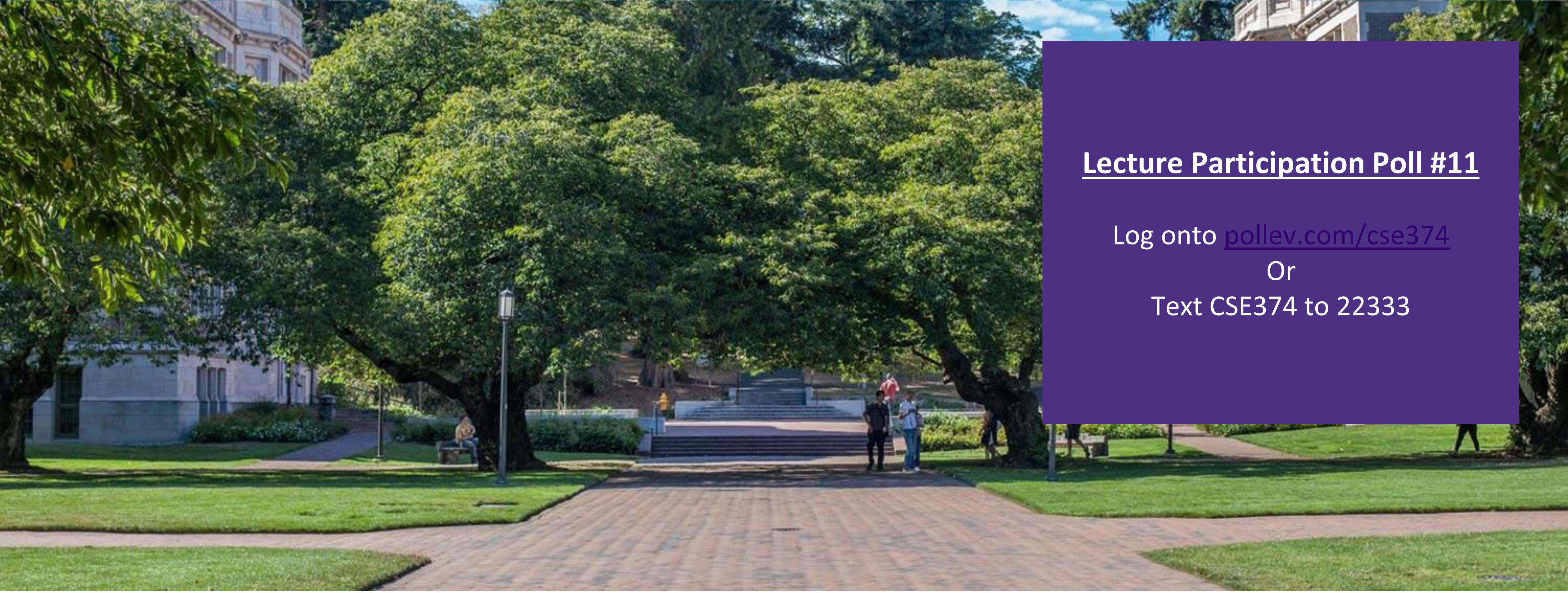


Lecture Participation Poll #11

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 11: Dynamic Memory Allocation Continued…

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Sorry Kasey is behind in email! Will be digging through this weekend and also posting a bunch more materials to the course website

Sorry about HW shenanigans -_-

HW1 Deadline pushed to end of quarter – Dec 3rd

HW 2 Deadline Extended due to Klaatu outage

- FYI autograder is still assessing old versions of course pages, it runs your code on the old versions so it still functions properly in assessing your code correctness, but if your code has differences the output is confusing

HW3 due next thursday

Midterm Topics

Midterm will be in class on Friday 10/29, paper exam
Midterm will be open paper note, closed electronic devices

Bash Commands

- file manipulation
- variables & aliases
- redirects & pipes
- scripting
- Regex
- HW1 & HW2

C Programming

- Pointers
- Arrays & Strings
- Automatic, static and dynamic memory allocation
- Struct basics (Monday)

Memory Allocation

- **Allocation** refers to any way of asking for the operating system to set aside space in memory
- How much space? Based on variable type & your system
  - to get specific sizes for your system use "sizeof(<datatype>)" function in stdlib.h
- Global Variables – **static** memory allocation
  - space for global variables is set aside at compile time, stored in RAM next to program data, not stack
  - space set aside for global variables is determined by C based on data type
  - space is preserved for entire lifetime of program, never freed
- Local variables – **automatic** memory allocation
  - space for local variables is set aside at start of function, stored in stack
  - space set aside for local variables is determined by C based on data type
  - space is deallocated on return

Type	Storage Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,786 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615
float	4 bytes	1.2E-38 to 3.4E+38
double	8 bytes	2.3E-308 to 1.7E+308
long double	10 bytes	3.4E-4932 to 1.1E+4932

* pointers require space needed for an address – dependent on your system - 4 bytes for 32-bit, 8 bytes for 64-bit

https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-and-C.html

Does this always work?

