The Stack, The Heap, and Dynamic Memory Allocation

CS 3410: Computer System Organization and Programming

Spring 2025

[G. Guidi, A. Sampson, Z. Susag, and H. Weatherspoon]

Roadmap

1. Finish Pointers
    - Strings
    - **Fun Pointer Tricks**
2. The Call Stack
3. The Heap

Pointers, Revisited

"Every computer, at the unreachable memory address 0x-1, stores a secret. I found it, and it is that all humans ar-- SEGMENTATION FAULT."

Strings are Null-Terminated Character Arrays

- Recall that we told you a string has type `char*` in C
  - Strings are arrays of `char` values
  - A `char` is generally 1-byte (8-bits)
- Strings keep track of length by ending with a *null character* (`‘\0’`)
  - All strings *should* end with a *null character*
- **Example:**
  - `“CS3410” = { 'C', 'S', '3', '4', '1', '0', '\0’ }`
  - “CS3410” has length 7, not 6!

Demo: Strings

```
void print_line(char *s) {
  for (int i = 0; s[i] != '\0'; ++i)
  {
    fputc(s[i], stdout);
  }
  fputc('\n', stdout);
}

int main() {
  char message[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
  print_line(message);
  return 0;
}
```

Roadmap

1. Finish Pointers
   - Strings
   - **Fun Pointer Tricks**
2. The Call Stack
3. The Heap

Pass by Reference

```c
#include <stdio.h>

void swap(int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}

int main() {
   int a = 34;
   int b = 10;
   printf("a: %d; b: %d\n", a, b);
   swap(a, b);
   printf("a: %d; b: %d\n", a, b);
}
```

https://pollev.com/zacharysusag306

What does the `printf` statement on line 14 print out?

Pass by Reference

```c
#include <stdio.h>

void swap(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int a = 34;
  int b = 10;
  printf("a: %d; b: %d\n", a, b);
  swap(a, b);
  printf("a: %d; b: %d\n", a, b);
}
```

a: 34; b: 10

a: 34; b: 10

Pass by Reference

```c
#include <stdio.h>

void swap(int* x, int* y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int a = 34;
  int b = 10;
  printf("a: %d; b: %d\n", a, b);
  swap(&a, &b);
  printf("a: %d; b: %d\n", a, b);
}
```

a: 34; b: 10

a: 10; b: 34

The Arrow Operator

```c
#include <stdio.h>
typedef struct {
  int x;
  int y;
} point_t;

void print_point(point_t* p) {
  printf("(%d, %d)\n", (*p).x, (*p).y);
}

int main() {
  point_t my_point;
  my_point.x = 3;
  my_point.y = 7;

  print_point(&my_point);

  return 0;
}
```

The Arrow Operator

```c
#include <stdio.h>
typedef struct {
  int x;
  int y;
} point_t;

void print_point(point_t* p) {
  printf("(%d, %d)\n", p->x, p->y);
}

int main() {
  point_t my_point;
  my_point.x = 3;
  my_point.y = 7;

  print_point(&my_point);

  return 0;
}
```

`(*struct).field`

is equivalent to

`struct->field`

Null Pointers

- Pointers are just integers (i.e., bits!), so what does 0 mean?
- `NULL` is a pointer with value `0`
  - Often used to signal failure
- Be Careful!
  - **NEVER dereference `NULL`**
  - When in doubt, always check!

Pointers to Anything

```c
#include <stdio.h>

void print_ptr(void* p) {
  printf("%p\n", p);
}

int main() {
  int x = 34;
  float y = 10.0f;
  print_ptr(&x);
  print_ptr(&y);
}
```

