

# The Heap and Structs

## CSE 333 Autumn 2018

**Instructor:** Hal Perkins

**Teaching Assistants:**

Tarkan Al-Kazily

Renshu Gu

Travis McGaha

Harshita Neti

Thai Pham

Forrest Timour

Soumya Vasisht

Yifan Xu

# Administrivia

- ❖ Exercise 3 out today and due Friday morning
- ❖ hw1 due a week from tomorrow!
  - You **may not** modify interfaces ( .h files)
  - But **do** read the interfaces while you're writing code
  - Suggestion: look at `example_program_{ll|ht}.c` for typical usage of lists and hash tables
- ❖ Remember: the only supported systems for the class are the Allen School Linux machines (workstations, attus, home VM). You should be working on those systems and the projects you build **must** work there.
  - We do not have the cycles to try to support other Unix-like things or chase bugs due to configuration or software differences (including file transfers to/from Windows systems)

# Administrivia

- ❖ We *highly* recommend doing the extra exercises that are at the end of each lecture
  - Also, Google for “C pointer exercises” and do as many as you can get your hands on
  - You MUST master pointers quickly, or you’ll have trouble the rest of the course (including hw1)

# Lecture Outline

## ❖ Heap-allocated Memory

- `malloc()` and `free()`
- Memory leaks

## ❖ `structs` and `typedef`

# 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

- ❖ 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 0;  
}
```



# 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 for shaking out bugs
- 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()

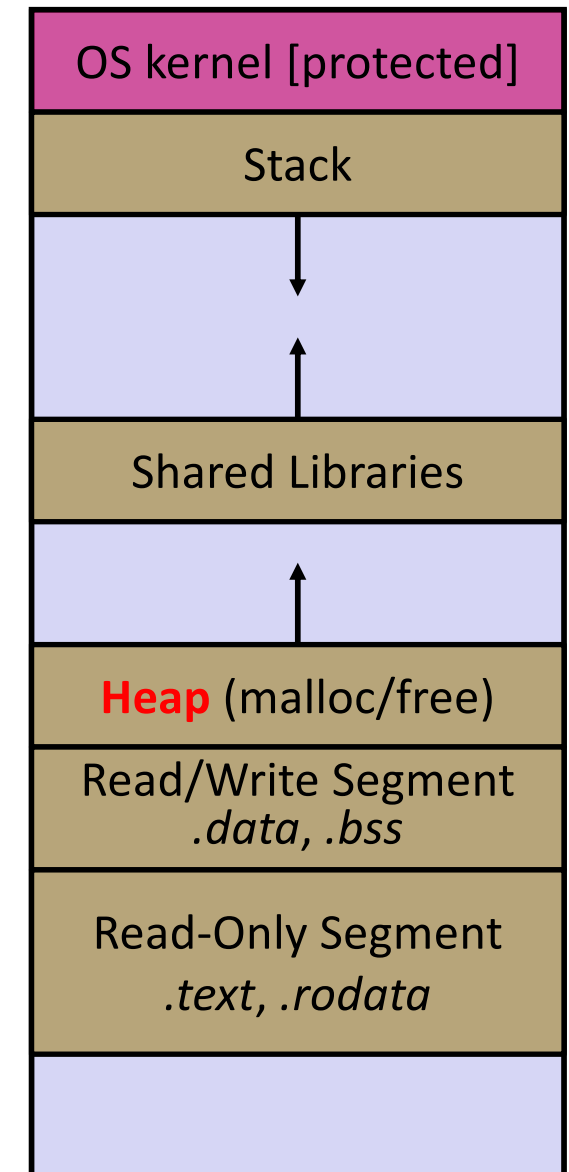
- ❖ Usage: `free(pointer);`
