CS107, Lecture 11
Stack and Heap

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap
Ed Discussion: https://edstem.org/us/courses/46162/discussion/3644944

CS107 Topic 3: **How can we effectively manage all types of memory in our programs?**

CS107 Topic 3

How can we effectively manage all types of memory in our programs?

Why is answering this question important?

• Shows us how we can pass around data efficiently with pointers (last time)

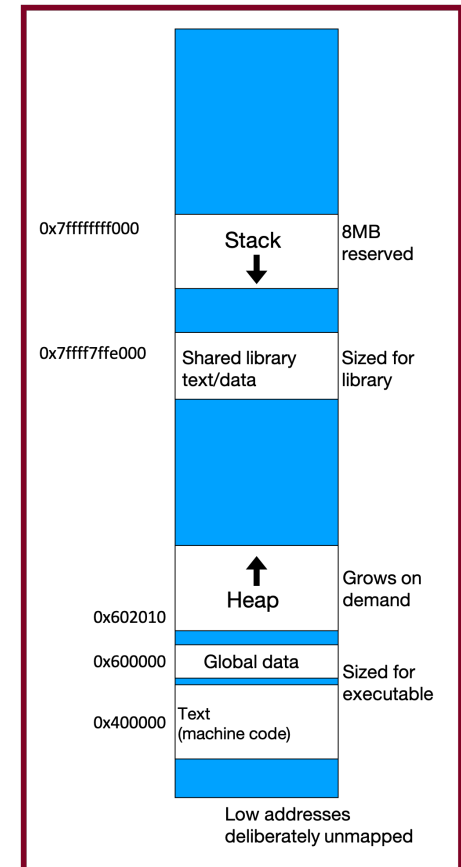• Introduces us to the heap and allocating memory that we manually manage (this time)

assign3: implement a function using resizable arrays to read lines of any length from a file and write 2 programs using that function to print the last N lines of a file and print just the unique lines of a file. These programs emulate the **tail** and **uniq** Unix commands!

Learning Goals

- Learn about the differences between the stack and the heap and when to use each one
- Become familiar with the **malloc**, **calloc**, **realloc** and **free** functions for managing memory on the heap

Memory Layout

- We are going to dive deeper into different areas of memory used by our programs.

- The **stack** is the place where all local variables and parameters live for each function. A function's stack frame goes away when the function returns.

- The stack grows **downwards** when a new function is called and shrinks **upwards** when the function is finished.

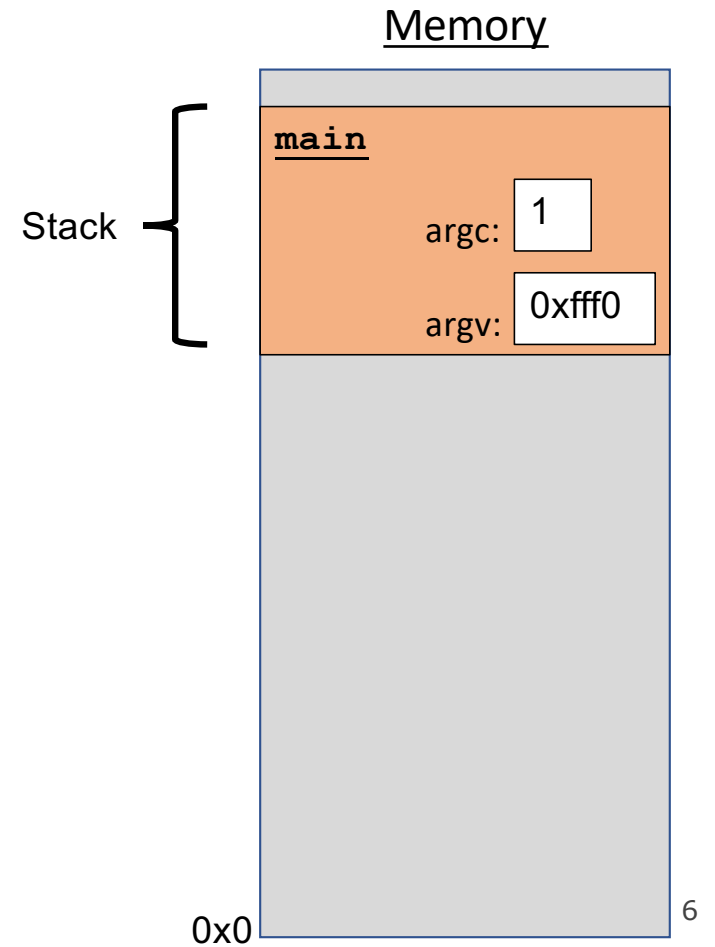| Address	Region	Note
	Stack	8MB reserved
0x7ffffffff000	Stack ↓	
0x7ffff7ffe000	Shared library text/data	Sized for library
		Grows on demand
	Heap ↑	
0x602010	Global data	Sized for executable
0x600000		
0x400000	Text (machine code)	
	Low addresses deliberately unmapped	

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
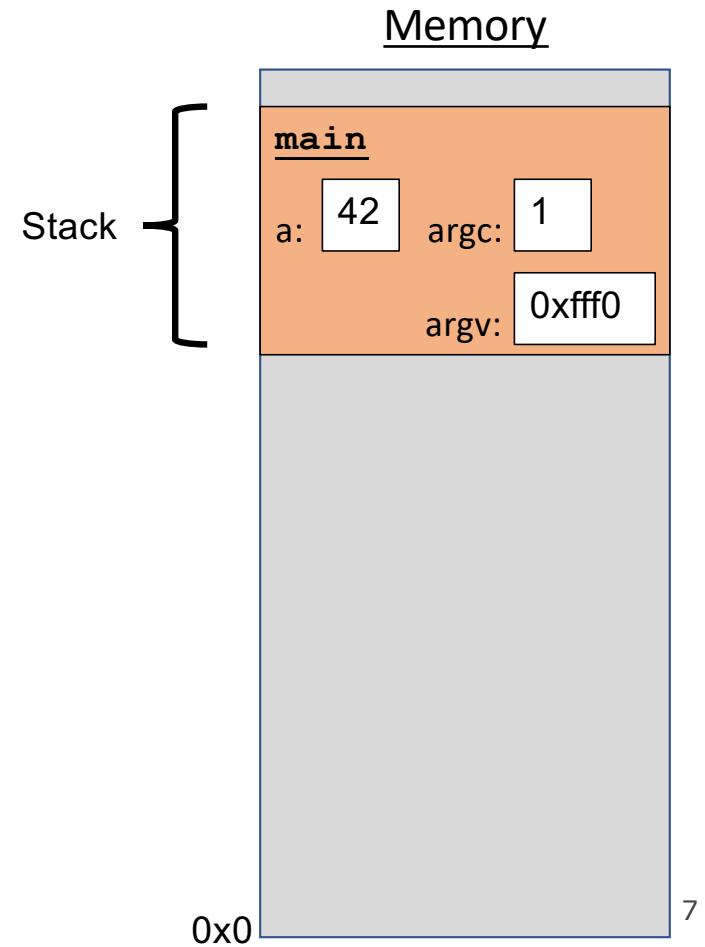
Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: 42   argc: 1
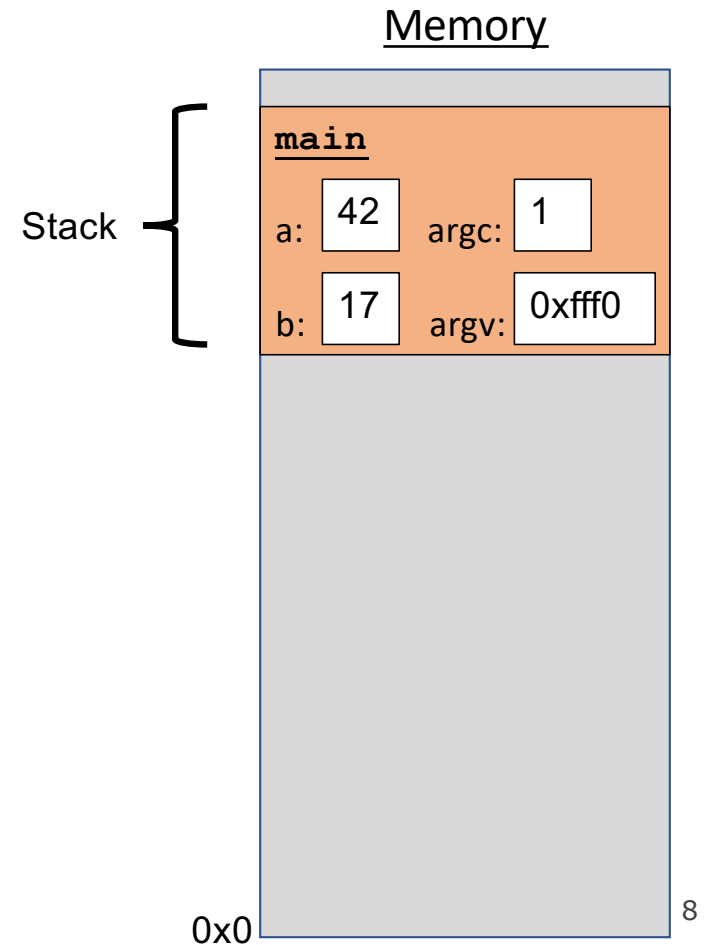
argv: 0xfff0