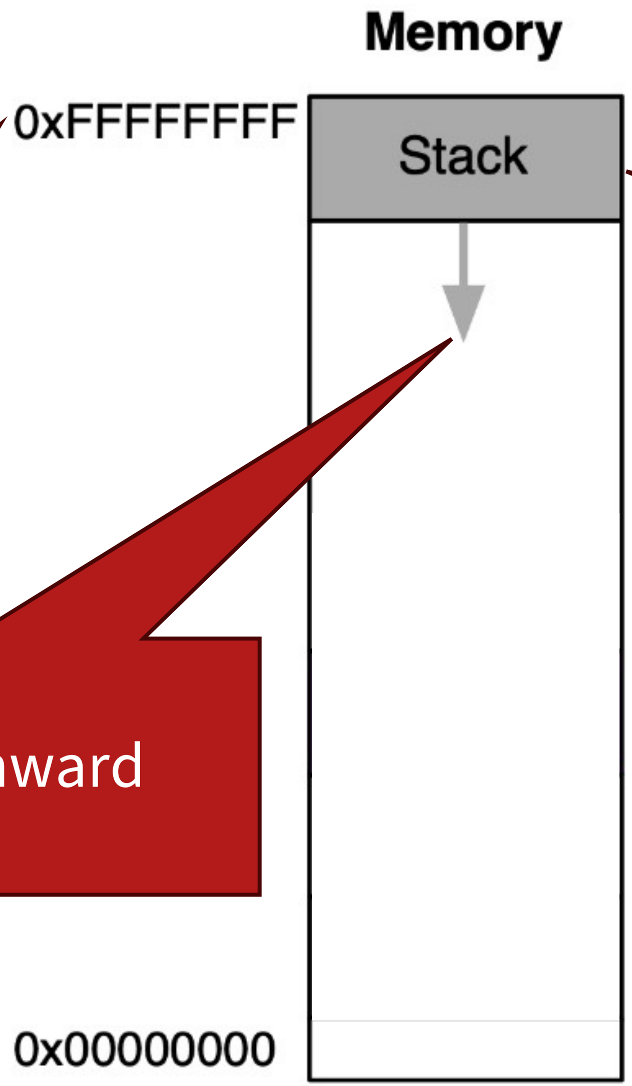
- Pointers are just bits!
  - No difference between **int***, **float***, and **char***
- **void*** is a “pointer to something”

Roadmap

1. Finish Pointers
    - Strings
    - Fun Pointer Tricks
2. **The Call Stack**
3. The Heap

Overview: The Call Stack

Starts at a *high* memory address



Grows downward

Stores local variables, function arguments

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)
0x1555d56bb0	
0x1555d56bac	
0x1555d56ba8	
0x1555d56ba4	
0x1555d56ba0	
…	…
0x1555d56b7c	
0x1555d56b78	
0x1555d56b74	
0x1555d56b70	
0x1555d56b6c	
0x1555d56b68	

The Stack

GROWS DOWN

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
…	…	
0x1555d56b7c		
0x1555d56b78		
0x1555d56b74		
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0		
0x1555d56bac		
0x1555d56ba8	values	main
0x1555d56ba4		
0x1555d56ba0		
...	...	
0x1555d56b7c		
0x1555d56b78		
0x1555d56b74		
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
…	…	
0x1555d56b7c	dest	fill_exp
0x1555d56b78		
0x1555d56b74	i	
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
...	...	
0x1555d56b7c		
0x1555d56b78		
0x1555d56b74		
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
...	...	
0x1555d56b7c	vals	
0x1555d56b78		
0x1555d56b74		
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
...	...	
0x1555d56b7c	vals	print_floats
0x1555d56b78		
0x1555d56b74	n	
0x1555d56b70	i	
0x1555d56b6c		
0x1555d56b68		

The Stack

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

void fill_exp(float* dest) {
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
    dest[i] = dest[i - 1] * EULER;
  }
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
    printf("%f\n", vals[i]);
  }
}

int main() {
  float values[COUNT];
  fill_exp(values);
  print_floats(values, COUNT);
  return 0;
}
```

Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
...	...	
0x1555d56b7c		
0x1555d56b78		
0x1555d56b74		
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

PollEv: create_exp()

```c
#include <stdio.h>

const float EULER = 2.71828f;
const int COUNT = 5;

float* create_exp() {
  float dest[COUNT];
  dest[0] = 1.0f;
  for (int i = 1; i < COUNT; ++i) {
      dest[i] = dest[i - 1] * EULER;
  }
  return dest;
}

void print_floats(float* vals, int n) {
  for (int i = 0; i < n; ++i) {
      printf("%f\n", vals[i]);
  }
}

