



# CS107, Lecture 10

## Stack and Heap

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap  
[Ed Discussion](#)

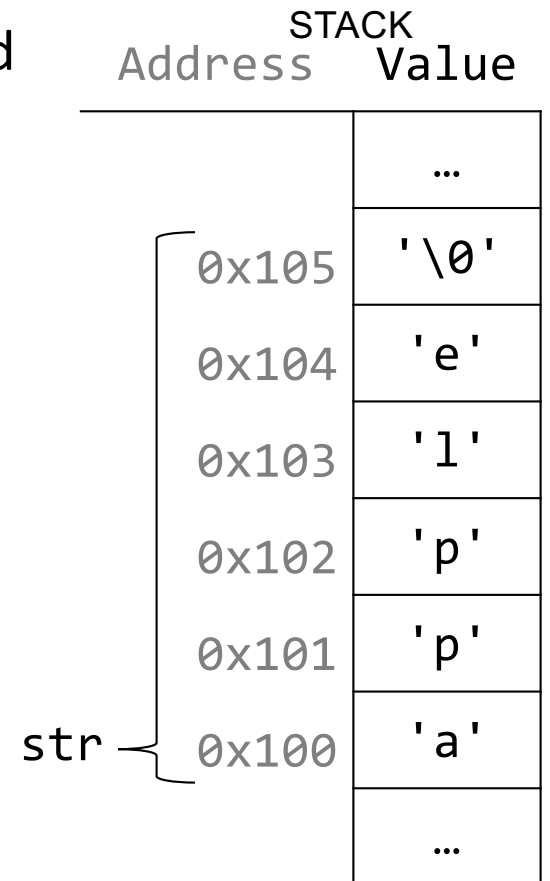
# Arrays

When you declare an array, contiguous memory is allocated on the stack to store the contents of the entire array.

```
char str[6];  
strcpy(str, "apple");
```

The array variable (e.g. **str**) is not a pointer; it refers to the entire array contents. In fact, **sizeof** returns the size of the entire array!

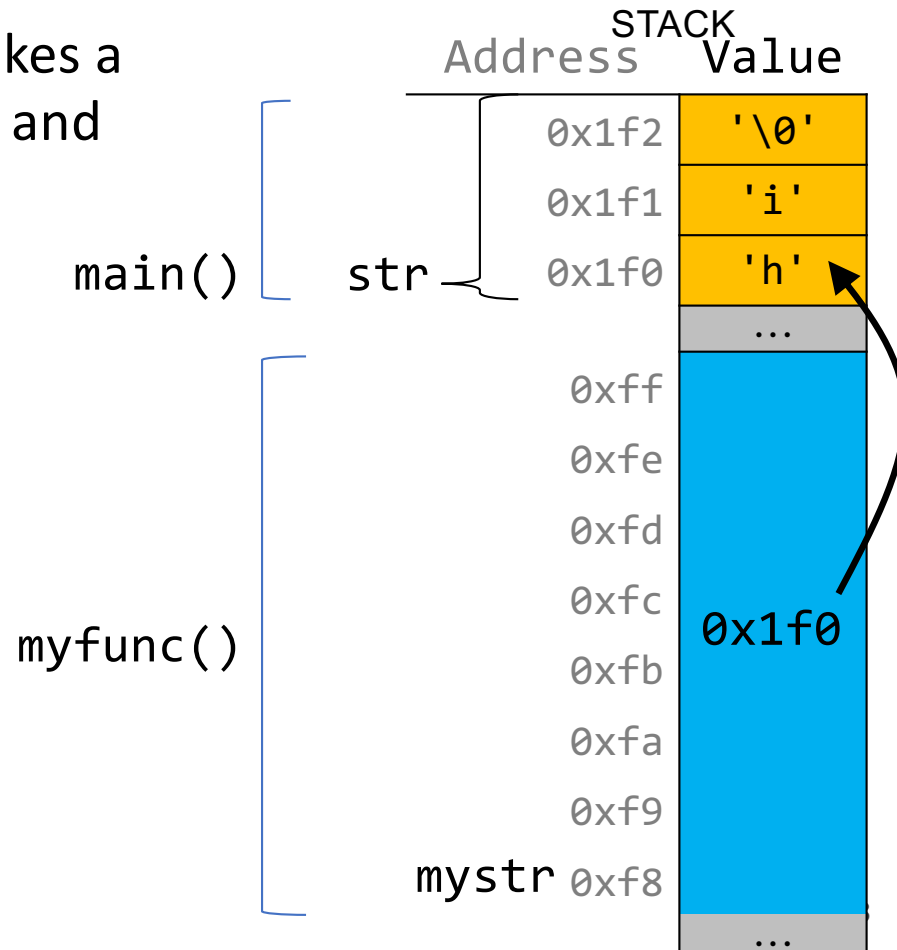
```
size_t num_bytes = sizeof(str); // 6
```



# Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element*, and passes it (a pointer) to the function.

```
void myfunc(char *mystr) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    char str[3];  
    strcpy(str, "hi");  
    myfunc(str);  
    ...  
}
```

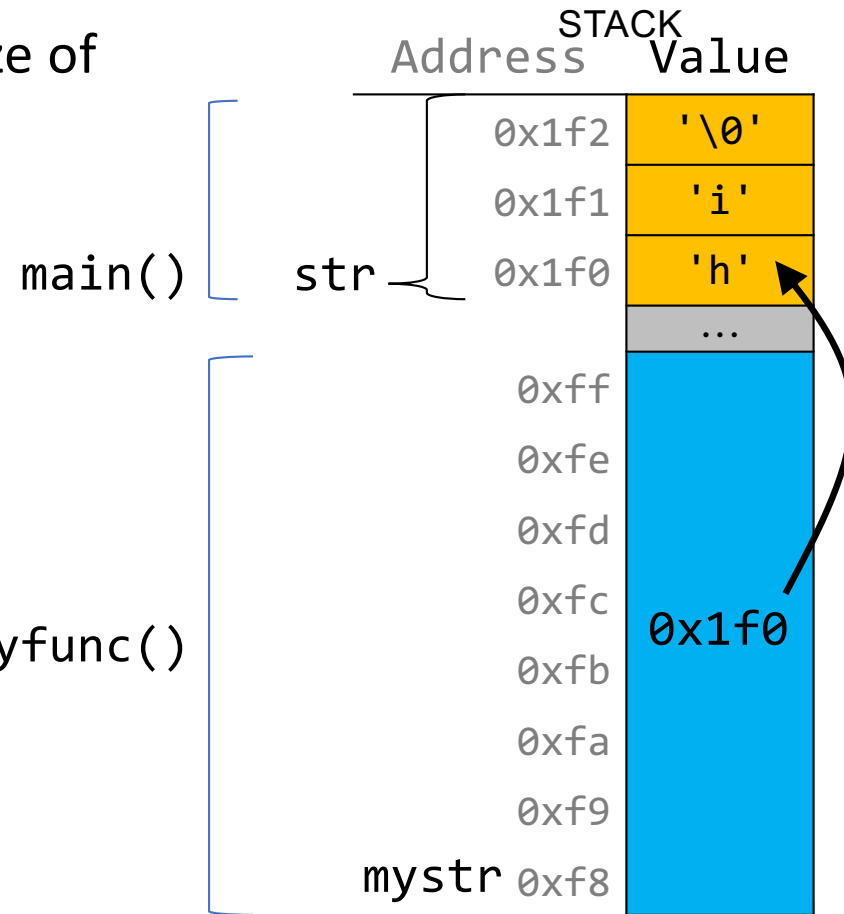



# Arrays as Parameters

This also means we can no longer get the full size of the array using **sizeof**, because now it is just a pointer.

```
void myfunc(char *myStr) {  
    size_t size = sizeof(myStr); // 8  
}
```

```
int main(int argc, char *argv[]) {  
    char str[3];  
    strcpy(str, "hi");  
    size_t size = sizeof(str); // 3  
    myfunc(str);  
    ...  
}
```





**sizeof** returns the size of an array, or 8 for a pointer. Therefore, when we pass an array as a parameter, we can no longer use **sizeof** to get its full size.

# Pointer Arithmetic

When you do pointer arithmetic, you are adjusting the pointer by a certain *number of places* (e.g., characters).

```
char *str = "apple"; // e.g. 0xff0
char *str1 = str + 1; // e.g. 0xff1
char *str3 = str + 3; // e.g. 0xff3

printf("%s", str); // apple
printf("%s", str1); // pple
printf("%s", str3); // le
```

DATA SEGMENT	
Address	Value
	...
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	...

# Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
int numbers[] = {52, 23, 12, 34, 16, 1};
int *nums = numbers;    // e.g., 0xff0
int *nums1 = nums + 1; // e.g., 0xff4
int *nums3 = nums + 3; // e.g., 0xffc

printf("%d", *nums);    // 52
printf("%d", *nums1);  // 23
printf("%d", *nums3);  // 34
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

# Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
int numbers[] = {52, 23, 12, 34, 16, 1};
int *nums = numbers;    // e.g., 0xff0
int *nums3 = nums + 3; // e.g., 0xffc
int *nums2 = nums3 - 1; // e.g., 0xff8

printf("%d", *nums);    // 52
printf("%d", *nums2);   // 12
printf("%d", *nums3);   // 34
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...

# Pointer Arithmetic

When you use bracket notation with a pointer, you are actually *performing pointer arithmetic and dereferencing*:

```
char *str = "apple";    // e.g. 0xff0

// both of these add four places to str,
// and then dereference to get the char there.
// e.g. get memory at 0xff4.
char fifth = str[4];    // 'e'
char fifth = *(str + 4); // 'e'
```

DATA SEGMENT	
Address	Value
	...
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	...


# Pointer Arithmetic

Pointer arithmetic with two pointers does *not* give the byte difference. Instead, it counts the number of **quantum elements** in between the two addresses.

```
int numbers[] = {52, 23, 12, 34, 16, 1};  
int *nums = numbers;           // e.g., 0xff0  
int *nums3 = nums + 3;        // e.g., 0xffc  
size_t diff = nums3 - nums;   // 3
```

STACK

Address	Value
	...
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	...



**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 using pointers (last time)
- Introduces us to the heap and the allocation of memory that we manage ourselves (this time)

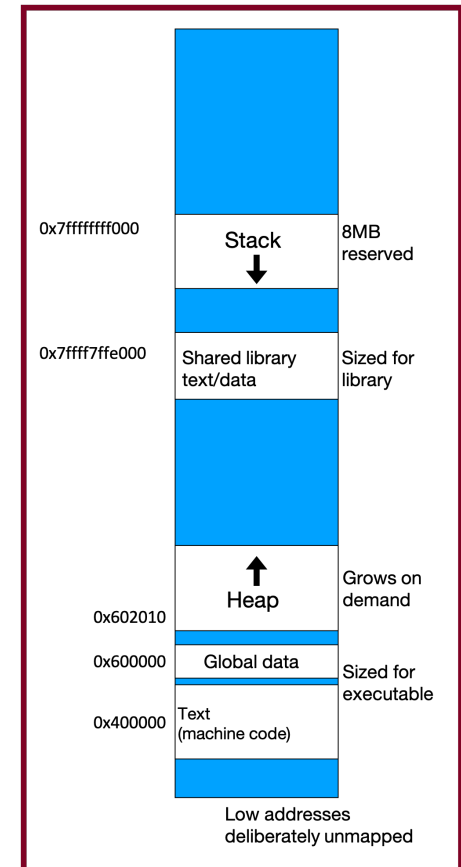
**assign3:** implement a function using resizable arrays to read lines of any length from a file and write 2 programs using that function to print the last N lines of a file and print just the unique lines of a file. These programs emulate the **tail** and **uniq** Unix builtins.

# Learning Goals

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

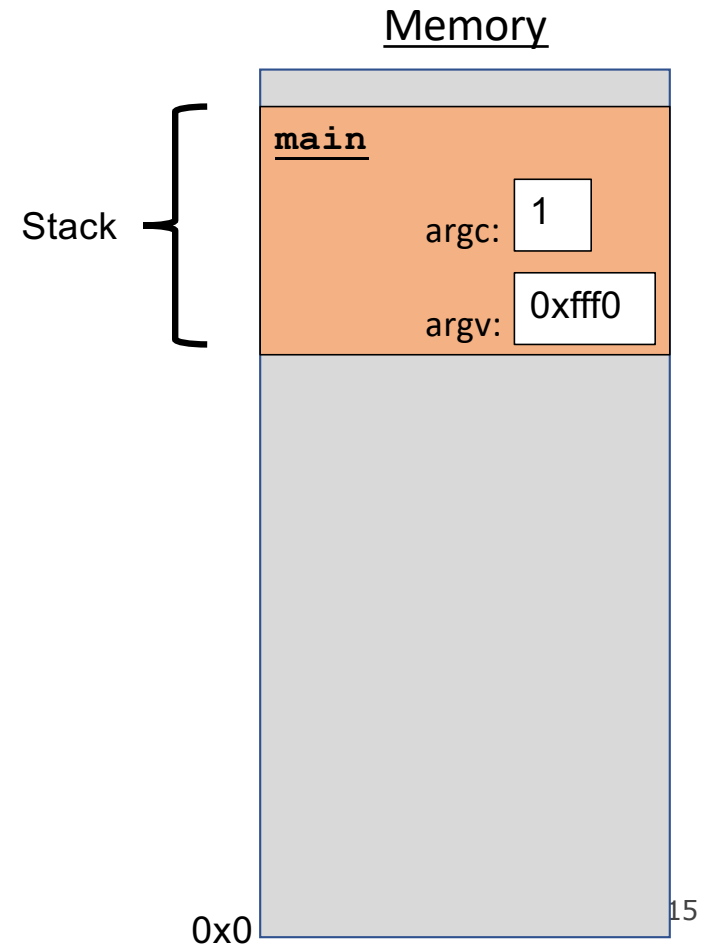
# Memory Layout

- 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 exits.