- Static and automatic memory allocation – memory set aside is known at runtime
  - Fast and easy to use
  - partitions the maximum size per data type – not efficient
  - life of data is automatically determined – not efficient
- What if we don't know how much memory we need until program starts running?

```
char* ReadFile(char* filename)
{
   int size = GetFileSize(filename);
   char* buffer = AllocateMem(size);

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

You don't know how big the filesize is

Dynamic Allocation

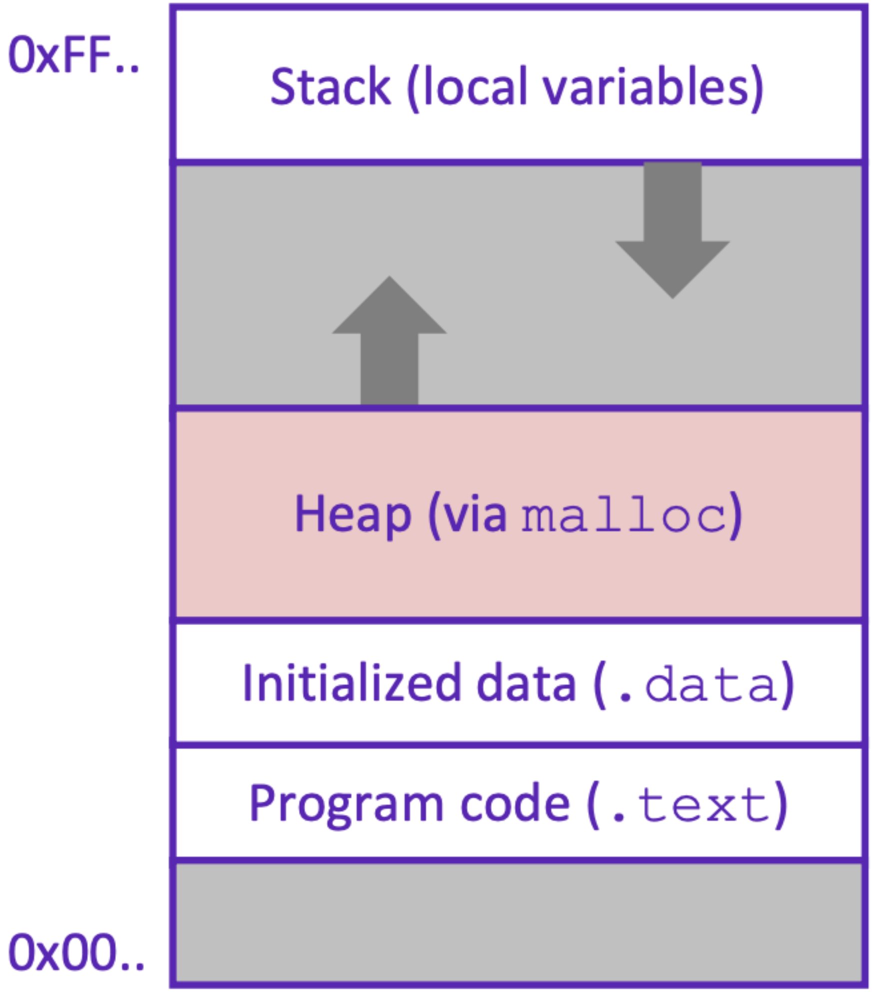
- Situations where static and automatic allocation aren't sufficient
  - Need memory that persists across multiple function calls
    - Lifetime is known only at runtime (long-lived data structures)
  - Memory size is not known in advance to the caller
    - Size is known only at runtime (ie based on user input)
- Dynamically allocated memory persists until:
  - A garbage collector releases it (automatic memory management)
    - Implicit memory allocator, programmer only allocates space, doesn't free it
    - "new" in Java, memory is cleaned up after program finishes
    - Your code explicitly deallocates it (manual memory management)
    - C requires you manually manage memory
    - Explicit memory allocation requires the programmer to both allocate space and free it up when finished
    - "malloc" and "free" in C
- Memory is allocated from the heap, not the stack
  - Dynamic memory allocators acquire memory at runtime

Storing Program Data in the RAM

- When you trigger a new program the operating system starts to allocate space in the RAM
  - Operating System will default to keeping all memory for a program as close together within the ram addresses as possible
  - Operating system manages where exactly in the RAM your data is stored
    - Space is first set aside for program code (lowest available addresses)
    - Then space is set side for initialized data (global variables, constants, string literals)
    - As program runs…
      - When the programmer manually allocates memory for data it is stored in the next available addresses on top of the initialized data, building upwards as space is needed
      - When the program requires local variables they are stored in the empty space at top of RAM, leaving space between stack and heap
      - When the space between the stack and heap is full – crash (out of memory)

The heap is a large pool of available memory set aside specifically for dynamically allocated data

Address Space Visualization



Allocating Memory in C with malloc()

- `void* malloc(size_t size)`
  - allocates a continuous block of "`size`" bytes of **uninitialized** memory
  - Returns `null` if allocation fails or if `size == 0`
    - Allocation fails if out of memory, very rare but always check allocation was successful before using pointer
  - `void*` means a pointer to any type (int, char, float)
    - `malloc` returns a pointer to the beginning of the allocated block
- `var = (type*) malloc(sizeInBytes)`
  - Cast void* pointer to known type
  - Use `sizeof(type)` to make code portable to different machines
- `free` deallocates data allocated by `malloc`
- Must add `#include <stdlib.h>`
- Variables in C are uninitialized by default
  - No default "0" values like Java
  - Invalid read – reading from memory before you have written to it