int main() {
  float* values = create_exp();
  print_floats(values, COUNT);
  return 0;
}
```

https://pollev.com/zacharysusag306

What does the program on the screen print?

Nobody has responded yet.

Hang tight! Responses are coming in.

Limitations of The Call Stack

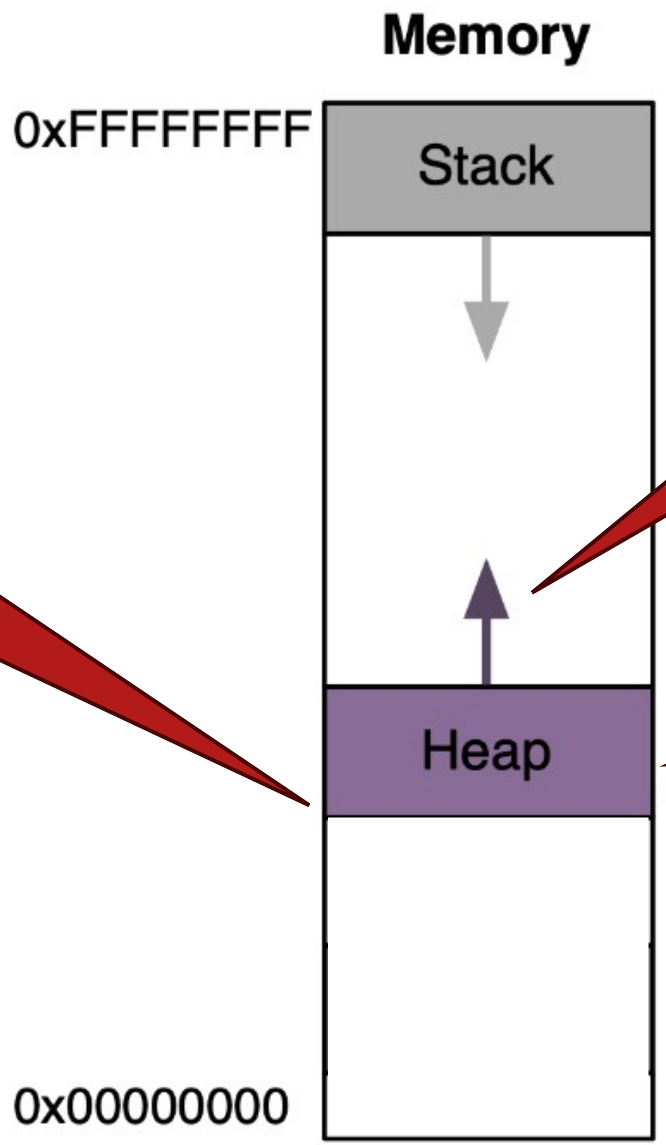
- Local variables only live as long as the function call
- **Never return a pointer to a local variable!**
  - Returning a pointer to data that is about to be "destroyed"
  - *Undefined behavior*
- Safe Operations:
  1. Passing a pointer to a local variable as an argument to a function
  2. Returning a non-pointer value
     - Compiler will handle copying these!

Demo: `create_exp()`

Roadmap

1. Finish Pointers
   - Strings
   - Fun Pointer Tricks
2. The Call Stack
3. **The Heap**

Overview: The Heap



Starts at a *low* memory address

Grows upward

Stores *dynamically allocated data*

The Stack vs. The Heap

The Stack

The Heap

malloc(…) & free(…)

Located in `stdlib.h`

Pointer to first byte in new block

Number of *bytes* to allocate

```
void* malloc(size_t size);
```

"Memory Allocate"

```
void free(void* ptr);
```

Pointer to the beginning of memory that was allocated by `malloc`

Demo: Heapified `create_exp()`

```c
#include <stdio.h>
#include <stdlib.h>

const float EULER = 2.71828f;
const int COUNT = 10;

float* create_exp() {
    float* dest = malloc(COUNT * sizeof(float));
    dest[0] = 1.0f;
    for (int i = 1; i < COUNT; ++i) {
        dest[i] = dest[i - 1] * EULER;
    }
    return dest;
}

void print_floats(float* vals, int count) {
    for (int i = 0; i < count; ++i) {
        printf("%f\n", vals[i]);
    }
}

int main() {
    float* values = create_exp();
    print_floats(values, 10);
    free(values);
    return 0;
}
```

The Laws of The Heap

Now we are really entering the “Danger Zone”!

The Laws of The Heap

1. **Use after free:** After you `free` memory, you can't use it.
2. **Double free:** You can only `free` memory once.
3. **Memory leak:** You must `free` all the memory you allocated with `malloc`
4. **Out-of-bounds access:** You can only access data *inside* allocated block.

The Laws of The Heap

```c
int main() {
  int *arr = malloc(10 * sizeof(int));
  for (int i = 0; i < 10; i++) {
    arr[i] = i + 1;
  }
  free(arr);

