CSE 333

Lecture 5 - malloc, free, struct, typedef

Hal Perkins

Department of Computer Science & Engineering

University of Washington

Administrivia

HW1 is due in 8 days: Thursday, July 5, 11 pm

- need to make steady progress
- see course overview web page for the late policy - but try not to use late days this early in the quarter

Be sure to check out the course discussion board

- Suggestion: “mail me whenever a new posting appears” setting if you don’t want to browse regularly
- Please post questions here, rather than emailing me or the TAs, so that everybody benefits from answers

Administrivia

We *highly* recommend doing the 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 problems for the rest of the course

Today's goals:

- understand heap-allocated memory
  - malloc( ), free( )
  - memory leaks
- quick intro to structs and typedef

Memory allocation

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

```
// a global variable
int counter = 0;

int main(int argc, char **argv) {
  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
  return 0;
}
```

`a,x,y` are ***automatically*** **allocated**

- allocated when function is called
- deallocated when function returns

We need more flexibility

Sometimes we want to allocate memory that:

- persists across multiple function calls but for less than the lifetime of the program
- is too big to fit on the stack
- is allocated and returned by a function and its size is not known in advance to the caller

```
// (this is pseudo-C-code)
char *ReadFile(char *filename) {
  int   size = FileSize(filename);
  char *buffer = AllocateMemory(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 runtime allocates it, perhaps with help from OS
- dynamically allocated memory persists 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 causes headaches

C and malloc

*variable = (type *) malloc(size in bytes);*

malloc allocates a block of memory of the given size

- returns a pointer to the first byte of that memory
  - malloc returns **NULL** if the memory could not be allocated
- you should assume the memory initially contains garbage
- normally 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;
arr[0] = 5.1;  // etc.
```

C and calloc

*variable = (type *) calloc(num copies, size in bytes);*

Like malloc, but also zeroes out the block of memory

- helpful for shaking out bugs
- slightly slower; preferred for non-performance-critical code
- malloc and calloc are found in *stdlib.h*

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

Deallocation

free(pointer);

Releases the memory pointed-to by the pointer

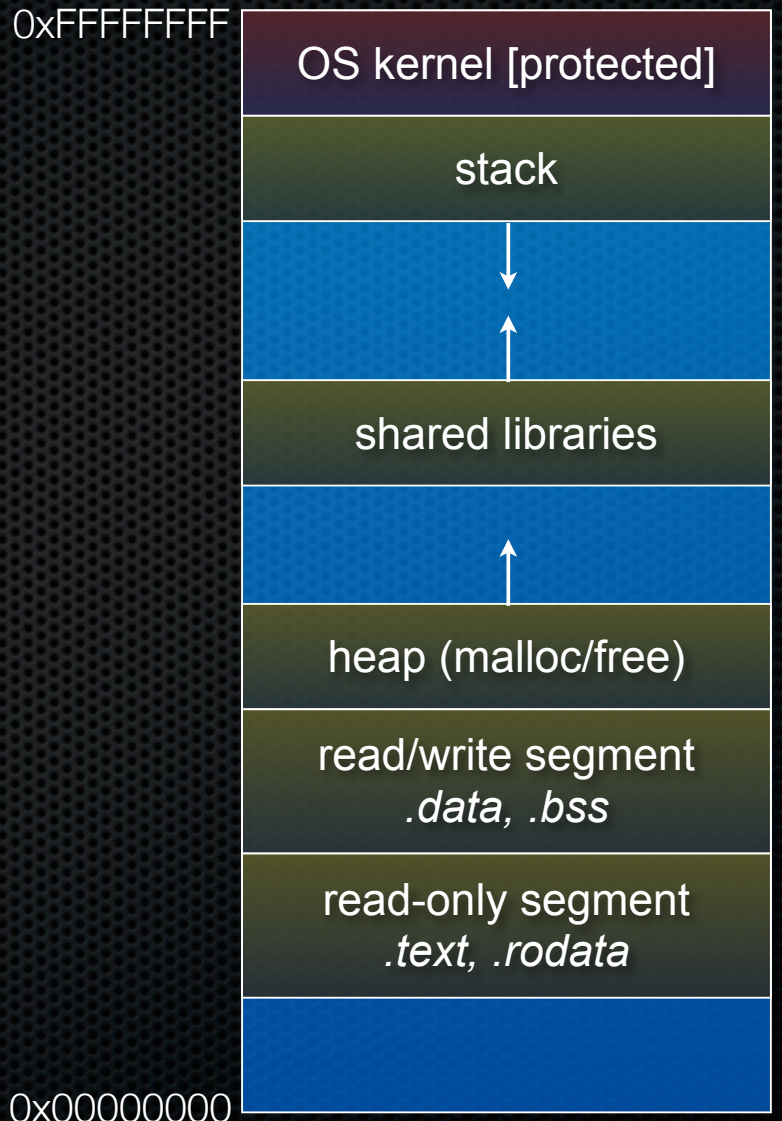
- pointer must point to the first byte of heap-allocated memory
  - i.e., something previously returned by malloc( ) or calloc( )
- after free( )'ing a block of memory, that block of memory might be returned in some future malloc( ) / calloc( )
- common habit: set a pointer to NULL after freeing it
  - (guards against some dangling pointer bugs, but code should be ok without it)

```
long *arr = (long *) calloc(10, sizeof(long));
if (arr == NULL)
  return errcode;
// .. do something ..
free(arr);
arr = NULL;
```

