

# COMP201

## Computer Systems & Programming

Lecture #11 – The Stack, The Heap and Dynamic Memory



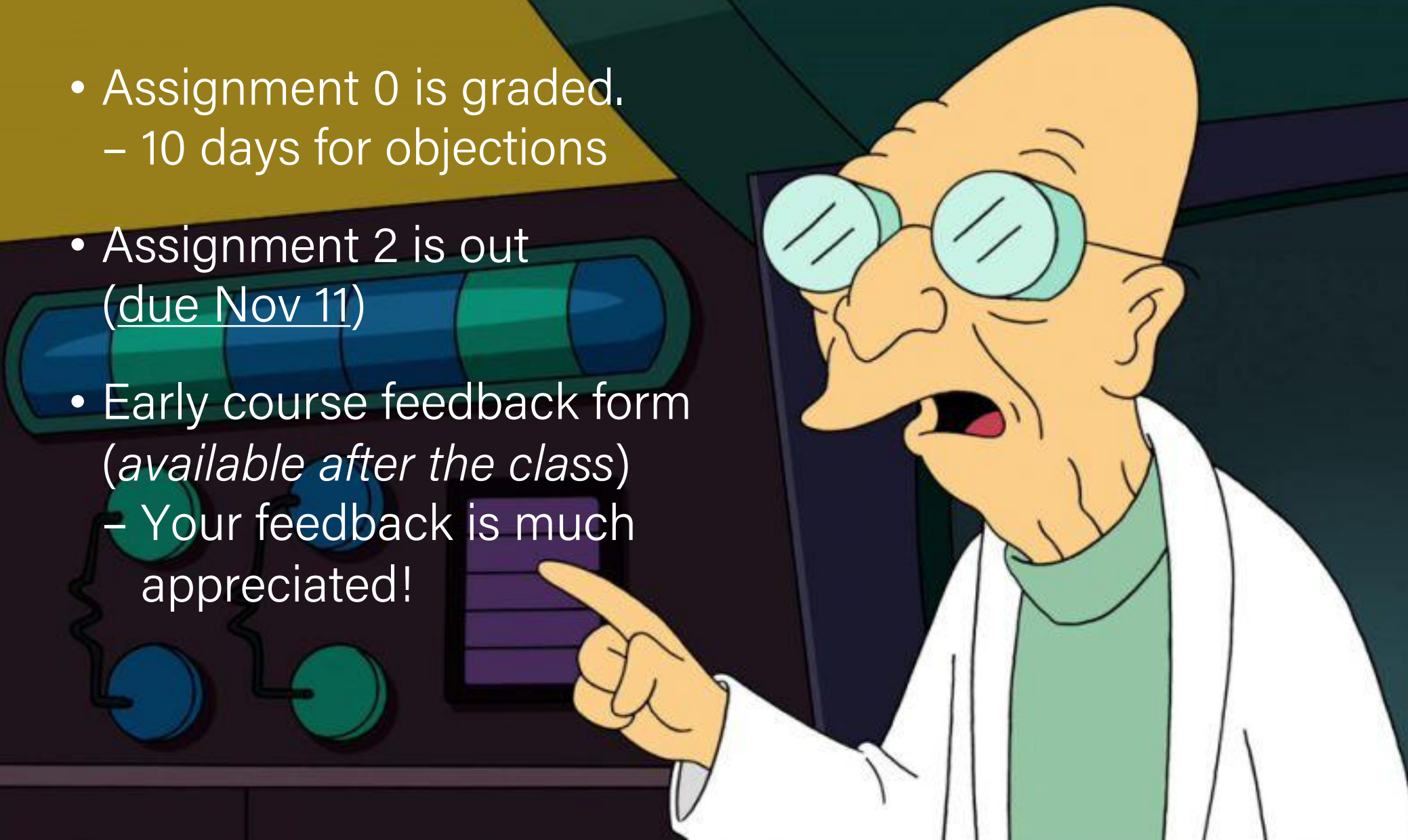
KOÇ  
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020



# Good news, everyone!

- Assignment 0 is graded.
  - 10 days for objections
- Assignment 2 is out  
(due Nov 11)
- Early course feedback form  
(*available after the class*)
  - Your feedback is much appreciated!



