Structs & The Heap

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevine Bernat

Mati Davis

Chandravaran Kunjeti

Shufan Liu

Jialin Cai

Donglun He

Heyi Liu

Eddy Yang

Poll

- How was your three day weekend?

Logistics

- Check-in00: **Due Monday 1/23 @ 10:00 AM**
  - Short “quiz” on canvas to make sure you are caught up with lectures
  - Unlimited Attempts
  - Typically released Wednesday nights or on Thursday
- Pre-Semester Survey: **Due Tuesday 1/24 @ 11:59 PM**
  - Survey to get information on how to make the course better suited to everyone
- HW00: **Due Thursday 1/26 @ 11:59 PM**
  - Implement LinkedList & HashTable
  - You should have everything you need after this lecture
  - HWs can take a while
  - **DO NOT PUT THIS OFF, PLEASE GET STARTED**

Lecture Outline

- **Structs in C**
- The Heap & Dynamic Memory
- Data Structures in C & Function Pointers

Structured Data

- A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Acts similarly to primitive variables
- Generic declaration:

```
struct Point {
  float x;
  float y;
};

struct Point pt;
struct Point origin = {0.0f, 0.0f};
pt = origin; // pt now contains 0.0f, 0.0f
```

Default values are still garbage!

<- Initializer List

Can be assigned into,
used as parameters, etc.

typedef

- Generic format: `typedef type name;`
- Allows you to define new data type *names/synonyms*
  - Both `type` and `name` are usable and refer to the same type

```
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char *str;

// make "Point" a synonym for "struct point_st { ... }"
typedef struct point_st {
  superlong x;
  superlong y;
} Point;

Point origin = {0, 0};
```

struct point_st == Point

Don't have to type "struct" when declaring a variable anymore ☺

Accessing struct Fields

- Use “ . ” to refer to a field in a struct
- Use “->” to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

```
typedef struct point_st {
  float x, y;
} Point;

int main(int argc, char** argv) {
  Point p1 = {0.0, 0.0};
  Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return 0;
}
```

Poll Everywhere

pollev.com/tqm

- When run,
  what does this code print?

  A. 24.0

  B. 59.50

  C. 35.1

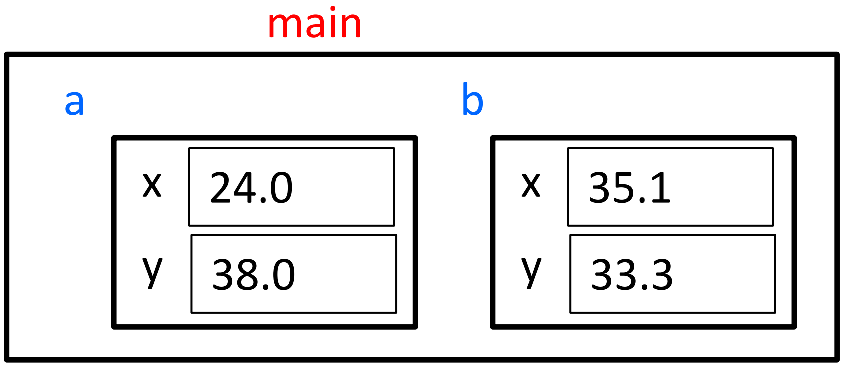
  D. Segmentation Fault

  E. We're Lost…

```c
#include <stdio.h>

typedef struct point_st {
  float x, y;
} Point;

void mystery(Point p, Point* ptr,
float val) {
  *ptr = p;
  p.x = val;
  p.y = val;
}

int main() {
  Point a = {24.0, 38.0};
  Point b = {35.1, 33.3};
  mystery(a, &b, 59.50);
  printf("x: %f\n",b.x);
}
```

Poll Everywhere

pollev.com/tqm



main

a

x 24.0

y 38.0

b

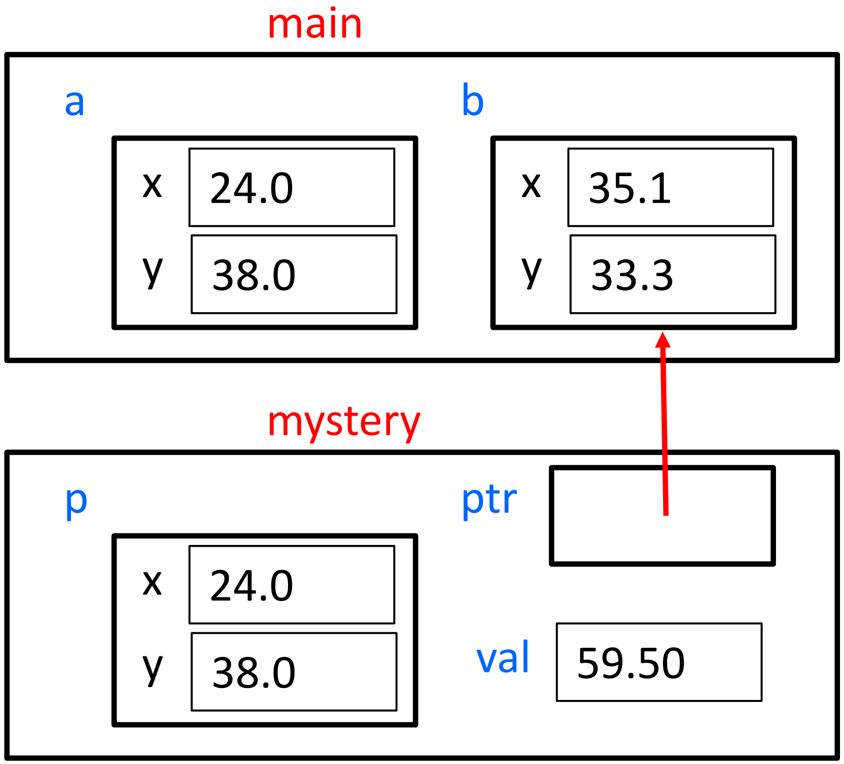
x 35.1

y 33.3

```
#include <stdio.h>

typedef struct point_st {
  float x, y;
} Point;

void mystery(Point p, Point* ptr,
float val) {
  *ptr = p;
  p.x = val;
  p.y = val;
}

int main() {
  Point a = {24.0, 38.0};
  Point b = {35.1, 33.3};
  mystery(a, &b, 59.50);
  printf("x: %f\n",b.x);
}
```

Poll Everywhere

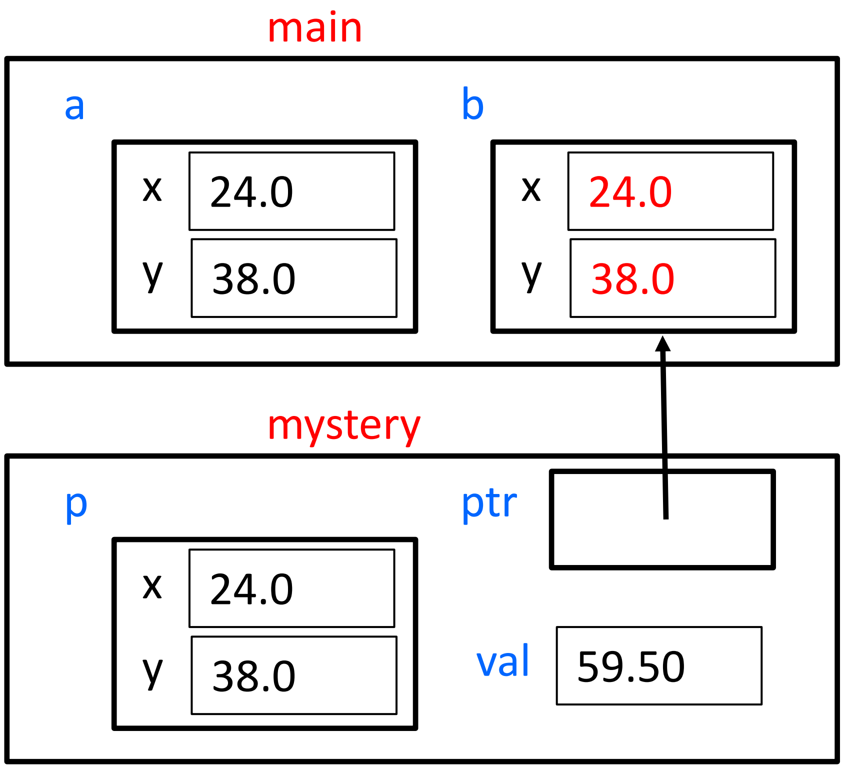
pollev.com/tqm



```
#include <stdio.h>

typedef struct point_st {
  float x, y;
} Point;

void mystery(Point p, Point* ptr,
float val) {
  *ptr = p;
  p.x = val;
  p.y = val;
}

int main() {
  Point a = {24.0, 38.0};
  Point b = {35.1, 33.3};
  mystery(a, &b, 59.50);
  printf("x: %f\n",b.x);
}
```

Poll Everywhere

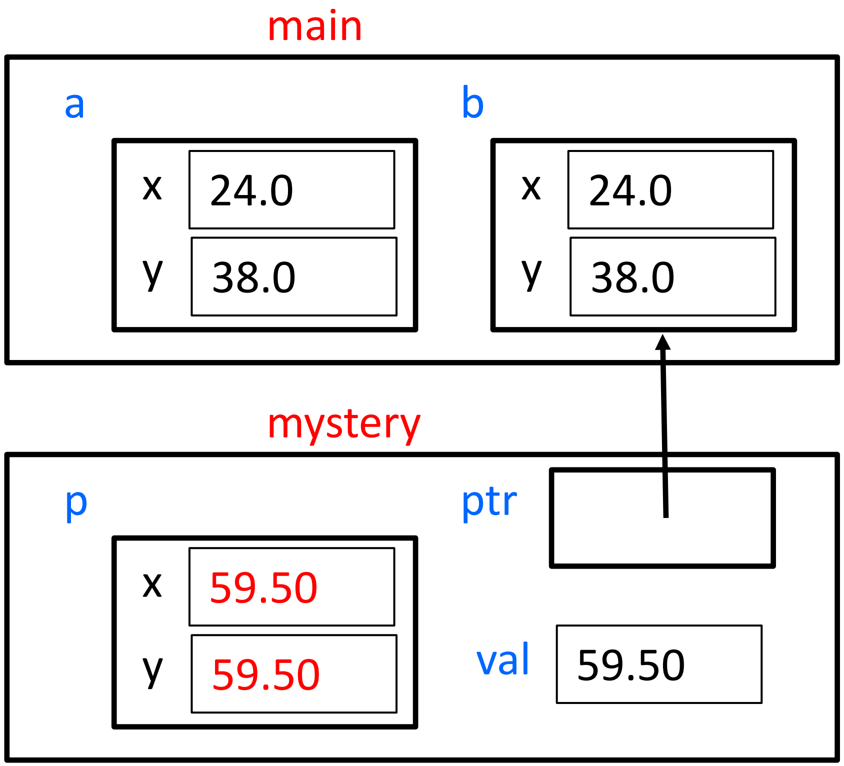
pollev.com/tqm



```
#include <stdio.h>

typedef struct point_st {
  float x, y;
} Point;

void mystery(Point p, Point* ptr,
float val) {
  *ptr = p;
  p.x = val;
  p.y = val;
}

int main() {
  Point a = {24.0, 38.0};
  Point b = {35.1, 33.3};
  mystery(a, &b, 59.50);
  printf("x: %f\n",b.x);
}
```

Poll Everywhere

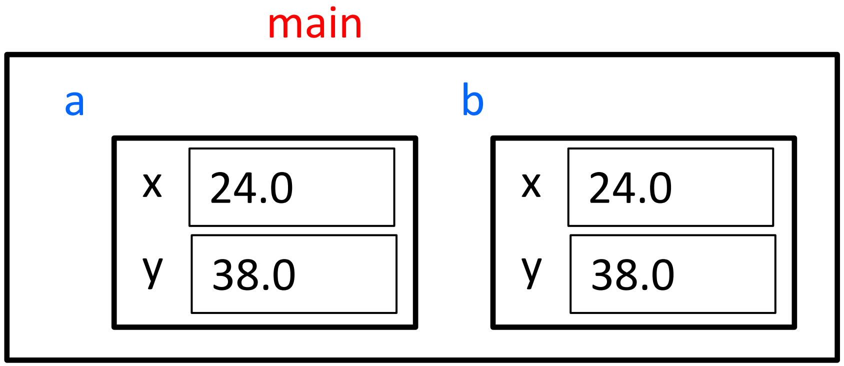
pollev.com/tqm



```
#include <stdio.h>

typedef struct point_st {
  float x, y;
} Point;

void mystery(Point p, Point* ptr,
float val) {
  *ptr = p;
  p.x = val;
  p.y = val;
}

int main() {
  Point a = {24.0, 38.0};
  Point b = {35.1, 33.3};
  mystery(a, &b, 59.50);
  printf("x: %f\n",b.x);
}
```

Poll Everywhere

pollev.com/tqm



A. **24.0**

```
#include <stdio.h>

typedef struct point_st {
  float x, y;
} Point;

void mystery(Point p, Point* ptr,
float val) {
  *ptr = p;
  p.x = val;
  p.y = val;
}

int main() {
  Point a = {24.0, 38.0};
  Point b = {35.1, 33.3};
  mystery(a, &b, 59.50);
  printf("x: %f\n",b.x);
}
```

Lecture Outline

- Structures in C
- **The Heap & Dynamic Memory**
- Data Structures in C

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 0;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

```
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 0;
}
```

- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns

Dynamic Allocation MUST be used if the number elements varies at run time

Poll Everywhere

pollev.com/tqm

- When run,
  what does this code print?