```
//allocate an array to store 10 floats
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
{
   return ERROR;
}
printf("%f\n", *arr) // Invalid read!
<add something to array>
<print f again, now it's ok>
```

calloc()

```
var = (type*) calloc(numOfElements, bytesPerElement);
```

- Like malloc, but also initializes the memory by filling it with 0 values
- Slightly slower, but useful for non-performance critical code
- Also in stdlib.h

```
//allocate an array to store 10 doubles
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL)
{
   return ERROR;
}
printf("%f\n", arr[0]) // Prints 0.00000
```

realloc()

- void* realloc(void* p, size_t size)
  - creates a new allocation with given size, copies the contents of p into it and then frees p
  - saves a few lines of code
  - can sometimes be faster due to allocator optimizations
  - part of stdlib.h

Freeing Memory in C with free()

- `void free(void* ptr)`
  - Released whole block of memory stored at location ptr to pool of available memory
  - ptr must be the address originally returned by malloc (the beginning of the block) otherwise system exception raised
  - ptr is unaffected by free
    - Set pointer to NULL after freeing it to deallocate that space too
  - Calling free on an already released block (double free) is undefined behavior – best case program crashes
  - Rule of thumb: for every runtime call to malloc there should be one runtime call to free
  - if you lose all pointers to an object you can no longer free it – memory leak!
    - be careful when reassigning pointers
    - this is usually the cause of running out of memory- unreachable data that cannot be freed
  - if you attempt to use an object that has been freed you hit a dangling pointer
  - all memory is freed once a process exits, and it is ok to rely on this in many cases

