



[pollev.com/cse333](https://pollev.com/cse333)

**What is your anticipated lecture/section attendance modality (for the time being)?**

- A. In person until the University dictates otherwise**
- B. In person with some hesitation**
- C. Remote, synchronously**
- D. Remote, asynchronously**

# The Heap and Structs

## CSE 333 Winter 2022

**Instructor:** Justin Hsia

**Teaching Assistants:**

Aakash Srazali

Assaf Vayner

Brenden Page

Cleo Chen

Dan Constantinescu

Dylan Hartono

Elizabeth Haker

Jacob Christy

Julia Wang

Kenzie Mihardja

Kyrie Dowling

Mengqi Chen

Mitchell Levy

Timmy Yang

# Relevant Course Information (1/2)

- ❖ Exercise grades
  - We will be giving “autograder correction” points
  - Regrade requests: open 24 hr after, close 72 hr after release
- ❖ hw0 due tonight *before* 11:59 pm (and 0 seconds)
  - Git: add/commit/push, then tag with `hw0-final`, then push tag
    - Then clone your repo somewhere totally different and do `git checkout hw0-final` and verify that all is well
- ❖ hw1 due Thursday, 1/20 @ 11:59 pm
  - You **may not** modify interfaces ( `.h` files), but **do** read the interfaces while you’re implementing them (!)
  - Record bugs in `bugjournal.md`
  - Suggestion: pace yourself and make steady progress

# Relevant Course Information (2/2)

- ❖ Gitlab repo usage
  - **Commit things regularly** (not all at once at the end)
    - Newly completed units of work / milestones / project parts
    - Don't push .o and executable files or other build products
  - Provides backups – can retrieve old versions of files 😊
  - Useful for sharing with staff members and partner

# Lecture Outline

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

# Why Dynamic Allocation?

- ❖ Situations where static and automatic allocation aren't sufficient:
  - We need memory that persists across multiple function calls but not for 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
    - e.g., reading file input:

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

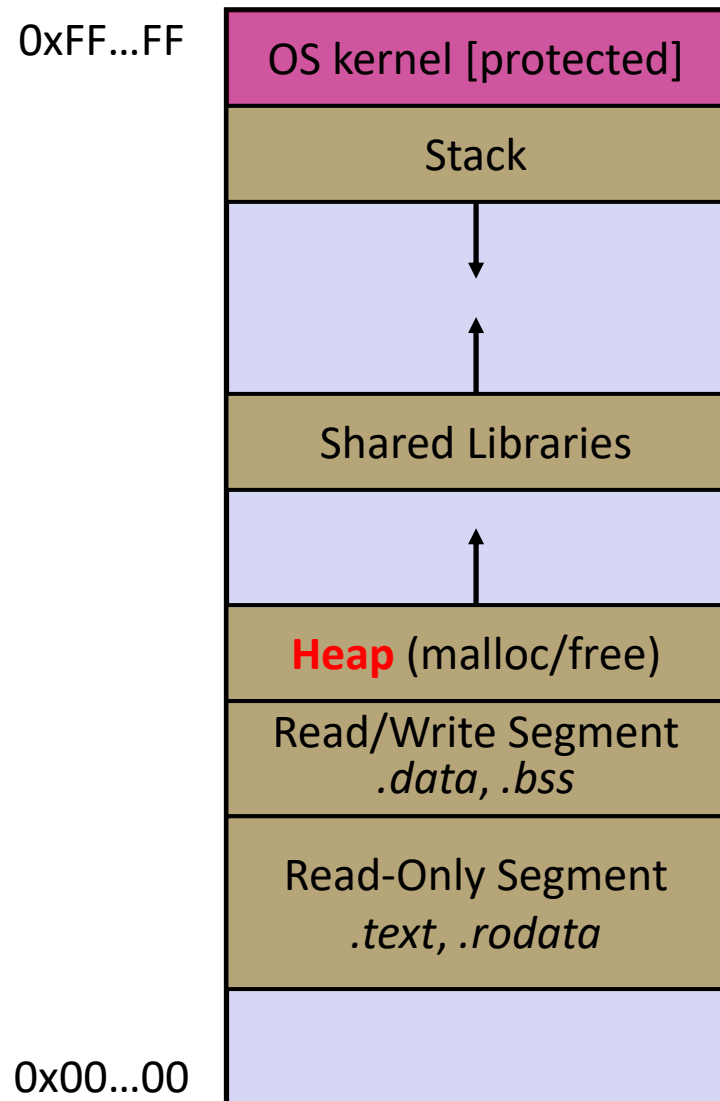
    ReadFileIntoBuffer(filename, buffer);
    return buffer;
}
```

# 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 deallocates it (manual/explicit memory management)
    - A garbage collector collects it (automatic/implicit memory management)
- ❖ C requires you to manually manage memory
  - Gives you more control, but causes headaches

# The Heap (351 Review)

- ❖ 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!



# 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 an uninitialized block of heap memory of at least the requested size
  - Returns a pointer to the first byte of that memory; **returns NULL** if the memory allocation failed!
  - Stylistically, you'll want to (1) use `sizeof` in your argument, (2) cast the return value, and (3) error check the return value

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

- ❖ Also, see **calloc**() and **realloc**()

# 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
  - Freeing `NULL` has no effect
  - The bits stored in the pointer are *not changed* by calling `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
```

# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

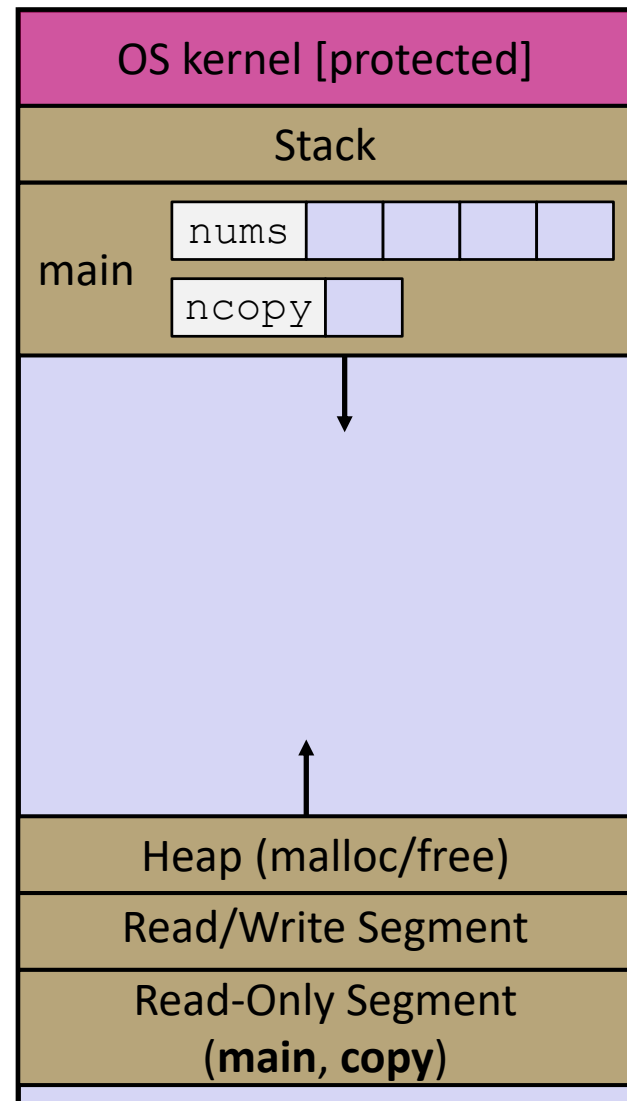
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

