Memory Management (Part 2)

Devin J. Pohly <djpohly@cse.psu.edu>

Scope

- The *scope* of a variable is the portion of a program where it can be accessed by name
- C has *lexical scope*
  - This means scope is known at compile-time
  - In C, the scope of a variable is always the innermost { } block (the whole file for globals)

```
int i;

void f(int i)
{
    i = 1;
}

void g(void)
{
    int i = 2;

    if (i > 0) {
        int i;

        i = 3;
    }

    i = 4;
}

void h(void)
{
    i = 5;
}
```

Scope vs. lifetime

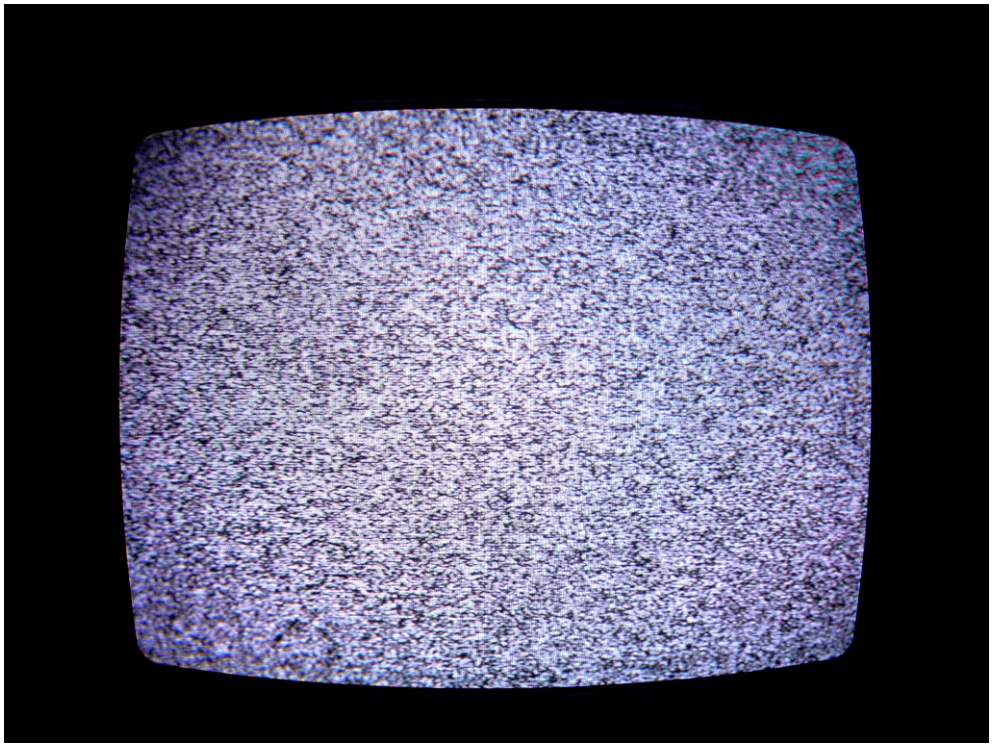
- The *lifetime* of a variable is how long the data is valid (allocated)
- Different from scope
  - Scope is about the *name* of a variable
  - Lifetime is about the *data* allocated for it

Automatic allocation

- Used for arguments and "normal" local variables
  - Generally allocated on the stack
- Scope: current function or block
- Lifetime: until the end of the function or block
  - Allocated when a function is called
  - Deallocated when it returns

Static allocation

- Used for global variables or anything else marked "static"
  - Generally allocated in the data segment
- Scope: current function or block
- Lifetime: entire life of the program
  - Allocated when the program is loaded
  - Deallocated when it exits