```
//allocate an array to store 10 floats
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
{
   return ERROR;
}
for (int i = 0; i < size*num; i++)
{
   arr[i] = 0;
}
free(arr);
arr = NULL; // Optional
```

Example

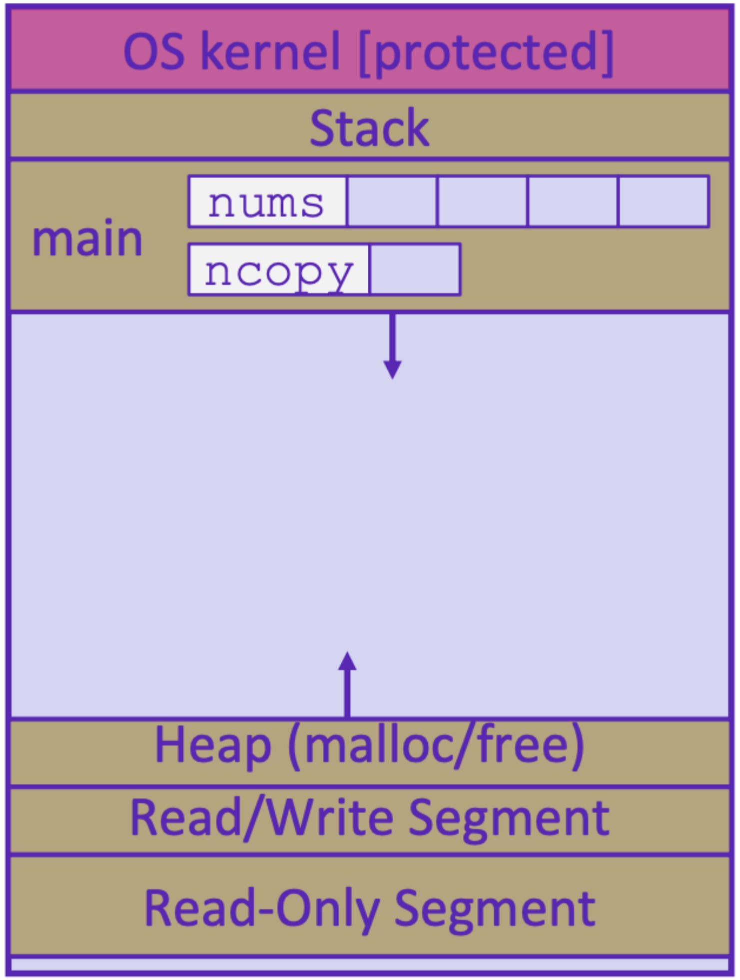
```
void foo(int n, int m)
{
   int i, *p; // declare local variables
   p = (int*) malloc(n*sizeof(int)); //allocate block of n ints
   if (p == NULL) // check for allocation error
   {
      perror("malloc"); //prints error message to stderr
      exit(0);
   }
   for (i=0; i<n; i++) // initialize int array
      p[i] = i;
   p = (int*) realloc(p, (n+m)*sizeof(int)); // add space for m at end of p block
   if (p == NULL) // check for allocation error
   {
      perror("realloc");
      exit(0);
   }
   for (i=n; i<n+m; i++) // initialize new space at back of array
      p[i] = i;
   for (i=0; i<n+m; i++) // print out array
      printf("%d\n", p[i]);
   free(p); // free p, pointer will be freed at end of function
}
```

Example: 1 – initialized data

```
#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 your copy!
   free(ncopy);
   return EXIT_SUCCESS;
}
```

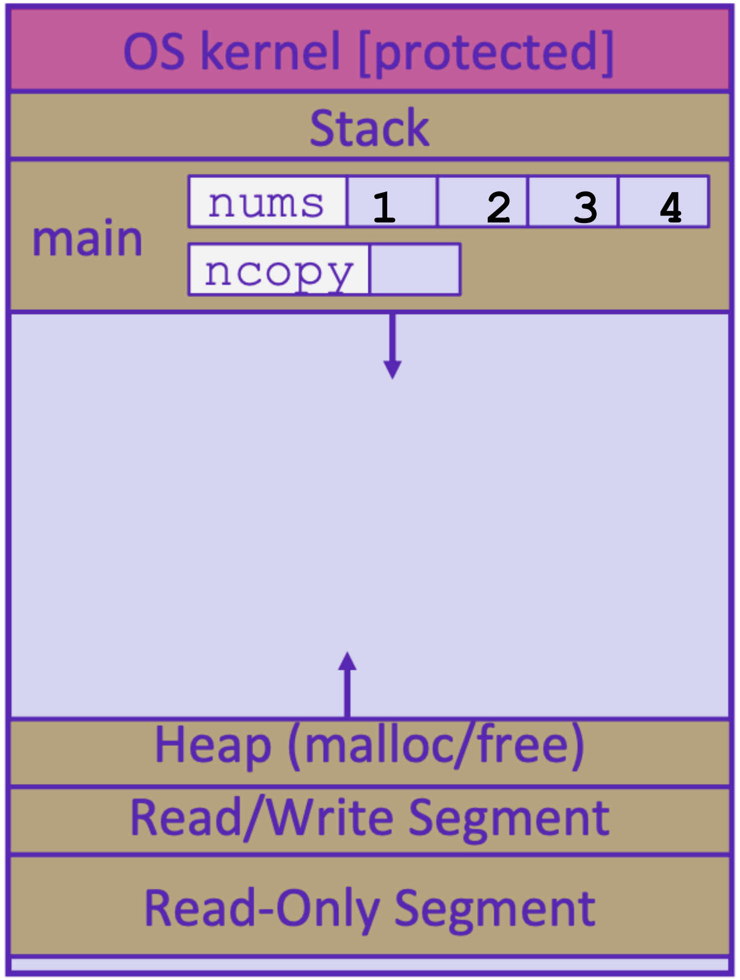


Example: 2 – main local variable in stack