0x0

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

8

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
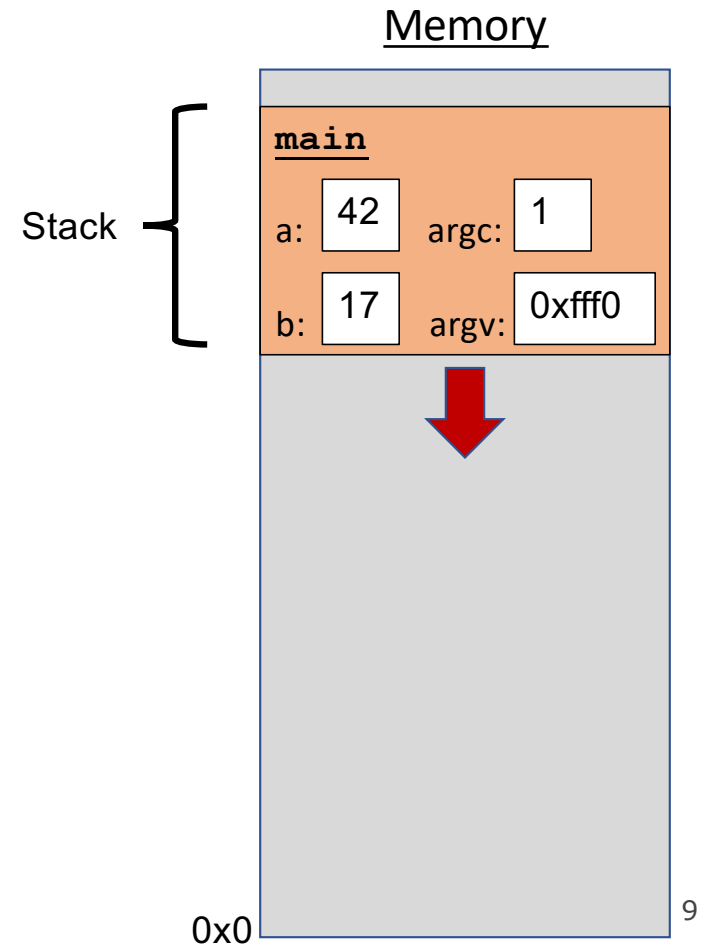
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
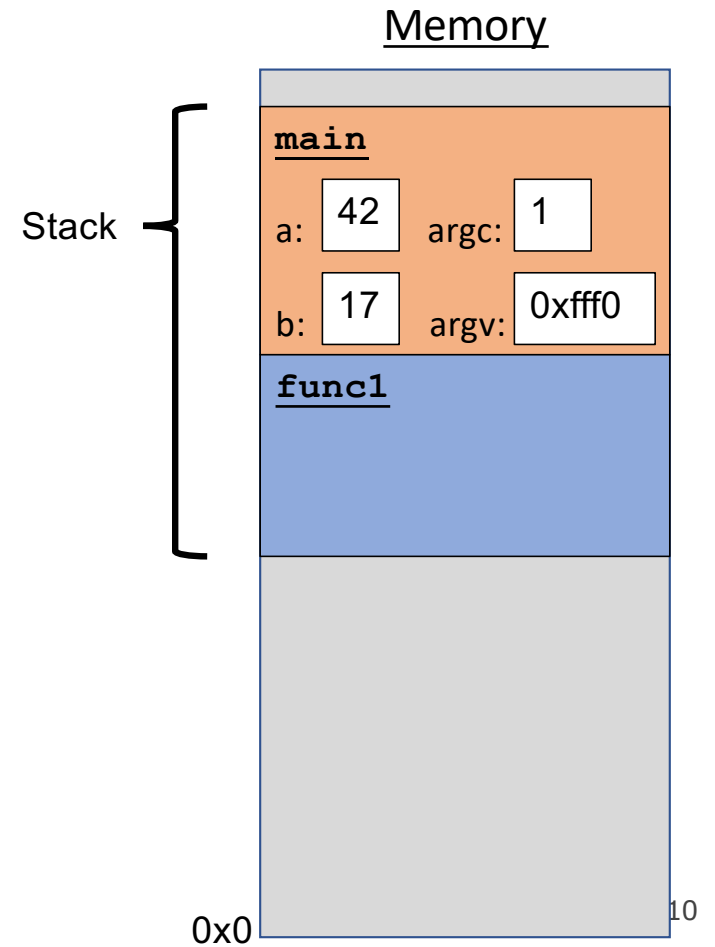
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

0x0
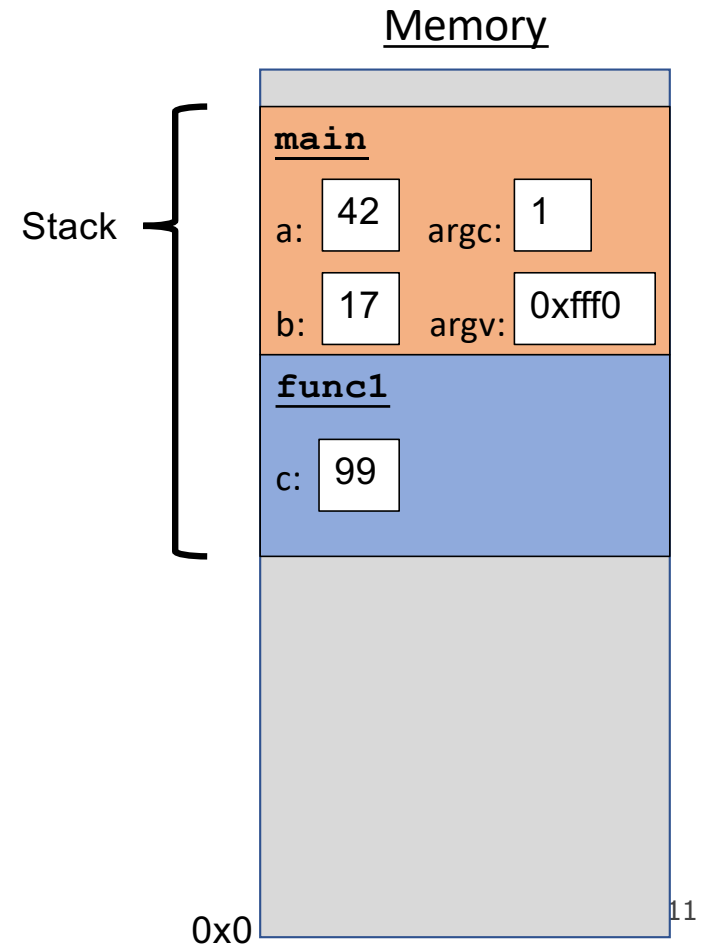
The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main			
a:	42	argc:	1
b:	17	argv:	0xfff0

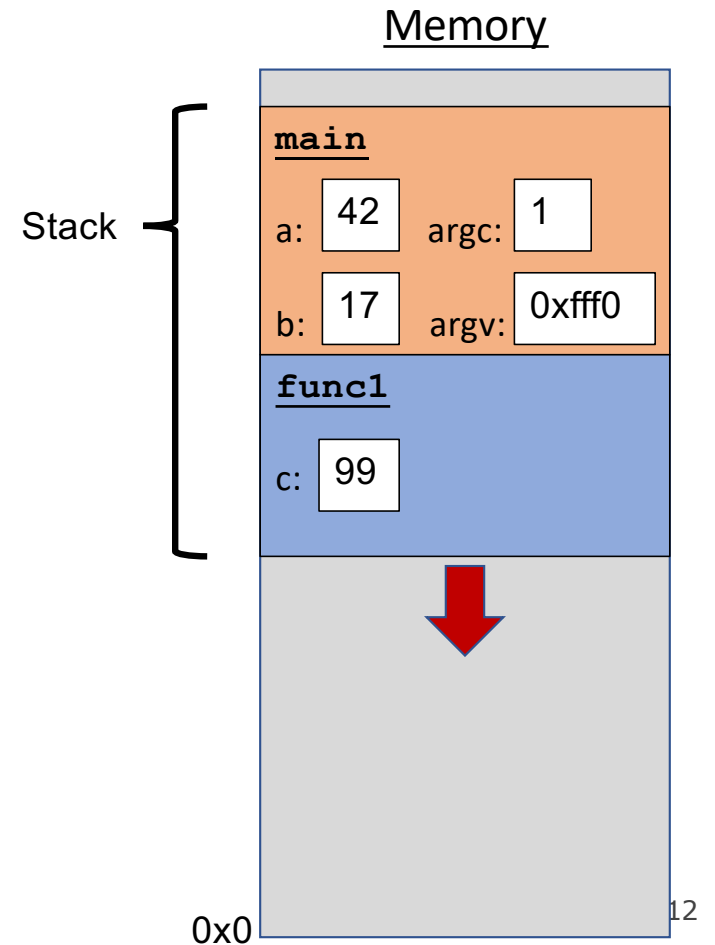| func1	
c:	99

0x0

11

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1
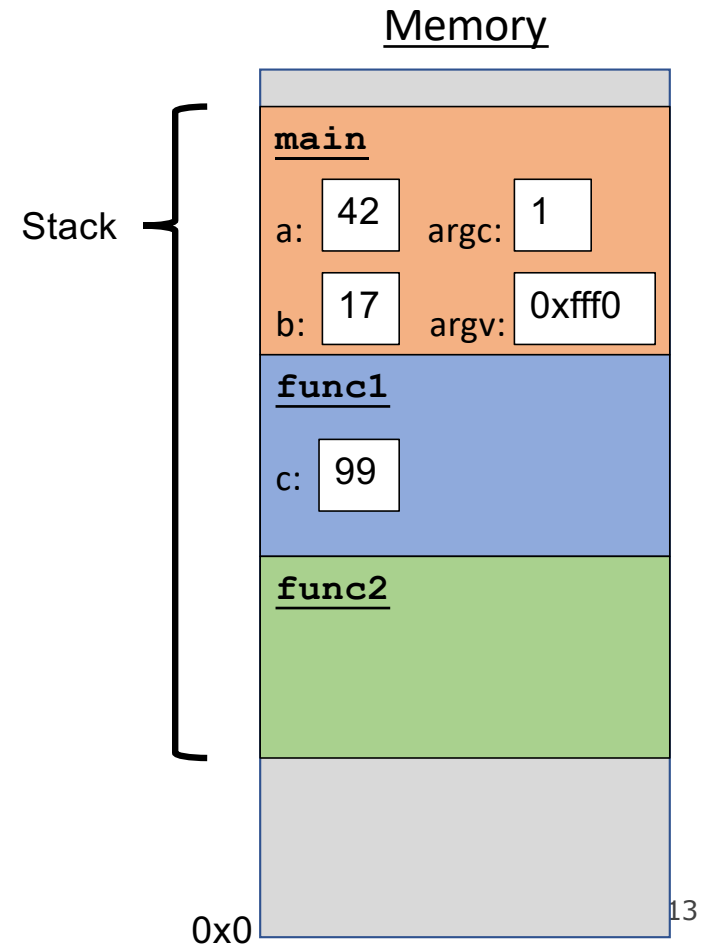
c: 99

