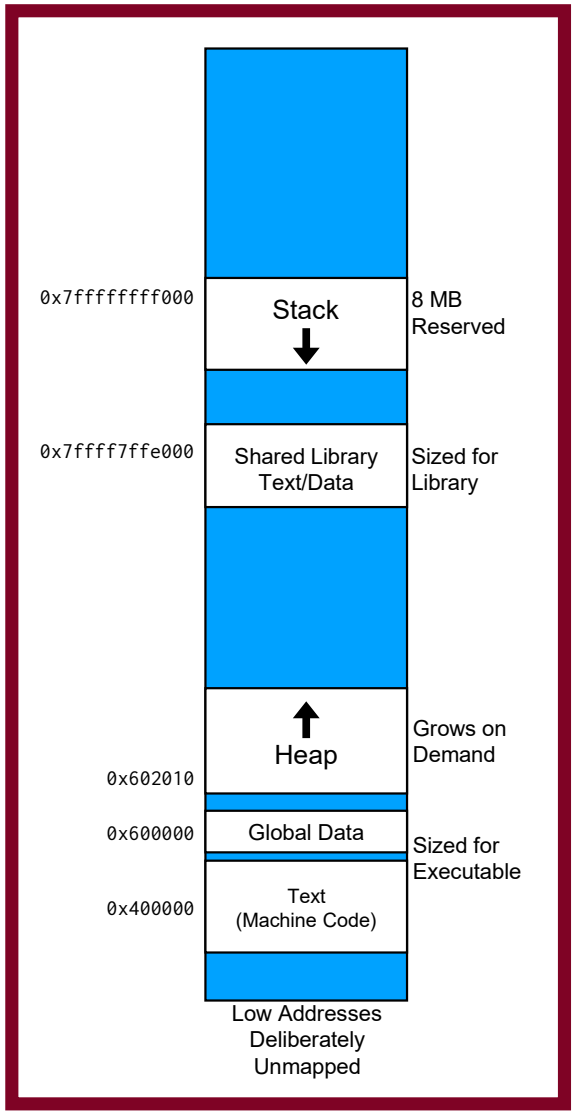
CS 107
Lecture 5: Stack and Heap

Friday, January 27, 2023

Computer Systems
Summer 2023
Stanford University
Computer Science Department

Reading: Reader: Ch 4, *C Primer,* K&R Ch 1.6, 5.1-5.5



Lecture Plan

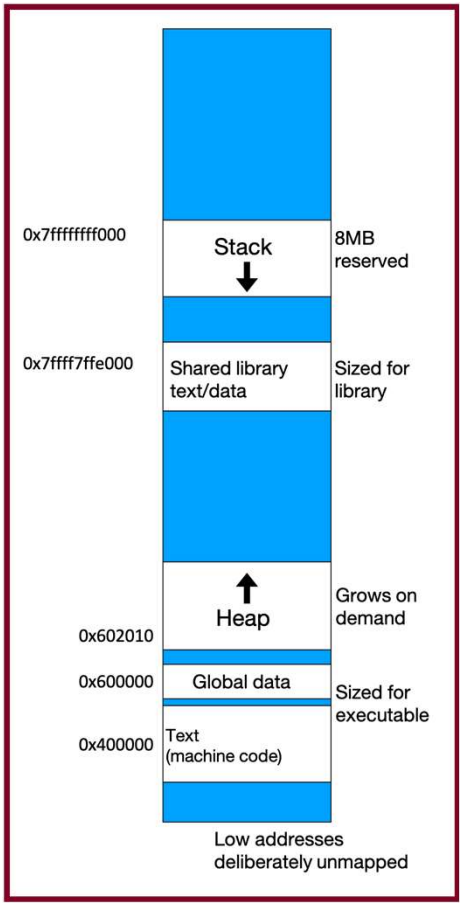
- The Stack 3
- The Heap and Dynamic Memory 59
- `realloc` 83
- Use After Free 99

Memory Layout

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

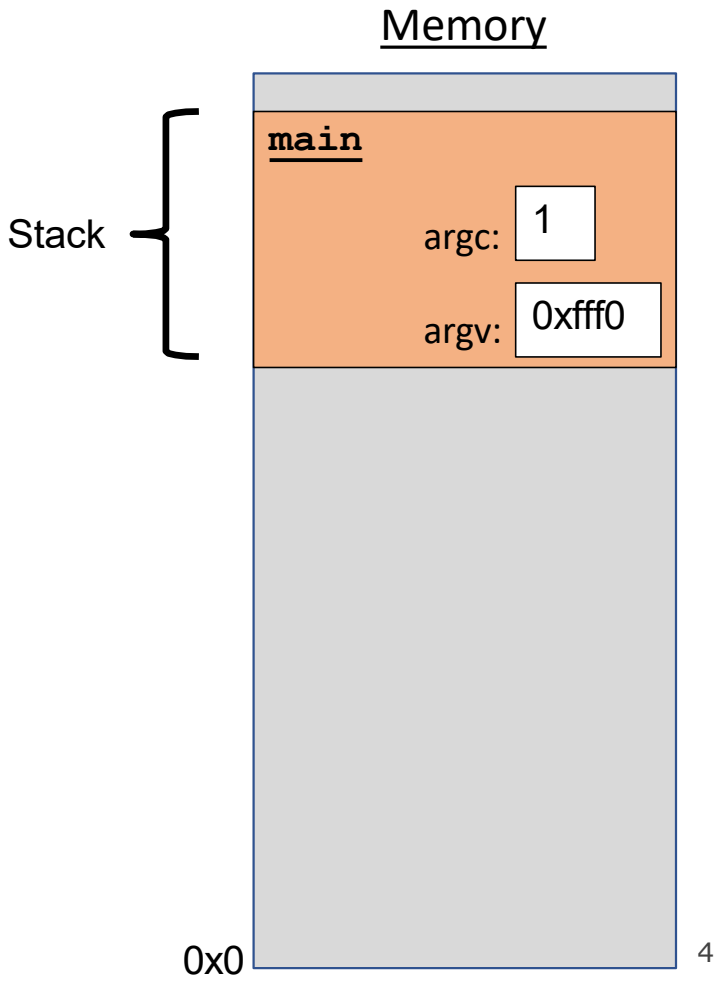
- The **stack** is the place where all local variables and parameters live for each function. A function's stack "frame" goes away when the function returns.

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



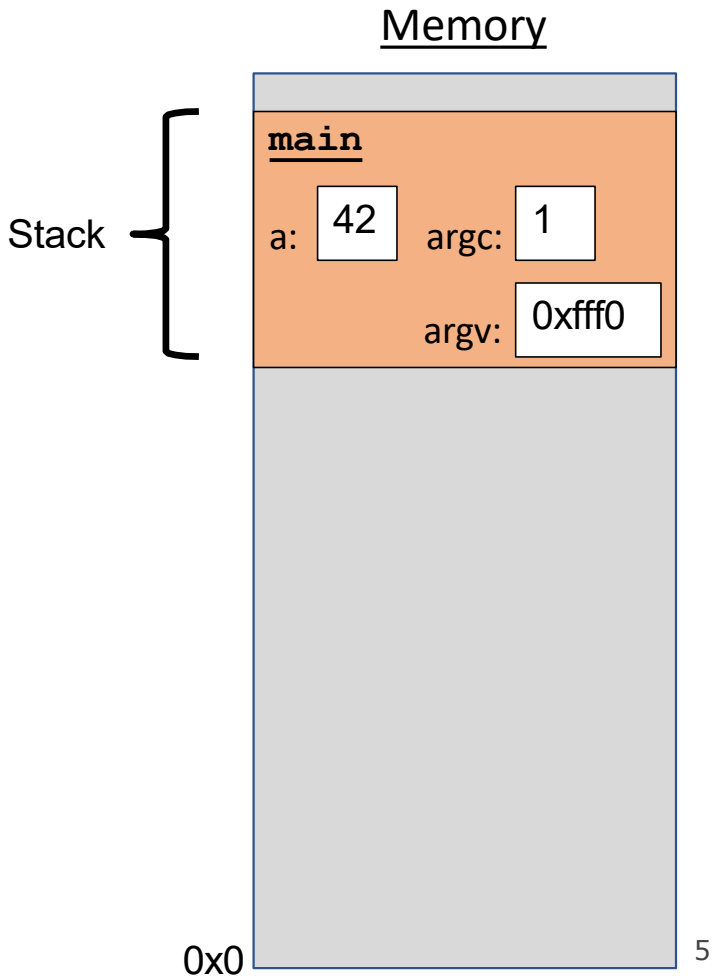
The Stack

```
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;
}
```



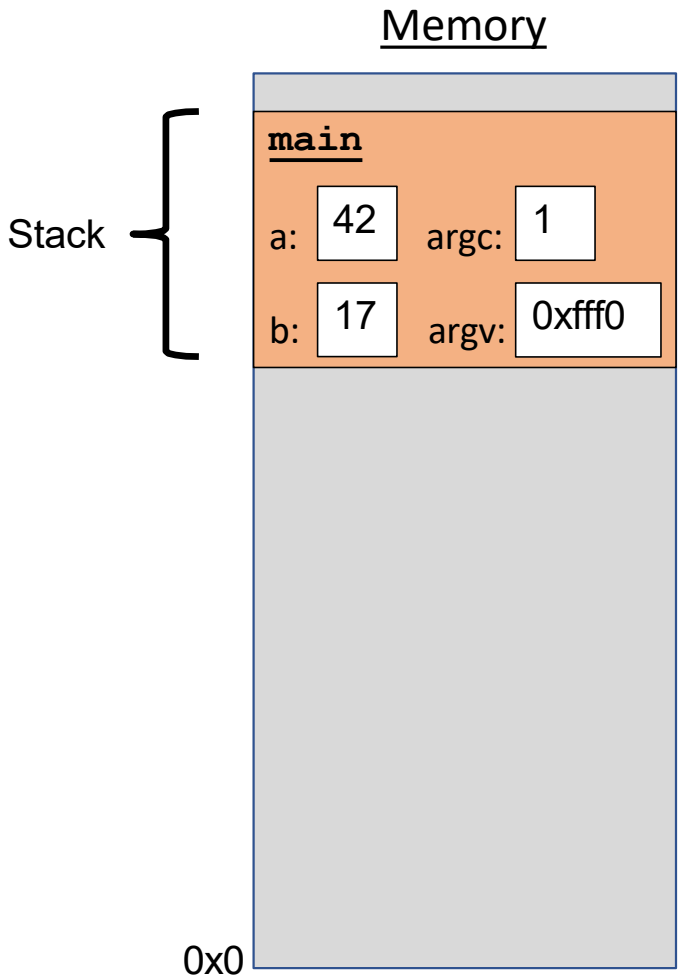
The Stack

```
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;
}
```



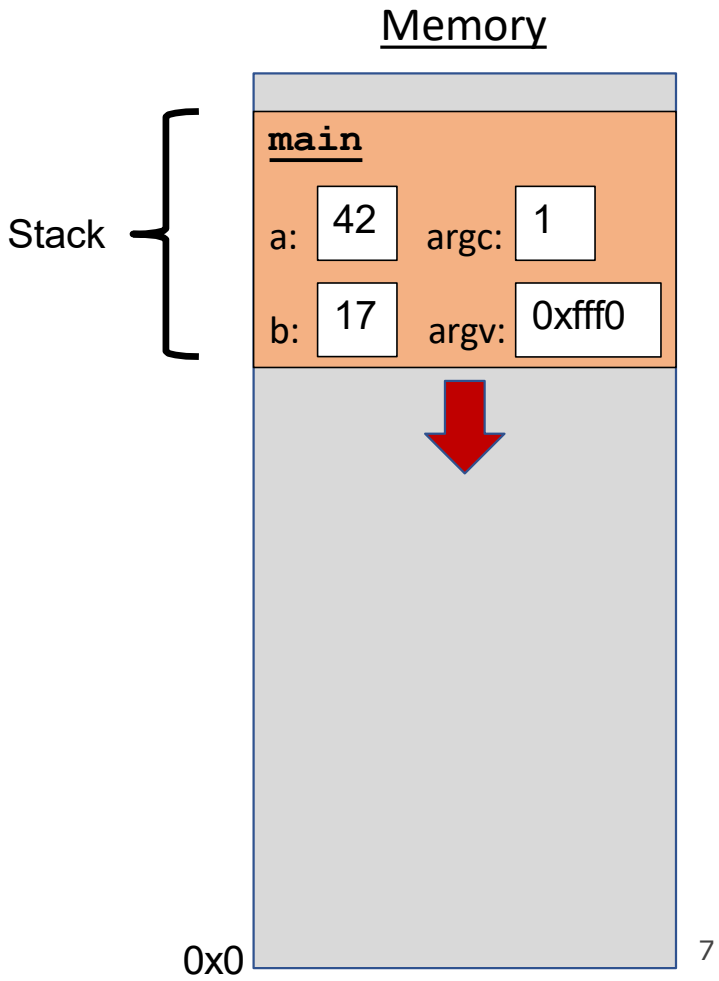
The Stack

```
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;
}
```



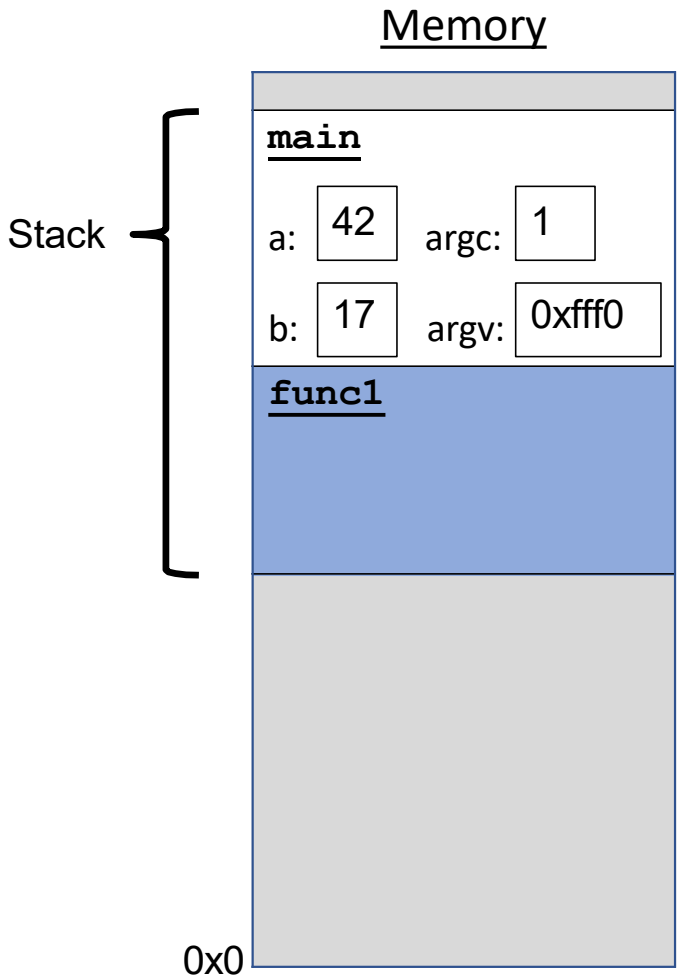
The Stack

```
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;
}
```



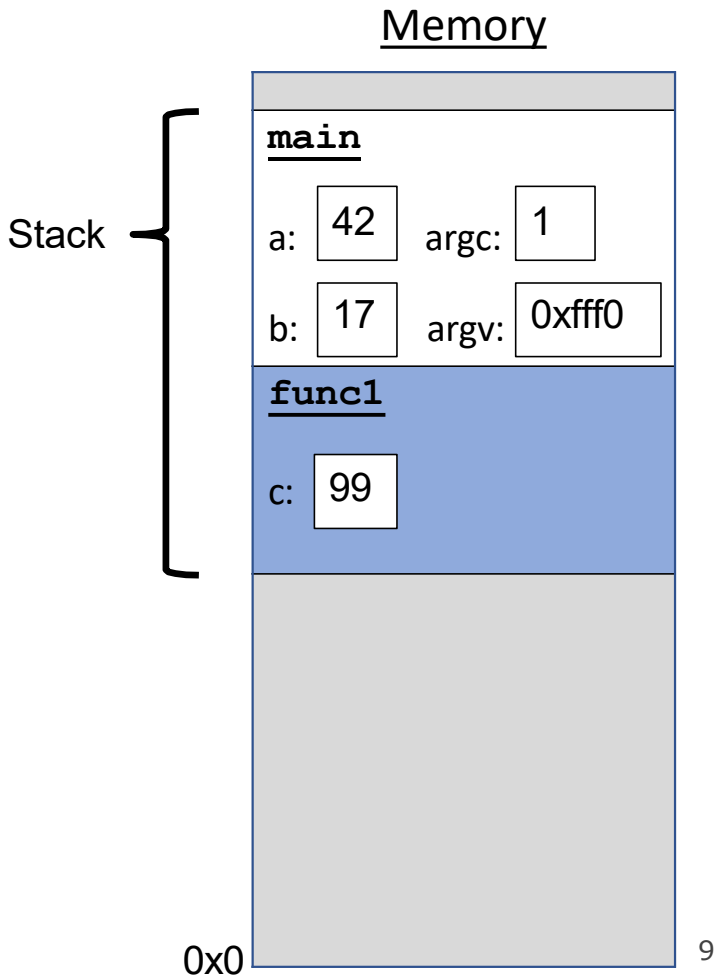
The Stack

```
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;
}
```



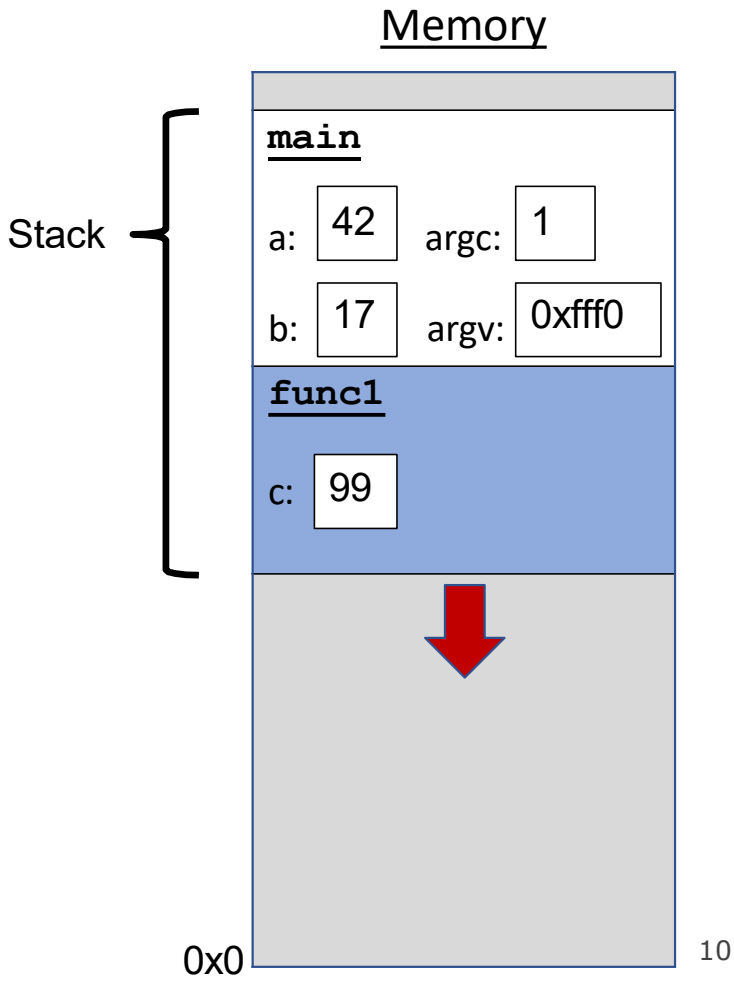
The Stack

```
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;
}
```