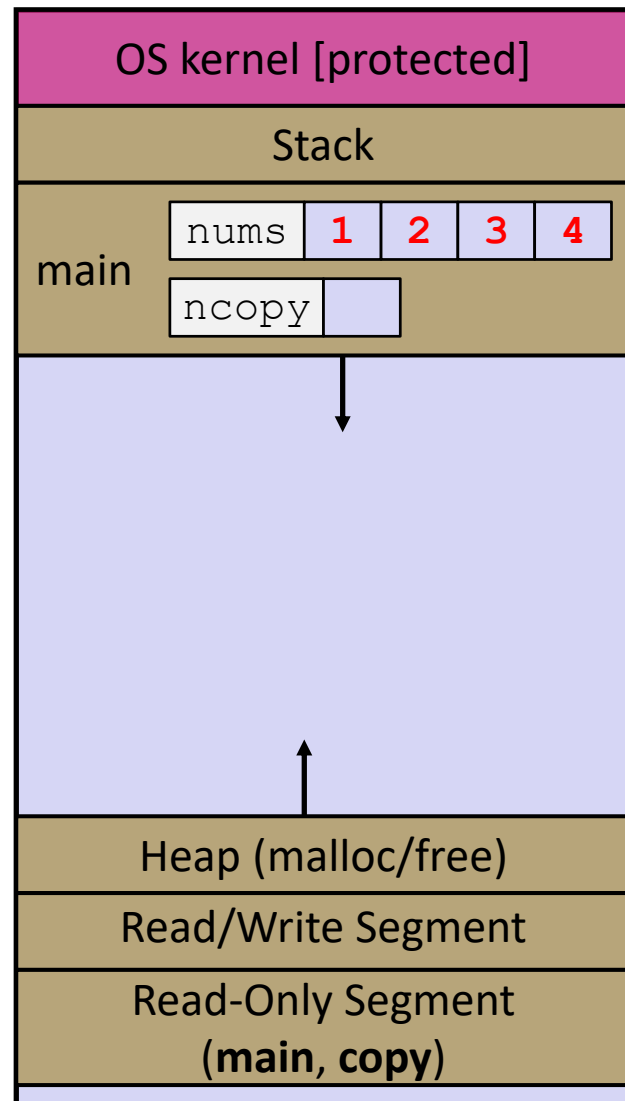
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

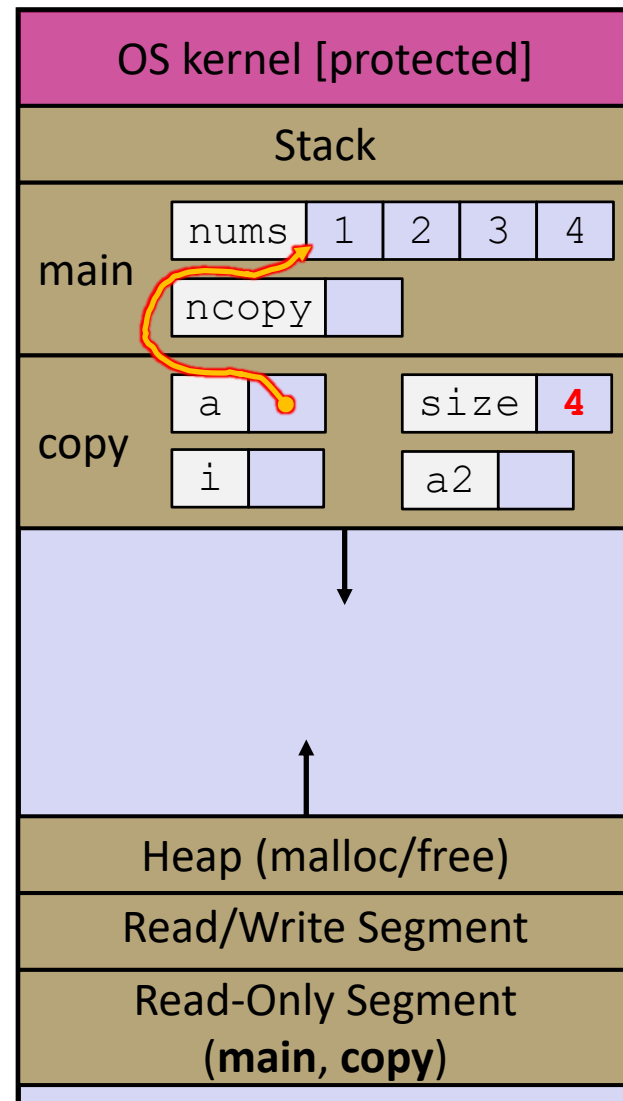
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