# Recap

- Pointers and Parameters
- Double Pointers
- Arrays in Memory
- Arrays of Pointers

# Pointers Practice

# \* Wars: Episode I (of 2)

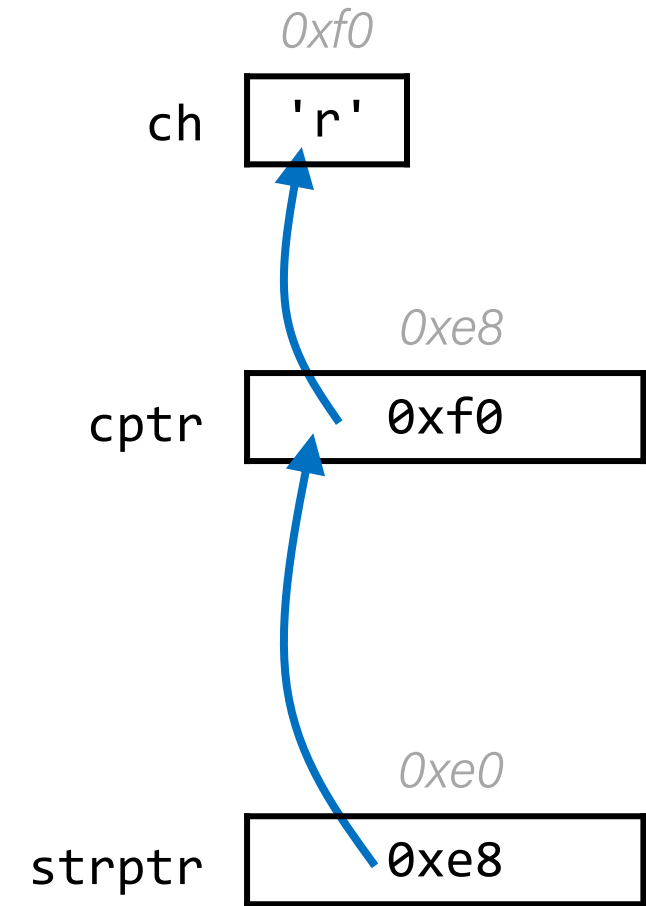
Review

In variable declaration, \* creates a **pointer**.

`char ch = 'r';`                      ch stores a char

`char *_cptr = &ch;`                      cptr stores an address  
of a char  
(**points to** a char)

`char **_strptr = &cptr;`                      strptr stores an address  
of a char \*  
(**points to** a char \*)



# \* Wars: Episode II (of 2)

Review

In reading values from/storing values, \* dereferences a pointer.

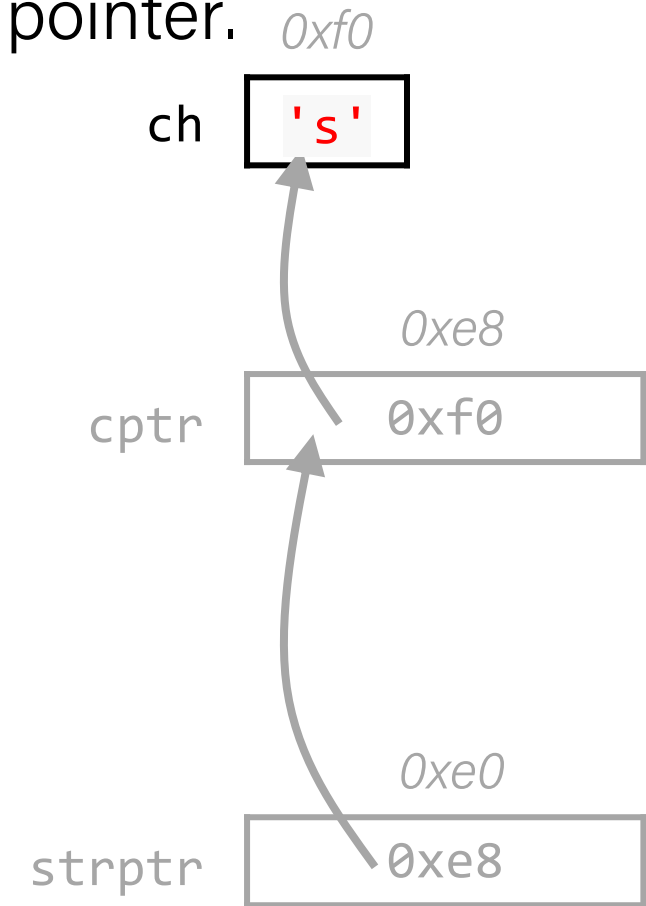
```
char ch = 'r';
```

```
ch = ch + 1;
```

```
char *cptr = &ch;
```

```
char **strptr = &cptr;
```

Increment value stored in ch



# \* Wars: Episode II (of 2)

Review

In reading values from/storing values, \* dereferences a pointer.