```
#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 your copy!
   free(ncopy);
   return EXIT_SUCCESS;
}
```

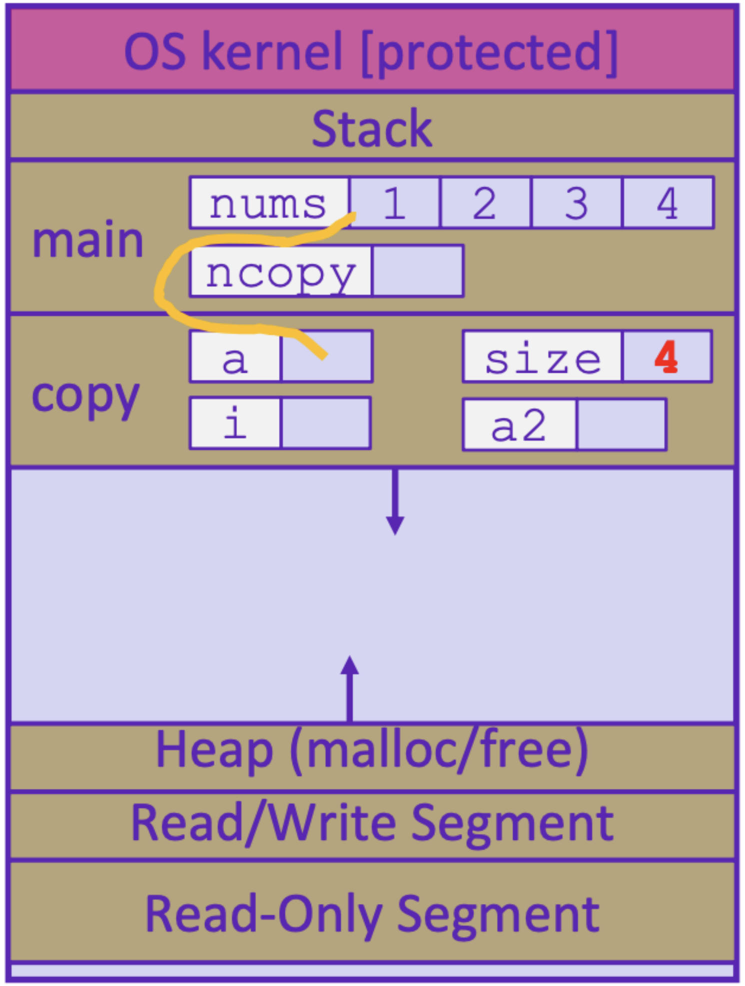


Example: 3 – copy local variables in stack