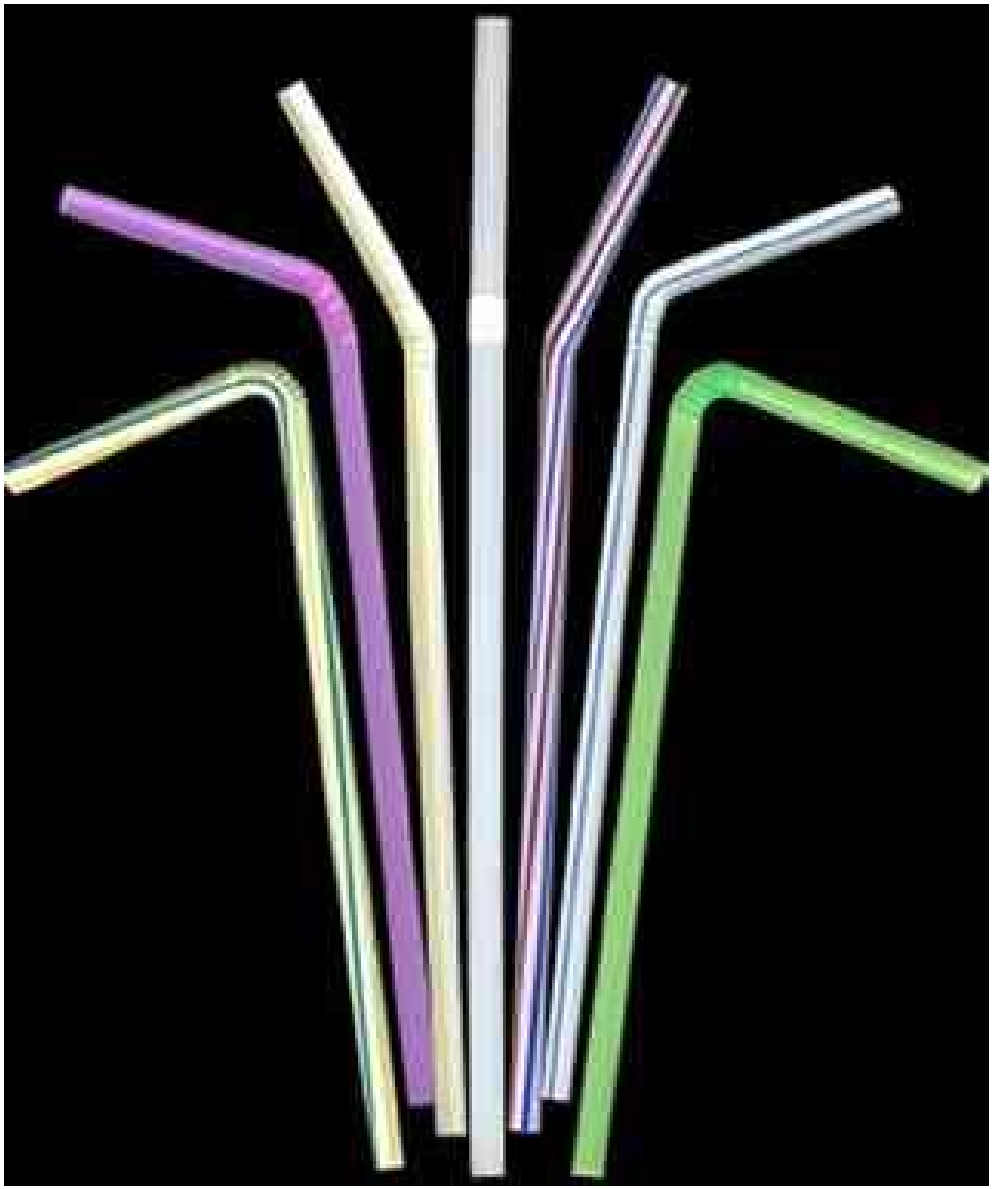
Automatic vs. static

- Which variables here are automatically allocated?
- Which are statically allocated?
- What are the scope and lifetime of the `total` variable?

```c
int counter = 0;

double running_avg(int value) {
    static int total = 0;
    double avg;

    total += value;
    counter++;

    avg = (double) total / counter;
    return avg;
}
```

More flexibility

- What if we need memory that is...
  - valid for multiple function calls, but not the entire program?
  - too big to allocate on the stack?
  - variable-size, or the size isn't known at compile time?
  - allocated and returned by a function?



Dynamic allocation

PENNSTATE 1855

- What we want is *dynamically allocated memory*
  - Your program explicitly requests a new block of memory
    - The C library allocates it, perhaps with help from the OS
    - Where in memory can this be allocated?
  - Dynamically allocated memory is valid until:
    - Your code explicitly deallocates it *(manual memory management)*
    - A garbage collector collects it *(automatic memory management)*
  - C requires you to manually manage memory
    - Gives you more control, but more opportunities to slip up... as usual

- Note: this essentially allows you to create custom scope and lifetime

The heap

- Large pool of memory that is used for dynamic allocation
  - Both allocated data and allocation metadata are stored here
  - Use `malloc` to allocate heap data and `free` to deallocate
  - Allocation functions are found in `<stdlib.h>`
- (Note: Unrelated to the "heap" data structure)

Allocating memory: `malloc`

- `malloc` allocates a block of memory of the given size:

```
malloc(size);
```

  - Returns a pointer to the first byte of the allocated memory
    - Returns `NULL` if the memory could not be allocated
  - You should assume the memory initially contains garbage
  - `sizeof` is typically used to calculate the size you need

```
// allocate a 10-float array
float *arr = malloc(10 * sizeof(float));
if (arr == NULL)
   return 1;
arr[0] = 5.1;
```

Allocating memory: `calloc`

PENNSTATE 1855

- Similar to malloc, but zeroes out allocated memory:

```
calloc(nmemb, size);
```

  - Returns an array of `nmemb` members, each of `size` bytes
  - Helpful for shaking out bugs
  - Slightly slower; preferred in non-performance-critical code

```
// allocate a 10-long array
long *arr = calloc(10, sizeof(long));
if (arr == NULL)
    return 1;
arr[0] = 5;
```

malloc vs. calloc

```
int malloc_calloc(void) {
    int i;
    unsigned char *mbuf, *cbuf;
    mbuf = malloc(5 * sizeof(char));
    cbuf = calloc(5, sizeof(char));
    for (i = 0; i < 5; i++) {
        printf("M[%d] = %02x, C[%d] = %02x\n",
               i, mbuf[i], i, cbuf[i]);
    }
    return 0;
}
```

```
M[0] = b8, C[0] = 0
M[1] = 9d, C[1] = 0
M[2] = 5e, C[2] = 0
M[3] = 40, C[3] = 0
M[4] = 4e, C[4] = 0
```