0x0

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: `42`   argc: `1`

b: `17`   argv: `0xfff0`

func1

c: `99`

func2

0x0

13

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
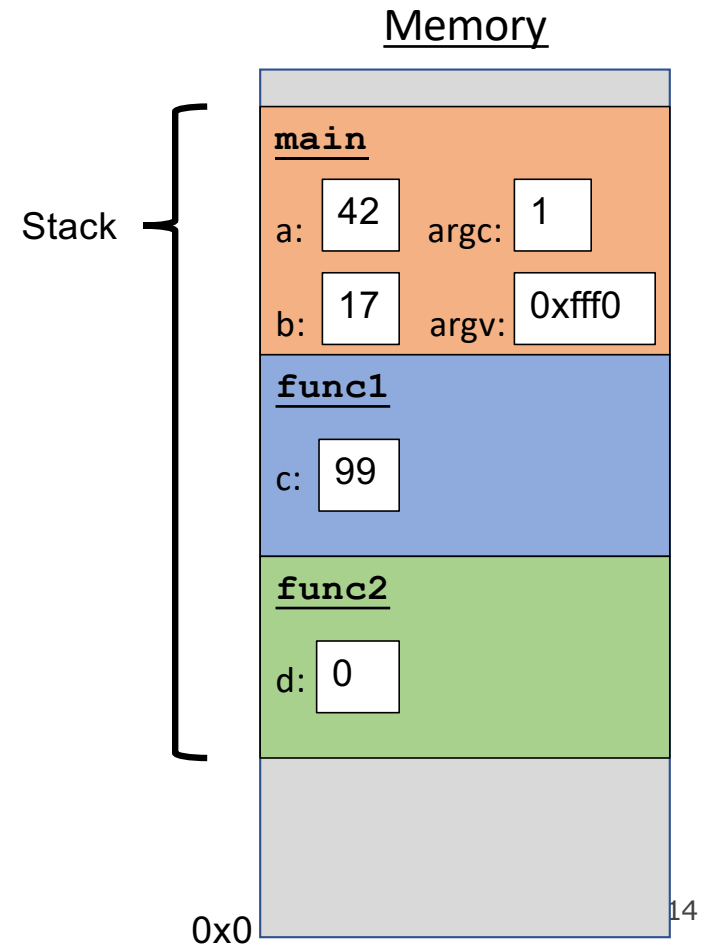
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

d: 0

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
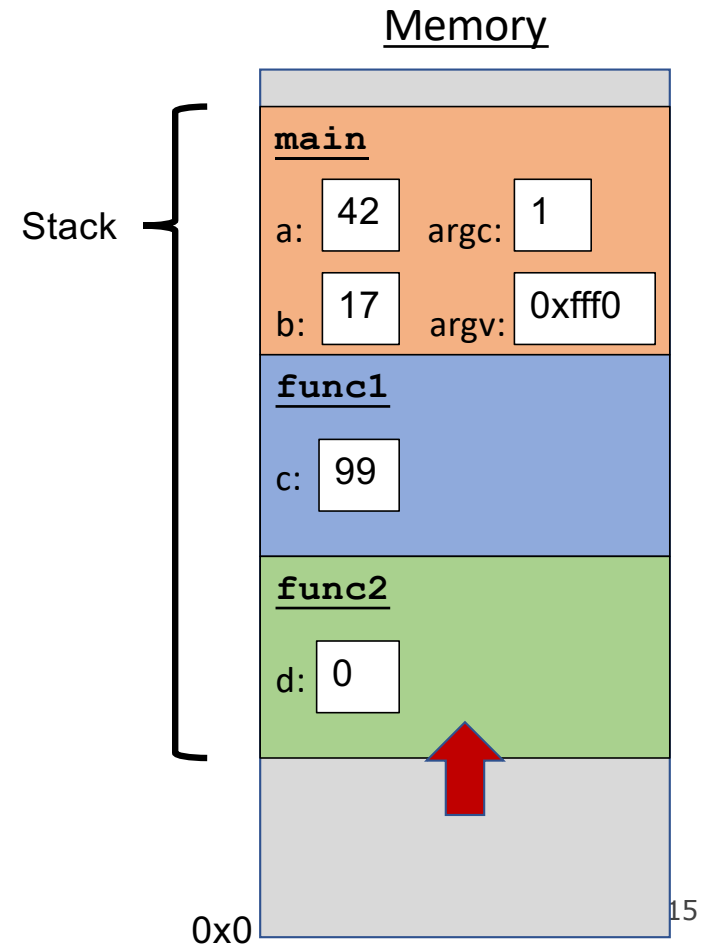
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

d: 0

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
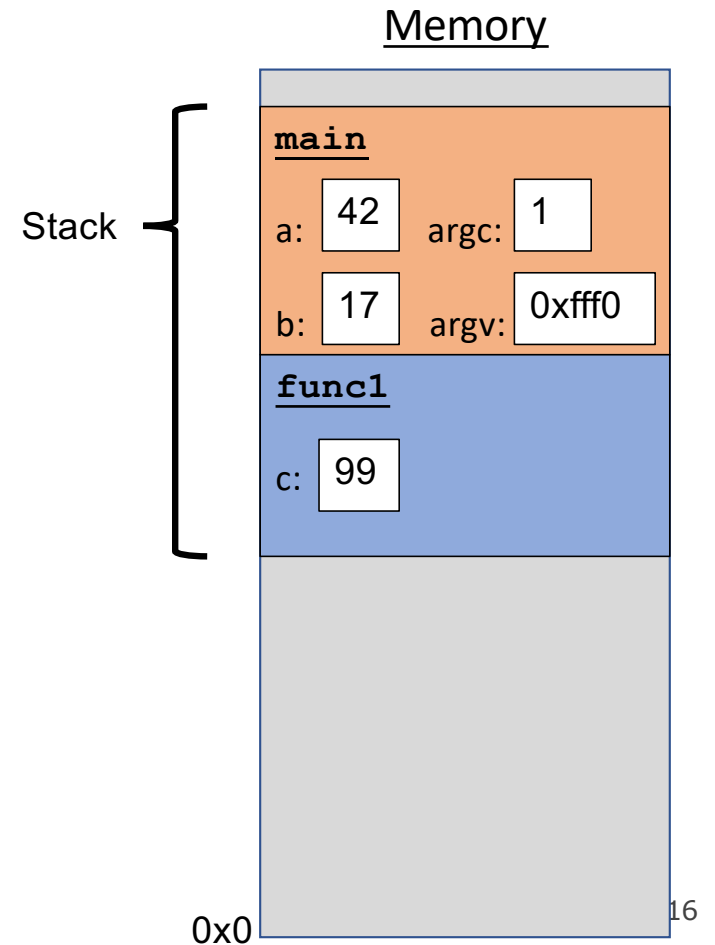
The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
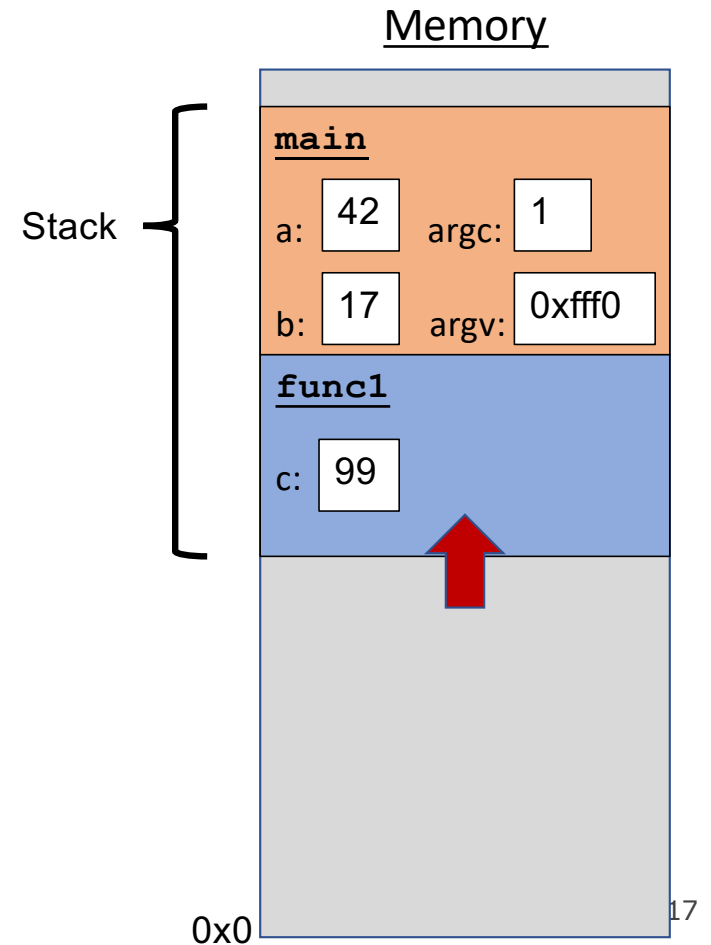
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0
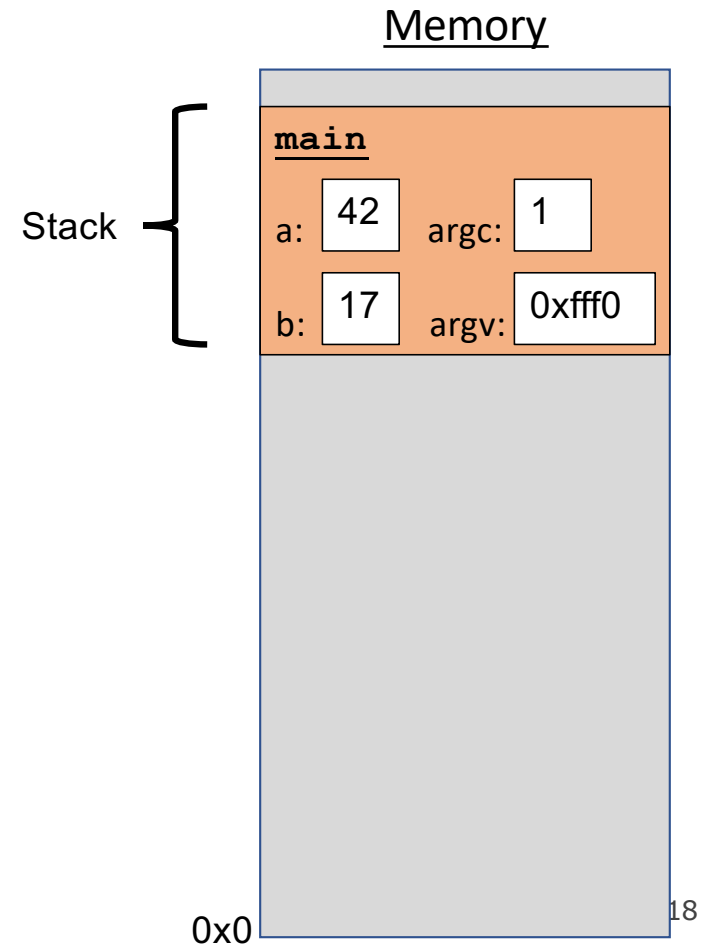
0x0

