The Heap and Structs
CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:
Audrey Seo
Deeksha Vatwani
Derek de Leuw
Katie Gilchrist

Administrivia (1)

- HW0 due tonight @ 11:00 PM!

- Exercise 2 was due this morning.
- Yet another exercise, exercise 3, out today, due Wednesday morning

- HW1 out now, due a week from Thursday @ 11:00PM
  - Get started now! Don't want to rush at the last minute
  - Ask lots of questions at office hours and the ed board!

Git

- Proper version control use is a skill you'll use and refine throughout your career
- Start in this class
  - Gitlab is not just for turning in like gradescope
  - It's for tracking your work as you're doing it.
- Make small commits *regularly*, and label them informatively
  - Describe **what** the change was
  - When possible, describe **why** you made that change
  - Committing regularly will save you later when something breaks

Example Commits for Exercise 2

```
commit 5a6118dd96aa901952ac0e1e5ebe7341e8e8e85c
Author: Alex Sanchez-Stern [redacted]>
Date:   Sun Jun 29 11:09:57 2025 -0700

    Add starter main from assignment webpage

diff --git a/ex2.c b/ex2.c
new file mode 100644
index 0000000..ab2562d
--- /dev/null
+++ b/ex2.c
@@ -0,0 +1,23 @@
+int main(int argc, char **argv) {
+  char     char_val = '0';
+  int32_t  int_val = 1;
+  float    float_val = 1.0;
+  double   double_val  = 1.0;
+
+  typedef struct {
+    char     char_val;
+    int32_t  int_val;
+    float    float_val;
+    double   double_val;
+  } Ex2Struct;
+
+  Ex2Struct struct_val = { '0', 1, 1.0, 1.0 };
+
+  PrintBytes(&char_val, sizeof(char));
+  PrintBytes(&int_val, sizeof(int32_t));
+  PrintBytes(&float_val, sizeof(float));
+  PrintBytes(&double_val, sizeof(double));
+  PrintBytes(&struct_val, sizeof(struct_val));
+
+  return EXIT_SUCCESS;
+}
```

Commit Message: "Add starter code from assignment webpage"

Commit changes

Example Commits for Exercise 2

```
commit 0854ff92c98b3e29a7661a668bb693a62f0c70
Author: Alex Sanchez-Stern
Date:   Sun Jun 29 11:11:08 2025 -0700

    Add #includes for starter code

    stdint for `int32_t` and stdlib for `EXIT_SUCCESS`

diff --git a/ex2.c b/ex2.c
index ab2562d..fcc3984 100644
--- a/ex2.c
+++ b/ex2.c
@@ -1,3 +1,6 @@
+#include <stdint.h>
+#include <stdlib.h>
+
 int main(int argc, char **argv) {
   char     char_val = '0';
   int32_t  int_val = 1;
```

More details on further lines

Example Commits for Exercise 2

```
commit dcbb9db51dfd7fabd1b8074eaff590d401943
Author: Alex Sanchez-Stern
Date:   Sun Jun 29 11:12:54 2025 -0700

    Add declaration and dummy definition of

diff --git a/ex2.c b/ex2.c
index fcc3984..1694ec9 100644
--- a/ex2.c
+++ b/ex2.c
@@ -1,6 +1,8 @@
 #include <stdint.h>
 #include <stdlib.h>

+void PrintBytes(void* mem_addr, int num_bytes);
+
 int main(int argc, char **argv) {
   char     char_val = '0';
   int32_t  int_val = 1;
@@ -24,3 +26,6 @@ int main(int argc, char **argv) {

   return EXIT_SUCCESS;
 }
+
+void PrintBytes(void* mem_addr, int num_bytes){
+}
```

As small as independently describable

Example Commits for Exercise 2

```
commit 4927076469cfcdbaf8d7cf900c11f6ff4fb2cef4
Author: Alex Sanchez-Stern [redacted]
Date:   Sun Jun 29 11:28:43 2025 -0700

    Change stdint for inttypes to get the print specifiers

diff --git a/ex2.c b/ex2.c
index 1694ec9..fa85db3 100644
--- a/ex2.c
+++ b/ex2.c
@@ -1,4 +1,4 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>

 void PrintBytes(void* mem_addr, int num_bytes);
```

Describe **why** when not immediately obvious

Example Commits for Exercise 2

```
commit f0efb7f8b0573e0553447d717ce1664ee2576e08 (HEAD -> main)
Author: Alex Sanchez-Stern [redacted]
Date:   Sun Jun 29 11:29:15 2025 -0700

    Add body to PrintBytes

diff --git a/ex2.c b/ex2.c
index fa85db3..40c7396 100644
--- a/ex2.c
+++ b/ex2.c
@@ -1,5 +1,6 @@
 #include <inttypes.h>
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

@@ -28,4 +29,10 @@ int main(int argc, char **argv) {
 }

 void PrintBytes(void* mem_addr, int num_bytes){
+    printf("The %d bytes starting at %p are:", num_bytes, mem_addr);
+    for (int i = 0; i < num_bytes; i++) {
+        // Could also use ((uint8_t*)mem_addr)[i]
+        printf(" %02" PRIx8, *(uint8_t*)(mem_addr+i));
+    }
+    printf("\n");
 }
```

Group logically related changes

Git Tips

- Run `git diff` before anything else to see your current changes.
- Run `git diff --cached` after adding but before committing to see what you're about to commit
- Run `git status` at any time to get an overview

See extra slides for more advanced git usage if you're feeling adventurous

Lecture Outline

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

Memory Allocation So Far

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

```
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;
}
```

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

int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

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

Why Dynamic Allocation?

- 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
  - For example, reading a file into memory….

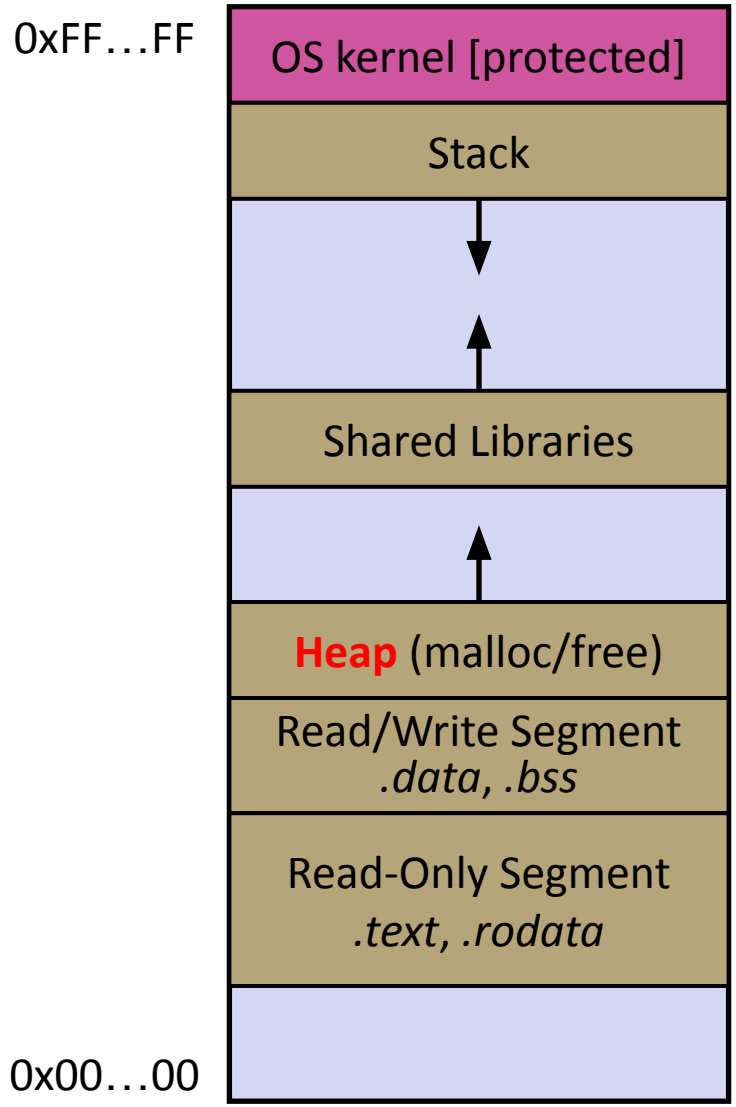
```
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

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

Dynamic Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The code allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocates it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)
    - **New**: compiler inserted code deallocates it (ownership types)
- C requires you to manually manage memory
  - Gives you more control, but causes headaches

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



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;
}
```

Lecture Outline

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

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 and cast the result to the desired pointer type

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

calloc()

- General usage:

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

- Like **malloc**, but also zeros out the block of memory
  - Helpful when zero-initialization wanted (but don't use it to mask bugs – fix those)
  - Slightly slower; but useful for non-performance-critical code or if you really are planning to zero out the new block of memory

```
// 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);`

All three of these functions are in `stdlib.h`

- 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 (re-)allocation
  - The bits in the pointer are *not changed* by calling free
    - Defensive programming: can set pointer to `NULL` after freeing it

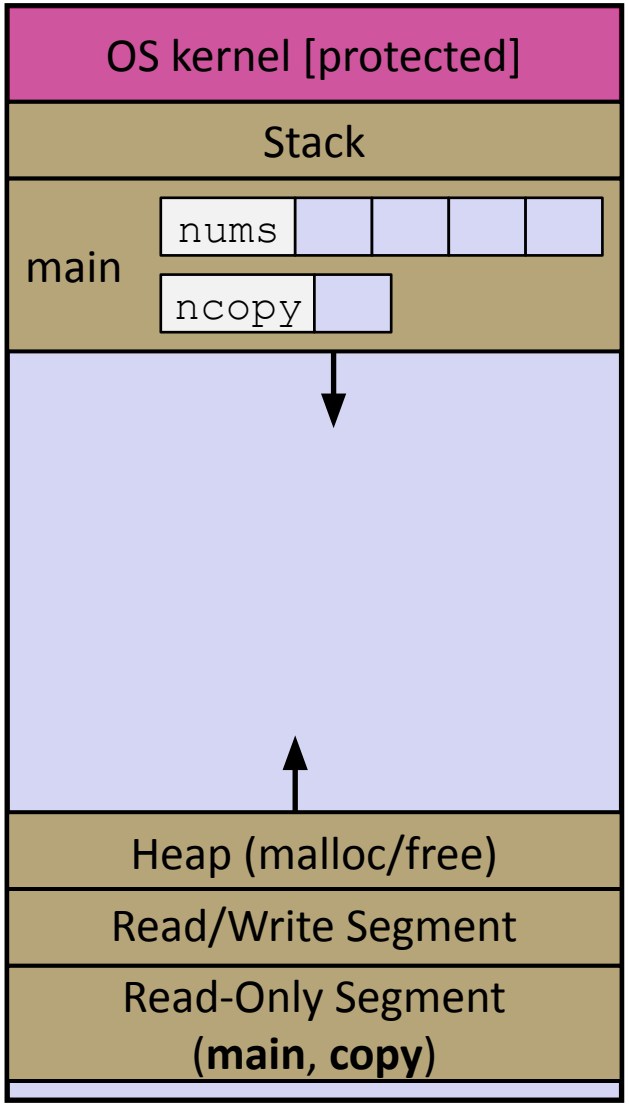
```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL
```

Heap and Stack Example

Note: Arrow points to *next* instruction.

arraycopy.c

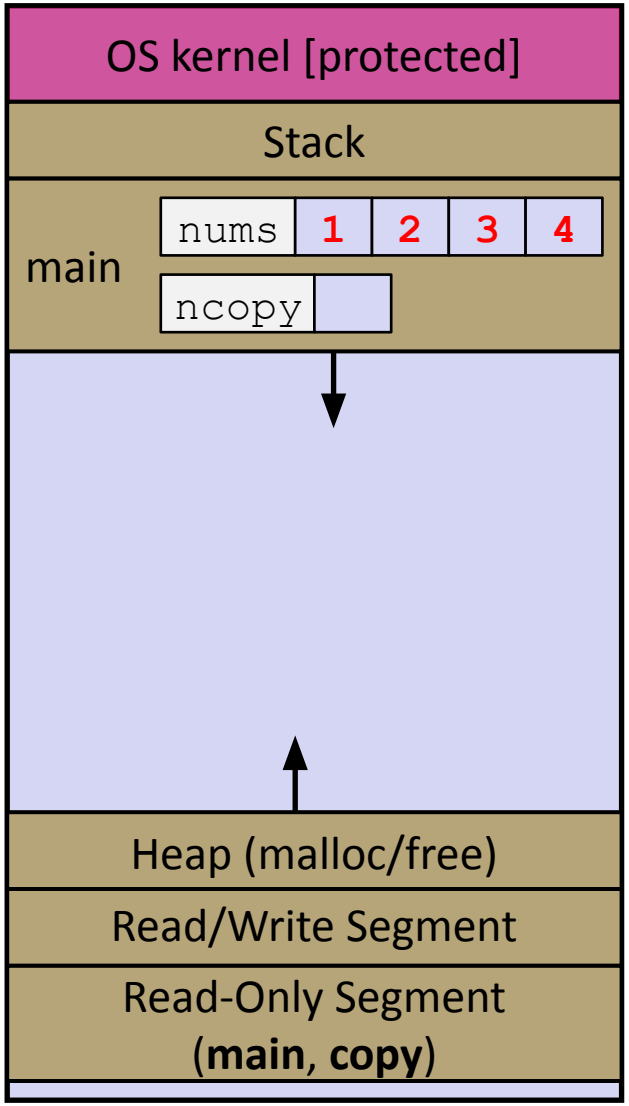
```
#include <stdlib.h>

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 0;
}
```



Heap and Stack Example

arraycopy.c

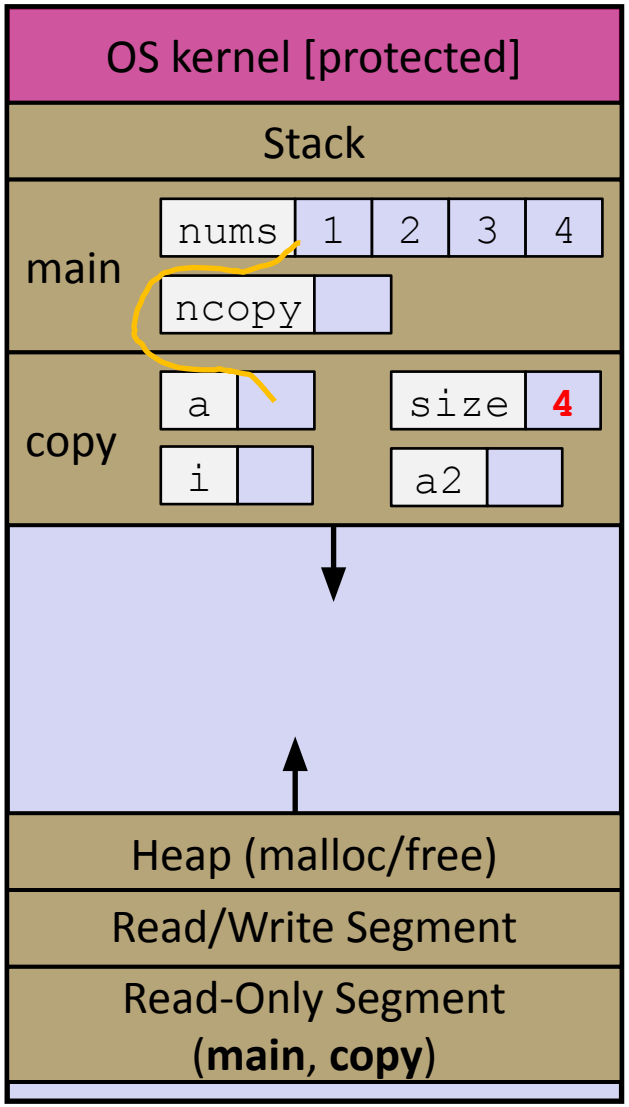
```c
#include <stdlib.h>