  A. **0**

  B. **5950**

  C. **5950 * 5950**

  D. **A return address**

  E. **Undefined Behavior**

  F. **Does Not Compile**

  G. **We're Lost…**

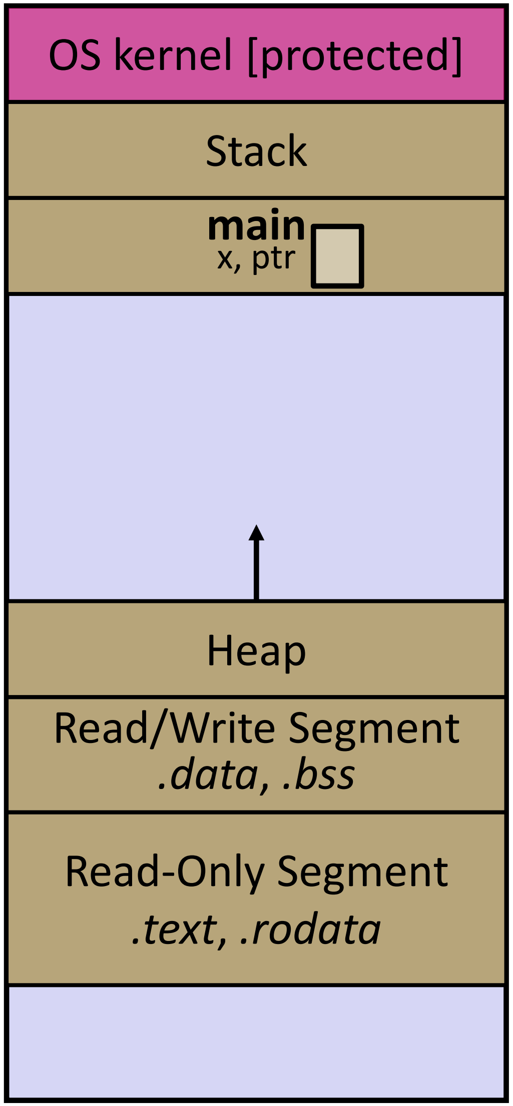
```c
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

Answer

Answer: Undefined Behaviour

local_addr.c

```
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

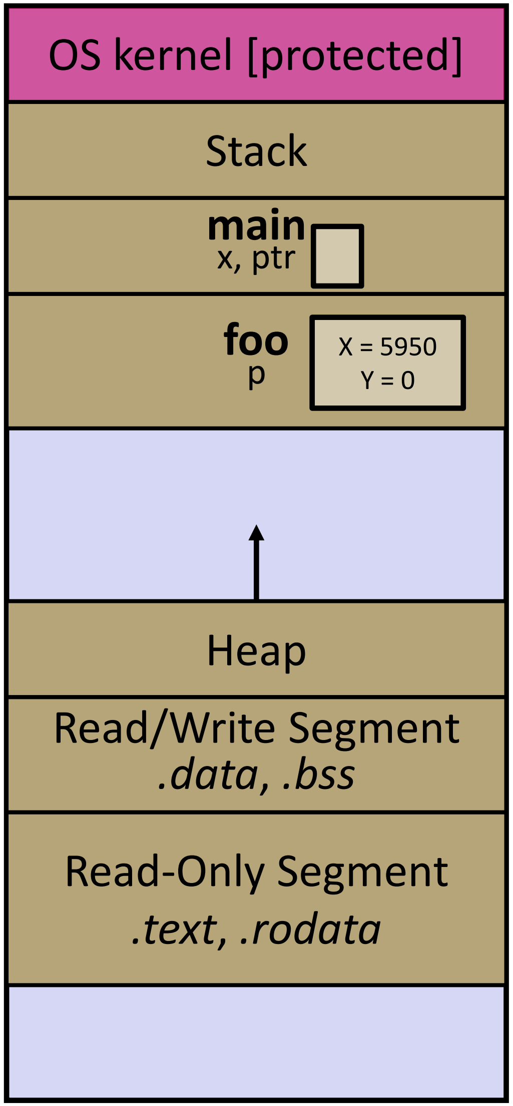


Answer

Answer: Undefined Behaviour

local_addr.c

```c
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

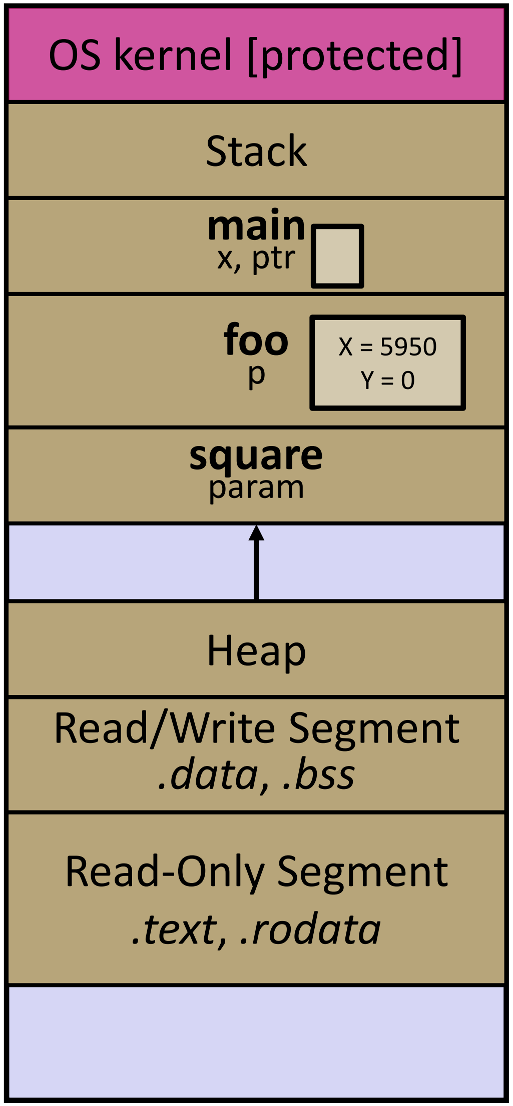


Answer

Answer: Undefined Behaviour

local_addr.c

```
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

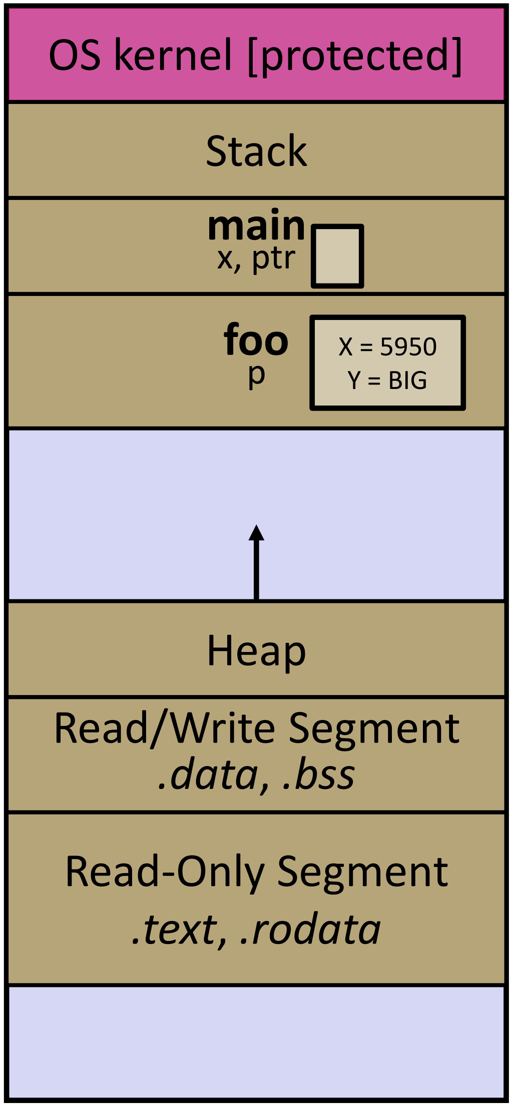


Answer

Answer: Undefined Behaviour

local_addr.c

```
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

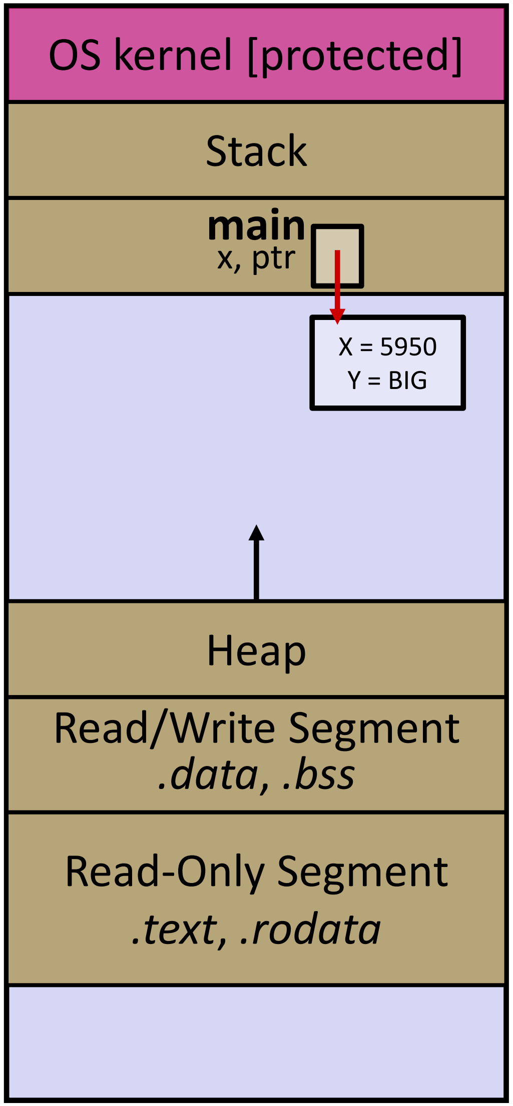


Answer

Answer: Undefined Behaviour

local_addr.c

```
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

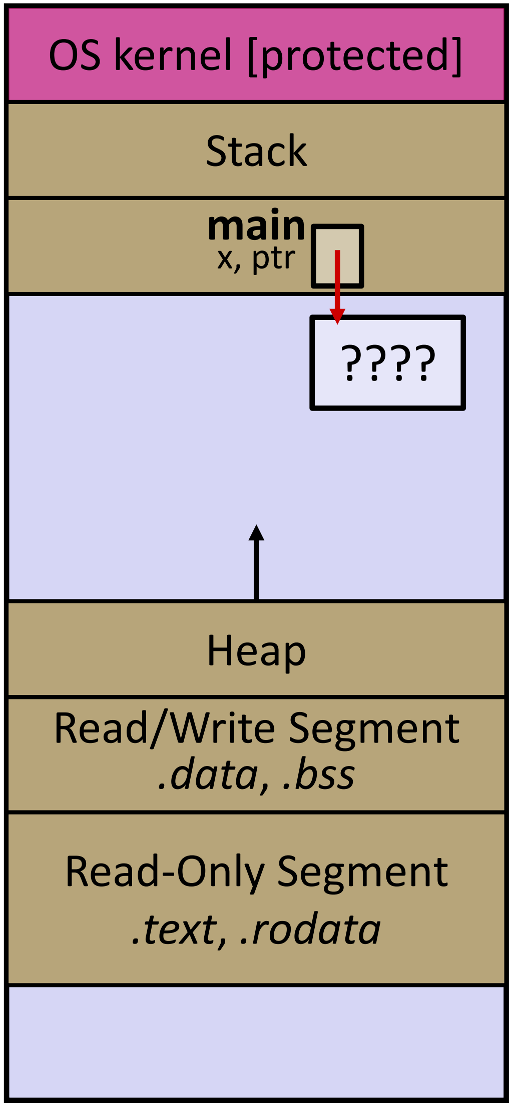


Answer

Answer: Undefined Behaviour

local_addr.c

```c
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```

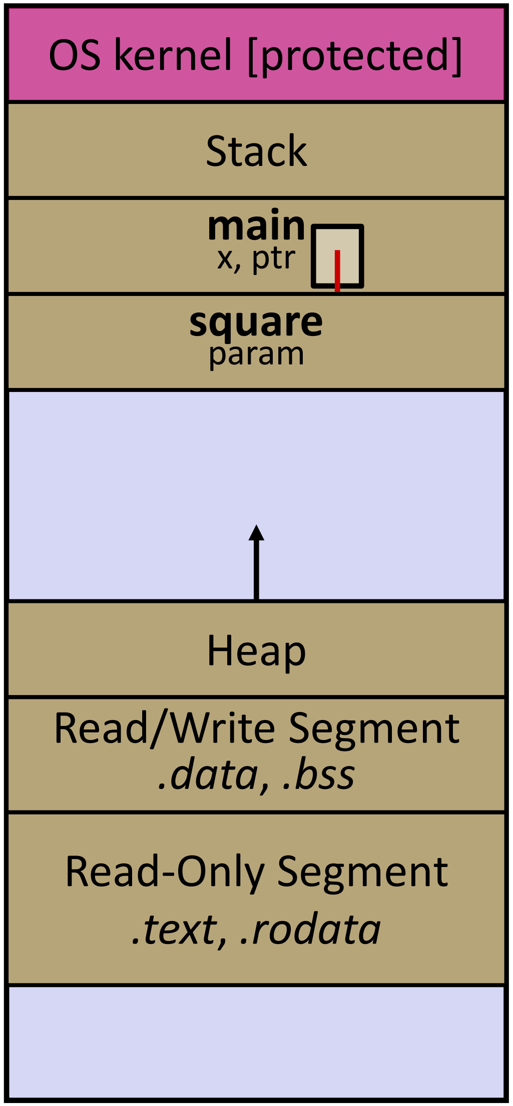


Answer

Answer: Undefined Behaviour

local_addr.c