Deallocating memory

- Releases the memory pointed to by `ptr`:

```
free(ptr);
```

  - Must be the address of the first byte of allocated memory
    - I.e., the address that was returned from malloc or calloc
  - This memory can then be allocated for something else later
  - Good idea: set a pointer to NULL after you free it

```
// allocate a 10-long array
long *arr = calloc(10, sizeof(long));
// do something with arr...
free(arr);
arr = NULL;
```

The NULL pointer

- `NULL` refers to an address that is guaranteed to be invalid
  - On Linux/GCC, `NULL` is `0x0`
  - Trying to dereference `NULL` causes a segmentation fault
- Good practice: set pointers to `NULL` whenever they *don't point to valid data*
  - Initialize to `NULL`
  - Set to `NULL` after `free`

Resizing allocated memory

- Resizes the allocated memory at `ptr` to `newsz`:

```
ptr = realloc(ptr, newsz);
```

  - Note: this returns the new pointer!
    - If the memory is bigger than space allows, it has to be reallocated somewhere else, so the *address can change*!
    - **Always always always** get the return value from `realloc`!
  - Behaves like `malloc` or `free` when convenient (see man page)

```
// allocate a 10-long array, then expand
long *arr = malloc(10 * sizeof(long));
arr[0] = 5;
arr = realloc(arr, 20 * sizeof(long));
arr[15] = -47;
```

Allocating structs

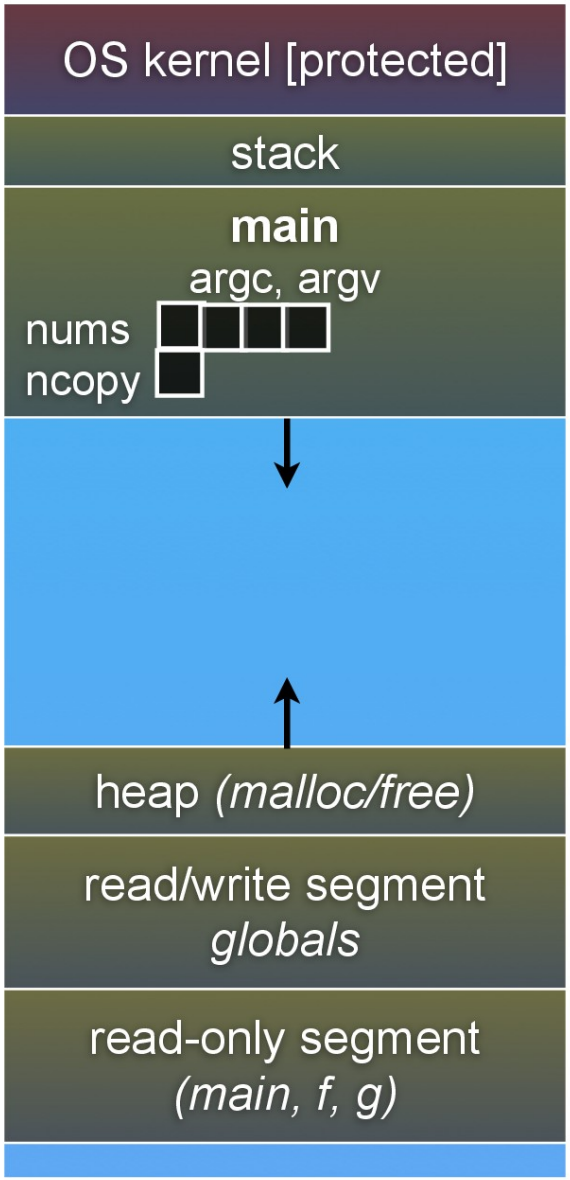
- You can `malloc` and `free` structs just like any other type
  - This is a very common scenario
  - `sizeof` is particularly helpful here

```c
struct complex {
    double real;
    double imag;
};

// Allocates and initializes a new complex number
struct complex *cplx_alloc(double r, double i)
{
    struct complex *c = malloc(sizeof(struct complex));
    if (!c)
        return NULL;
    c->real = r;
    c->imag = i;
    return c;
}

// Provides a consistent way to free the struct
void cplx_free(struct complex *c)
{
    free(c);
}
```

Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

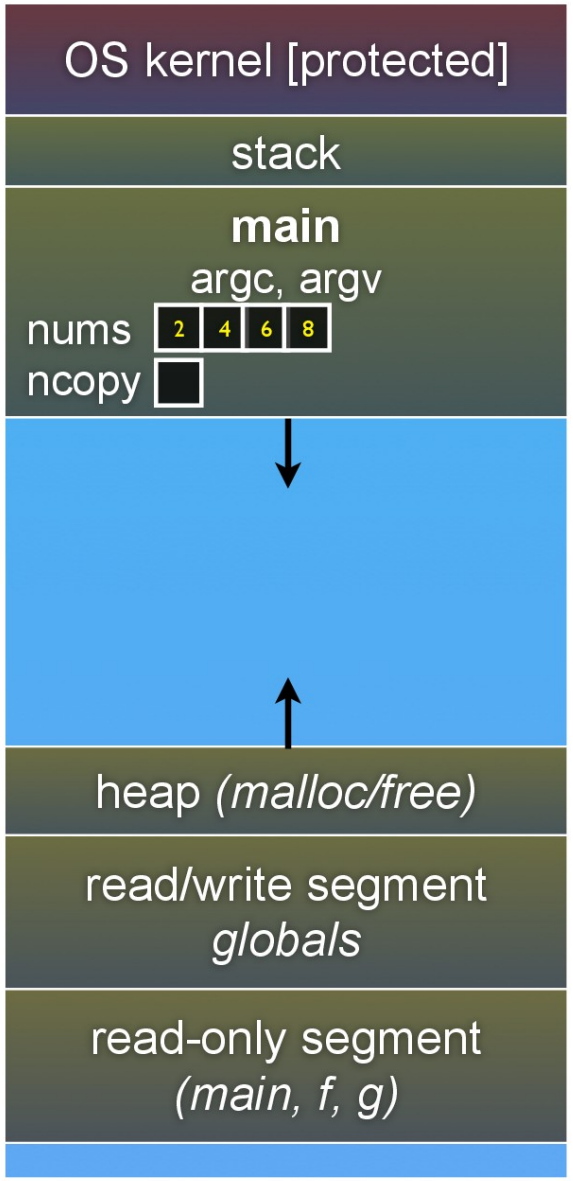
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