The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
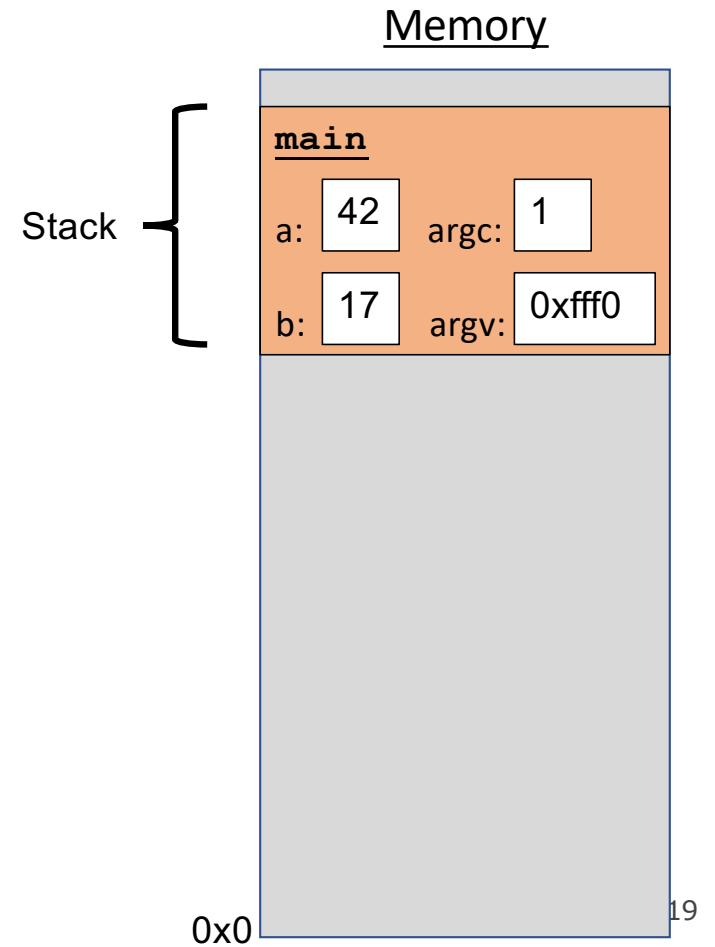
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```
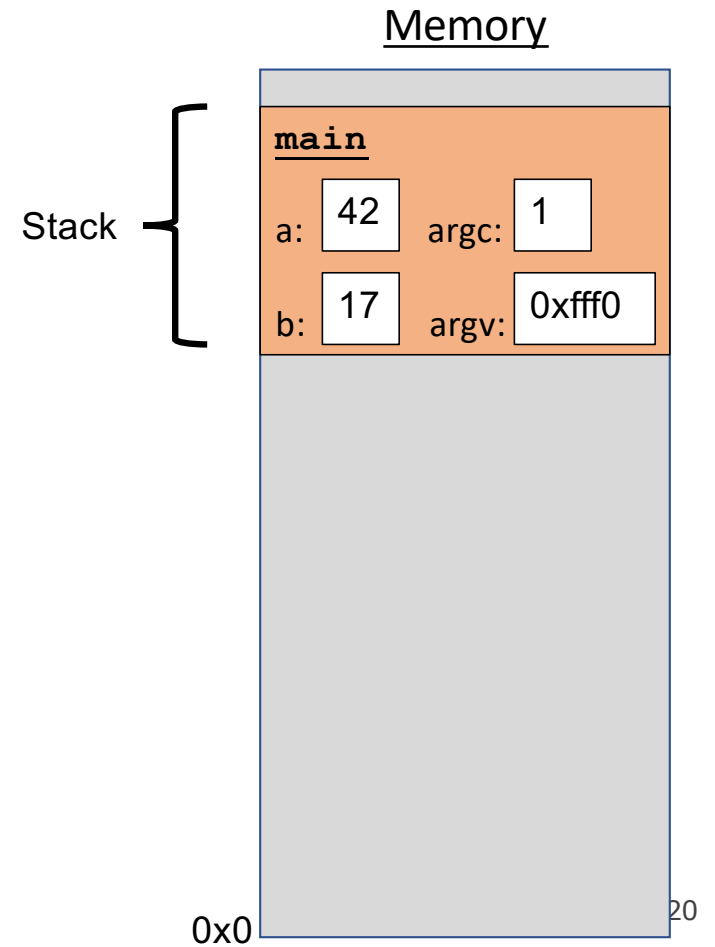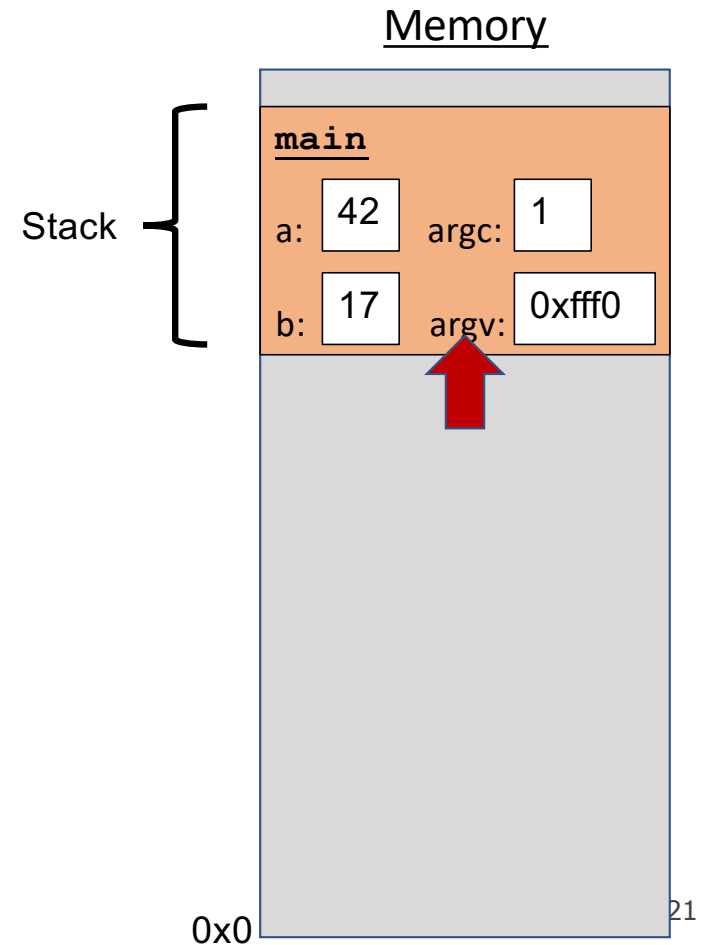
Memory

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

main
argc: 1
argv: 0xfff0

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

main
argc: 1   str:
argv: 0xfff0

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
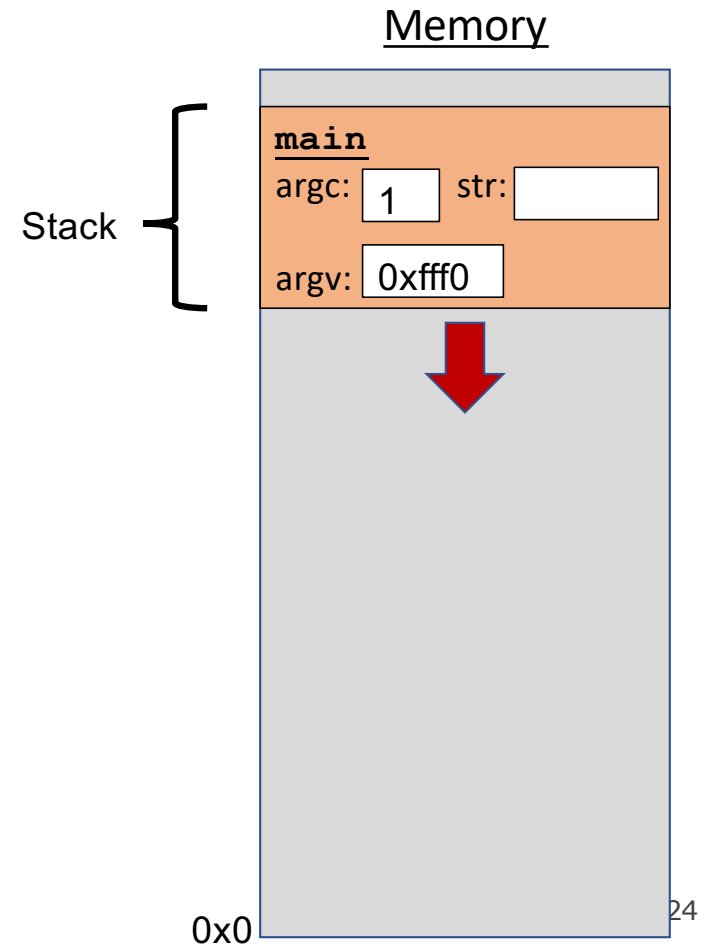
Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

new_str:

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
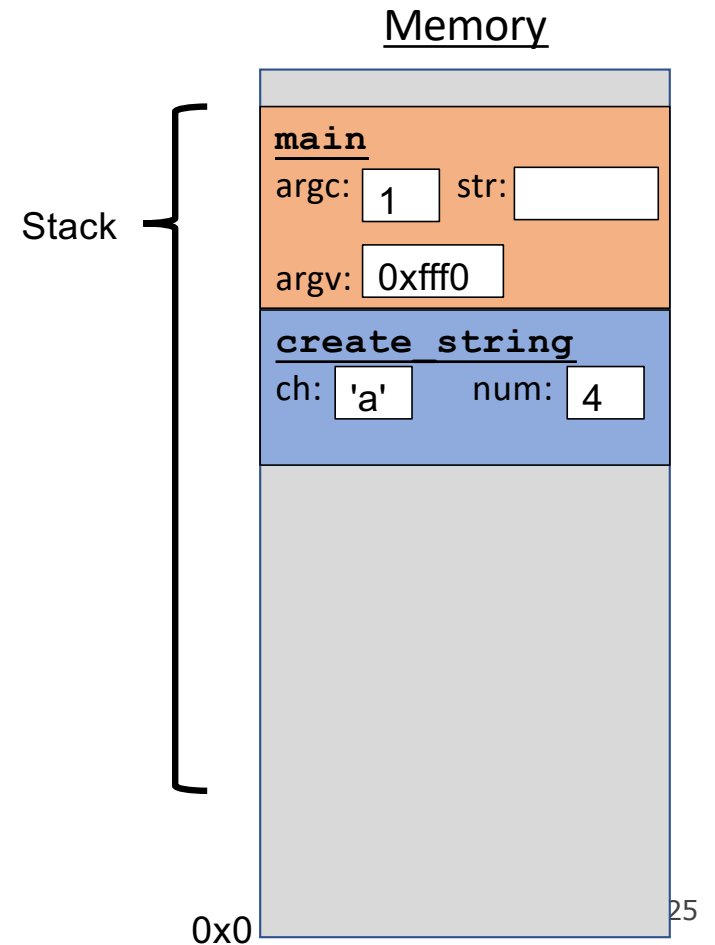
Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

