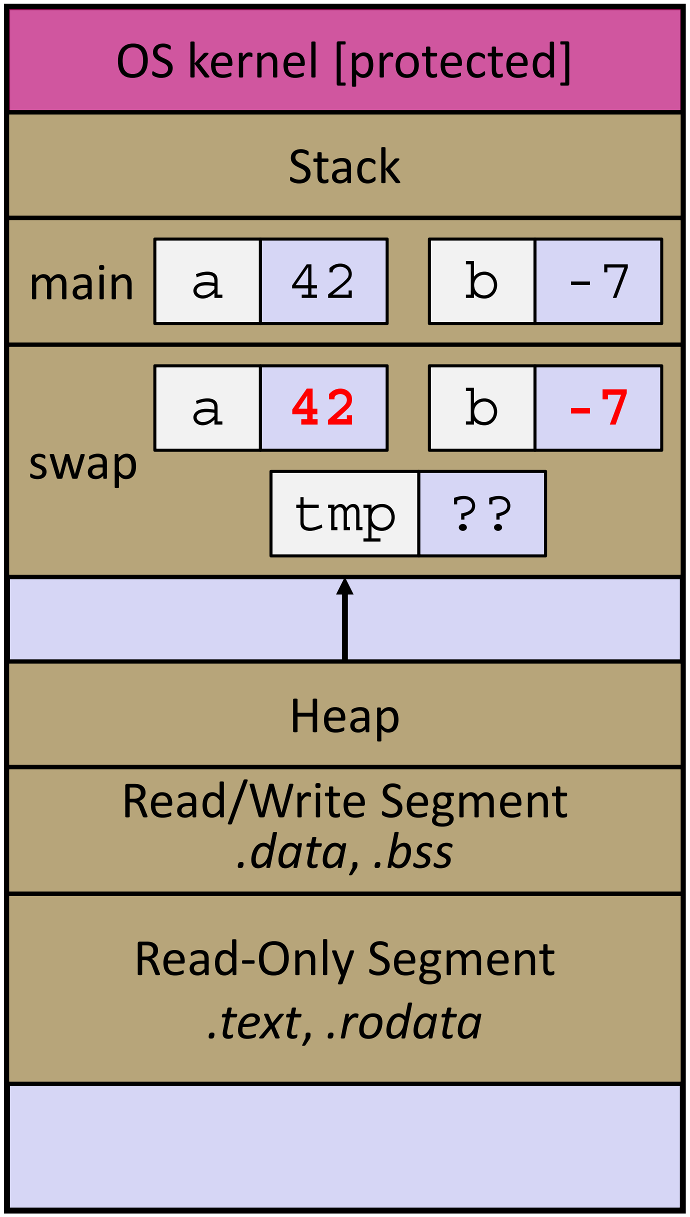
Read/Write Segment
*.data, .bss*

Read-Only Segment
*.text, .rodata*

# Broken Swap

brokenswap.c

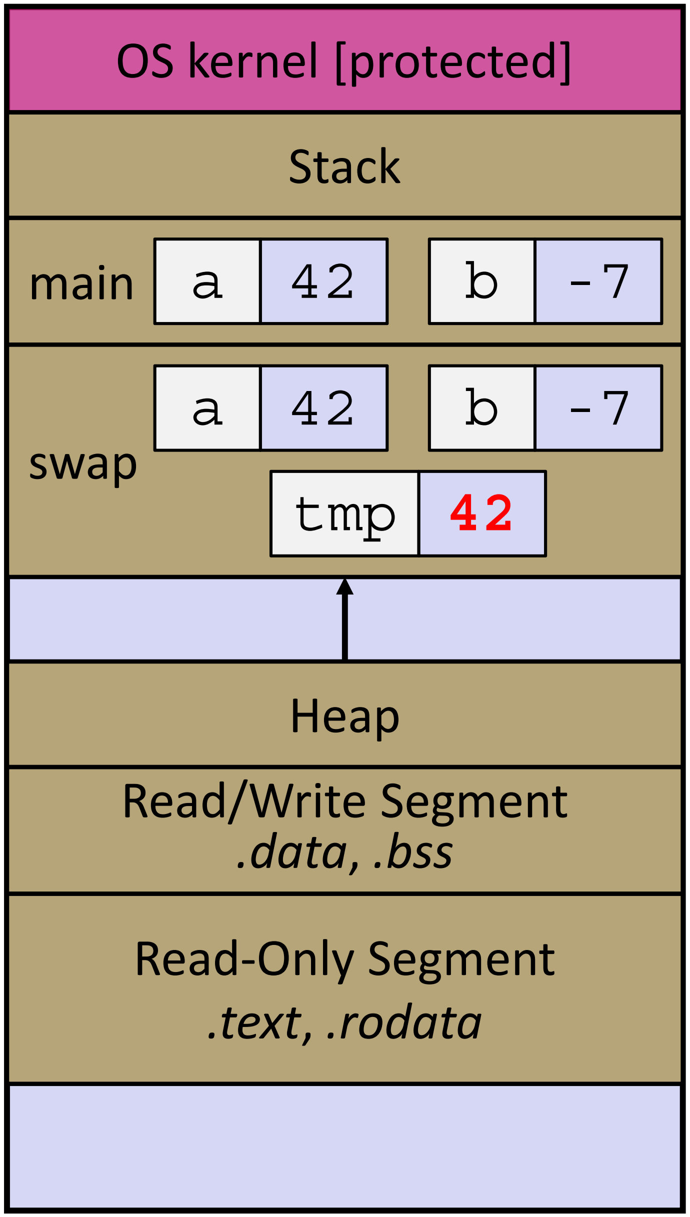
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