# 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();  
    func2();  
    printf("Done.");  
    return 0;  
}
```

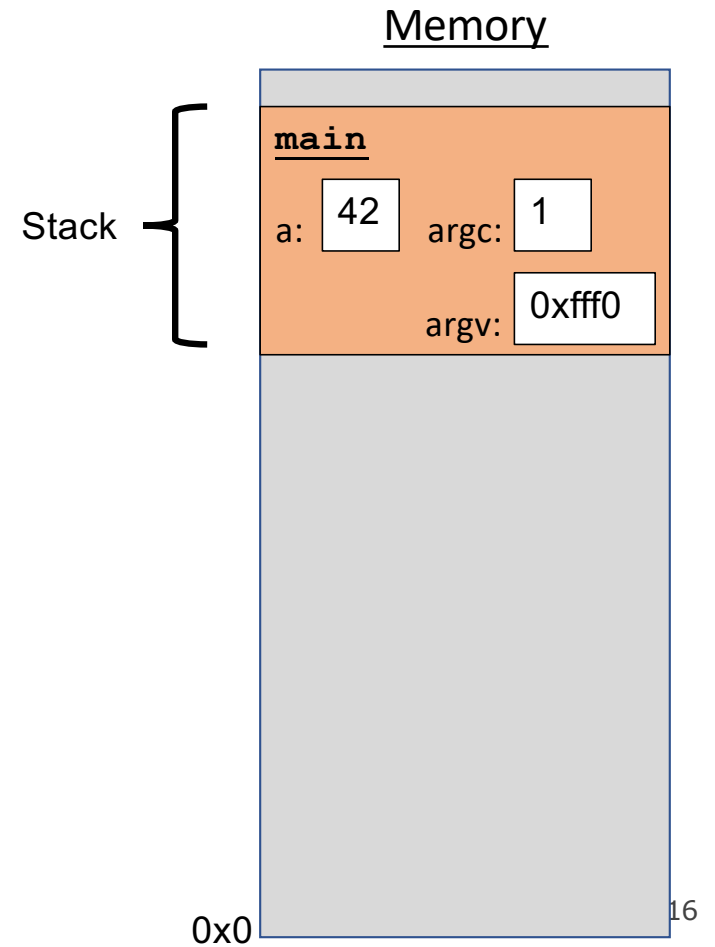


# The Stack

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

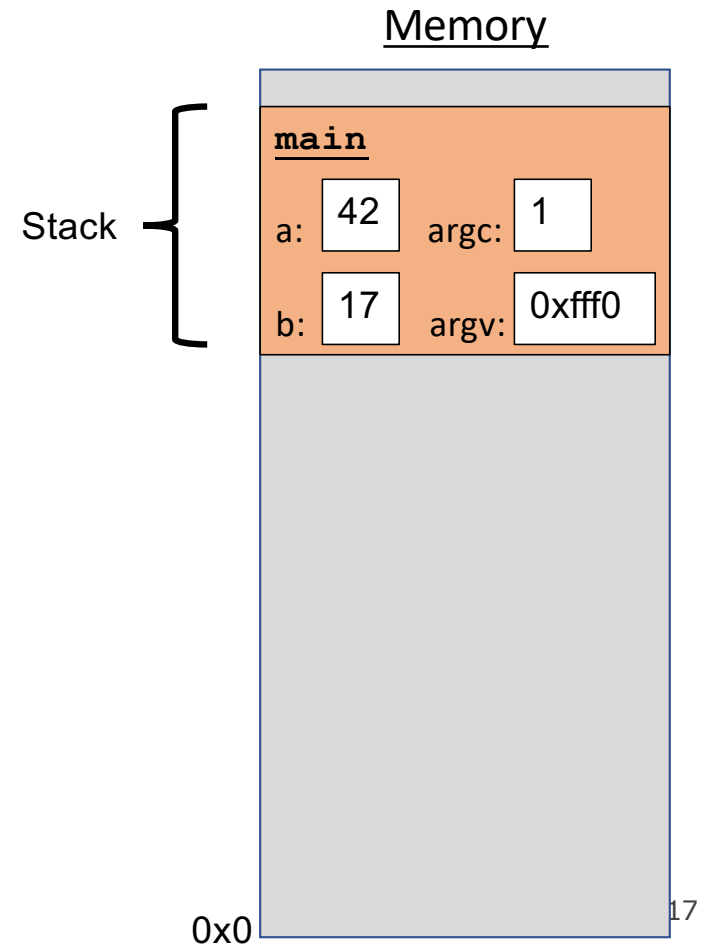
void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    printf("Done.");
    return 0;
}
```



# 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();  
    func2();  
    printf("Done.");  
    return 0;  
}
```

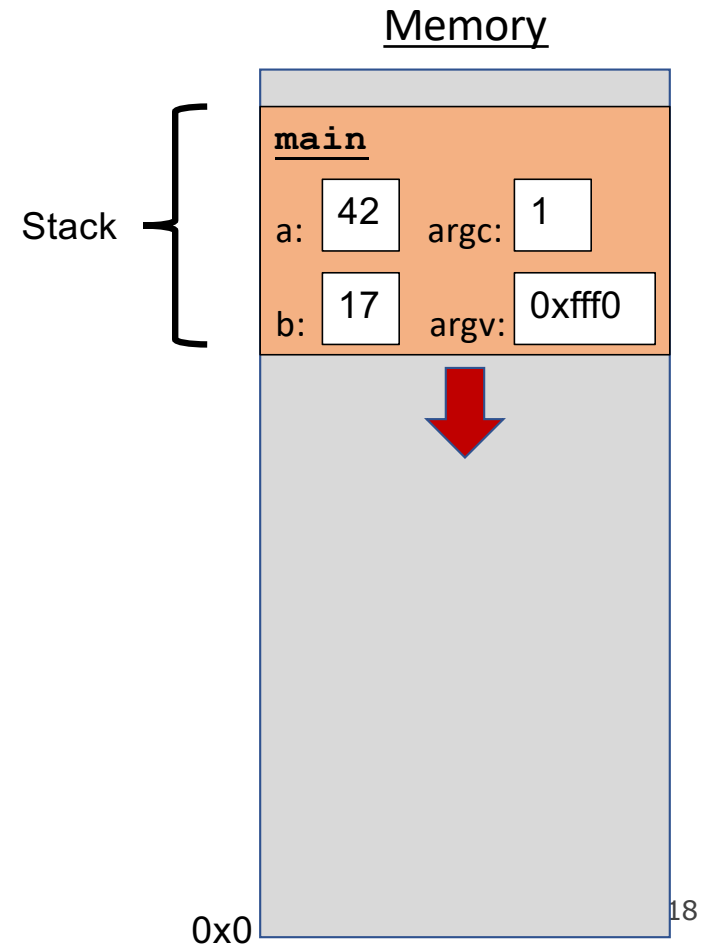


# The Stack

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

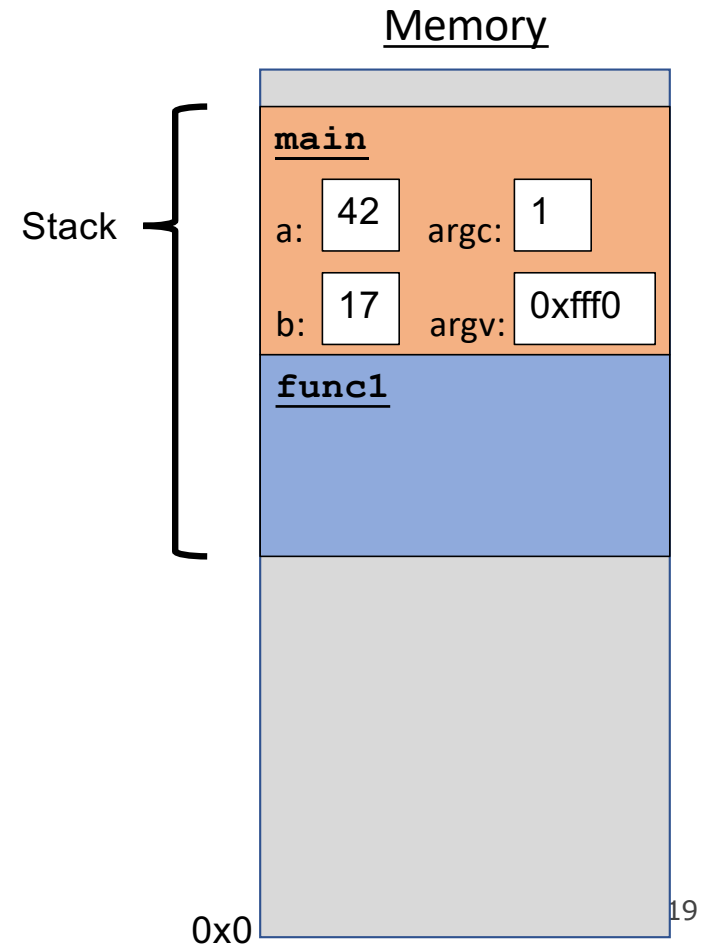
void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    printf("Done.");
    return 0;
}
```



# 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();  
    func2();  
    printf("Done.");  
    return 0;  
}
```

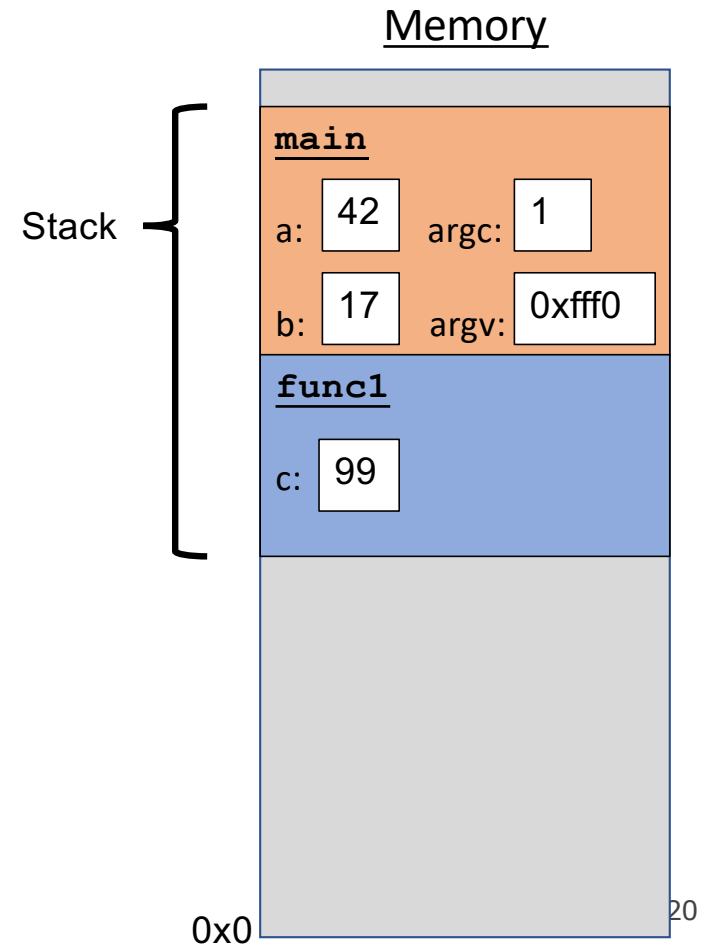


# The Stack

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

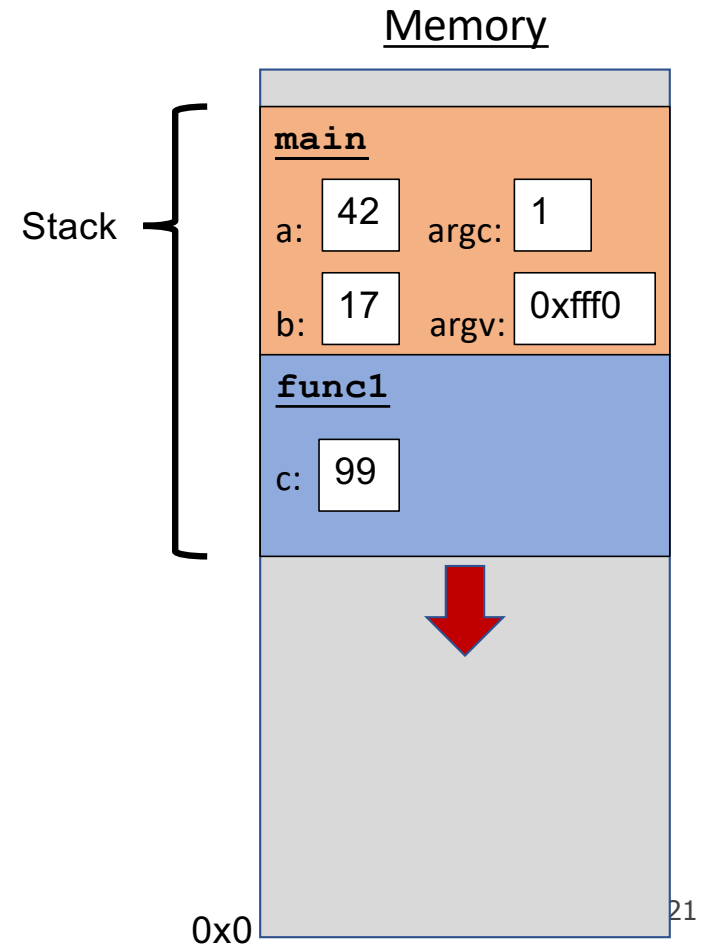
void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    printf("Done.");
    return 0;
}
```



