CSE333 Systems Programming
Structs, Typedefs and the Heap

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski	Barbora Buzkova	Mendel Carroll
Camden Harris	Hannah Hempstead	Rishabh Jain
Irene Lau	Jonathan Nister	Advay Patil
Aviel Wood	Angela Wu	Jennifer Xu

Administrivia!

- Exercise 2 due Friday before 11 AM
- Homework 1 due a week from Thursday
  - You should be well under way now
  - Be sure to read headers *carefully* while implementing
  - Use git add/commit/push regularly to save work
- Section this week will involve debugging!

Lecture Outline

- **structs and typedef**
- Heap-allocated Memory
- Simple Linked List Example

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 **(shallow)** 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;
```

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};
```

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

structarg.c

```
typedef struct point_st {
  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: (  ,  )
  DoubleXWorks(&a);
  printf("(%d,%d)\n", a.x, a.y);   // prints: (  ,  )
  return EXIT_SUCCESS;
}
```

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 stucts (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

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

```c
// 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

Lecture Outline

- structs and typedef
- **Heap-allocated Memory**
  - **`malloc()` and `free()`**
  - **Memory leaks**
- Simple Linked List Example

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;
}
```

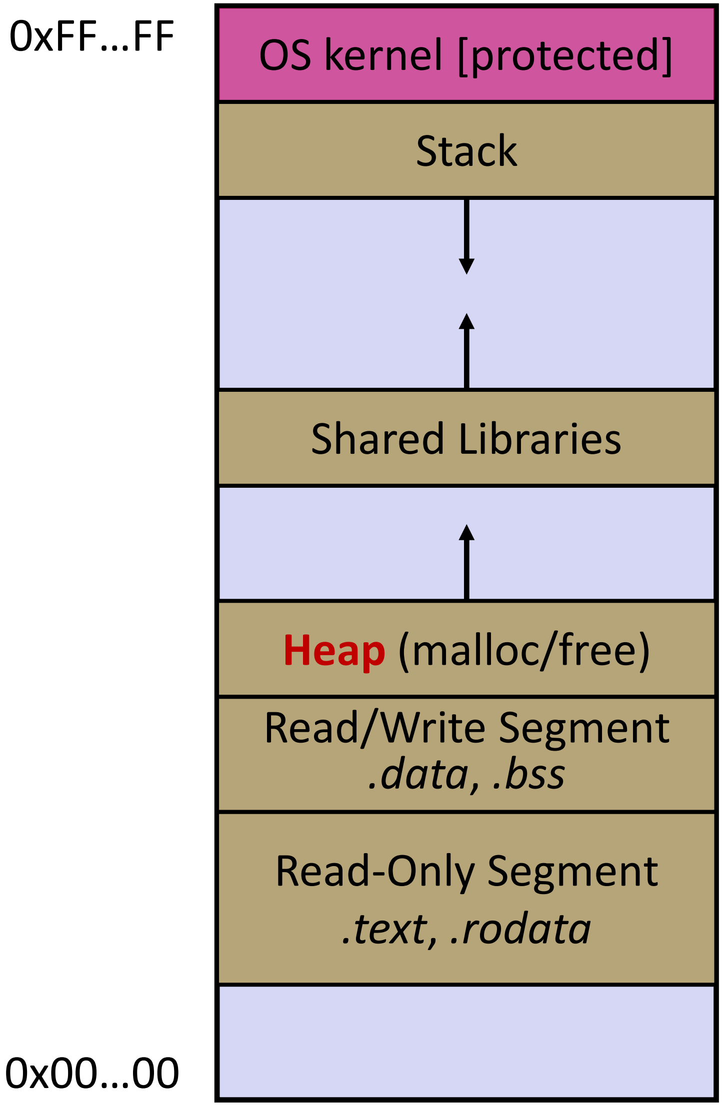
Dynamic Allocation

- 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()

STYLE TIP

- 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 allocation failed!
  - You'll always 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()`

Coco is watching your malloc() style...