```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 your copy!
   free(ncopy);
   return EXIT_SUCCESS;
}
```

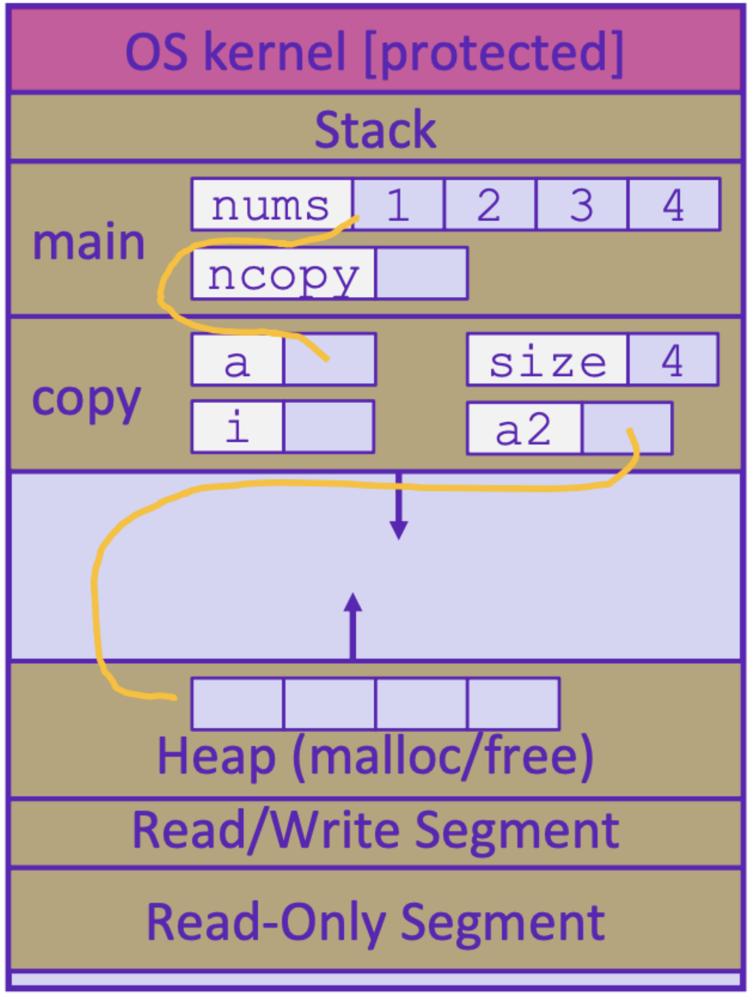


Example: 4 – malloc space for int array

```
#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 your copy!
   free(ncopy);
   return EXIT_SUCCESS;
}
```

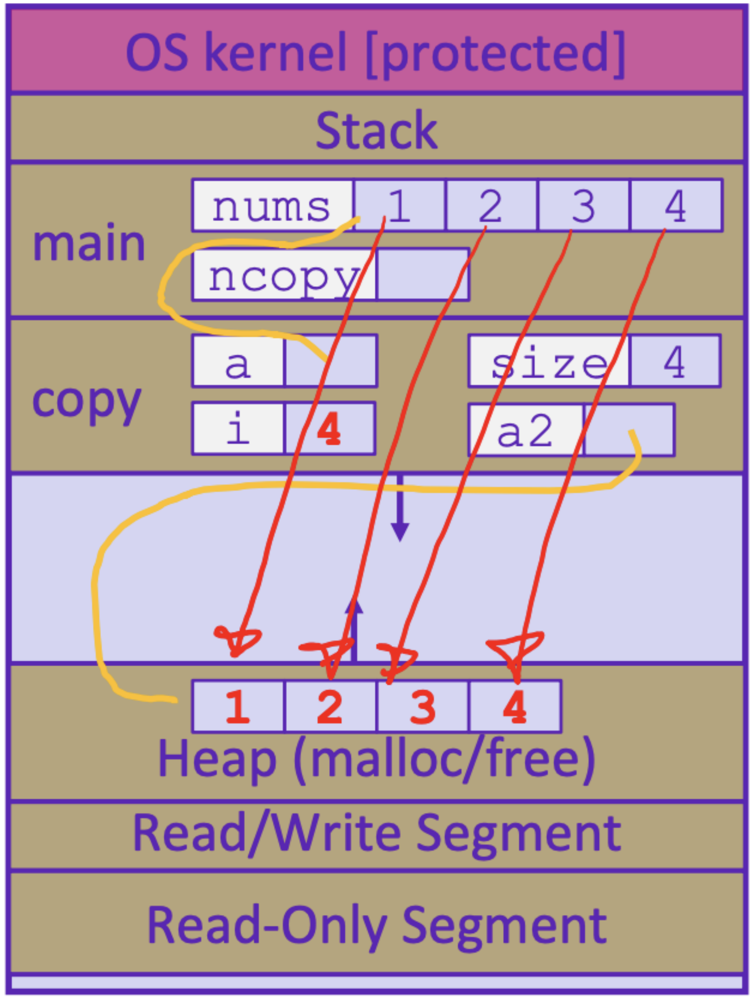


Example: 5 – fill available space from local var