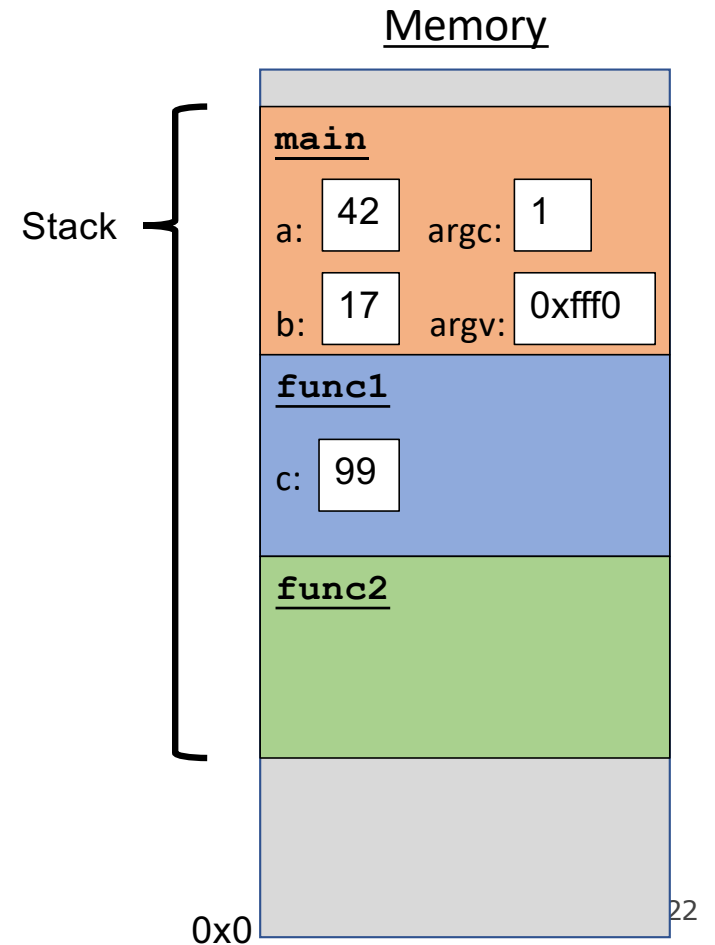
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



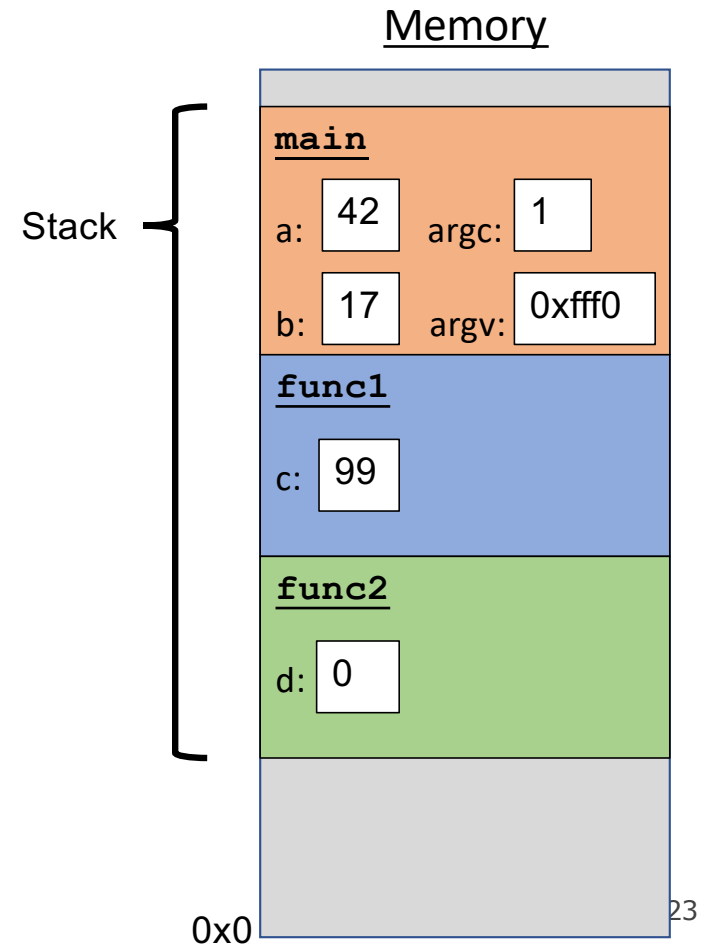
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



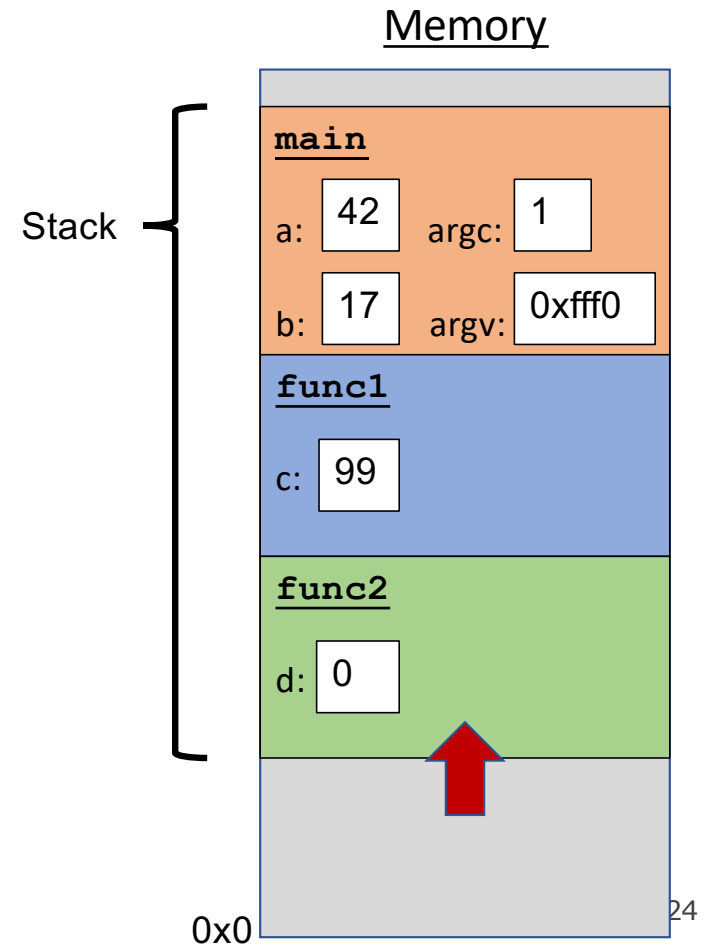
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



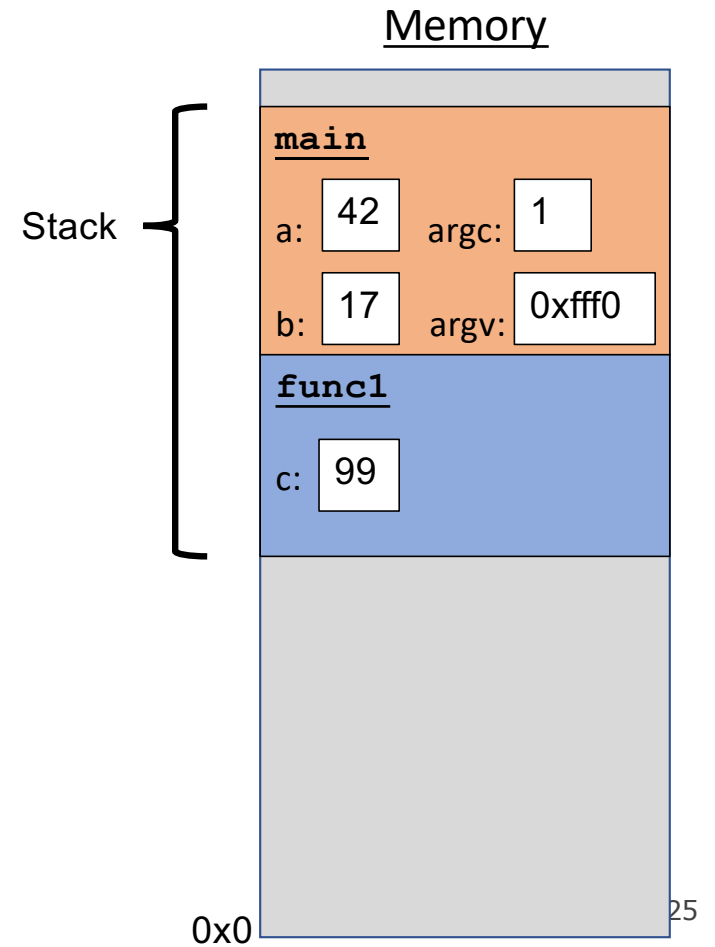
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



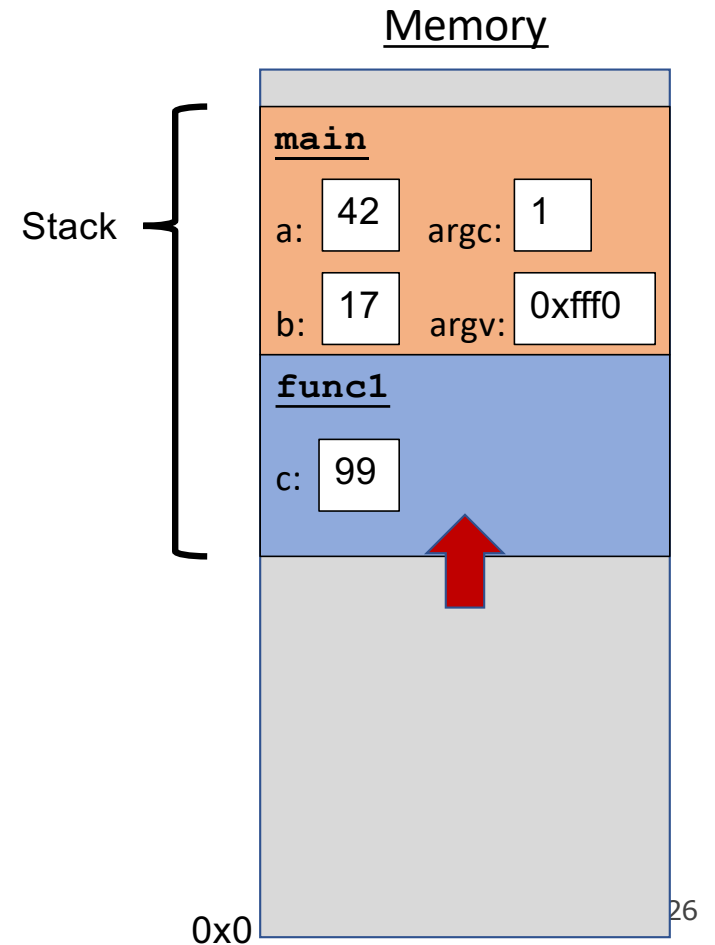
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



