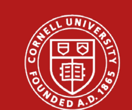
The Stack, The Heap, and Dynamic Memory Allocation

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

```
 1  void print_line(char *s) {
 2    for (int i = 0; s[i] != '\0'; ++i)
 3    {
 4      fputc(s[i], stdout);
 5    }
 6    fputc('\n', stdout);
 7  }
 8
 9  int main() {
10    char message[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
11    print_line(message);
12    return 0;
13  }
```

Roadmap

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

Pass by Reference

```
1   #include <stdio.h>
2
3   void swap(int x, int y) {
4     int tmp = x;
5     x = y;
6     y = tmp;
7   }
8
9   int main() {
10    int a = 34;
11    int b = 10;
12    printf("a: %d; b: %d\n", a, b);
13    swap(a, b);
14    printf("a: %d; b: %d\n", a, b);
15  }
```

https://pollev.com/zacharysusag306

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

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Pass by Reference

```
1   #include <stdio.h>
2
3   void swap(int x, int y) {
4     int tmp = x;
5     x = y;
6     y = tmp;
7   }
8
9   int main() {
10    int a = 34;
11    int b = 10;
12    printf("a: %d; b: %d\n", a, b);
13    swap(a, b);
14    printf("a: %d; b: %d\n", a, b);
15  }
```

Pass by Reference

```
1   #include <stdio.h>
2
3   void swap(int* x, int* y) {
4     int tmp = *x;
5     *x = *y;
6     *y = tmp;
7   }
8
9   int main() {
10    int a = 34;
11    int b = 10;
12    printf("a: %d; b: %d\n", a, b);
13    swap(&a, &b);
14    printf("a: %d; b: %d\n", a, b);
15  }
```

The Arrow Operator

```
1  #include <stdio.h>
2  typedef struct {
3    int x;
4    int y;
5  } point_t;
6
7  void print_point(point_t* p) {
8    printf("(%d, %d)\n", (*p).x, (*p).y);
9  }
10
11 int main() {
12   point_t my_point;
13   my_point.x = 3;
14   my_point.y = 7;
15
16   print_point(&my_point);
17
18   return 0;
19 }
```

The Arrow Operator

```c
1   #include <stdio.h>
2   typedef struct {
3     int x;
4     int y;
5   } point_t;
6
7   void print_point(point_t* p) {
8     printf("(%d, %d)\n", p->x, p->y);
9   }
10
11  int main() {
12    point_t my_point;
13    my_point.x = 3;
14    my_point.y = 7;
15
16    print_point(&my_point);
17
18    return 0;
19  }
```