  // This violates Law 1: Use after free!
  printf("arr[0] = %d\n", arr[0]);

  return 0;
}
```

1. **Use after free:** After you `free` memory, you can't use it.

The Laws of The Heap

```c
int main() {
  int *arr = malloc(10 * sizeof(int));
  for (int i = 0; i < 10; i++) {
    arr[i] = i + 1;
  }
  free(arr);

  // This violates Law 2: Double free!
  free(arr);

  return 0;
}
```

2. **Double free:** You can only `free` memory once.

The Laws of The Heap

```c
int main() {
  int *arr = malloc(10 * sizeof(int));
  for (int i = 0; i < 10; i++) {
    arr[i] = i + 1;
  }

  // This violates Law 3: Memory leak!
  // no free :(

  return 0;
}
```

3. **Memory leak:** You must `free` all the memory you allocated with `malloc`

The Laws of The Heap

```c
int main() {
  int *arr = malloc(10 * sizeof(int));
  for (int i = 0; i < 10; i++) {
    arr[i] = i + 1;
  }

  // This violates Law 4:
  // Out-of-bounds Access!
  printf("arr[10] = %d\n", arr[10]);

  return 0;
}
```

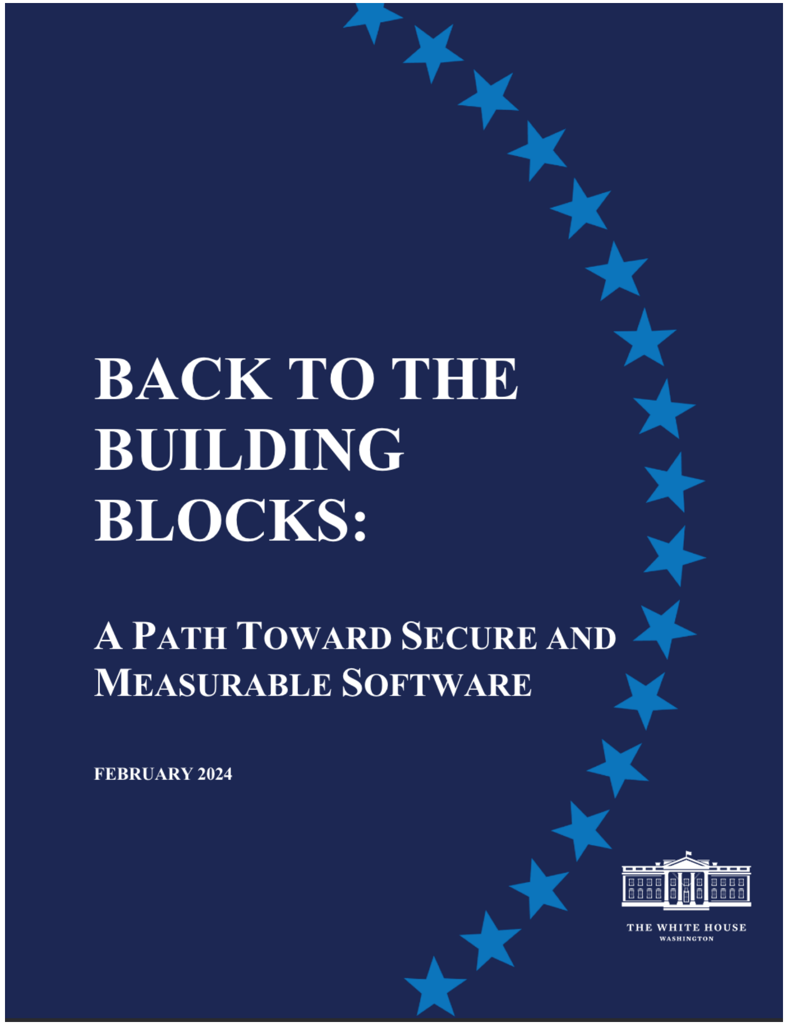
4. **Out-of-bounds Access:** You can only access data *inside* allocated block.

Demo: `memory_bugs.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const float EULER = 2.71828f;
const int COUNT = 10;

// Allocate a new array containing `COUNT` values from an exponential series.
float* create_exp() {
    float* dest = malloc(COUNT * sizeof(float));  // New!
    dest[0] = 1.0f;
    for (int i = 1; i < COUNT; ++i) {
        dest[i] = dest[i - 1] * EULER;
    }
    return dest;
}

// Print the first `count` values in a float array.
void print_floats(float* vals, int count) {
    for (int i = 0; i < count; ++i) {
        printf("%f\n", vals[i]);
    }

    // Let's see what's nearby...
    char* ptr = (char*)vals;
    for (int j = 0; j < 100; ++j) {
        char* byte = ptr - j;
        printf("%p: %d %c\n", byte, *byte, *byte);
    }
}

// Generate a secret.
char* gen_secret() {
    char* secret = malloc(16);
    strcpy(secret, "seekrit!");
    return secret;
}

int main() {
    char* password = gen_secret();
    float* values = create_exp();

    print_floats(values, COUNT);

    free(values);
    free(password);
    return 0;
}
```