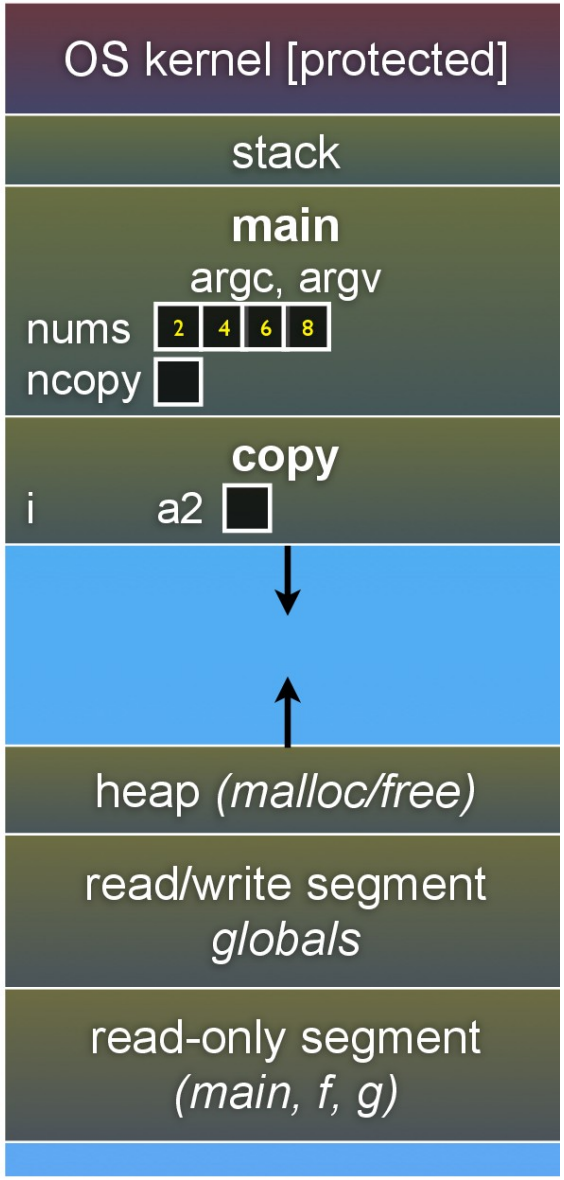
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

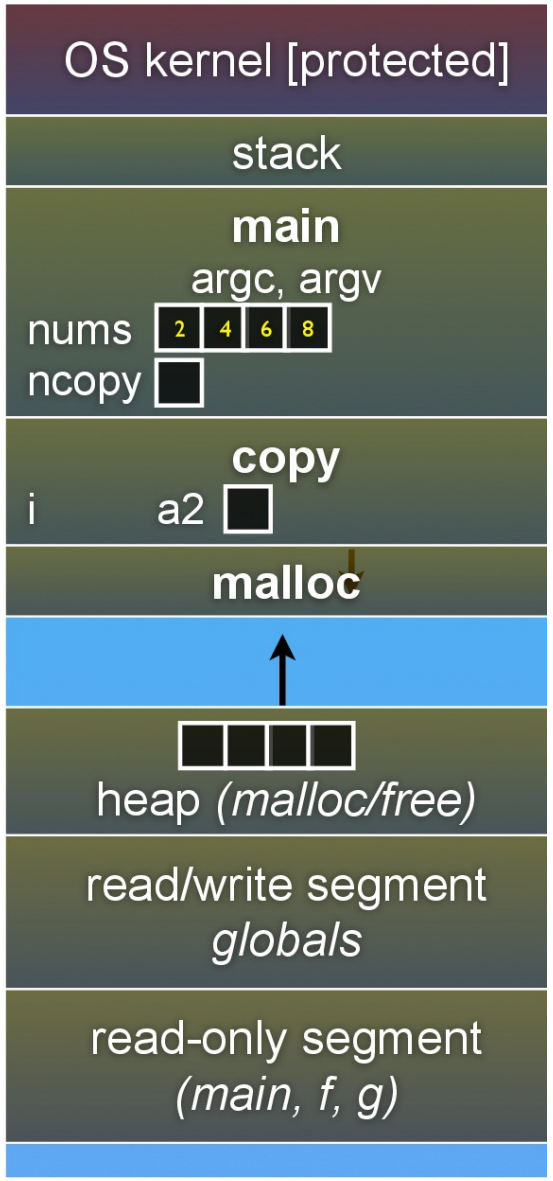
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