`(*struct).field`

is equivalent to

`struct->field`

Cornell Bowers CIS
Computer Science

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

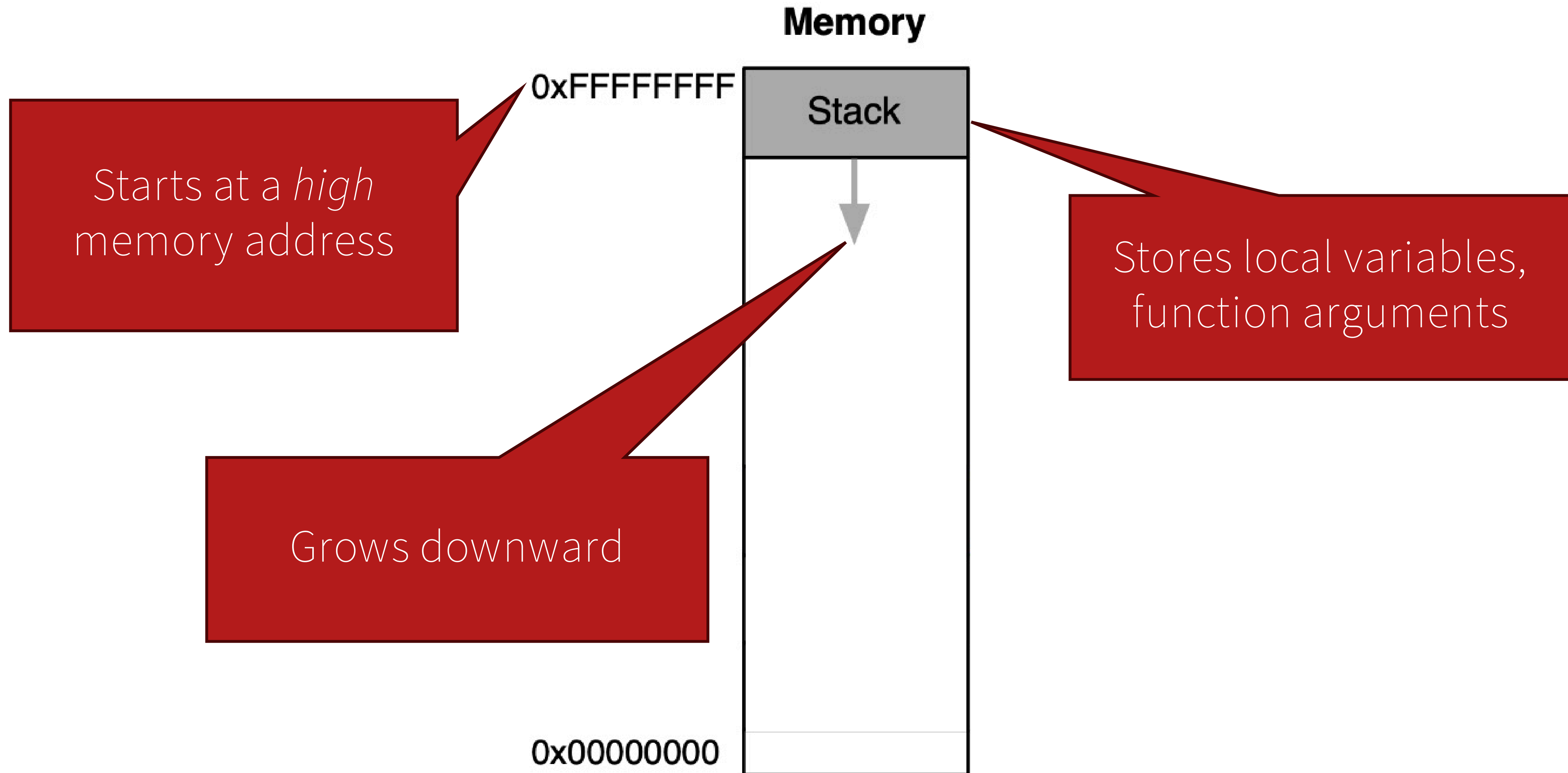
```
1   #include <stdio.h>
2
3   void print_ptr(void* p) {
4     printf("%p\n", p);
5   }
6
7   int main() {
8     int x = 34;
9     float y = 10.0f;
10    print_ptr(&x);
11    print_ptr(&y);
12  }
```