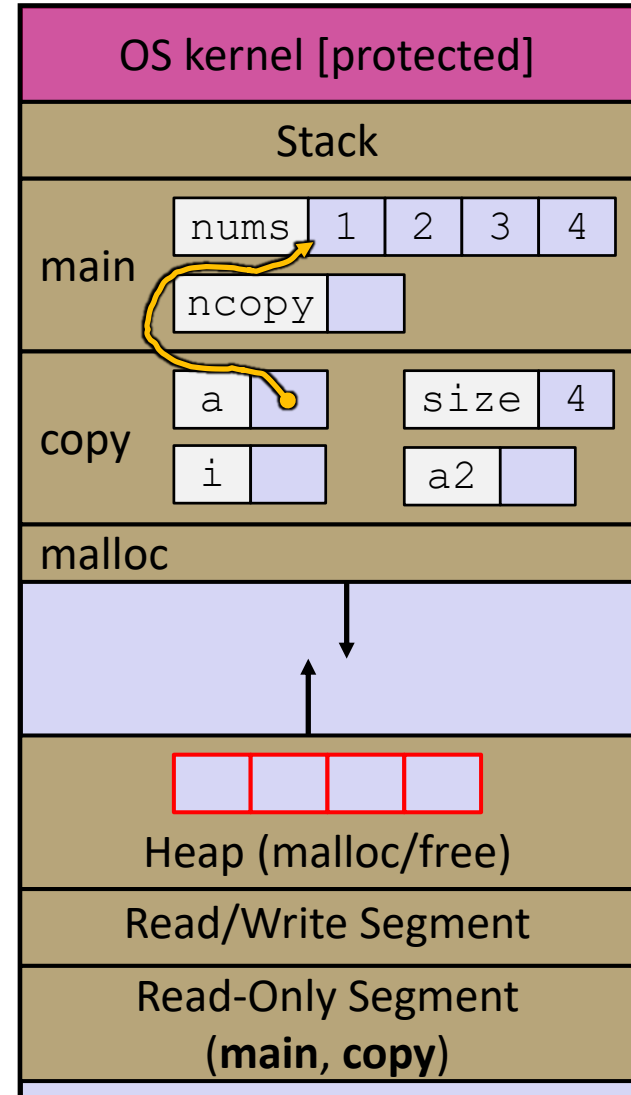
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

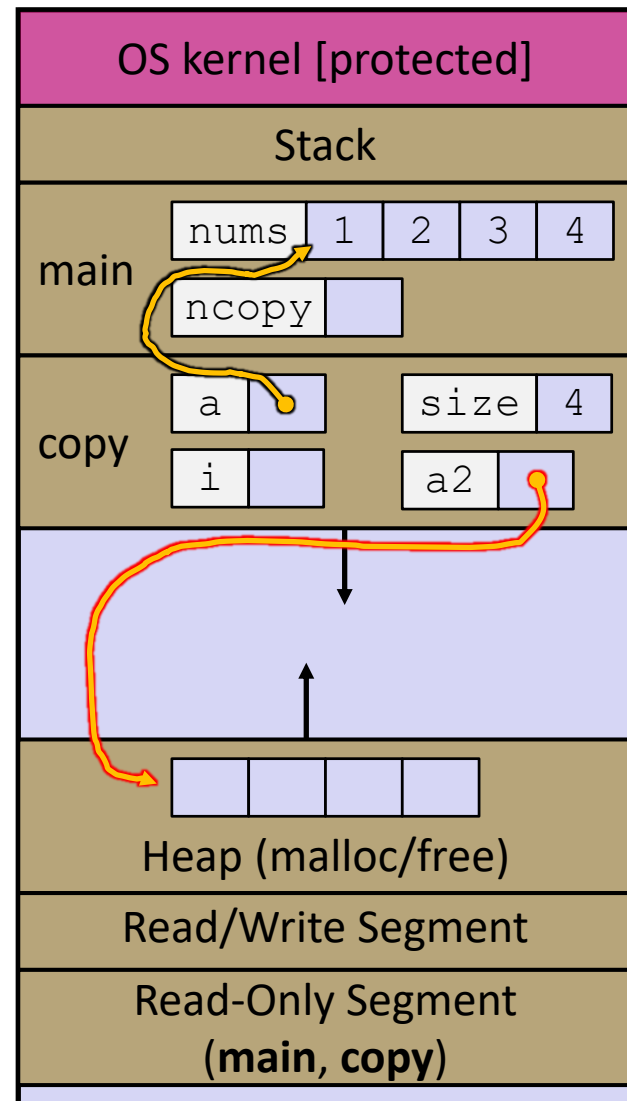
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