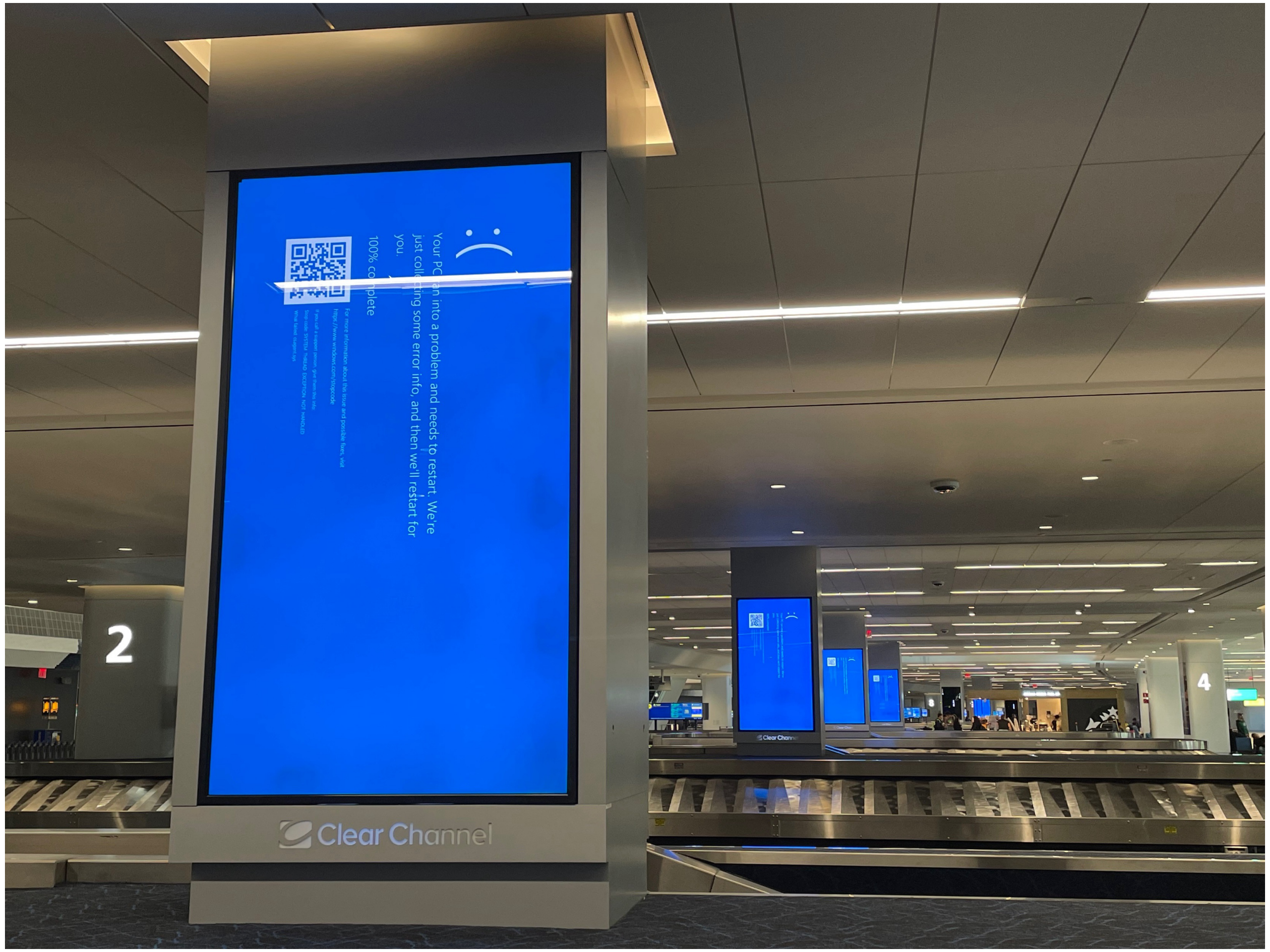
Memory Safety is Hard

- In 2019, Microsoft found that 70% of all security vulnerabilities were **memory safety violations**
- These bugs are only possible in languages like C/C++
  - *Memory safe* languages: Python, Java, OCaml, Rust, Swift