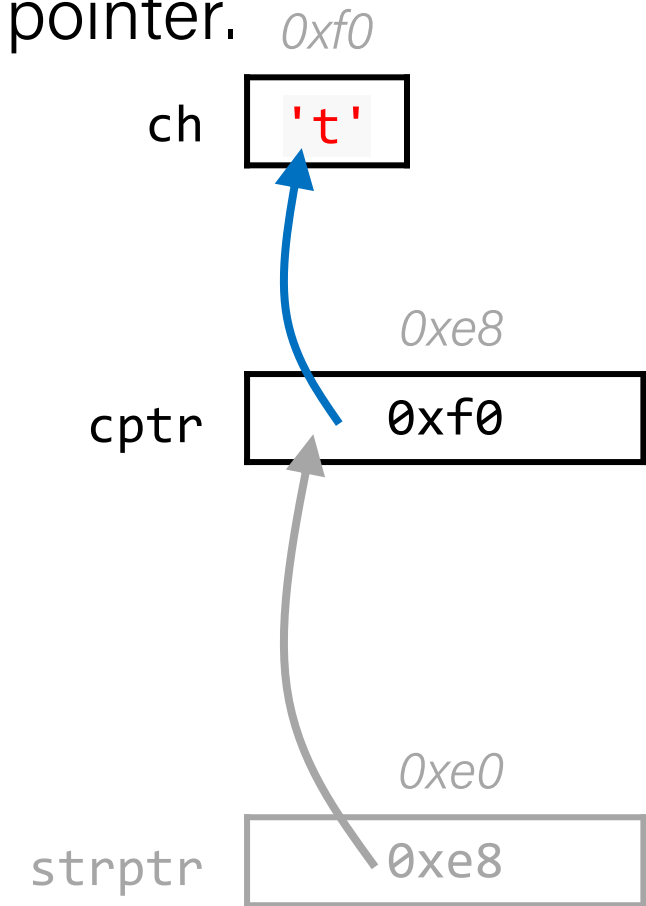
```
char ch = 'r';  
ch = ch + 1;
```

Increment value stored in ch

```
char *cptr = &ch;  
*cptr = *cptr + 1;
```

Increment value stored at  
memory address in cptr  
(increment char **pointed to**)

```
char **strptr = &cptr;
```



# \* Wars: Episode II (of 2)

Review

In reading values from/storing values, \* dereferences a pointer.

```
char ch = 'r';  
ch = ch + 1;
```

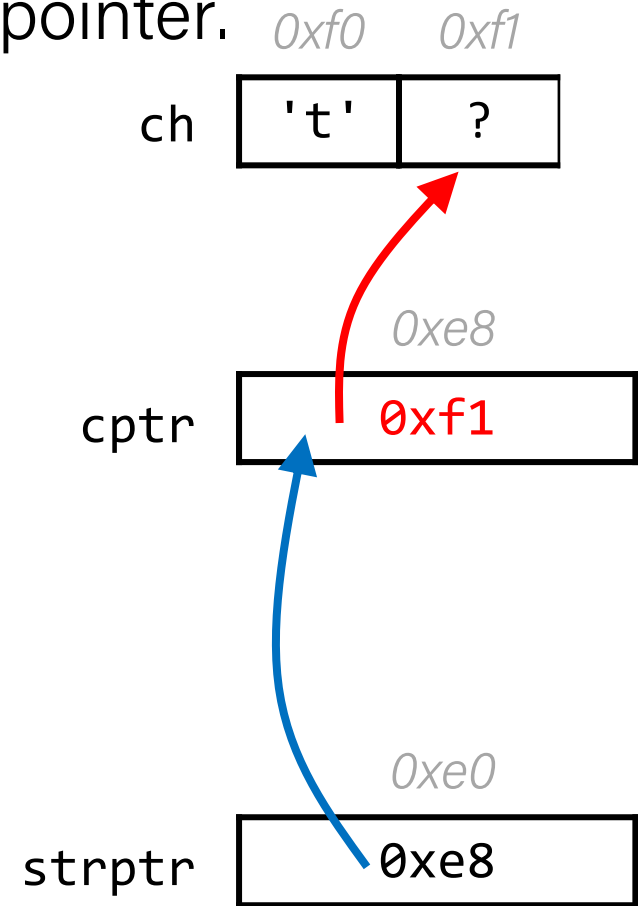
Increment value stored in ch

```
char *cptr = &ch;  
*cptr = *cptr + 1;
```

Increment value stored at  
memory address in cptr  
(increment char **pointed to**)

```
char **_struptr = &cptr;  
*_struptr = *_struptr + 1;
```

Increment value stored at  
memory address in cptr  
(increment address **pointed to**)