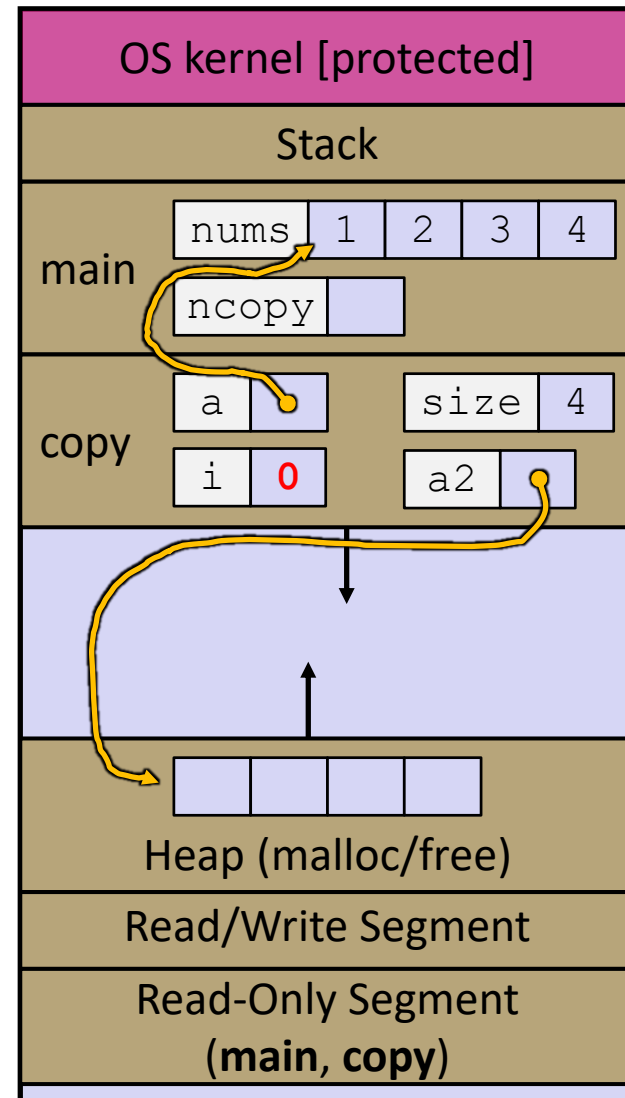
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