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

arraycopy.c

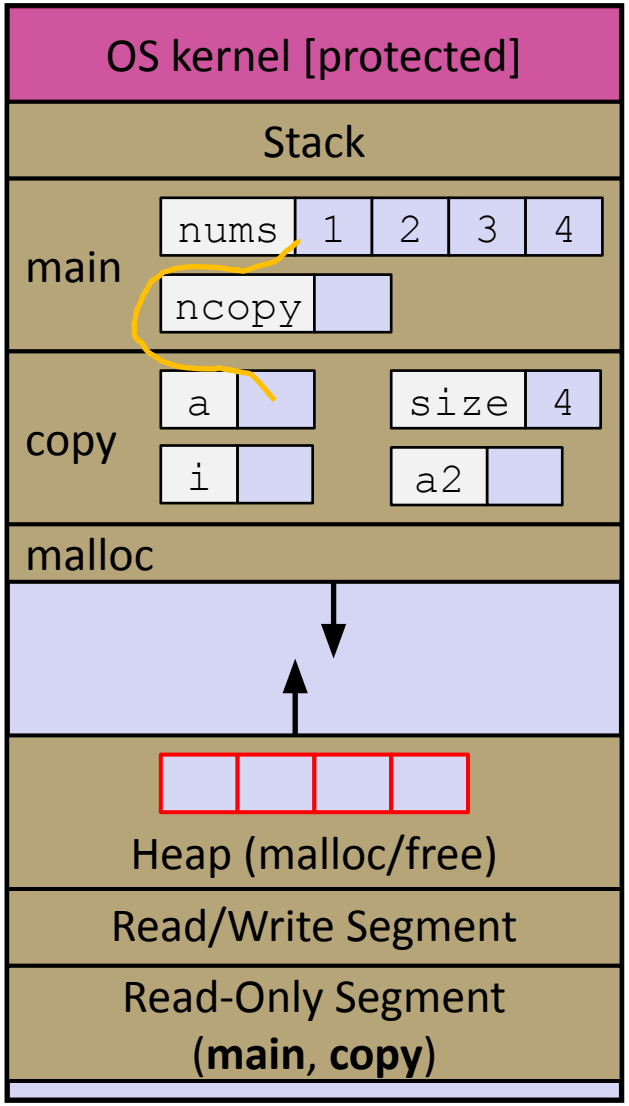
```
#include <stdlib.h>

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

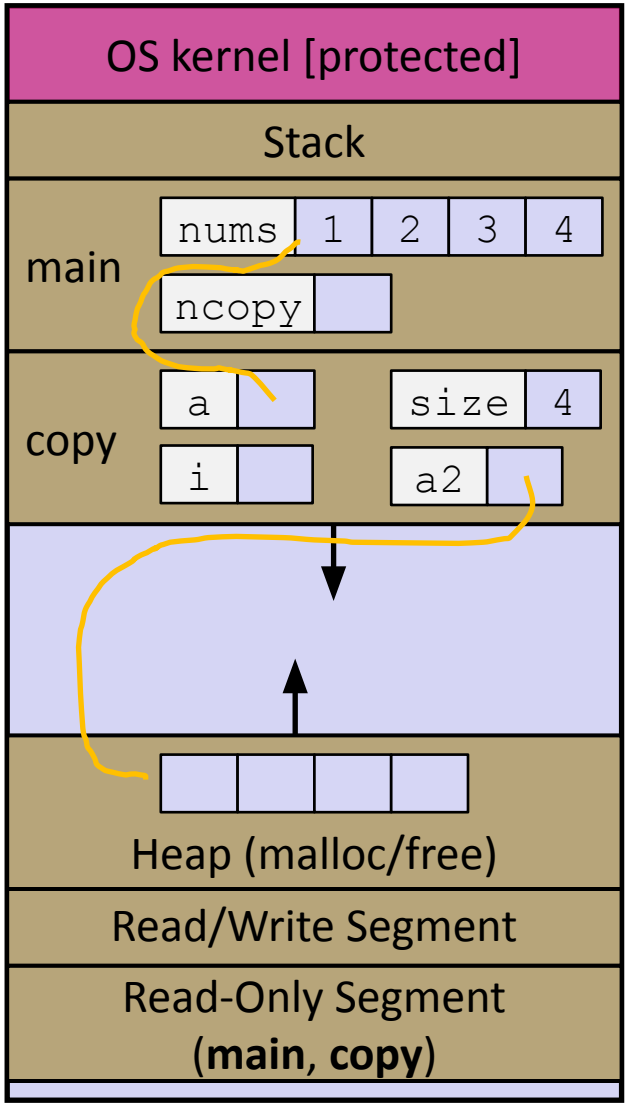
```
#include <stdlib.h>

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

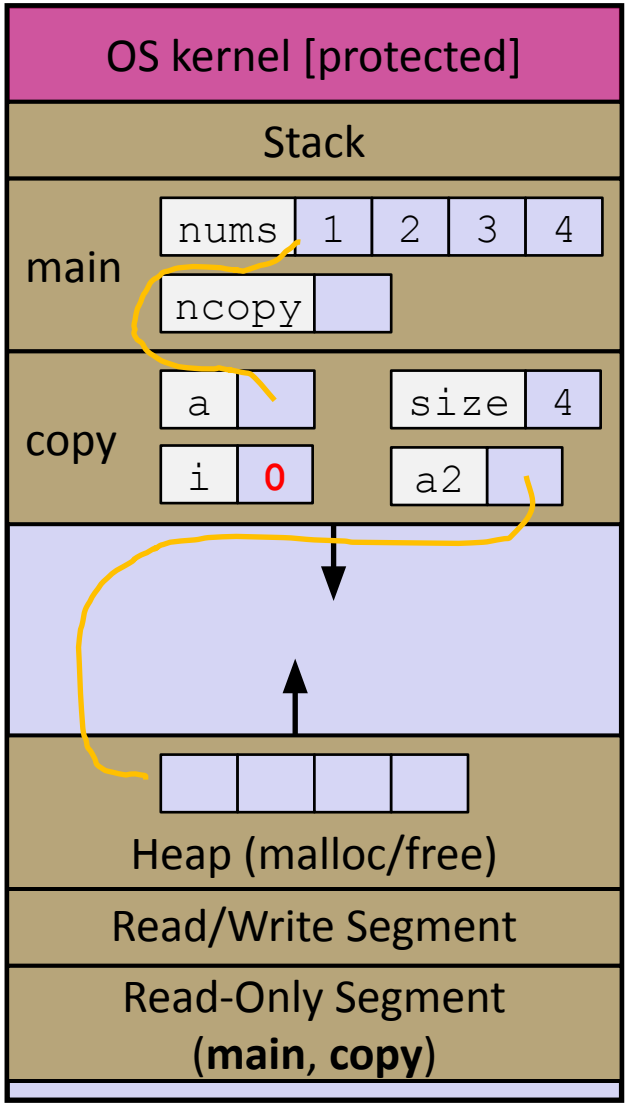
```
#include <stdlib.h>

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

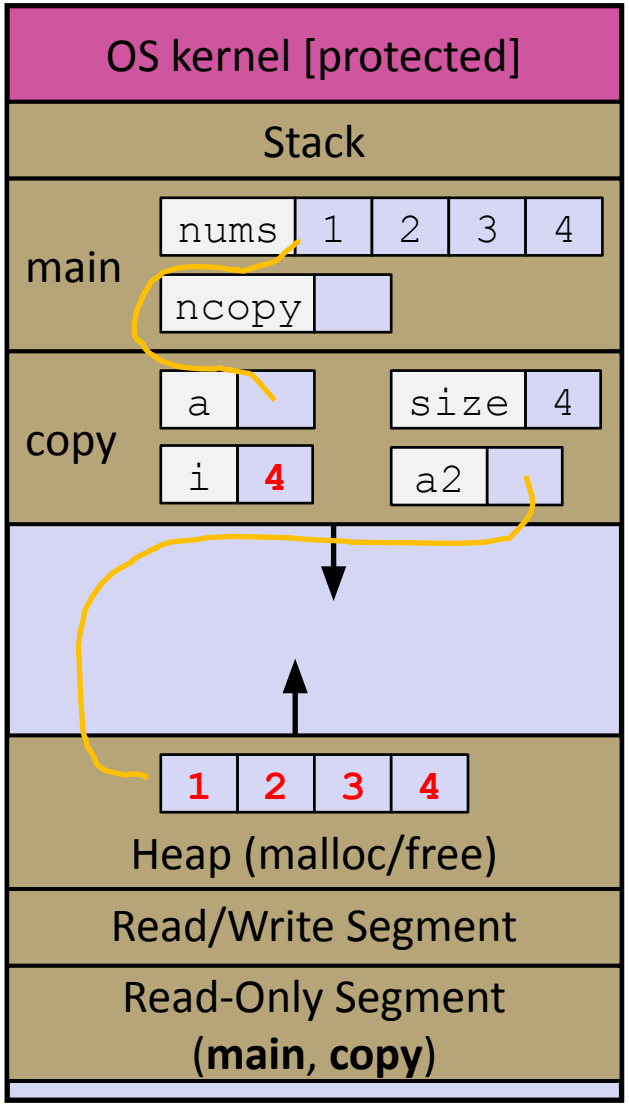
```
#include <stdlib.h>