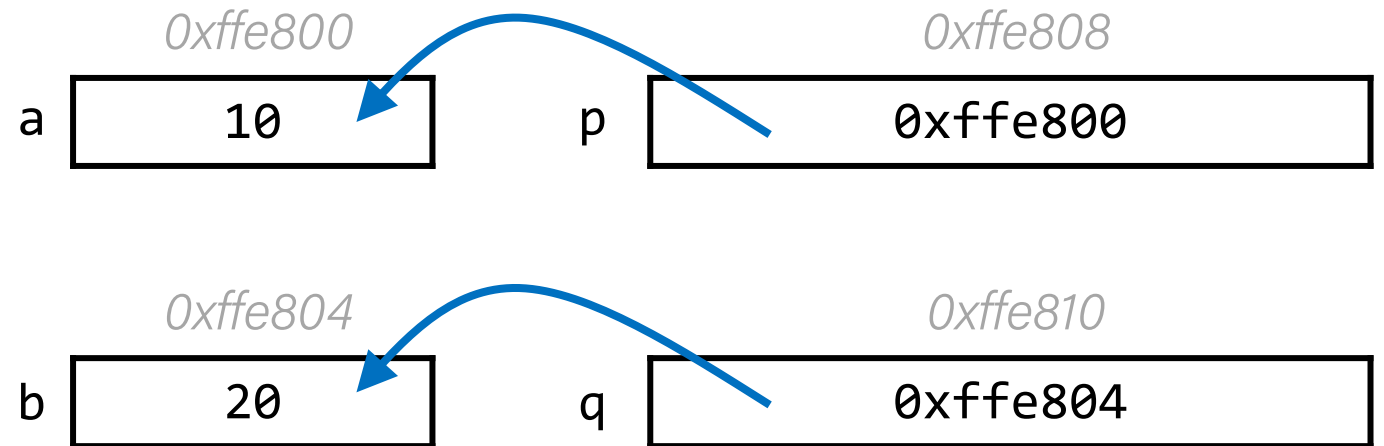




# Pen and paper: A \* Wars Story

```
1 void binky() {  
2     int a = 10;  
3     int b = 20;  
4     int *p = &a;  
5     int *q = &b;  
6  
7     *p = *q;  
8     p = q;  
9 }
```

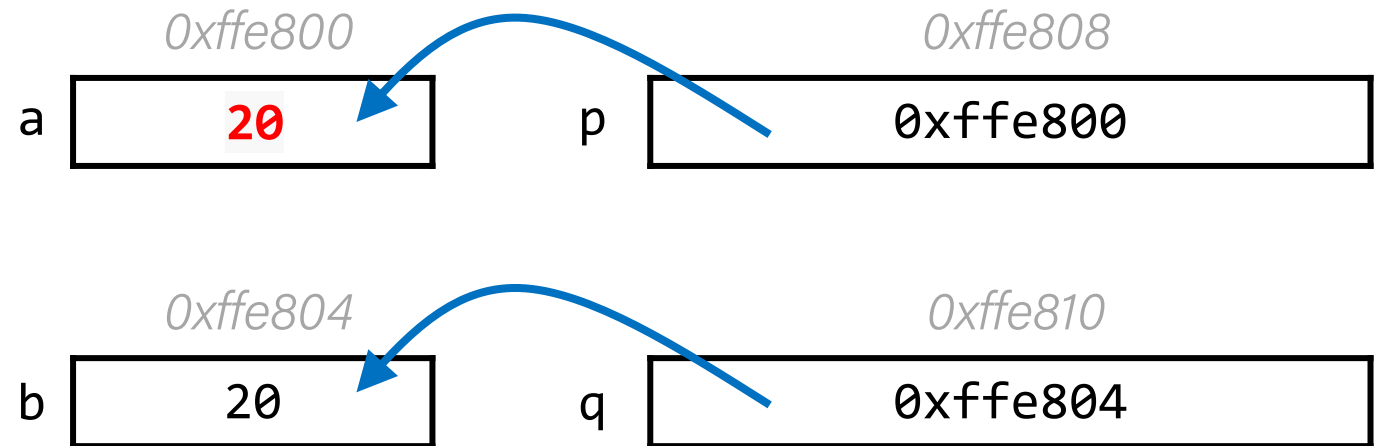
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



# Pen and paper: A \* Wars Story

```
1 void binky() {  
2     int a = 10;  
3     int b = 20;  
4     int *p = &a;  
5     int *q = &b;  
6  
7     *p = *q;  
8     p = q;  
9 }
```

