# C, Memory

## CS 2130: Computer Systems and Organization 1

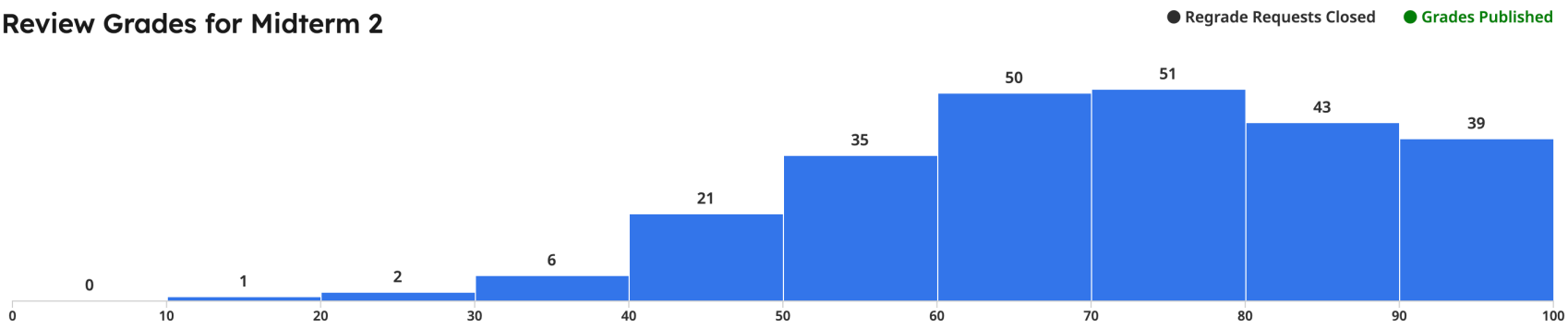
**Xinyao Yi** Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

## Announcements

- Homework 8 due Monday on Gradescope

## Review Grades for Midterm 2




| Minimum | Median | Maximum | Mean | Std Dev |
|---|---|---|---|---|
| 15.0 | 71.75 | 103.0 | 70.79 | 17.08 |

# Garbage

**Garbage** - memory on the heap our code will never use again

- Weird: defined in terms of the future!
- Compiler can't figure out when to free for you

It cannot tell what is actually garbage and what is not.

# Garbage

**Garbage** - memory on the heap our code will never use again

- Weird: defined in terms of the future!
- Compiler can't figure out when to free for you

What about Java?

Does Java have something like free? Set the pointer to Null ? JAVA has the garbage collector.

# Garbage Collector

"The garbage collector goes through and frees memory automatically that I'm not using anymore."

But it doesn't free all of garbage. It was not possible.

**Garbage Collector** - frees garbage "automatically"

- **Unreachable memory** - memory on heap that is unreachable through pointers on the stack (or reachable by them)
  - Subset of all the garbage
  - Identifiable!
- Takes resources to work
- *Very* popular - most languages have garbage collectors
  - Java, Python, C#, …

"Imagine your program stack has pointers to heap objects, and those objects might have references to other objects…"
The garbage collector pauses your program, walks through memory, and finds all heap objects that cannot be reached from the stack."

"This takes resources to work — it pauses your program and makes Java a little slower, but it's very popular."



"Most languages have it — only a few don't, like C, C++, and Fortran. Some make it optional like Rust, D, and Go."

# Common Memory Bugs (reading)

you should read it in detail by lab and will see a lot of these there.

# List Example

makeList, destroyList and append.

You should have seen all in the homework.

# ① header file. (list.h).

```
#ifndef __LIST_H__
#define __LIST_H__

typedef struct {
   unsigned length;
   int *array;
} List;

void append(List *list, int item);

List *makeList();
void destroyList(List *);

#endif
```

it's okay for compiler.

For better reading: (List *list)

1. #ifndef __LIST_H__

#ifndef means "if not defined."

This checks whether the macro __LIST_H__ has not been defined yet.

It is the start of an include guard, which prevents this header from being included multiple times.

2. #define __LIST_H__

This defines the macro __LIST_H__.

Now the compiler knows that this header has been included once.

Everything between this line and #endif will only be included one time, even if you #include it repeatedly.

3. typedef struct {

This begins a structure definition.
typedef means you will give this struct a type name (List) later.

4. unsigned length;

This declares a field named length.
Type: unsigned (same as unsigned int).
It stores the number of elements currently in the list.
Because it is unsigned, it can never be negative.

5. int *list;

This declares a field named list.
Type: int * — a pointer to an integer.
This pointer will point to a dynamically allocated array of integers.
This is where the actual list elements are stored.

6. void append(List *list, int item);

This is a function declaration.

Parameters:
List *list — a pointer to the list you want to modify.
int item — the integer to append.

Purpose: Append item to the end of the list.

7. List *makeList();

This is another function declaration.

Return type: List * — a pointer to a newly created list.

Purpose: Allocate a new List and initialize it.

8. void destroyList(List *);

Function declaration.

Returns nothing (void).

Parameter: a List * (the name is omitted, but the type is given).

Purpose: Free all memory used by a List.

9. #endif

This matches the #ifndef at the top.
It ends the include guard.
Ensures the header file is only included once during compilation.

## ② list.c

```c
#include "list.h"
#include <stdlib.h>

List *makeList() {
    List *ret = (List *) malloc(sizeof(List));

    ret->length = 0;
    ret->array = (int *) calloc(ret->length, sizeof(int));

    return ret;
}

void destroyList(List *list) {
    free(list->array);
    free(list);
}

void append(List *list, int item) {
    list->length += 1;
    list->array = realloc(list->array, list->length * sizeof(int));
    list->array[list->length - 1] = item;
}
```

Allocates enough memory for one List struct.
Cast (List *) is unnecessary in C but not harmful.
ret is a pointer to the newly allocated struct.

Allocates storage for the array of integers.
ret->length is 0, so this allocates 0 bytes.
Legal: calloc(0, …) usually returns NULL.
Prepares the list to grow later.

Returns the pointer to the newly created empty list.

Frees the dynamically allocated integer array.

Frees the struct itself.
After this, the list pointer is invalid.

Increase length by 1 because we add one item.

Expands or creates the array so it can hold the new element.
If old pointer was NULL (on first append), realloc(NULL, size) behaves like malloc(size).

Writes the new element into the last position.
Example: if length = 2 after increment, last index = 1.

## ③ uselist.c

```c
#include "list.h"
#include <stdio.h>

int main() {
    List *myList = makeList();

    append(myList, 42);
    append(myList, 3);

    printf("%p\n", myList);

    for(int i = 0; i < myList->length; i++) {
        printf("%d ", myList->array[i]);
    }

    puts("");
    destroyList(myList);
    return 0;
}
```

Creates an empty list.

Adds two integers to the list.
After this:
length = 2
array = {42, 3}

Prints the memory address of myList.

Loops over each element and prints it.

just want a new line.

Frees array and struct.