- ❖ Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by `malloc` or `calloc`)
  - Freed memory becomes eligible for future allocation
  - Pointer is unaffected by call 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 unused memory that is used for 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!

0xFF...FF



0x00...00

# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

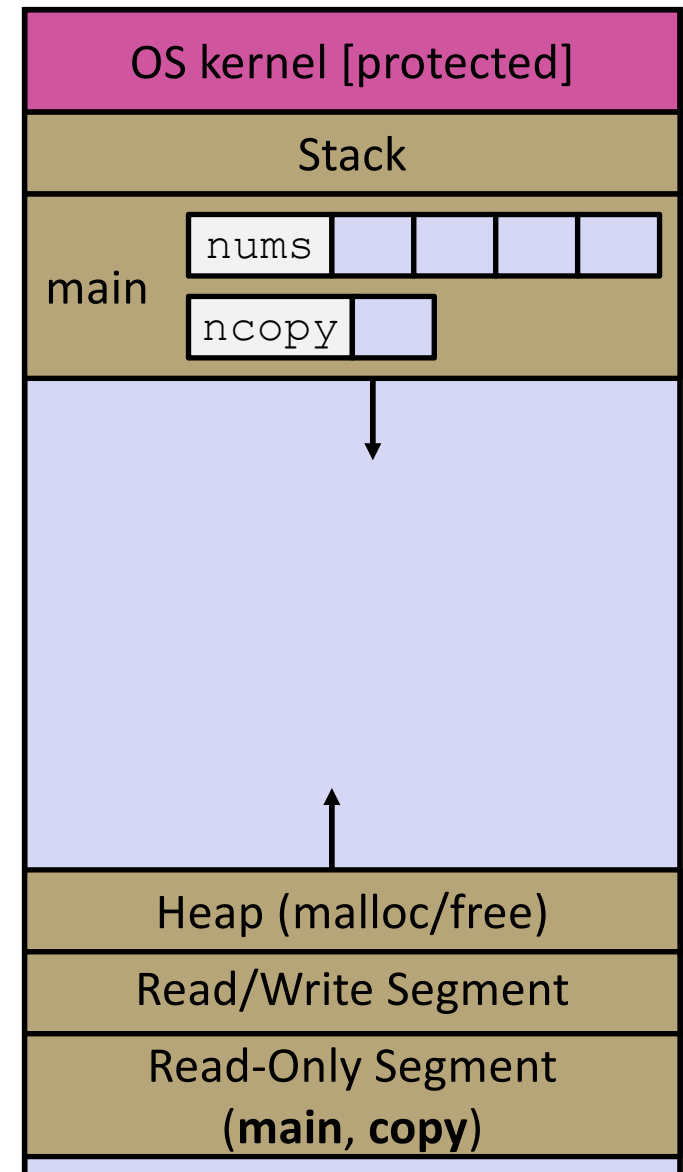
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

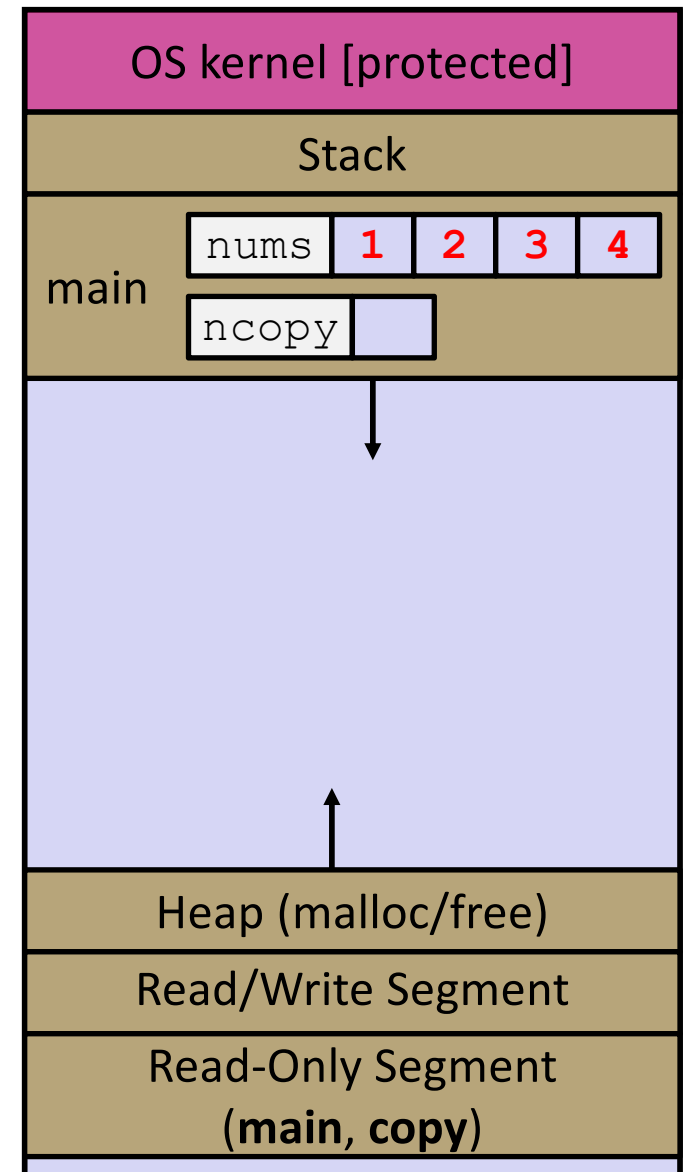
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

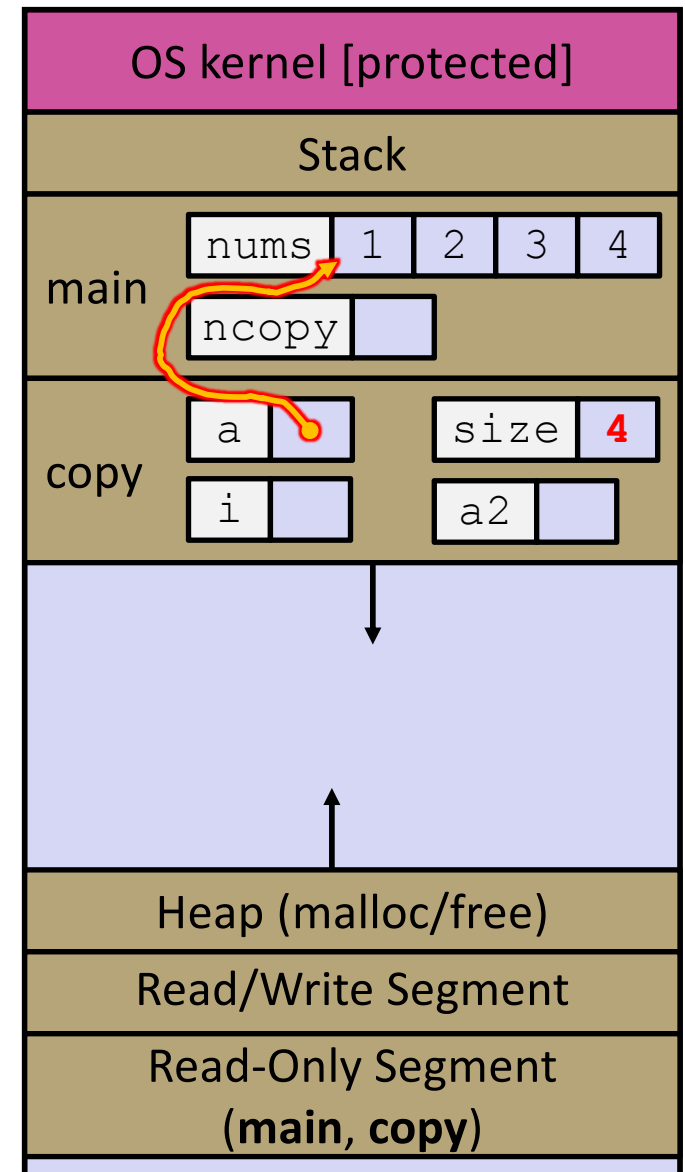
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

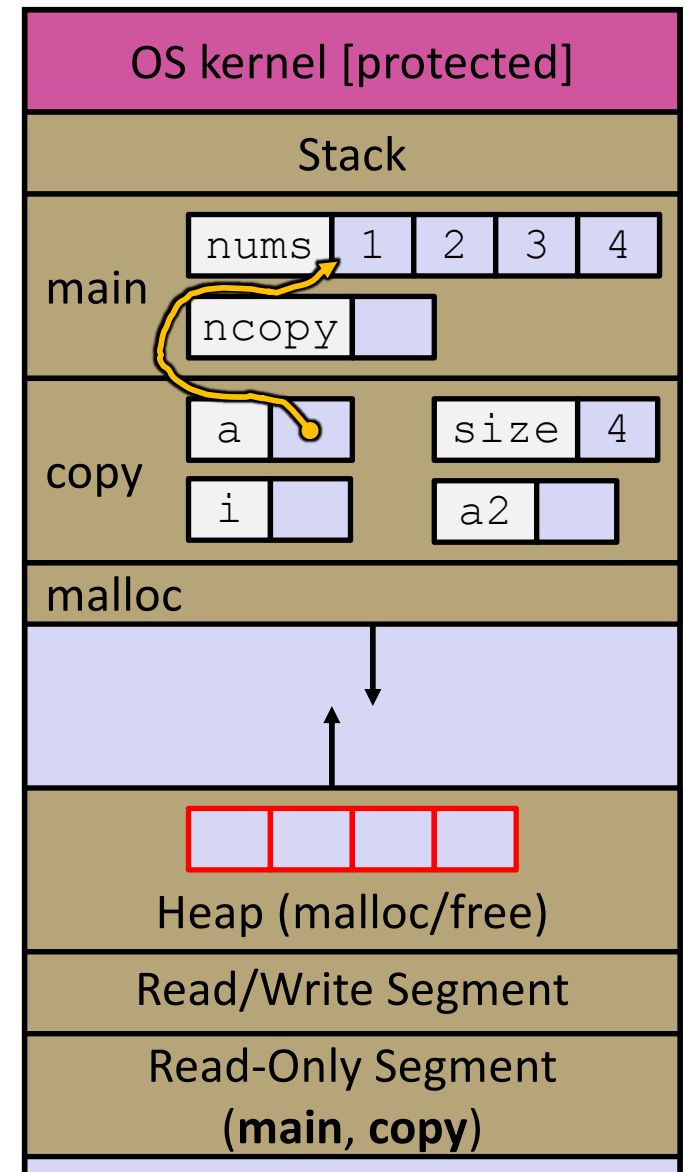
int* copy(int a[], int size) {
    int i, *a2;

    → 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

arraycopy.c