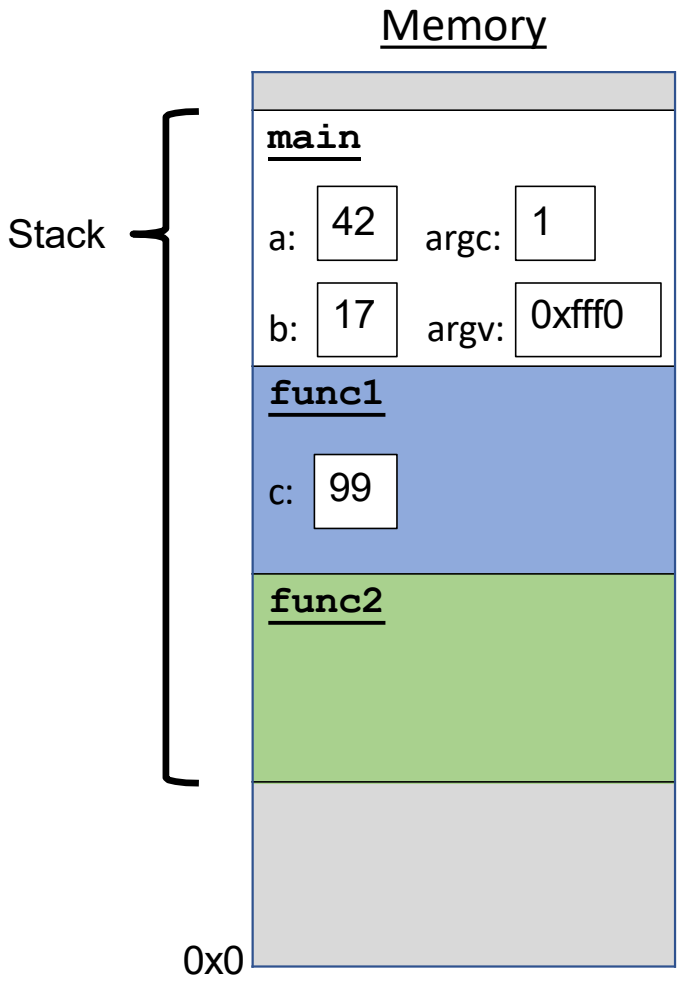
The Stack

```
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;
}
```



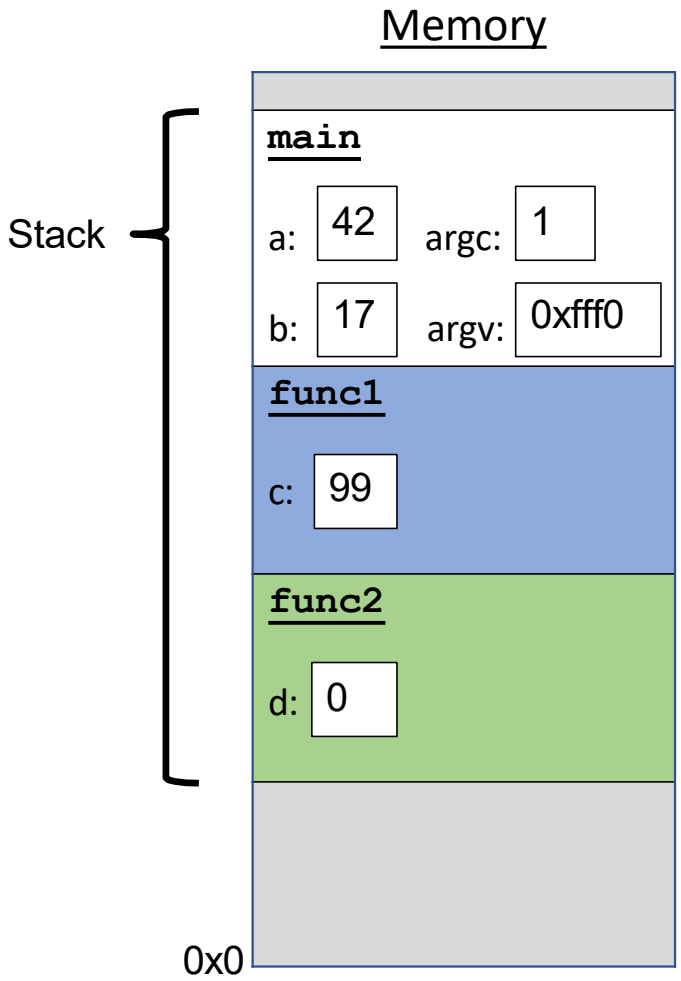
The Stack

```
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;
}
```



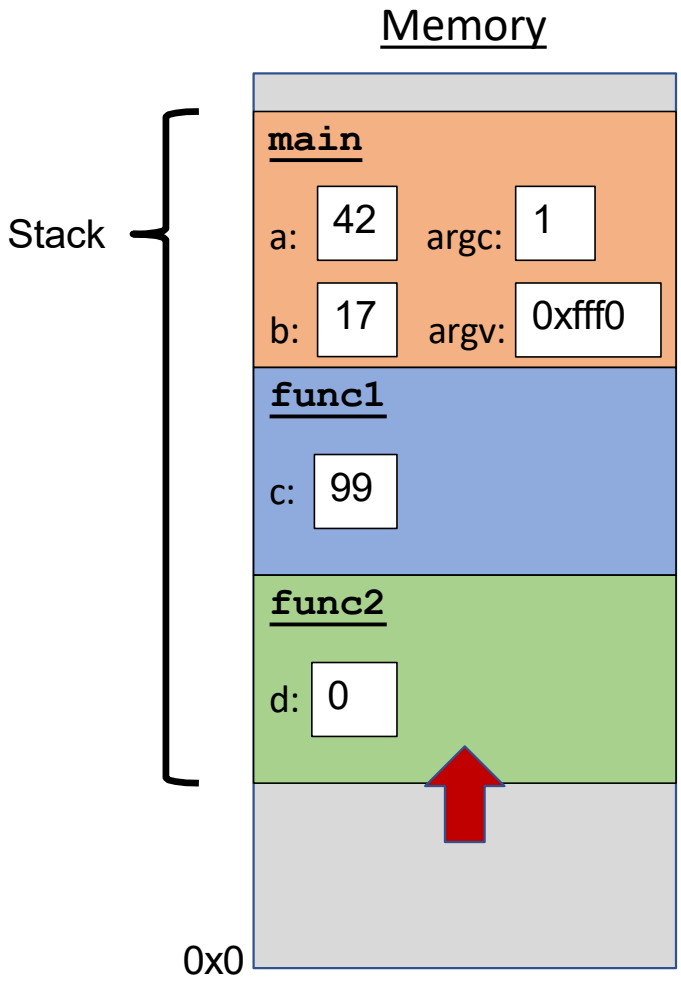
The Stack

```
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;
}
```



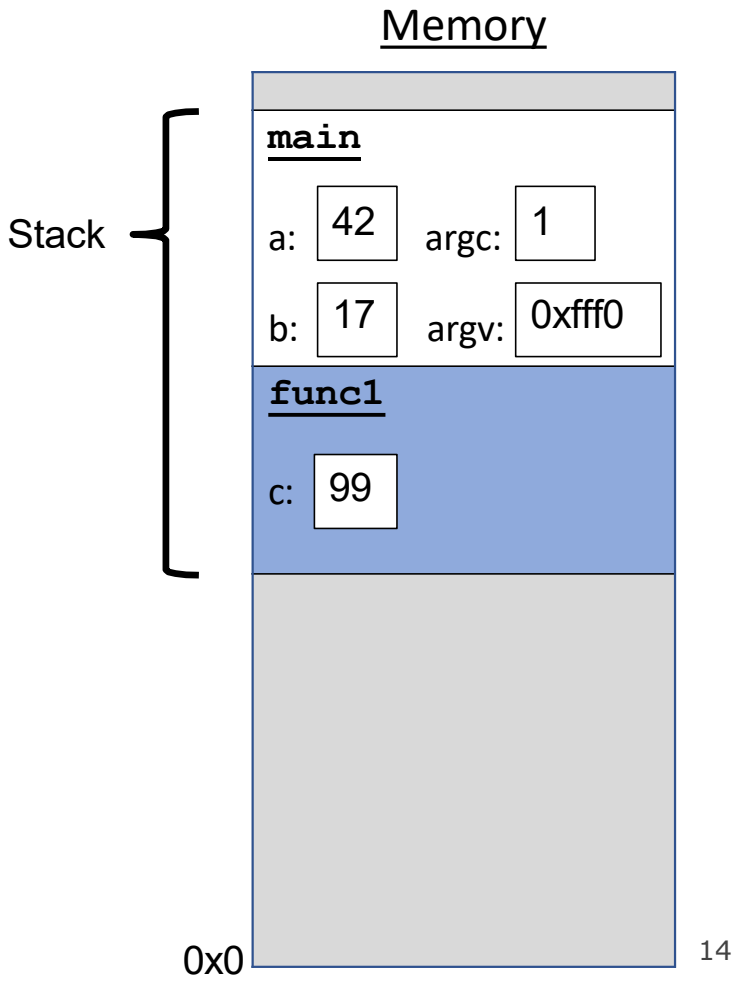
The Stack

```
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;
}
```



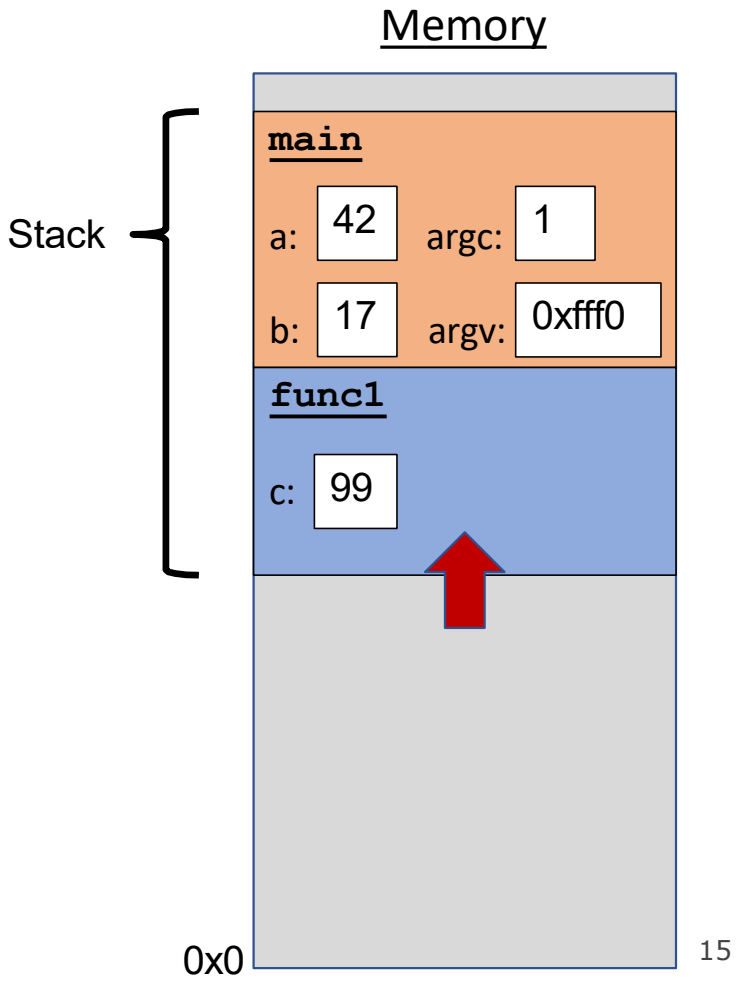
The Stack

```
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;
}
```



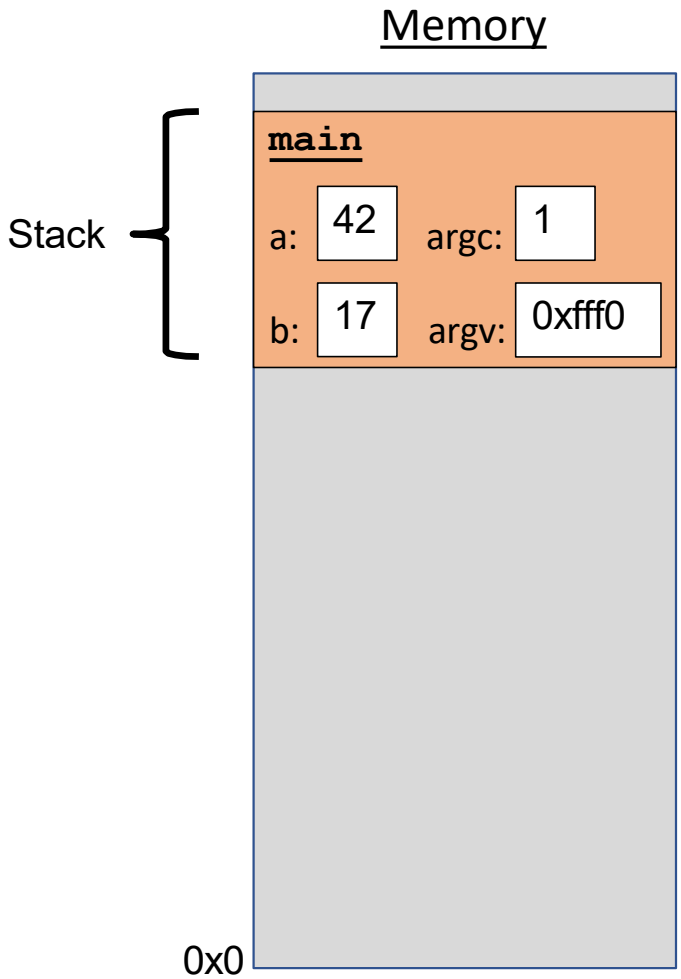
The Stack

```
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;
}
```



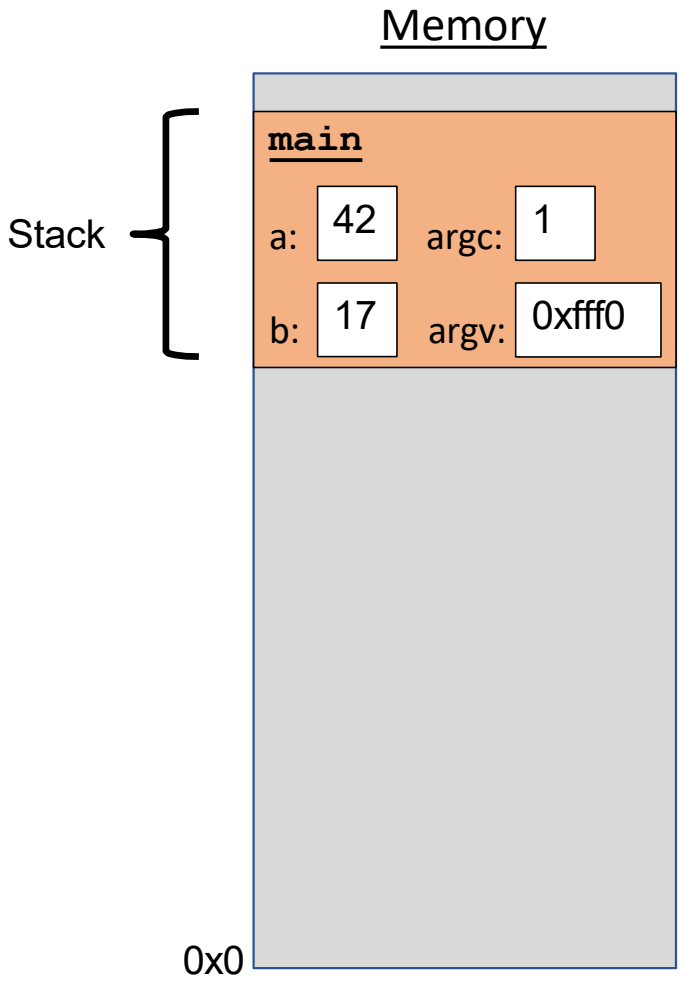
The Stack

```
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;
}
```



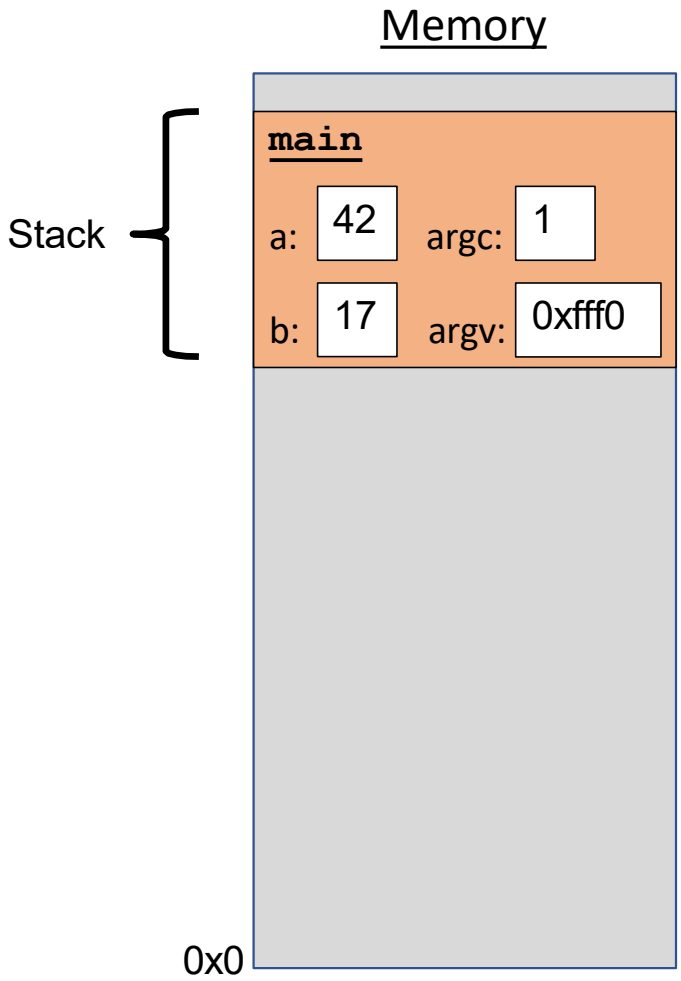
The Stack

```
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;
}
```