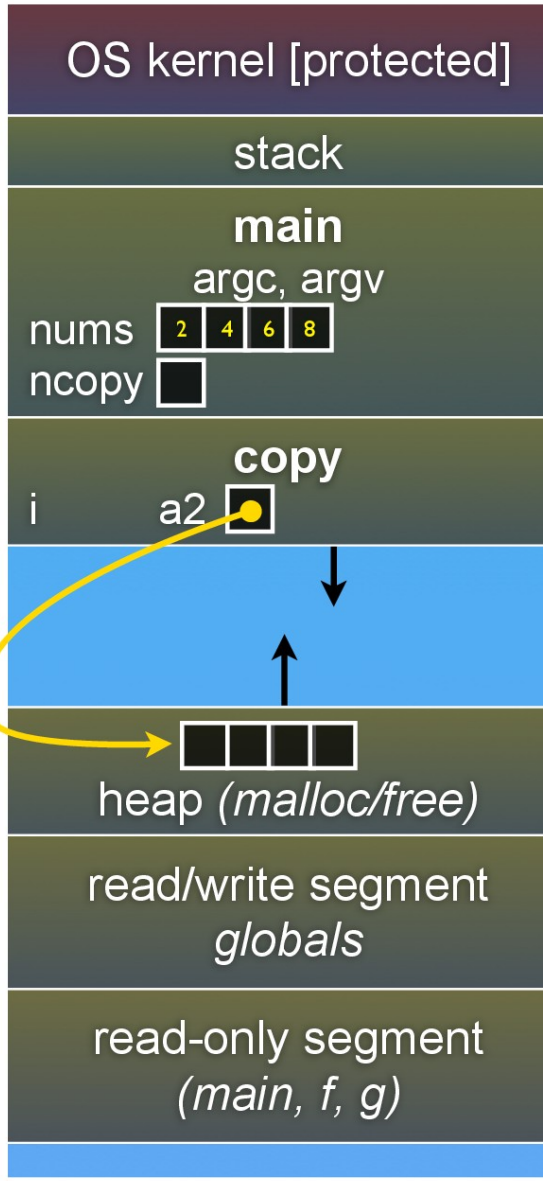
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

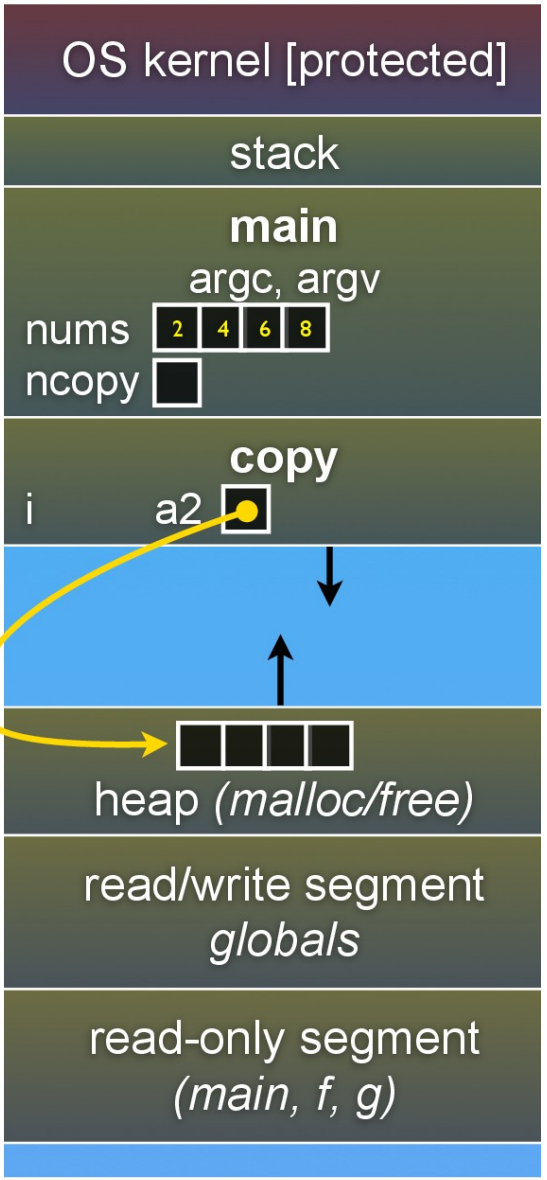
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