# Broken Swap

brokenswap.c

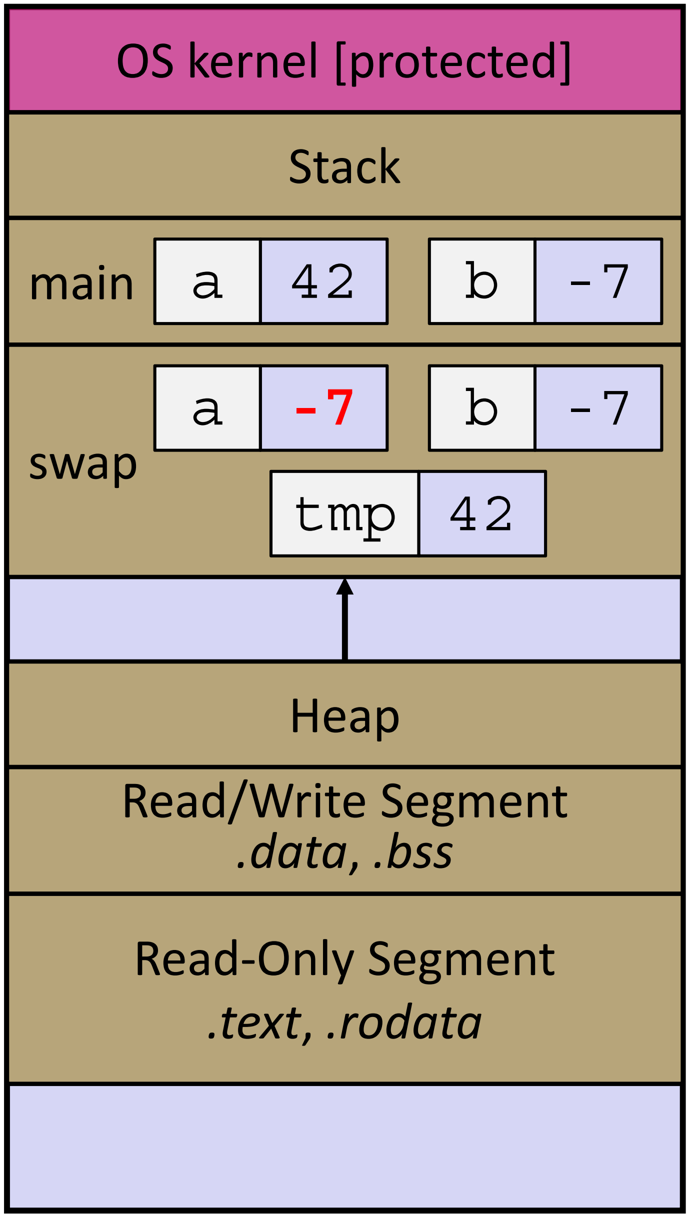
```c
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



# Broken Swap

brokenswap.c

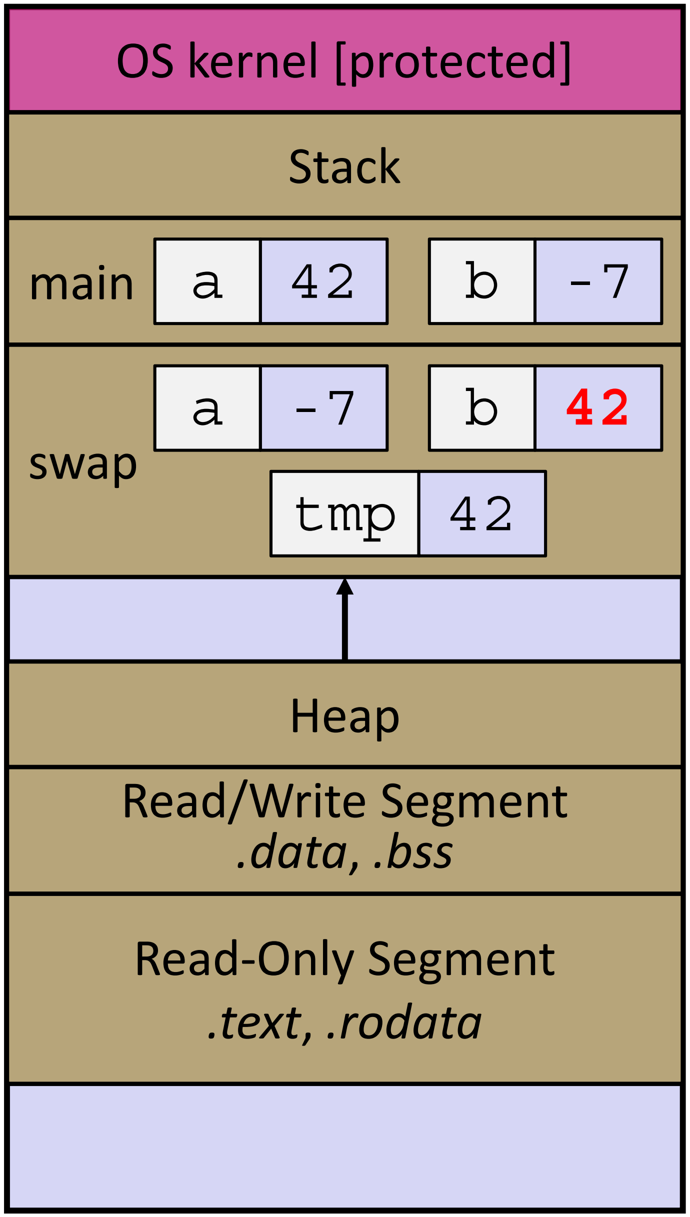
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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



# Broken Swap

brokenswap.c

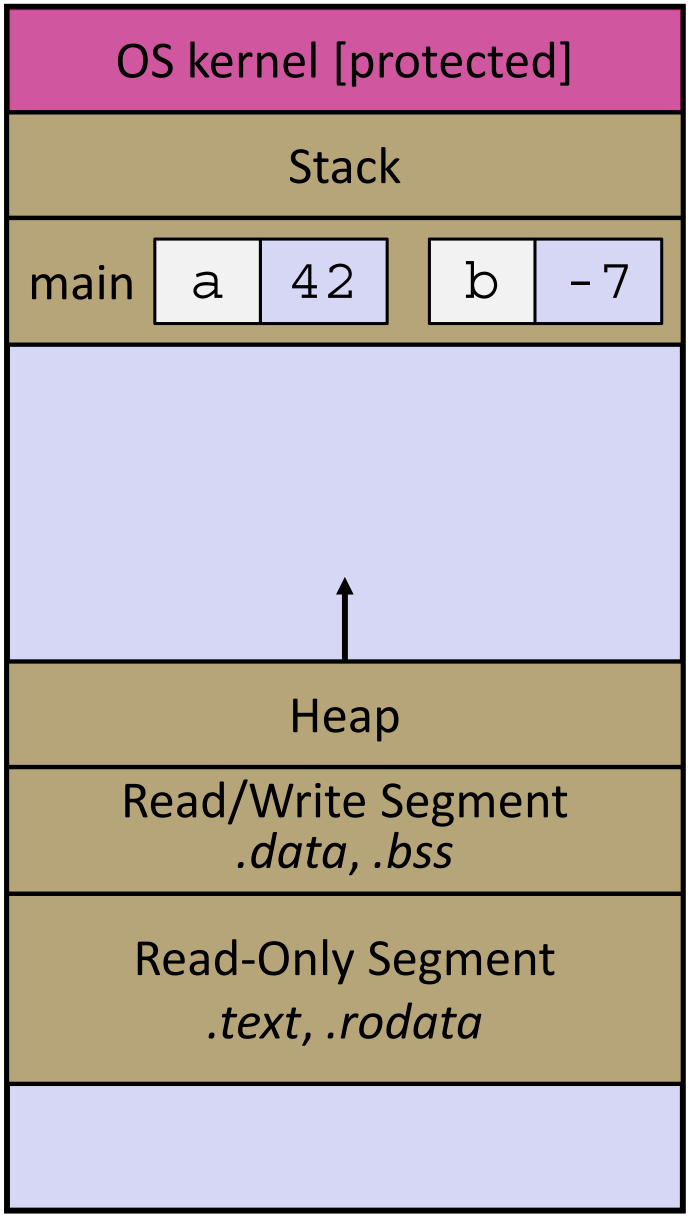
```
void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

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

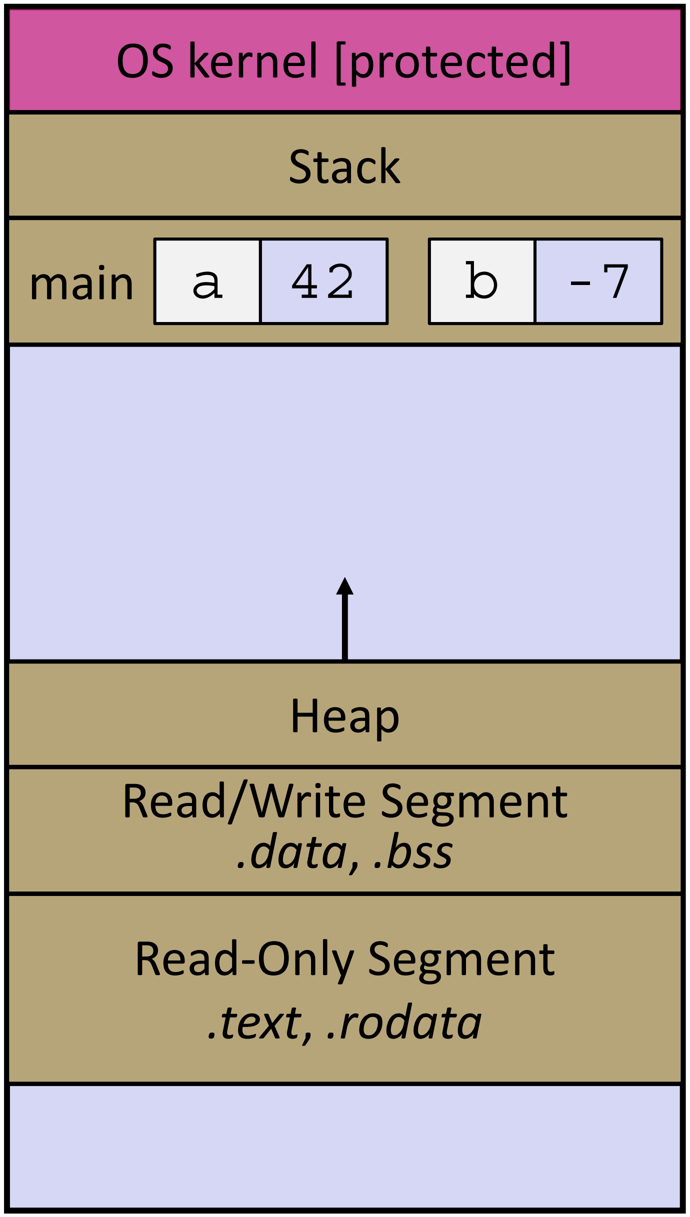
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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

# Fixed Swap

Note: Arrow points to *next* instruction.