# 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();  
    func2();  
    printf("Done.");  
    return 0;  
}
```

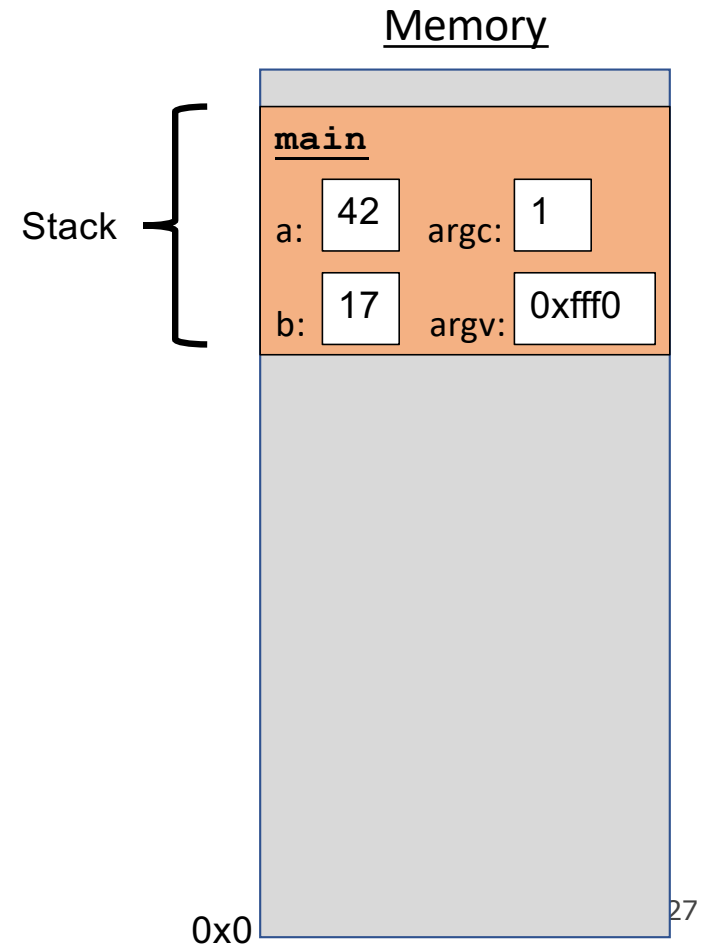


# The Stack

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

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

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    printf("Done.");
    return 0;
}
```

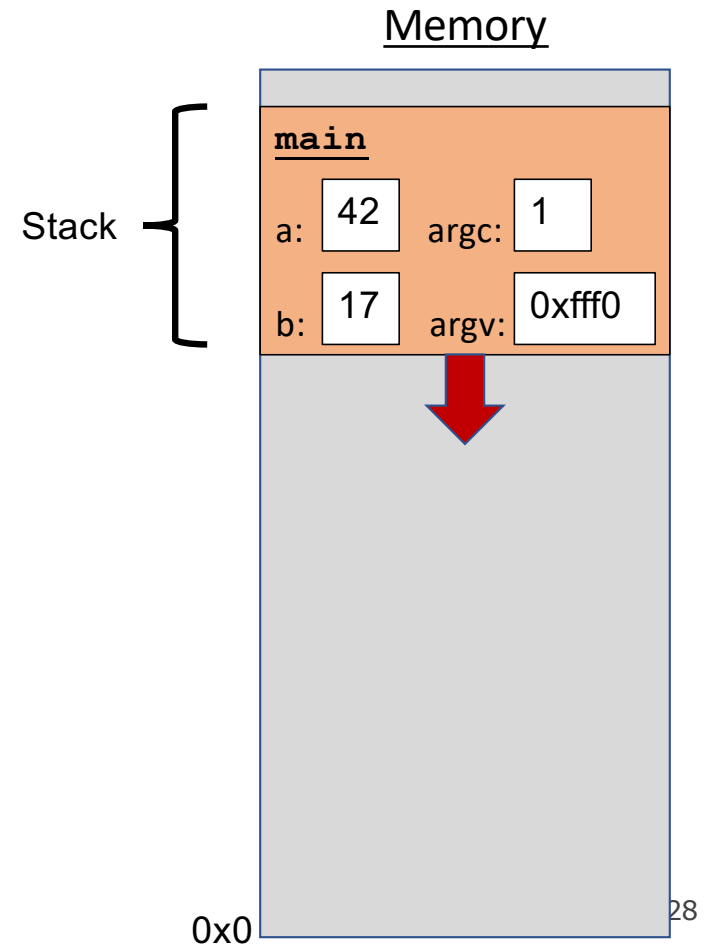


# The Stack

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

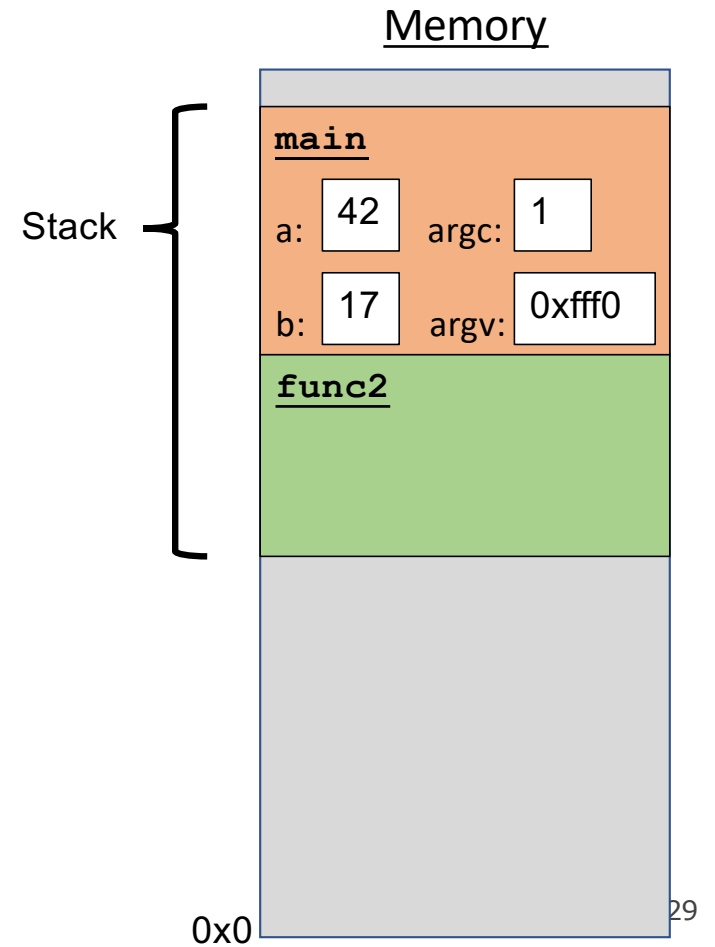
void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    printf("Done.");
    return 0;
}
```



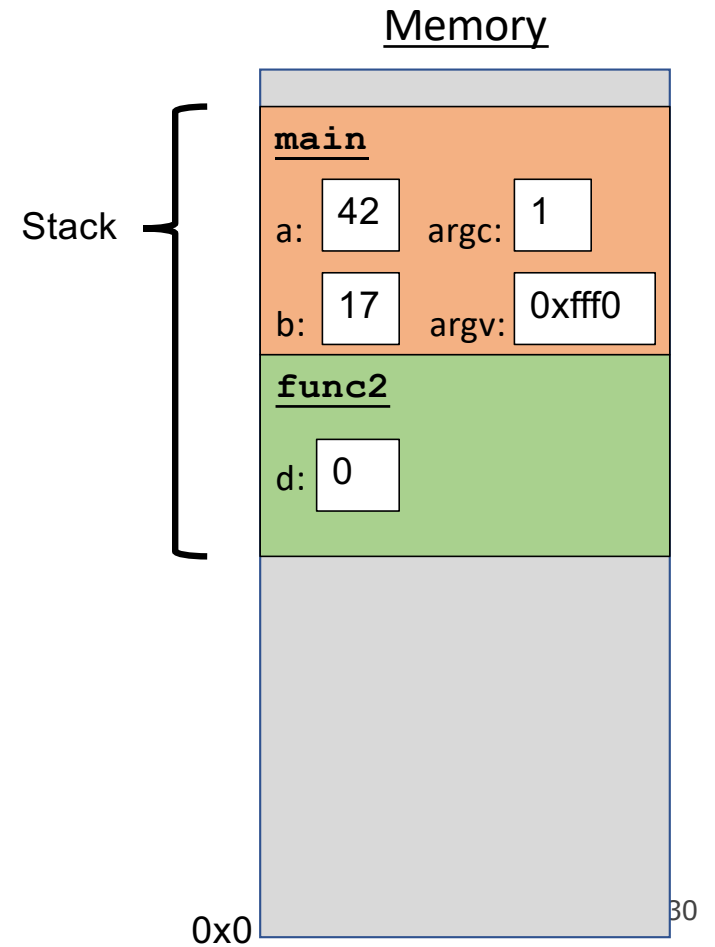
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



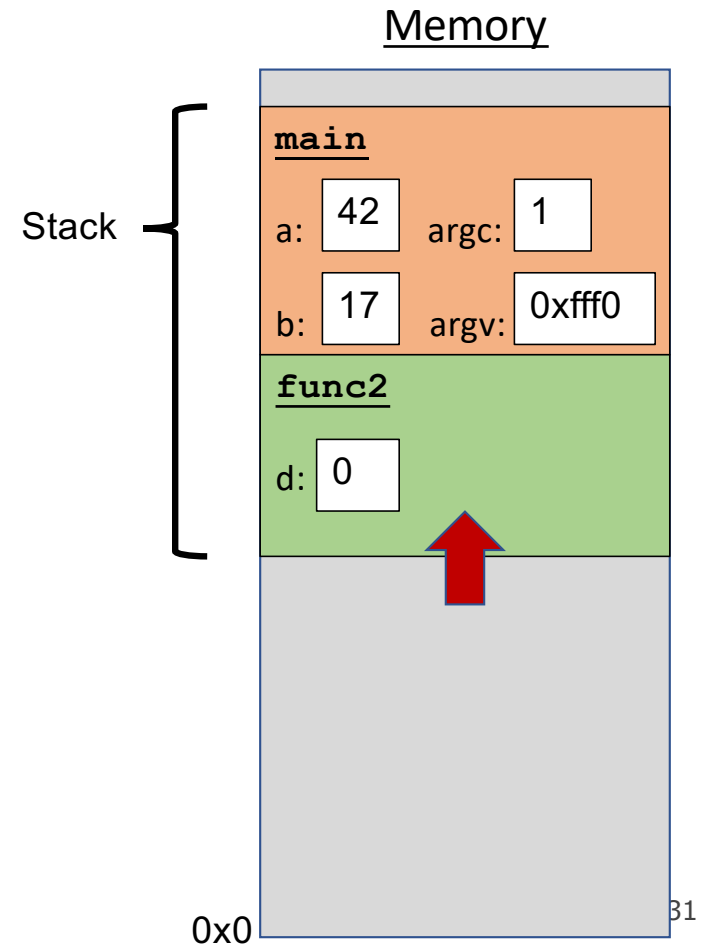
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



# 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();  
    func2();  
    printf("Done.");  
    return 0;  
}
```