```
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```



OS kernel [protected]

Stack

main
x, ptr

square
param

Heap

Read/Write Segment
.data, *.bss*

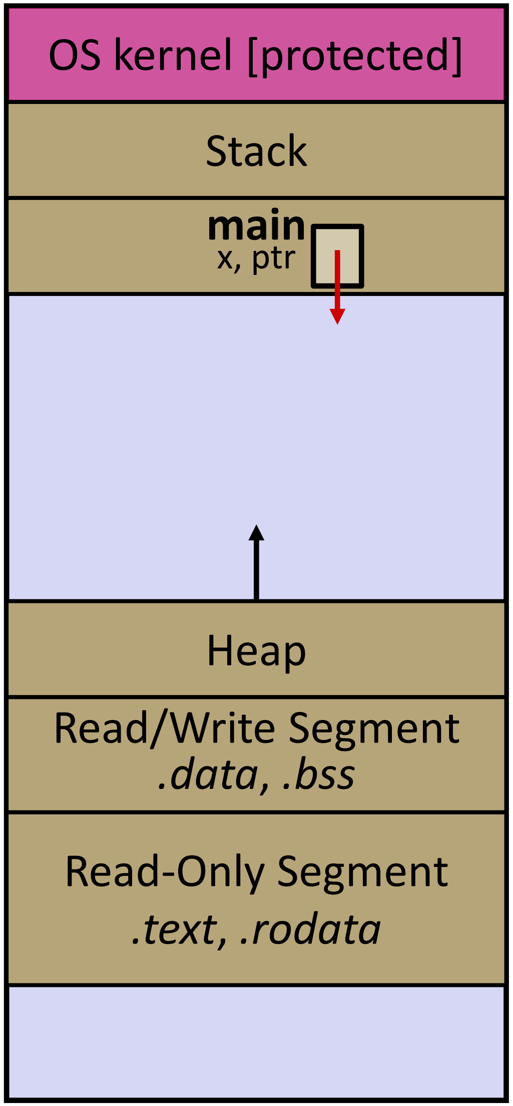
Read-Only Segment
.text, *.rodata*

Answer

Answer: Undefined Behaviour

local_addr.c

```
#include <stdio.h>

typedef struct point_st {
  int x, y;
} Point;

int square(int param){
  return param * param;
}

Point* foo() {
  Point p = {5950, 0};
  p.y = square(p.x);
  return &p;
}

int main(){
  Point* ptr = foo();
  int x = square(2);
  printf("%d\n", ptr->x);
}
```



OS kernel [protected]

Stack

main
x, ptr

Heap

Read/Write Segment
.data, *.bss*

Read-Only Segment
.text, *.rodata*

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!

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

Aside: `sizeof`

- `sizeof` operator can be applied to a variable or a type and it evaluates to the size of that type in bytes
- Examples:
  - `sizeof(int)` – returns the size of an integer
  - `sizeof(double)` – returns the size of a double precision number
  - `struct my_struct s;`
    - `sizeof(s)` – returns the size of the struct s
  - `my_type *ptr`
    - `sizeof (*ptr)` – returns the size of the type pointed to by ptr

- Very useful for Dynamic Memory

What is Dynamic Memory Allocation?

- We want Dynamic Memory Allocation
  - Dynamic means "at run-time"
  - The compiler and the programmer don't have enough information to make a final decision on how much to allocate
  - Your program explicitly requests more memory at run time
  - The language allocates it at runtime, maybe with help of the OS

- Dynamically allocated memory persists until either:
  - A garbage collector collects it (automatic memory management)
  - Your code explicitly deallocates it (manual memory management)

- C requires you to manually manage memory
  - More control, and more headaches

Heap API

- Dynamic memory is managed in a location in memory called the "Heap"
  - The heap is managed by user-level runetime library (libc)
  - Interface functions found in `<stdlib.h>`
- Most used functions:
  - **`void *malloc(size_t size);`**
    - Allocates memory of specified size
  - **`void free(void *ptr);`**
    - Deallocates memory

- Note: **`void*`** is “generic pointer”. It holds an address, but doesn’t specify what it is pointing at.
- Note 2: **`size_t`** is the integer type of **`sizeof()`**

malloc()