Heap

The heap (aka “free store”)

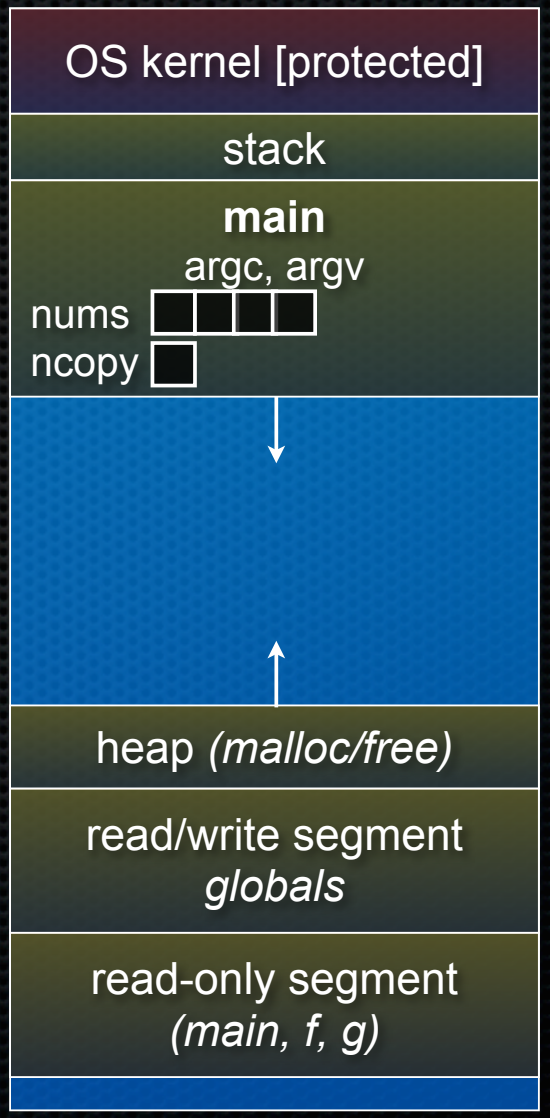
- is a large pool of unused memory that is used for dynamically allocated data
- malloc allocates chunks of data in the heap, free deallocates data
- malloc maintains book-keeping data in the heap to track allocated blocks