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

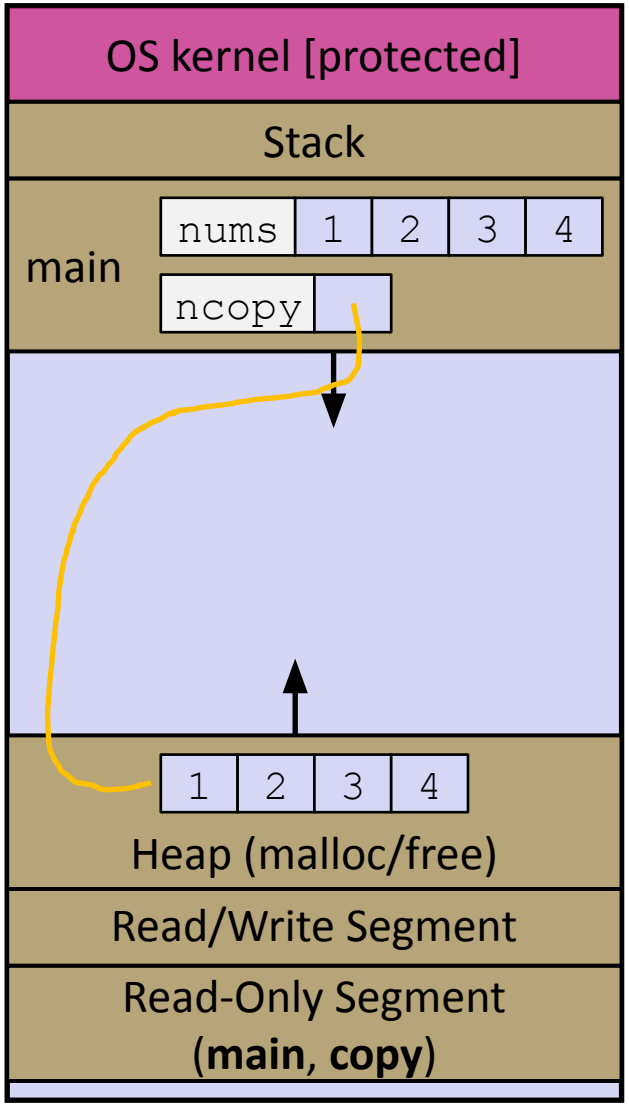
```
#include <stdlib.h>

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

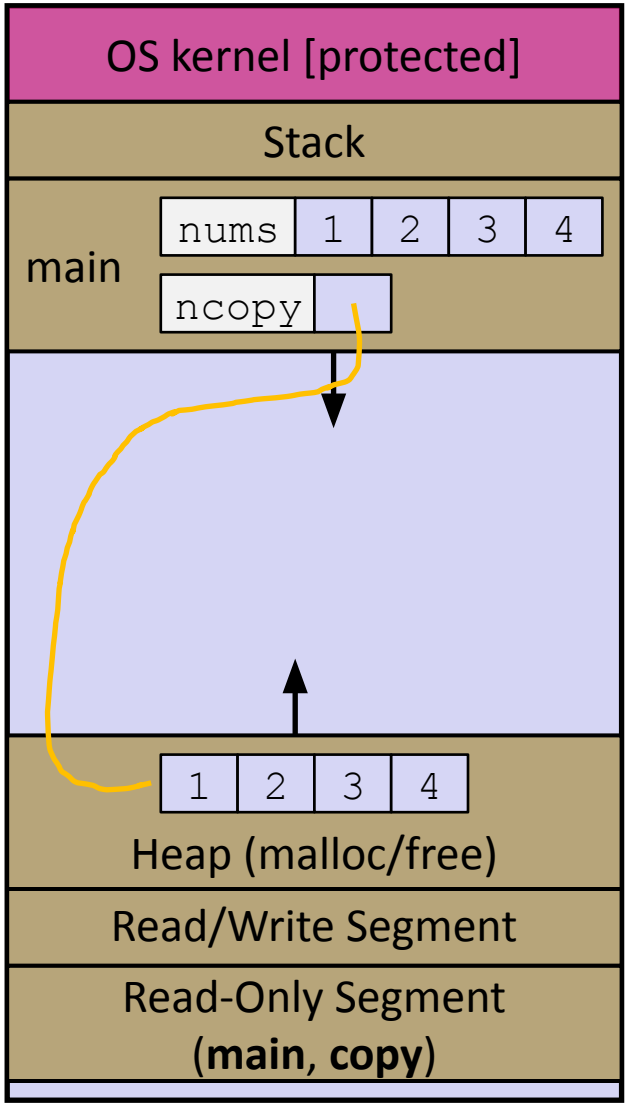
```
#include <stdlib.h>

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

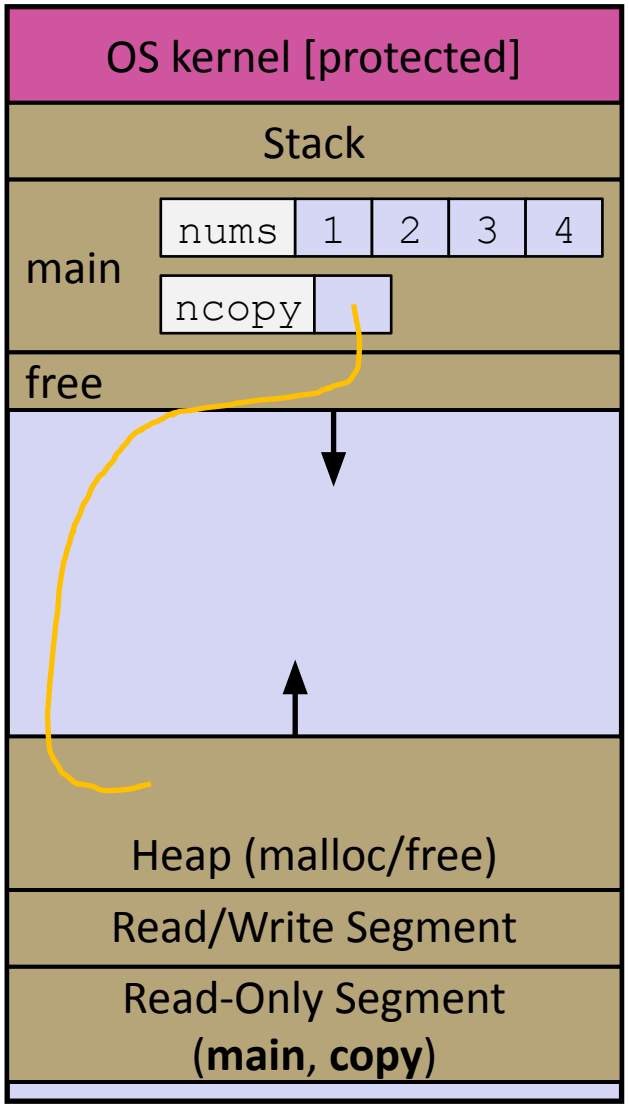
```
#include <stdlib.h>

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

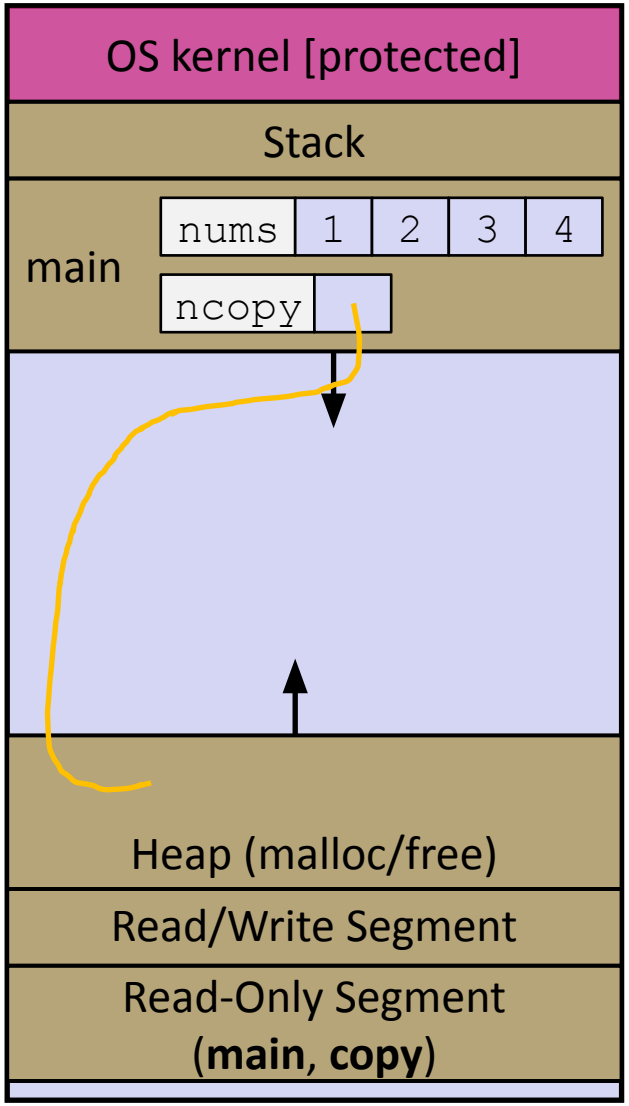
```
#include <stdlib.h>

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;
}
```



Lecture Outline

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

Memory Corruption

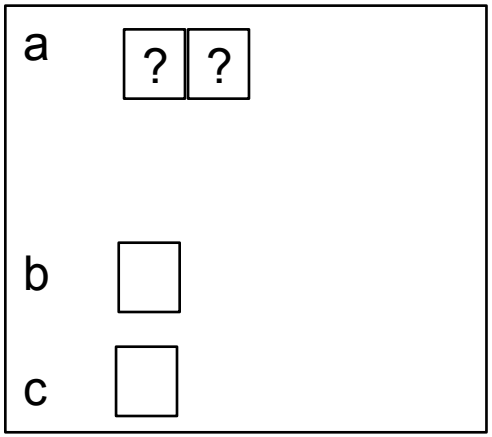
- There are all sorts of ways to corrupt memory in C

```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 (dangling) pointer

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

memcorrupt.c

Memory Corruption - What Happens?