swap.c

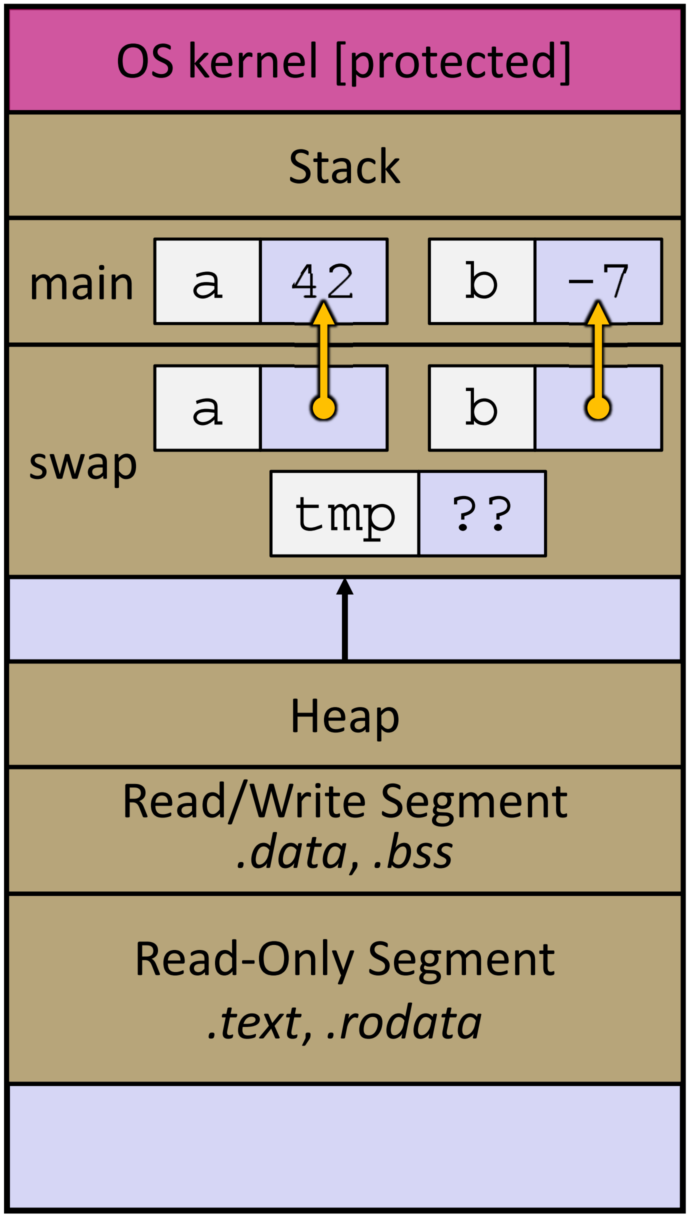
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



# Fixed Swap

swap.c

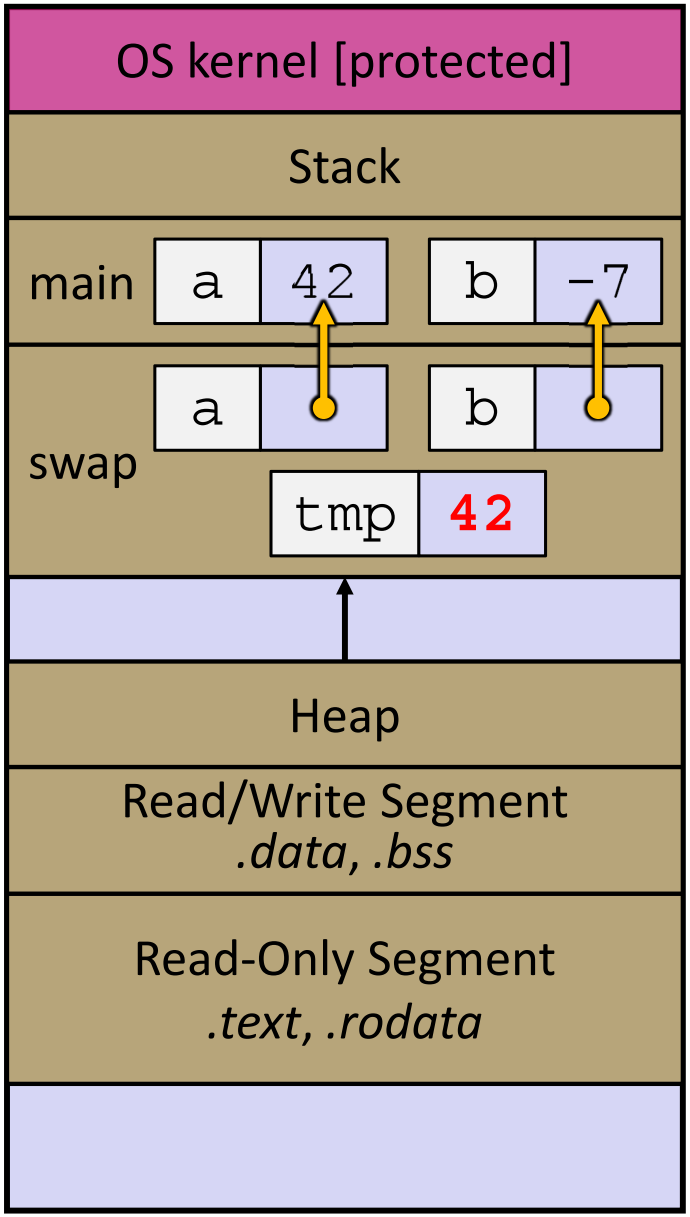
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



# Fixed Swap

swap.c

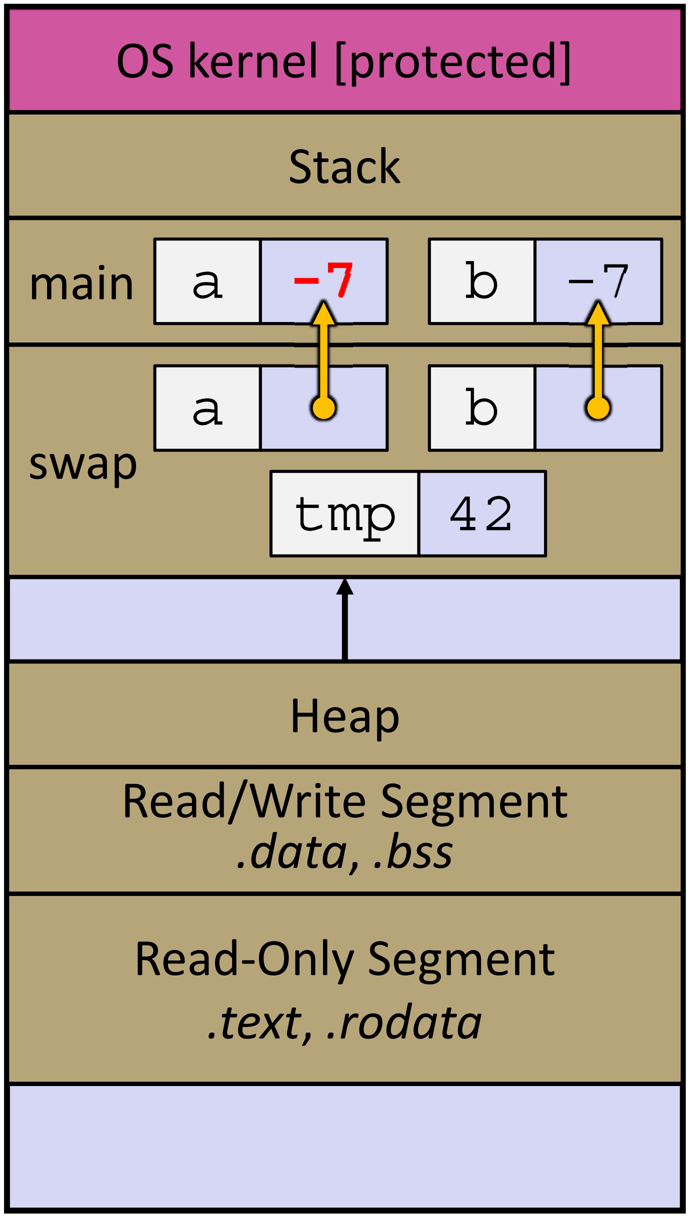
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