Heap + stack

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

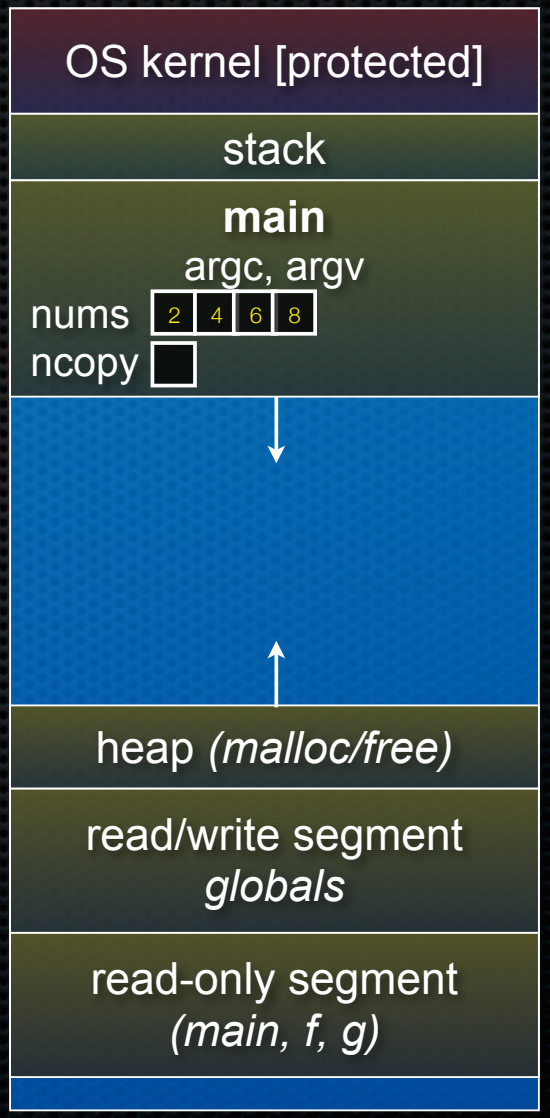
arraycopy.c



Heap + stack

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

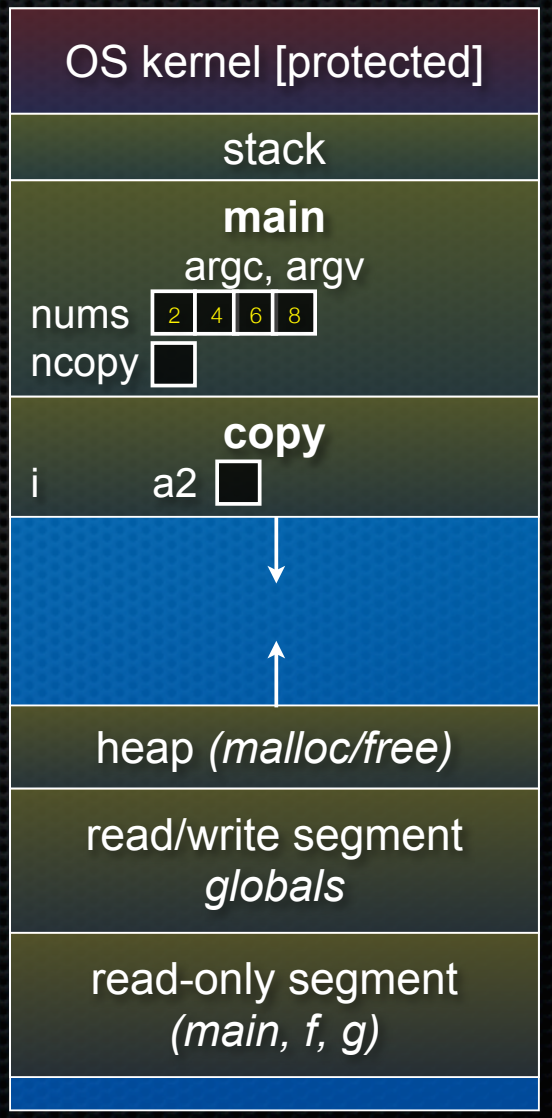
arraycopy.c



Heap + stack

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

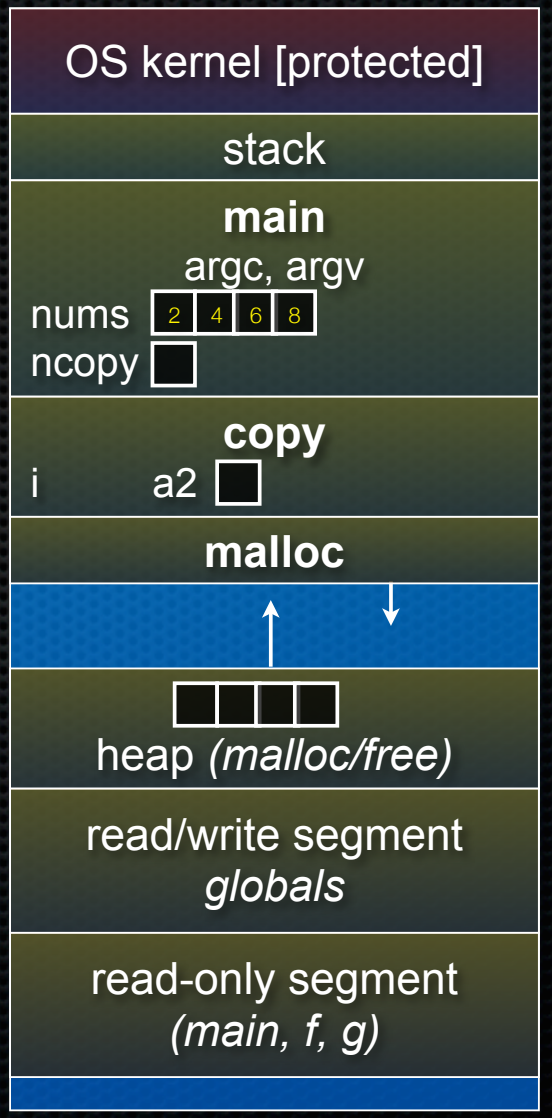
arraycopy.c



Heap + stack

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

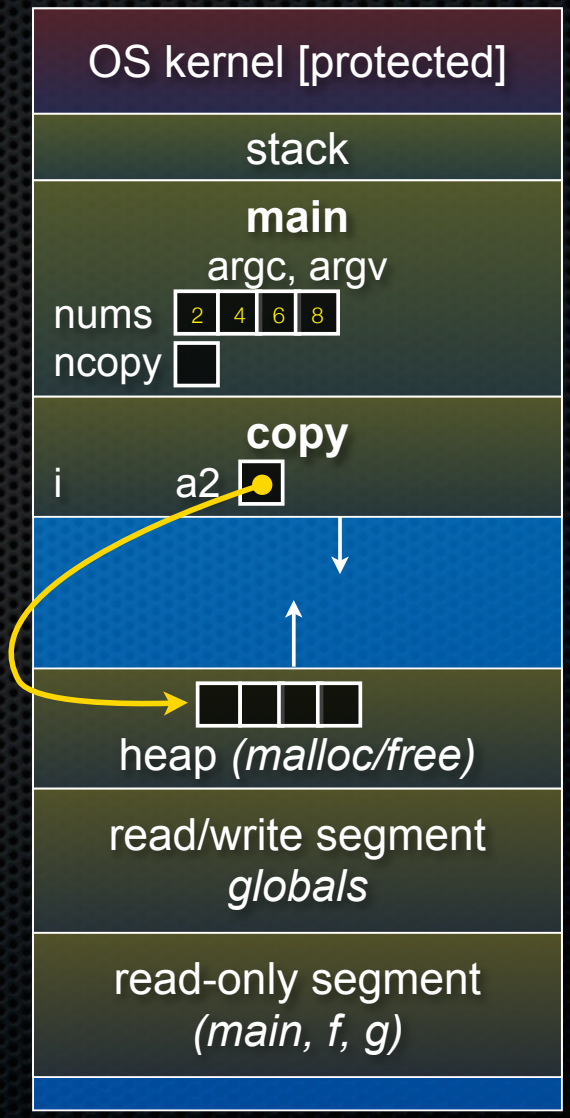