```
#include <stdlib.h>

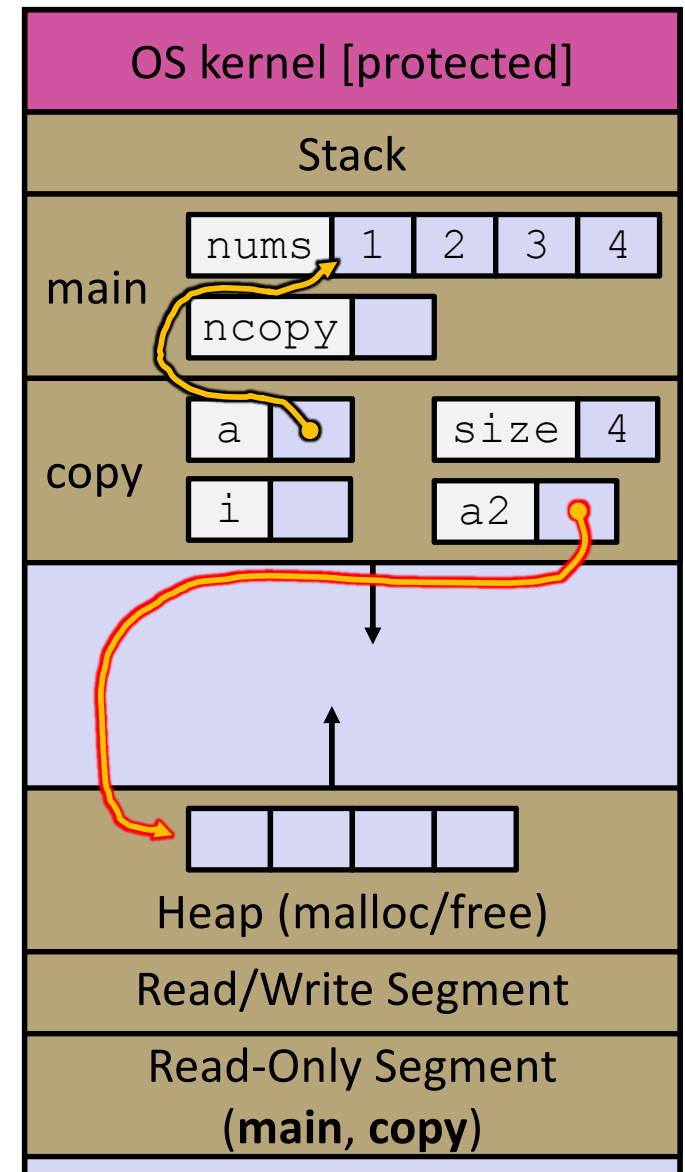
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

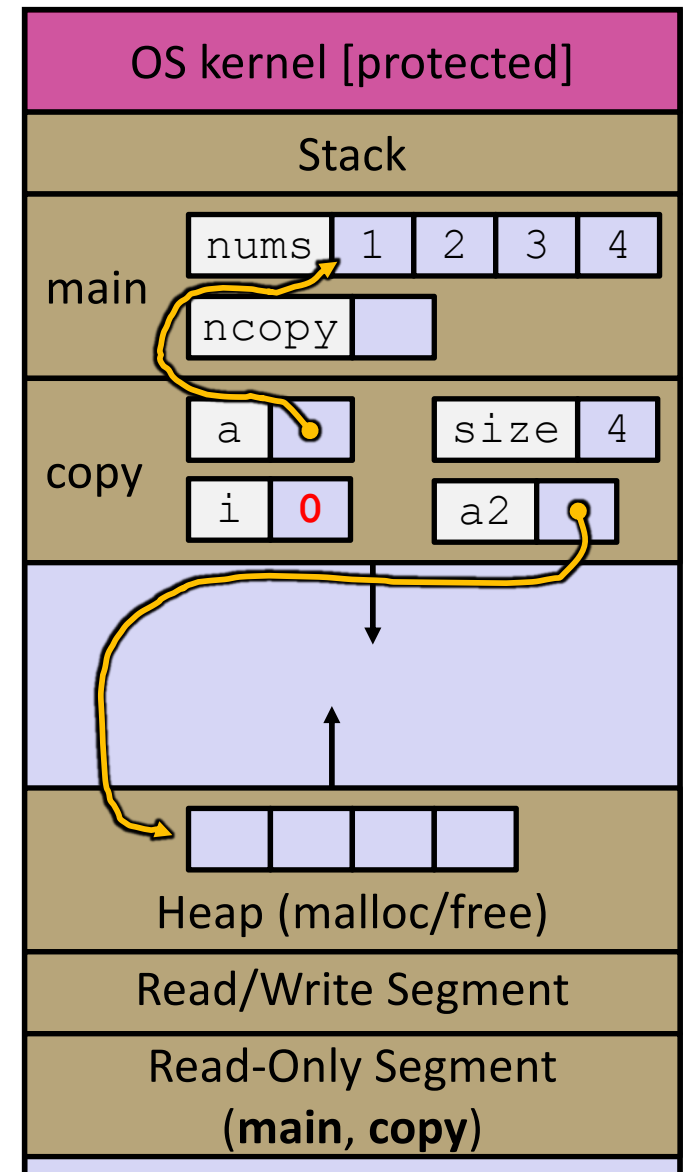
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