- `void *malloc(size_t size);`
- `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 = malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...    // do stuff with arr
```

ALWAYS CHECK FOR NULL

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`**)
  - Freed memory becomes eligible for future allocation
  - `free(NULL);` does nothing.
  - The bits in the pointer are *not changed* by calling free
    - Defensive programming: can set pointer to `NULL` after freeing it

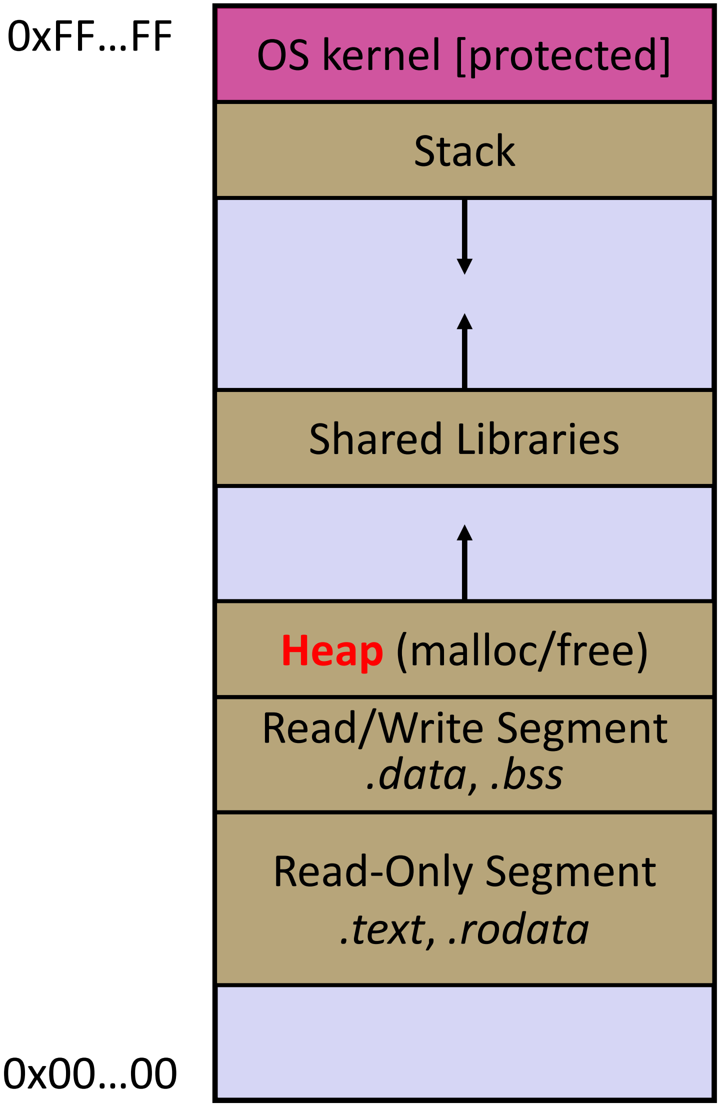
```
float* arr = malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...              // do stuff with arr
free(arr);
arr = NULL;      // OPTIONAL
```

The Heap

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`malloc`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`free`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

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

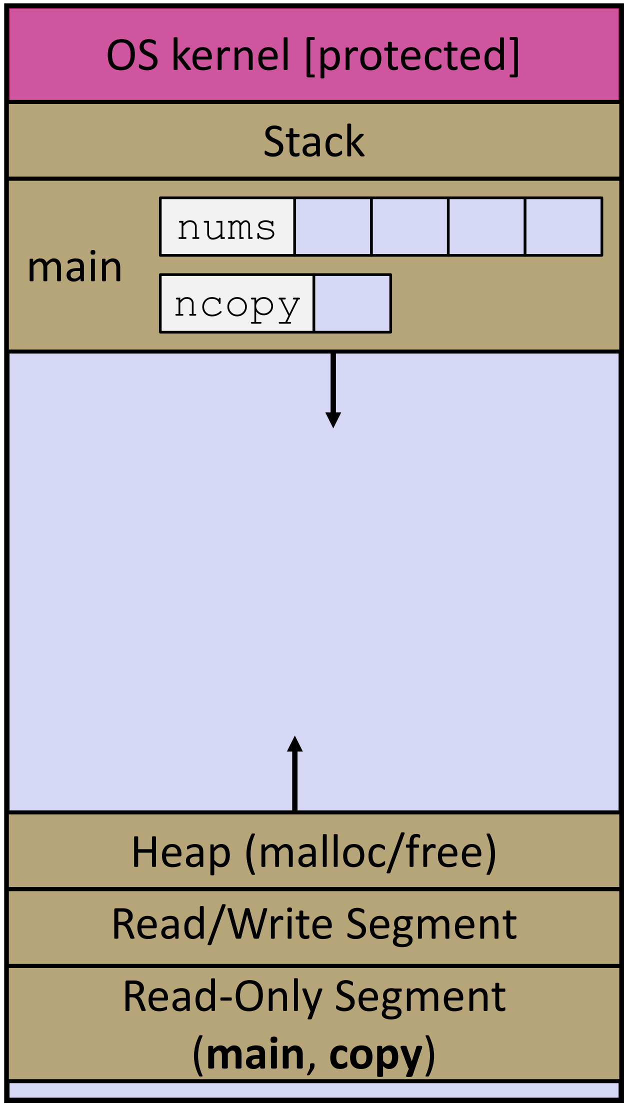


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```



OS kernel [protected]

Stack

main

nums

ncopy

Heap (malloc/free)

Read/Write Segment

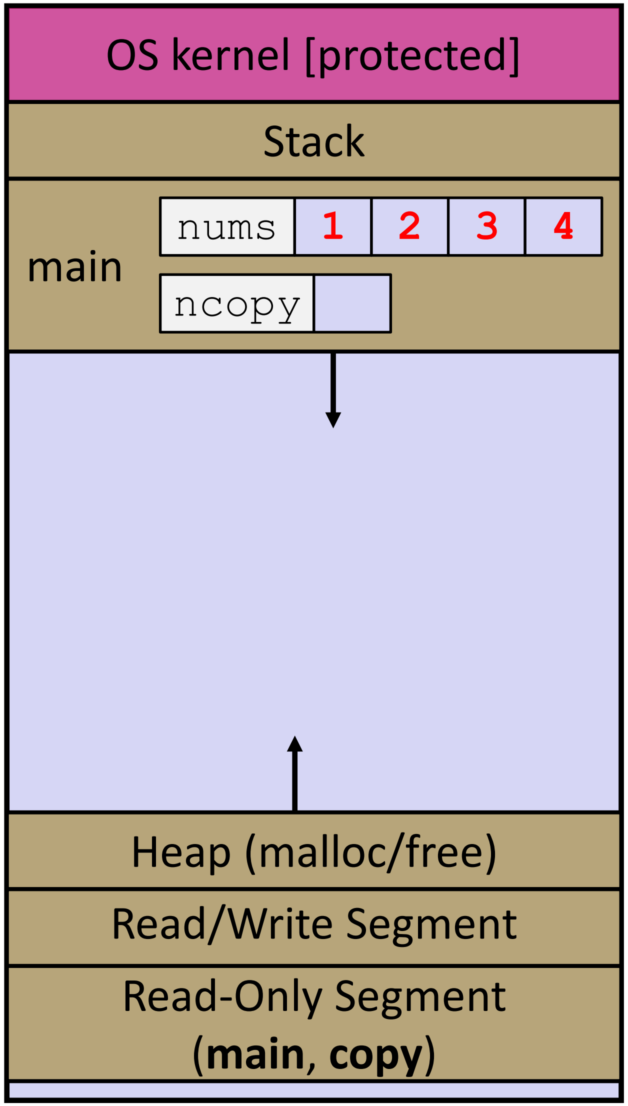
Read-Only Segment (**main**, **copy**)

Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

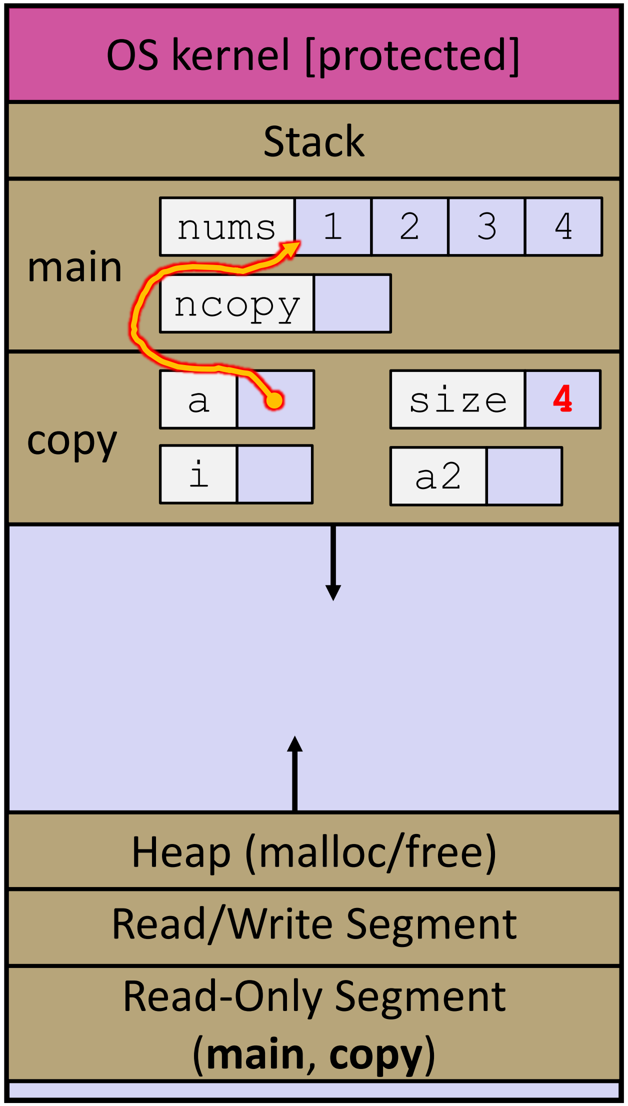


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

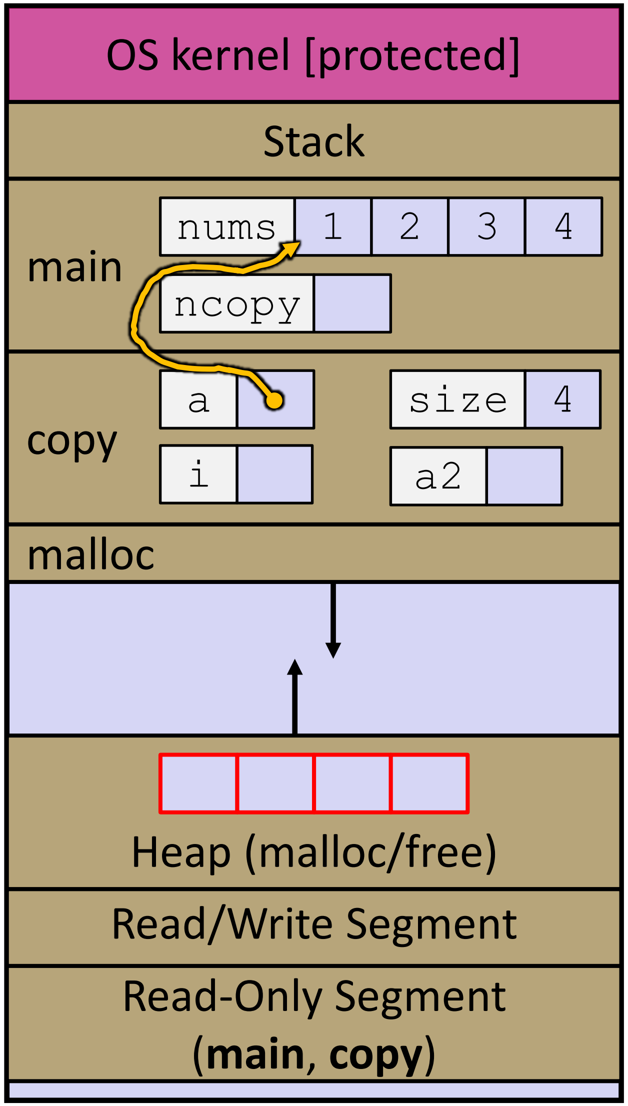


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

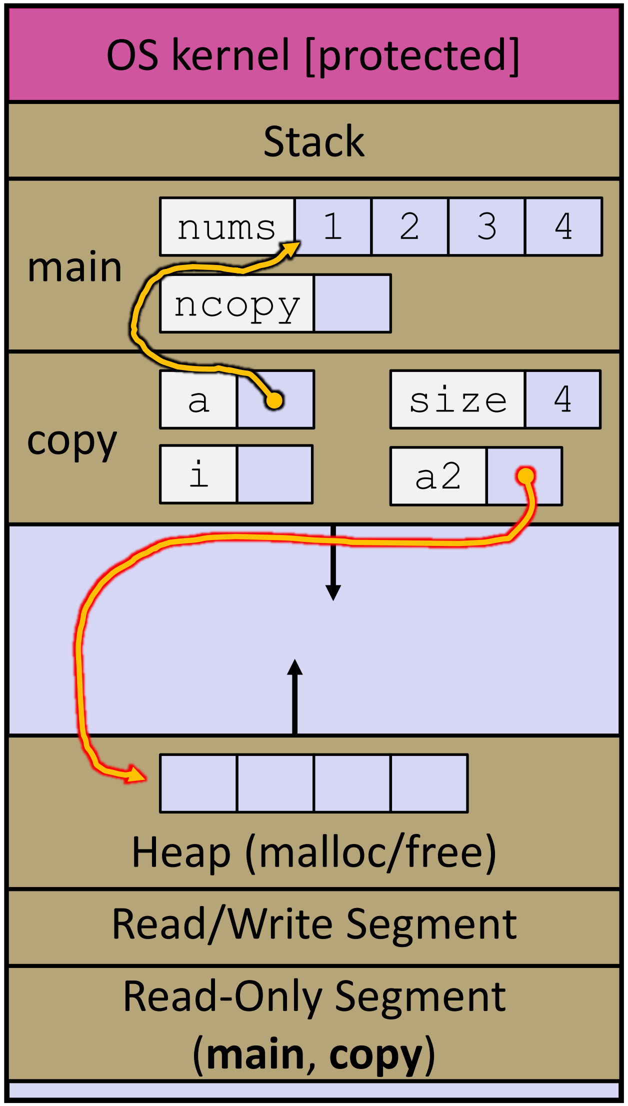


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

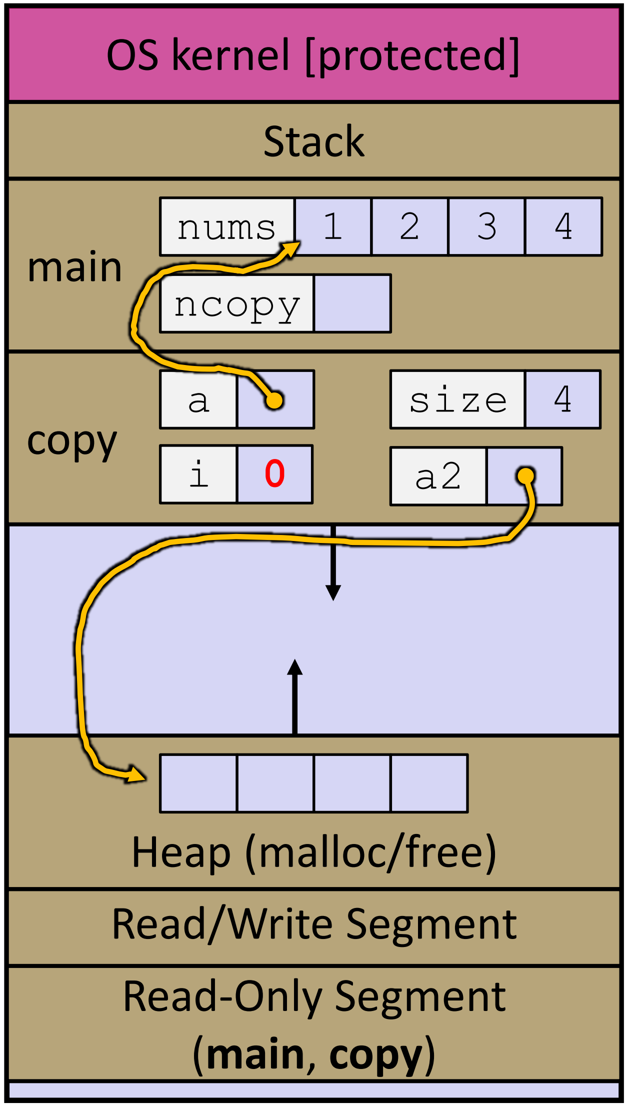


Heap and Stack Example

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 0;
}
```