# Fixed Swap

swap.c

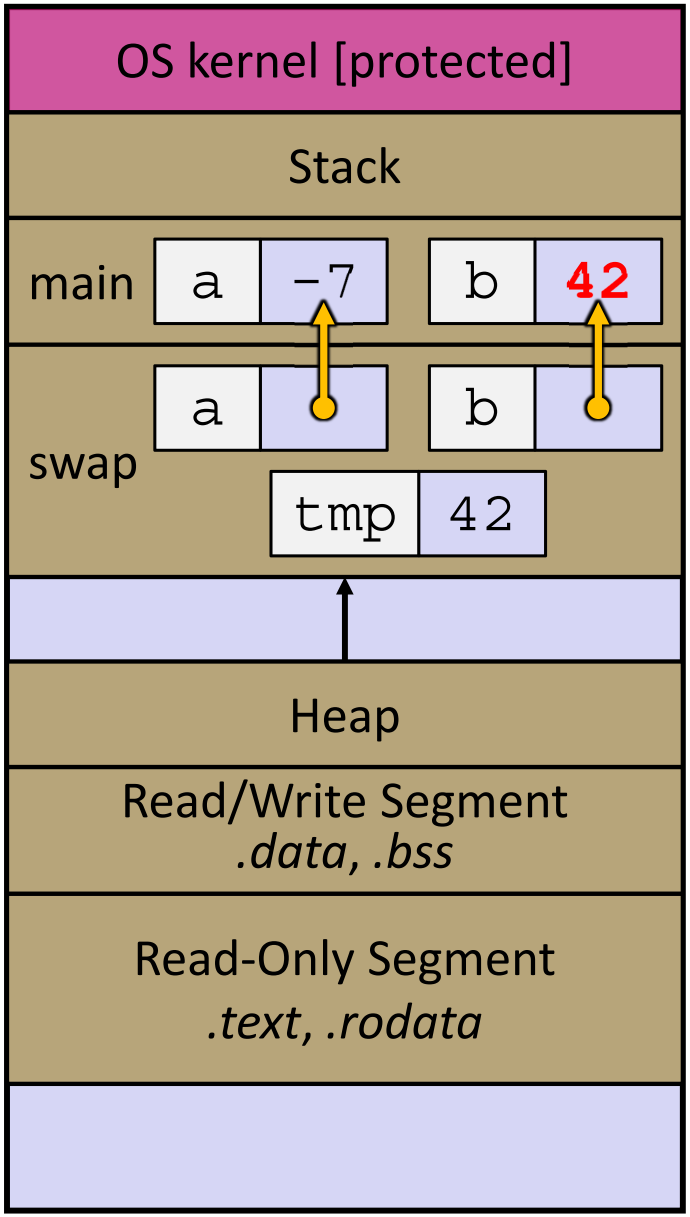
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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



# Fixed Swap

swap.c

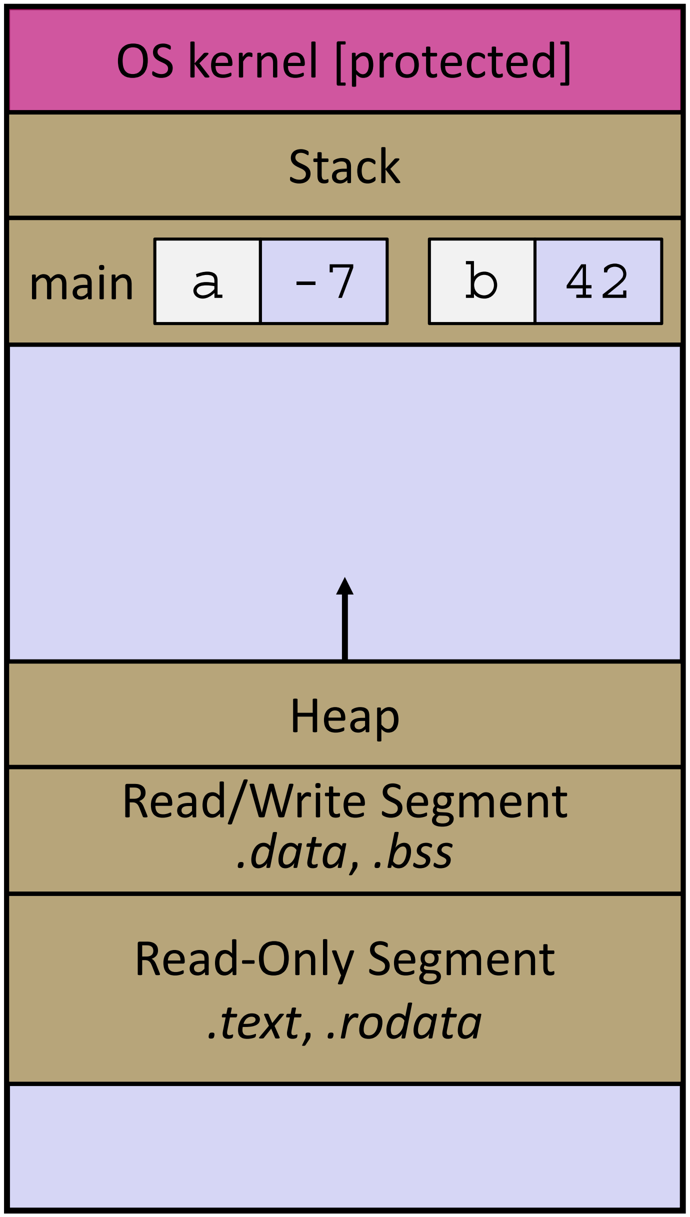
```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

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);
  ...
```



OS kernel [protected]

Stack

main a -7 b 42

Heap

Read/Write Segment
*.data, .bss*

Read-Only Segment
*.text, .rodata*

# Lecture Outline

- Pointer Arithmetic
- Pointers as Parameters
- **Pointers and Arrays**
- Function Pointers
- Heap-allocated Memory

# 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
```

# Array Parameters

STYLE TIP

- 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 EXIT_SUCCESS;
}

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

Equivalent to:

```
void f(int* a);

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

void f(int* a) {
```

# Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- **Function Pointers**
- Heap-allocated Memory

# 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};
  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

# Lecture Outline

- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers
- **Heap-allocated Memory**

# Memory Allocation So Far

- So far, we have seen two kinds of memory allocation:

```
int counter = 0;      // global var

int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return EXIT_SUCCESS;
}
```

```
int foo(int a) {
  int x = a + 1;       // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);     // local var
  printf("y = %d\n",y);
  return EXIT_SUCCESS;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when process gets reaped
- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns

# Dynamic Allocation

- Situations where static and automatic allocation aren’t sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than can fit on the Stack
  - We need memory whose size is not known in advance to the caller

```
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

# Dynamic Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)

- C requires you to manually manage memory
  - Gives you more control, but causes headaches

# Aside: `NULL`

- `NULL` is a memory location that is guaranteed to be invalid
  - In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

segfault.c

```
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

# `malloc()`

- General usage: `var = (type*) malloc(`*size in bytes*`)`
- `malloc` allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns `NULL` if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