- 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

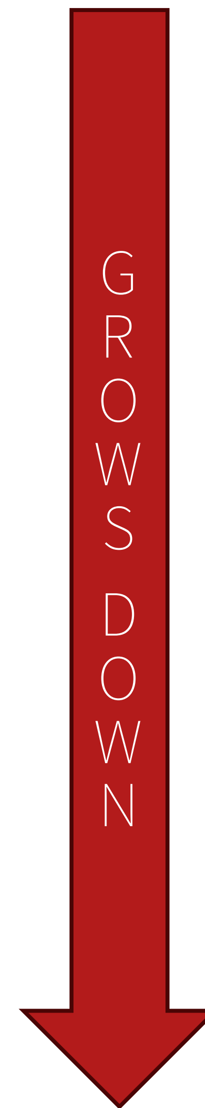
Overview: The Call Stack



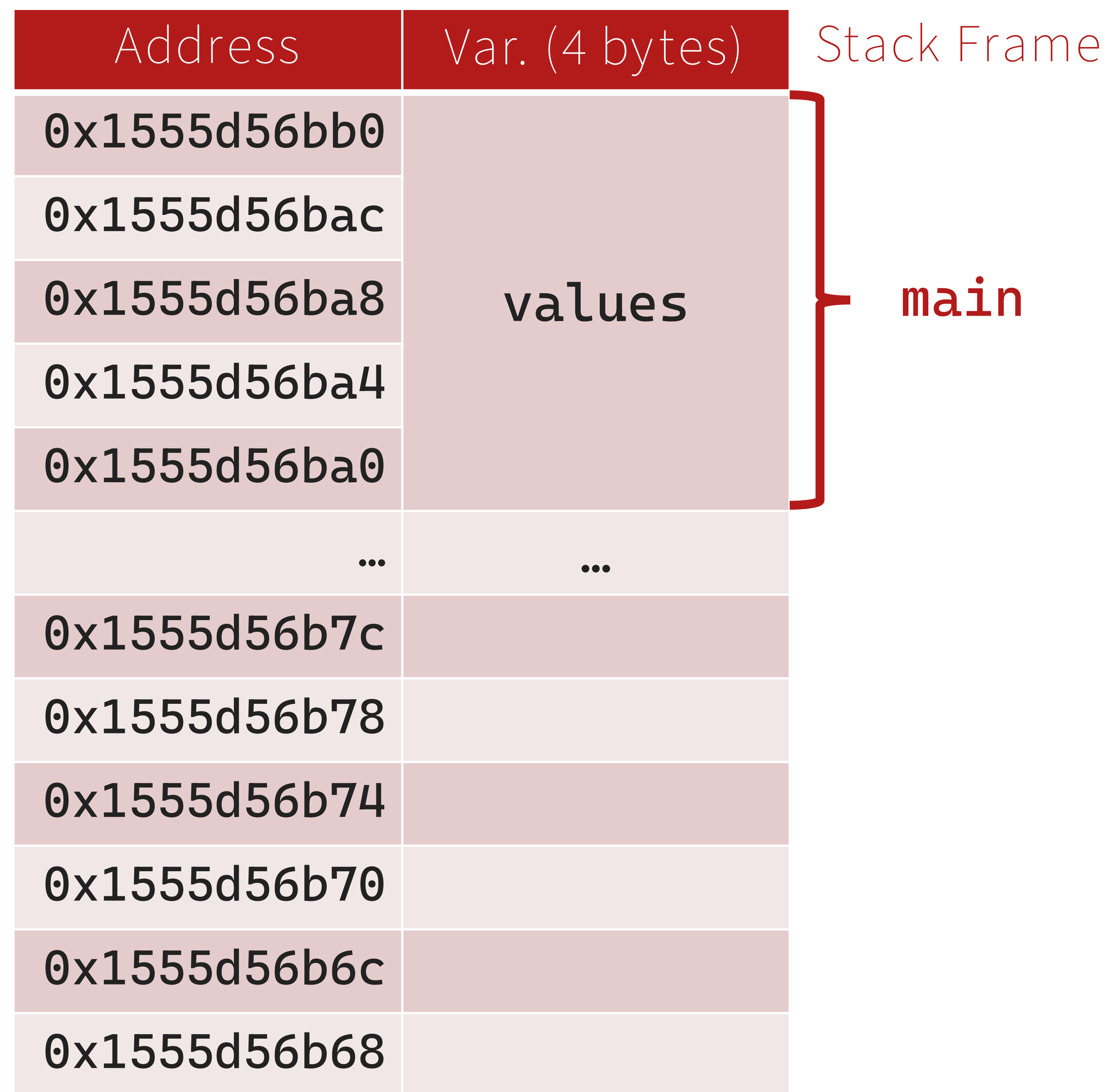
```c
1   #include <stdio.h>
2
3   const float EULER = 2.71828f;
4   const int COUNT = 5;
5
6   void fill_exp(float* dest) {
7     dest[0] = 1.0f;
8     for (int i = 1; i < COUNT; ++i) {
9       dest[i] = dest[i - 1] * EULER;
10    }
11  }
12
13  void print_floats(float* vals, int n) {
14    for (int i = 0; i < n; ++i) {
15      printf("%f\n", vals[i]);
16    }
17  }
18
19  int main() {
20    float values[COUNT];
21    fill_exp(values);
22    print_floats(values, COUNT);
23    return 0;
24  }
```

Address	Var. (4 bytes)
0x1555d56bb0	
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;
}
```



```c
1  #include <stdio.h>
2
3  const float EULER = 2.71828f;
4  const int COUNT = 5;
5
6  void fill_exp(float* dest) {
7    dest[0] = 1.0f;
8    for (int i = 1; i < COUNT; ++i) {
9      dest[i] = dest[i - 1] * EULER;
10   }
11 }
12
13 void print_floats(float* vals, int n) {
14   for (int i = 0; i < n; ++i) {
15     printf("%f\n", vals[i]);
16   }
17 }
18
19 int main() {
20   float values[COUNT];
21   fill_exp(values);
22   print_floats(values, COUNT);
23   return 0;
24 }
```

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

The Stack