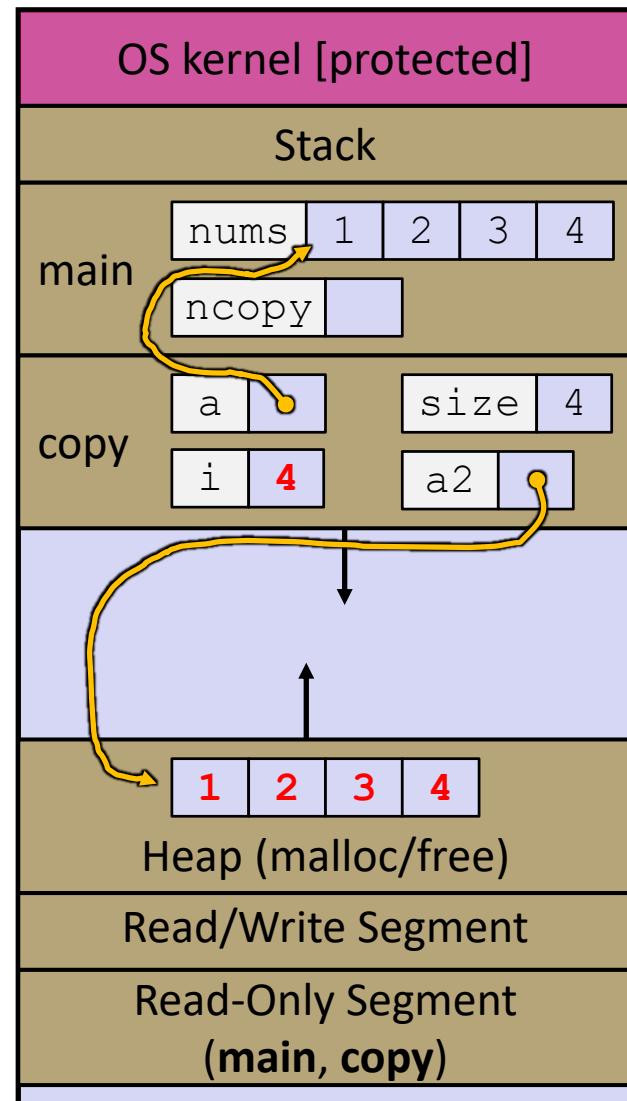
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

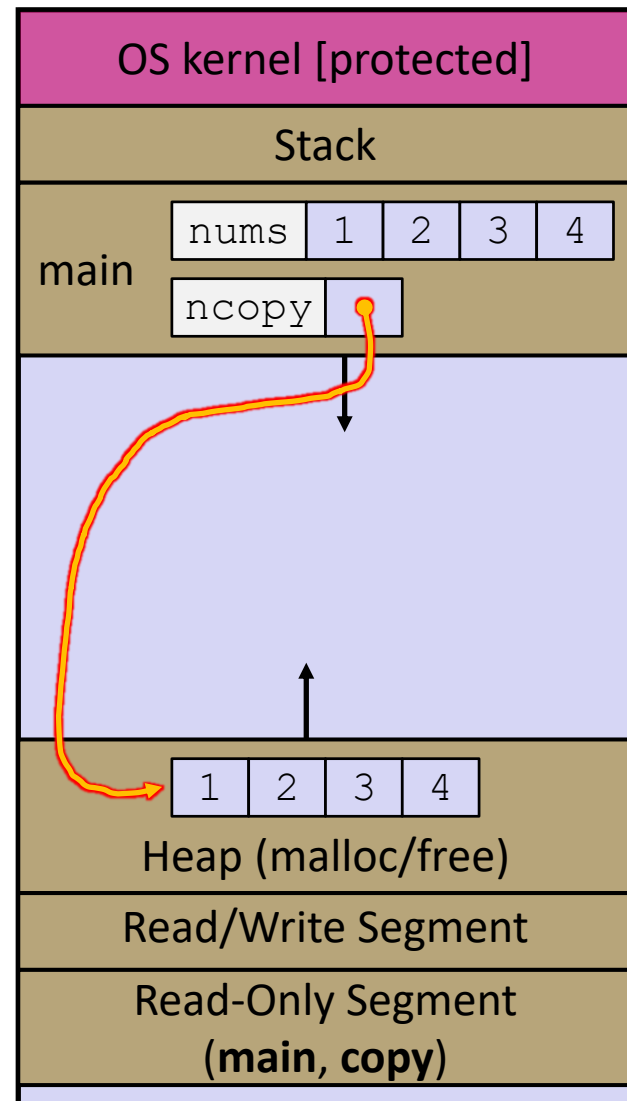
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