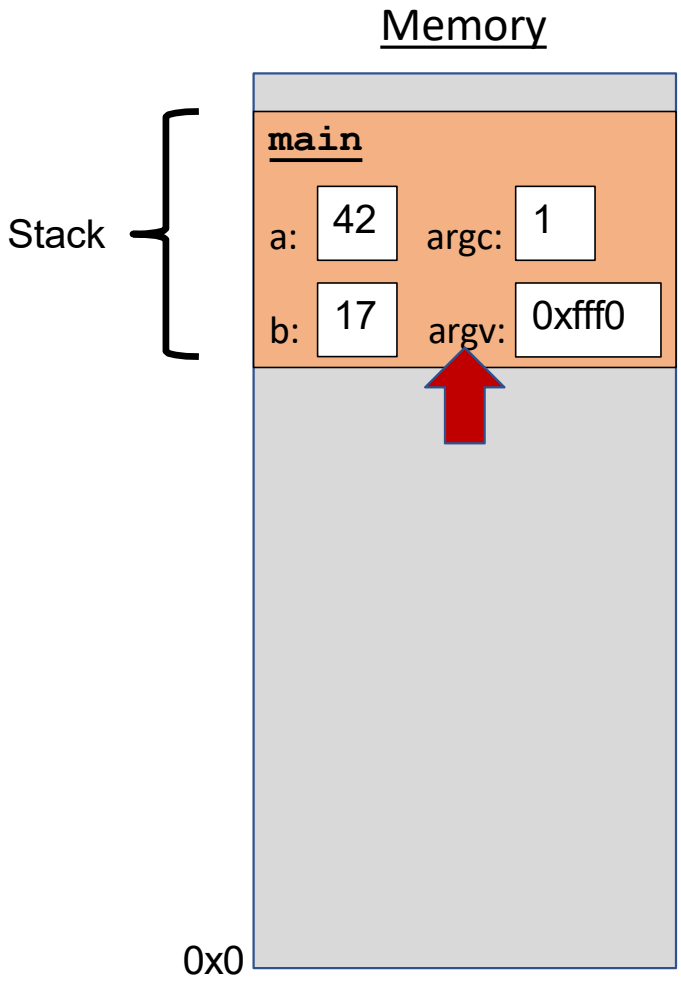
The Stack

```
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;
}
```



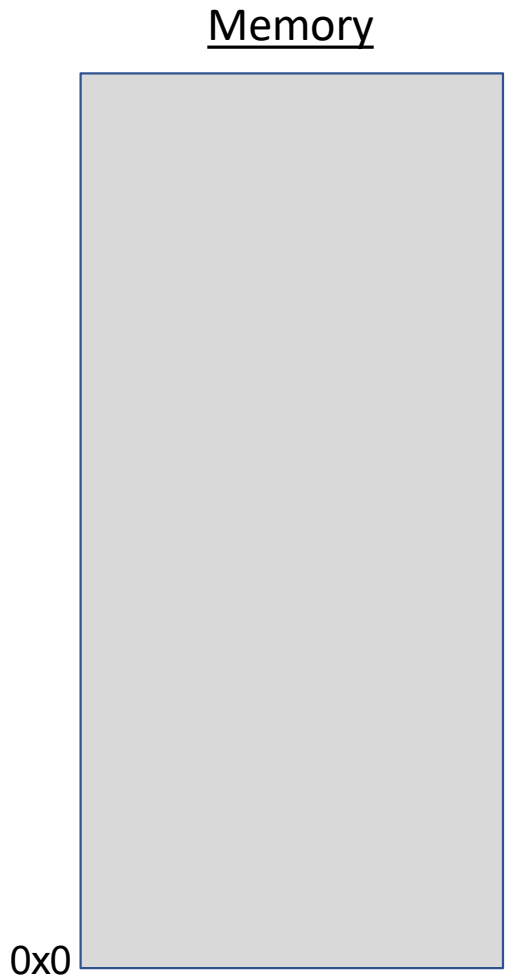
The Stack

```
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;
}
```