# `calloc()`

- General usage:

```
var = (type*) calloc(num, bytes per element)
```

- Like `malloc`, but also zeros out the block of memory
  - Helpful when zero-initialization wanted (but don't use it to mask bugs – fix those)
  - Slightly slower; but useful for non-performance-critical code
  - `malloc` and `calloc` are found in `stdlib.h`

```
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

# `free()`

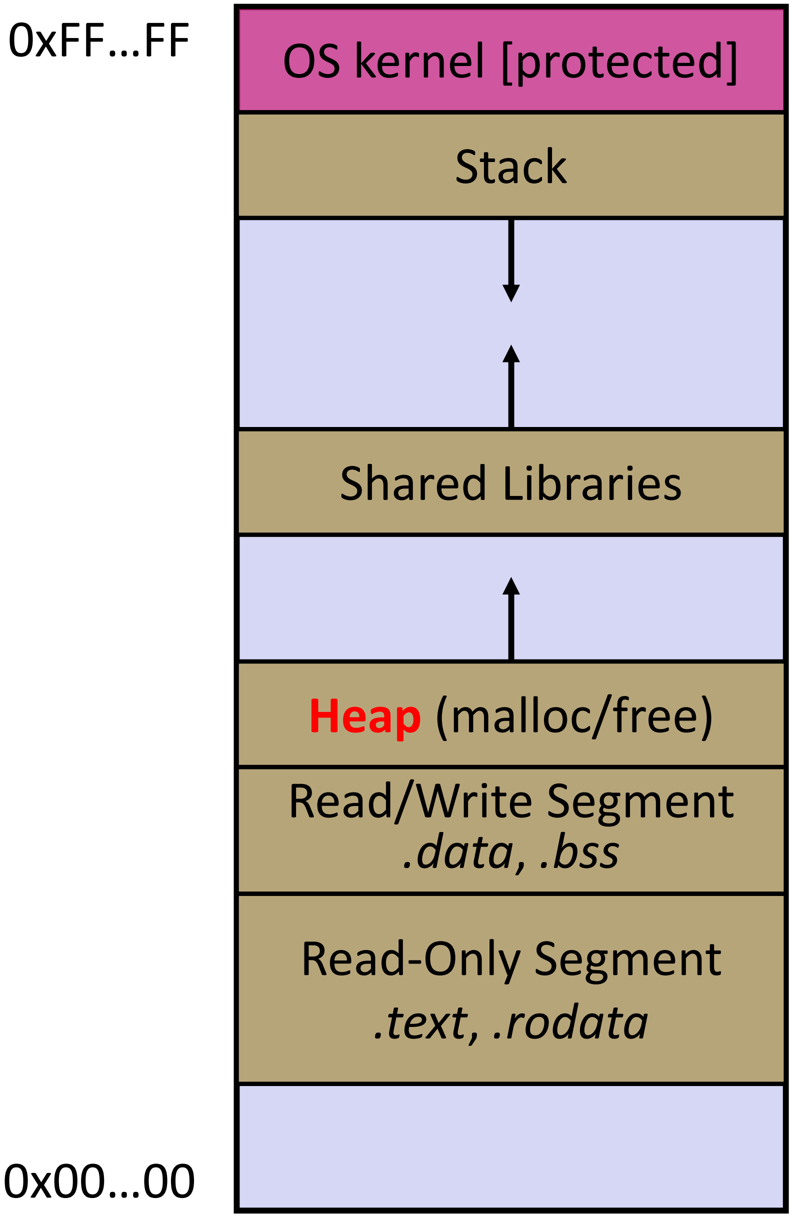
- Usage: `free(pointer);`

- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by `malloc` or `calloc`)
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call to free
    - Defensive programming: can set pointer to `NULL` after freeing it

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL
```

# The Heap

- The Heap is a large pool of available memory used to hold dynamically-allocated data
  - `malloc` allocates chunks of data in the Heap; `free` deallocates those chunks
  - `malloc` maintains bookkeeping data in the Heap to track allocated blocks
    - Lab 5 from 351!

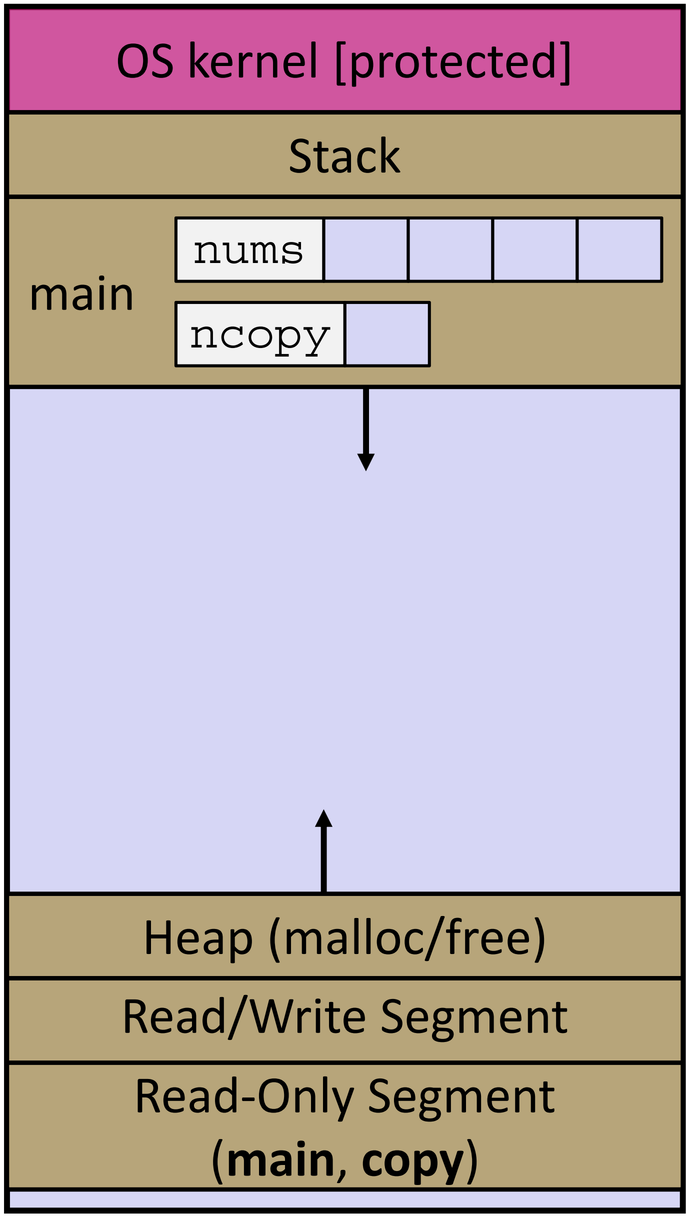


# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```c
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



OS kernel [protected]

Stack

main

nums

ncopy

Heap (malloc/free)