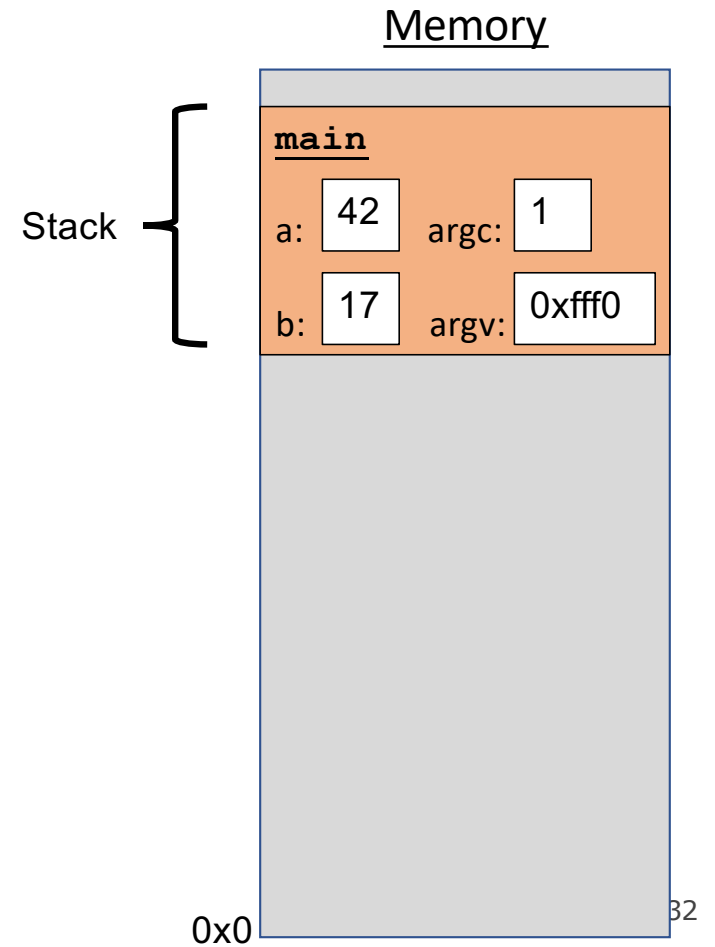


# The Stack

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

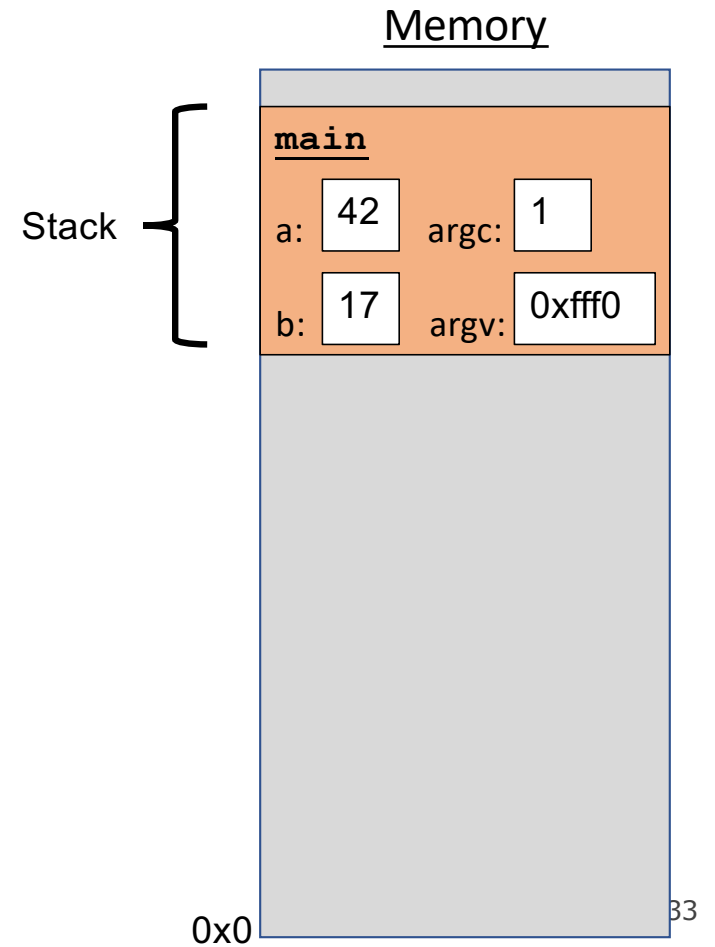
void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    func2();
    printf("Done.");
    return 0;
}
```



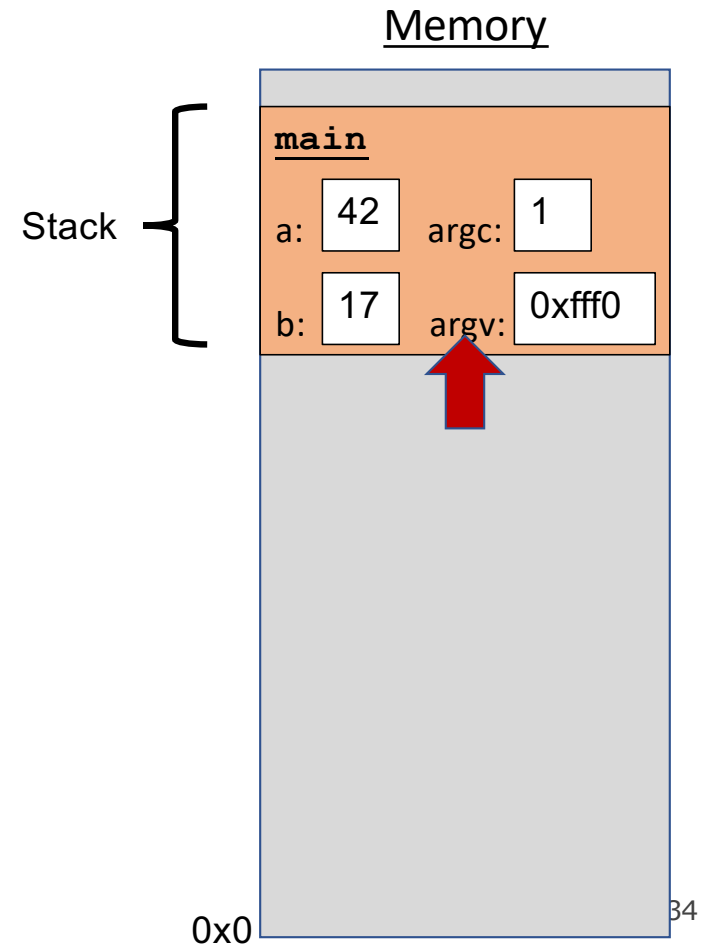
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



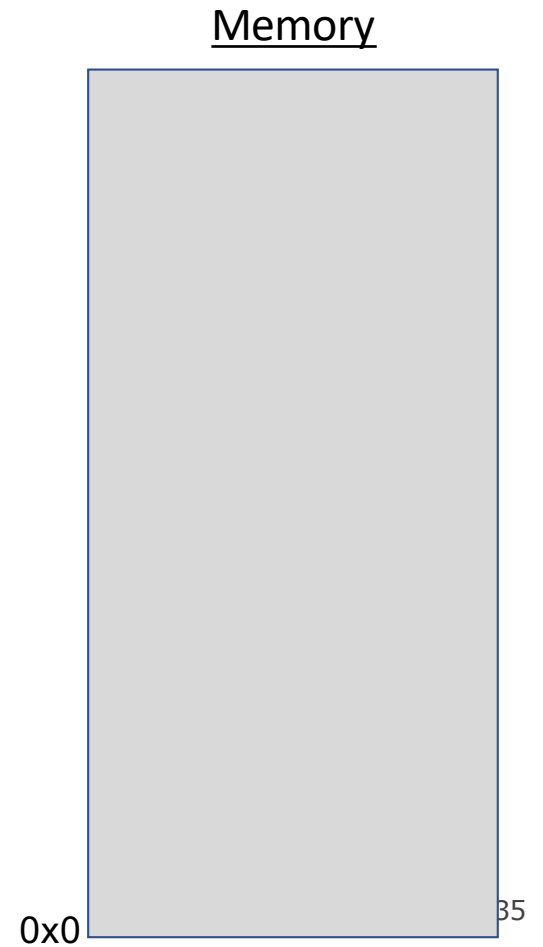
# The Stack

```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



# The Stack

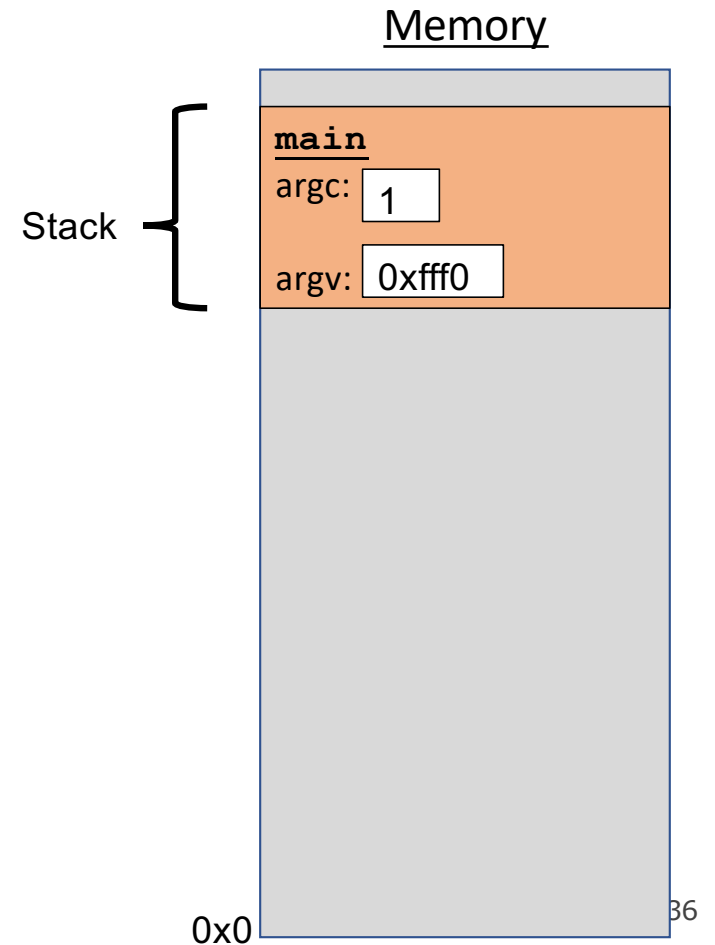
```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    func2();  
    printf("Done.");  
    return 0;  
}
```



# The Stack Failing Us

```
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;  
}
```

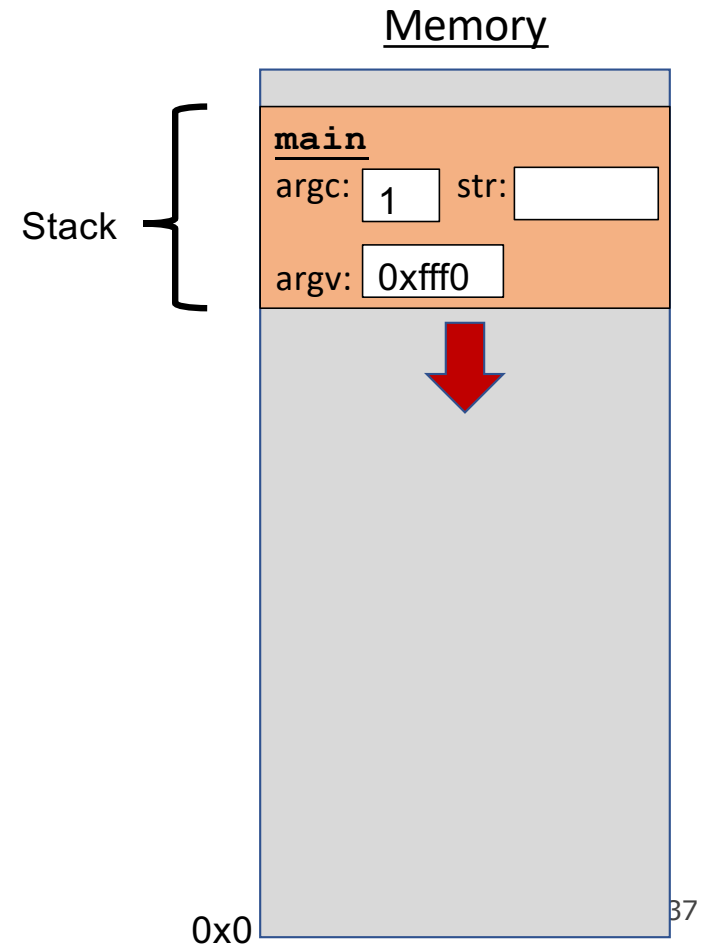
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# The Stack Failing Us

```
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;  
}
```

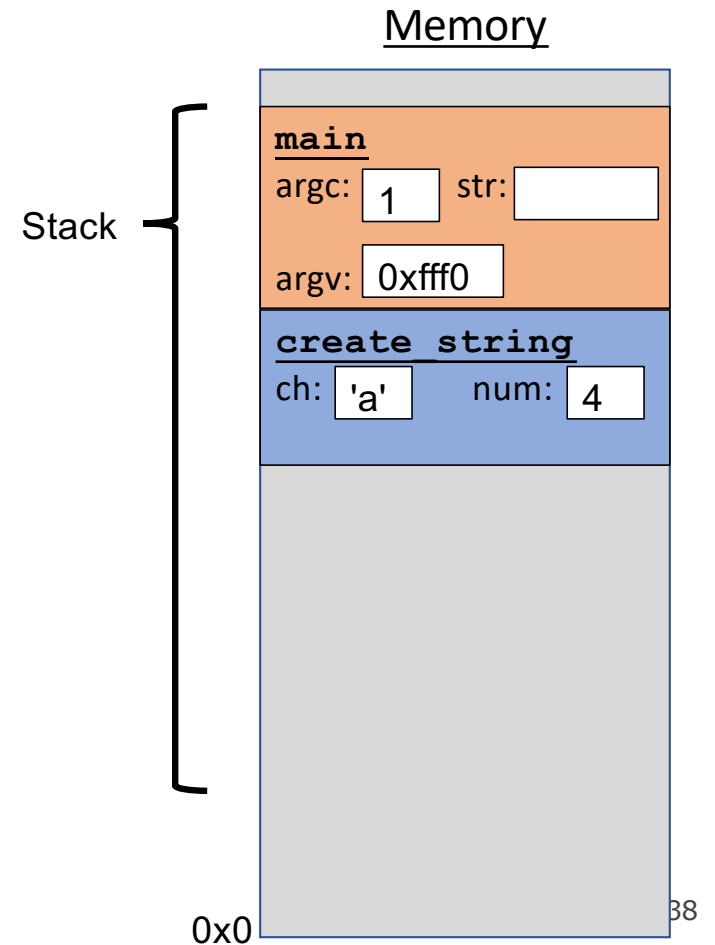
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# The Stack Failing Us