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
```

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;
}
```

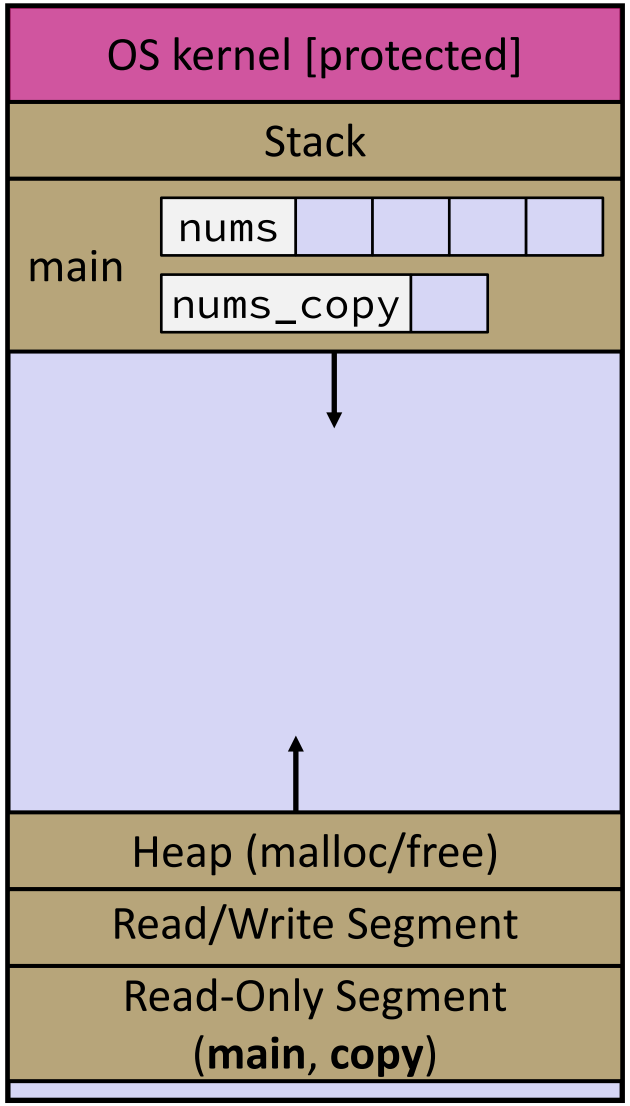
complexstruct.c

Heap and Stack Example (1/11)

Note: Arrow points to *next* instruction.

arraycopy.c

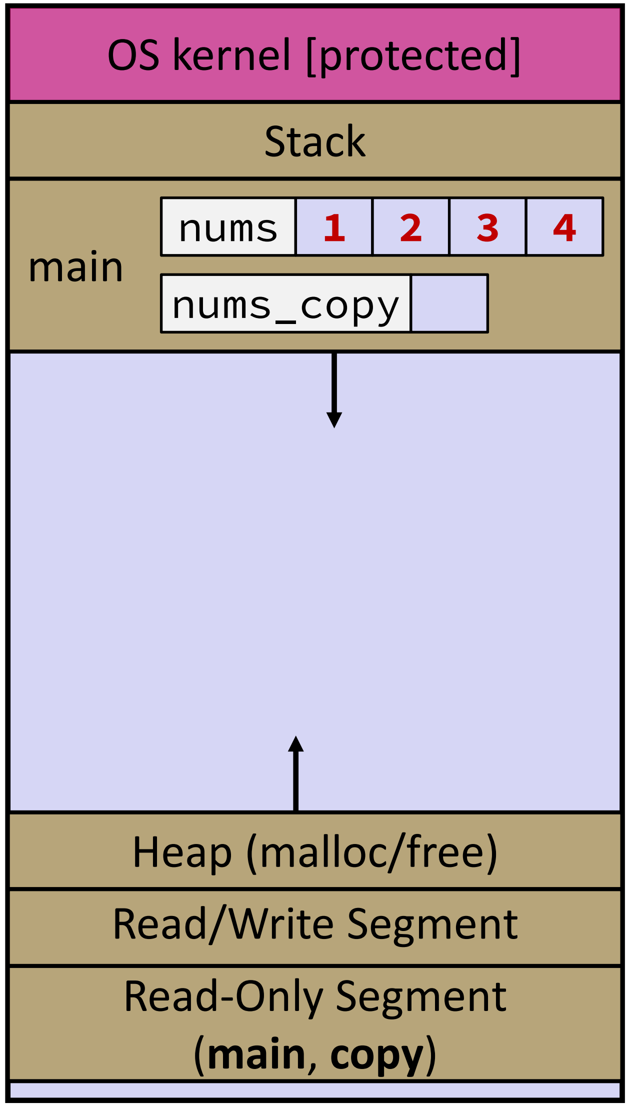
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (2/11)