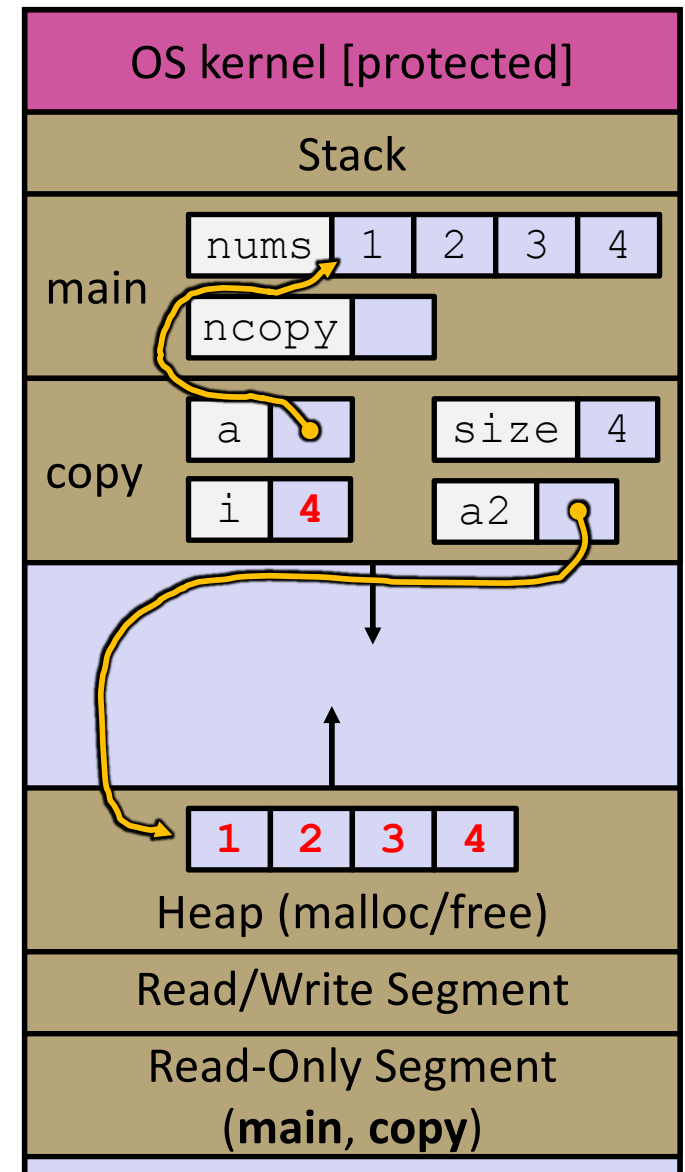
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

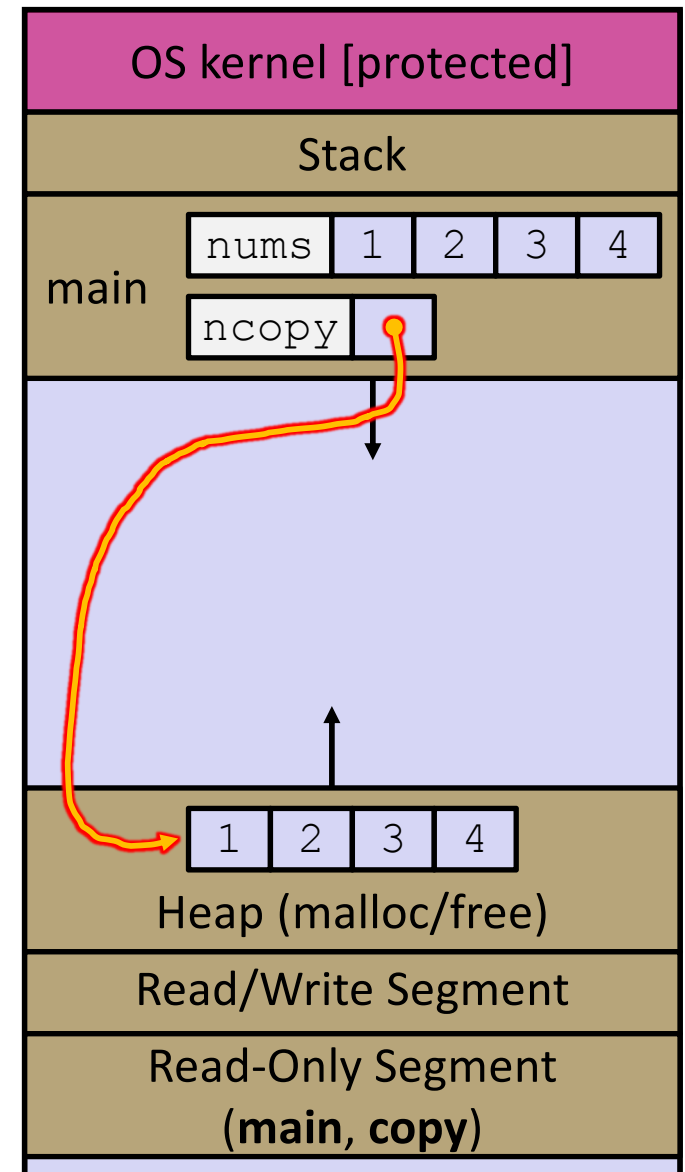
int* copy(int a[], int size) {
    int i, *a2;