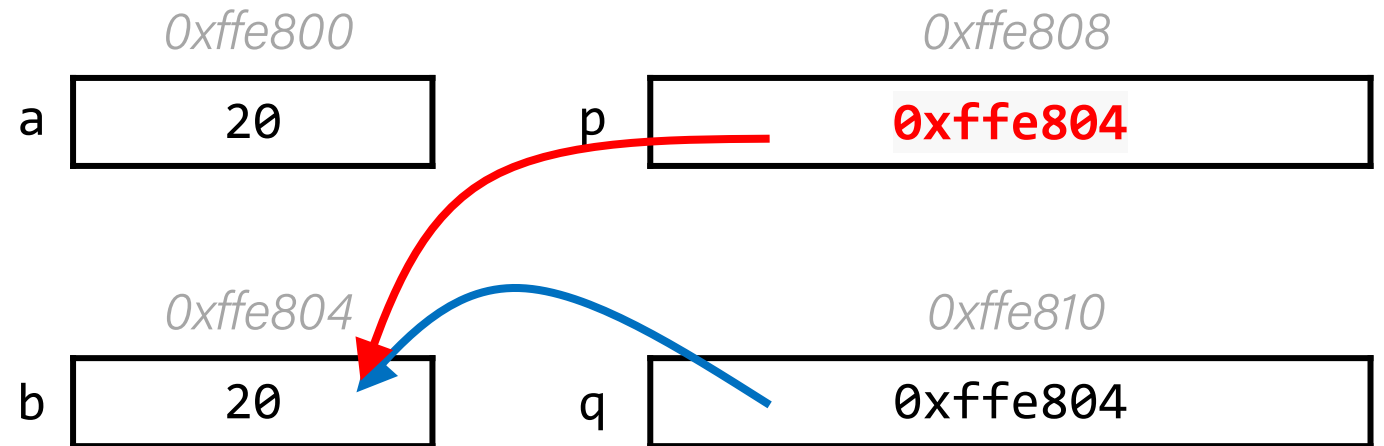
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



# Pen and paper: A \* Wars Story

```
1 void binky() {  
2     int a = 10;  
3     int b = 20;  
4     int *p = &a;  
5     int *q = &b;  
6  
7     *p = *q;  
8     p = q;  
9 }
```

- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



# Plan for Today

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory
- `realloc`

**Disclaimer:** Slides for this lecture were borrowed from  
—Nick Troccoli's Stanford CS107 class

# Lecture Plan

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory
- `realloc`



# 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*.

```
// nums points to an int array
int *nums = ...           // 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

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 two places to str,  
// and then dereference to get the char there.  
// E.g. get memory at 0xff2.
```

```
char thirdLetter = str[2];           // 'p'
```

```
char thirdLetter = *(str + 2);       // 'p'
```

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 gives the number of places they differ by.

```
// nums points to an int array
int *nums = ...           // e.g. 0xff0
int *nums3 = nums + 3;    // e.g. 0xffc
int diff = nums3 - nums;  // 3
```

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

**String Behavior #6:** Adding an offset to a C string gives us a substring that many places past the first character.



# Pointer Arithmetic

How does the code know how many bytes it should look at once it visits an address?

```
int x = 2;  
int *xPtr = &x;           // e.g. 0xff0  
  
// How does it know to print out just the 4 bytes at xPtr?  
printf("%d", *xPtr);      // 2
```

# Pointer Arithmetic

How does the code know how many bytes it should add when performing pointer arithmetic?

```
int nums[] = {1, 2, 3};
```

```
// How does it know to add 4 bytes here?
```

```
int *intPtr = nums + 1;
```

```
char str[6];
```

```
strcpy(str, "COMP201");
```

```
// How does it know to add 1 byte here?
```

```
char *charPtr = str + 1;
```

# Pointer Arithmetic

- At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.
- For this reason, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type.

# Pointer arithmetic

Array indexing is “syntactic sugar” for pointer arithmetic:

<code>ptr + i</code>	$\Leftrightarrow$	<code>&amp;ptr[i]</code>
<code>*(ptr + i)</code>	$\Leftrightarrow$	<code>ptr[i]</code>

⚠ Pointer arithmetic **does not work in bytes**; it works on the type it points to. On `int*` addresses scale by `sizeof(int)`, on `char*` scale by `sizeof(char)`.

- This means too-large/negative subscripts will compile 😊

`arr[99]`

`arr[-1]`

- You can use either syntax on either pointer or array.