```c
1   #include <stdio.h>
2
3   const float EULER = 2.71828f;
4   const int COUNT = 5;
5
6   void fill_exp(float* dest) {
7     dest[0] = 1.0f;
8     for (int i = 1; i < COUNT; ++i) {
9       dest[i] = dest[i - 1] * EULER;
10    }
11  }
12
13  void print_floats(float* vals, int n) {
14    for (int i = 0; i < n; ++i) {
15      printf("%f\n", vals[i]);
16    }
17  }
18
19  int main() {
20    float values[COUNT];
21    fill_exp(values);
22    print_floats(values, COUNT);
23    return 0;
24  }
```

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

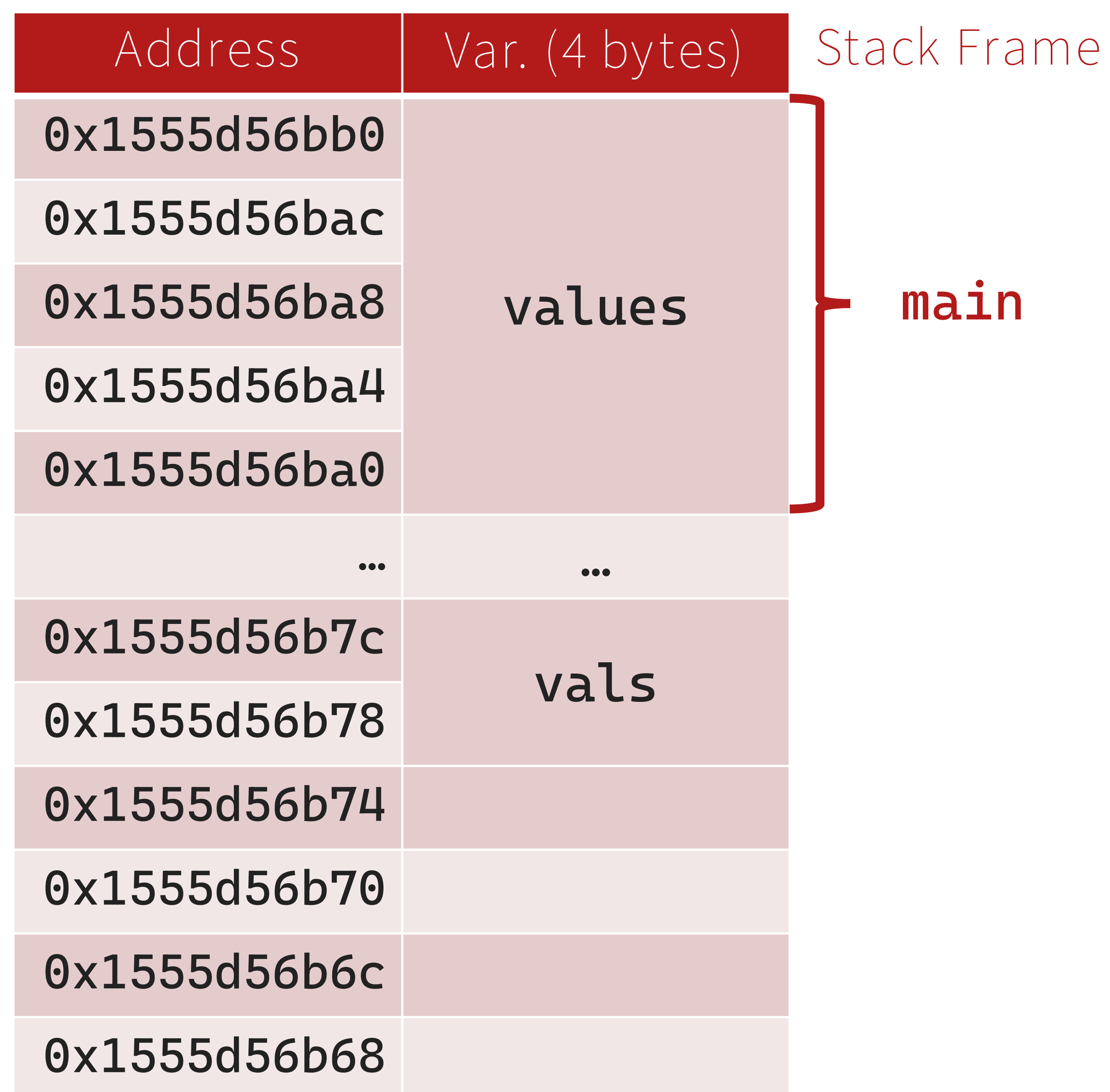
The Stack