    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

## arraycopy.c

```
#include <stdlib.h>

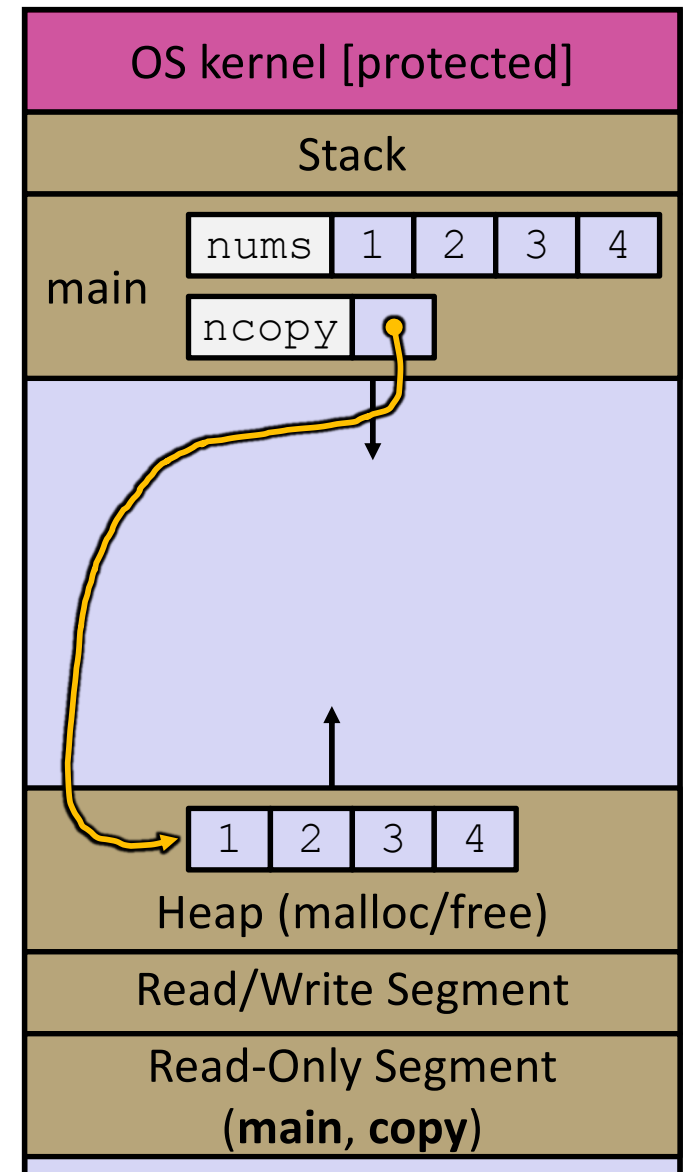
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

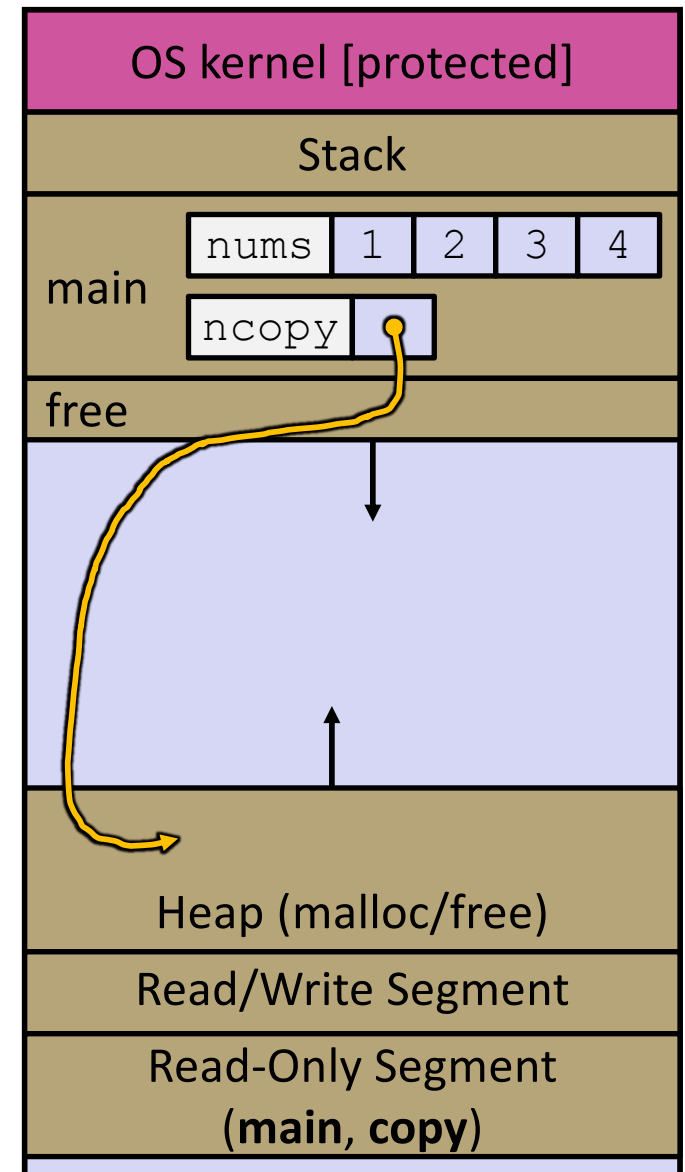
int* copy(int a[], int size) {
    int i, *a2;

    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

arraycopy.c