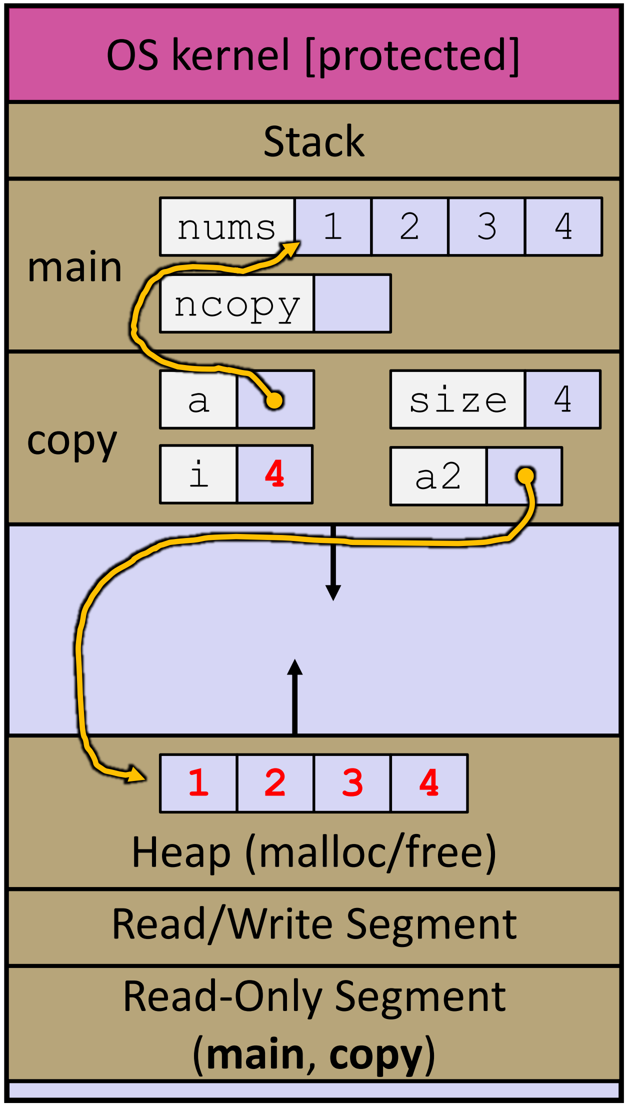


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

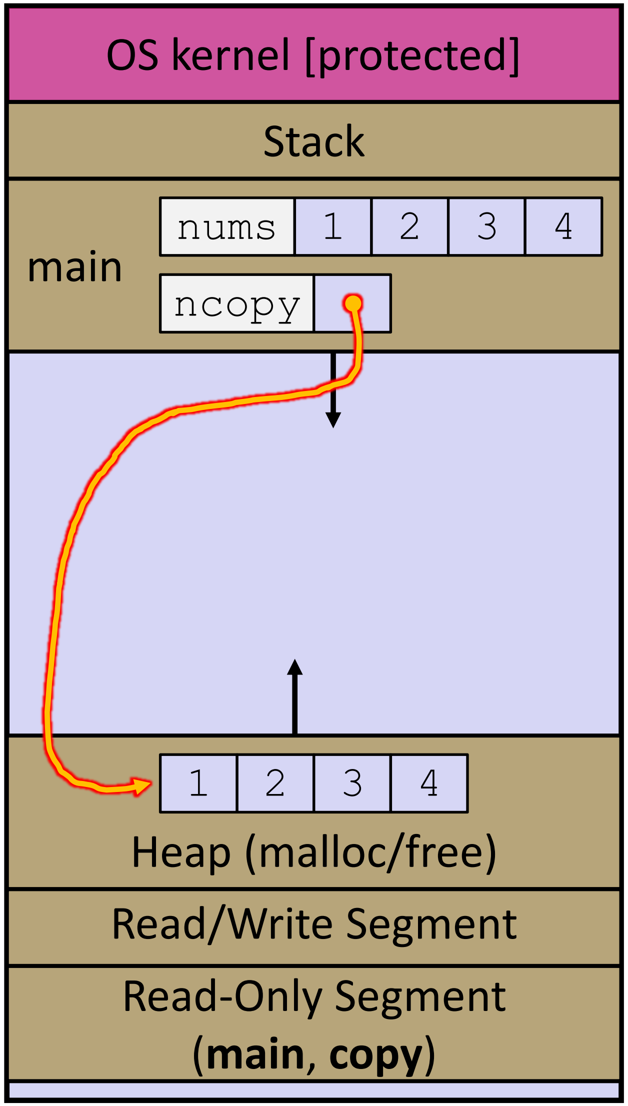


Heap and Stack Example

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 0;
}
```

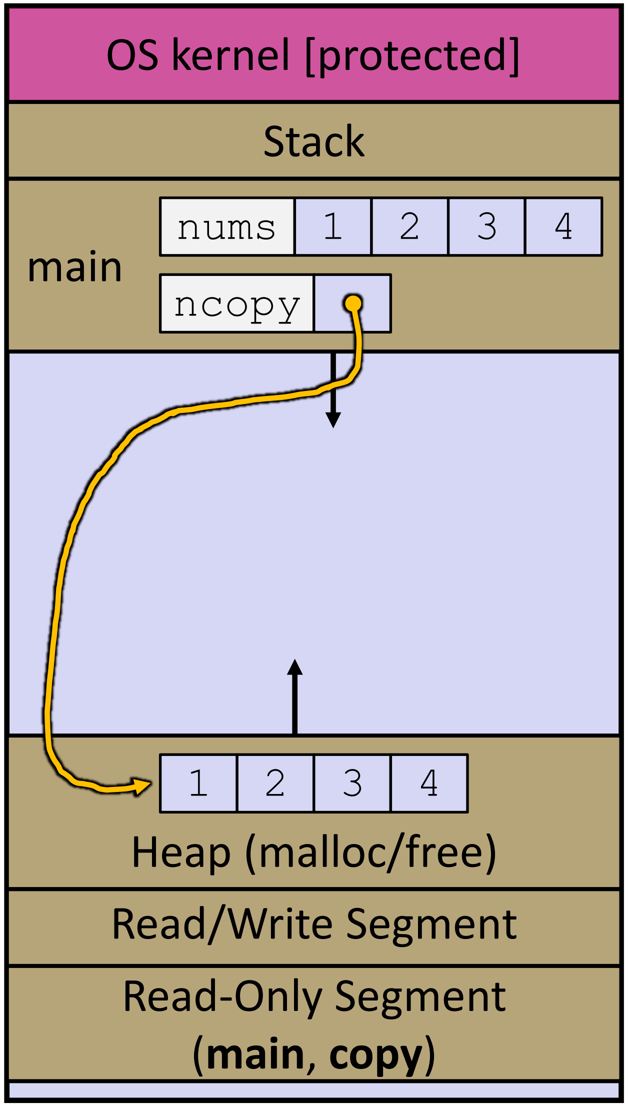


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

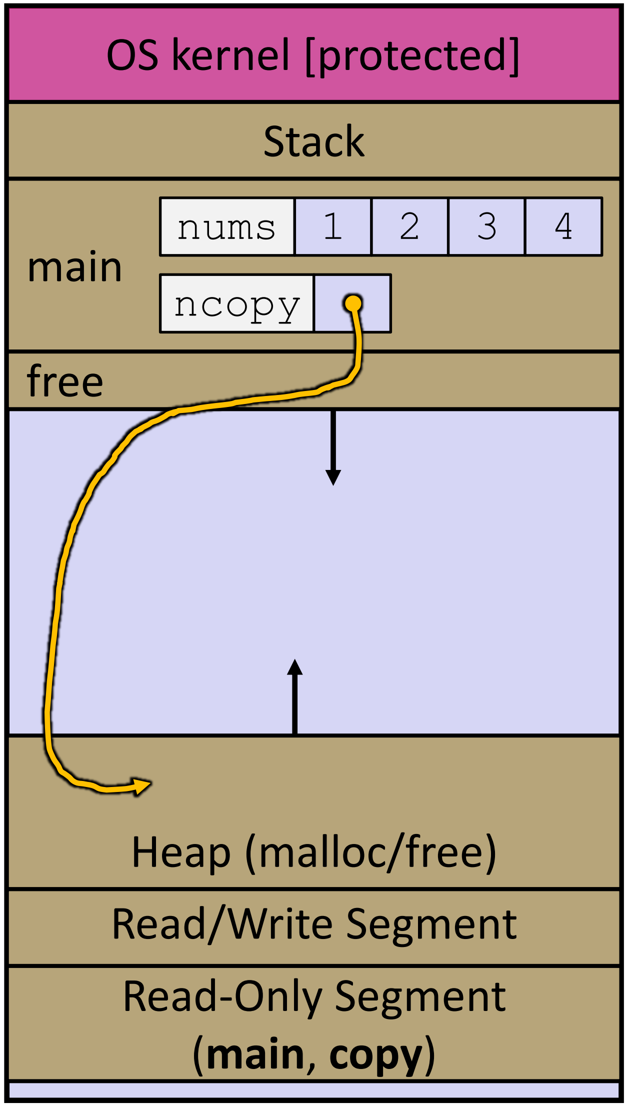


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```

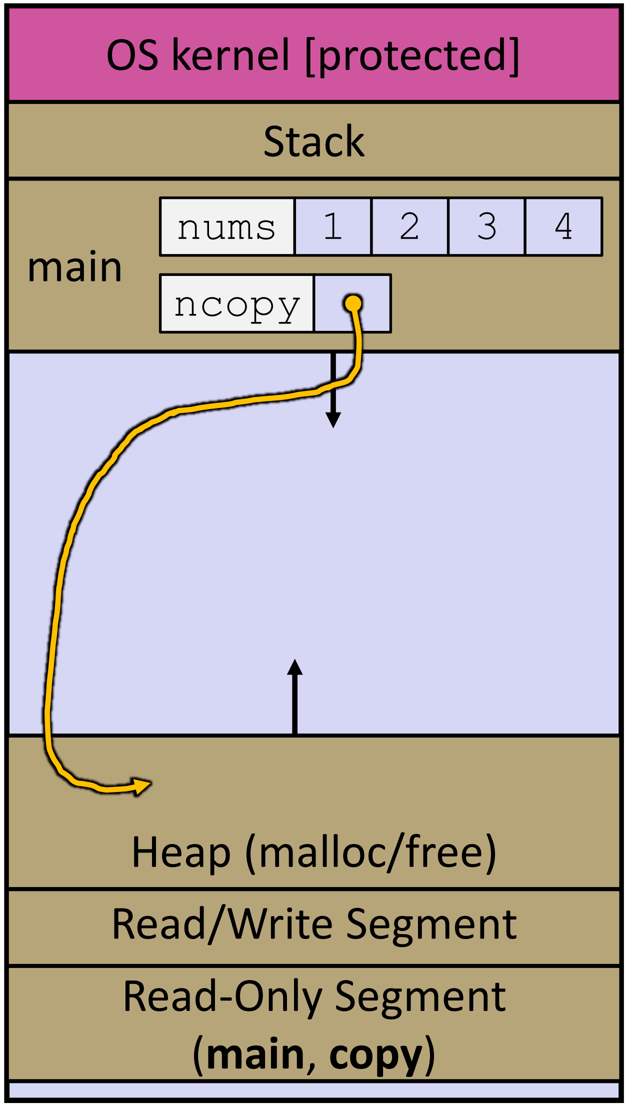


Heap and Stack Example

Note: Arrow points to *next* instruction.

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 0;
}
```



Dynamic Memory Pitfalls

- Buffer Overflows
  - E.g. ask for 10 bytes, but write 11 bytes
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on malloc'd strings
- Not checking for **`NULL`**
  - Malloc returns NULL if out of memory
  - Should check this after every call to malloc
- Giving **`free()`** a pointer to the middle of an allocated region
  - Free won't recognize the block of memory and may crash
- Giving free() a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- **`malloc`** does NOT initialize memory
  - There are other functions like **`calloc`** that will zero out memory

Memory Leaks

- The most common Memory Pitfall
- What happens if we malloc something, but don't free it?
  - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **`free`**d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash

- Garbage Collection
  - Automatically "frees" anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C doesn't

Poll Everywhere

pollev.com/tqm

- Which line below is first to cause a crash?
  - Yes, there are a lot of bugs, but not all cause a crash ☺
  - See if you can find all the bugs!

A. Line 1

B. Line 4

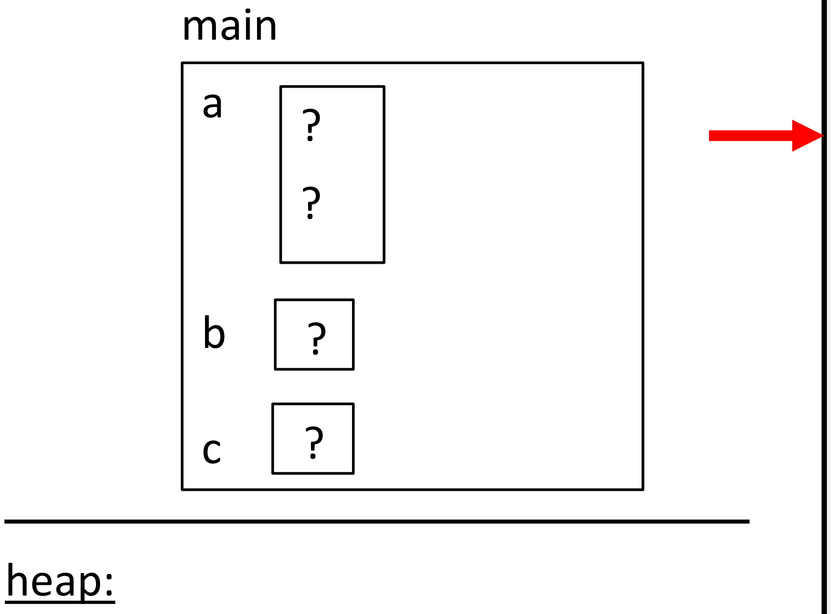
C. Line 6

D. Line 7

E. We're lost…

```
  #include <stdio.h>
  #include <stdlib.h>

  int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

    return 0;
  }
```

Memory Corruption - What Happens?