'\0'
'a'
'a'
'a'
'a'

new_str:

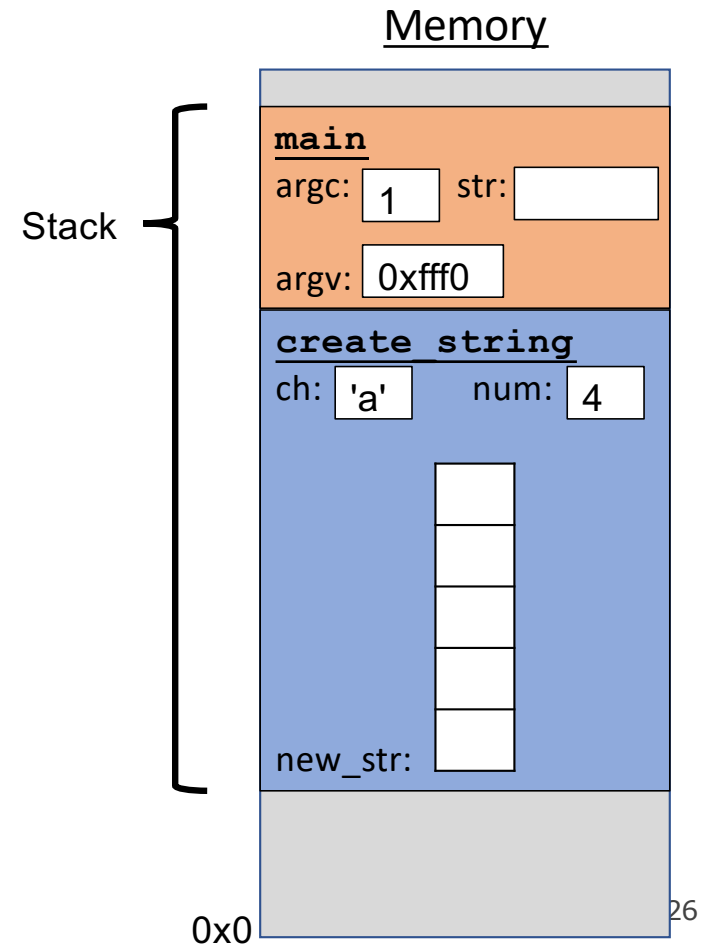0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

Returns e.g. 0xff50

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

'\0'
'a'
'a'
'a'
new_str: 'a'

0x0

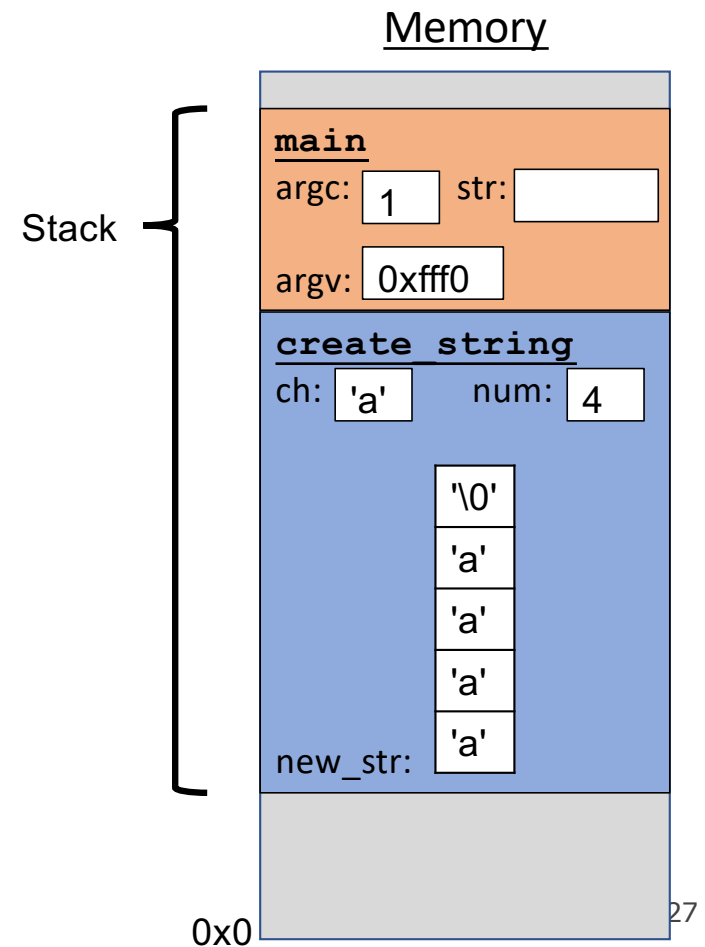# The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

main

argc: 1   str: 0xff50

argv: 0xfff0

create_string

ch: 'a'   num: 4

'\0'

'a'

'a'

'a'

new_str: 'a'

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

main
argc: 1   str: 0xff50
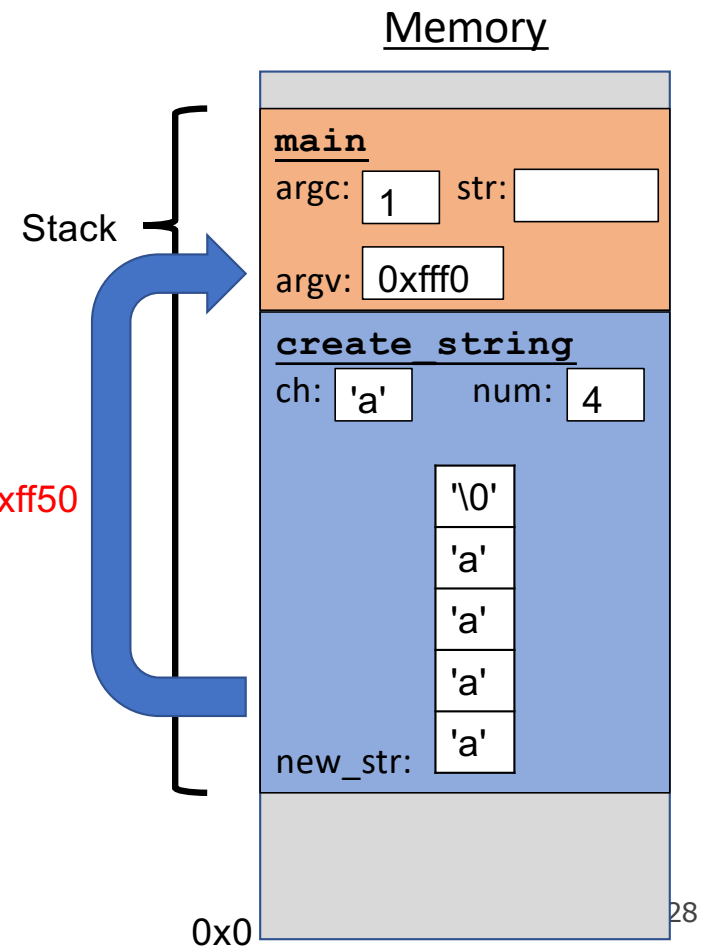argv: 0xfff0

0x0

The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
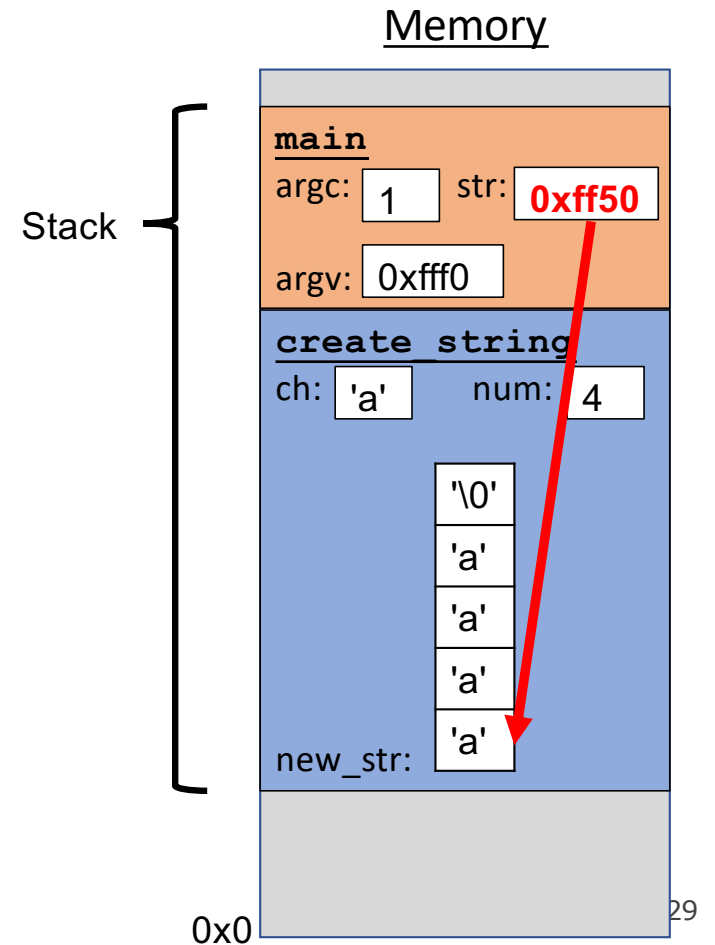
Memory