arraycopy.c



Heap + stack

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

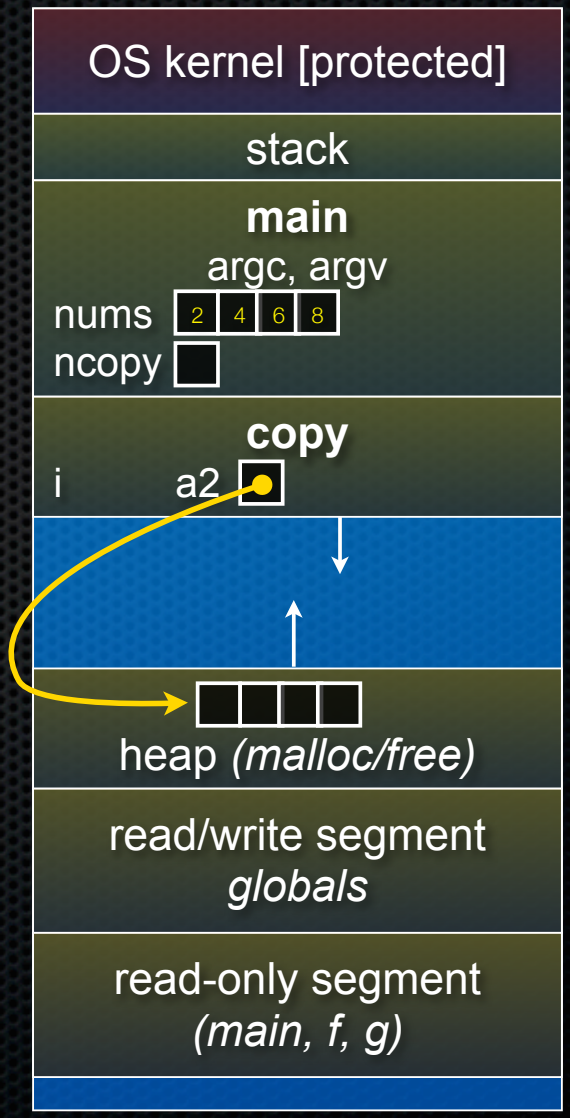
arraycopy.c



Heap + stack

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

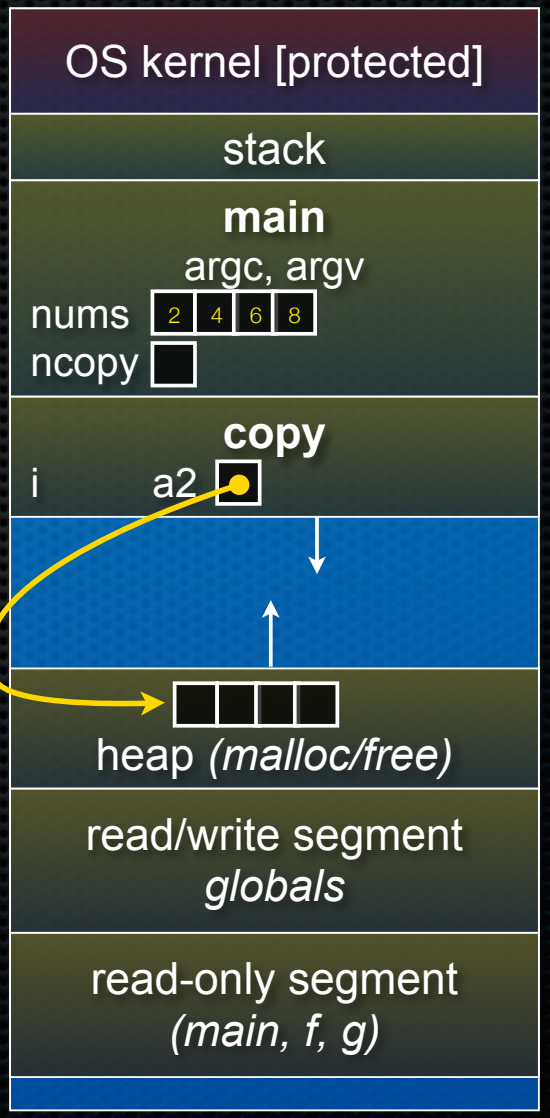
arraycopy.c



Heap + stack

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

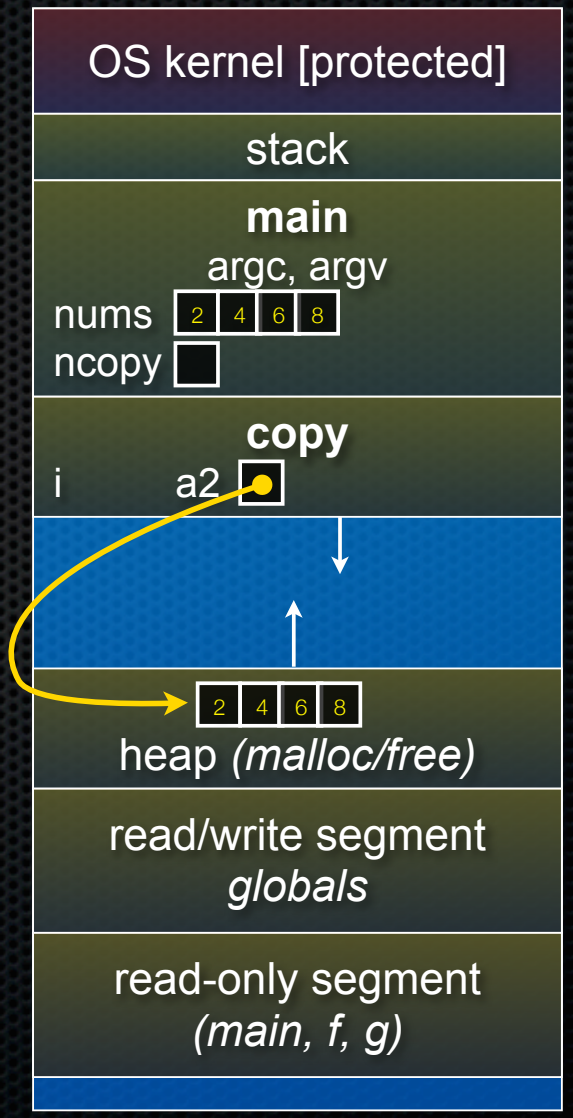
arraycopy.c



Heap + stack

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

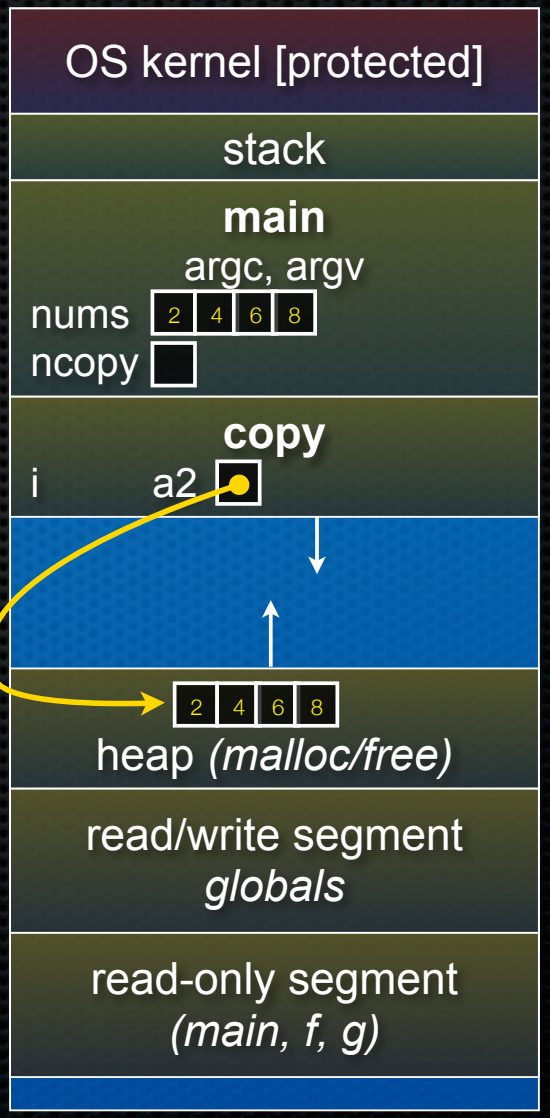