```c
1  #include <stdio.h>
2
3  const float EULER = 2.71828f;
4  const int COUNT = 5;
5
6  void fill_exp(float* dest) {
7    dest[0] = 1.0f;
8    for (int i = 1; i < COUNT; ++i) {
9      dest[i] = dest[i - 1] * EULER;
10   }
11 }
12
13 void print_floats(float* vals, int n) {
14   for (int i = 0; i < n; ++i) {
15     printf("%f\n", vals[i]);
16   }
17 }
18
19 int main() {
20   float values[COUNT];
21   fill_exp(values);
22   print_floats(values, COUNT);
23   return 0;
24 }
```

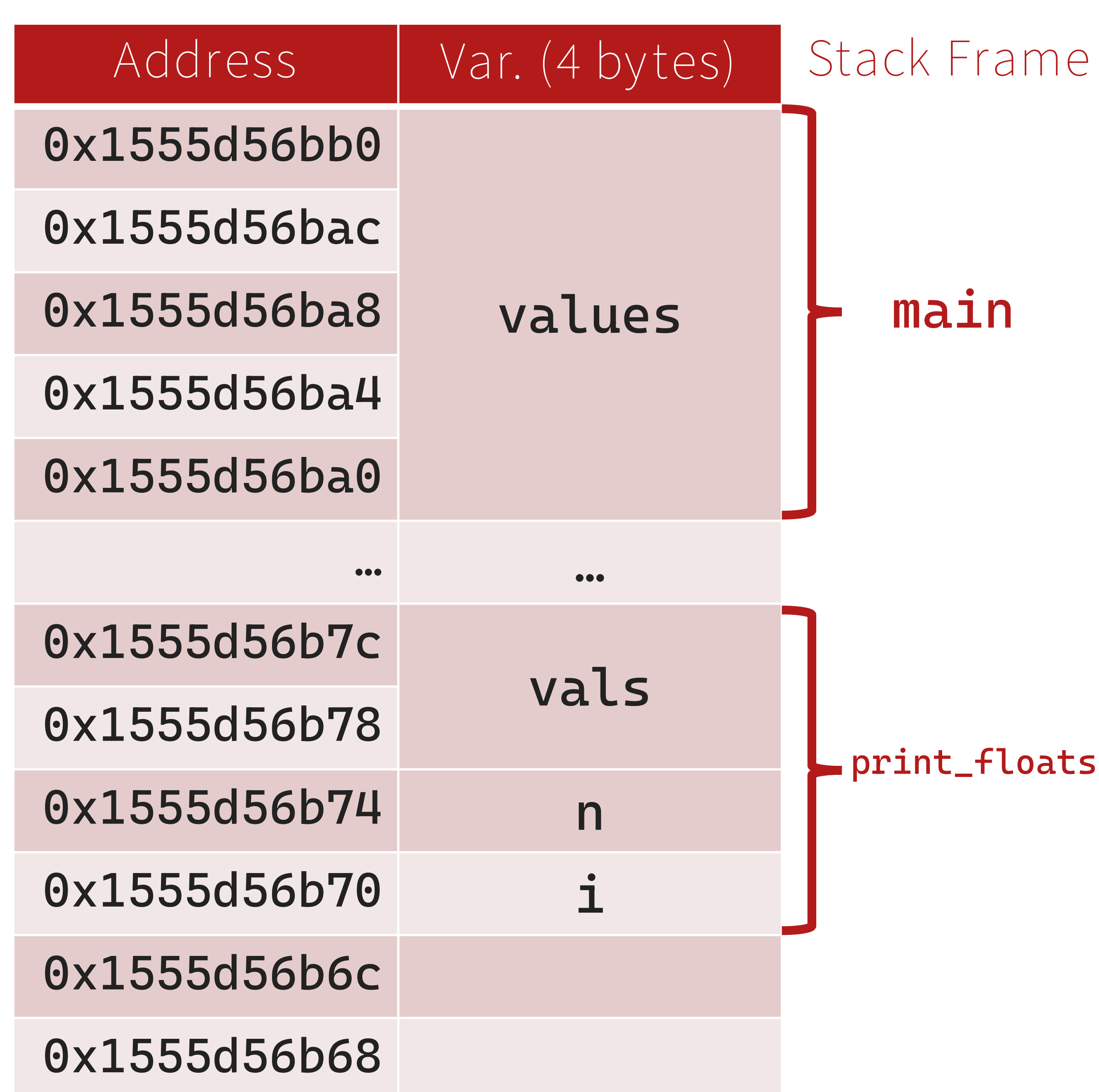
Address	Var. (4 bytes)	Stack Frame
0x1555d56bb0	values	main
0x1555d56bac		
0x1555d56ba8		
0x1555d56ba4		
0x1555d56ba0		
…	…	
0x1555d56b7c		
0x1555d56b78		
0x1555d56b74		
0x1555d56b70		
0x1555d56b6c		
0x1555d56b68		

The Stack

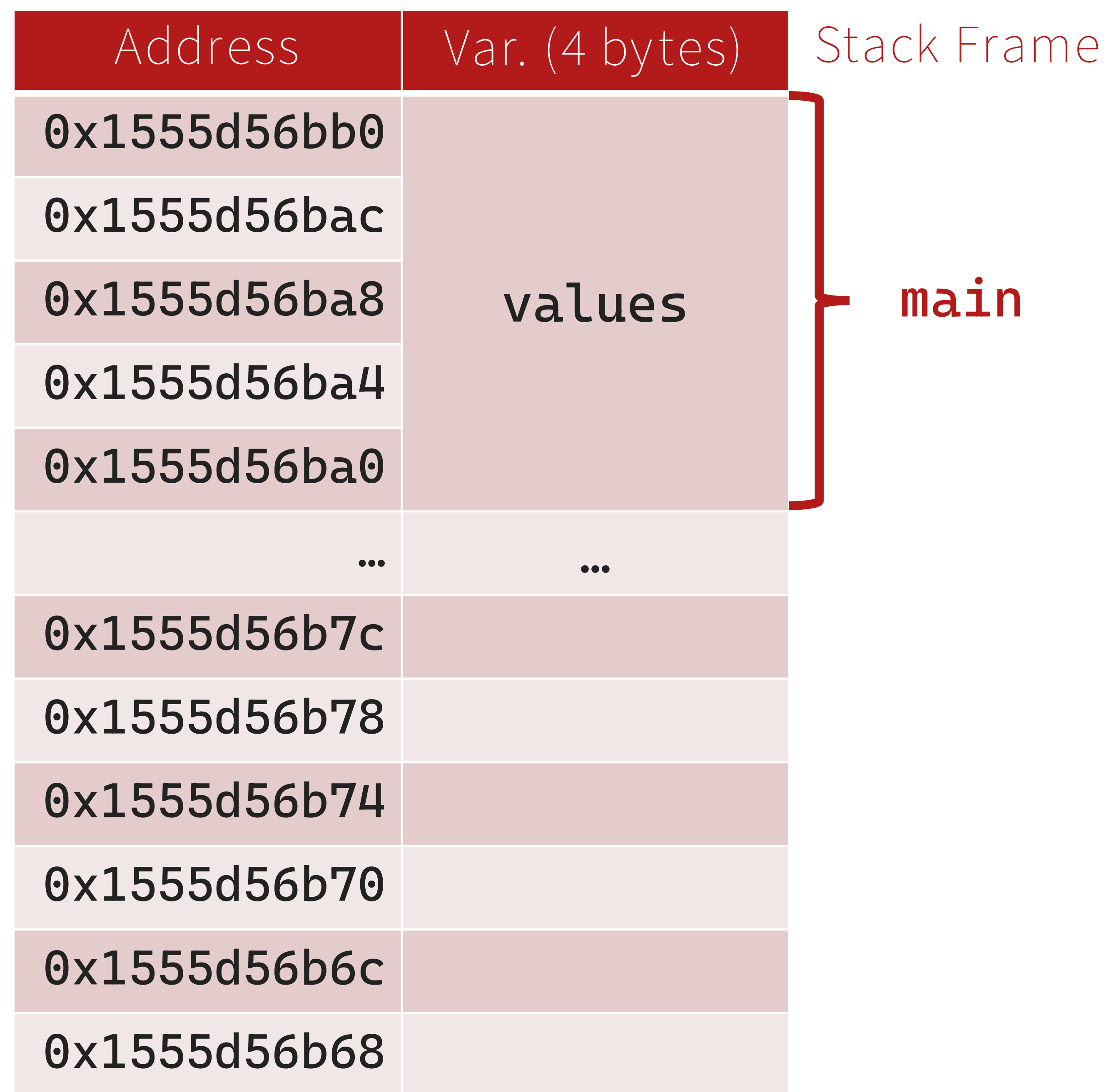
```c
#include <stdio.h>