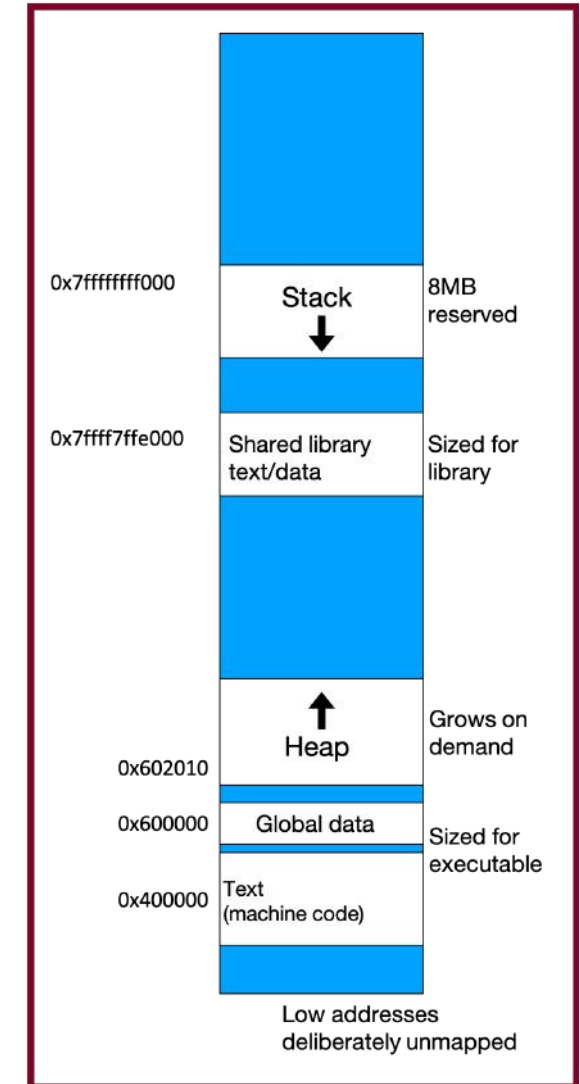
# Lecture Plan

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory
- `realloc`



# 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.

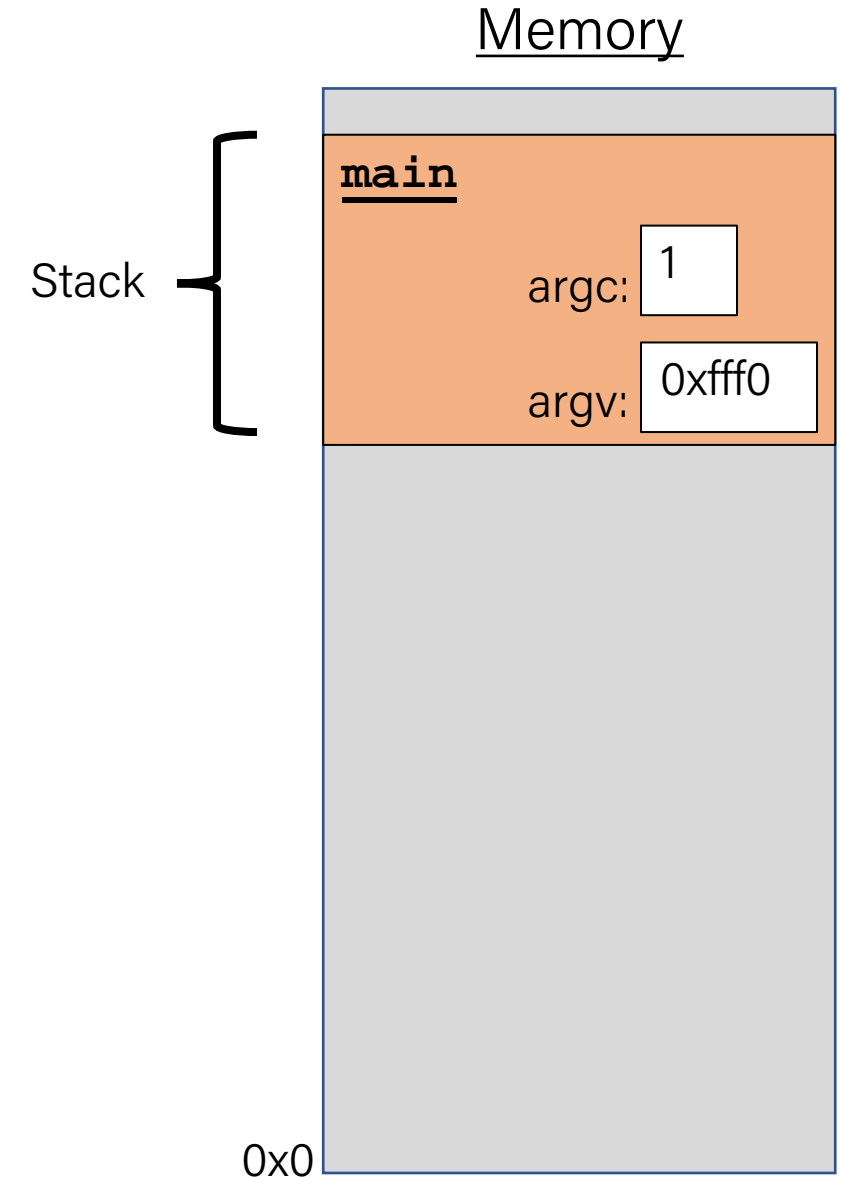


# The Stack

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