arraycopy.c



Heap + stack

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

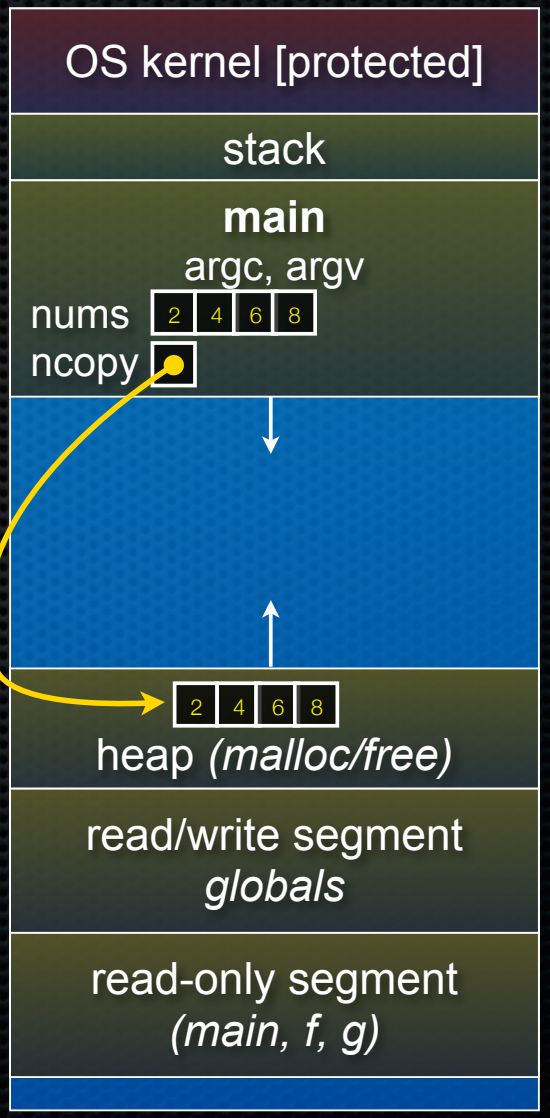
arraycopy.c



Heap + stack

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

arraycopy.c



Heap + stack

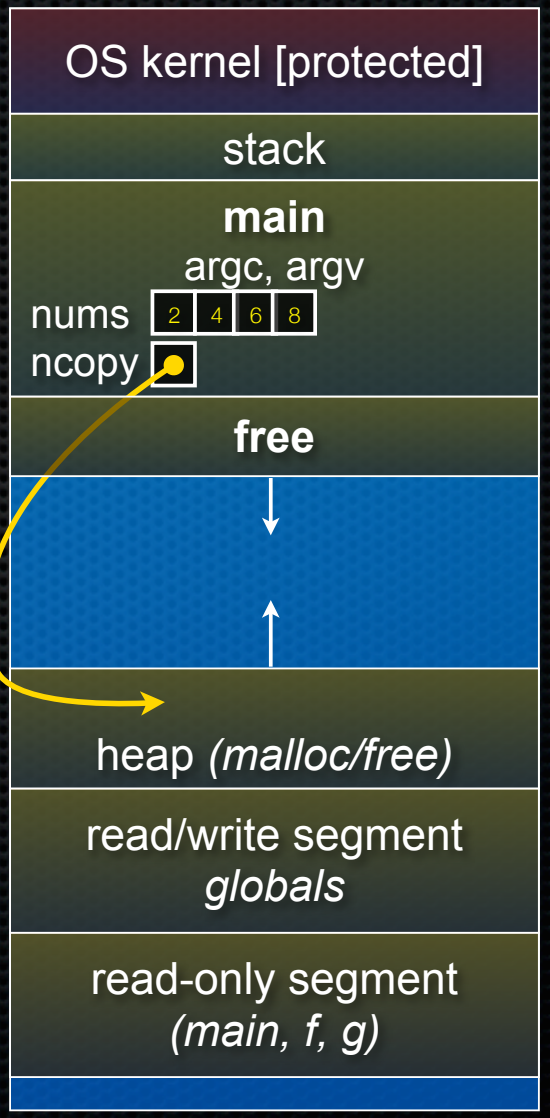
```
#include <stdlib.h>

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

arraycopy.c

Heap + stack

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

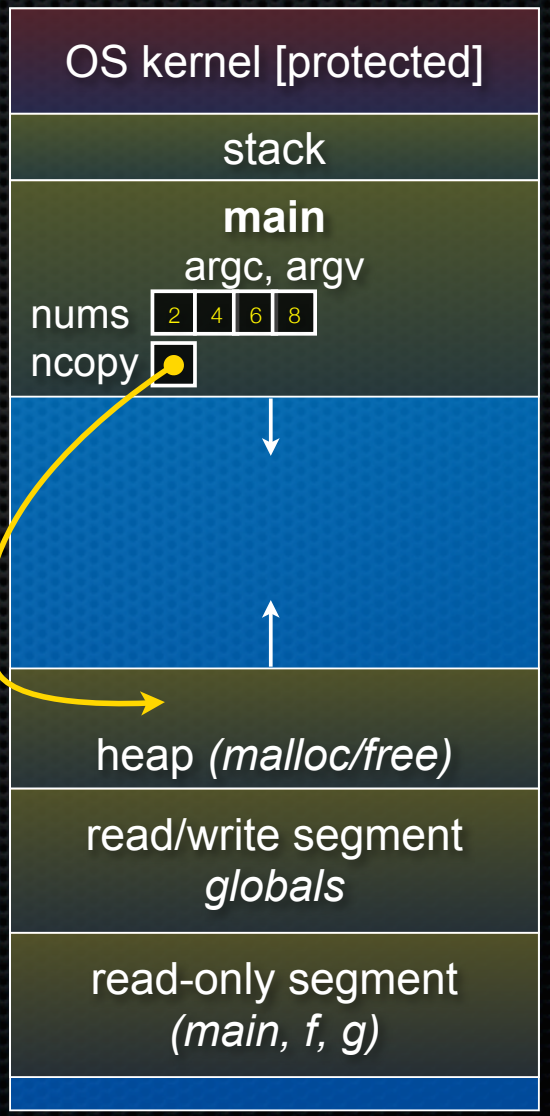
arraycopy.c



Heap + stack

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

arraycopy.c



NULL

NULL: a guaranteed-to-be-invalid memory location

- in C on Linux:
  - NULL is 0x00000000
  - an attempt to deference NULL causes a segmentation fault
- that's why it's useful to NULL a pointer after you have free( )'d it
  - it's better to have a segfault than to corrupt memory!

segfault.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int *p = NULL;
  *p = 1;  // causes a segmentation fault
  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 = (int *)malloc(2*sizeof(int)), *c;

  a[2] = 5;   // assign past the end of an array
  a[0] += 2;  // assume malloc zeroes 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 free()'d pointer

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

memcorrupt.c

Memory leak

A memory leak happens when code fails to deallocate dynamically allocate memory that will no longer be used