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;
}
```



```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;
}
```



```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;
}
```



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.

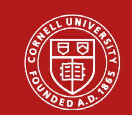
Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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!

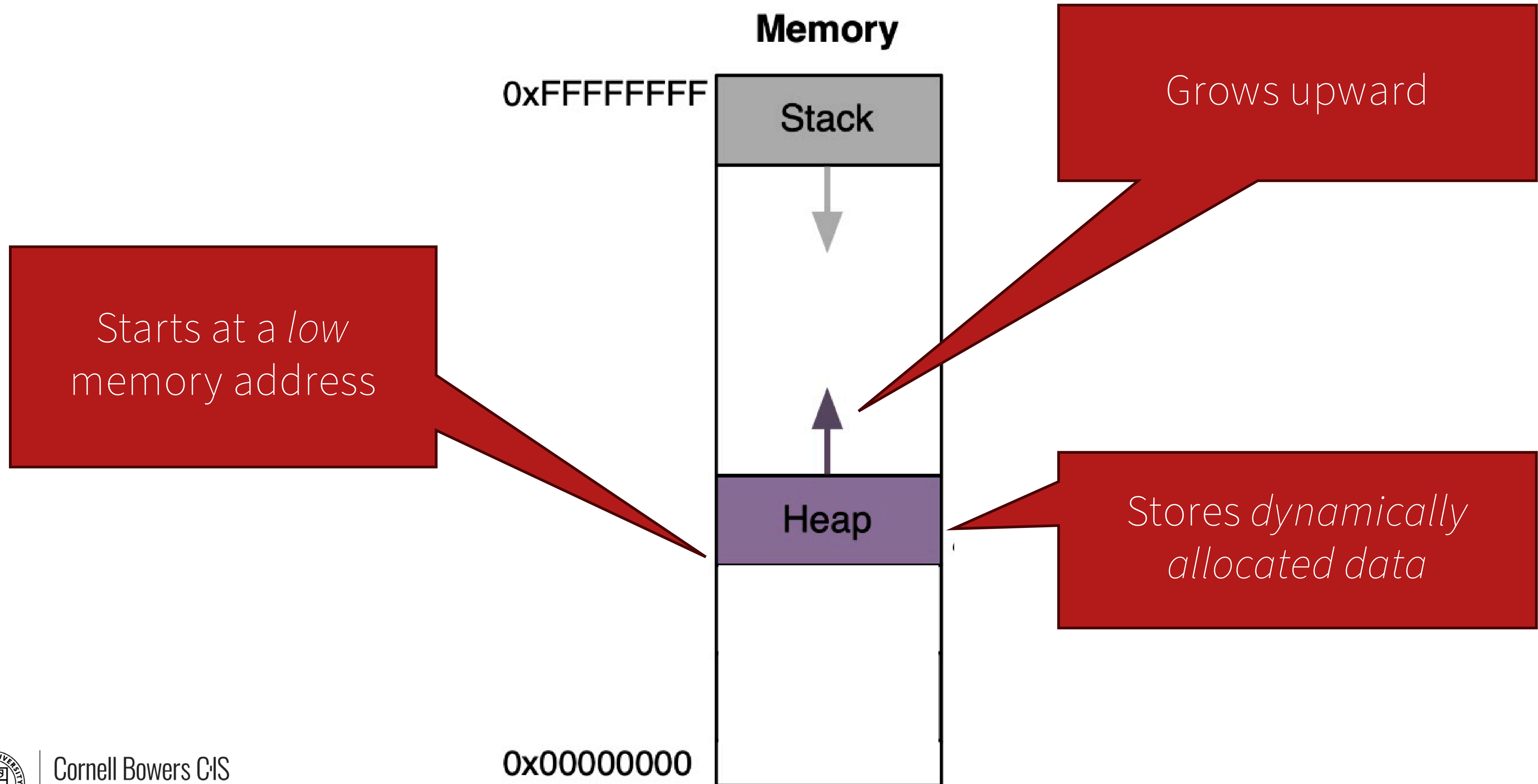
Demo: `create_exp()`



Roadmap

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

Overview: The Heap



The Stack vs. The Heap

The Stack

The Heap

By Free Software Foundation, Inc. - From http://gcc.gnu.org/img/gccegg.svg, with tweaks, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2388847
By National Agricultural Library - https://www.nal.usda.gov/exhibits/speccoll/exhibits/show/smokey-bear, Public Domain, https://commons.wikimedia.org/w/index.php?curid=80086014

`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()`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  const float EULER = 2.71828f;
5  const int COUNT = 10;
6
7  float* create_exp() {
8      float* dest = malloc(COUNT * sizeof(float));
9      dest[0] = 1.0f;
10     for (int i = 1; i < COUNT; ++i) {
11         dest[i] = dest[i - 1] * EULER;
12     }
13     return dest;
14 }
15
16 void print_floats(float* vals, int count) {
17     for (int i = 0; i < count; ++i) {
18         printf("%f\n", vals[i]);
19     }
20 }
21
22 int main() {
23     float* values = create_exp();
24     print_floats(values, 10);
25     free(values);
26     return 0;
27 }
```

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

```
1   int main() {
2     int *arr = malloc(10 * sizeof(int));
3     for (int i = 0; i < 10; i++) {
4       arr[i] = i + 1;
5     }
6     free(arr);
7
8     // This violates Law 1: Use after free!
9     printf("arr[0] = %d\n", arr[0]);
10
11    return 0;
12  }
```

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

The Laws of The Heap

```c
1  int main() {
2    int *arr = malloc(10 * sizeof(int));
3    for (int i = 0; i < 10; i++) {
4      arr[i] = i + 1;
5    }
6    free(arr);
7
8    // This violates Law 2: Double free!
9    free(arr);
10
11   return 0;
12 }
```