```
#include <stdlib.h>

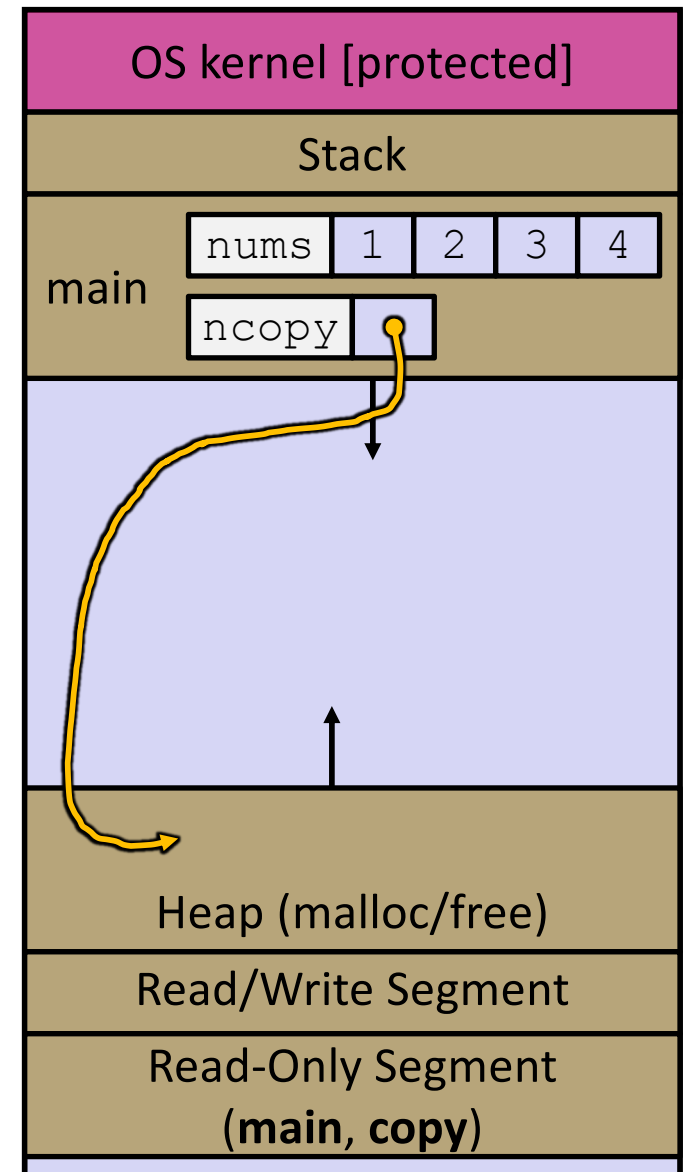
int* copy(int a[], int size) {
    int i, *a2;

    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;
}
```



# Exercise

❖ Which line below is first *guaranteed* to cause an error?

- A. Line 1
- B. Line 4
- C. Line 6
- D. Line 7
- E. Something else

What else is wrong here?