stack: main

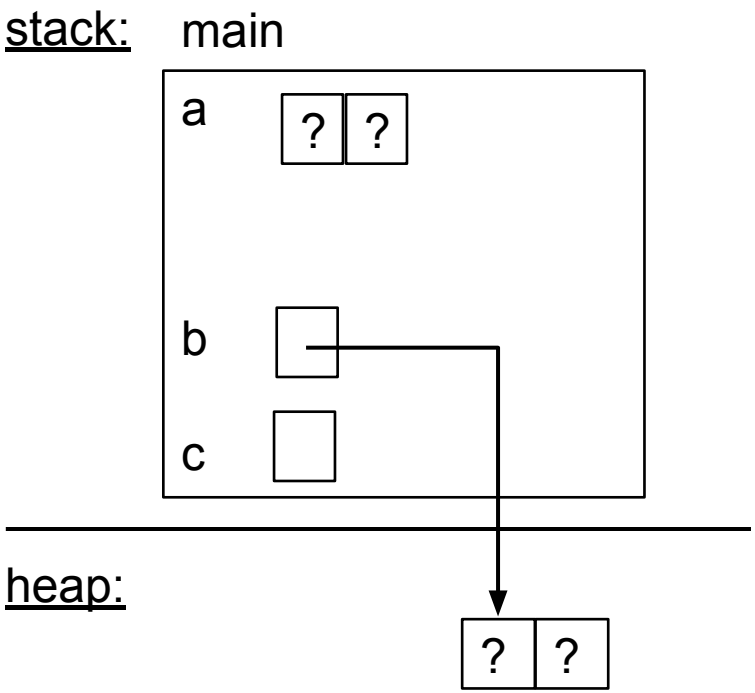


heap:

```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 (dangling) pointer

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

memcorrupt.c

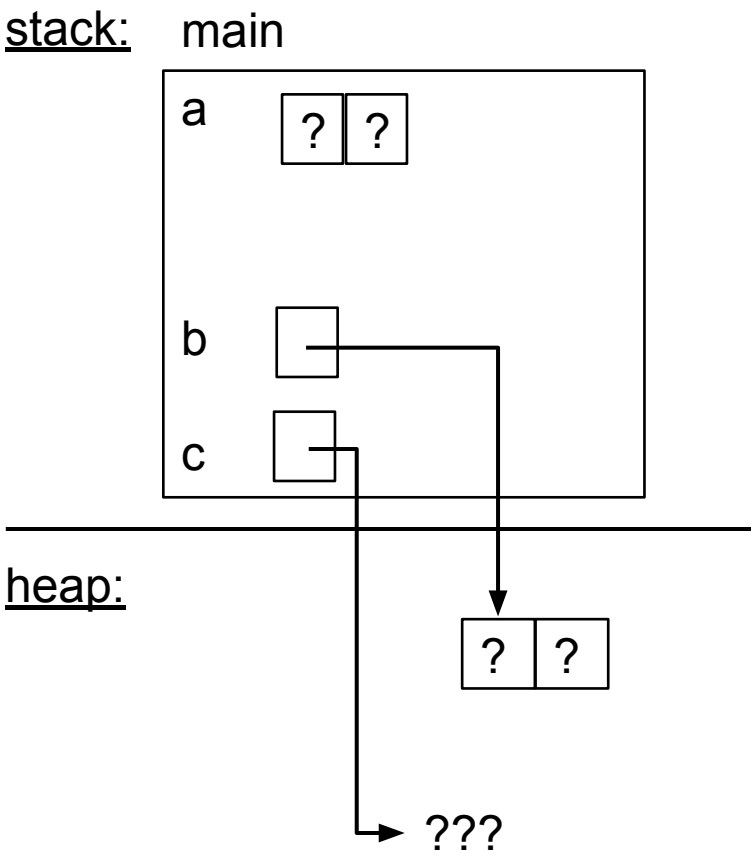
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

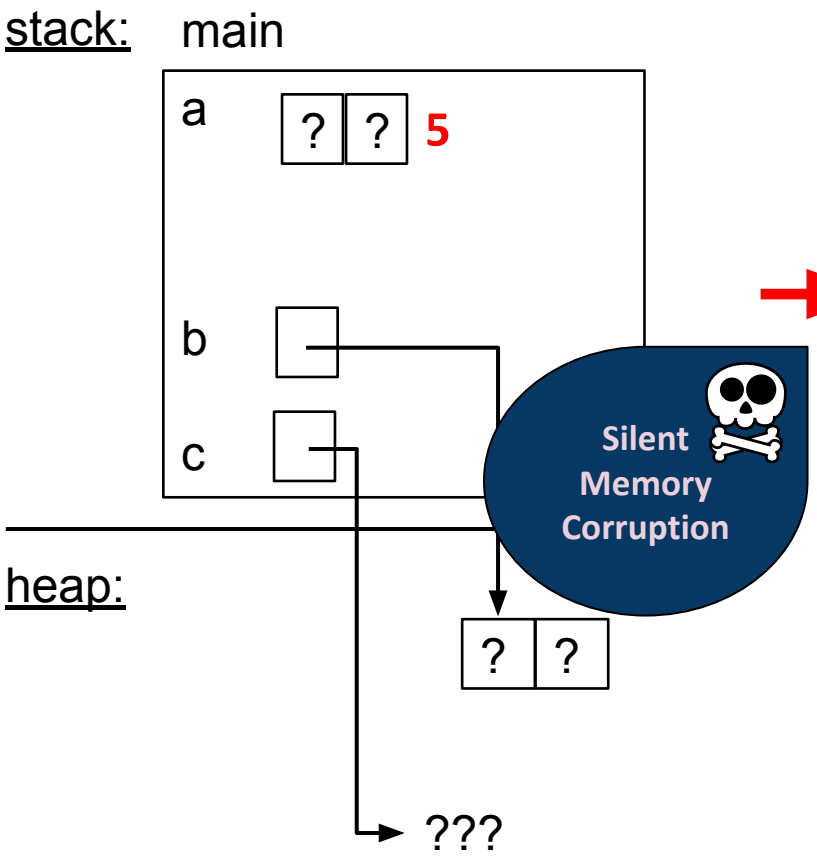
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

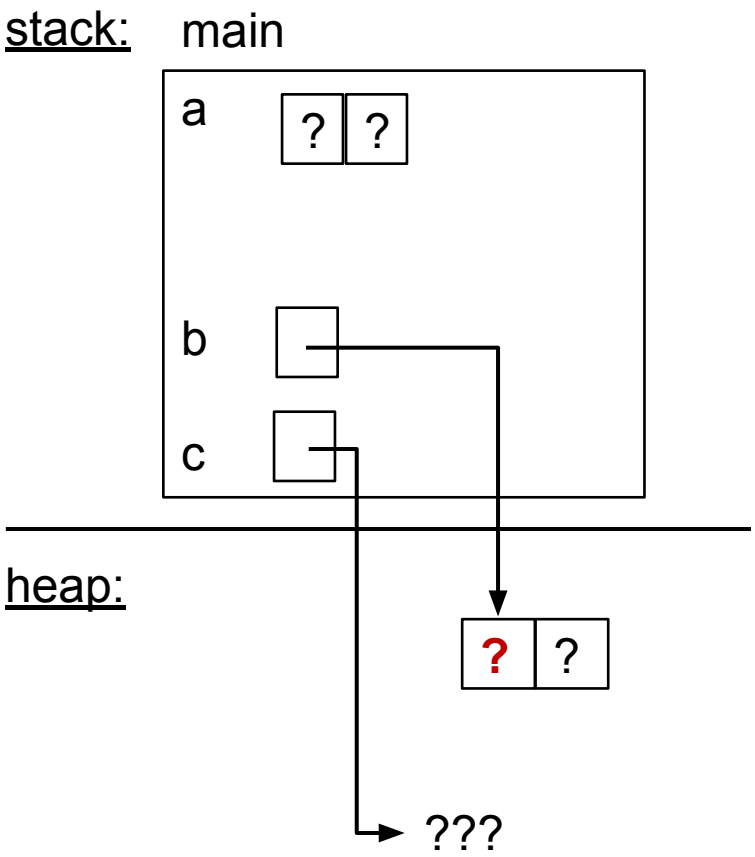
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