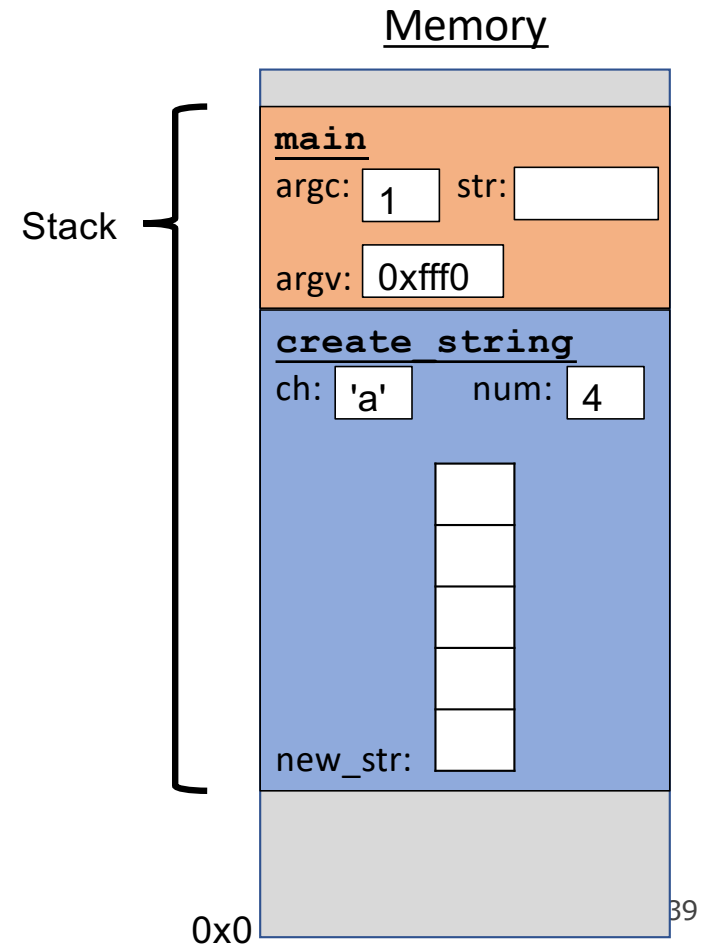
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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



# The Stack Failing Us

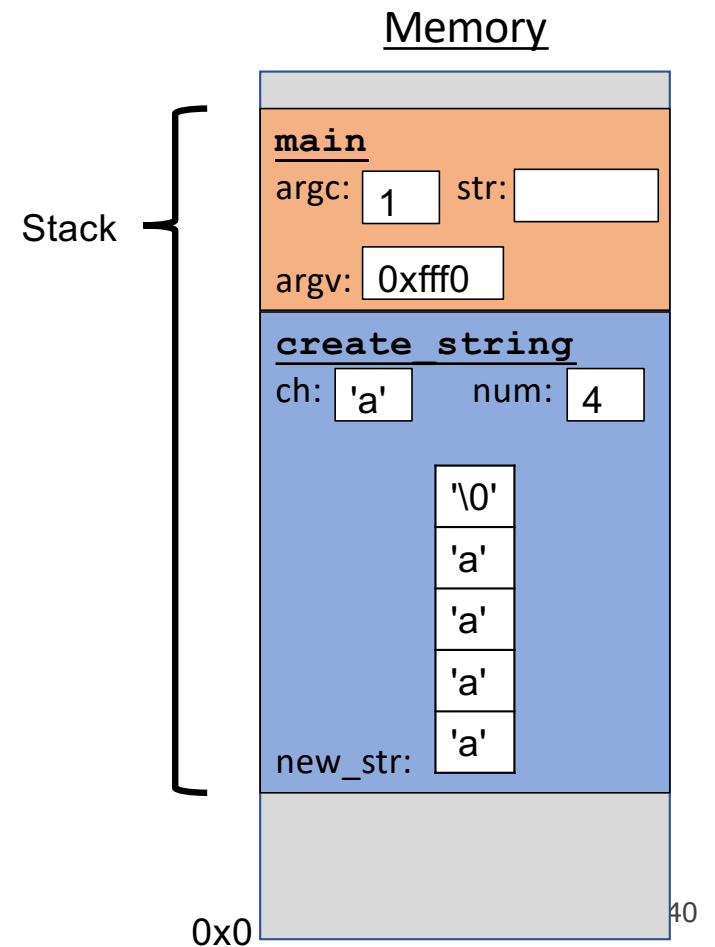
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# The Stack Failing Us

```
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;  
}
```

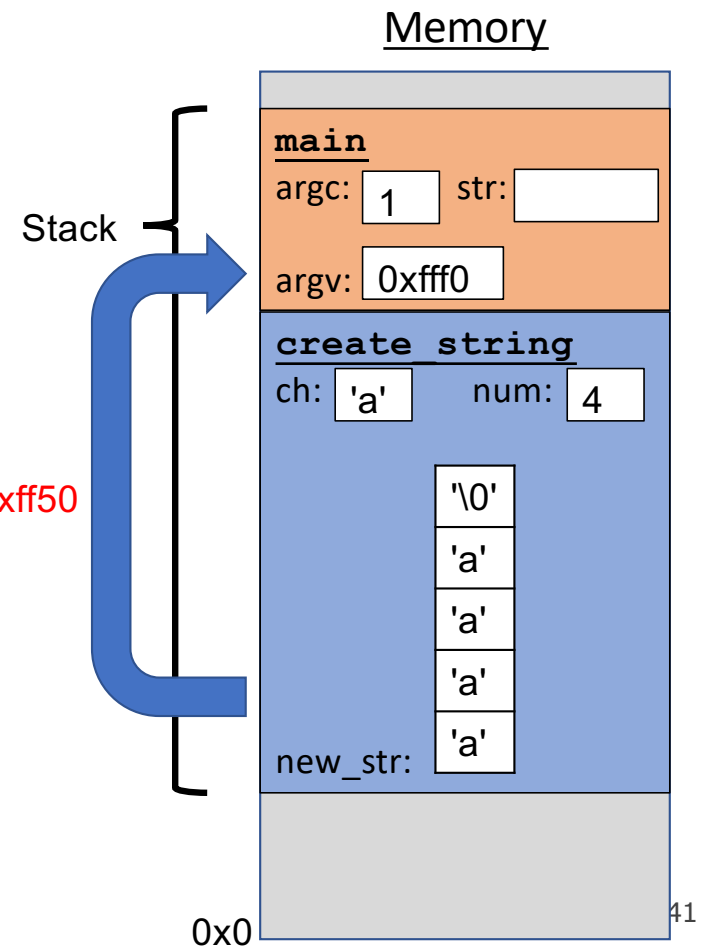


# The Stack Failing Us

```
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;  
}
```

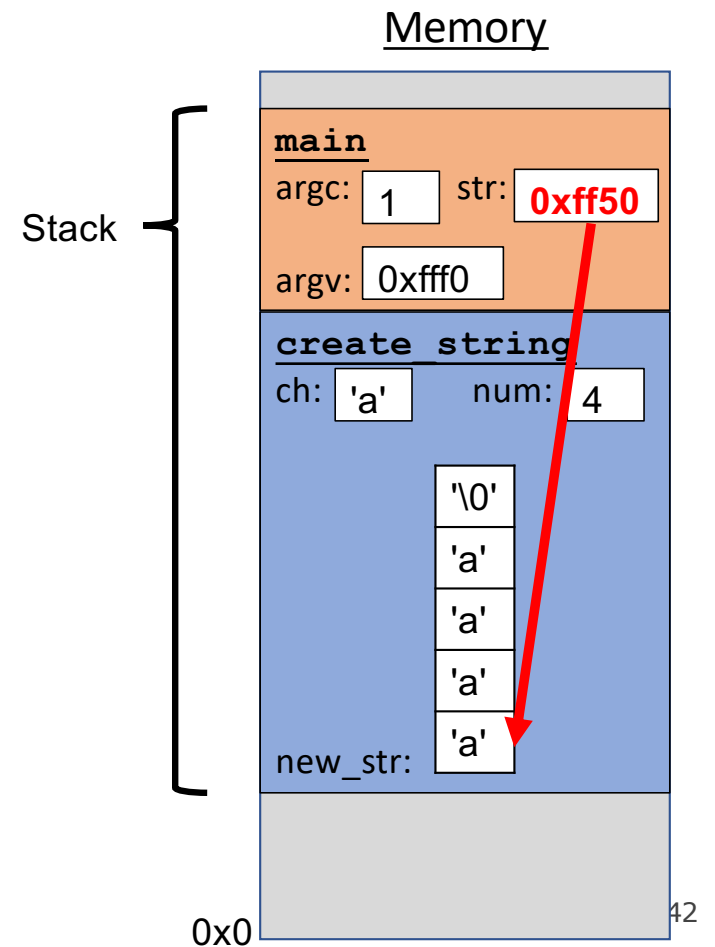
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```

Returns e.g. 0xff50



# The Stack Failing Us

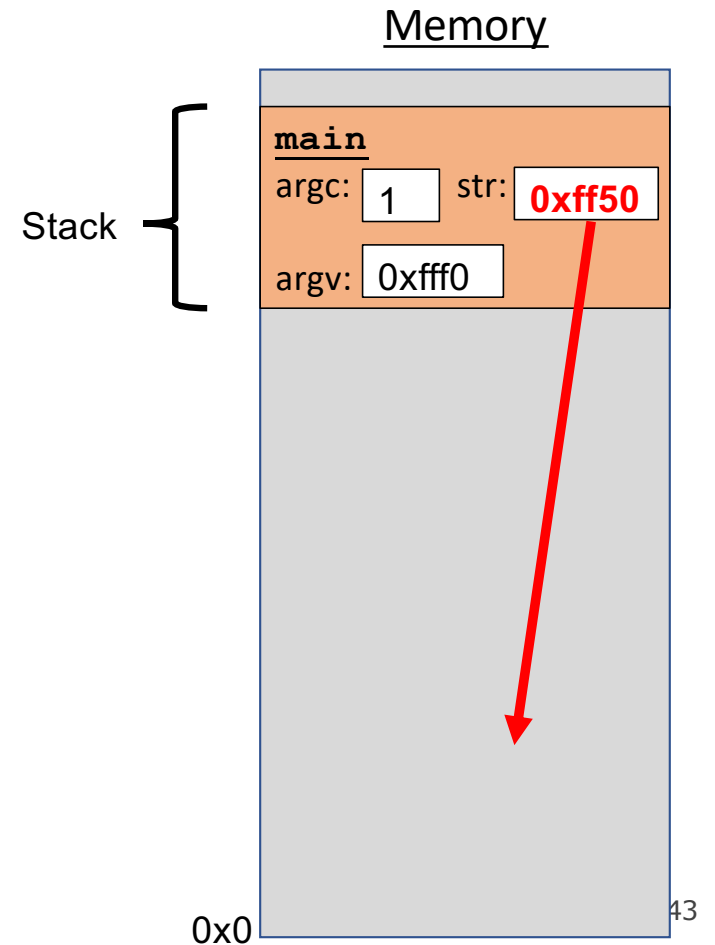
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# The Stack Failing Us

```
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;  
}
```

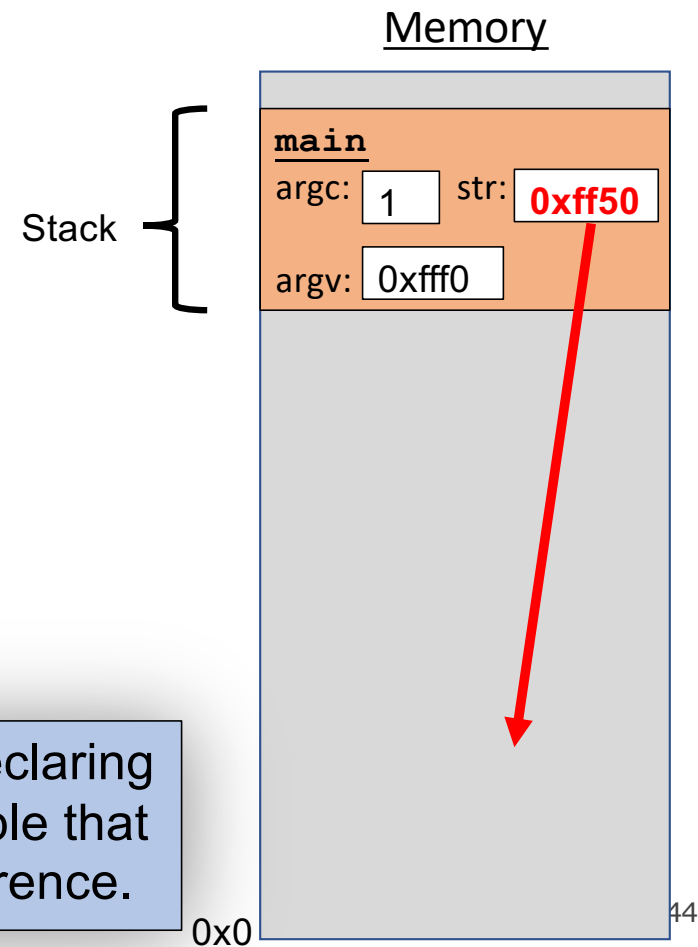


# The Stack Failing Us

```
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;  
}
```

**Ack!** Local variables go away when the function declaring them exits. These 'a's were embedded in a variable that no longer exists, so **str** is not our address to deference.