```
#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 your copy!
   free(ncopy);
   return EXIT_SUCCESS;
}
```

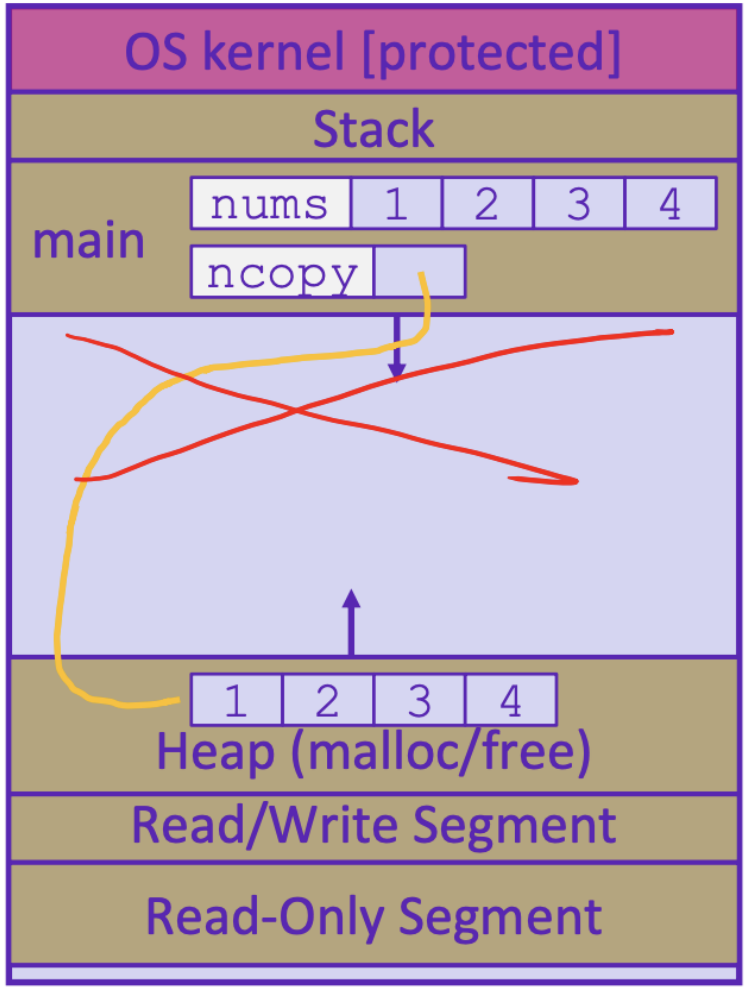


Example: 6 – finish copy and free stack space

```
#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 your copy!
   free(ncopy);
   return EXIT_SUCCESS;
}
```

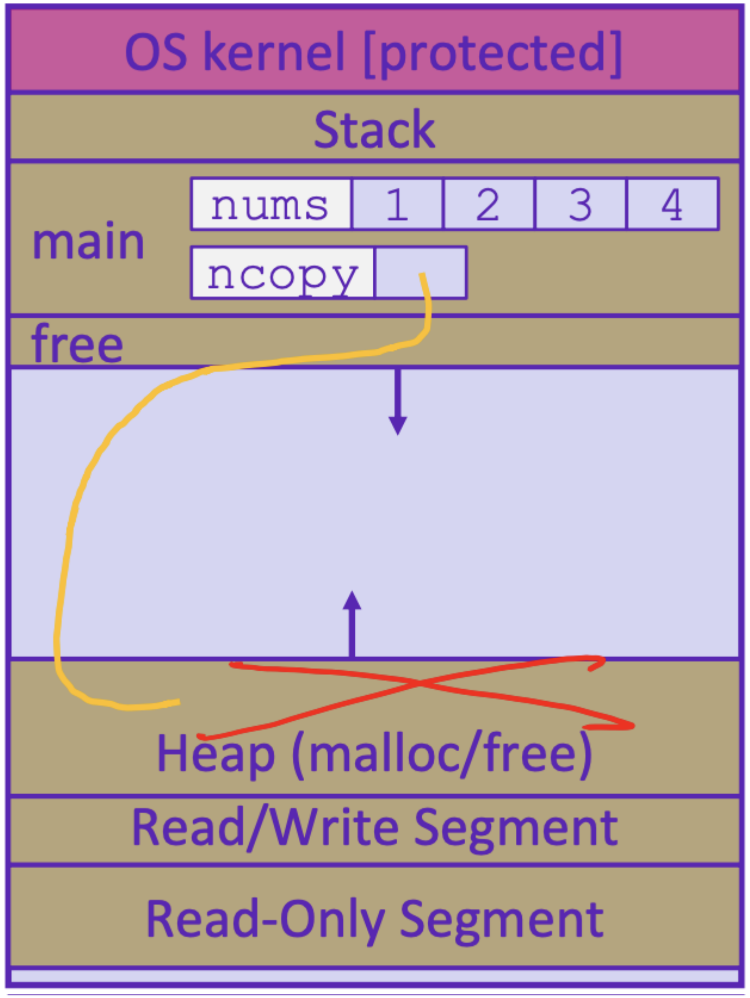


Example: 7 – free ncopy from heap