The Stack

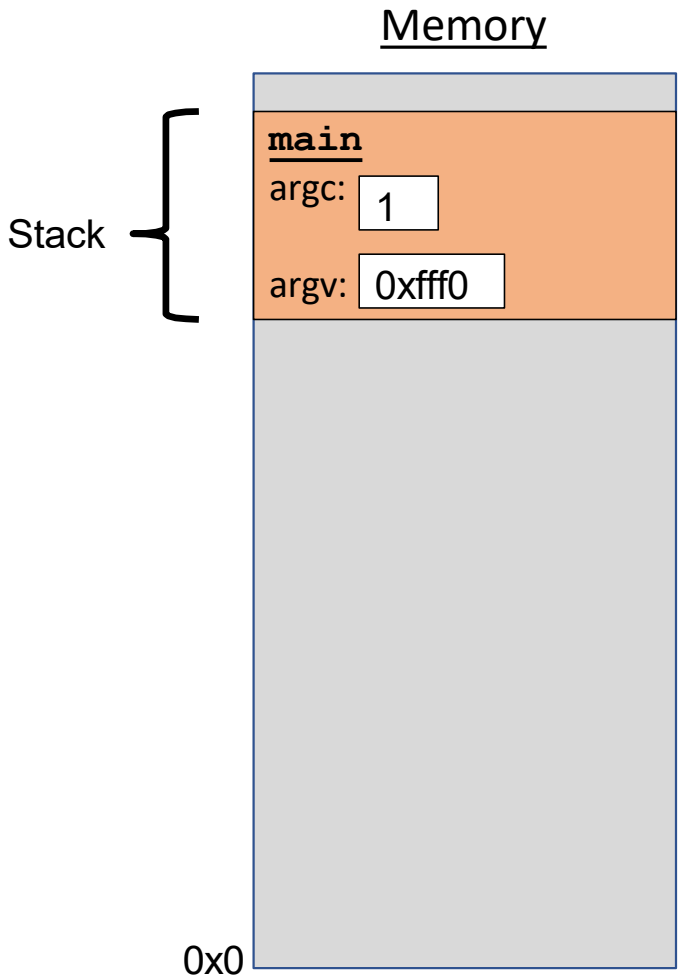
```
void func2() {
    int d = 0;
}

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

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



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

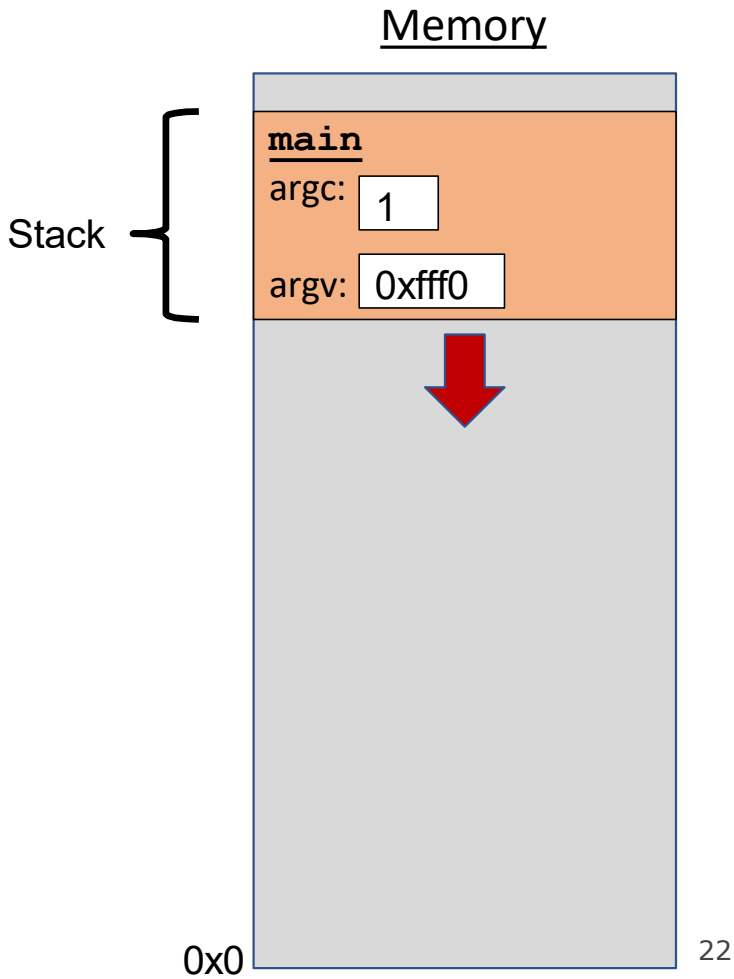
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

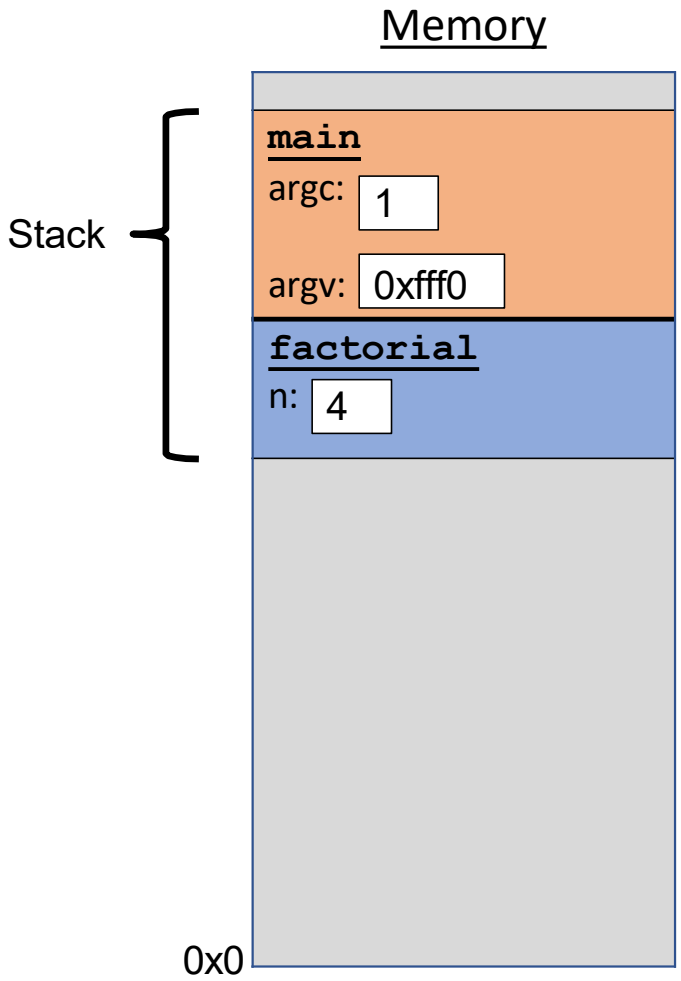
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

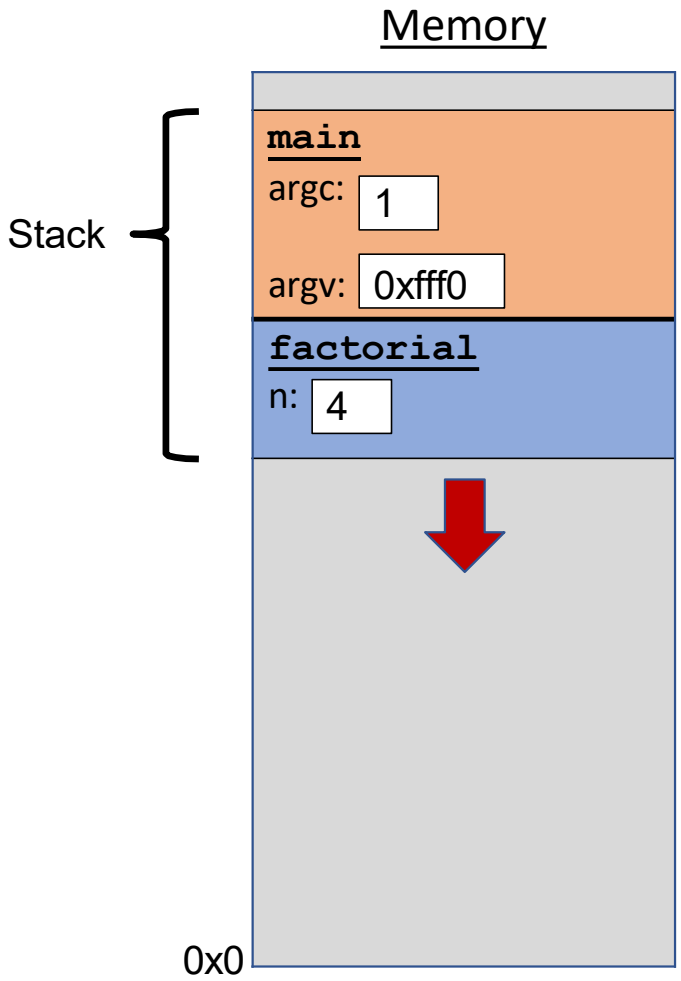
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

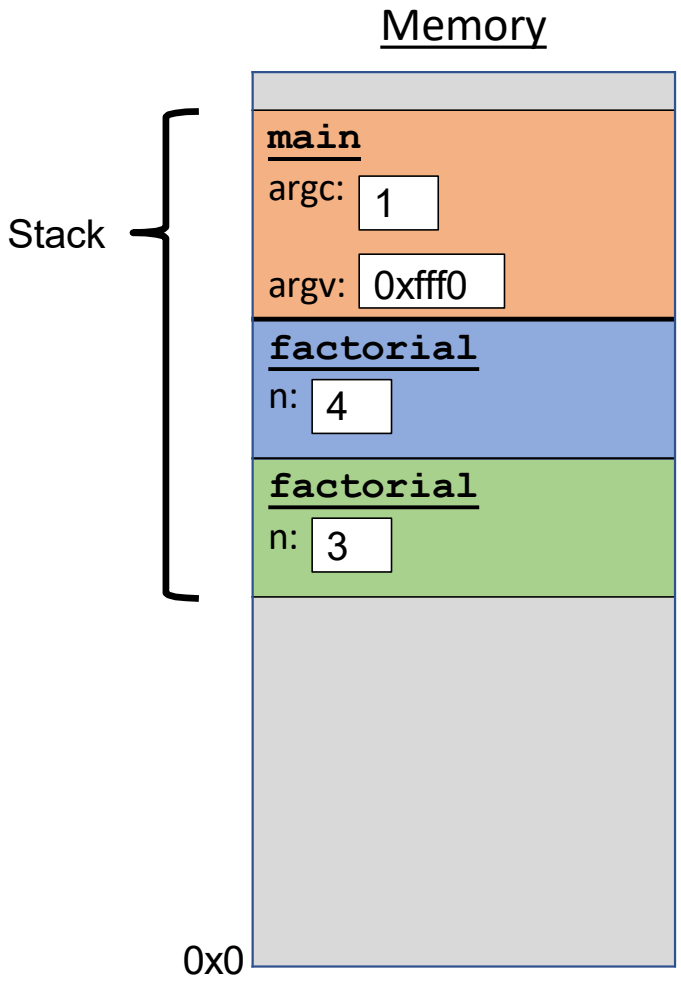
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

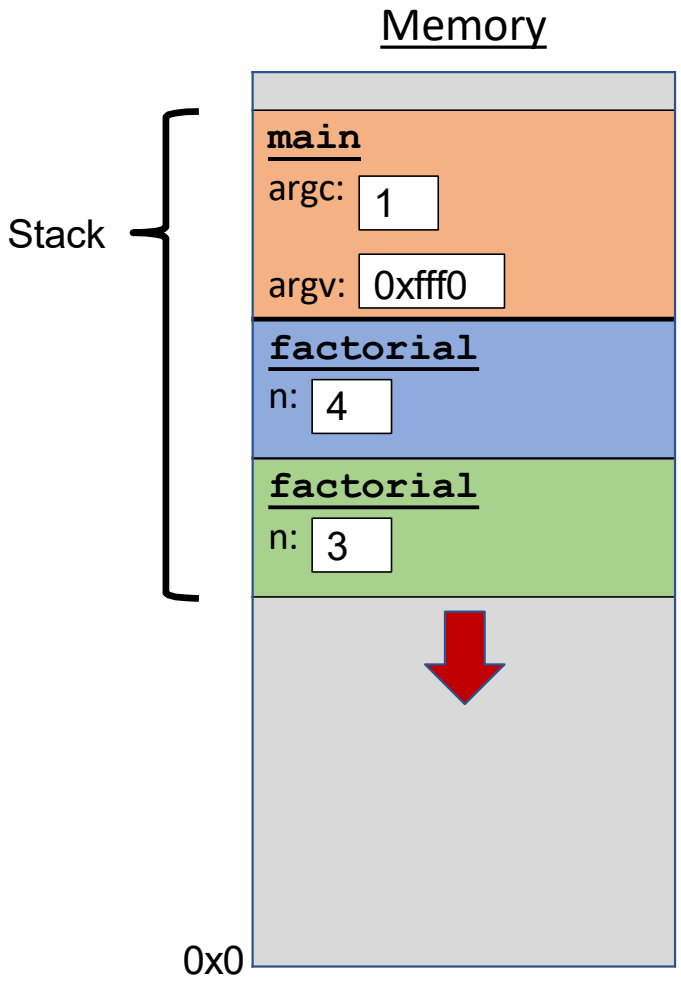
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

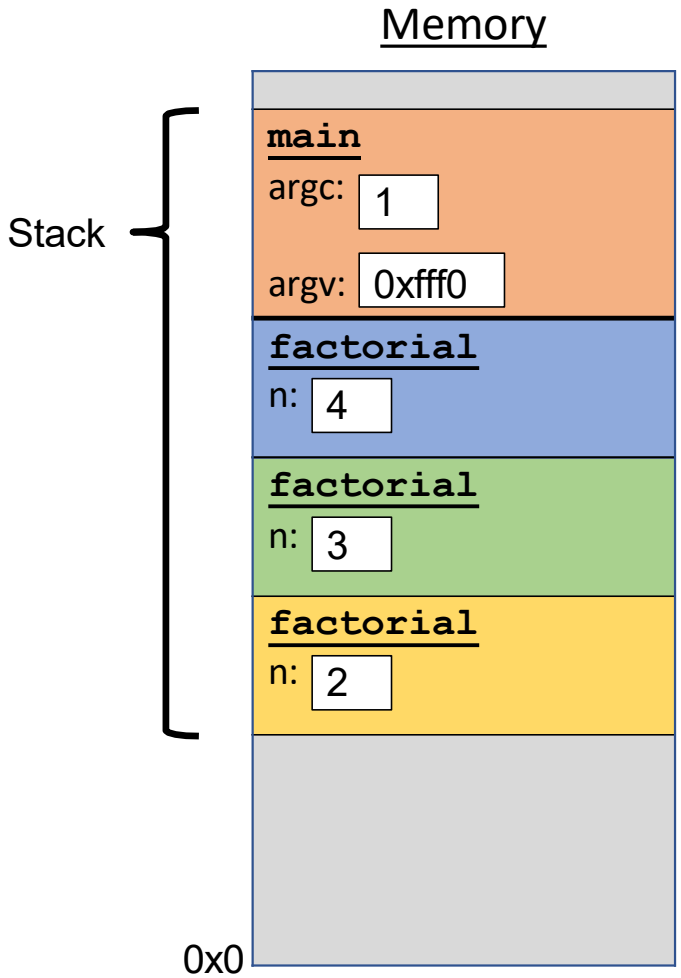
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

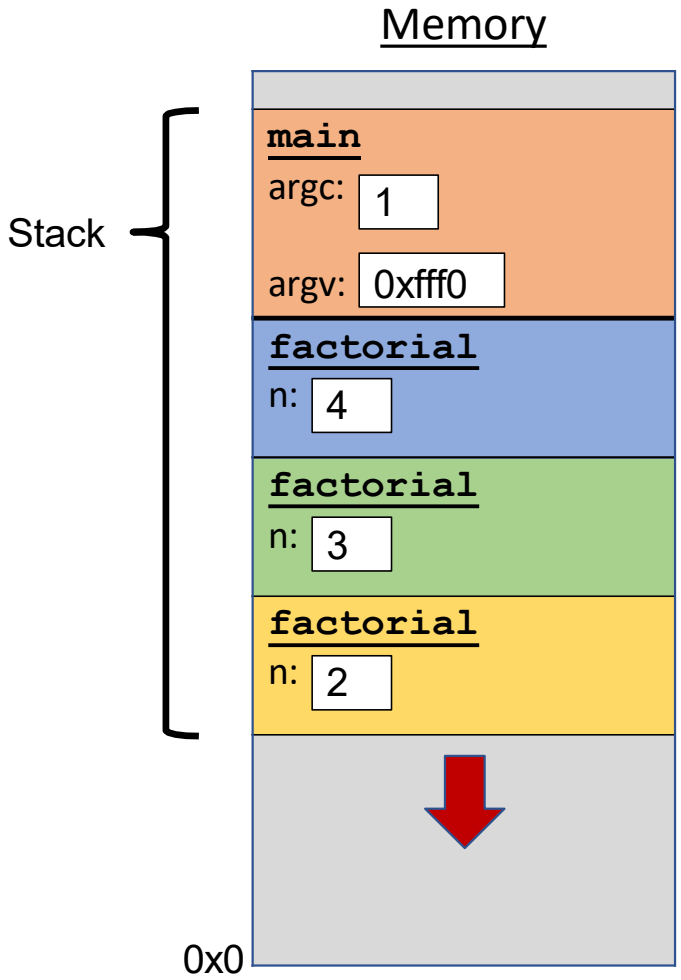
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

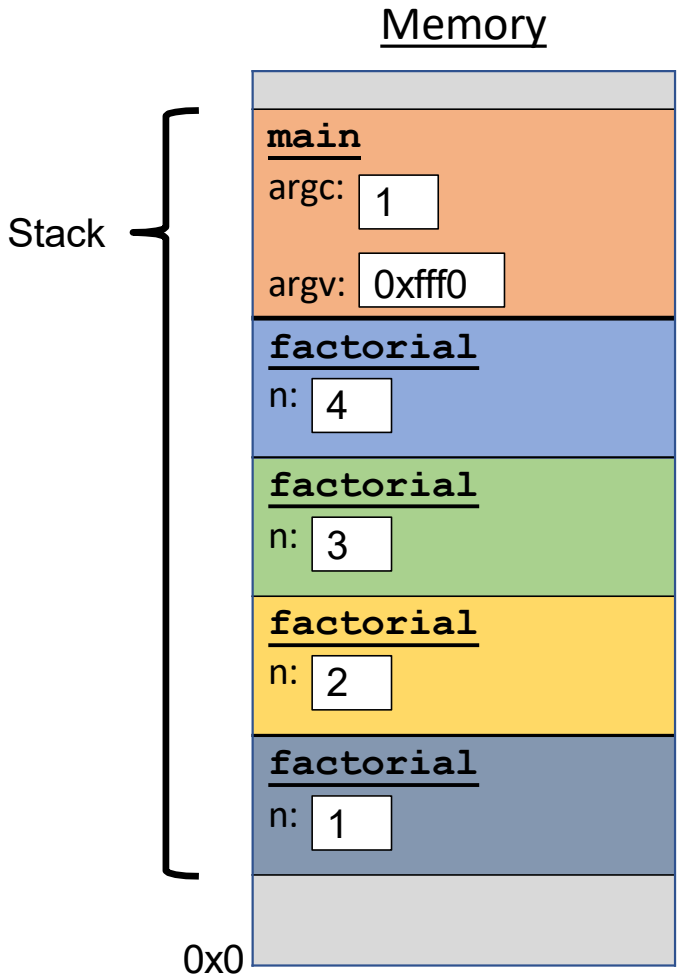
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

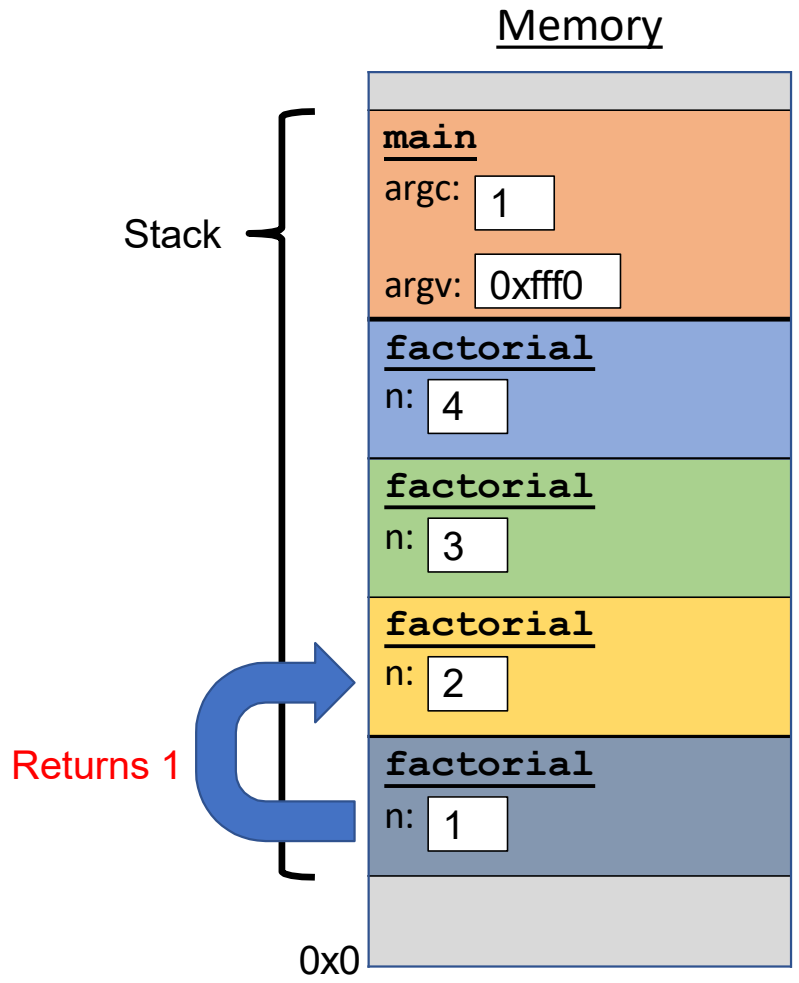
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

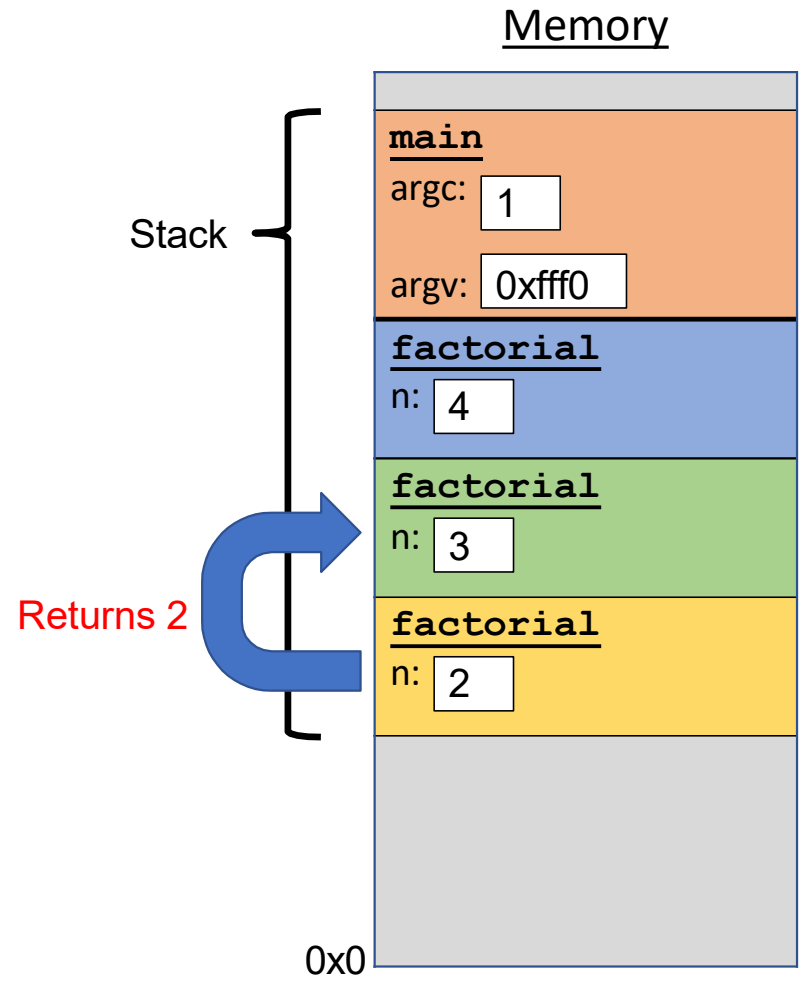
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

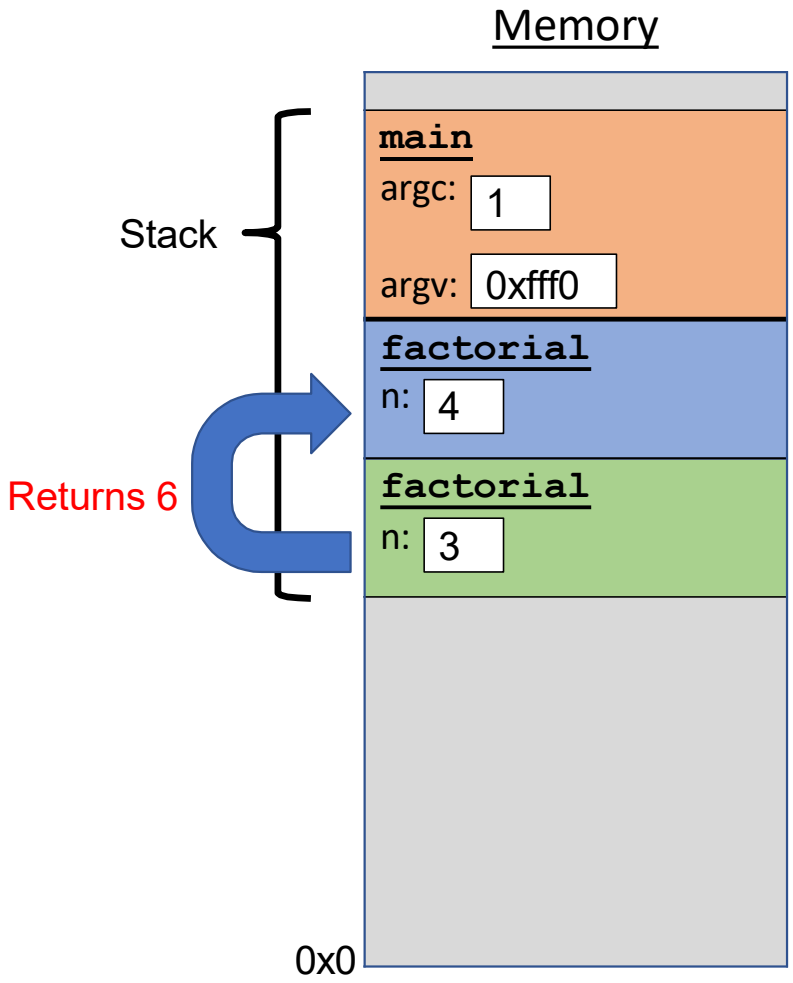
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

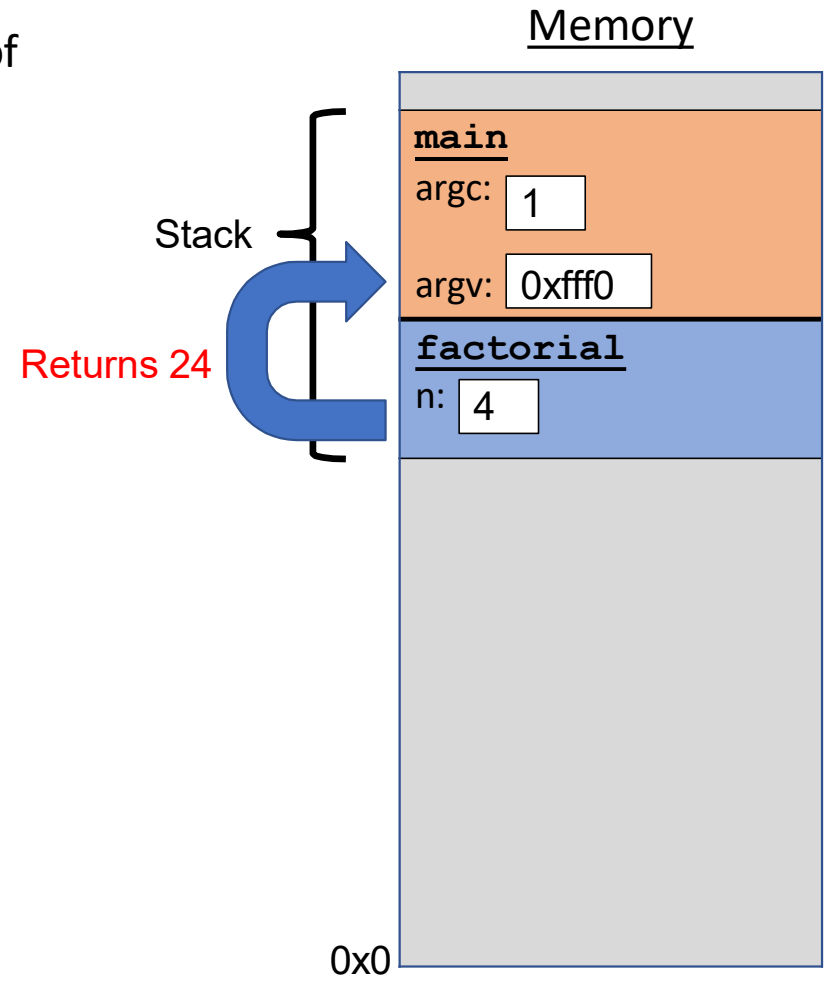
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

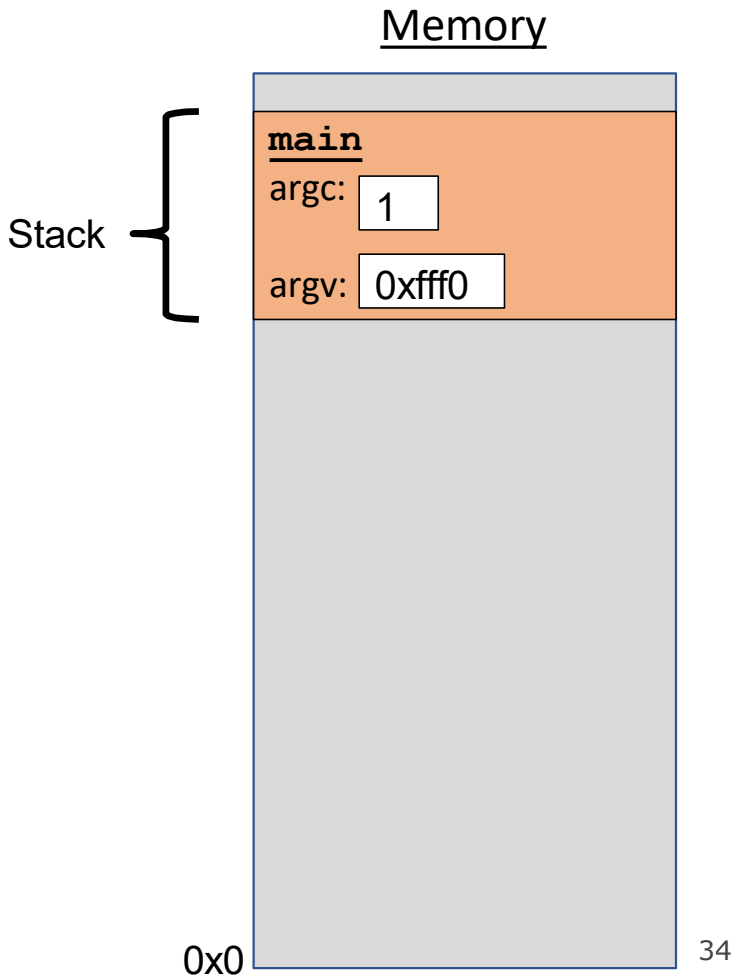
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

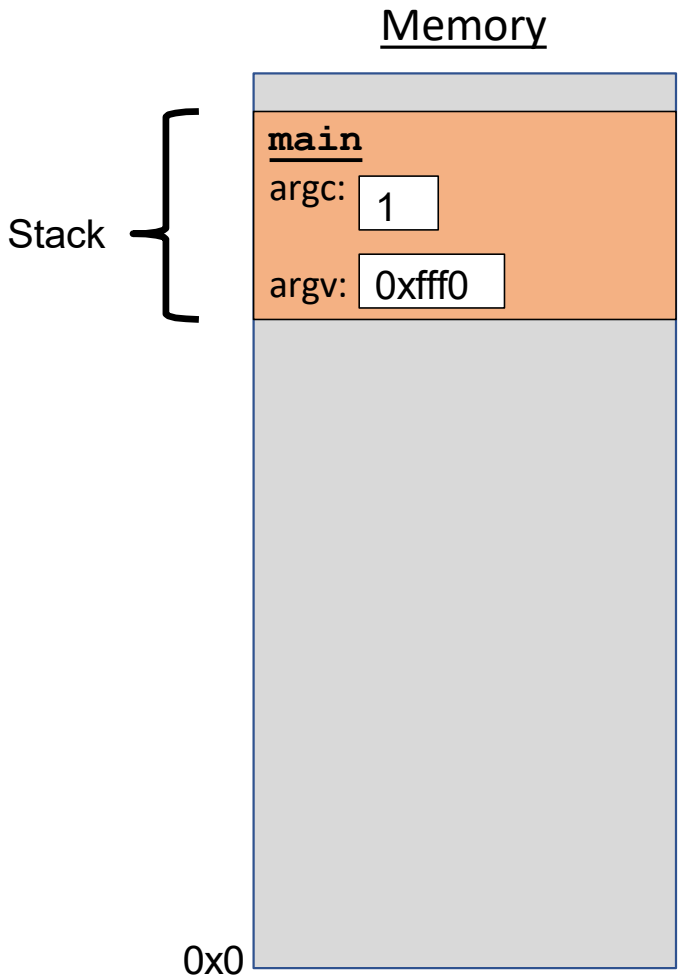
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

- The stack behaves like a…well…stack! A new function call **pushes** on a new frame. A completed function call **pops** off the most recent frame.

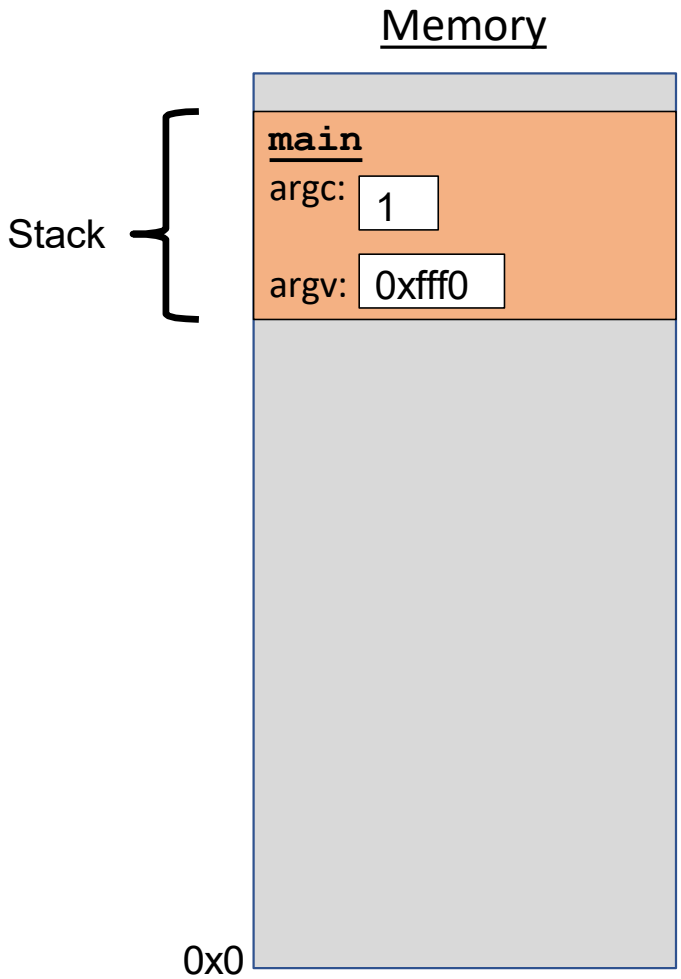
- *Interesting fact:* C does not clear out memory when a function's frame is removed. Instead, it just marks that memory as usable for the next function call. This is more efficient!

- What are the limitations of the stack?

- A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.

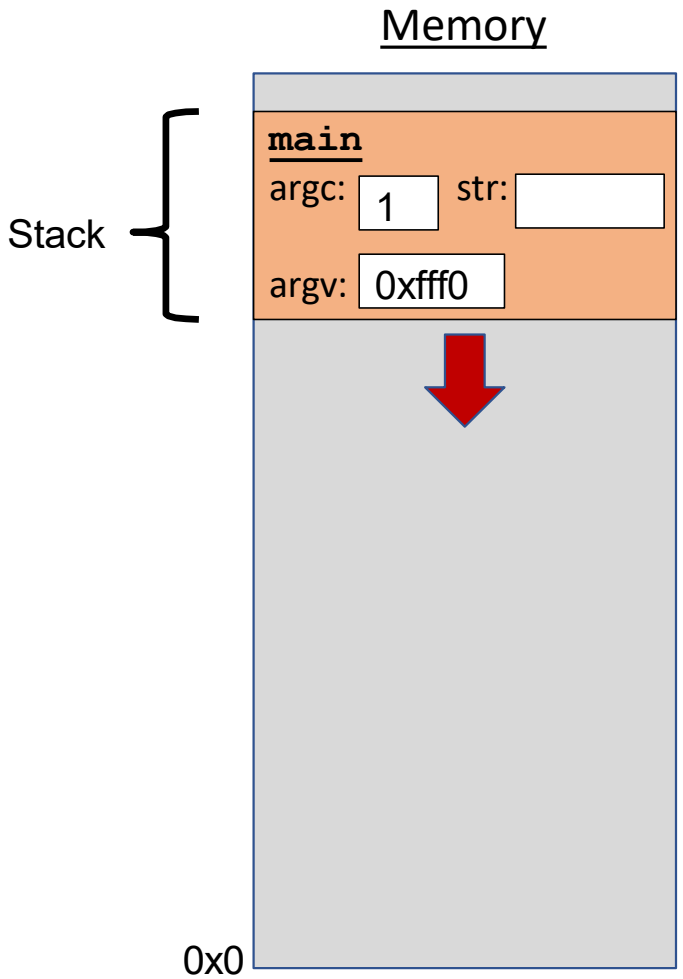
The Stack