Read/Write Segment

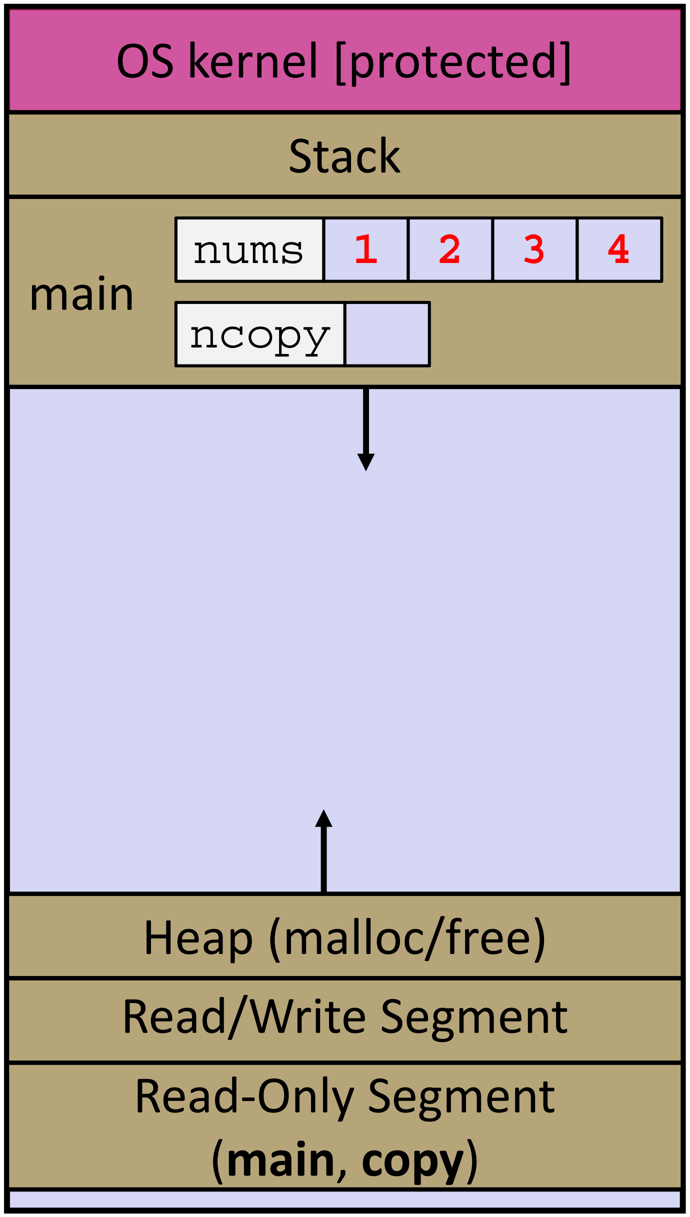
Read-Only Segment

**(main, copy)**

# Heap and Stack Example

arraycopy.c

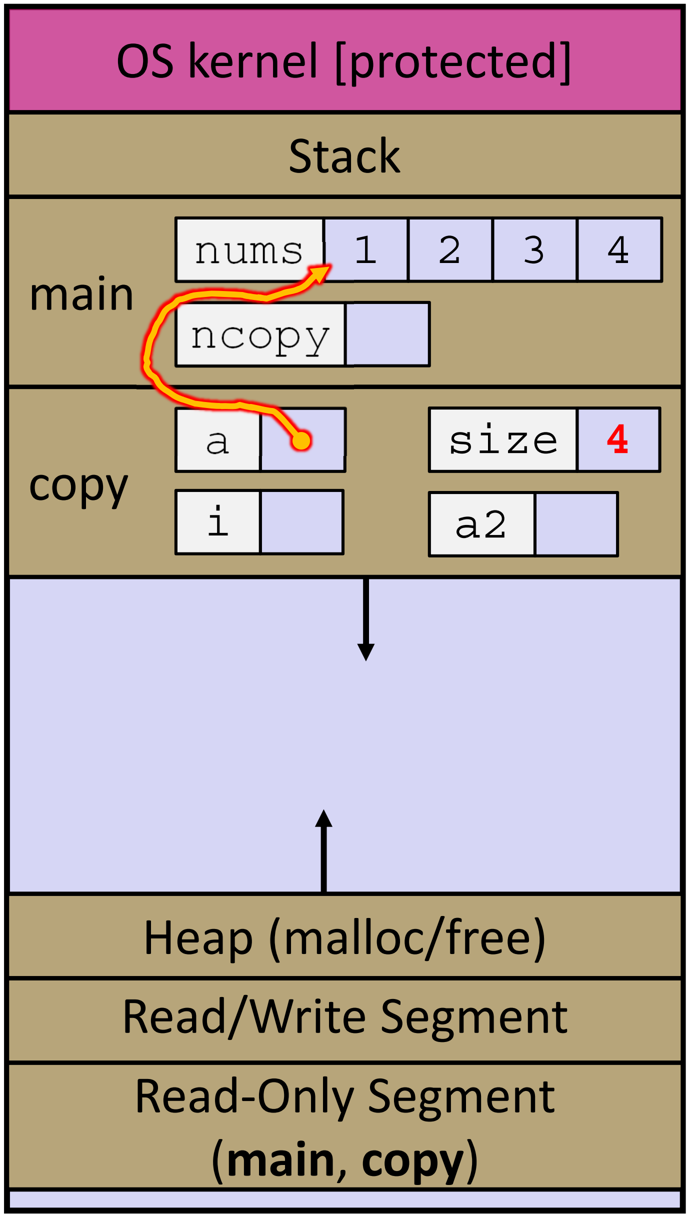
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

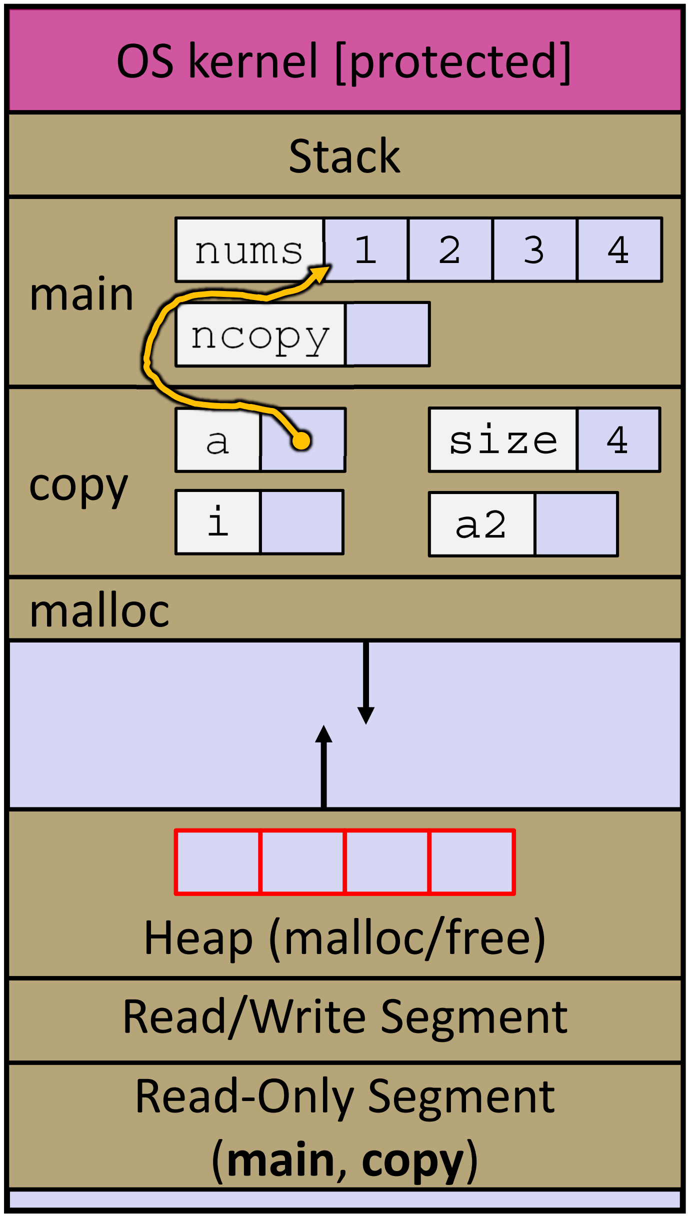
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



OS kernel [protected]

Stack

main

copy

malloc

Heap (malloc/free)