```
#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 your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```



Memory Leak

- A memory leak occurs when code fails to deallocate dynamically-allocated memory that is no longer used
  - Caused by forgetting to call free() on a malloc'd block of memory or losing a pointer to a malloc-d block
  - Program's memory will keep growing
- What's the problem?
  - Short-lived program might not be an issue, all memory is deallocated when program ends
  - Long-lived programs might slow down over time, exhaust all available memory and crash or starve other programs of memory

Common Memory Errors

```
x = (int*)malloc(M*sizeof(int));
free(x);
y = (int*)malloc(M*sizeof(int));
free(x);
```

Double free and Forgetting to free memory "memory leak"

```
int x[] = {1, 2, 3};
free(x);
```

x is a local variable stored in stack, cannot be freed

```
char** strings = (char**)malloc(sizeof(char)*5);
free(strings);
```

Mismatch of type - wrong allocation size

```
x = (int*)malloc(M*sizeof(int));
free(x);
y = (int*)malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i];
```

Accessing freed memory

Common Memory Errors

```
#define LEN 8
int arr[LEN];
for (int i = 0; i <= LEN; i++)
   arr[i] = 0;
```

Out of bounds access

```
int* foo()
{
   int val = 0;
   return &val;
}
```

dangling pointer

```
long val;
printf("%d", &val);
```

Dereferencing a non-pointer

```
int foo()
{
   int* arr = (int*)malloc(sizeof(int)*N);
   read_n_ints(N, arr);
   int sum = 0;
   for (int i = 0; i < N; i++)
      sum += arr[i];
   return sum;
}
```

memory leak – failing to free memory

```
int sum_int(int* arr, int len)
{
   int sum;
   for (int i = 0; i < len; i++)
      sum += arr[i];
   return sum;
}
```

Reading memory before allocation

Finding and Fixing Memory Errors

- Valgrind is a tool that simulates your program to find memory errors
  - it can detect all of the errors we've discussed so far!
  - catches pointer errors during execution
  - prints summary of heap usage, including details of memory leaks

```
valgrind [options] ./myprogram arg1 arg2
```

Useful option: `--leak-check=full`

Appendix

Pointers to pointers

Levels of pointers make sense:

I.e.: `argv, *argv, **argv`

Or: `argv, argv[0], argv[0][0]`

But

`&(&p)` doesn't make sense

```
void f(int x) {
    int*p = &x;
    int**q = &p;
    // x, p, *p, q, *q, **q
}
```

Integer, pointer to integer, pointer to pointer to integer

`&p` is the address of '`p`',

`&(&p)` would be the address of the address of `p`, but that value isn't stored separately anywhere and doesn't have an address

Try using `printf ("The address of x is %p\n", &x);`

Arrays again

"A reference to an object of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T."

http://c-faq.com/aryptr/aryptrequiv.html

Right: x is the array, which decays to a pointer to an int and &x returns a pointer to the entire array.

```
void f1(int* p) {  // takes a pointer
  *p = 5;
}
int* f2() {
    int x[3];      // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5; // address of x, points to
                 // same place but different T
    *x = 5;        // put value at location x
    *(x+0) = 5;  // Also put value at x
    f1(x);
    f1(&x);      // wrong - watch types!
    x = &x[2]; // No!  X isn't really a pointer
    int *p = &x[2];
    return x; // correct type, but is a
              // dangling pointer
}
```

errno

- How do you know if an error has occurred in C?
  - no exceptions like Java
- usually return a special error value (NULL, -1)
- stdlib functions set a global variable called errno
  - check errno for specific error types
  - if (errno == ENOMEM) // allocation failure
  - perror("error message") prints to stderr

C Garbage Collector

- garbage collection is the automatic reclamation of heap-allocated memory that is never explicitly freed by application
  - used in many modern languages: Java, C#, Ruby, Python, Javascript etc…
  - "conservative" garbage collectors do exist for C and C++ but cannot collect all garbage
- Data is considered "garbage" if it is no longer reachable
  - lost pointers to data (Like a dropped link list node in Java)
  - memory allocator can sometimes get help from the compiler to know what data is a pointer and what is not