```
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;
}
```



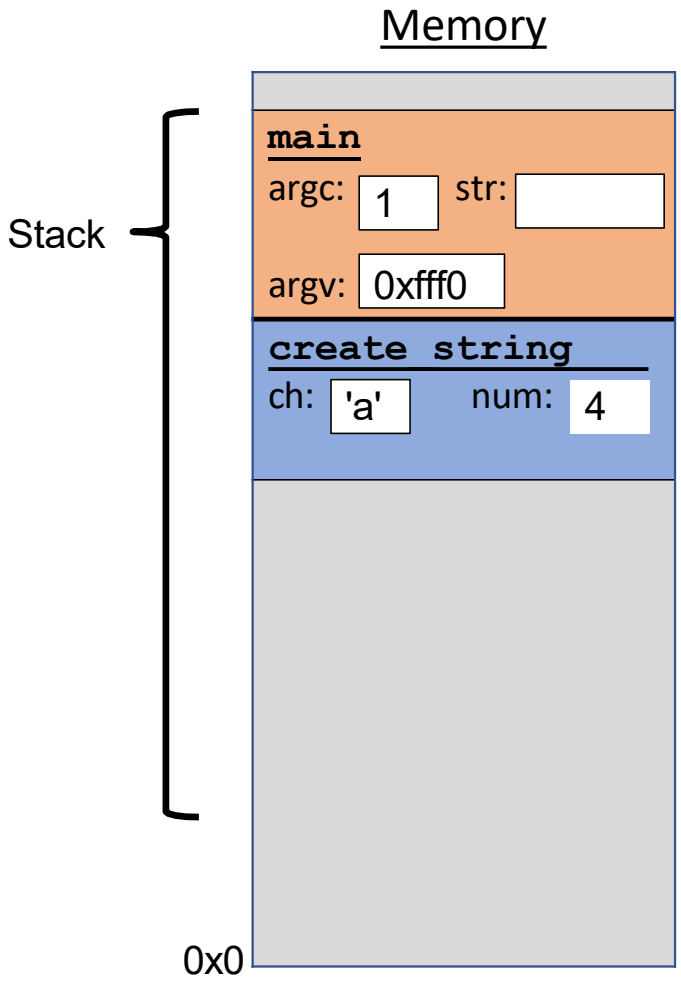
The Stack

```
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;
}
```



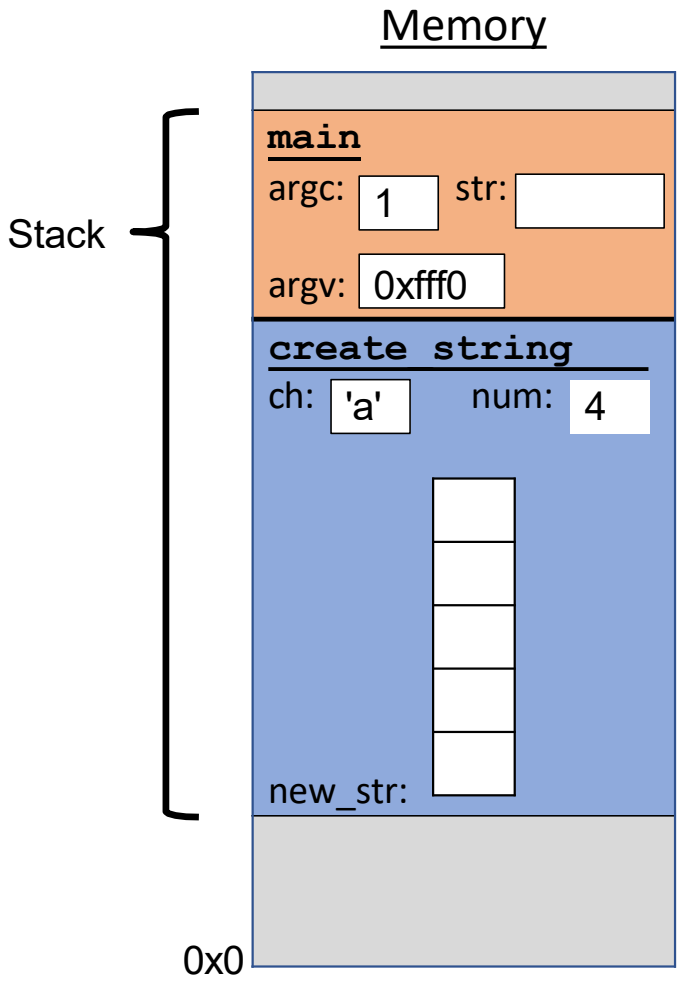
The Stack

```
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;
}
```



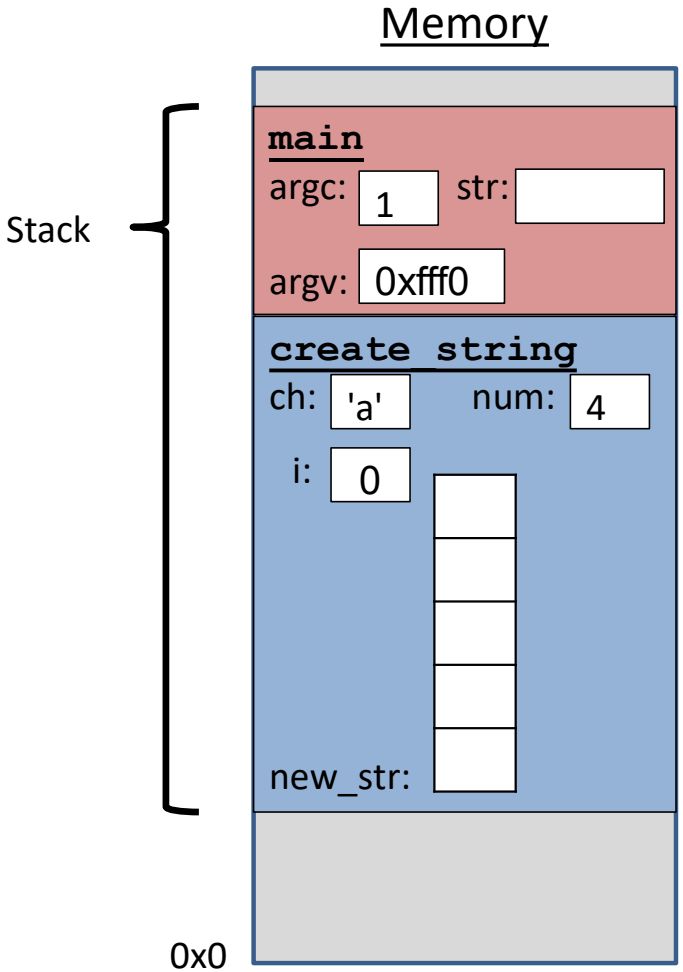
The Stack

```
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;
}
```



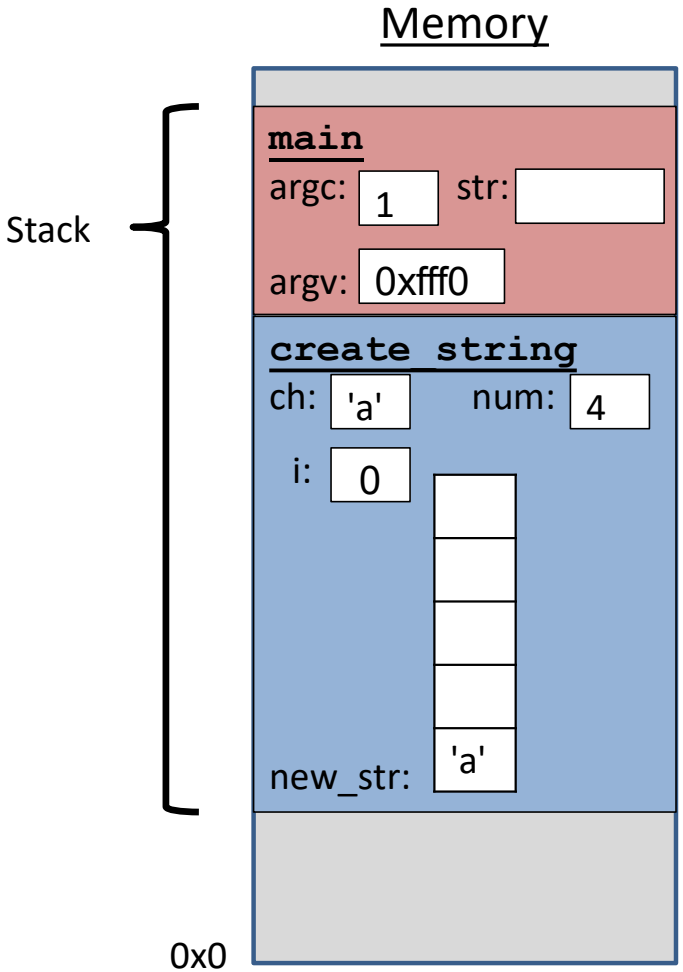
The Stack

```
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;
}
```



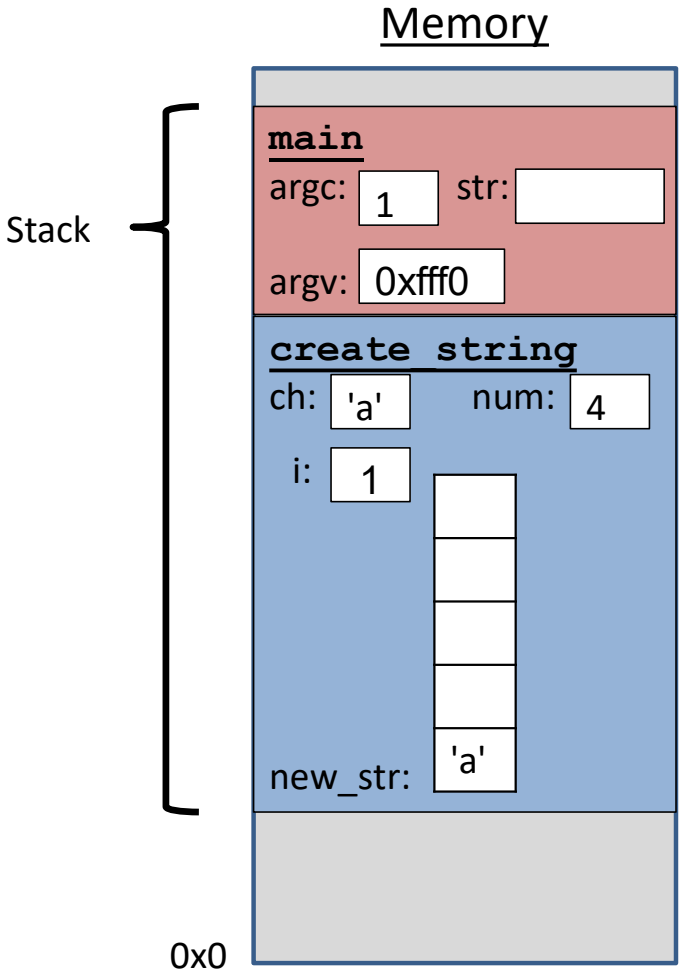
The Stack

```
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;
}
```



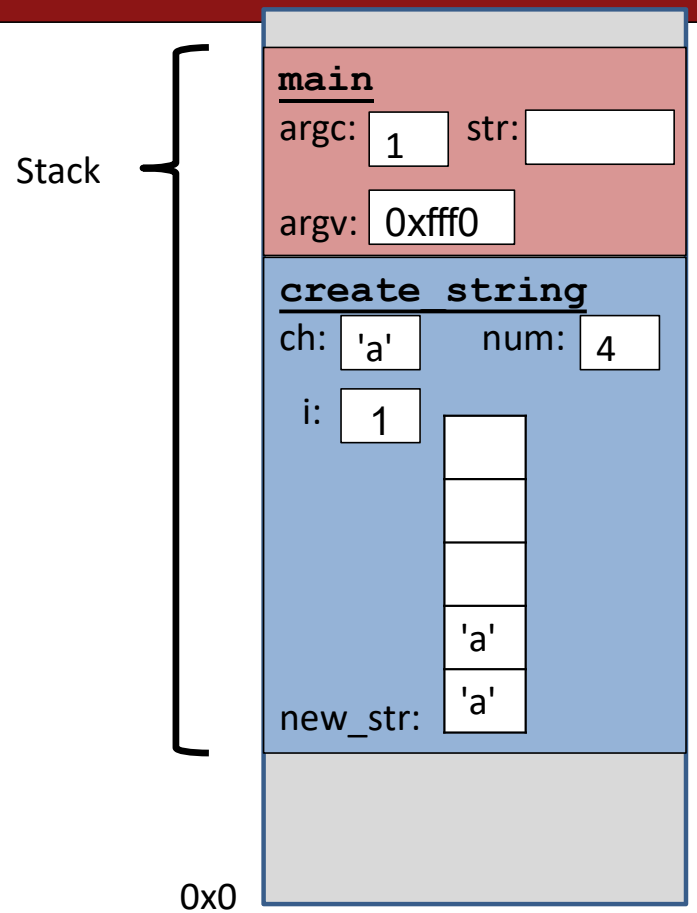
The Stack

```
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;
}
```



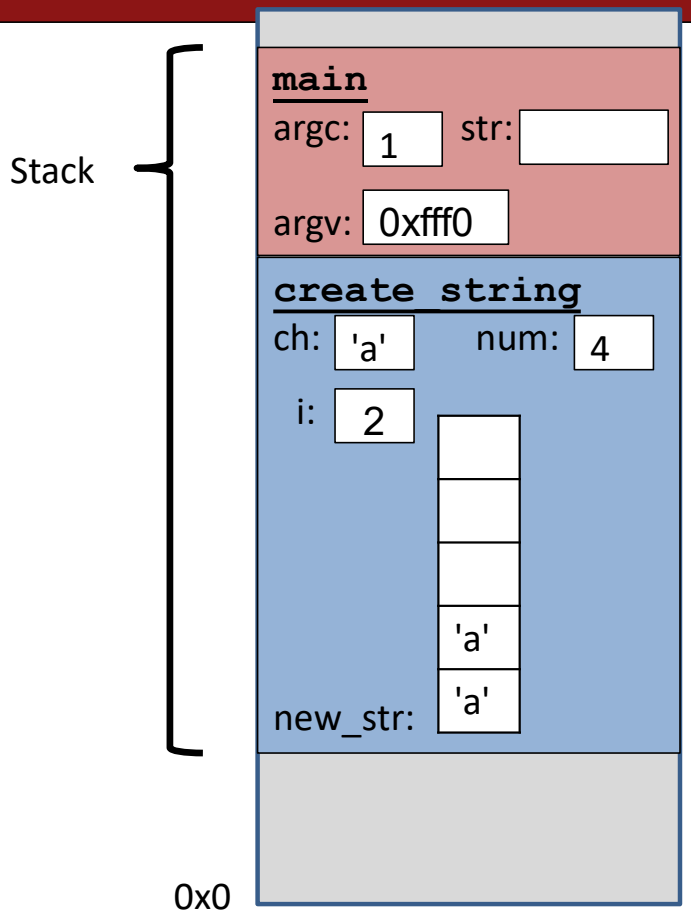
The Stack