arraycopy.c

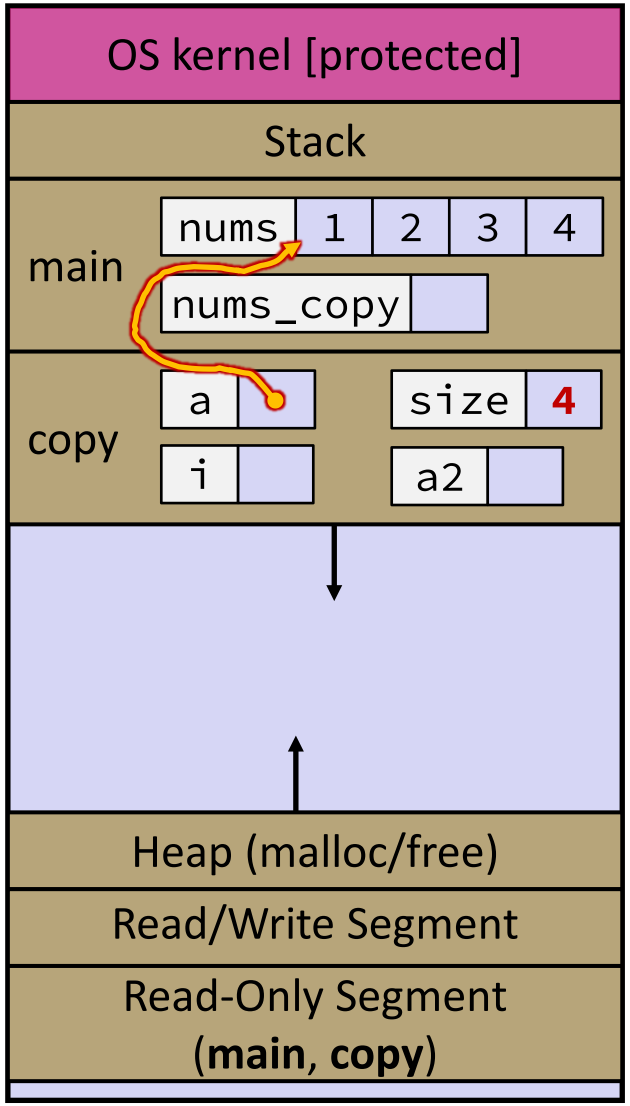
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (3/11)

arraycopy.c

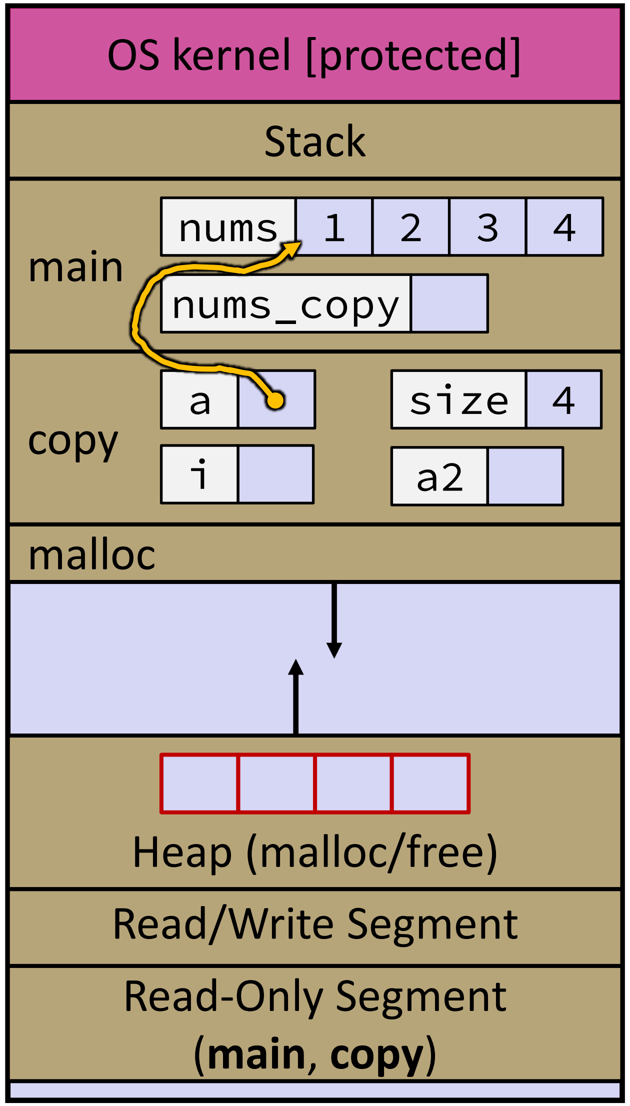
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (4/11)