Stack

main
argc: 1   str: 0xff50
argv: 0xfff0

0x0

Problem: local variables go away when a function finishes. The characters will no longer exist, and the address will be for unknown memory!
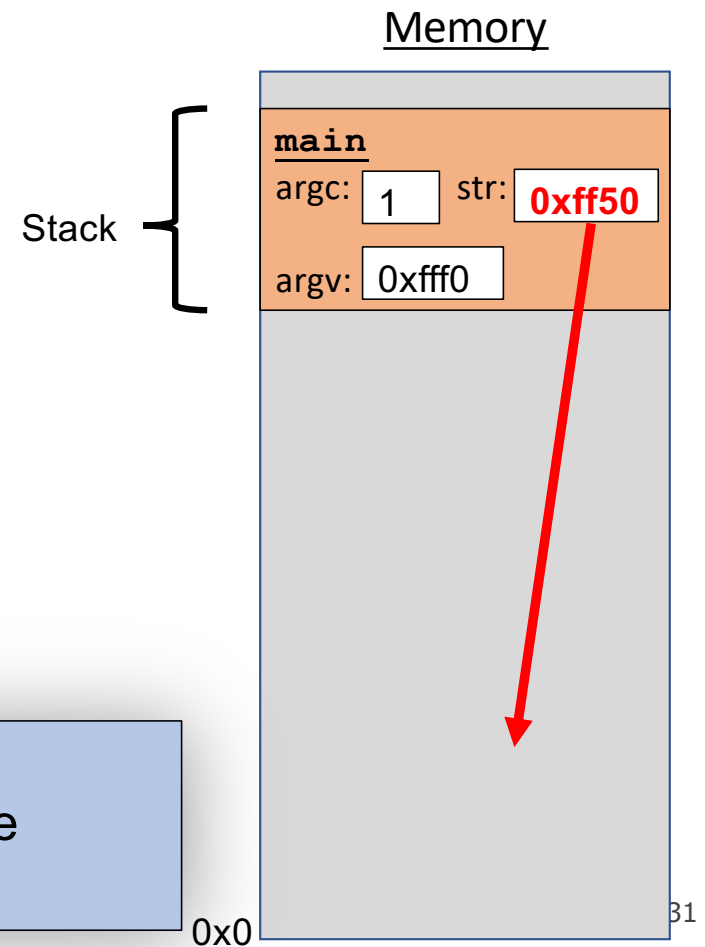
The Stack Failing Us

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
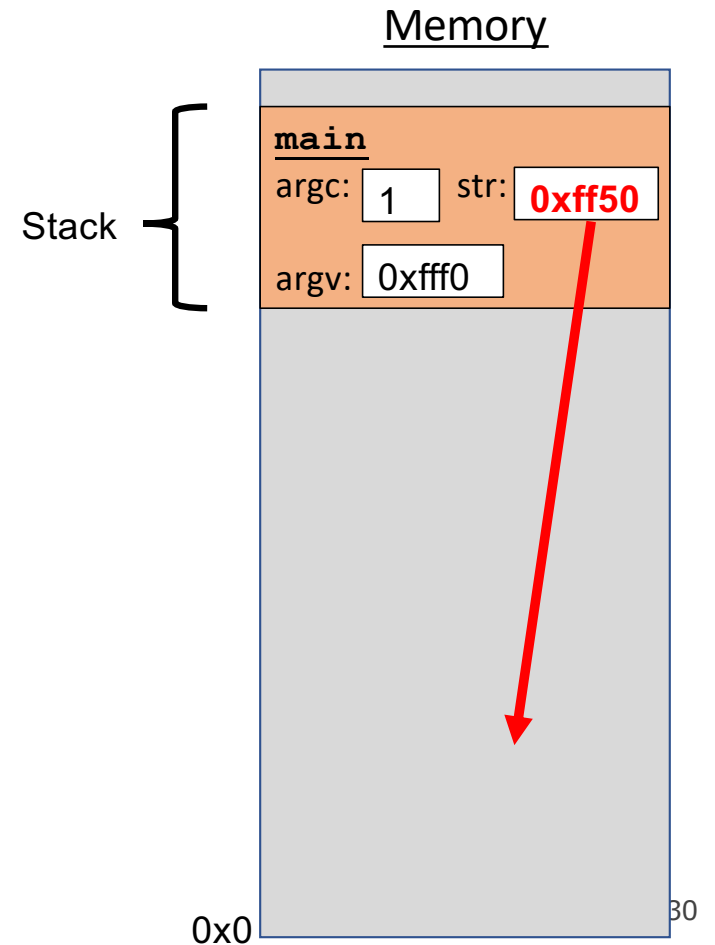
Sometimes, we can make the array in the caller and pass it as a parameter. But this isn't always possible if the size isn't known in advance.

Memory

Stack

main

argc: 1   str: **0xff50**

argv: 0xfff0

0x0

The Stack Failing Us

This is a problem! We need a way to allocate memory that persists even after a function exits.

The Heap

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Us: Hey C, is there a way to allocate this variable so it persists beyond the lifetime of the function allocating it?
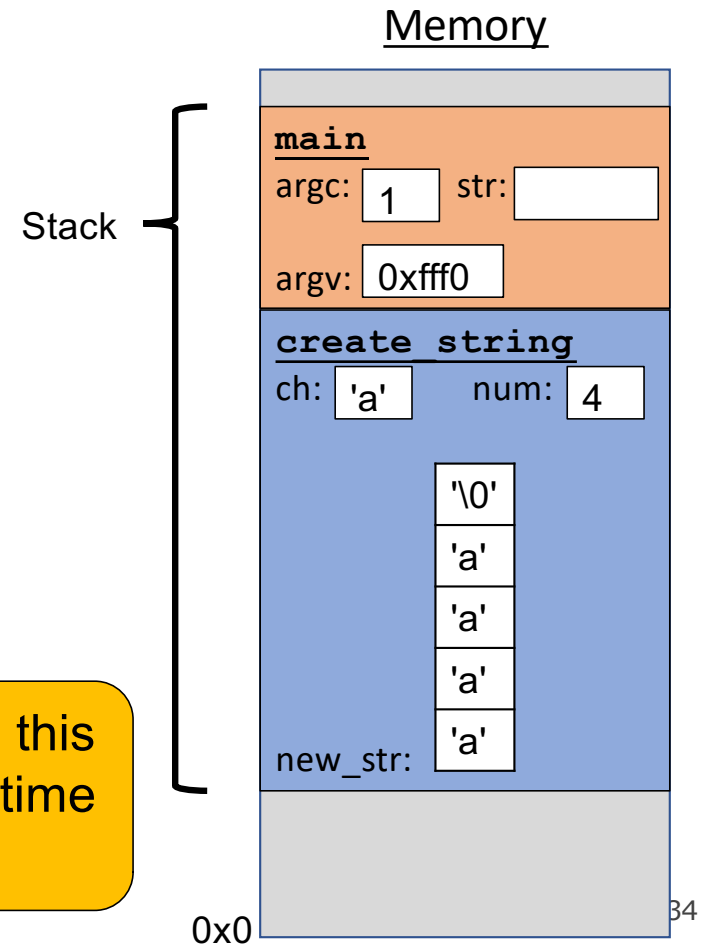
Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

```
'\0'
'a'
'a'
'a'
'a'
```
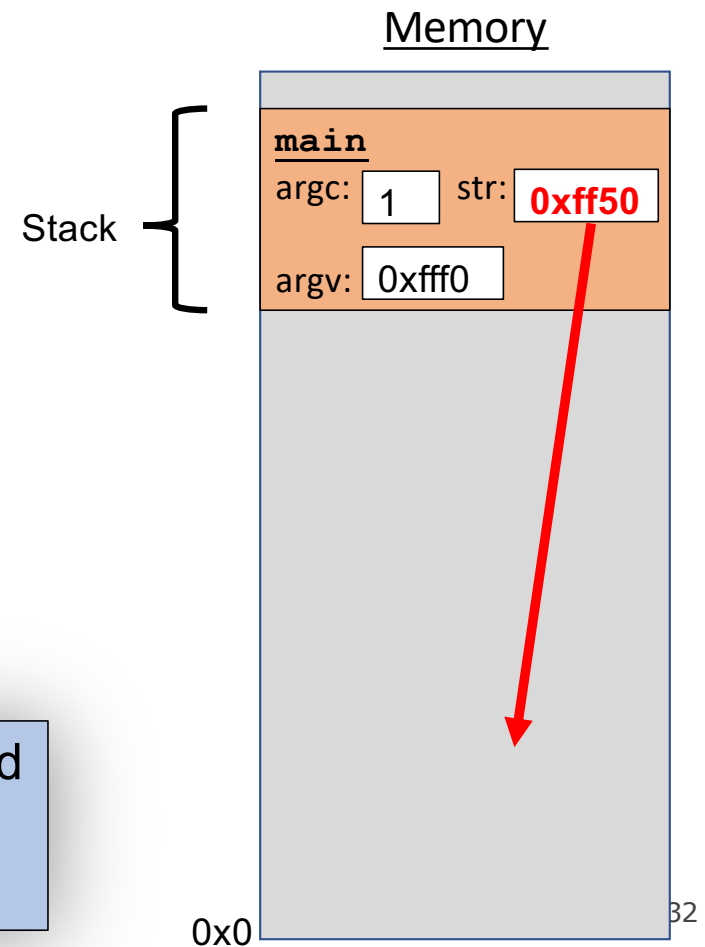new_str:

0x0

34

The Heap

```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

C: sure, but since I don't know when to deallocate it, it's your job to do eventually do that…

Memory

Stack

main	
argc: 1	str:
argv: 0xfff0	

create_string	
ch: 'a'	num: 4

	'\0'
	'a'
	'a'
	'a'
new_str:	'a'

0x0
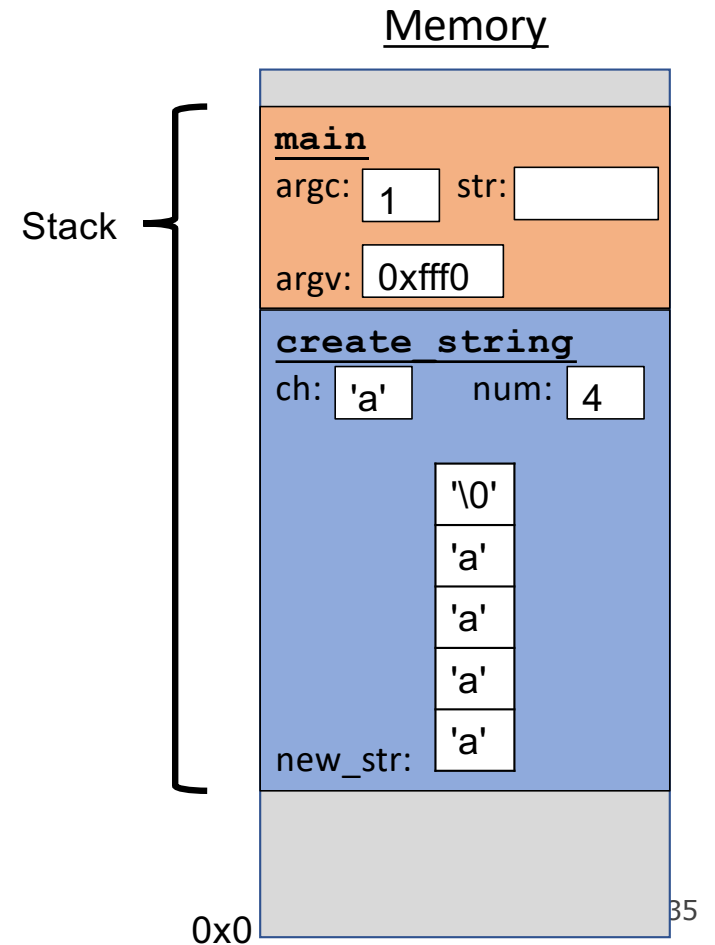
The Heap

- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you explicitly delete it yourself.

- Unlike the stack, the heap grows **upwards** as more memory is allocated.

The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.

0x7fffffff000   Stack ↓   8MB reserved

0x7ffff7ffe000   Shared library text/data   Sized for library

↑ Heap   Grows on demand

0x602010
0x600000   Global data   Sized for executable

0x400000   Text (machine code)

Low addresses deliberately unmapped

Working with the heap

Working with the heap consists of 3 core steps:

1. Allocate memory with `malloc`/`realloc`/`strdup`/`calloc`
2. Assert heap pointer is not `NULL`
3. Free memory when done using `free`.

The heap is **dynamic memory**, so you may encounter **runtime errors**, even if your code compiles!

malloc

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

To allocate memory on the heap, use the **malloc** function and specify the number of bytes you'd like.

- This function returns a pointer to *the **leading address** of the new memory*. It doesn't know or care whether it will be used for an array, a single block of memory, a struct, or anything else.
- **void *** denotes a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* zeroed out!
- If **malloc** returns **NULL**, the heap couldn't service the allocation request.

The Heap