```
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;
}
```



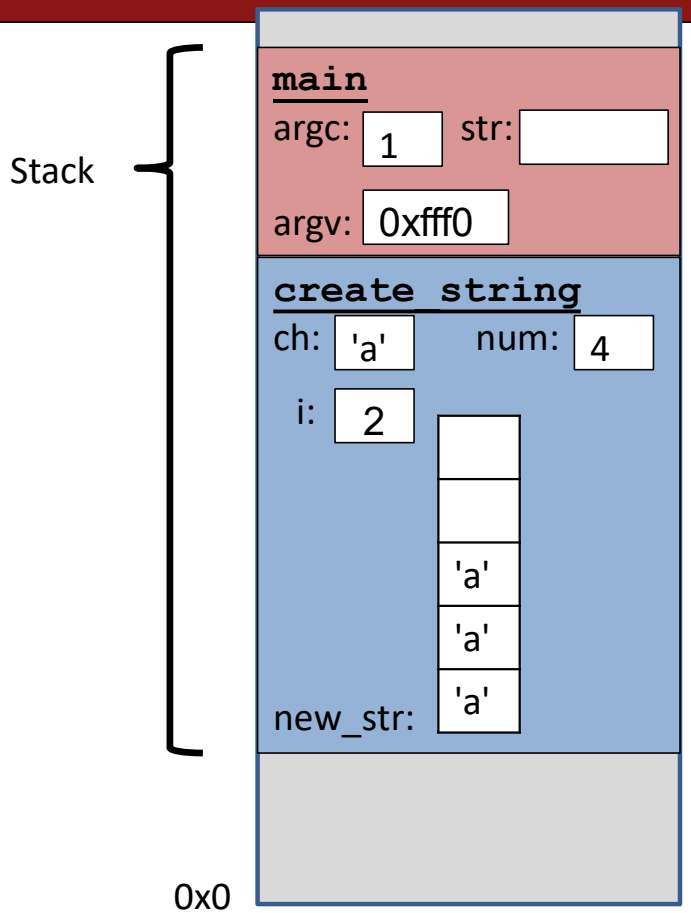
The Stack

```
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;
}
```



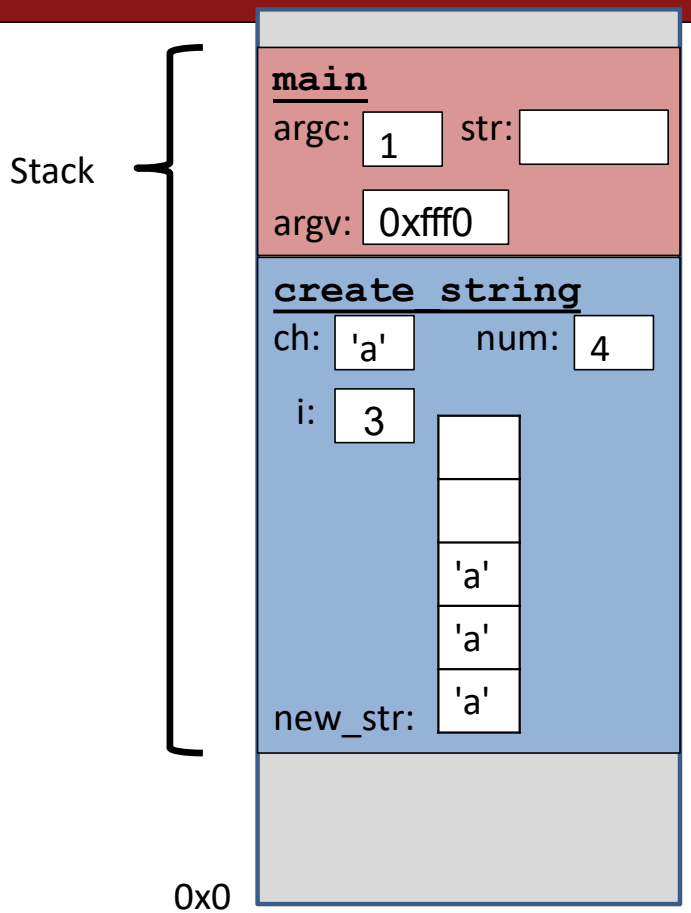
The Stack

```
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;
}
```



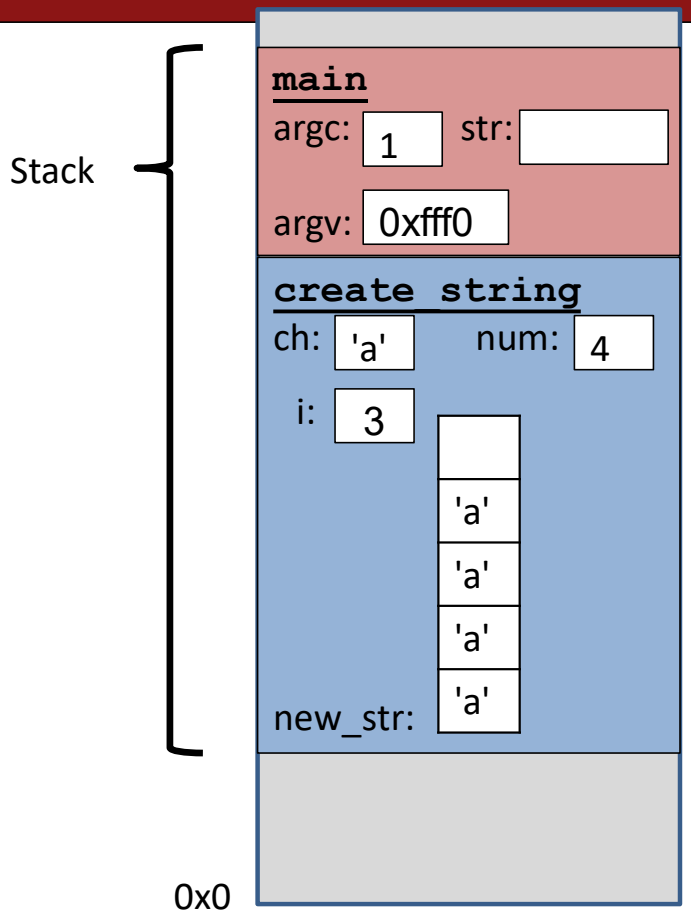
The Stack

```
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;
}
```



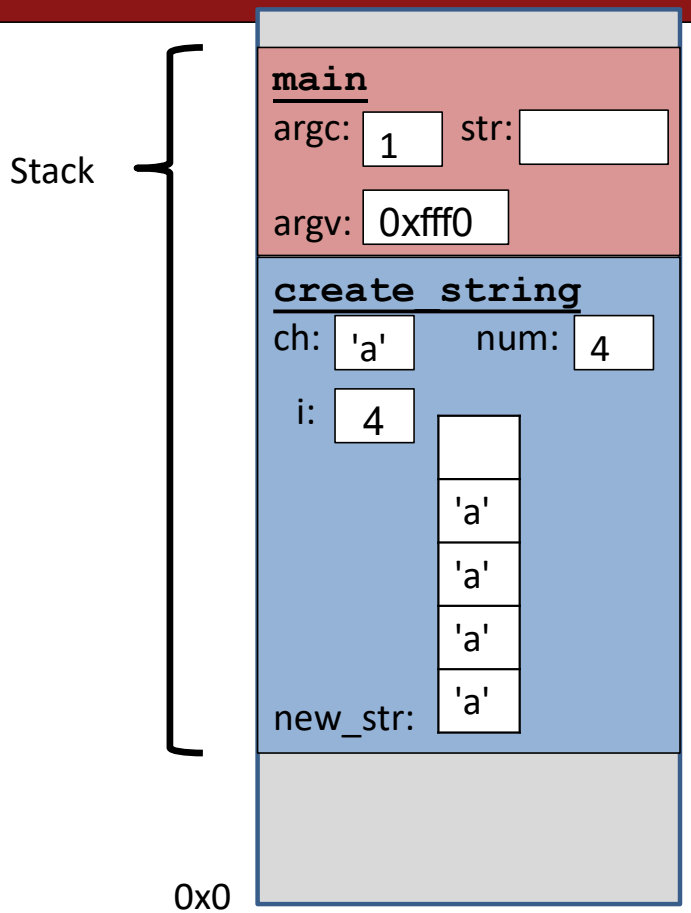
The Stack

```
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;
}
```



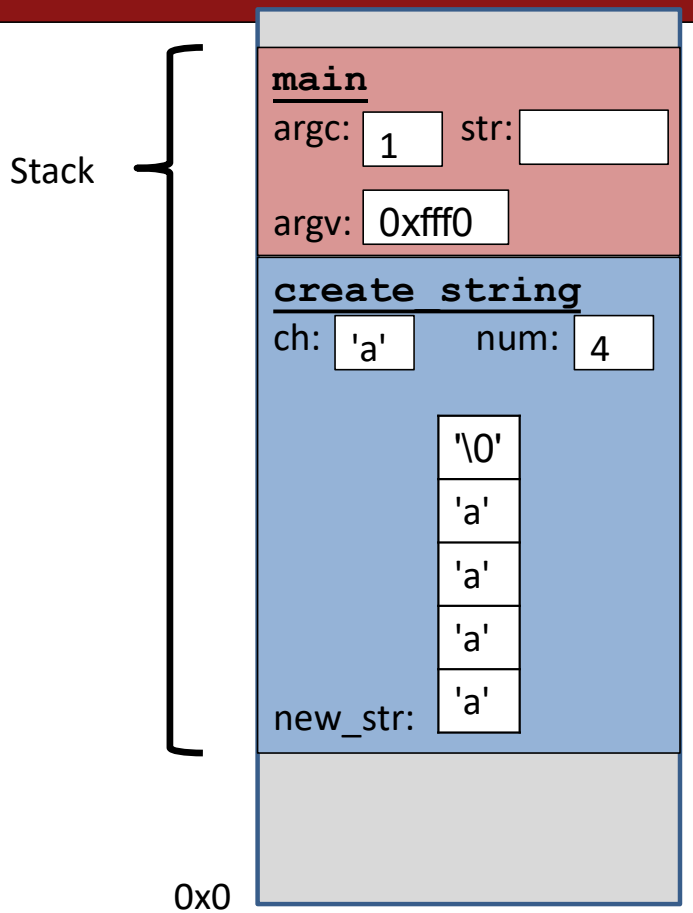
The Stack

```
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;
}
```



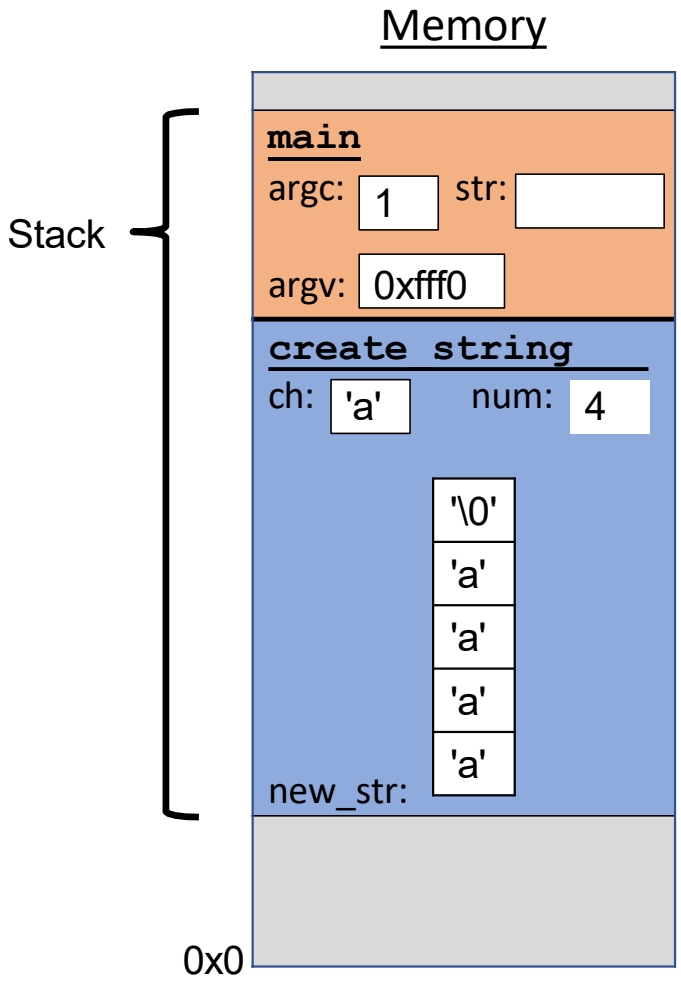
The Stack

```
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

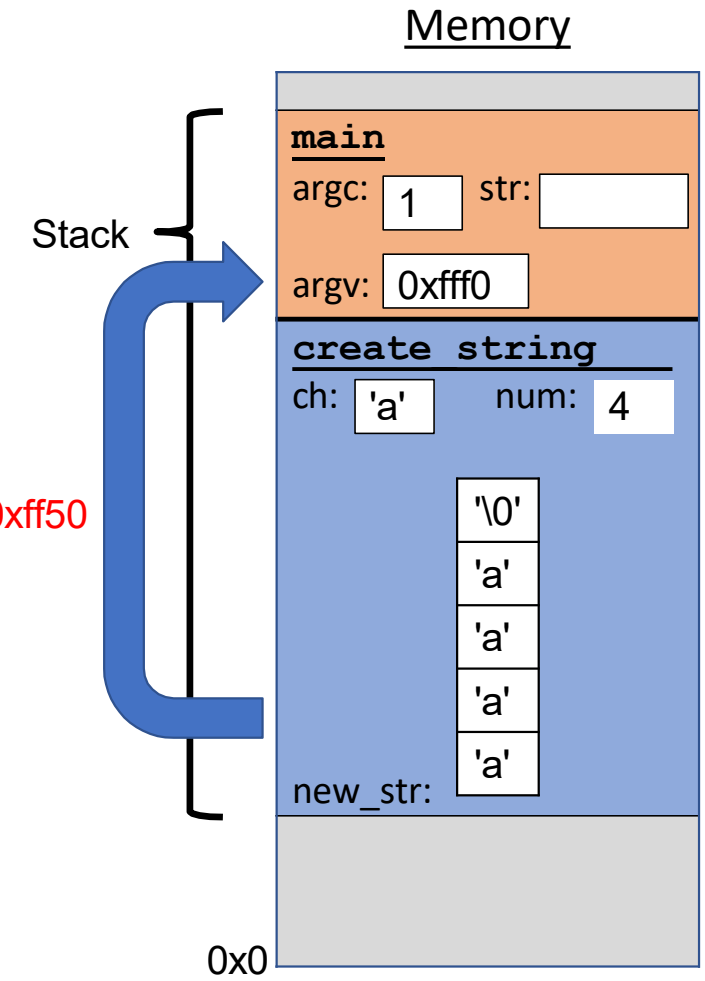
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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



The Stack

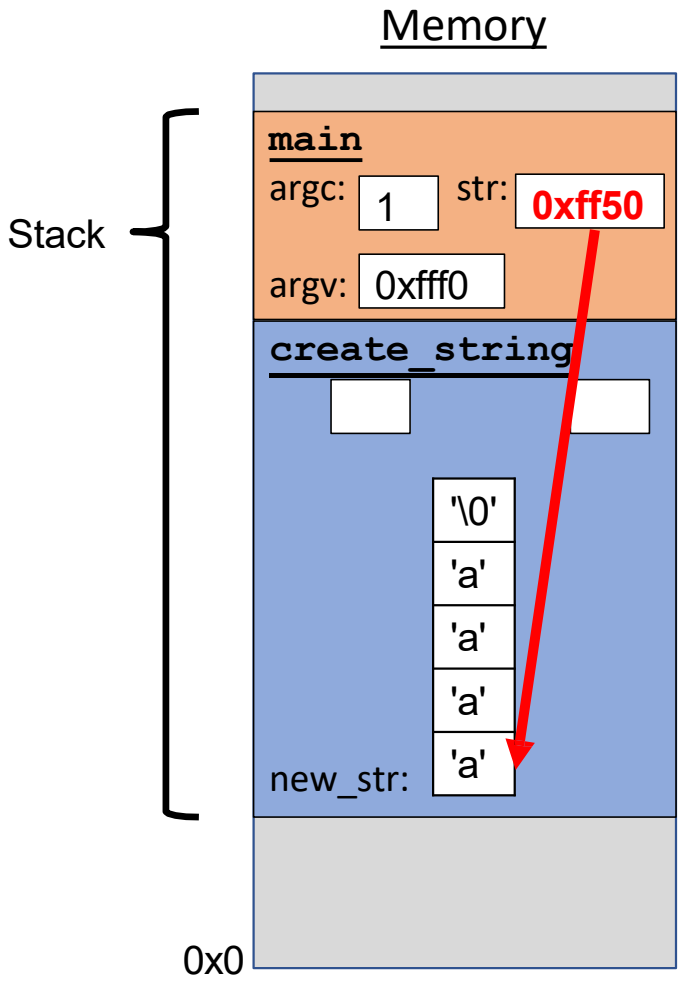
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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

Returns e.g. 0xff50



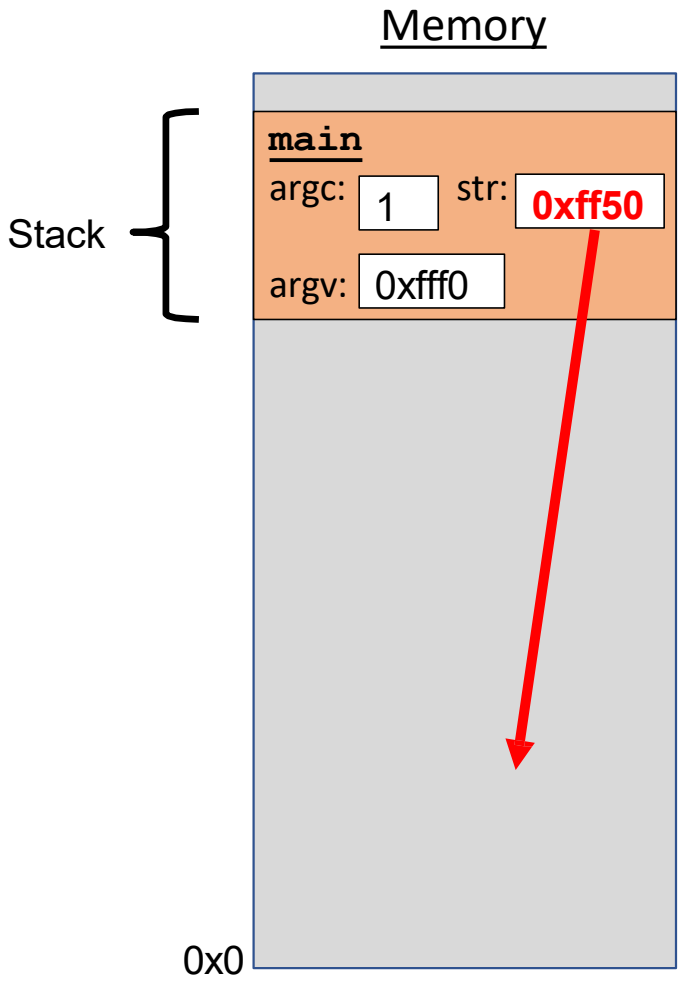
The Stack

```
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

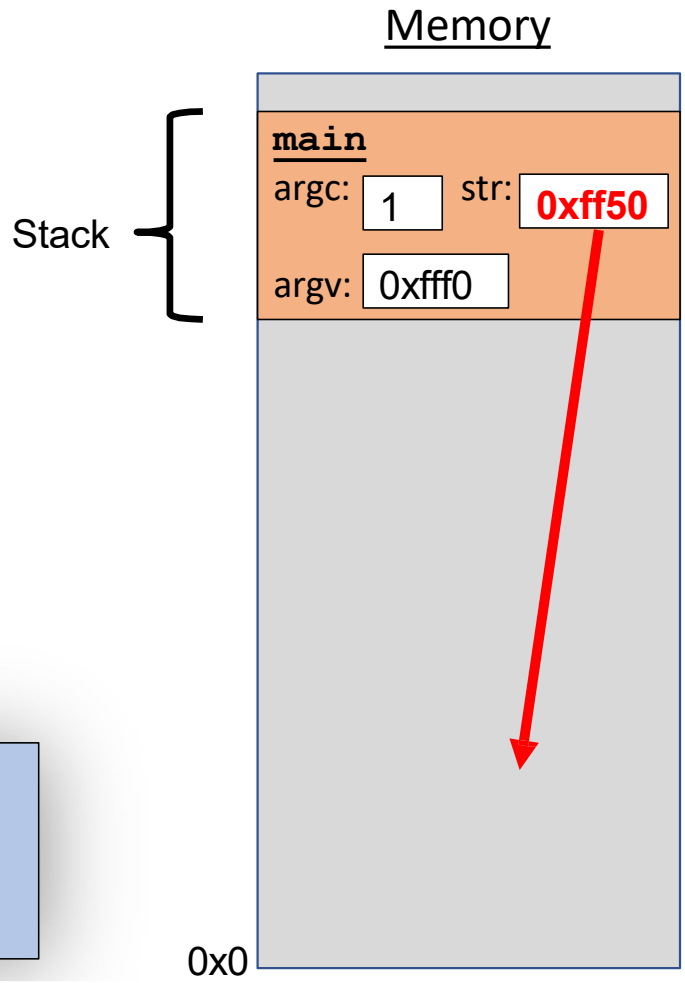
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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



The Stack

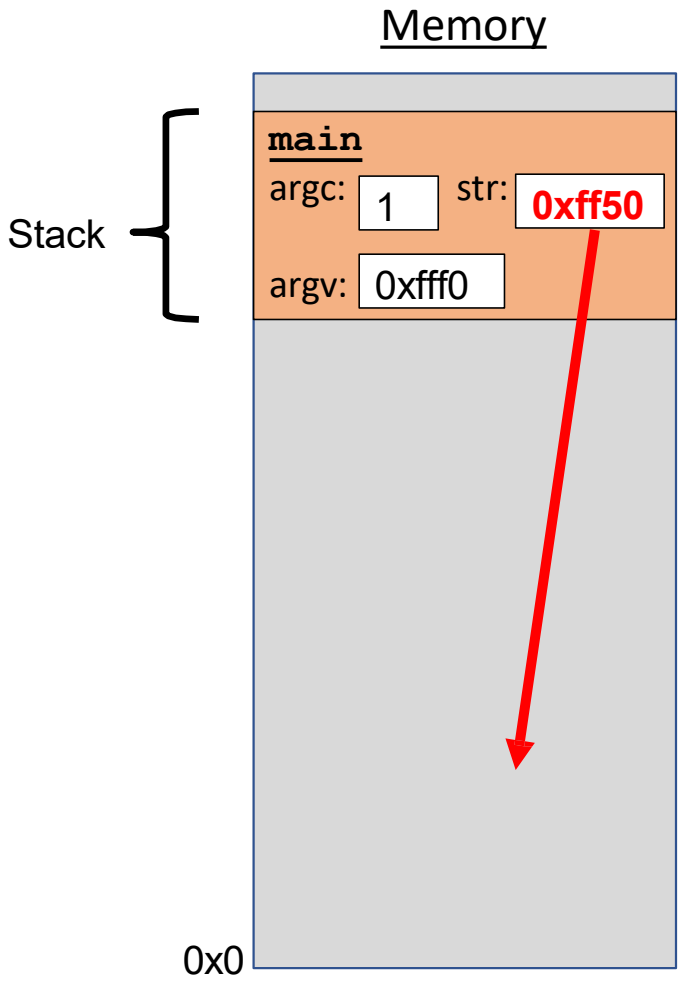
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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

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



The Stack