2024 CrowdStrike Outage

- ~8.5 million PCs crashed and were unable to restart across the planet
- Estimated to cost ~$10 billion
- Ultimately due to an out-of-bounds access!



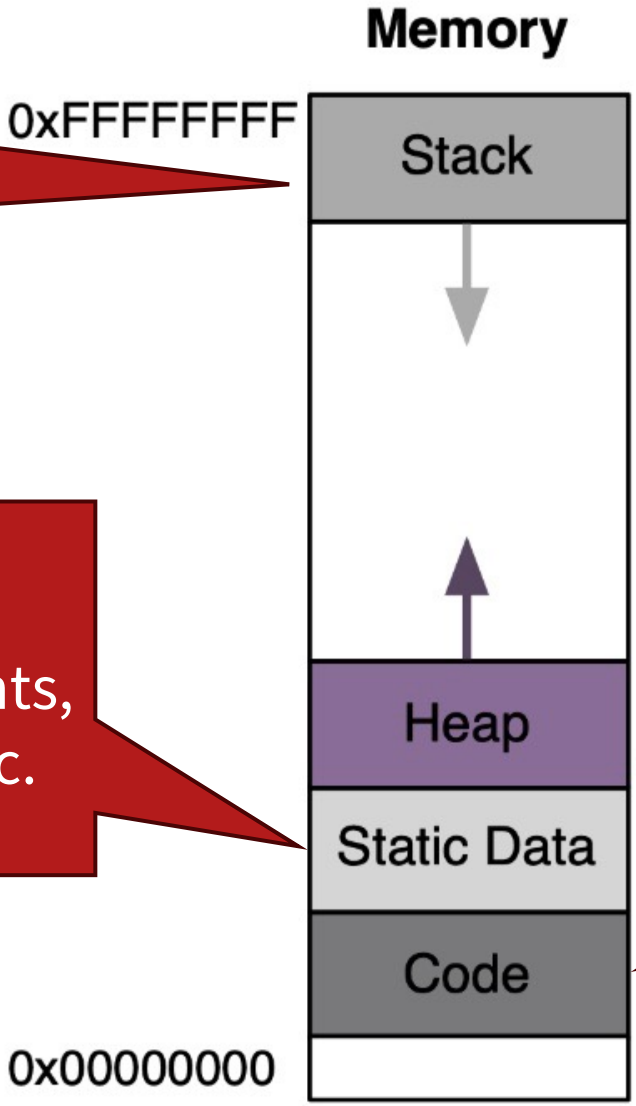
Demo: Address Sanitizer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const float EULER = 2.71828f;
const int COUNT = 10;

// Allocate a new array containing `COUNT` values from an exponential series.
float* create_exp() {
    float* dest = malloc(COUNT * sizeof(float));  // New!
    dest[0] = 1.0f;
    for (int i = 1; i < COUNT; ++i) {
        dest[i] = dest[i - 1] * EULER;
    }
    return dest;
}

// Print the first `count` values in a float array.
void print_floats(float* vals, int count) {
    for (int i = 0; i < count; ++i) {
        printf("%f\n", vals[i]);
    }

    // Let's see what's nearby...
    char* ptr = (char*)vals;
    for (int j = 0; j < 100; ++j) {
        char* byte = ptr - j;
        printf("%p: %d %c\n", byte, *byte, *byte);
    }
}

// Generate a secret.
char* gen_secret() {
    char* secret = malloc(16);
    strcpy(secret, "seekrit!");
    return secret;
}

int main() {
    char* password = gen_secret();
    float* values = create_exp();

    print_floats(values, COUNT);

    free(values);
    free(password);
    return 0;
}
```

Memory Layout

Memory Layout

Local variables, function arguments



Dynamically allocated data

Stores global variables, constants, string literals, etc.

Machine code (i.e., the program!)