```c
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
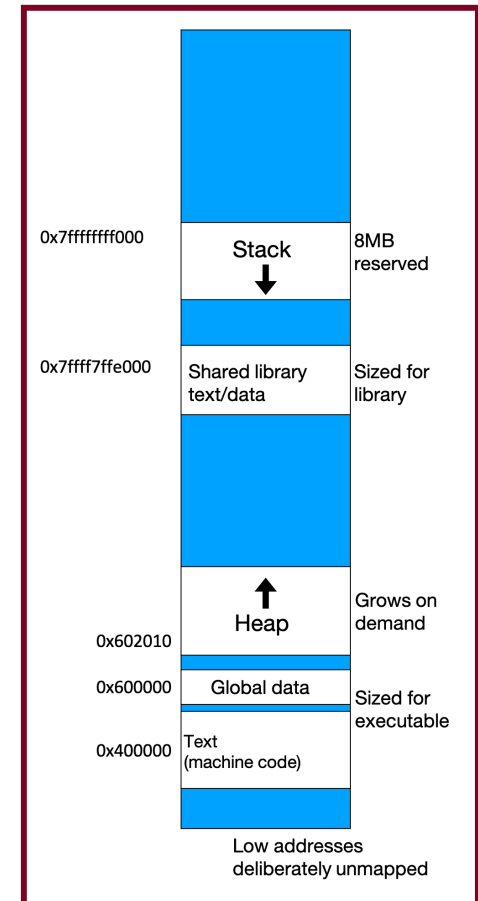
Memory

Stack

Heap

0x0

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4
new_str: 0xed0

'\0'
'a'
'a'
'a'
'a'

The Heap

```
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

Returns 0xed0

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

new_str: **0xed0**

Heap
'\0'
'a'
'a'
'a'
'a'

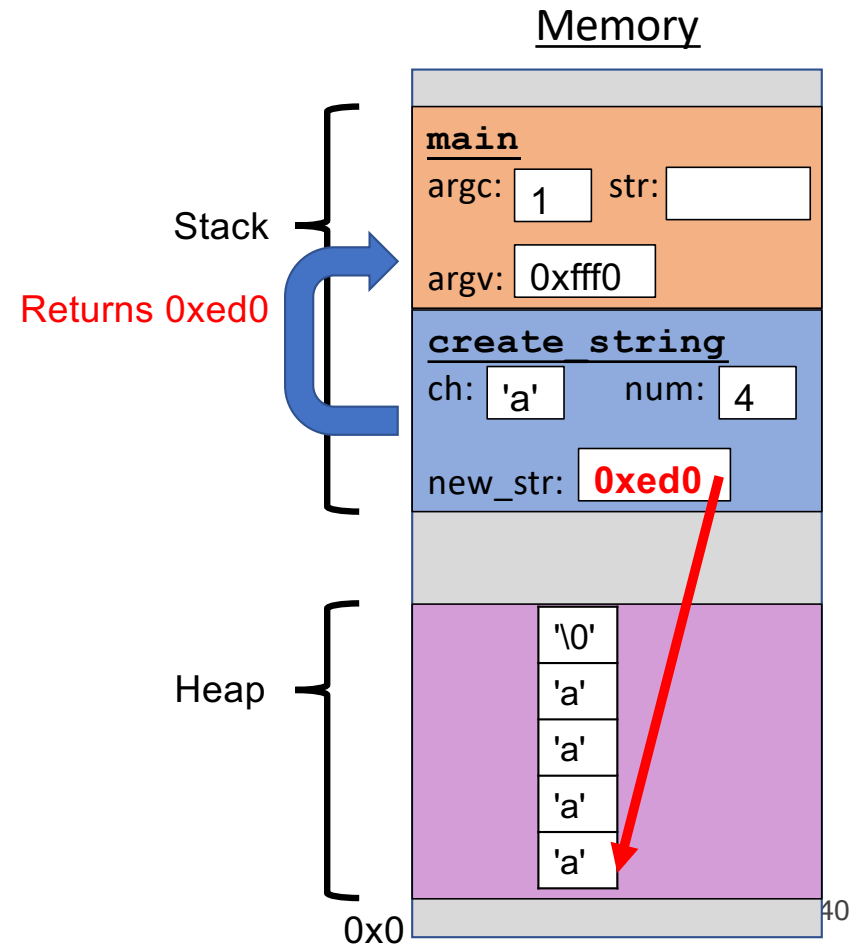0x0

40

The Heap

```c
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```

Memory

Stack

Returns 0xed0

main

argc: 1   str: **0xed0**

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str: **0xed0**

Heap

'\0'

'a'

'a'

'a'

'a'

0x0

41

The Heap

```c
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
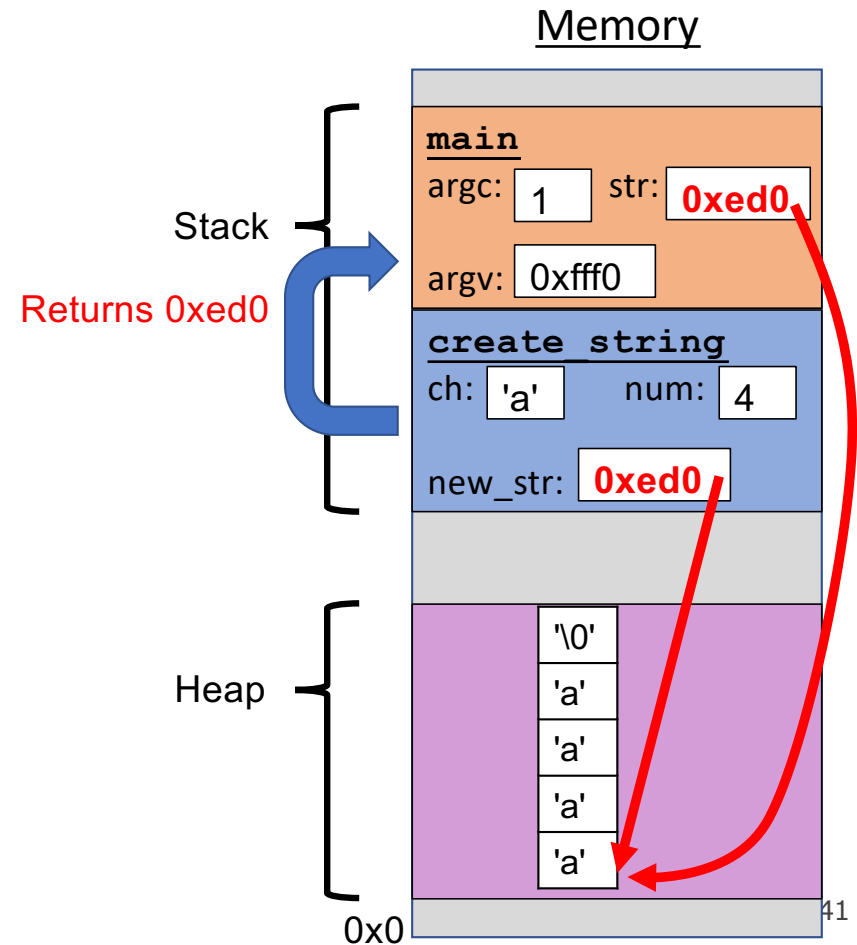
Memory

Stack

main
argc: 1   str: 0xed0
argv: 0xfff0

Heap

'\0'
'a'
'a'
'a'
'a'

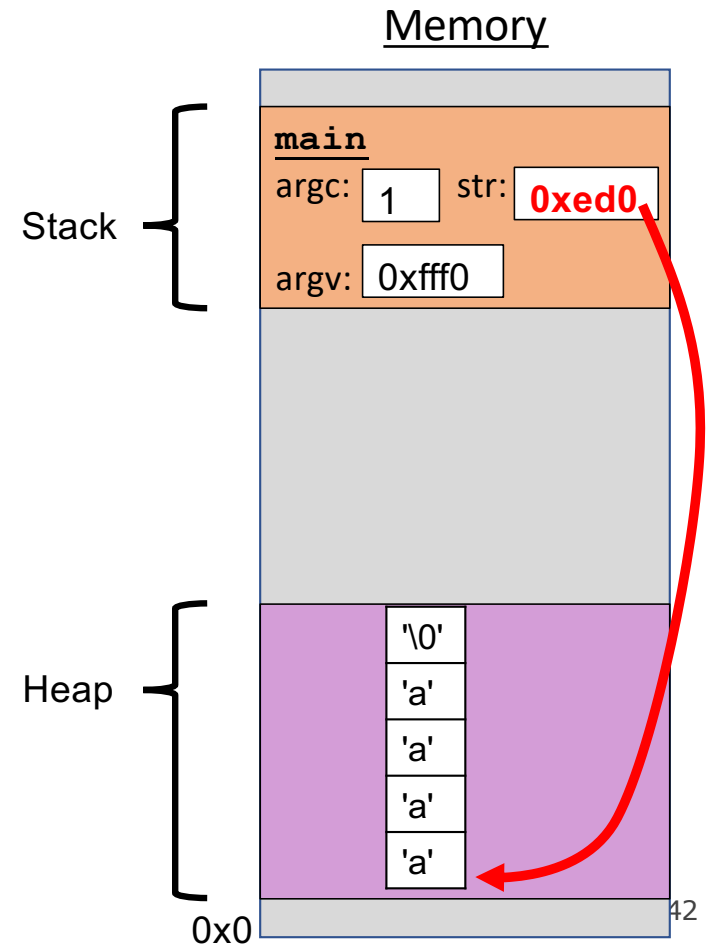0x0

42

The Heap

