CSE 374: Programming Concepts and Tools

Eric Mullen
Spring 2017
Lecture 13: Dynamic Memory Allocation
Administrivia

• Midterm on Friday (During classtime)
  • Review Session Tomorrow, 4-6pm, CSE 203
  • Topics: Everything up through hw4 and last lecture
  • Starting Today: not on the midterm (will be on the final)

• Homework 4: Out already, due Thursday at midnight
  • Make sure to READ THE ASSIGNMENT
Leftover Demos

- strings.c
- reverse.c
Pointers

- A Pointer is *just* the address of something
- The actual thing does **not** belong to the pointer
```c
void zero_array(int[] p, int l) {
    for (int i = 0; i < l; i++) {
        p[i] = 0;
    }
}

int main() {
    int size = 48;
    int arr[48];
    zero_array(arr, 48);
    printf("arr[37]: %d\n", arr[37]);
    return 0;
}
```
int* make_array(int l) {  
  //how do we write this?
}

int main() {
  int size; scanf("%d", &size);
  int* arr = make_array(size);
  if (size > 37) {
    printf("arr[37]: %d\n", arr[37]);
  }
  return 0;
}
“Owning” Memory

• You own stack memory until you return from that function

• No Guarantee what will happen if you read/write memory you don’t own!

• No Guarantee that a pointer points to memory you own! You must make sure you know this before you follow a pointer
The Heap

• Memory from the heap is memory you ask for in a different way.

• You own memory on the heap until you decide to release it

• Extremely useful, but easy to mess up
Asking for Heap Memory

- **malloc** is the primary function to ask for heap memory
- It takes one argument (the number of bytes to allocate), and returns a pointer to the newly allocated memory (or **NULL**, if it fails for some reason)
- If **malloc** succeeds, you have **no idea** what that memory contains
- You own that memory until you decide to explicitly give it up (with **free**)
void* x = malloc(8);

int* x = (int*)malloc(sizeof(int));

int* x = (int*)malloc(sizeof(int)*32);

int** x = (int**)malloc(sizeof(int*));

int** x = (int**)malloc(sizeof(int*)*18);
Good Practice

• **Always** cast the result of `malloc` to the type you want
  
  • This will be a pointer type

• **Always** use `sizeof` to calculate the number of bytes you want

• **Always** free memory exactly once when done with it
```c
int* make_array(int l) {
    int *arr =
        (int*)malloc(sizeof(int)*l);
    for (int i=0; i<l; i++) {
        arr[i] = 0;
    }
    return arr;
}

int main() {
    int size; scanf("%d", &size);
    int* arr = make_array(size);
    if (size > 37) {
        printf("arr[37]: %d\n",
                arr[37]);
    }
    free(arr);
    return 0;
}
```
Be careful

- **ALWAYS** check the return value of malloc for NULL
  - don’t do anything else first
- **NEVER** call `free` more than once on the same pointer
- **NEVER** dereference a pointer after calling `free`
- **NEVER** access memory past the end
Common Errors

```c
int* x = (int*)malloc(sizeof(int) * 32);
x = NULL; // memory leak

int* x = (int*)malloc(sizeof(int) * 32);
free(x);
free(x); // double free

int* x = (int*)malloc(sizeof(int) * 32);
free(x);
*x = 4; // use after free
```
How to find mistakes

• Don’t make mistakes (think before you code)

• Draw a picture (extremely good idea)

• `valgrind <program> <arguments>` will run <program> with <arguments>, and print out summary of these common errors

• `valgrind` will slow down execution

• `use --leak-check=full` for more info
What if I don’t free?

• When process execution is over, all memory is reclaimed by the OS

• This can be OK to rely on, especially for short lived processes

• We will tell you whether this is OK for this class: be sure to read the entirety of the homework writeup carefully