arraycopy.c

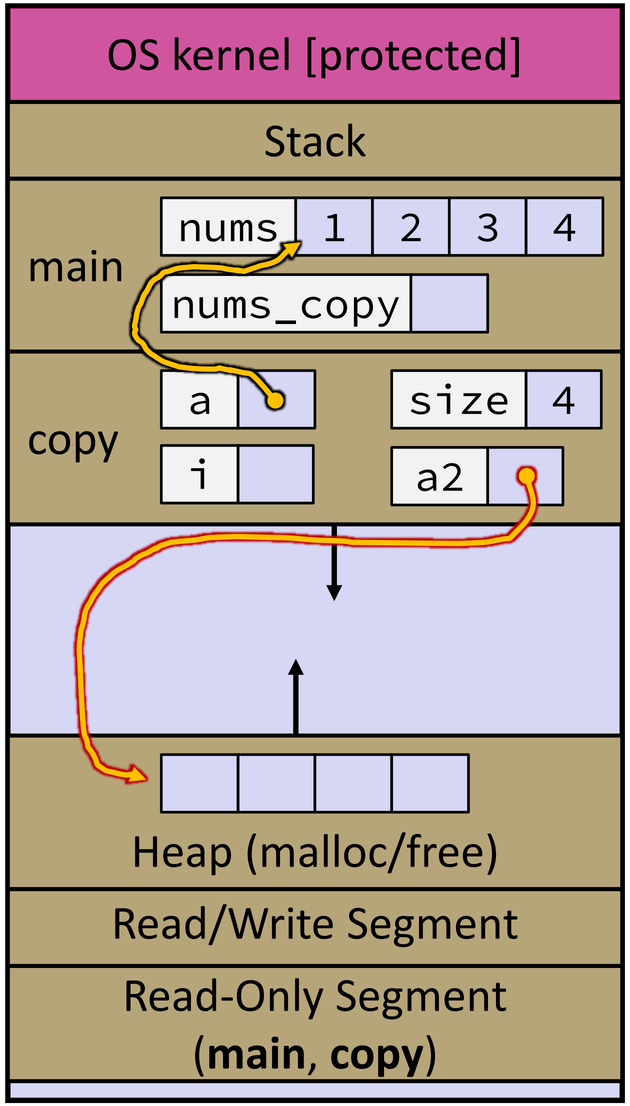
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (5/11)