Read/Write Segment

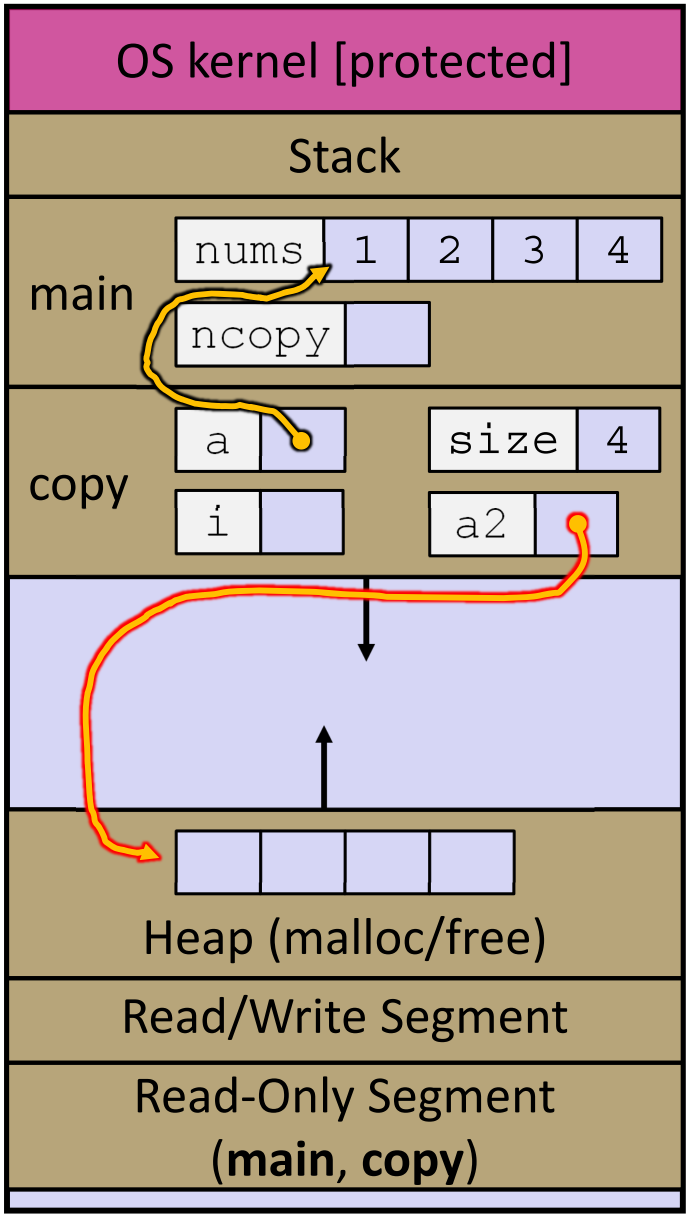
Read-Only Segment

**(main, copy)**

# Heap and Stack Example

arraycopy.c

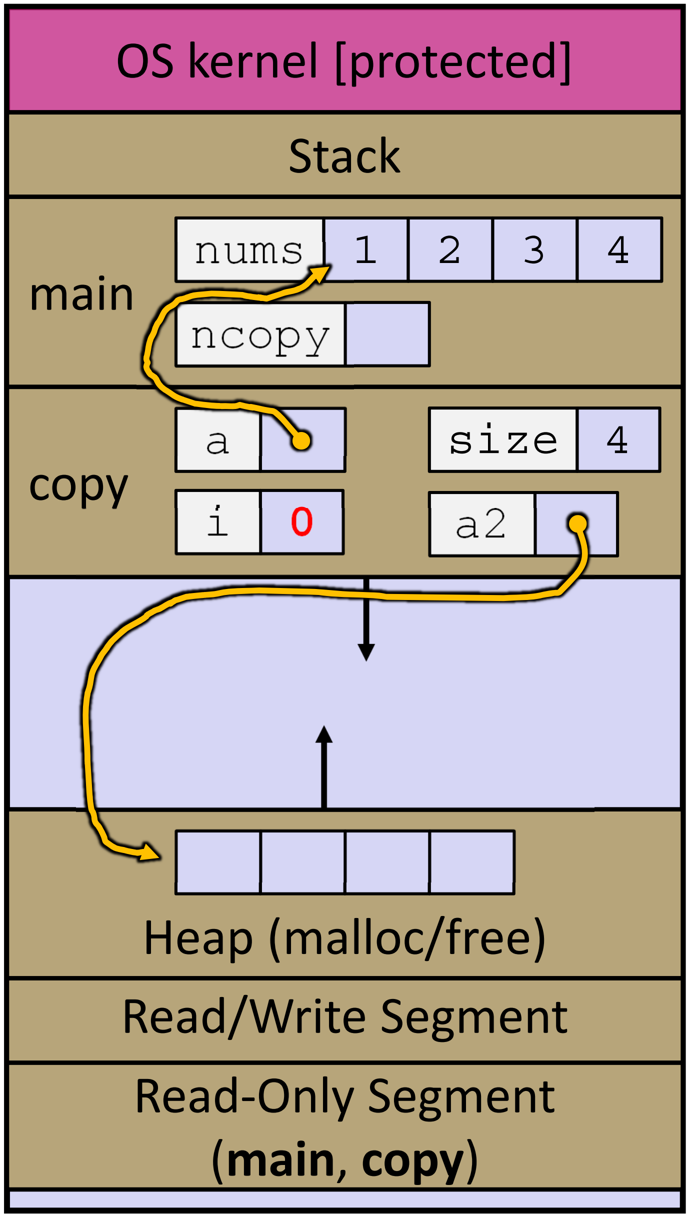
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

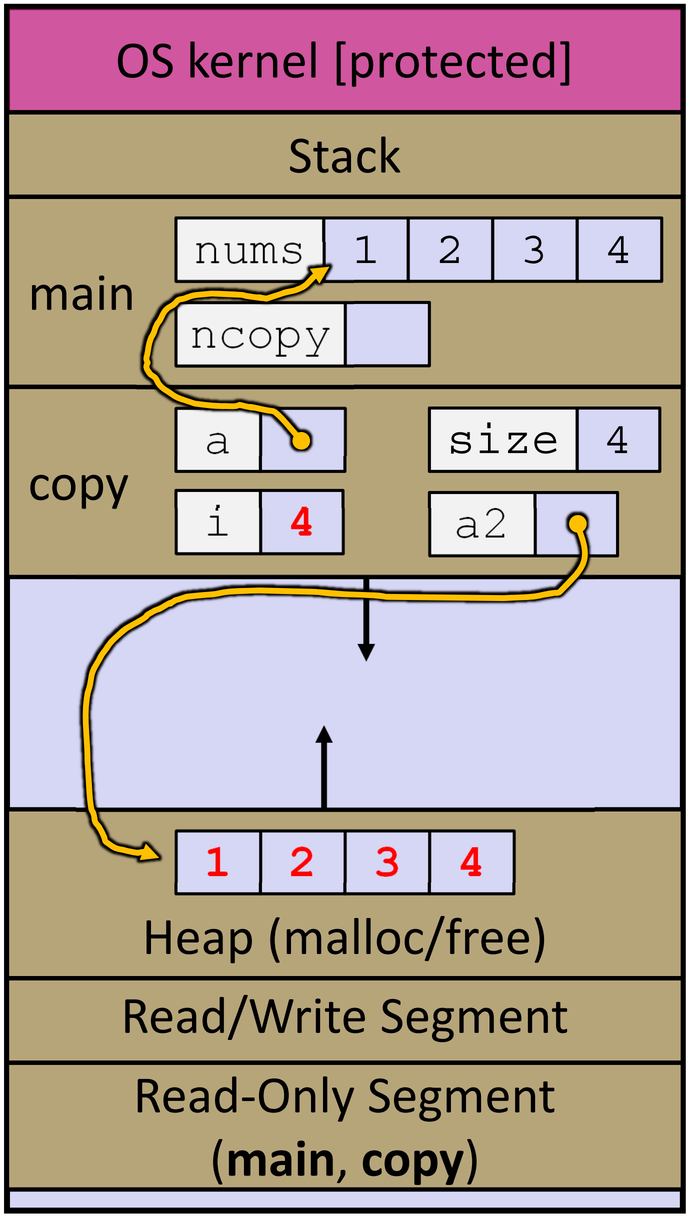
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

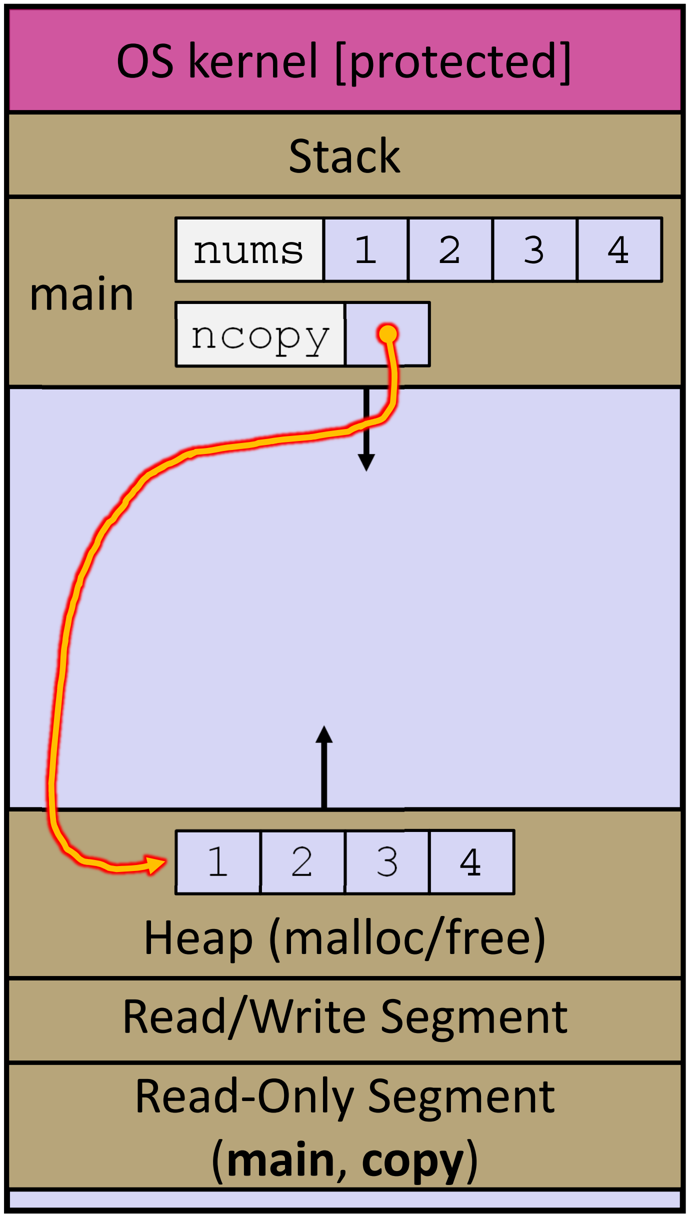
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