```
#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

- ❖ There are all sorts of ways to corrupt memory in 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;    // assign past the end of an array
    b[0] += 2;    // assume malloc zeros out memory
    c = b+3;     // mess up your pointer arithmetic
    free(&(a[0])); // free something not malloc'ed
    free(b);
    free(b);     // double-free the same block
    b[0] = 5;    // use a freed pointer

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

memcorrupt.c

# Memory Leak

- ❖ A **memory leak** occurs when code fails to deallocate dynamically-allocated memory that is no longer used
  - *e.g.* forget to **free** malloc-ed block, lose/change pointer to malloc-ed block
- ❖ What happens: program's VM footprint will keep growing
  - This might be OK for *short-lived* program, since all memory is deallocated when program ends
  - Usually has bad repercussions for *long-lived* programs
    - Might slow down over time (*e.g.* lead to VM thrashing)
    - Might exhaust all available memory and crash
    - Other programs might get starved of memory

# Lecture Outline

- ❖ Heap-allocated Memory
  - `malloc()` and `free()`
  - Memory leaks
- ❖ **structs and typedef**

# 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
  - Act similarly to primitive variables

- ❖ Generic declaration:

```
struct tagname {  
    type1 name1;  
    ...  
    typeN nameN;  
};
```

```
// the following defines a new  
// structured datatype called  
// a "struct Point"  
struct Point {  
    float x, y;  
};  
  
// declare and initialize a  
// struct Point variable  
struct Point origin = {0.0, 0.0};
```

# Using structs

- ❖ 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

```
struct Point {  
    float x, y;  
};  
  
int main(int argc, char** argv) {  
    struct Point p1 = {0.0, 0.0}; // p1 is stack allocated  
    struct Point* p1_ptr = &p1;  
  
    p1.x = 1.0;  
    p1_ptr->y = 2.0; // equivalent to (*p1_ptr).y = 2.0;  
    return 0;  
}
```

simplestruct.c

# Copy by Assignment

- ❖ You can assign the value of a struct from a struct of the same type – *this copies the entire contents!*

```
#include <stdio.h>

struct Point {
    float x, y;
};

int main(int argc, char** argv) {
    struct Point p1 = {0.0, 2.0};
    struct Point p2 = {4.0, 6.0};

    printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
    p2 = p1;
    printf("p1: {%f,%f}  p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);
    return 0;
}
```

structassign.c

# 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
  - Be careful with pointers – `*` before `name` is part of `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 { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
    superlong x;
    superlong y;
} Point, *PointPtr; // similar syntax to "int n, *p;"

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

# Dynamically-allocated Structs

- ❖ You can **malloc** and **free** structs, just like other data type
  - **sizeof** is particularly helpful here

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;    // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
    Complex* retval = (Complex*) malloc(sizeof(Complex));
    if (retval != NULL) {
        retval->real = real;
        retval->imag = imag;
    }
    return retval;
}
```



# Structs as Arguments

- ❖ Structs are passed by value, like everything else in C
  - Entire struct is copied – where?
  - To manipulate a struct argument, pass a pointer instead

```
typedef struct point_st {  
    int x, y;  
} Point, *PointPtr;  
  
void DoubleXBroken(Point p)    { p.x *= 2; }  
  
void DoubleXWorks(PointPtr p) { p->x *= 2; }  
  
int main(int argc, char** argv) {  
    Point a = {1,1};  
    DoubleXBroken(a);  
    printf("(%d,%d)\n", a.x, a.y);    // prints: (  ,  )  
    DoubleXWorks(&a);  
    printf("(%d,%d)\n", a.x, a.y);    // prints: (  ,  )  
    return 0;  
}
```

# Returning Structs

- ❖ Exact method of return depends on calling conventions
  - Often in `%rax` and `%rdx` for small structs
  - Often returned in memory for larger structs

```
// a complex number is a + bi
typedef struct complex_st {
    double real;      // real component
    double imag;      // imaginary component
} Complex, *ComplexPtr;

Complex MultiplyComplex(Complex x, Complex y) {
    Complex retval;

    retval.real = (x.real * y.real) - (x.imag * y.imag);
    retval.imag = (x.imag * y.real) - (x.real * y.imag);
    return retval; // returns a copy of retval
}
```

complexstruct.c

# Pass Copy of Struct or Pointer?

- ❖ Value passed: passing a pointer is cheaper and takes less space unless struct is small
- ❖ Field access: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize
- ❖ For small structs (like `struct complex_st`), passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers

# Extra Exercise #1

- ❖ Write a program that defines:
  - A new structured type `Point`
    - Represent it with `floats` for the `x` and `y` coordinates
  - A new structured type `Rectangle`
    - Assume its sides are parallel to the `x`-axis and `y`-axis
    - Represent it with the bottom-left and top-right `Points`
  - A function that computes and returns the area of a `Rectangle`
  - A function that tests whether a `Point` is inside of a `Rectangle`

## Extra Exercise #2

- ❖ Implement `AllocSet()` and `FreeSet()`
  - `AllocSet()` needs to use `malloc` twice: once to allocate a new `ComplexSet` and once to allocate the “points” field inside it
  - `FreeSet()` needs to use `free` twice

```
typedef struct complex_st {
    double real;      // real component
    double imag;      // imaginary component
} Complex;

typedef struct complex_set_st {
    double    num_points_in_set;
    Complex*  points;    // an array of Complex
} ComplexSet;

ComplexSet* AllocSet(Complex c_arr[], int size);
void FreeSet(ComplexSet* set);
```