```
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;
}
```



Stacked Against Us

This is a problem! We need a way to have memory that doesn't get cleaned up when a function exits.

Lecture Plan

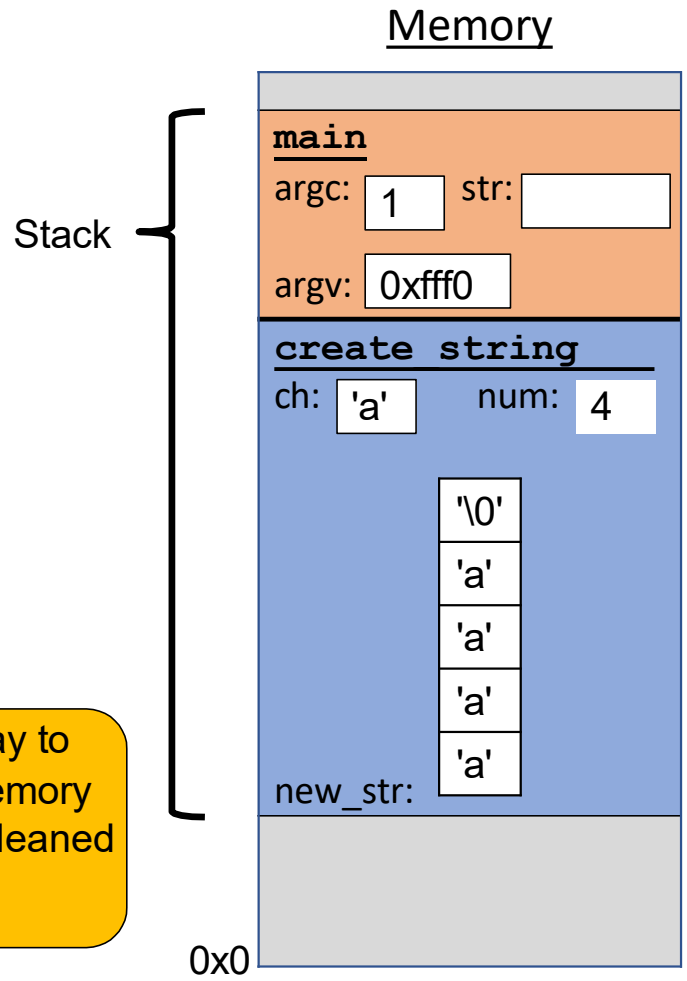
- The Stack 3
- **The Heap and Dynamic Memory** 59
- `realloc` 83
- Use After Free 99


```
cp -r /afs/ir/class/cs107/lecture-code/lect07 .
```

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;
}
```

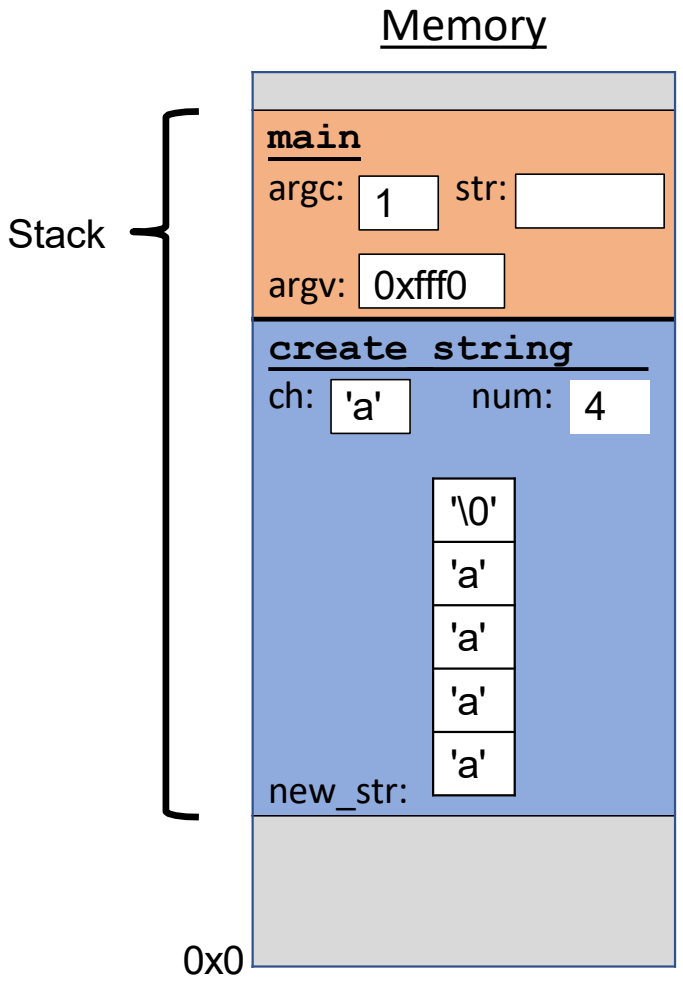
Us: Hey C, is there a way to make this variable in memory that isn't automatically cleaned up?



The Heap

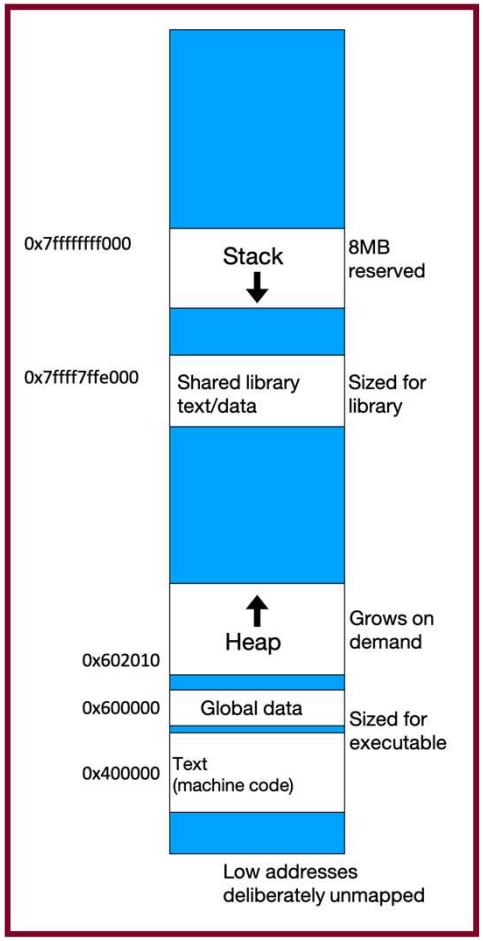
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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 clean it up anymore, it's your responsibility…



The Heap

- The **heap** is a part of memory that you can manage yourself.
- 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 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**.



malloc

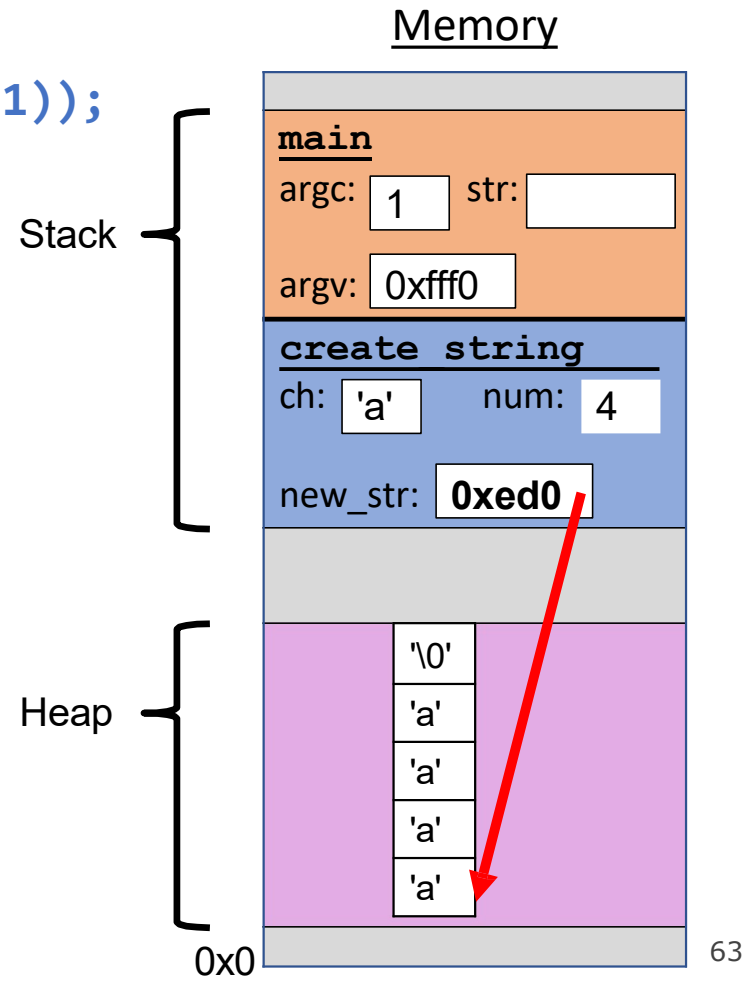
```
void *malloc(size_t size);
```

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

- This function returns a pointer to *the **starting address** of the new memory*. It doesn't know or care whether it will be used as an array, a single block of memory, etc.
- **`void *`**means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns `NULL`, then there wasn't enough memory for this request.

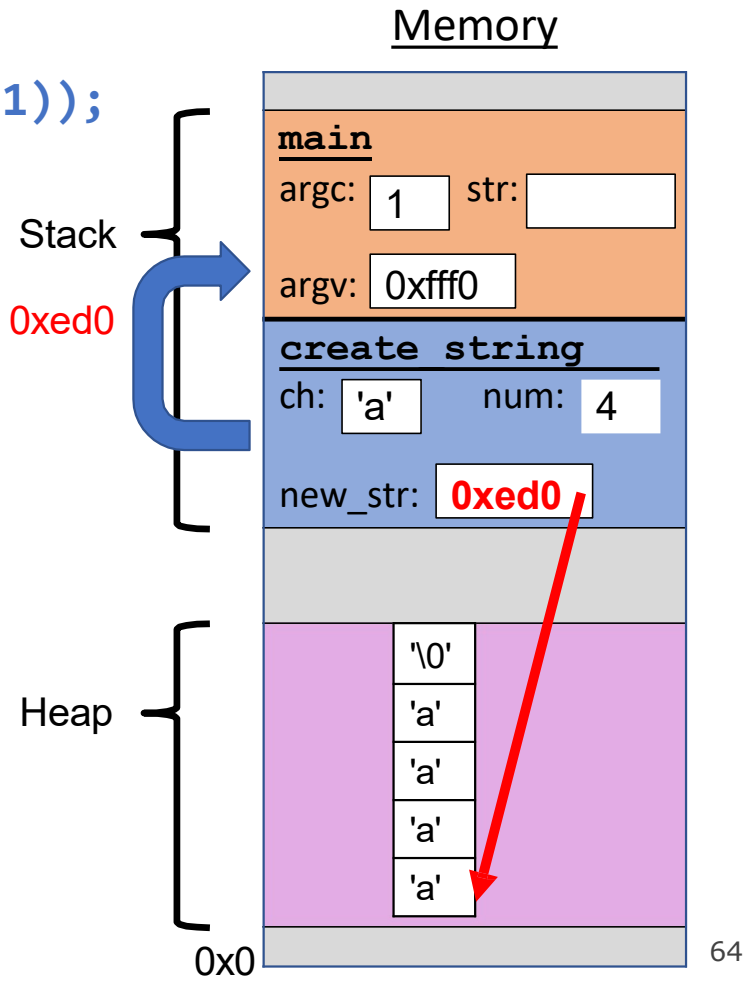
The Heap

```
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;
}
```



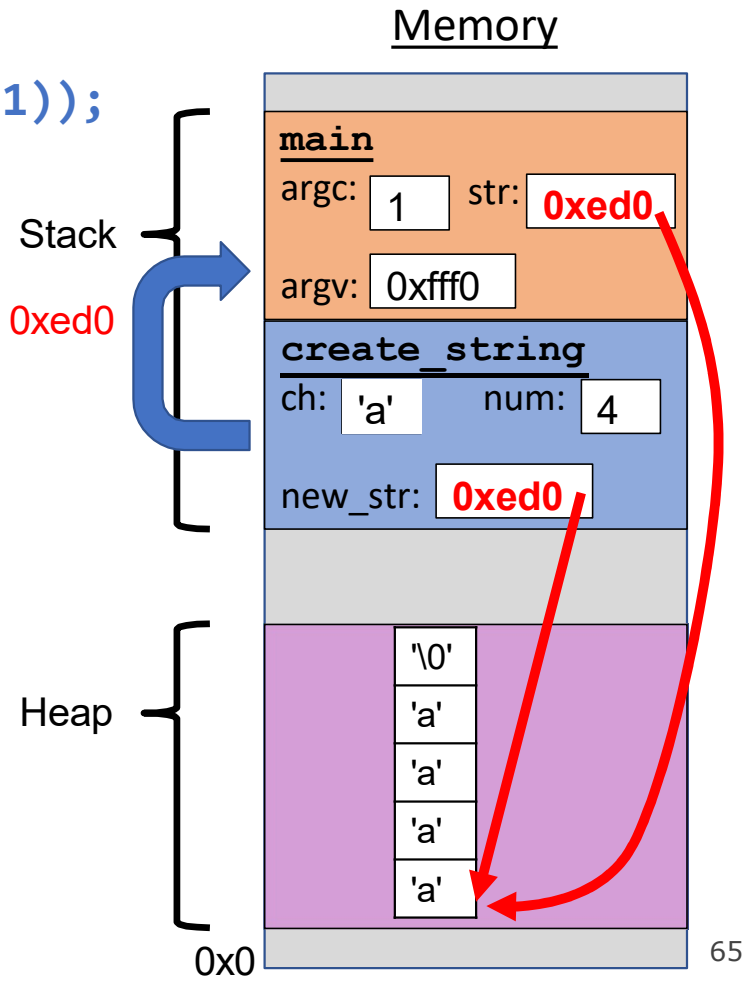
The Heap

```
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;
}
```



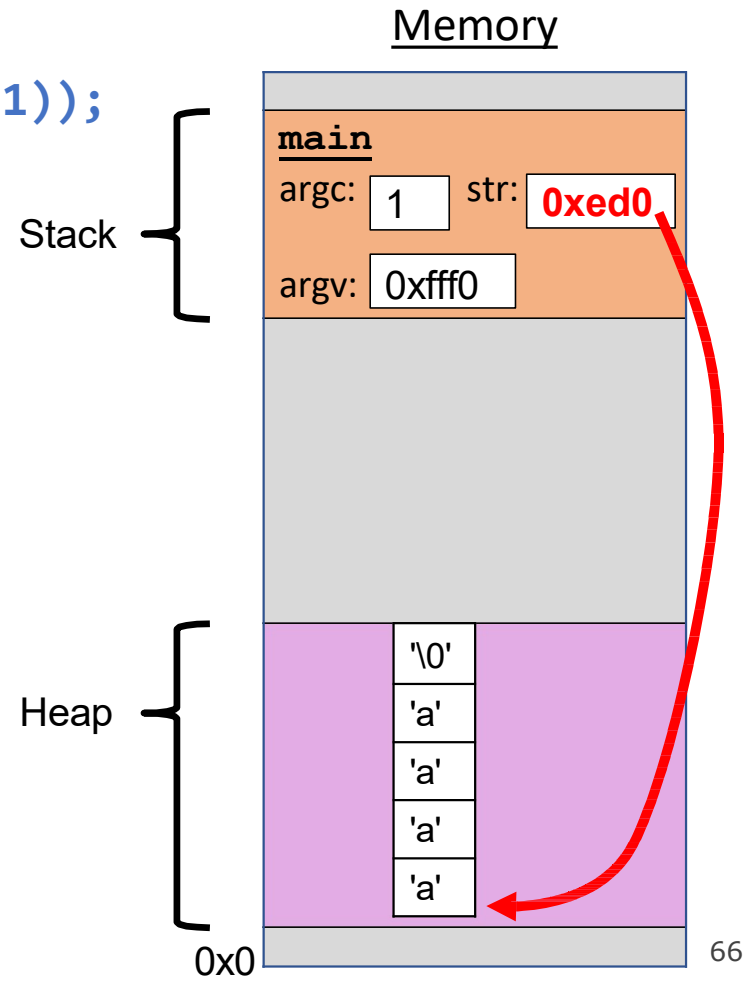
The Heap

```
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;
}
```