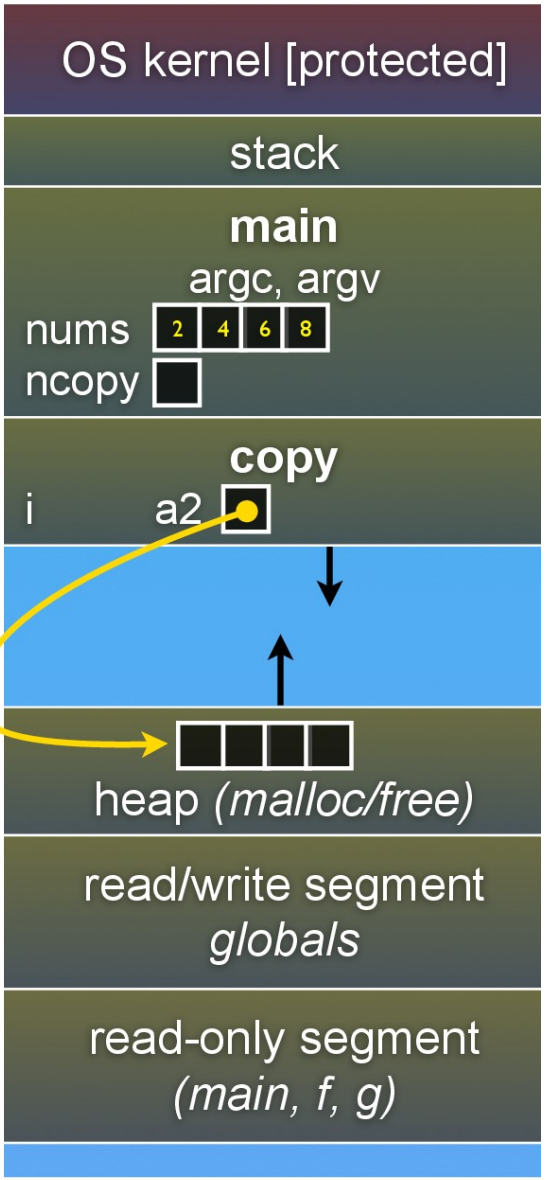
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

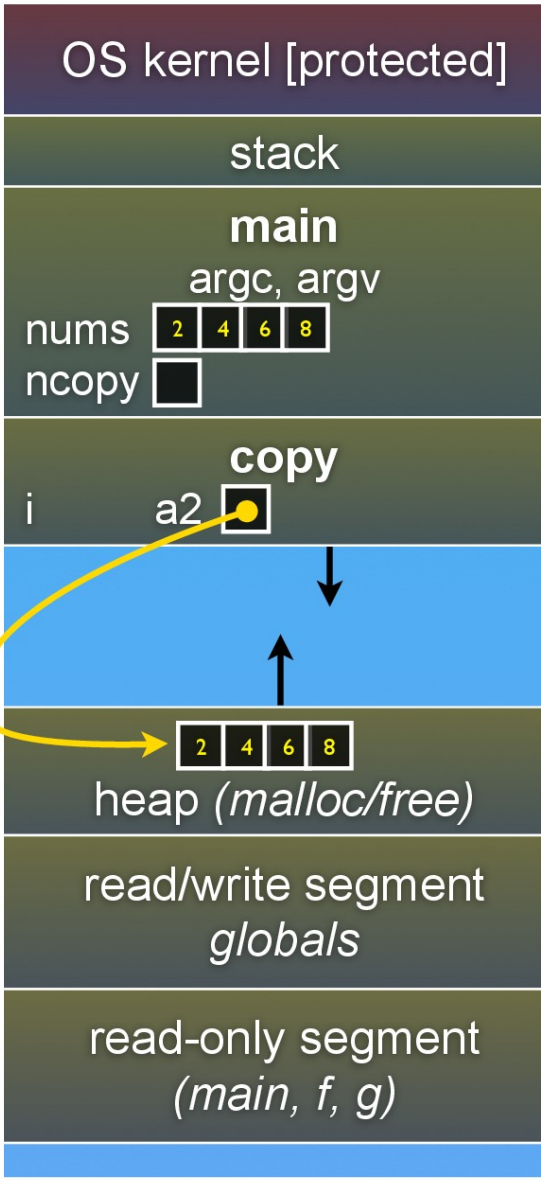
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

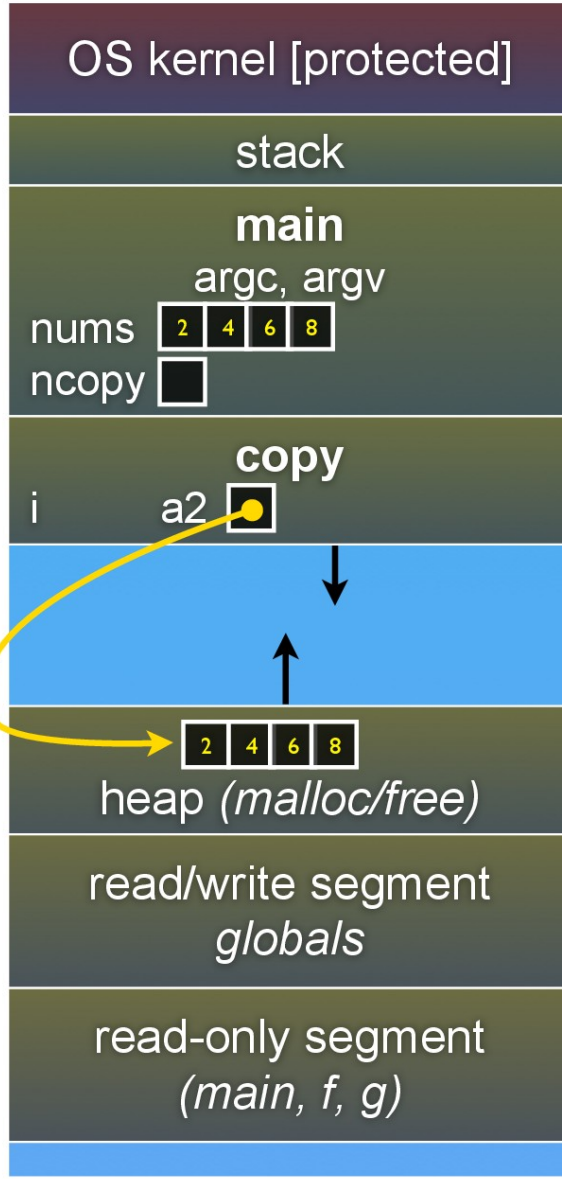
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