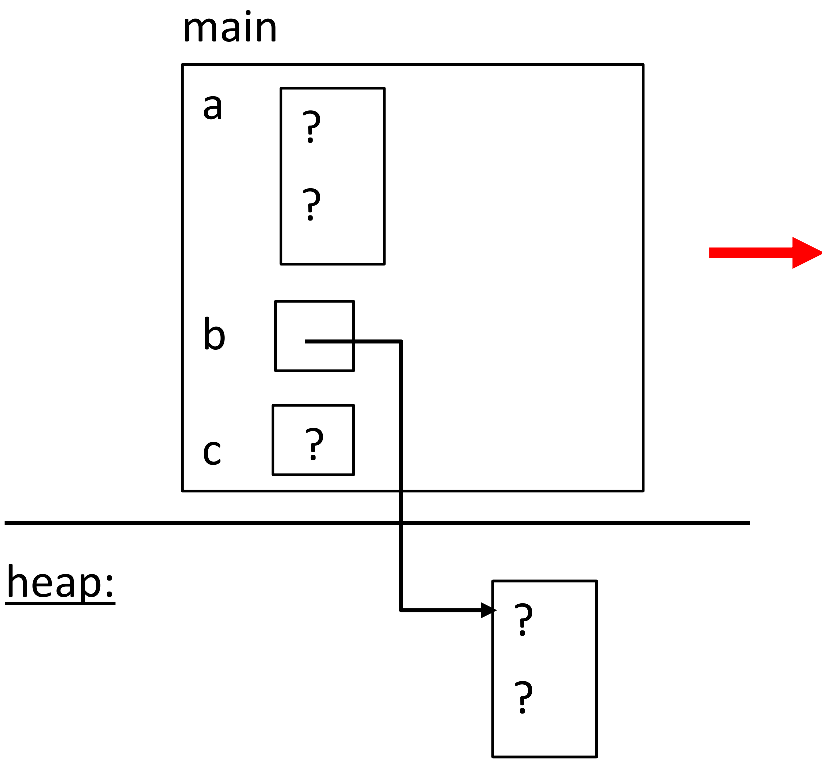


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

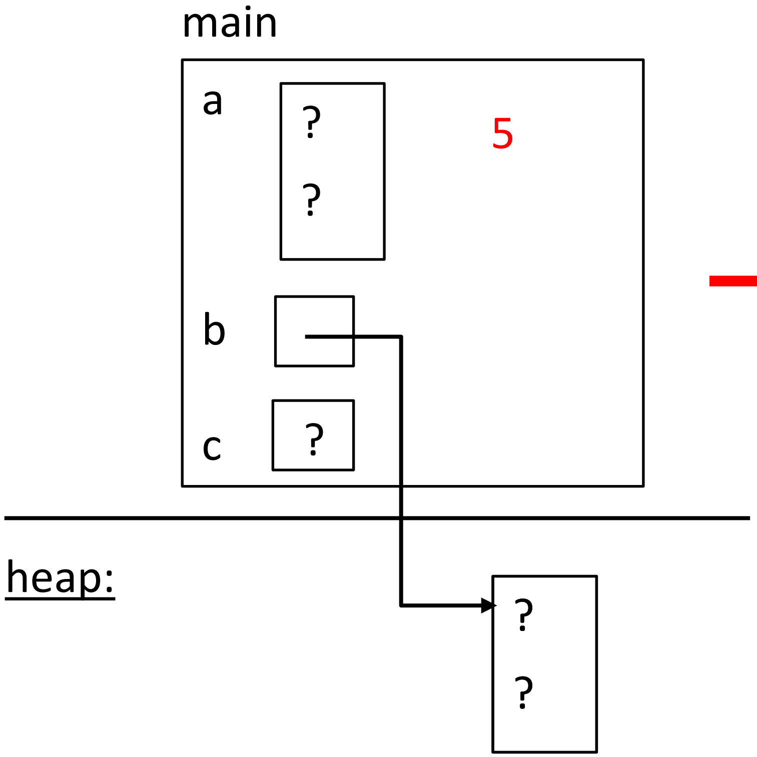


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

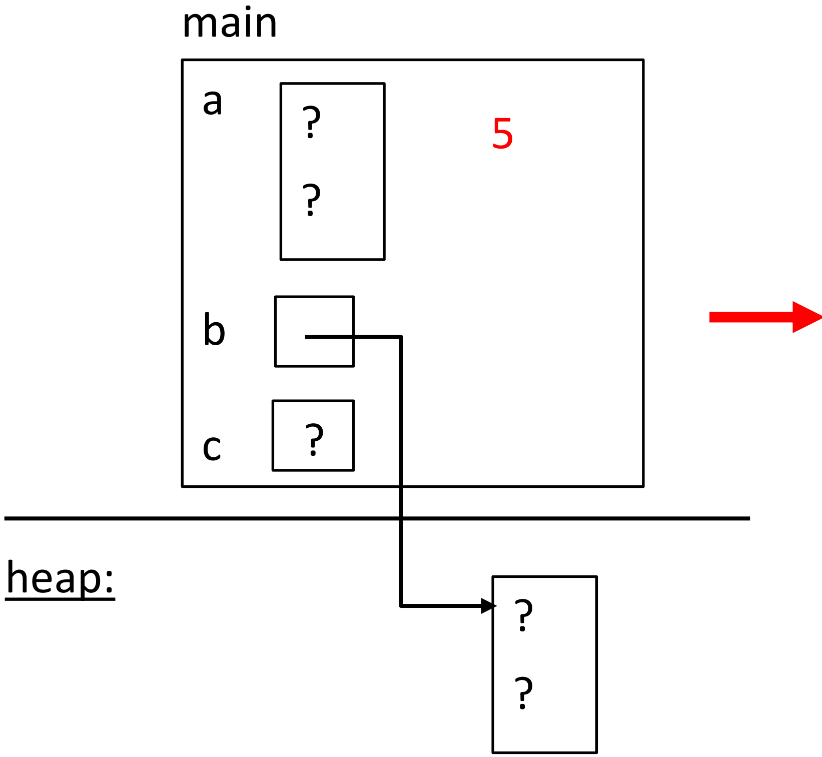


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

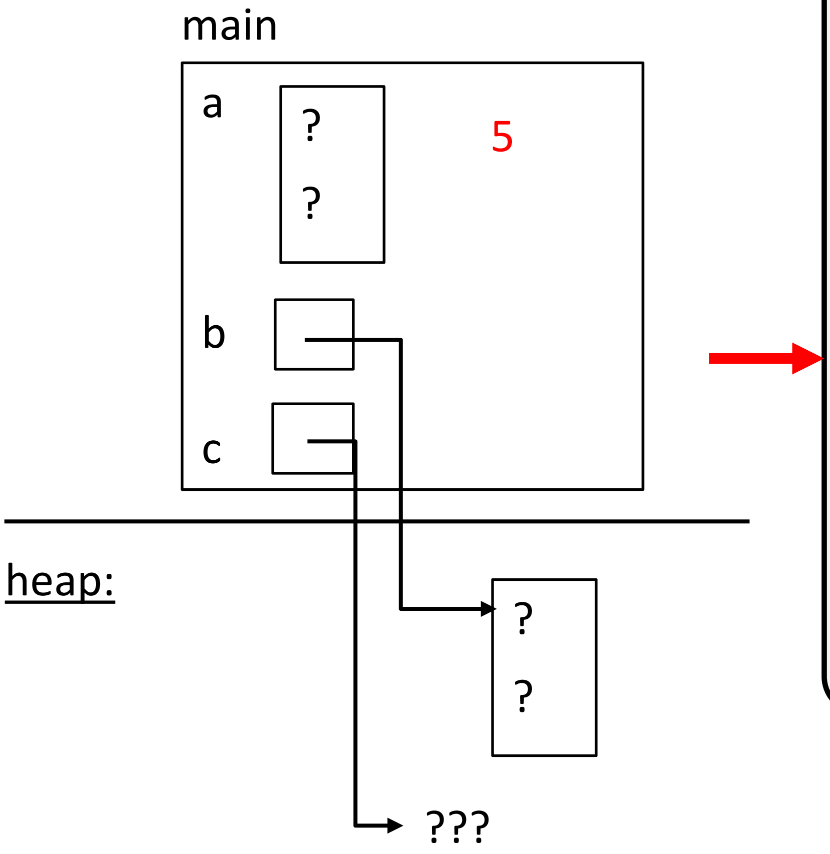


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

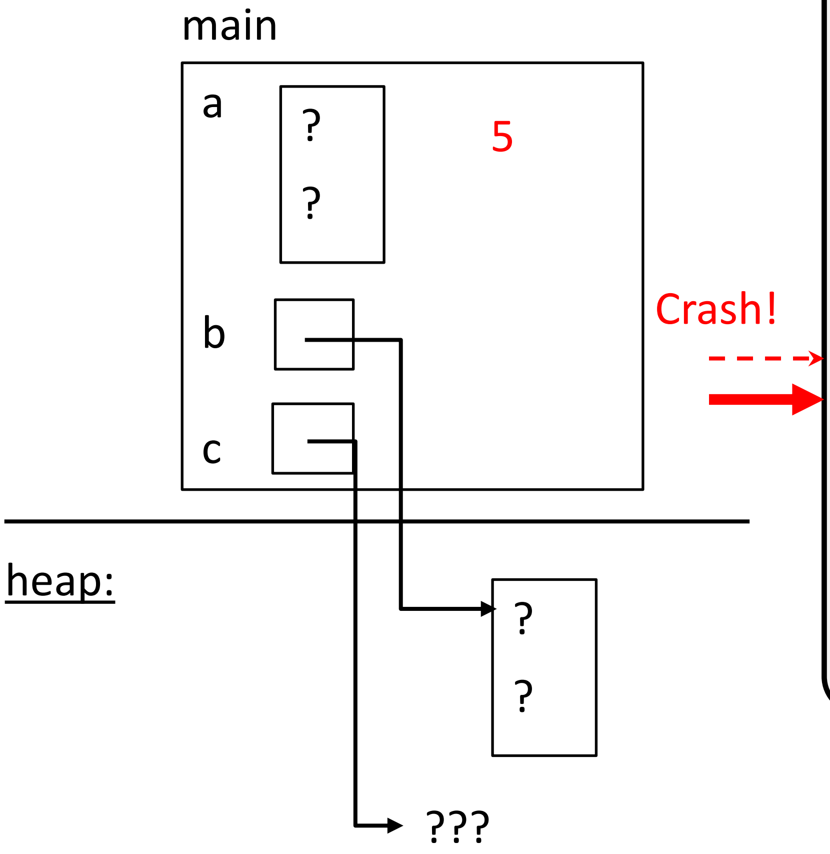


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?

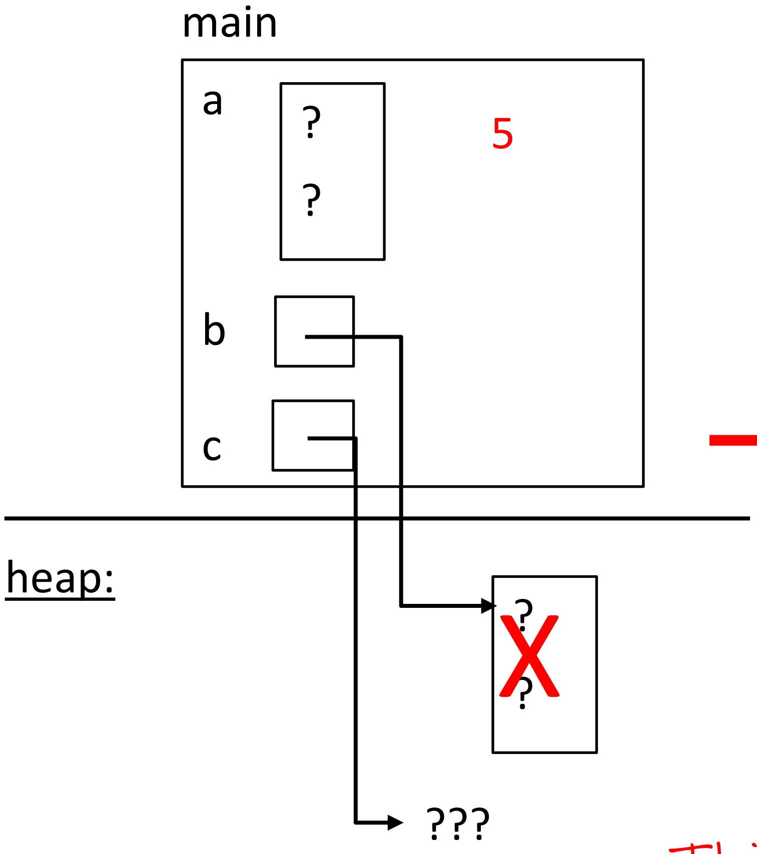


```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));   // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



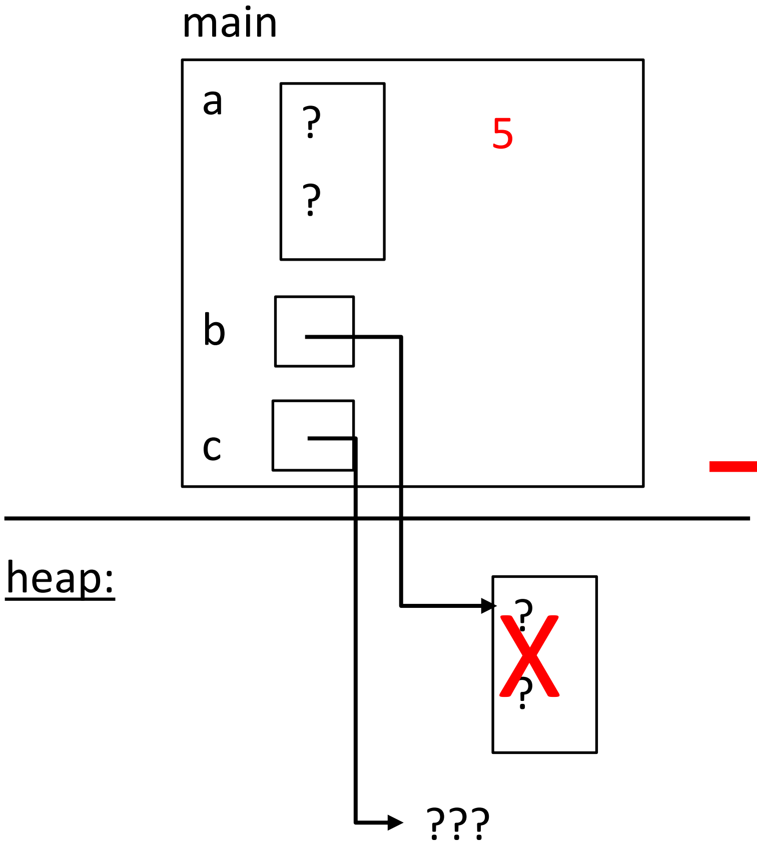
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

This "double free" would also cause the program to crash

memcorrupt.c

Memory Corruption - What Happens?

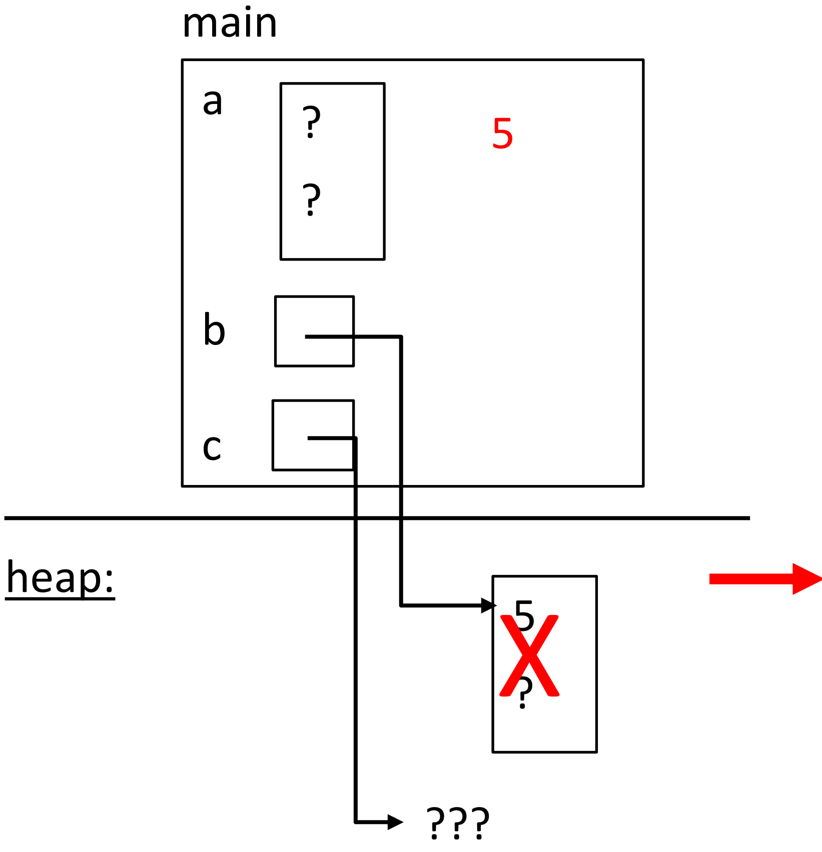