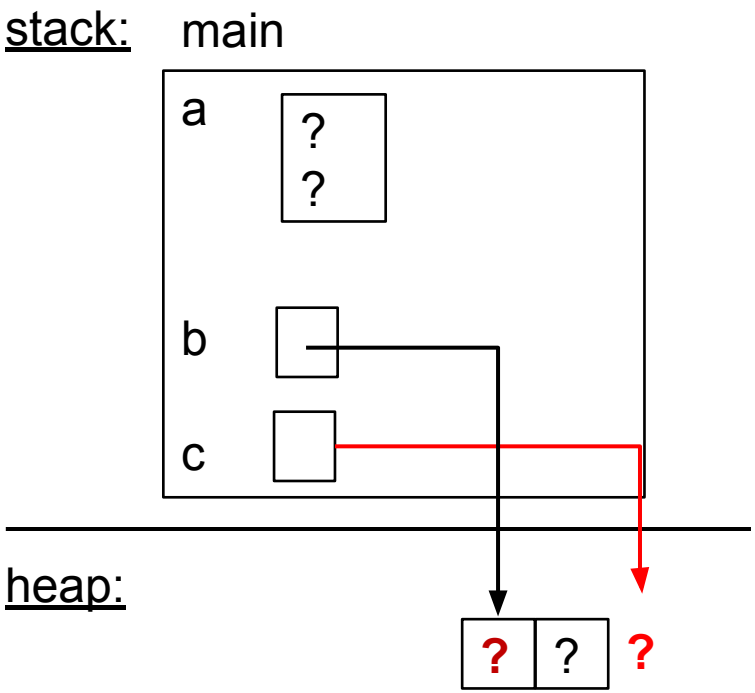
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

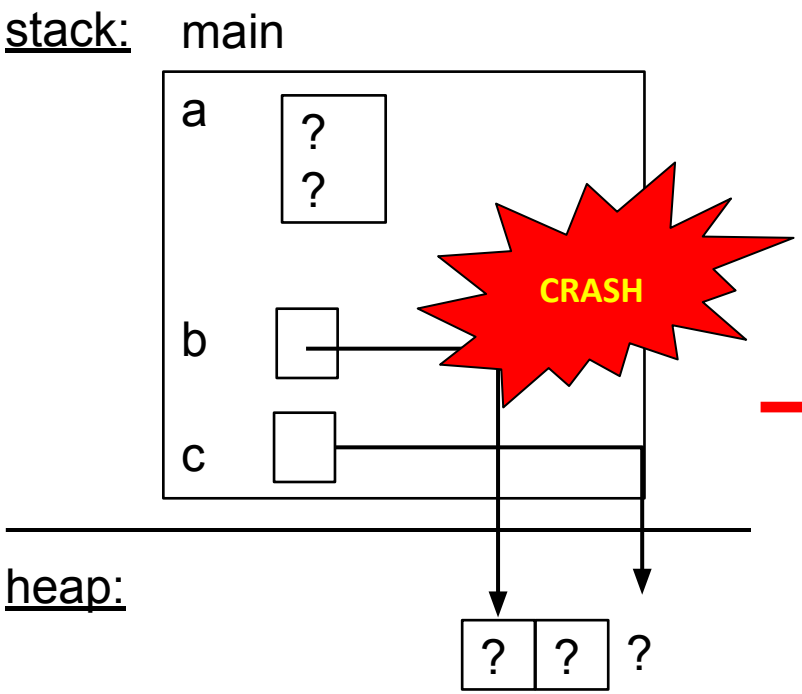
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

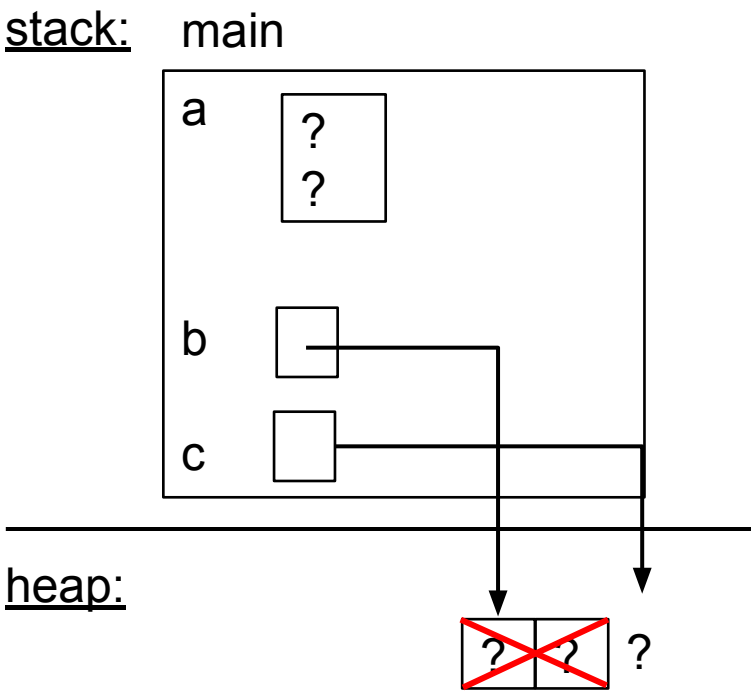
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

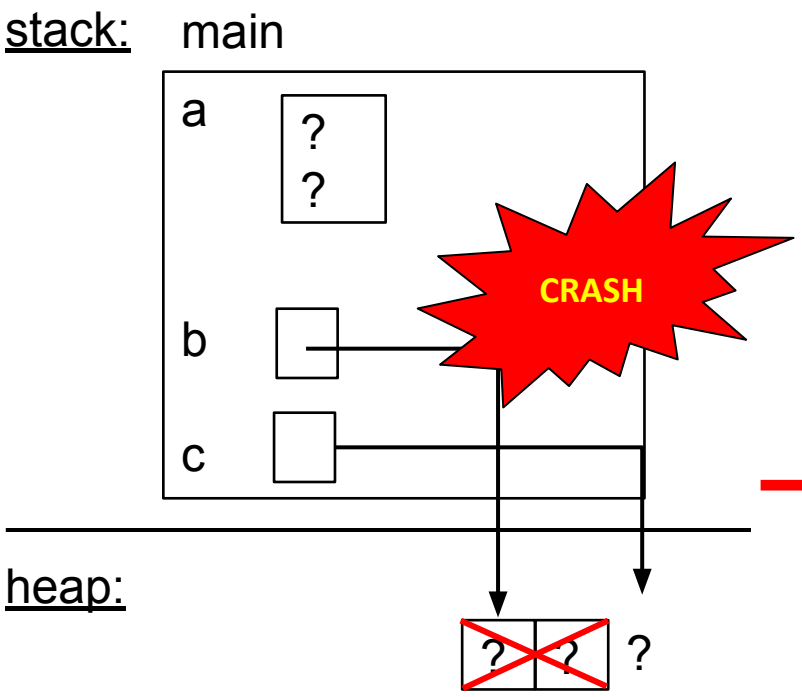
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

memcorrupt.c

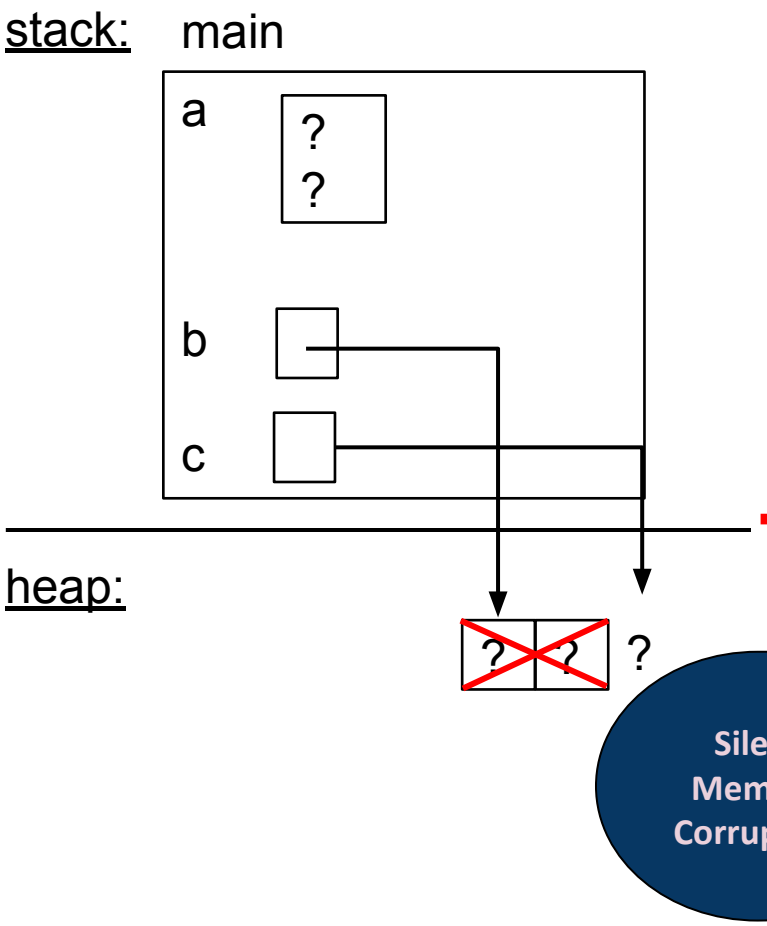
Memory Corruption - What Happens?