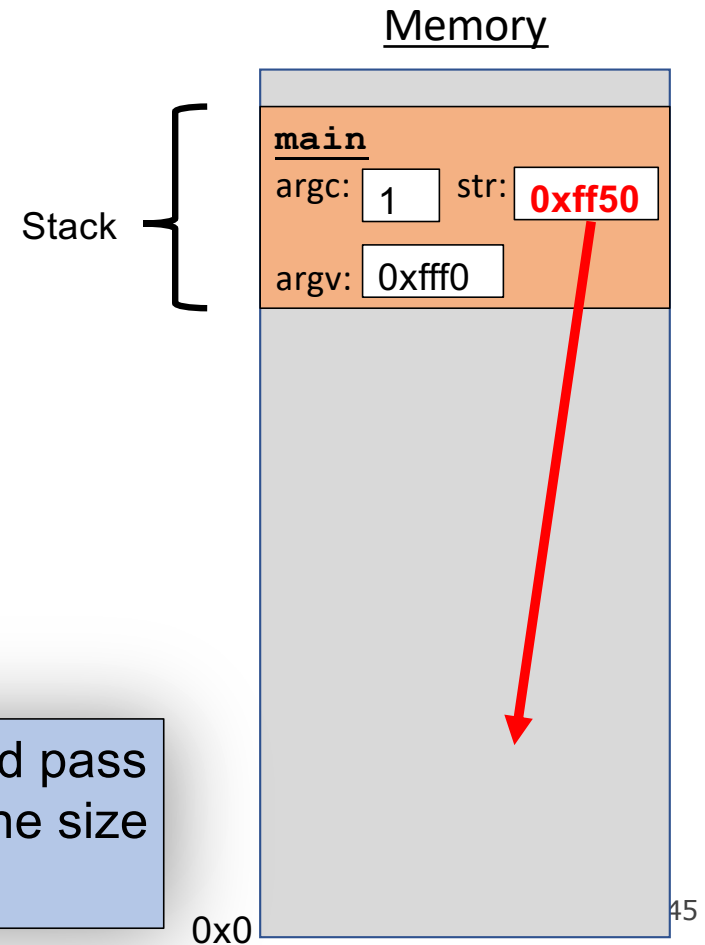


# The Stack Failing Us

```
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.



# The Stack Failing Us

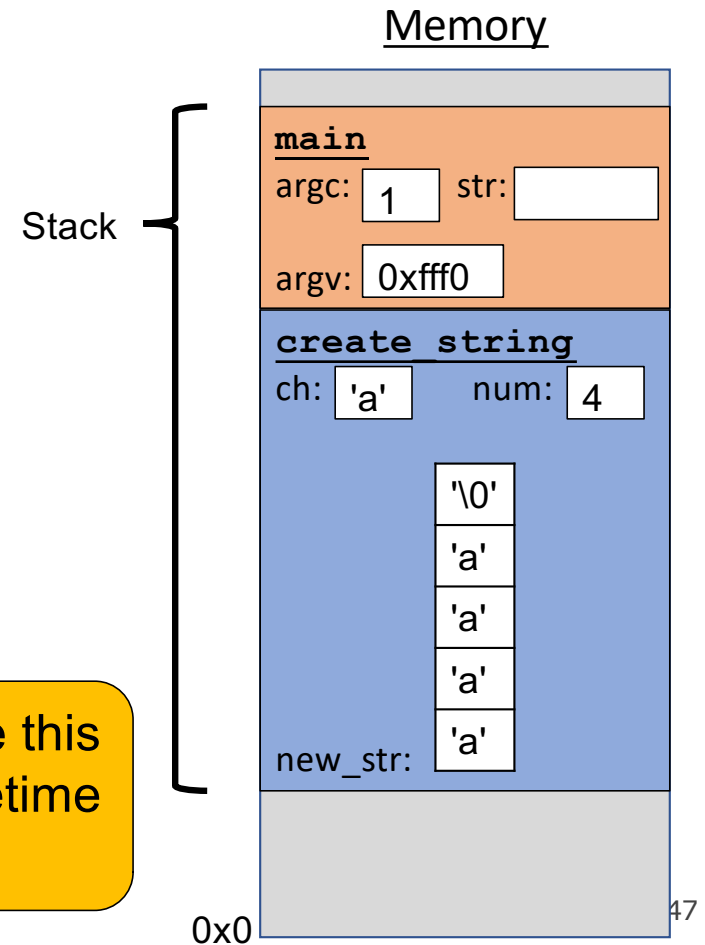
This is a problem! We need a way to allocate memory that persists even after the allocating function exits.

# The Heap

```
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 that allocates it?

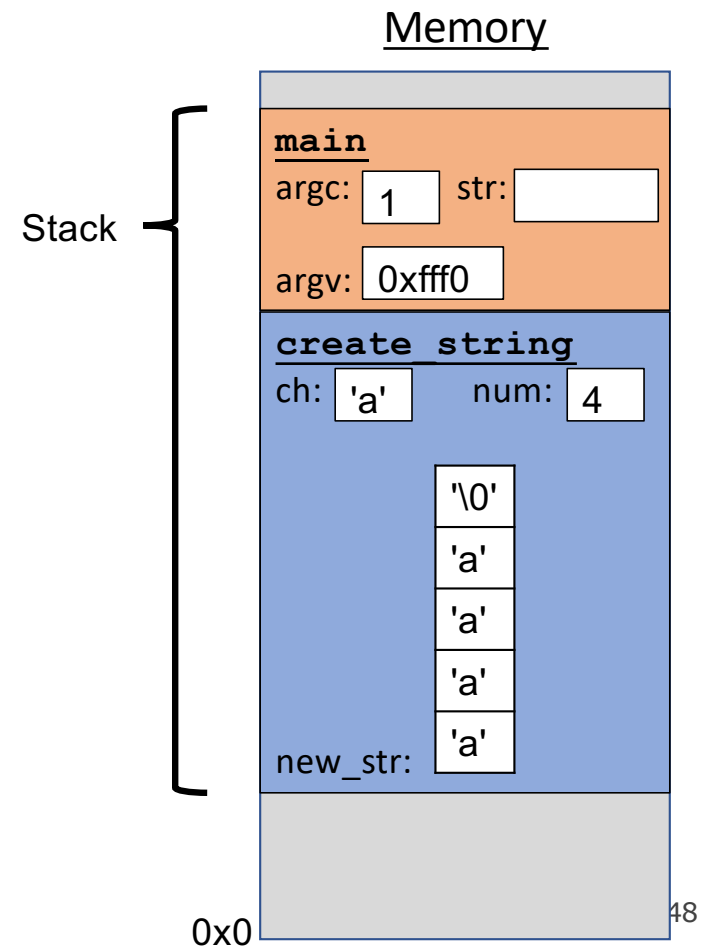


# The Heap

```
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;  
}
```

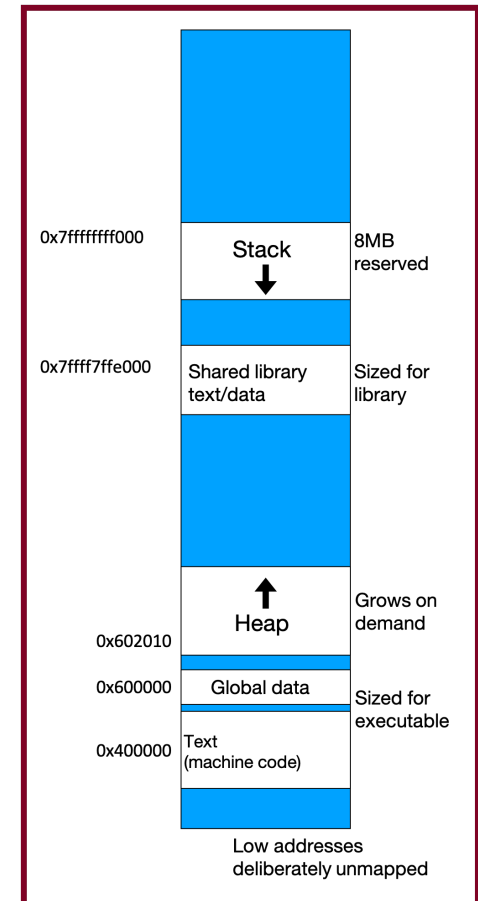
**C:** sure, but since I don't know when to **deallocate** it, it's your responsibility to do that



# The Heap

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



# 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 provides **dynamic memory** that you programmatically introduce—sometimes incorrectly—to the program. That means you may encounter **runtime errors**, even if your code compiles! It's your responsibility to allocate properly and debug when there are problems.

# malloc

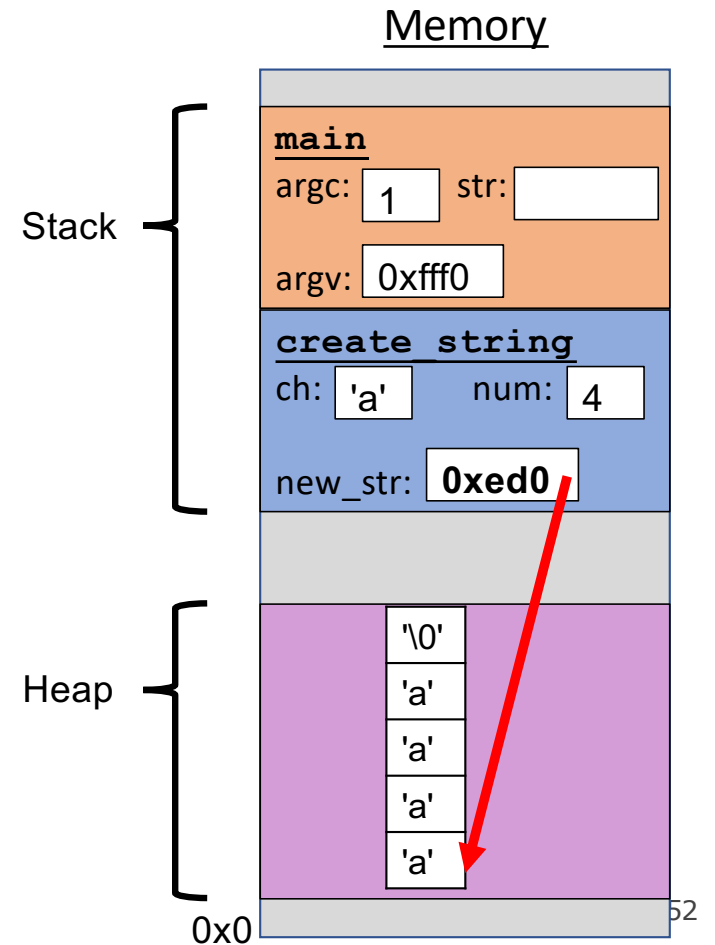
```
void *malloc(size_t size);
```

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

- This function returns a pointer to *the leading address of the new memory block*. It doesn't know or care if it's to be used for an array, 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.