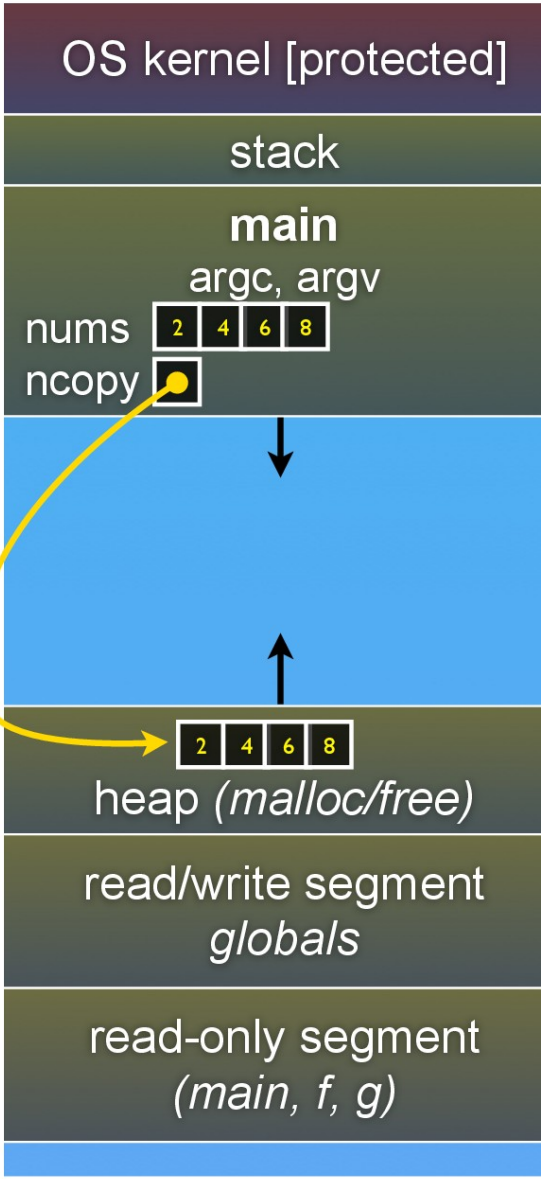
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

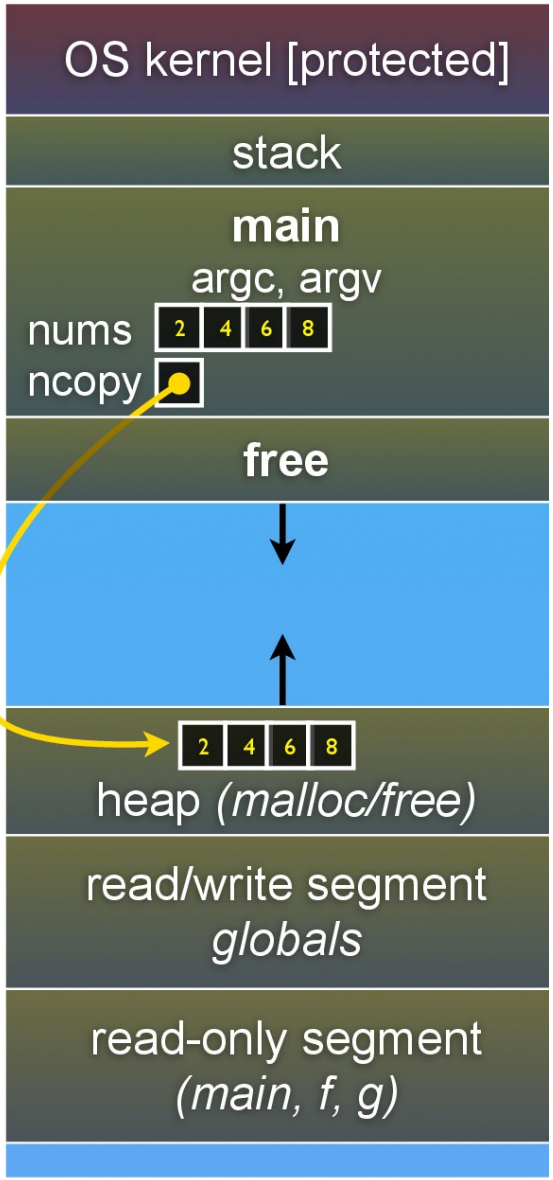
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