```
// assume we have access to functions FileLen,
// ReadFileIntoBuffer, and NumWordsInString.

int NumWordsInFile(char *filename) {
  char *filebuf = (char *) malloc(FileLen(filename)+1);
  if (filebuf == NULL)
    return -1;

  ReadFileIntoBuffer(filename, filebuf);

  // leak! we never free(filebuf)
  return NumWordsInString(filebuf);
}
```

Implications of a leak?

Your program's *virtual memory* footprint will keep growing

- for short-lived programs, this might be OK
- for long-lived programs, this usually has bad repercussions
  - might slow down over time (VM thrashing – see cse451)
    - *potential "DoS attack" if a server leaks memory*
  - might exhaust all available memory and crash
  - other programs might get starved of memory
- in some cases, you might prefer to leak memory than to corrupt memory with a buggy free( )

Structured data

struct tname {
* type name;*
* type name;*
* ...*
* type name;*
};

```
// The following defines a new structured
// data type called a "struct Point".
struct Point {
  float x, y;
};

struct Point origin = {0.0, 0.0};
```

struct: a C type that contains a set of fields

- similar to a Java class, but without methods / constructors
- instances can be allocated on the stack or heap
- useful for defining new structured types of data
- *tname* is not a full-fledged type like int - it's a struct tag

Using structs

Use "." to refer to fields in a struct

Use "->" to refer to fields through a pointer to a struct

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

int main(int argc, char **argv) {
  int i = 1;
  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
  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

typedef type name;

Allows you to define a new type whose name is *name*

- especially useful when dealing with structs

```
// make "superlong" be a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "Point" be a synonym for "struct Point { ... }"
typedef struct Point {
  superlong x;
  superlong y;
} Point;

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

structs as arguments

```
// Point is a (struct point_st)
// PointPtr is a (struct point_st *)
typedef struct point_st {
  int x, y;
} Point *PointPtr, **PointPtrPtr;

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);
  DoubleXWorks(&a);
  printf("(%d,%d)\n", a.x, a.y);
  return 0;
}
```

structarg.c

structs are passed by value

- like everything else in C
  - entire structure is copied
- to pass-by-reference, pass a pointer to a struct

You can return structs

```
// a complex number is a + bi
typedef struct complex_st {
  double real;  // real component (i.e., a)
  double imag;  // imaginary component (i.e., b)
} Complex, *ComplexPtr;

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

  retval.real = x.real + y.real;
  retval.imag = x.imag + y.imag;
  return retval;  // returns a copy of retval
}

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;
}
```

complexstruct.c

Dynamically allocated structs

You can malloc and free structs, as with other types

- sizeof is particularly helpful here

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

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

Don’t repeat yourself…

Suppose we have many similar chunks of code:

```
for (k = 0; k < size; k++)
  list[k] = dbl(list[k]);
...
for (k = 0; k < size; k++)
  list[k] = incr(list[k]);
```

Better: extract duplicate code and make a function

A function parameter

```
void map(int a[], int len, ??? f) {
  for (int k = 0; k < len; k++)
    a[k] = (*f)(a[k]);
}
...

map(list, size, &dbl);
...
map(list, size, &incr);
```

What is the type of f?

- Roughly: “pointer to function with int parameter and int result”
- How do we say that in C?

Function parameter type

```
void map(int a[], int len, int (*f) (int)) {
  for (int k = 0; k < len; k++)
    a[k] = (*f)(a[k]);
}
...

map(list, size, &dbl);
...
map(list, size, &incr);
```

funcparm.c

Read the declaration from inside out:

- f is a pointer that can be dereferenced to get a function that takes an int parameter and returns an int result

Typedef for function parameters

Better: use typedef to give a name to the function ptr type

- Only need to get it right once

```
typedef int (*PFII) (int);

void map(int a[], int len, PFII f) {
  for (int k = 0; k < len; k++)
    a[k] = (*f)(a[k]);
}
...

map(list, size, &dbl);
...
map(list, size, &incr);
```

funcparmt.c

Uses for function pointers

Call-backs in system code and elsewhere

- e.g., code that gets called when an OS or UI event happens

Google map-reduce, databases, other frameworks

- framework calls user code for application-specific work

Functional programming

- elegant, compact code - take CSE 341!

& more...

Exercise 1

Write and test a program that defines:

- a new structured type Point
  - represent it with floats for the x, y coordinate
- 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/returns the area of a Rectangle
- a function that tests whether a Point is in a Rectangle

Exercise 2

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

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

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

Implement AllocSet( ), 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

See you on Friday!