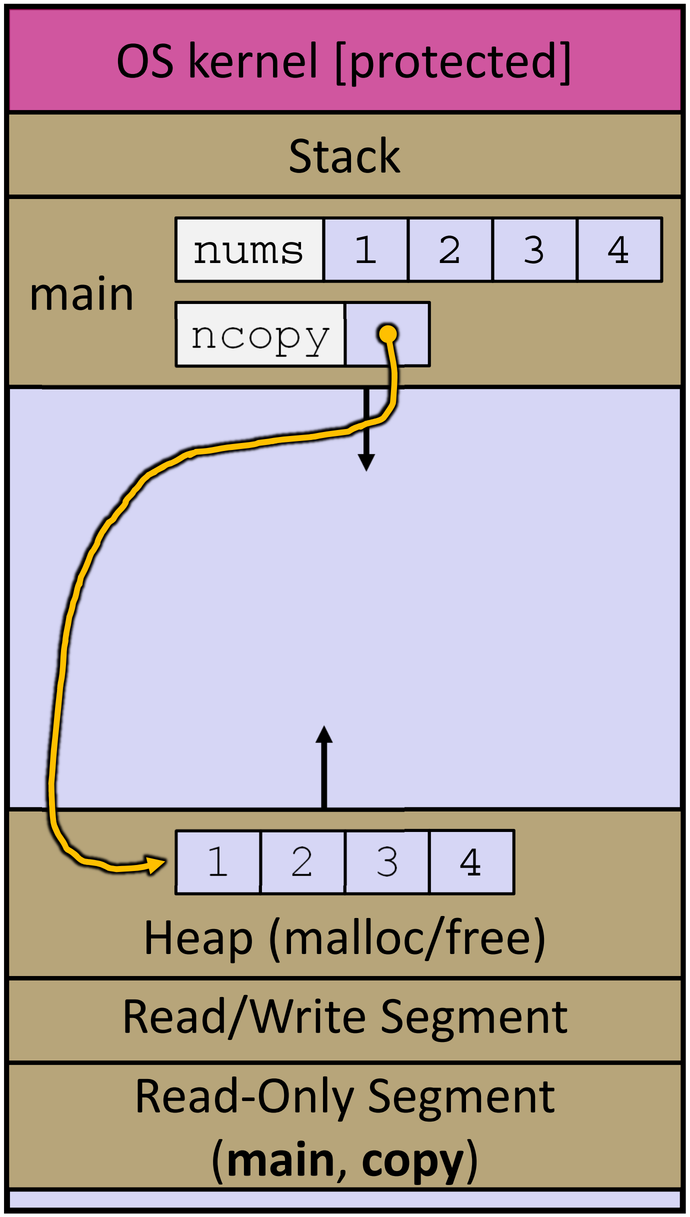
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

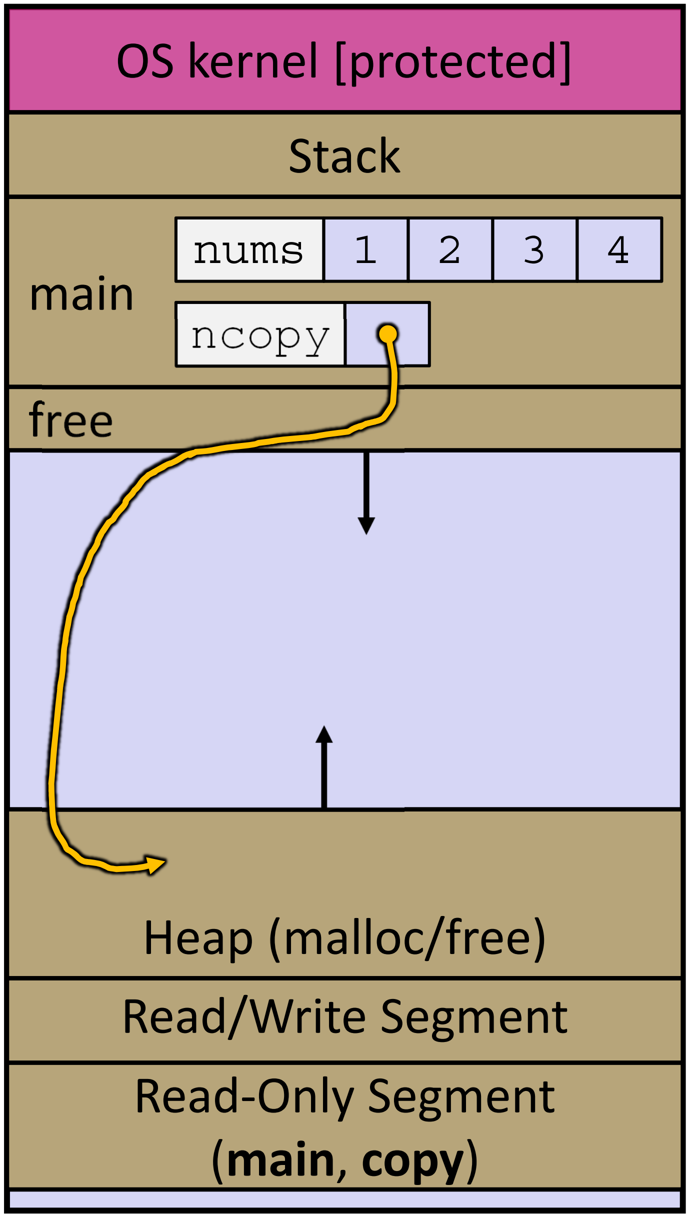
```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# Heap and Stack Example

arraycopy.c

```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



OS kernel [protected]

Stack

main

| nums | 1 | 2 | 3 | 4 |
|---|---|---|---|---|

ncopy

free

Heap (malloc/free)

Read/Write Segment

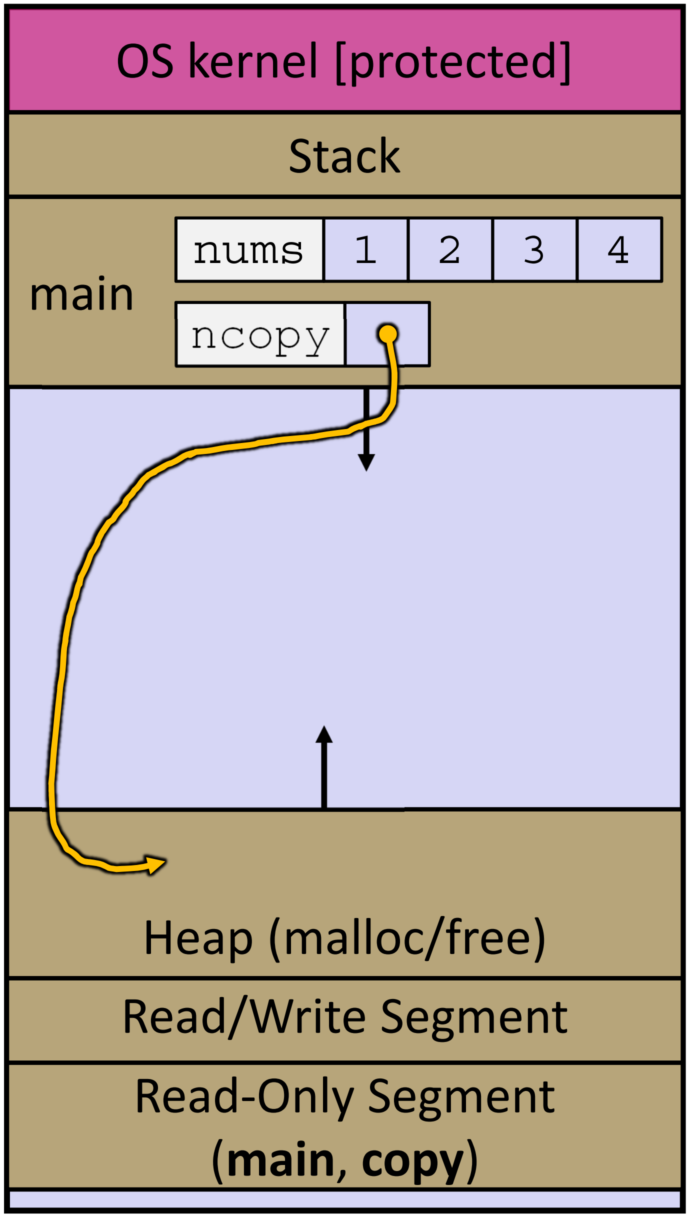
Read-Only Segment

**(main, copy)**

# Heap and Stack Example

arraycopy.c

```
#include <stdlib.h>

int* copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(size*sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];

  return a2;
}

int main(int argc, char** argv) {
  int nums[4] = {1, 2, 3, 4};
  int* ncopy = copy(nums, 4);
  // .. do stuff with the array ..
  free(ncopy);
  return EXIT_SUCCESS;
}
```



# 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