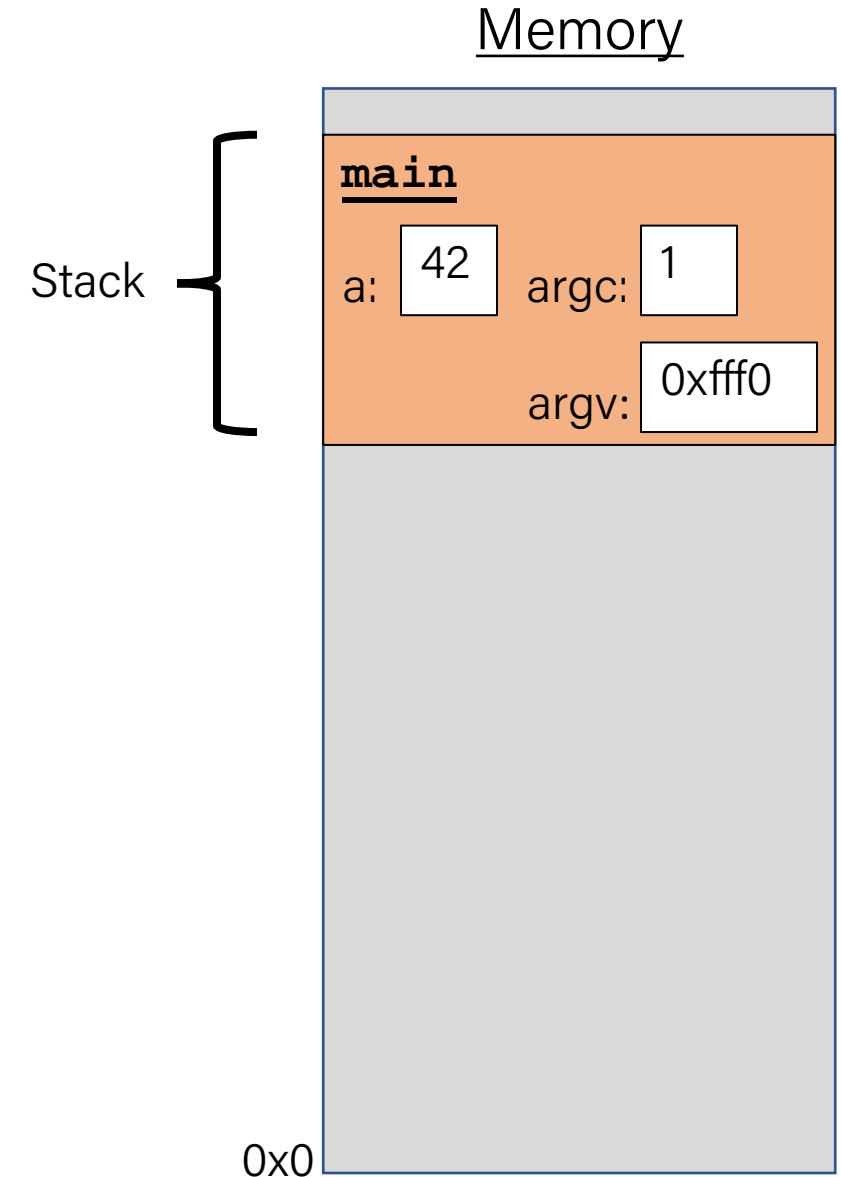
```
void func1() {  
    int c = 99;  
    func2();  
}
```

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



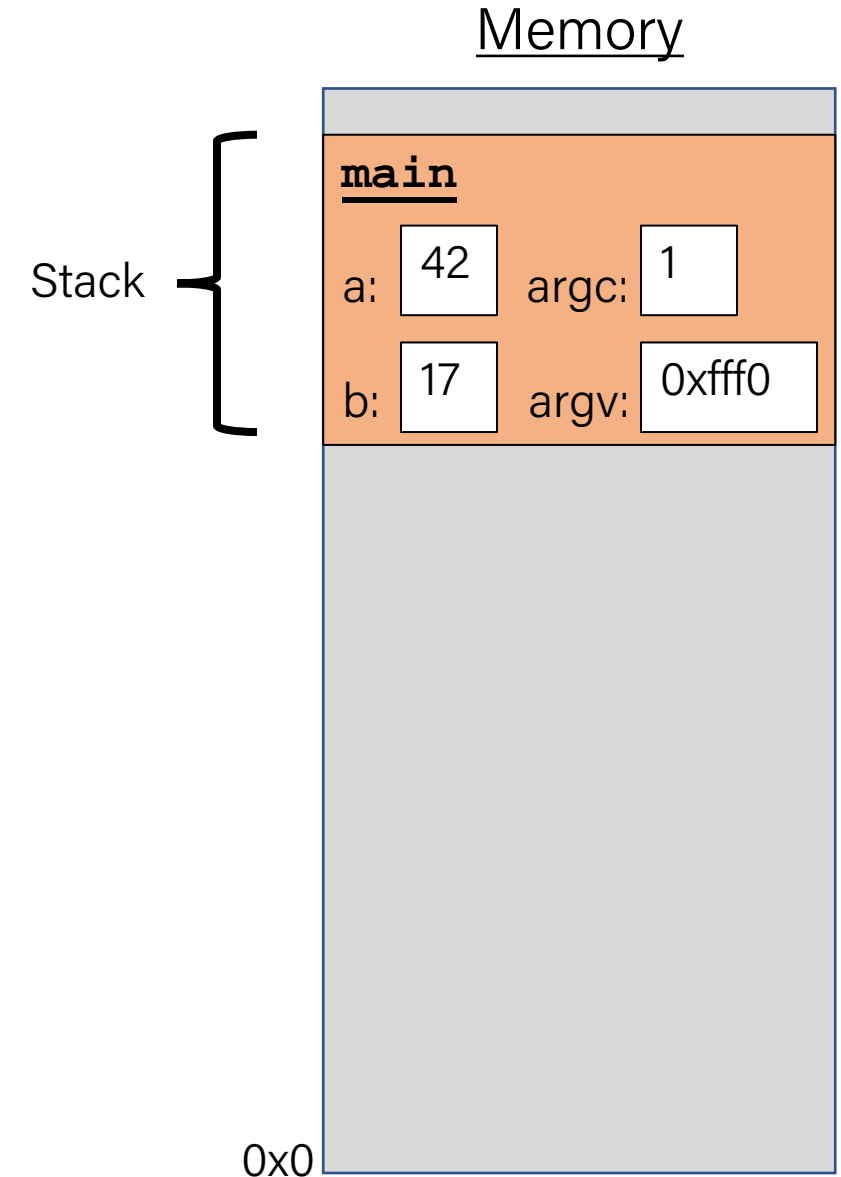
# The Stack

```
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;  
}
```