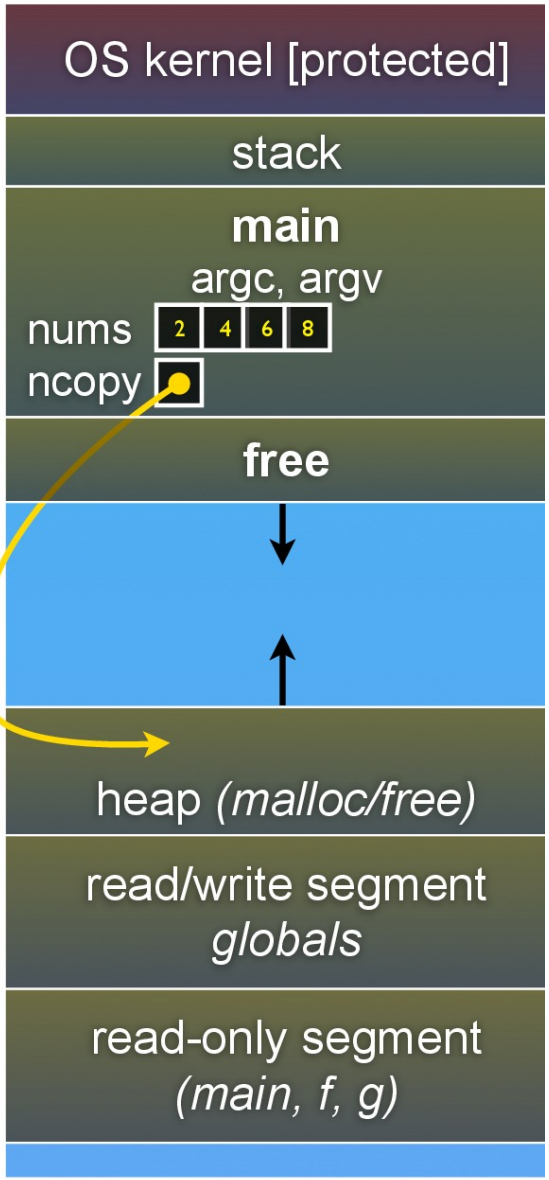
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

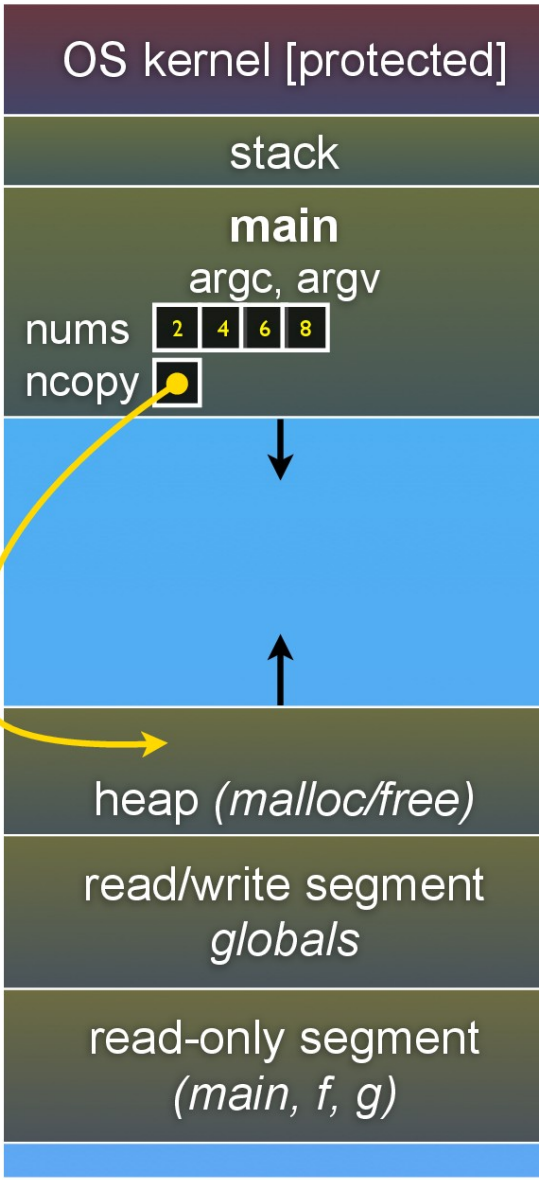
arraycopy.c



Heap/stack in action

```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 nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

arraycopy.c



Memory corruption

- There are all sorts of ways to use memory irresponsibly in C
  - Writing things where they don't belong or using allocation functions incorrectly can cause "memory corruption"

```
void use_memory_irresponsibly() {
    int a[2];
    int *b = malloc(2 * sizeof(int));
    int *c;

    // Many ways to corrupt memory...

    a[2] = 5;       // assign past the end of an array
    a[0] += 2;      // assume memory is zeroed by malloc
    c = b + 3;      // mess up pointer arithmetic
    free(&a[0]);    // free something that wasn't malloced
    free(b);        // (this one is OK)
    free(b);        // free something more than once (!)
    b[0] = 5;       // use a pointer after it is freed
    // ... and many more!
}
```

Memory leak

- Happens when you allocate memory but forget to free it
  - Can be tricky to find
  - Sometimes hidden by a control statement like `return`, `break`, or `continue`
- Result: program's memory usage keeps growing
  - Slows down over time
  - Might be killed by the OS for using too much memory!

```c
int string_length_leak() {
    char *str = malloc(256);

    scanf("%255s", str);
    return strlen(str);
}
```