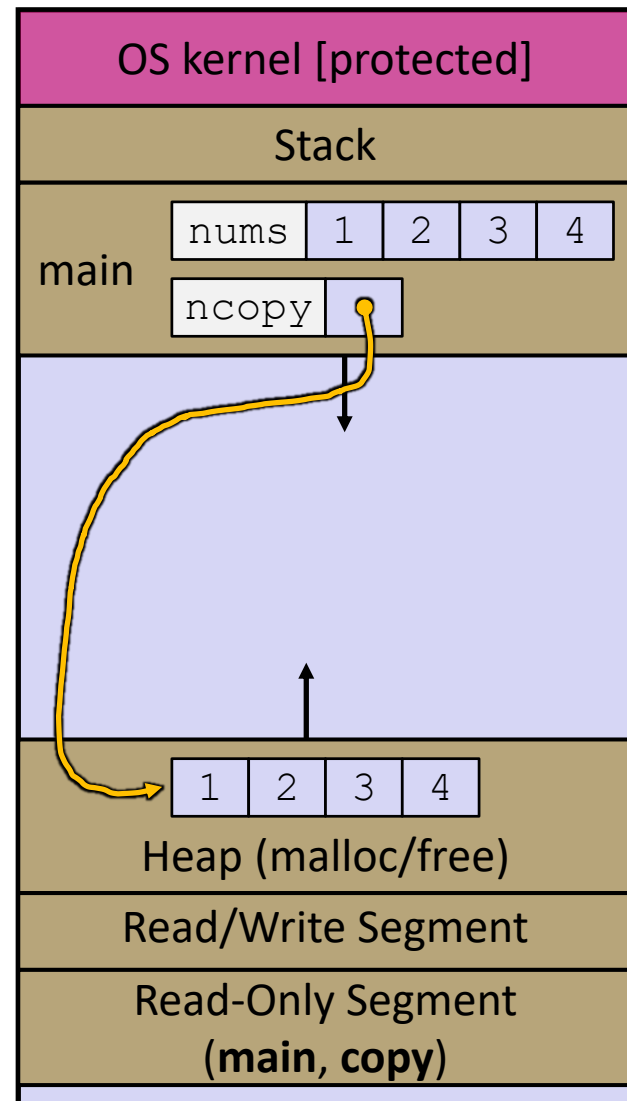
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

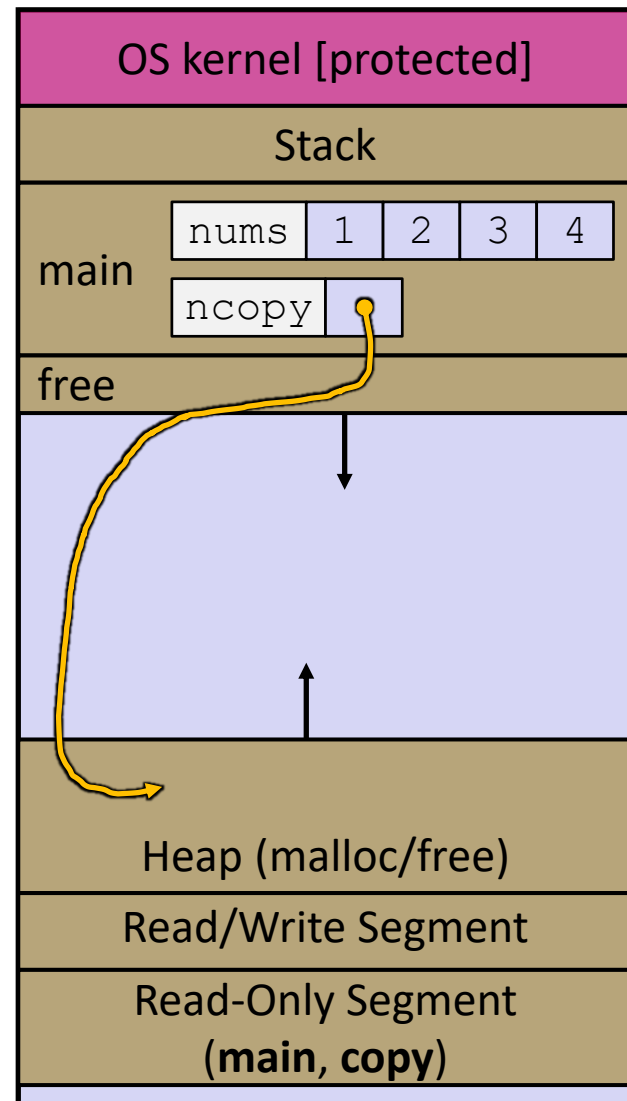
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```



# Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

```
#include <stdlib.h>

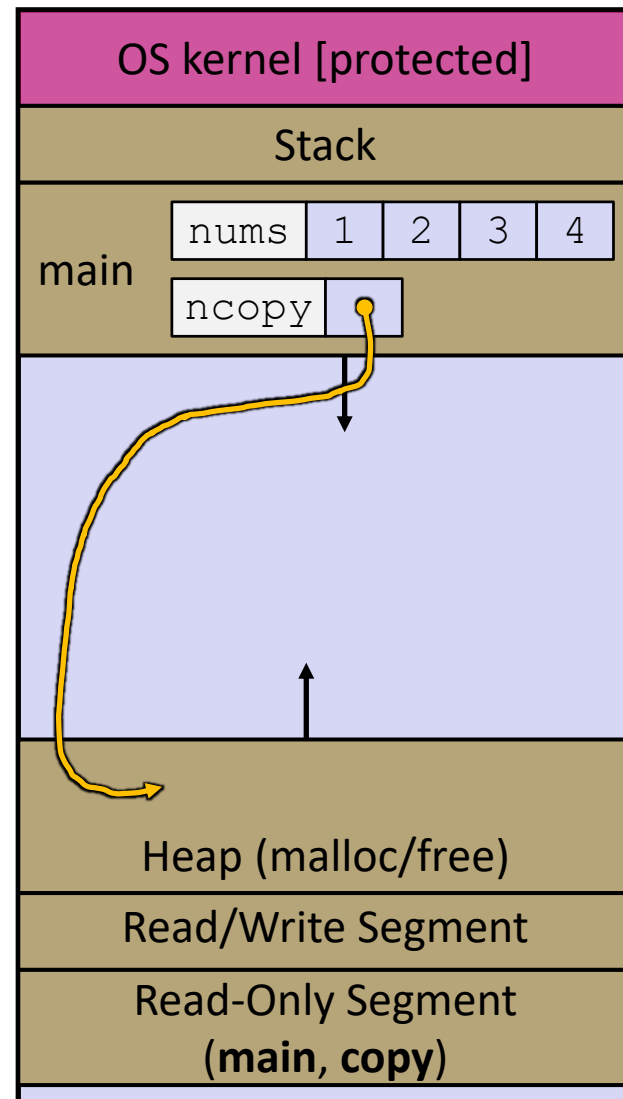
int* copy(int a[], int size) {
    int i, *a2;

    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 EXIT_SUCCESS;
}
```





# Poll Everywhere

[pollev.com/cse333](http://pollev.com/cse333)

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

- A. Line 1
- B. Line 4
- C. Line 6
- D. Line 7
- E. We're lost...

memcorrupt.c

```
#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 EXIT_SUCCESS;
}
```

# Memory Leaks

- ❖ 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
  - Easier said than done; just passing pointers around – who's responsible for freeing?
- ❖ 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 memory and performance repercussions for *long-lived* programs

# Lecture Outline

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

# Structured Data (351 Review)

- ❖ 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
  - Behave 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 (351 Review)

- ❖ 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 EXIT_SUCCESS;
}
```

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!*

```
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 EXIT_SUCCESS;
}
```

structassign.c

# Typedef (351 Review)

- ❖ 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;

Complex* 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 {                                     structarg.c
    int x, y;
} Point;

void DoubleXBroken(Point p)    { p.x *= 2; }

void DoubleXWorks(Point* 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: ( 1, 1 )
    DoubleXWorks(&a);
    printf("( %d, %d) \n", a.x, a.y);    // prints: ( 2, 2 )
    return EXIT_SUCCESS;
}
```

# 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;

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

# Discussion Activity

- ❖ Write out a C snippet that:
  - Defines a struct for a linked list that holds (1) a character pointer and (2) a pointer to an instance of this struct
  - Typedefs the struct as `Node`
- ❖ Assume that all `Node` instances are dynamically-allocated and that we are designing a `Pop(Node* head)` function that removes the first node and returns it, discuss:
  - Should the return type be `Node` or `Node*`?
  - Should we free the popped `Node` or not?

# 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);
```