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

The Laws of The Heap

```
1   int main() {
2     int *arr = malloc(10 * sizeof(int));
3     for (int i = 0; i < 10; i++) {
4       arr[i] = i + 1;
5     }
6
7     // This violates Law 3: Memory leak!
8     // no free :(
9
10    return 0;
11  }
```

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

The Laws of The Heap

```
1   int main() {
2     int *arr = malloc(10 * sizeof(int));
3     for (int i = 0; i < 10; i++) {
4       arr[i] = i + 1;
5     }
6
7     // This violates Law 4:
8     // Out-of-bounds Access!
9     printf("arr[10] = %d\n", arr[10]);
10
11    return 0;
12  }
```

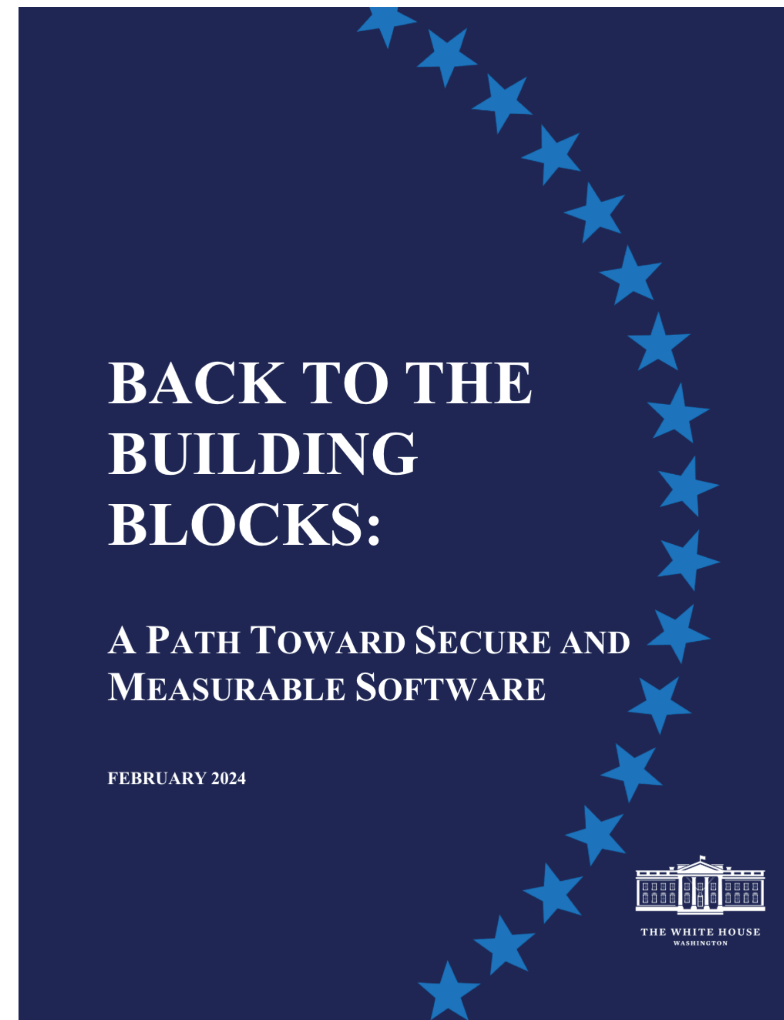
4. Out-of-bounds Access: You can only access data *inside* allocated block.

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!

By Smishra1 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=150535443

Demo: Address Sanitizer

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  const float EULER = 2.71828f;
6  const int COUNT = 10;
7
8  // Allocate a new array containing `COUNT` values from an exponential
series.
9  float* create_exp() {
10     float* dest = malloc(COUNT * sizeof(float));  // New!
11     dest[0] = 1.0f;
12     for (int i = 1; i < COUNT; ++i) {
13         dest[i] = dest[i - 1] * EULER;
14     }
15     return dest;
16 }
17
18 // Print the first `count` values in a float array.
19 void print_floats(float* vals, int count) {
20     for (int i = 0; i < count; ++i) {
21         printf("%f\n", vals[i]);
22     }
23
24     // Let's see what's nearby...
25     char* ptr = (char*)vals;
26     for (int j = 0; j < 100; ++j) {
27         char* byte = ptr - j;
28         printf("%p: %d %c\n", byte, *byte, *byte);
29     }
30 }
31
32 // Generate a secret.
33 char* gen_secret() {
34     char* secret = malloc(16);
35     strcpy(secret, "seekrit!");
36     return secret;
37 }
38
39 int main() {
40     char* password = gen_secret();
41     float* values = create_exp();
42
43     print_floats(values, COUNT);
44
45     free(values);
46     free(password);
47     return 0;
48 }
```

Memory Layout

Memory Layout