```c
char *create_string(char ch, int num) {
    char *new_str = malloc(num + 1);
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```
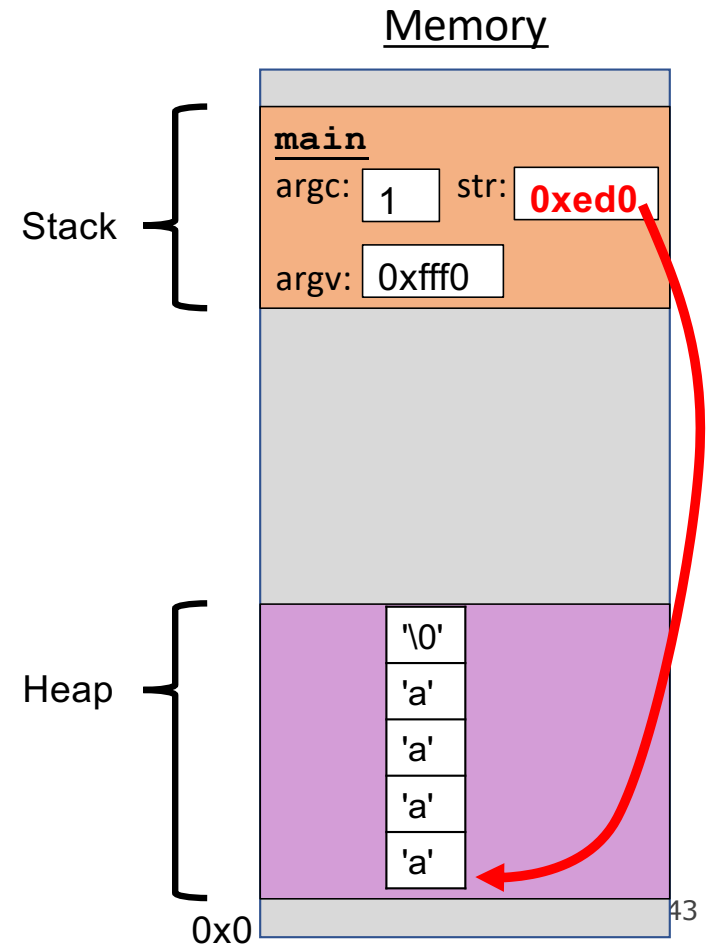
Memory

Stack

main

argc: 1   str: 0xed0

argv: 0xfff0

Heap

'\0'
'a'
'a'
'a'
'a'

0x0

43

The Heap

```c
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0; // should free str, we will soon
}
```
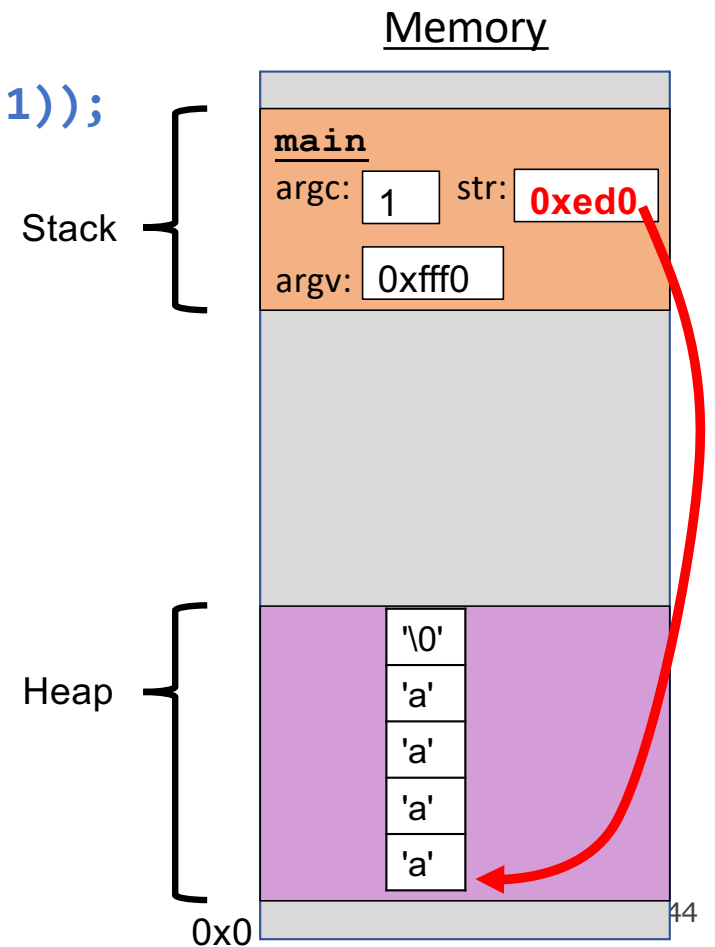
Memory

main
argc: 1   str: 0xed0
argv: 0xfff0

Stack

Heap

'\0'
'a'
'a'
'a'
'a'

0x0

44

Exercise: `malloc` multiples

Let's write a function that returns an array of the first **len** multiples of **mult**.

```c
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

Line 2: How should we declare `arr`?

A. `int arr[len];`
B. `int arr[] = malloc(sizeof(int));`
C. `int *arr = malloc(sizeof(int) * len);`
D. `int *arr = malloc(sizeof(int) * (len + 1));`

Exercise: `malloc` multiples

Let's write a function that returns an array of the first **len** multiples of **mult**.

```c
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

Line 2: How should we declare `arr`?

A. `int arr[len];`
B. `int arr[] = malloc(sizeof(int));`
C. `int *arr = malloc(sizeof(int) * len);`
D. `int *arr = malloc(sizeof(int) * (len + 1));`

- Use a pointer to store the address returned by malloc.
- malloc's argument is the **number of bytes** to allocate.
- ⚠ **This code is missing an assertion.**

46

Always assert with the heap

Let's write a function that returns an array of the first **len** multiples of **mult**.

```c
1 int *array_of_multiples(int mult, int len) {
2     int *arr = malloc(sizeof(int) * len);
3     assert(arr != NULL);
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

- If an allocation error occurs (e.g., out of heap memory), malloc will return NULL. This is an important case to check **for robustness**.

- **assert** will end the program if the supplied condition is false. An allocation error is a big deal, so we should terminate the program if we detect one.

Other heap allocations: `calloc`

```
void *calloc(size_t nmemb, size_t size);
```

calloc is like `malloc` that **zeros out** the memory for you—thanks, `calloc`!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- calloc is more expensive than malloc because it zeros out memory. Use only when necessary!

Other heap allocations: `strdup`

```
char *strdup(char *s);
```

strdup is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **malloc** and copy in the string yourself.

```
char *str = strdup("Hello, world!");  // on heap
str[0] = 'h';
```

You might imagine **strdup** can be implemented in terms of **malloc** + **strcpy**. (It is!)

Cleaning Up with `free`

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

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.

- To do this, use the **free** command and pass in the *starting address on the heap for the memory you no longer need.*

- Example:

```
char *bytes = malloc(4);

…

free(bytes);
```

free

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

When you free an allocation, you are freeing up what it *points* to. You are not deallocating the pointer itself. You can still use the pointer variable to address something new.

```
char *str = strdup("hello");
...
free(str);
str = strdup("hi");
```

realloc

```
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.

- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.

- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

```c
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

Cleaning Up with free and realloc

You only need to free the new memory coming out of `realloc`—the previous (smaller) one was already reclaimed by `realloc`.

```c
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Double frees bad

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);        ⬅ ✅
…
free(ptr);          ⬅ ❌ Memory at this
                         address was already
                         freed!
```

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);        ⬅ ✅
…
free(ptr + 1);      ⬅ ❌
```

55

Memory Leaks

A **memory leak** is when you do not free memory you previously allocated.

```c
char *str = strdup("hello");
...
str = strdup("hi"); // memory leak!  Lost previous str
```

Memory Leaks

- A memory leak is when you allocate memory on the heap, but do not free it.

- Your program should be responsible for cleaning up any memory it allocates but no longer needs.

- If you never free any memory and allocate a large amount, you may run out of memory in the heap!

- However, memory leaks rarely (if ever) cause crashes.

- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.

- Valgrind is a very helpful tool for finding memory leaks!

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- NULL on failure, so check with `assert`

- Memory is contiguous; it is not recycled unless you call free

- `realloc` preserves existing data

- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)

- If you use after `free`, or if `free` is called twice on a location.

- If you `realloc/free` non-heap address

Engineering principles: stack vs heap

Stack (for local variables)   **Heap** (dynamic memory)

- **Fast**
Fast to allocate/deallocate; okay to oversize

- **Convenient**.
Automatic allocation/ deallocation;
declare/initialize in one step

- **Reasonable type safety**
Thanks to the compiler

⚠ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠ **Somewhat inflexible**
Cannot add/resize at runtime, scope
dictated by control flow in/out of functions

Engineering principles: stack vs heap

Stack (for local variables)

- **Fast**
  Fast to allocate/deallocate; okay to oversize

- **Convenient**.
  Automatic allocation/ deallocation;
  declare/initialize in one step

- **Reasonable type safety**
  Thanks to the compiler

- ⚠ **Not especially plentiful**
  Total stack size fixed, default 8MB

- ⚠ **Somewhat inflexible**
  Cannot add/resize at runtime, scope
  dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful**.
  Can provide more memory on demand!

- **Very flexible.**
  Runtime decisions about how much/when to
  allocate, can resize easily with realloc

- **Scope under programmer control**
  Can precisely determine lifetime

- ⚠ **Lots of opportunity for error**
  Low type safety, forget to allocate/free
  before done, allocate wrong size, etc.,
  Memory leaks (much less critical)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.

- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory's lifetime and/or memory must persist beyond a function call

Example: Pig Latin

Let's write a program that can convert text to Pig Latin! Simplified Pig Latin rules:

- If the word starts with a vowel, append "way": *apple -> appleway*

- Otherwise, move all starting consonants to the end and append "ay": *bridge -> idgebray*

We want to write a function **char *pig_latin(const char *in)** that returns the Pig Latin version of the given string.

- Good use case for heap allocation – array size is unknown until we convert it to Pig Latin! We'll create and return a heap-allocated string.

- The caller must free the string when it is done.

Example: Pig Latin

We will also see an example of how to uncover memory leaks using Valgrind.

valgrind --leak-check=full --show-leak-kinds=all [program info here]

Demo: Pig Latin + Valgrind

pig_latin.c