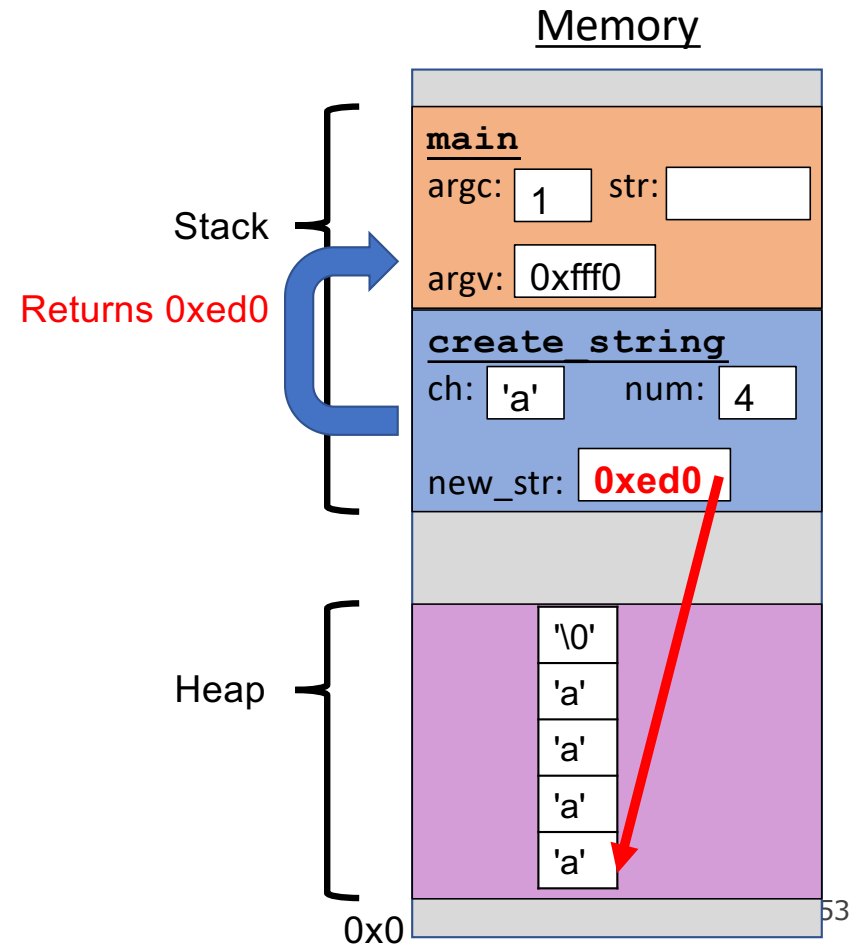
# The Heap

```
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;  
}
```



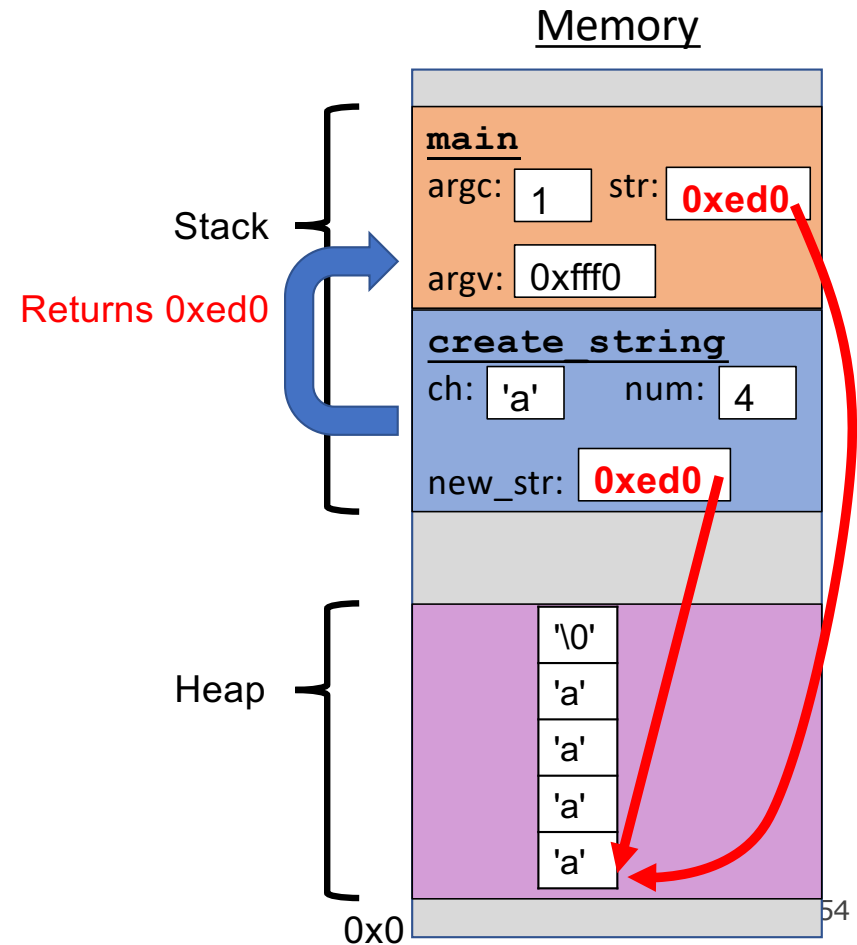
# The Heap

```
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;  
}
```



# The Heap

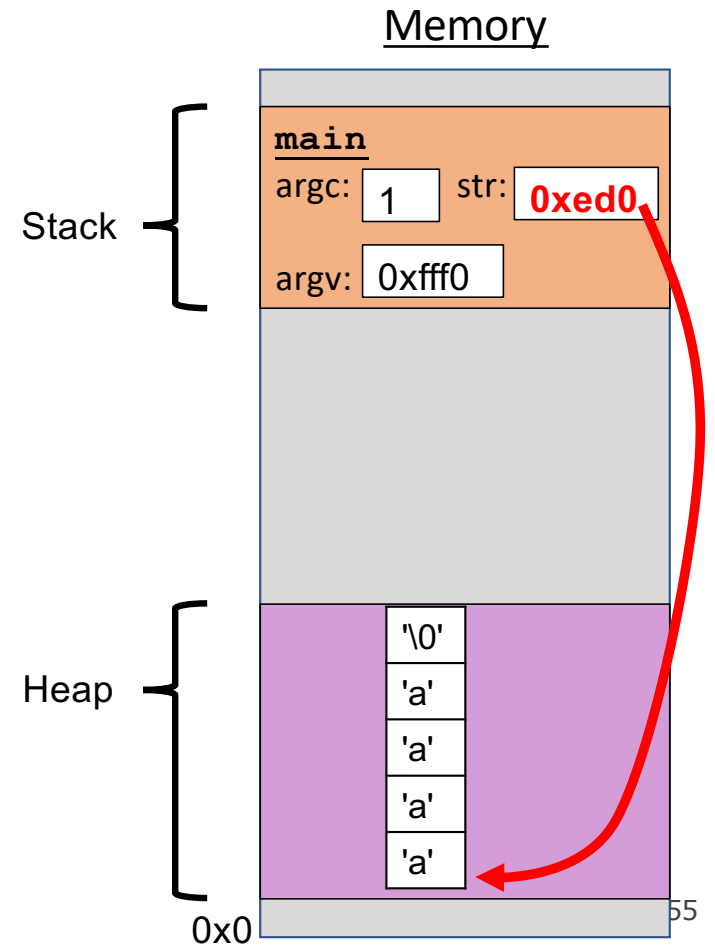
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(num + 1);  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# 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;  
}
```

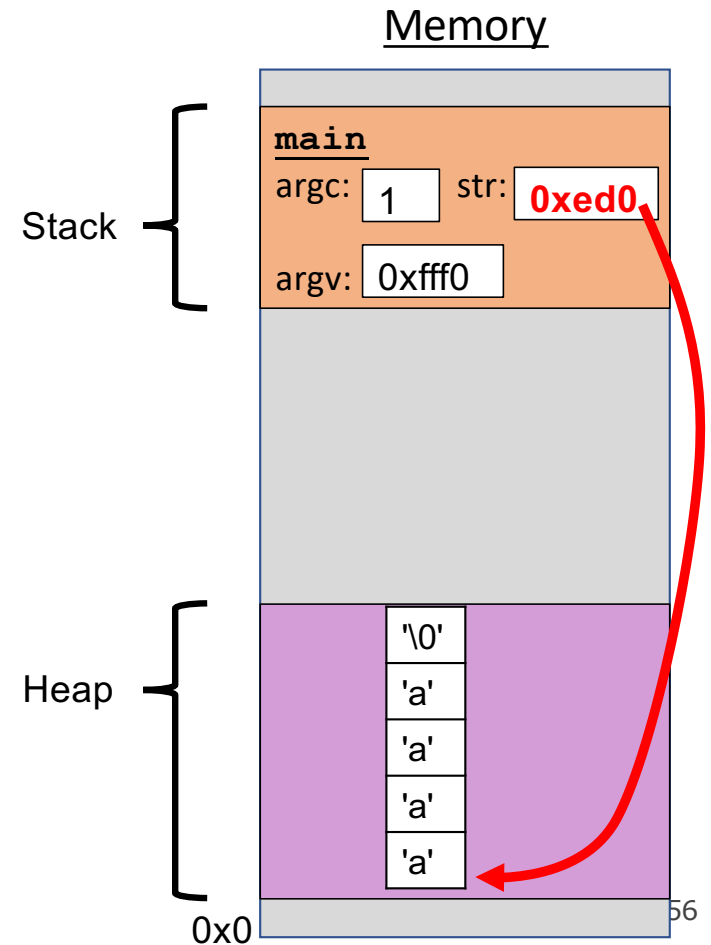
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# 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;  
}
```

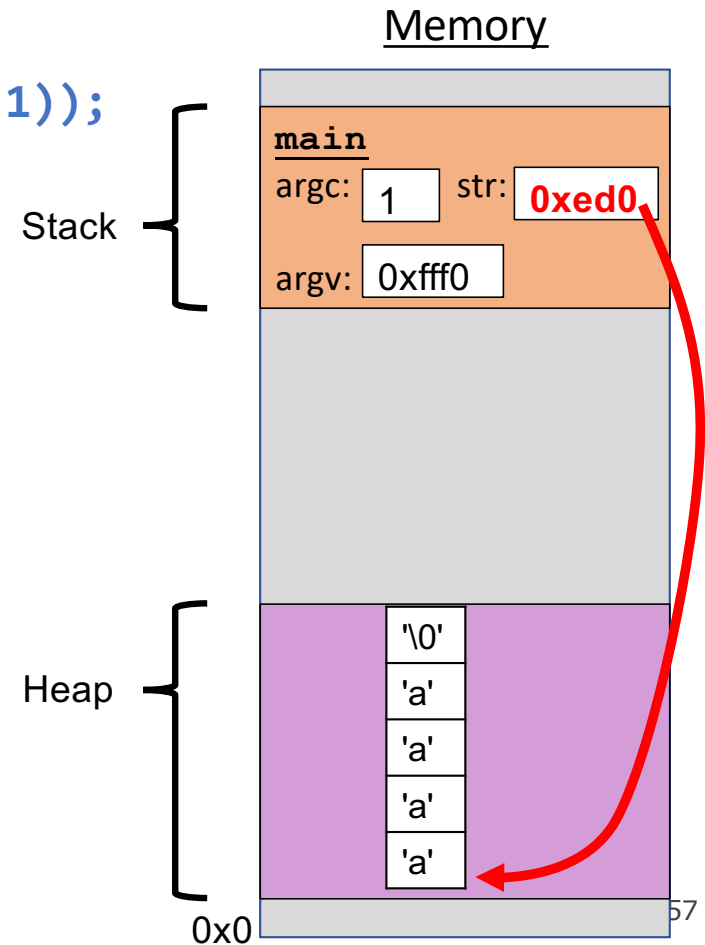
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# The Heap

```
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  
}
```



# Exercise: malloc multiples

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

```
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**.

```
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 }
```

- 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.**

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));`

# Always assert with the heap

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

```
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 intentionally end the program if the provided condition is false. A memory allocation error is significant, and we should terminate the program when we see them.

# 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 zeroes out all memory. Use only when absolutely 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 requiring you to **malloc** and copy in the string yourself.

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

You could imagine **strdup** might be implemented in terms of **malloc** + **strcpy**. (In fact, it pretty much 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 deallocate 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 to point to something else.

```
char *str = strdup("hello");
```

```
...
```

```
free(str);
```

```
str = strdup("hi");
```



# free details

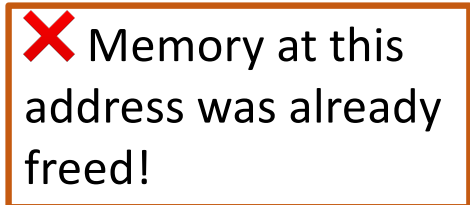
Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

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

```
...  
free(bytes);
```



```
...  
free(ptr);
```



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);
```



# Memory Leaks

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

```
char *str = strdup("hello");
```

```
...
```

```
str = strdup("hi"); // memory leak!  Lost previous str
```

# Memory Leaks

- A memory leak occurs when you dynamically allocate a block of memory on the heap but fail to 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 heap memory! (Running out of memory is rare, but it can happen if the program is designed to run for a very, long time—e.g., a web server.)
- However, memory leaks rarely 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 so they can be plugged.

# 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

```
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);
```

# realloc

- **realloc** only accepts pointers that were previously returned by malloc/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

# 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`.

```
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);
```

# 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)

- **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)

# 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