```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;    // 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 (dangling) pointer

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

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;    // 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 (dangling) pointer

  // and 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 the block
  - Takes real work to prevent – as pointers are passed around, what part of the program is responsible for freeing each malloc-ed block?
- What happens: program's memory 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 memory 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 (can assign, pass by value, …)
  - A struct *tagname* is a *tag*; ***not*** a full first-class type name
- Declaration template:

```
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
Point origin = {0.0, 0.0};
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
  - Shorthand for: dereference pointer first, then accesses field
    - Using p->x instead of (*p).x is standard practice – do it that way

```
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

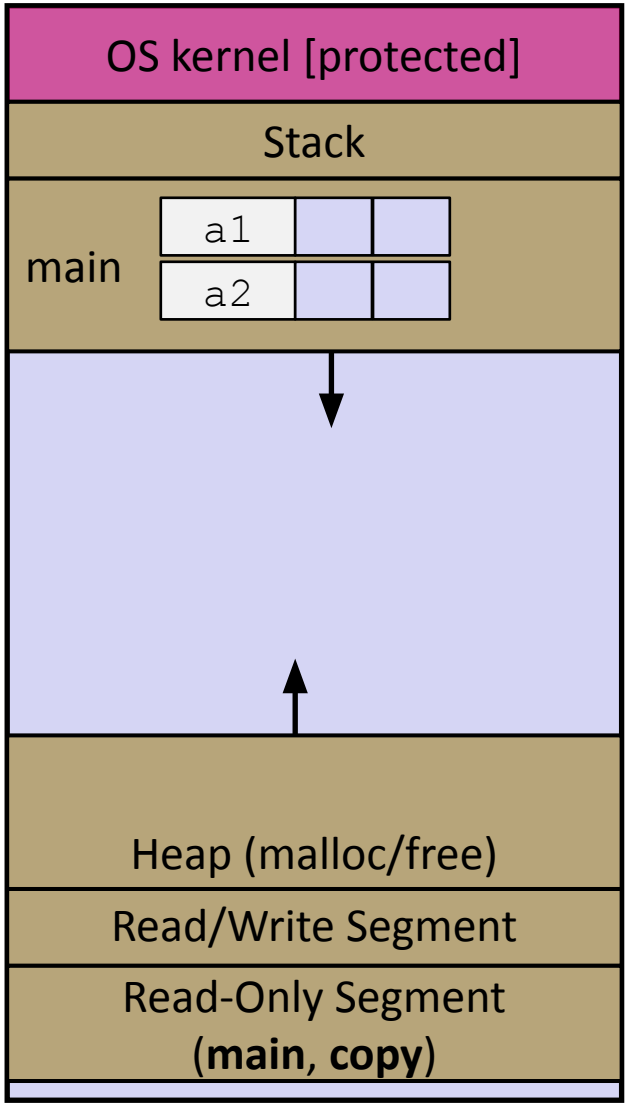
Copy by Assignment

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

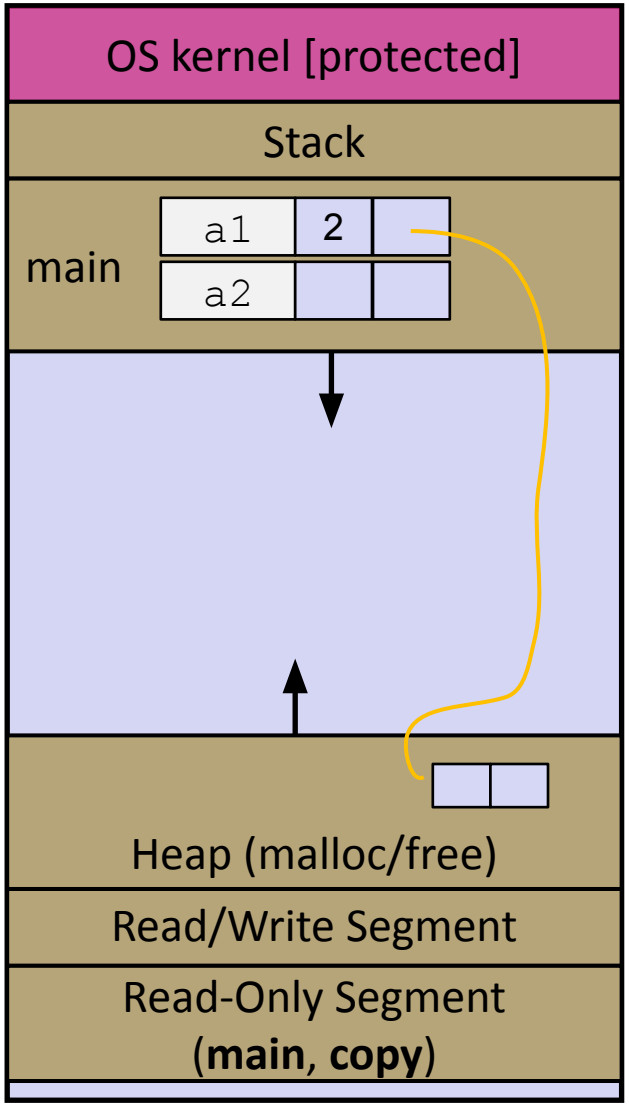
```c
#include <stdio.h>

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: {0.0,2.0}
         p1.x, p1.y, p2.x, p2.y);        // p2: {4.0,6.0}
  p2 = p1;
  printf("p1: {%f,%f}  p2: {%f,%f}\n", // p1: {0.0,2.0}
         p1.x, p1.y, p2.x, p2.y);      // p2: {0.0,2.0}
  return 0;
}
```

structassign.c

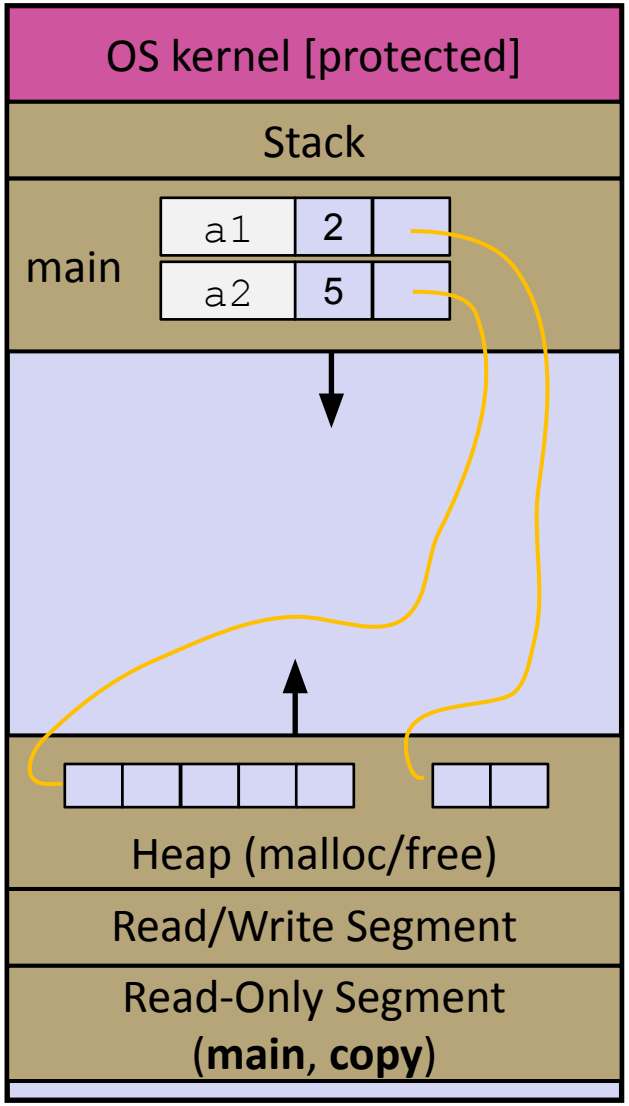
```
struct SmartArray {
  int len;
  char* arr;
};

int main(int argc, char** argv) {
  struct SmartArray a1
    = {2, (char*)malloc(sizeof(char)*2)};
  struct SmartArray a2
    = {5, (char*)malloc(sizeof(char)*5)};
  a1 = a2;

  return 0;
}
```



```
struct SmartArray {
  int len;
  char* arr;
};

int main(int argc, char** argv) {
  struct SmartArray a1
    = {2, (char*)malloc(sizeof(char)*2)};
  struct SmartArray a2
    = {5, (char*)malloc(sizeof(char)*5)};
  a1 = a2;

  return 0;
}
```

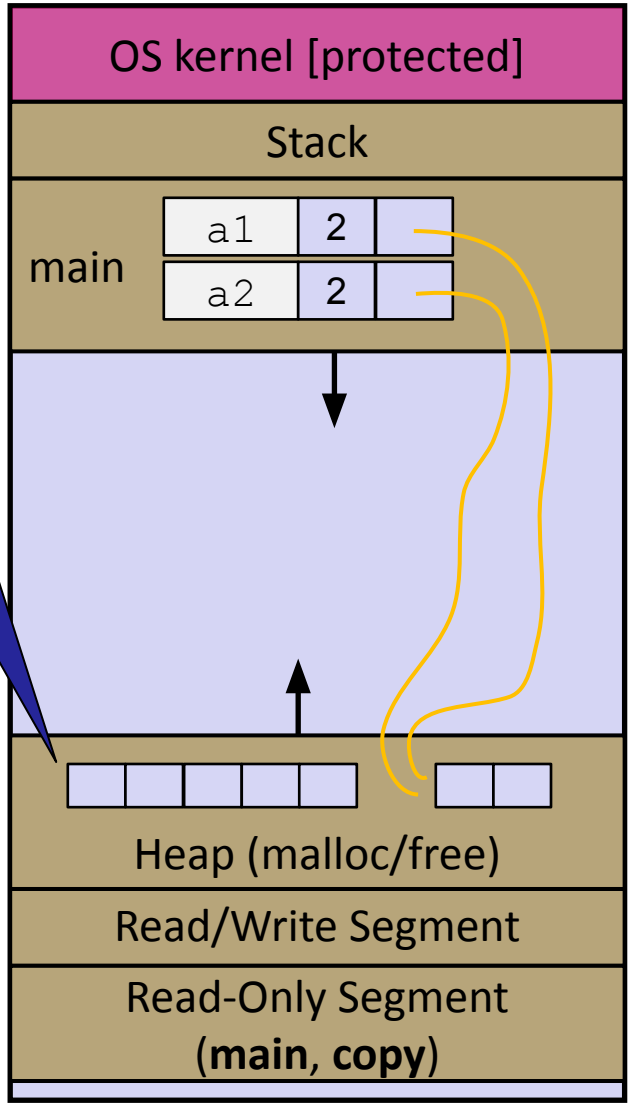


```
struct SmartArray {
  int len;
  char* arr;
};

int main(int argc, char** argv) {
  struct SmartArray a1
    = {2, (char*)malloc(sizeof(char)*2)};
  struct SmartArray a2
    = {5, (char*)malloc(sizeof(char)*5)};
  a1 = a2;

  return 0;
}
```



This memory is unreachable now!

```
struct SmartArray {
  int len;
  char* arr;
};

int main(int argc, char** argv) {
  struct SmartArray a1
    = {2, (char*)malloc(sizeof(char)*2)};
  struct SmartArray a2
    = {5, (char*)malloc(sizeof(char)*5)};
  a1 = a2;

  return 0;
}
```



Structs as Arguments

- Structs are passed by value, like everything else in C
  - Entire struct is copied to the callees stack
  - To manipulate a struct argument, pass a pointer instead

```
struct Point{
  int x, y;
};

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

void DoubleXWorks(struct Point* p) {  p->x *= 2; }

int main(int argc, char** argv) {
  struct 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: (2,1)
  return 0;
}
```

Aside: typedef

- Definition template: `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;
}
```

complexstruct.c

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
  - The compiler will handle the details of this

```
// 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

Passing Structs: Copy or Pointer?