```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

memcorrupt.c

Memory Corruption - What Happens?



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;   // assigns past the end of an array
  b[0] += 2;  // assumes malloc zeros out memory
  c = b+3;    // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);    // double-free the same block
  b[0] = 5;   // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to *next* instruction.

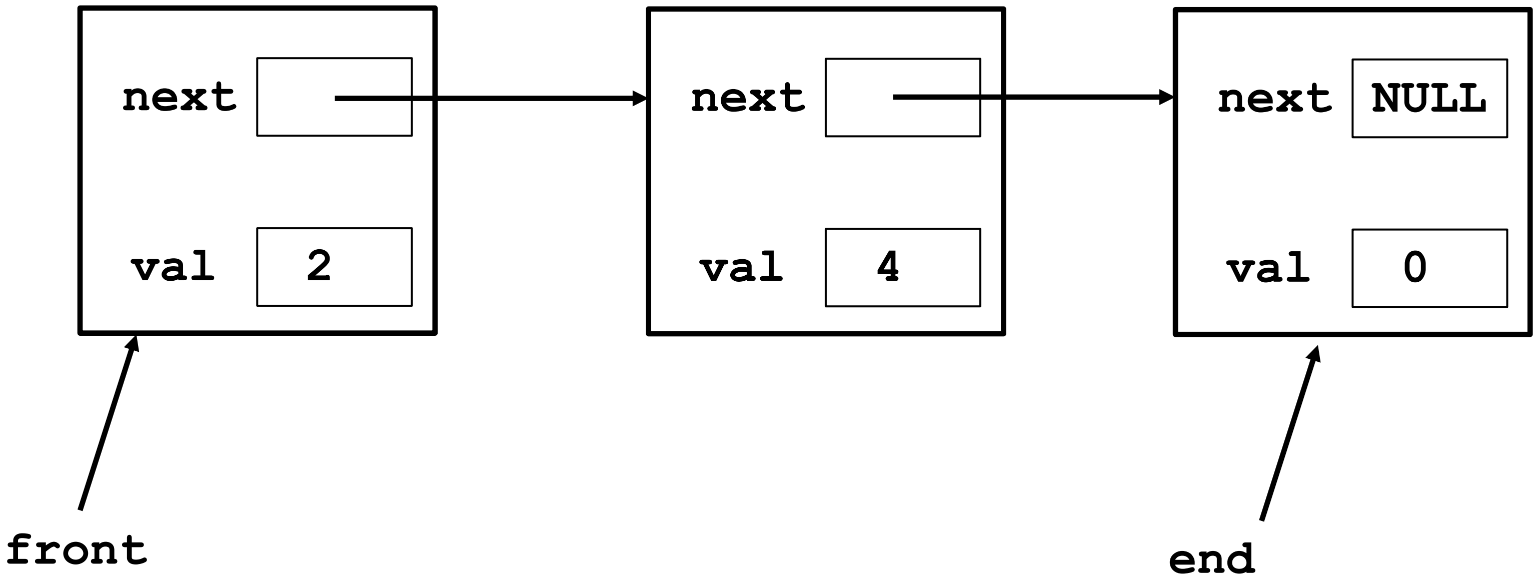
memcorrupt.c

Lecture Outline

- **Structures in C**
- The Heap & Dynamic Memory
- Data Structures in C

Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.

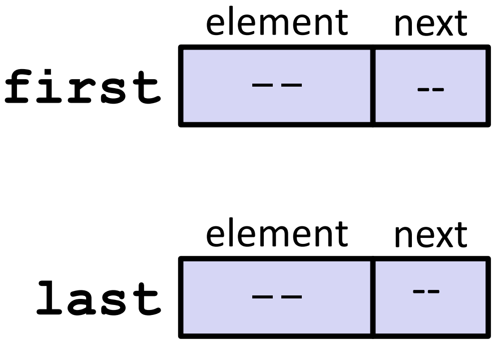


Queue Implementation Demo

- Let’s create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c

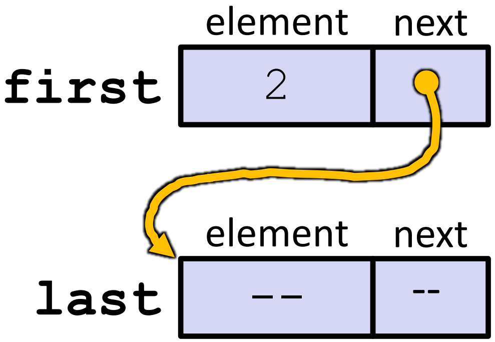


Queue Implementation Demo

- Let’s create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c

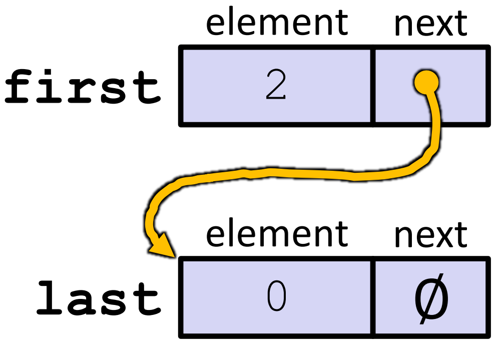


Queue Implementation Demo

- Let's create a naïve implementation for our queue

```
#include <stdio.h>

typedef struct node_st {
  struct node_st* next;
  int val;
} Node;

int main(int argc, char** argv) {
  Node first, last;

  first.val = 2;
  first.next = &last;
  last.val = 0;
  last.next = NULL;
  return 0;
}
```

naive_queue.c



What happens if we want more than two elements?

What happens if we don't know the size we need until run-time?

Naïve Queue "Methods"

- Let's Implement some "methods" for interacting with the queue

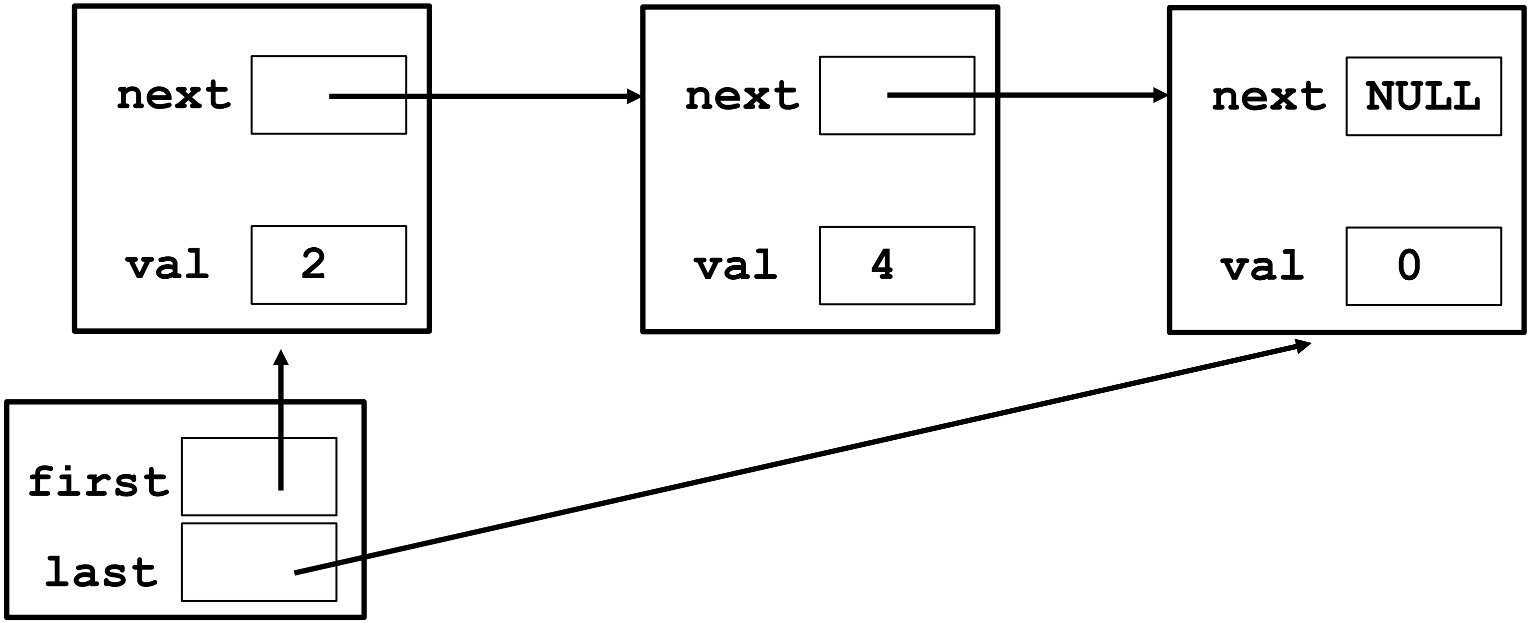
```
Node* Queue_Add(Node* head, int val) {
  Node new_head;
  new_head.next = head;
  new_head.val = val;
  return &new_head;
}

int main(int argc, char** argv) {
  Node* head = NULL;

  head = Queue_Add(head, 2);
  head = Queue_Add(head, 0);
  return 0;
}
```

What's wrong here?