arraycopy.c

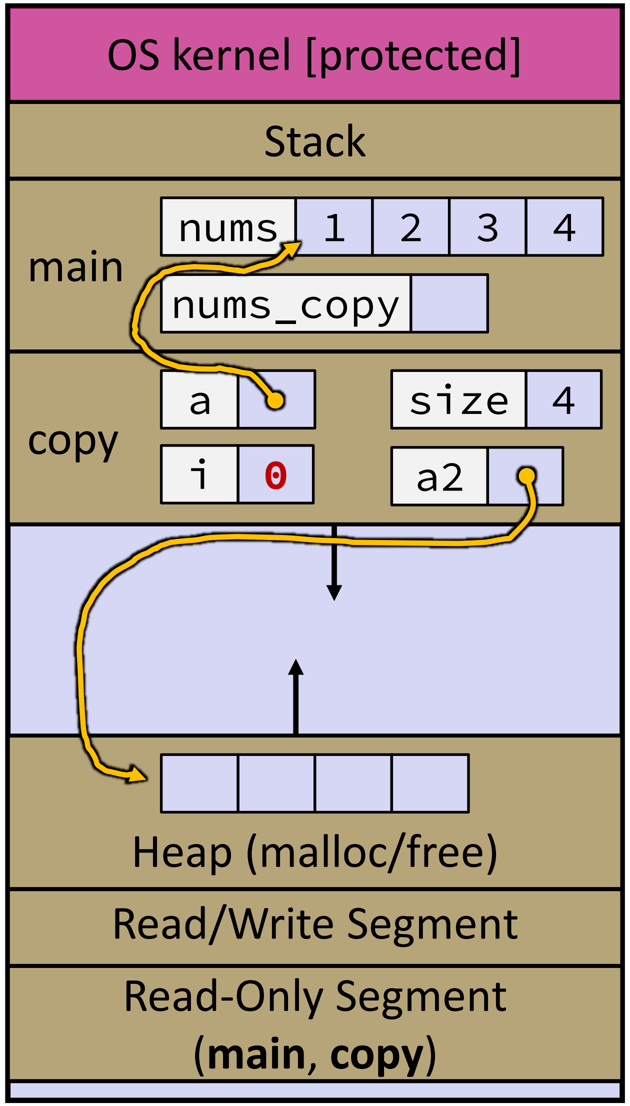
```c
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (6/11)

arraycopy.c

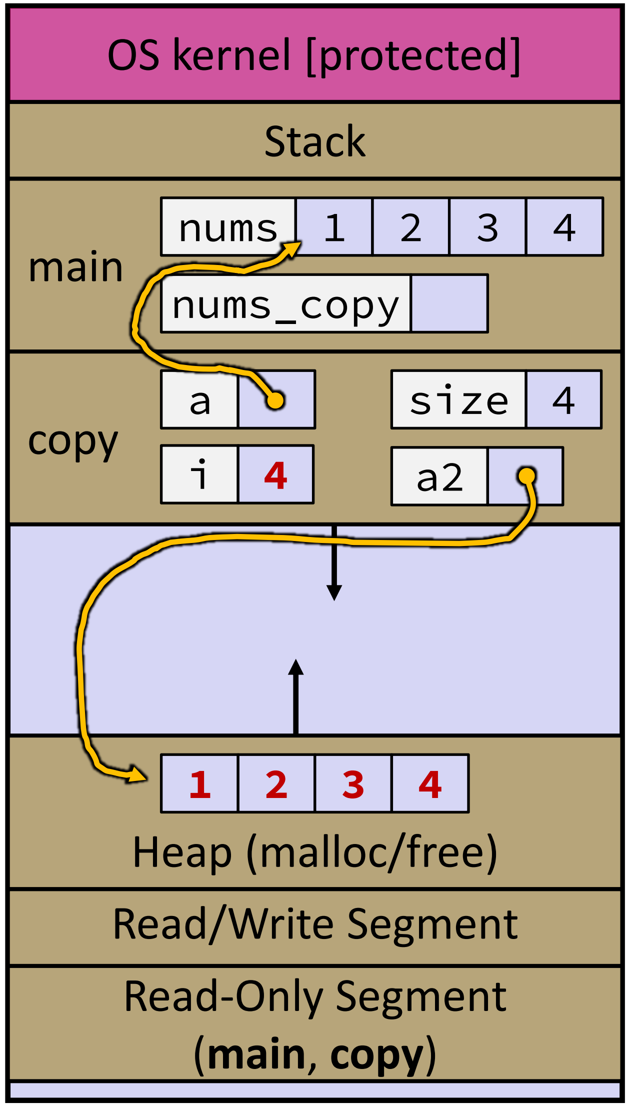
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (7/11)

arraycopy.c

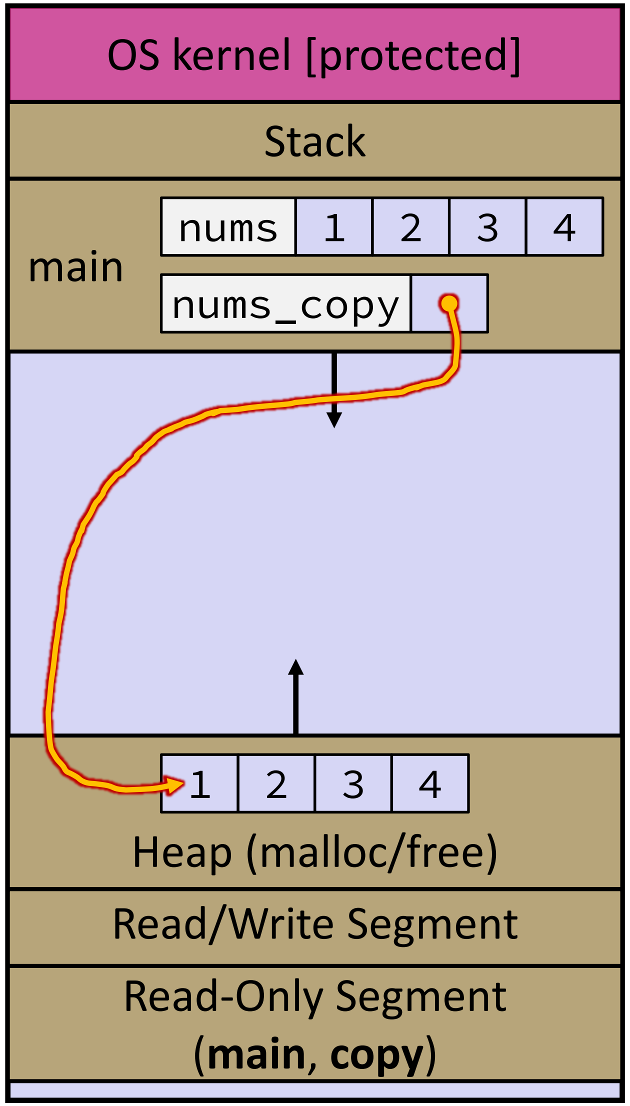
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (8/11)

arraycopy.c

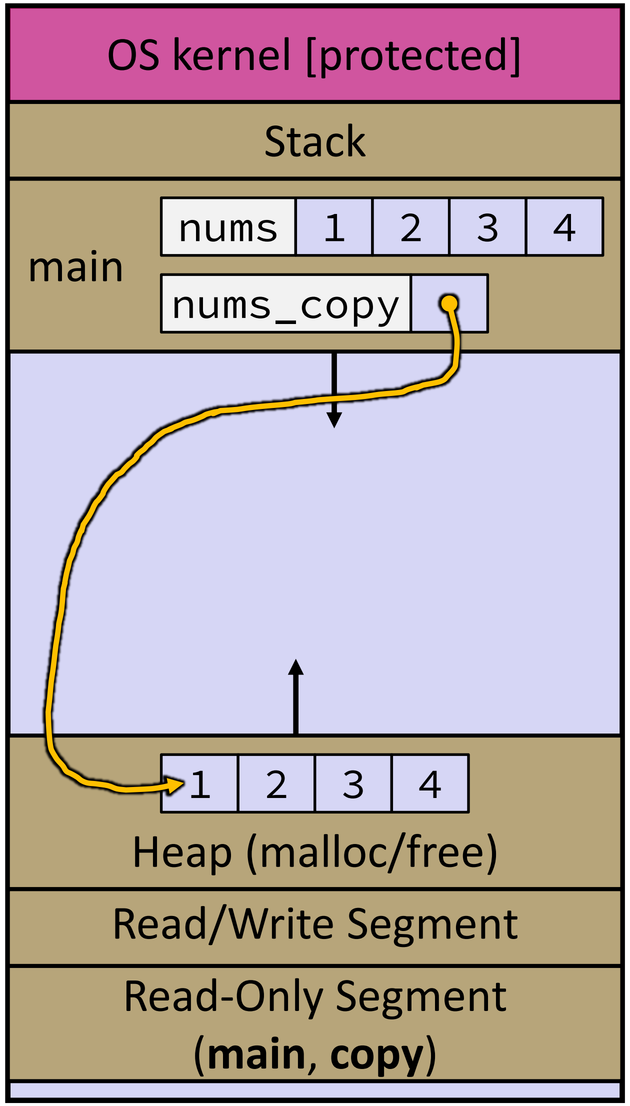
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (9/11)

arraycopy.c

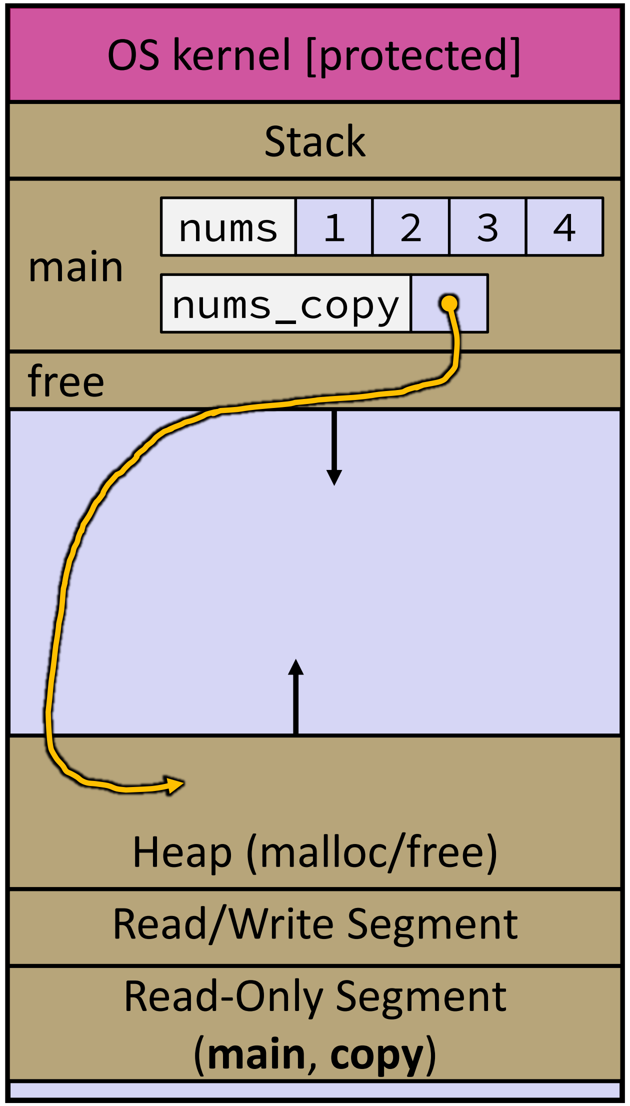
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (10/11)

arraycopy.c

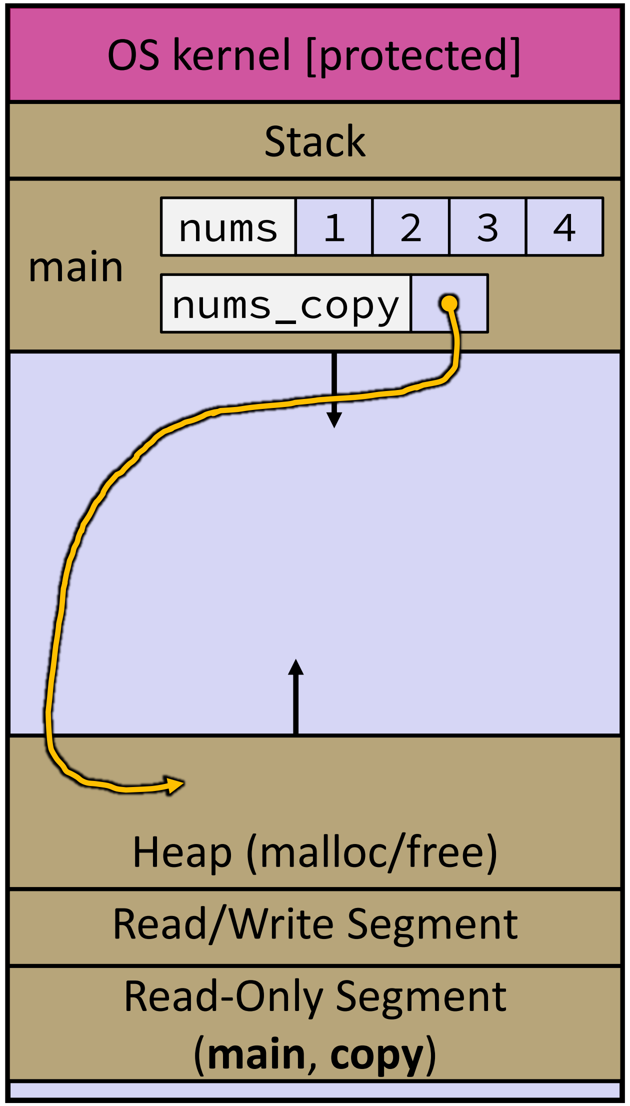
```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  return EXIT_SUCCESS;
}
```



Heap and Stack Example (11/11)

arraycopy.c

```
#include <stdlib.h>

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

  a2 = (int *) 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* nums_copy = Copy(nums, 4);
  // .. do stuff with the array ..
  free(nums_copy);
  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: process' virtual memory 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

Poll Everywhere

pollev.com/naomila

memcorrupt.c

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

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

0   return EXIT_SUCCESS;
  }
```

Take the first digit of your birthday 🎂 and look at the line with that number:

What, if anything, is buggy about it?

Extra Exercise #1

- Implement a Complex number module
  - `complex.c`, `complex.h`
  - Includes a typedef to define a complex number
    - a + bi, where a and b are `doubles`
  - Includes functions to:
    - add, subtract, multiply, and divide complex numbers
  - Implement a test driver in `test_complex.c`
    - Contains `main()`

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

```c
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);
```

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