- Cost of Copies: if the struct is smaller than a pointer type, passing by copy is cheaper

- Cost of Accesses: accesses through pointers require more "jumping around memory"; more expensive and can be harder for compiler to optimize

- Decision:
  - For small structs (like `struct complex_st`), passing a copy of the struct can be faster and often preferred if function only reads data
  - or large structs or if the function should change caller's data, use pointers

Todo

- ex3 is due Wednesday morning
- HW1 due a week from Thursday

Advanced Git: add -p

```
diff --git i/ex2.c w/ex2.c
index 1694ec9..40c7396 100644
--- i/ex2.c
+++ w/ex2.c
@@ -1,5 +1,6 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

@@ -28,4 +29,10 @@ int main(int argc, char **argv) {
 }

 void PrintBytes(void* mem_addr, int num_bytes){
+    printf("The %d bytes starting at %p are:", num_bytes, mem_addr);
+    for (int i = 0; i < num_bytes; i++) {
+        // Could also use ((uint8_t*)mem_addr)[i]
+        printf(" %02" PRIx8, *(uint8_t*)(mem_addr+i));
+    }
+    printf("\n");
 }
```

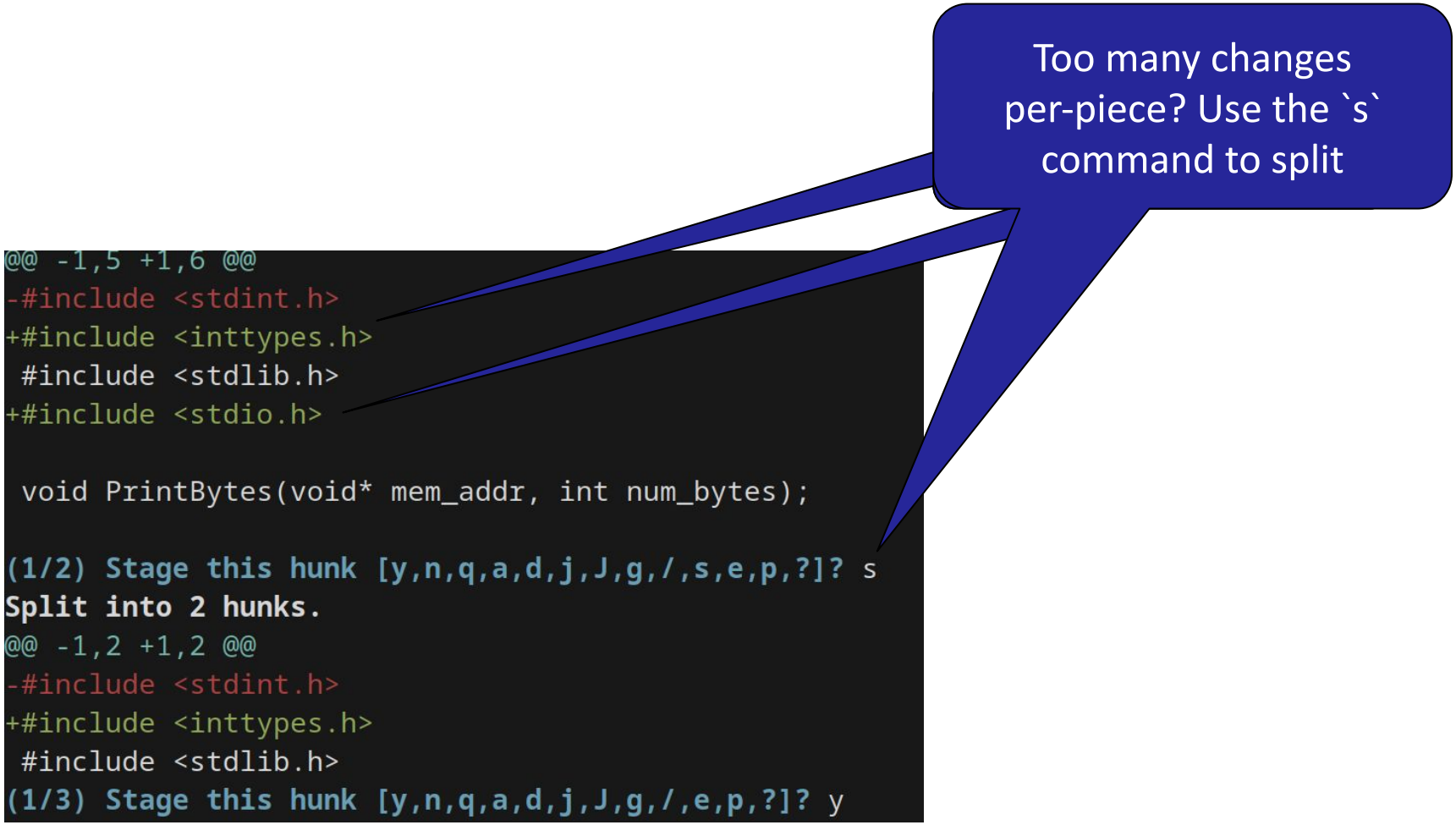
Oops, made multiple changes at once that should be separate commits!

Advanced Git: `add -p`

`git add -p` to the rescue: interactive add, by pieces

```
alex@alexLaptop> git add -p
diff --git a/ex2.c b/ex2.c
index 1694ec9..40c7396 100644
--- a/ex2.c
+++ b/ex2.c
@@ -1,5 +1,6 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,s,e,p,?]?
```

Advanced Git: add -p



Advanced Git: add -p

```
@@ -1,5 +1,6 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,s,e,p,?]? s
Split into 2 hunks.
@@ -1,2 +1,2 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>
(1/3) Stage this hunk [y,n,q,a,d,j,J,g,/,e,p,?]? y
```

'y' for "yes" to commit the current piece

Advanced Git: add -p

```
alex@alexLaptop> git add -p
diff --git a/ex2.c b/ex2.c
index 1694ec9..40c7396 100644
--- a/ex2.c
+++ b/ex2.c
@@ -1,5 +1,6 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

(1/2) Stage this hunk [y,n,q,a,d,j,J,g,/,s,e,p,?]? s
Split into 2 hunks.
@@ -1,2 +1,2 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>
(1/3) Stage this hunk [y,n,q,a,d,j,J,g,/,e,p,?]? y
@@ -2,4 +2,5 @@
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

(2/3) Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,p,?]? q
```

‘q’ for “quit” when you’ve added what you need

Advanced Git: `add -p`

```
alex@alexLaptop> git diff --cached
diff --git c/ex2.c i/ex2.c
index 1694ec9..fa85db3 100644
--- c/ex2.c
+++ i/ex2.c
@@ -1,4 +1,4 @@
-#include <stdint.h>
+#include <inttypes.h>
 #include <stdlib.h>

 void PrintBytes(void* mem_addr, int num_byte
alex@alexLaptop> git diff
diff --git i/ex2.c w/ex2.c
index fa85db3..40c7396 100644
--- i/ex2.c
+++ w/ex2.c
@@ -1,5 +1,6 @@
 #include <inttypes.h>
 #include <stdlib.h>
+#include <stdio.h>

 void PrintBytes(void* mem_addr, int num_bytes);

@@ -28,4 +29,10 @@ int main(int argc, char **argv) {
 }

 void PrintBytes(void* mem_addr, int num_bytes){
+    printf("The %d bytes starting at %p are:", num_b
+    for (int i = 0; i < num_bytes; i++) {
+        // Could also use ((uint8_t*)mem_addr)[i]
+        printf(" %02" PRIx8, *(uint8_t*)(mem_addr+i)
+    }
+    printf("\n");
 }
```

`git diff --cached` will show you the changes you've added

`git diff` will show you the changes you haven't added

Now you're ready to `git commit` just the first part of the changes.

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

- Write a function that:
  - Arguments: [1] an array of ints and [2] an array length
  - Malloc's an `int*` array of the same element length
  - Initializes each element of the newly-allocated array to point to the corresponding element of the passed-in array
  - Returns a pointer to the newly-allocated array

Extra Exercise #3

- 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);
```