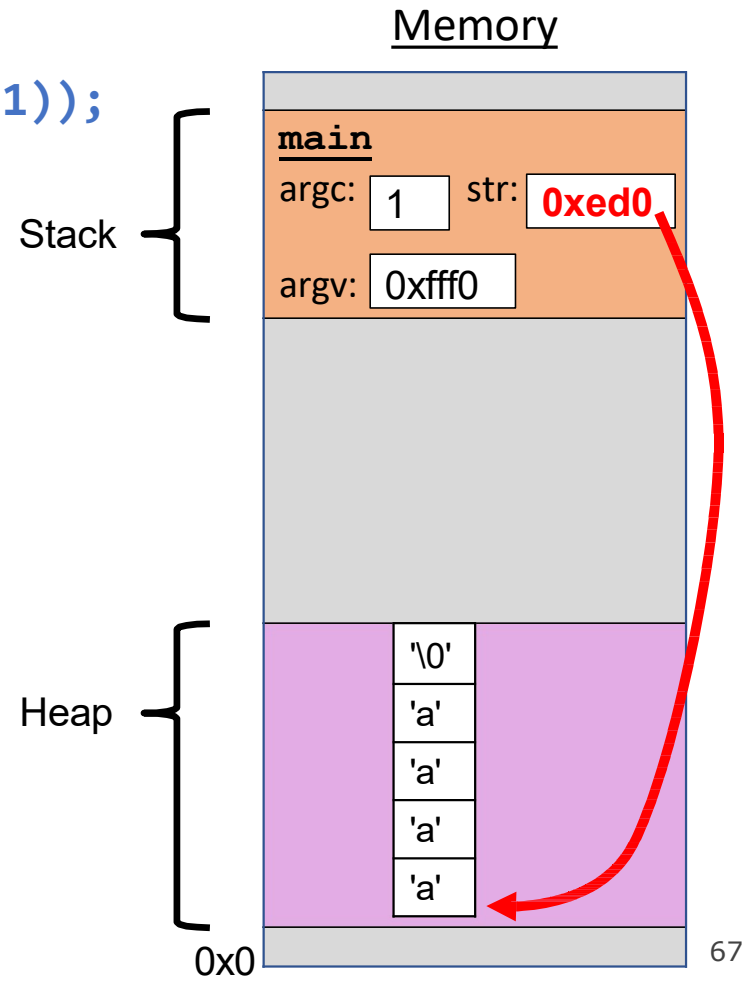
The Heap

```
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;
}
```



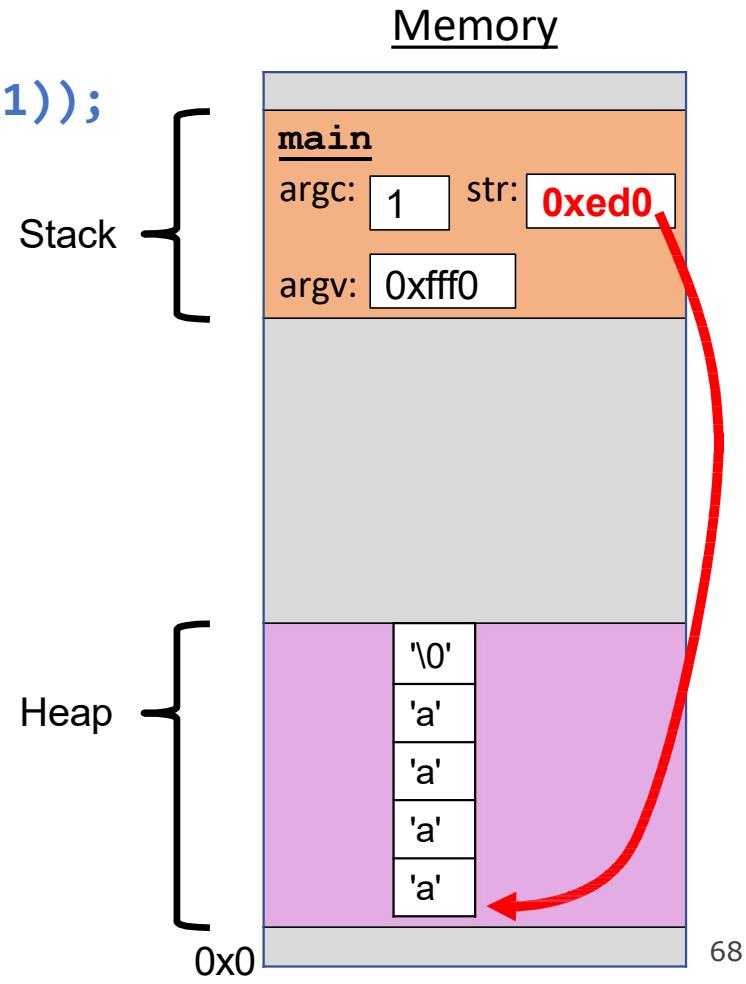
The Heap

```
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;
}
```



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;
}
```



Exercise: `malloc` multiples

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

And adds a '\0' at the end

```
1 int *array_of_multiples(int mult, int len)
2 {
3     /* TODO: arr declaration here */
4     for (int i = 0; i < len; i++){
5         arr[i] = mult * (i + 1);
6     }
7     arr[len] = '\0';
8     return arr;
9 }
```

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));`
E. Something else

Exercise: `malloc` multiples

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

And adds a '\0' at the end

```
1 int *array_of_multiples(int mult, int len)
2 {
3     /* TODO: arr declaration here */
4     for (int i = 0; i < len; i++){
5         arr[i] = mult * (i + 1);
6     }
7     arr[len] = '\0';
8     return arr;
9 }
```

- 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));`
E. Something else

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 + 1) );
3     assert(arr != NULL);
4     for (int i = 0; i < len; i++){
5         arr[i] = mult * (i + 1);
6     }
7     arr[len] = '\0';
8     return arr;
9 }
```


- 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 crash the program if the provided condition is false. A memory allocation error is significant, and we should terminate the program.

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';
```

`strdup` means string duplicate

How can we implement **strdup** using functions we've already seen?

```
1 char *mystrdup(const char *str) {
2     char *heapstr = ____(A)____;
3     ____(B)____;
4     ____(C)____;
5     return heapstr;
6 }
```

[Note] Use library functions:
`<stdlib.h>`: malloc
`<assert.h>`: assert
`<string.h>`: strcpy, strlen

`strdup` means string duplicate

How can we implement **strdup** using functions we've already seen?

```
1 char *mystrdup(const char *str) {
2     char *heapstr = malloc(strlen(str) + 1);
3     assert(heapstr != NULL);
4     strcpy(heapstr, str);
5     return heapstr;
6 }
```

`char` arrays differ from other arrays in that valid strings must be null-terminated (i.e., have an extra ending `char`).

(Note: library strdup doesn't have an `assert`—it leaves the `assert` to the callee)

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);
```

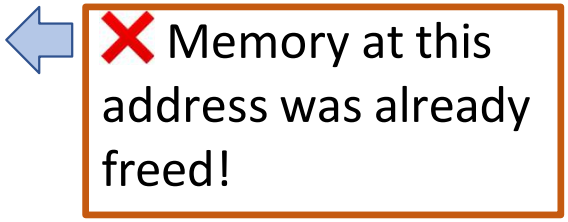
When you free an allocation, you are freeing up what it *points* to. You are not freeing the pointer itself. You can still use the pointer to point to something else.

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

free details

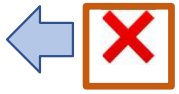
Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
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);
```



Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *str = strdup("Hello!");
…
free(str);     // our responsibility to free!
```

Free

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

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
```

Recommendation: Don't worry about putting in frees until **after** you're finished with functionality.

Memory leaks will rarely crash your CS107 programs.

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 an extremely 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!

Lecture Plan

- The Stack 3
- The Heap and Dynamic Memory 59
- **`realloc`** 83
- Use After Free 99


```
cp -r /afs/ir/class/cs107/lecture-code/lect07 .
```

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 allocator analogy: A hotel

Request memory by size (`malloc`)
- Receive room key to first of connecting rooms

Need more room? (`realloc`)
- Extend into connecting room if available
- If not, trade for new digs, employee moves your stuff for you

Check out when done (`free`)
- You remember your room number though

Errors! What happens if you…
- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes…
- Return to room after checkout?

Heap Allocation: Error prone

Invalid usage of heap functions includes when you:

- Realloc a stack pointer
- Free the middle of an allocation
- Free a stack pointer
- Etc….

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 when done

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

Goodbye, Free Memory

Where/how should we free memory below so that all memory is freed properly?

```
1  char *str = strdup("Hello");
2  assert(str != NULL);
3  char *ptr = str + 1;
4  for (int i = 0; i < 5; i++) {
5    int *num = malloc(sizeof(int));
6    *num = i;
7    printf("%s %d\n", ptr, *num);
8    free(num);
9  }
10 printf("%s\n", str);
11 free(str);
   //Leak Check: valgrind --leak-check=full --show-leak-kinds=all …
```

Recommendation: Don't worry about putting in frees until **after** you're finished with functionality.

Memory leaks will rarely crash your CS107 programs.

strcat_extend

Write a function that takes in a heap-allocated **str1,** enlarges it, and concatenates **str2** onto it.

```
1 char *strcat_extend(char *heap_str, const char *concat_str) {
2     (_____________________(1)_____________________);
3     heap_str = realloc(___(2A)___,___(2B)___);
4     (_____________________(3)_____________________);
5     strcat(___(3A)___, ___(3B)___);
6     return heapstr;
7 }
```

Example usage:

```
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```

strcat_extend

Write a function that takes in a heap-allocated **str1,** enlarges it, and concatenates **str2** onto it.

```
1 char *strcat_extend(char *heap_str, const char *concat_str) {
2     int new_length = strlen(heap_str) + strlen(concat_str) + 1;
3     heap_str = realloc(heap_str, new_length);
4     assert(heap_str != NULL);
5     strcat(heap_str, concat_str);
6     return heapstr;
7 }
```

Example usage:

```
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", 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);
```

Compare and contrast the heap memory functions we've learned about.

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 ("local variables")

- **Fast**
  Fast to allocate/deallocate
- **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 ("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 lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

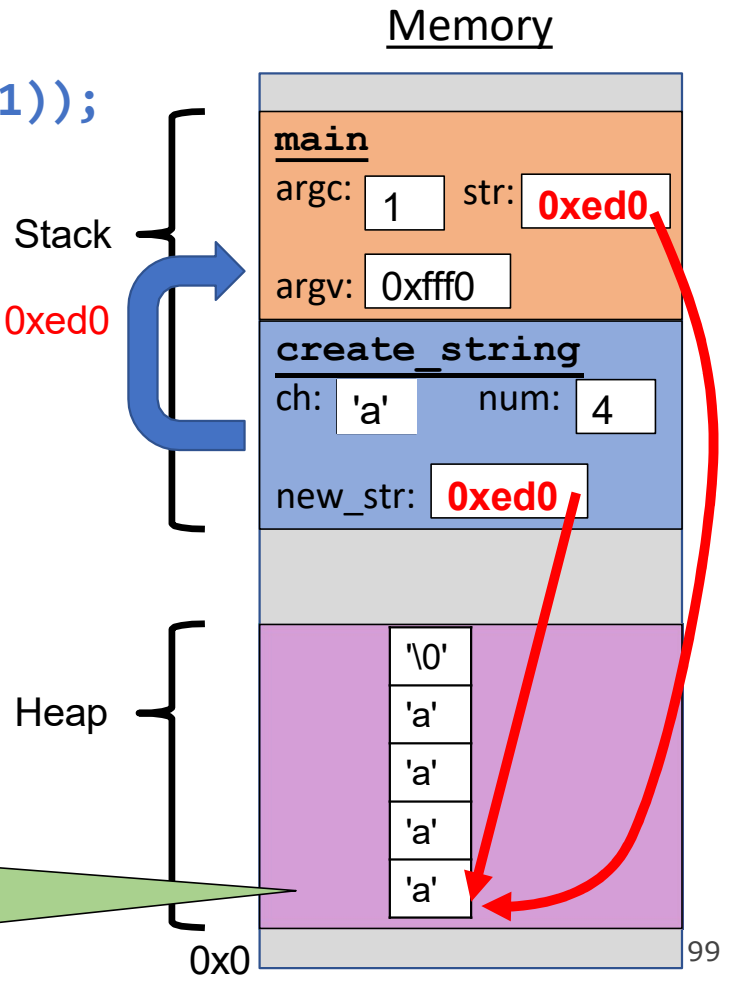
Lecture Plan

- The Stack 3
- The Heap and Dynamic Memory 59
- `realloc` 83
- **Use After Free** 99

```
cp -r /afs/ir/class/cs107/lecture-code/lect07 .
```

Recall: 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;
}
```



C: Nice memory you got there ... would be a shame if someone tried to use it after you freed it.

Use after Free

What happens if we attempt to reference something on the heap after it has been freed?

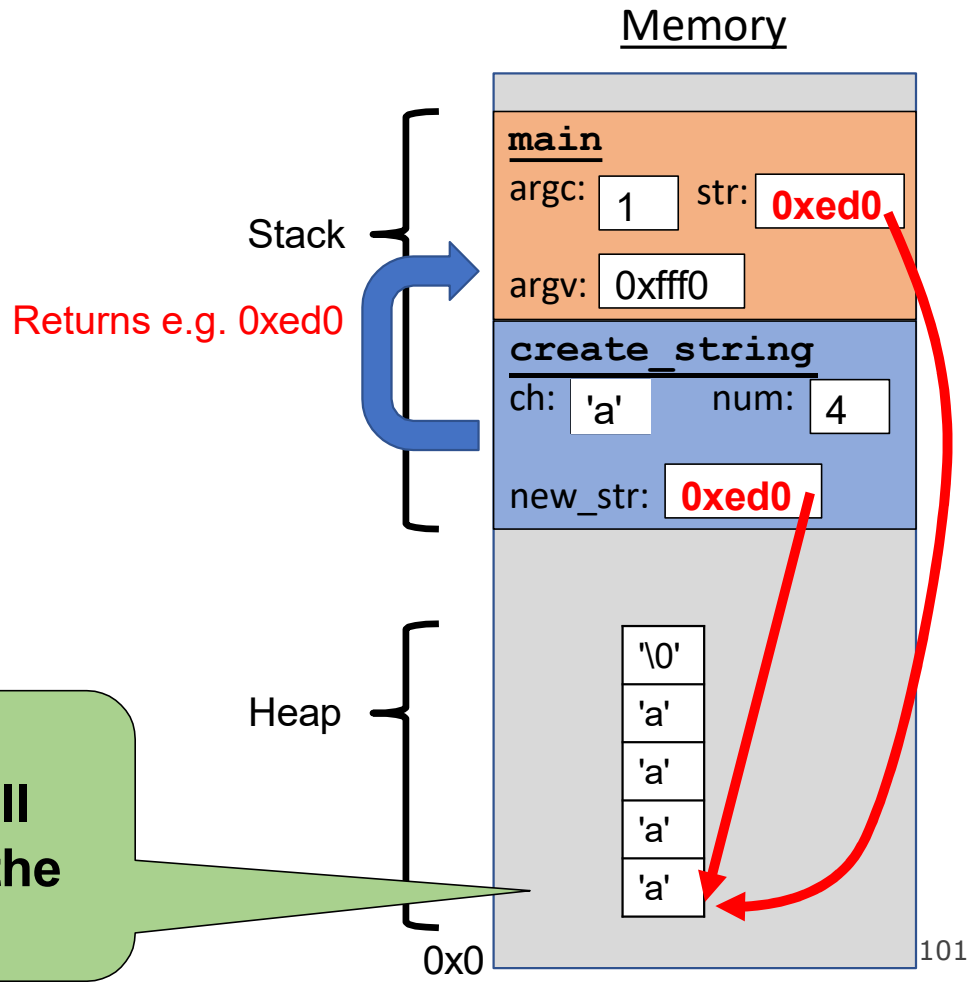
```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);
…
strncpy(ptr, argv[1], 3);
```

We freed bytes but did not set ptr to NULL

Memory at this address was already freed, but now we are using it!

What Could Happen?

- Each program has its own memory space (heap + stack)
- Vulnerabilities are within the context of a program.



The memory has been freed, but the pointer still reaches its location on the heap.

“undefined behavior occurs”

Using a pointer to the heap after the pointer’s memory has been freed results in undefined behavior because the memory could have its original contents or could have been overwritten.

This undefined behavior also creates a vulnerability and an opportunity to hack.

Use After Free As A Vulnerability

Image is of the CVE database listing of “use after free” vulnerabilities, which can be found at http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=use+after+free

What Should I Do If I Find a Vulnerability?

Responsible Disclosure

In a Responsible Disclosure process, the finder of the vulnerability:

- Contacts the makers of the software
- Informs them about the vulnerability
- Negotiates a reasonable timeline for a patch or fix
- Considers a deadline extension if necessary

time passes while the developers fix the bug

- Work with the developers to add the vulnerability to CVE Details (https://www.cvedetails.com/), from which it is added to the National Vulnerability Database (https://nvd.nist.gov/)

Full Disclosure

In a Full Disclosure process:

- The finder of a vulnerability releases it publicly, fully, and immediately.
- Vulnerabilities unknown to developers before release are sometimes called "0-days," because they have zero days to fix the problem.

Why would someone do this?

This approach puts pressure on the developers to fix the vulnerability quickly, and informs the public immediately about their exposure. But it also leaves the users of the software vulnerable until they receive the patch. Few people now endorse this approach for this reason.

Responsible Disclosure

Responsible disclosure is the most common approach, and it is recommended by the ACM code of ethics:

Responsible disclosure is the approach more consistent with the ACM Code of Ethics. By keeping the existence of the vulnerability secret for a longer amount of time, it reduces the chance of harm to others (Principle 1.2). It also supports more robust patching (Principles 2.1, 2.9, and 3.6), as the company can take more time to develop the patch and confirm that it will not induce unintended consequences. Full disclosure puts individuals at risk of harm sooner, and those harms may be irreversible and onerous (contravening Principles 1.2 and 3.1). As such, full disclosure should the exception and should only be used when attempts at responsible disclosure have failed. Furthermore, the individual committing to the full disclosure needs to consider carefully the risks that they are imposing on others and be willing to accept the moral and possibly legal consequences (Principles 2.3 and 2.5).

Vulnerability Commercialization

Many companies now offer “Bug Bounties,”
or rewards for responsible disclosure.

Image: Two Burlap Sacks Filled with Gold Coins

Good Version of a bug bounty process:

- Responsible disclosure process is followed
- Company is buying information & time to fix the bug

Bad version of a bug bounty process:

- Company does not fix the bug *or* notify the public.
- Not knowing what vulnerabilities exist makes it harder for users to calibrate trust
- Company is effectively buying silence

Vulnerabilities Equities Process

The US federal government is one of the largest discoverers and purchasers of 0-day vulnerabilities.

It follows a “Vulnerabilities Equities Process” (VEP) to determine which vulnerabilities to responsibly disclose and which to keep secret and use for espionage or intelligence gathering.

VEP claimed in 2017 that 90% of vulnerabilities are disclosed, but it is not clear what the impact or scope of the un-disclosed 10% of vulnerabilities are.