Nodes are on the stack, they don't exist after returning from Queue_Add

Revisiting the Queue Example

- Simple Data structure modeling a queue
  - Implemented with a singly linked list
- Items added to the end and removed from the front.
- We maintain a list of queue elements chained together with pointers.
- We can use Dynamic Allocation to create new elements



Queue Structs

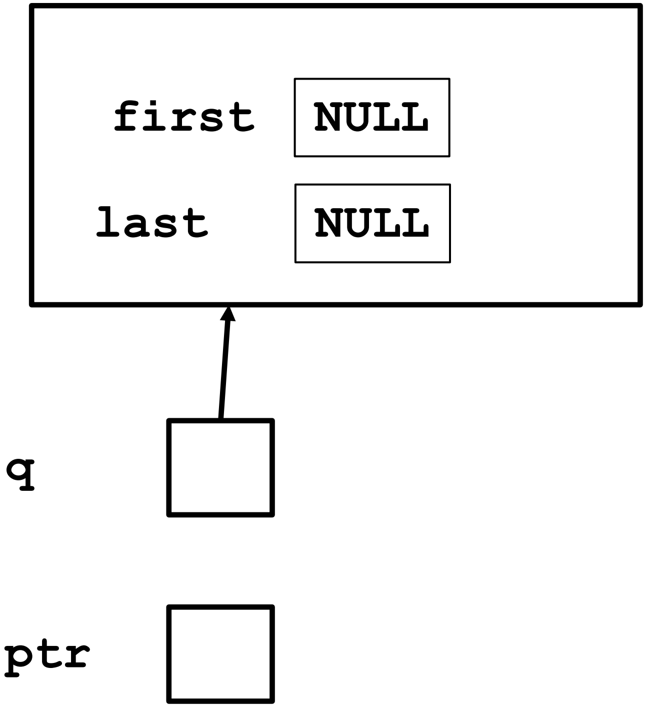
- Note the separate structs:
  - Struct for nodes
  - Struct for overall Queue

- Queue_Allocate() returns a pointer to a Queue struct on the heap that we can later use as an “Object”

```c
typedef struct qnode_st {
  struct qnode_st* next;
  int val;
} Queue_Node;

typedef struct queue_st {
  Queue_Node* first;
  Queue_Node* last;
} Queue;

Queue* Queue_Allocate() {
  Queue* res = malloc(sizeof(Queue));
  if (res != NULL) {
    res->first = NULL;
    res->last = NULL;
  }
  return res;
}
```

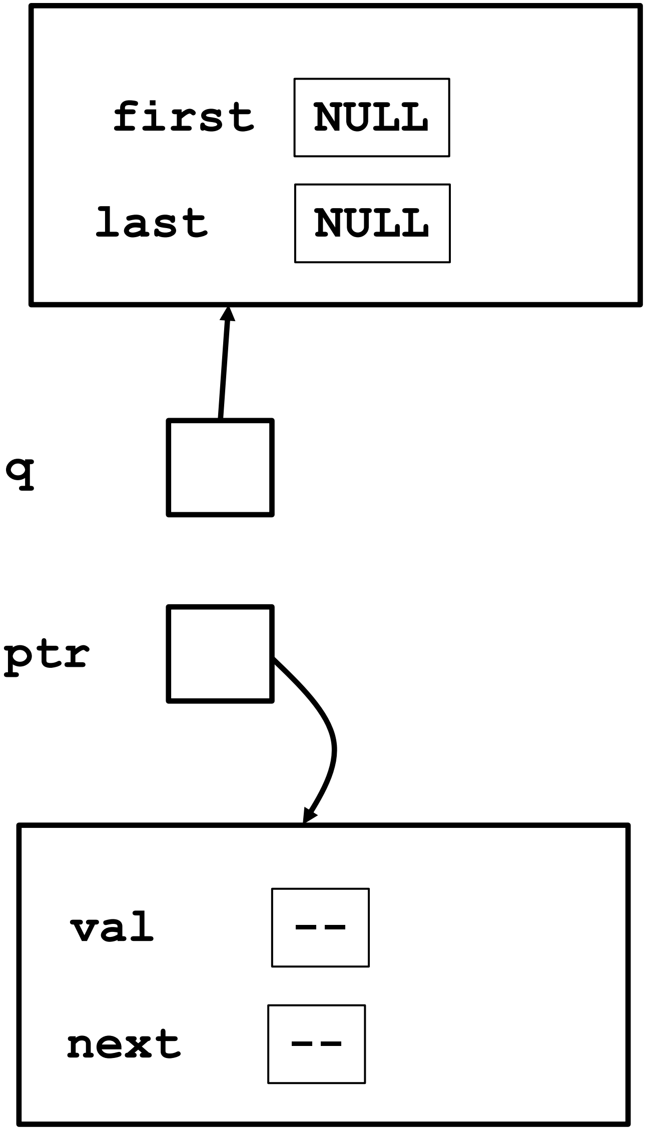
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* ptr;
  ptr = malloc(sizeof(Queue_Node));

  ptr->next = NULL;
  ptr-> val = val;
  if (q->last) {
    q->last->next = ptr;
    q->last = ptr;
  } else {
    q->first = ptr;
    q->last = ptr;
  }
}
```



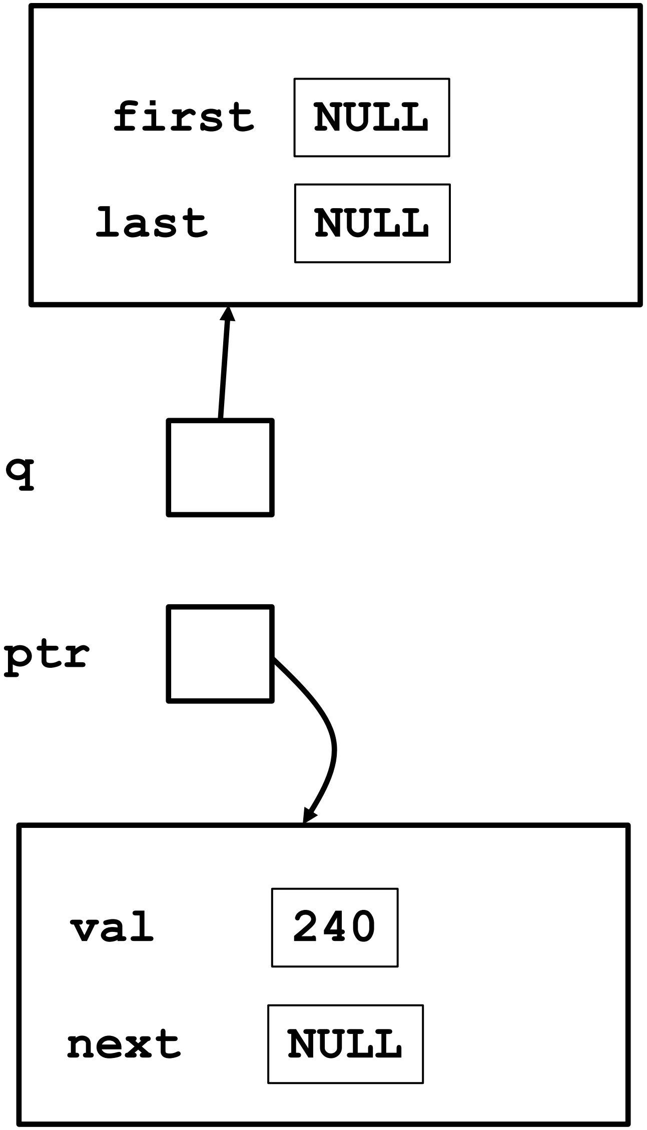
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* ptr;
  ptr = malloc(sizeof(Queue_Node));

  ptr->next = NULL;
  ptr-> val = val;
  if (q->last) {
    q->last->next = ptr;
    q->last = ptr;
  } else {
    q->first = ptr;
    q->last = ptr;
  }
}
```



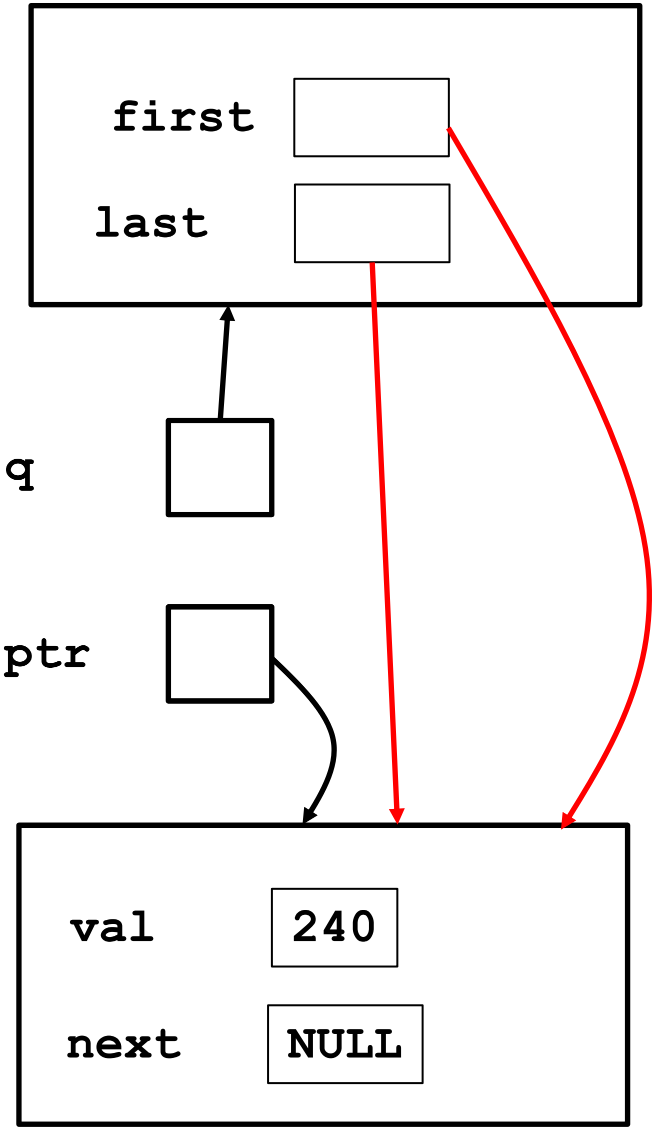
Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* ptr;
  ptr = malloc(sizeof(Queue_Node));

  ptr->next = NULL;
  ptr-> val = val;
  if (q->last) {
    q->last->next = ptr;
    q->last = ptr;
  } else {
    q->first = ptr;
    q->last = ptr;
  }
}
```



Queue_Add

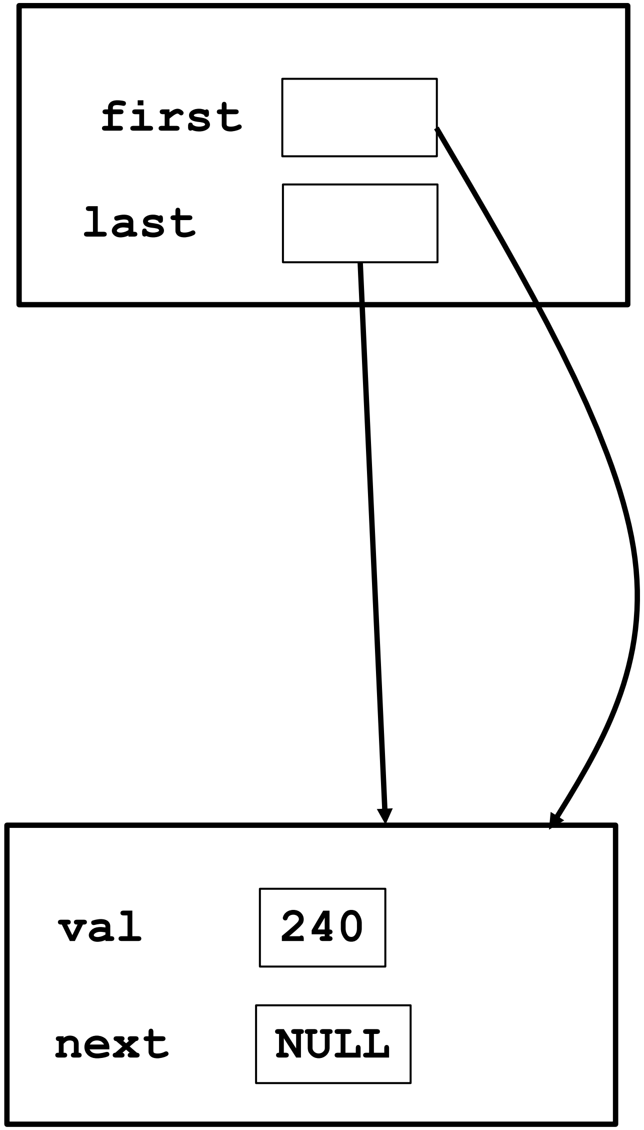
```
void Queue_Add(Queue *q, int val) {
  Queue_Node* ptr;
  ptr = malloc(sizeof(Queue_Node));

  ptr->next = NULL;
  ptr-> val = val;
  if (q->last) {
    q->last->next = ptr;
    q->last = ptr;
  } else {
    q->first = ptr;
    q->last = ptr;
  }
}
```



Queue_Add

```
void Queue_Add(Queue *q, int val) {
  Queue_Node* ptr;
  ptr = malloc(sizeof(Queue_Node));

  ptr->next = NULL;
  ptr-> val = val;
  if (q->last) {
    q->last->next = ptr;
    q->last = ptr;
  } else {
    q->first = ptr;
    q->last = ptr;
  }
}
```

Since node is dynamically allocated, it persists after the function returns



Aside: Casting

- In older implementations of the C language, malloc returned a (char*) instead of a (void*) and you would have to employ casting to convert the returned value to the appropriate type
  - double *ptr = (double*) malloc (sizeof(double) * 10);

- Casting also used for casting between non-pointer types.
  - Needed when casting from larger data representation to smaller ones.
    - E.g. casting to convert from double -> float or long -> short

Function Pointers

- Based on what you know about assembly, what is a function name, really? They are just labels corresponding to an address
  - 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?

Could also just do `name(arg1, … argN)`

- 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

```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

Function Pointer Example

- C allows you to omit & on a function name (like arrays) and omit * when calling pointed-to function

```
#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