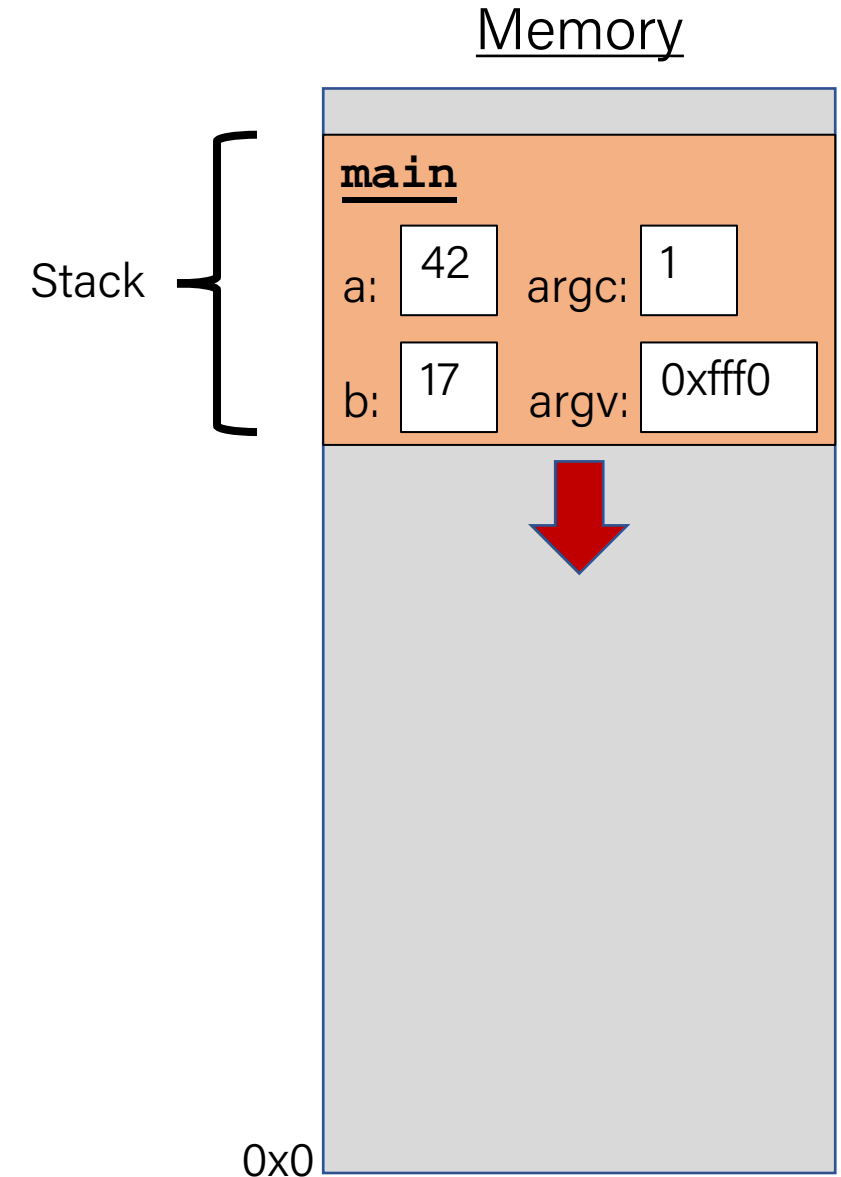
# The Stack

```
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;  
}
```



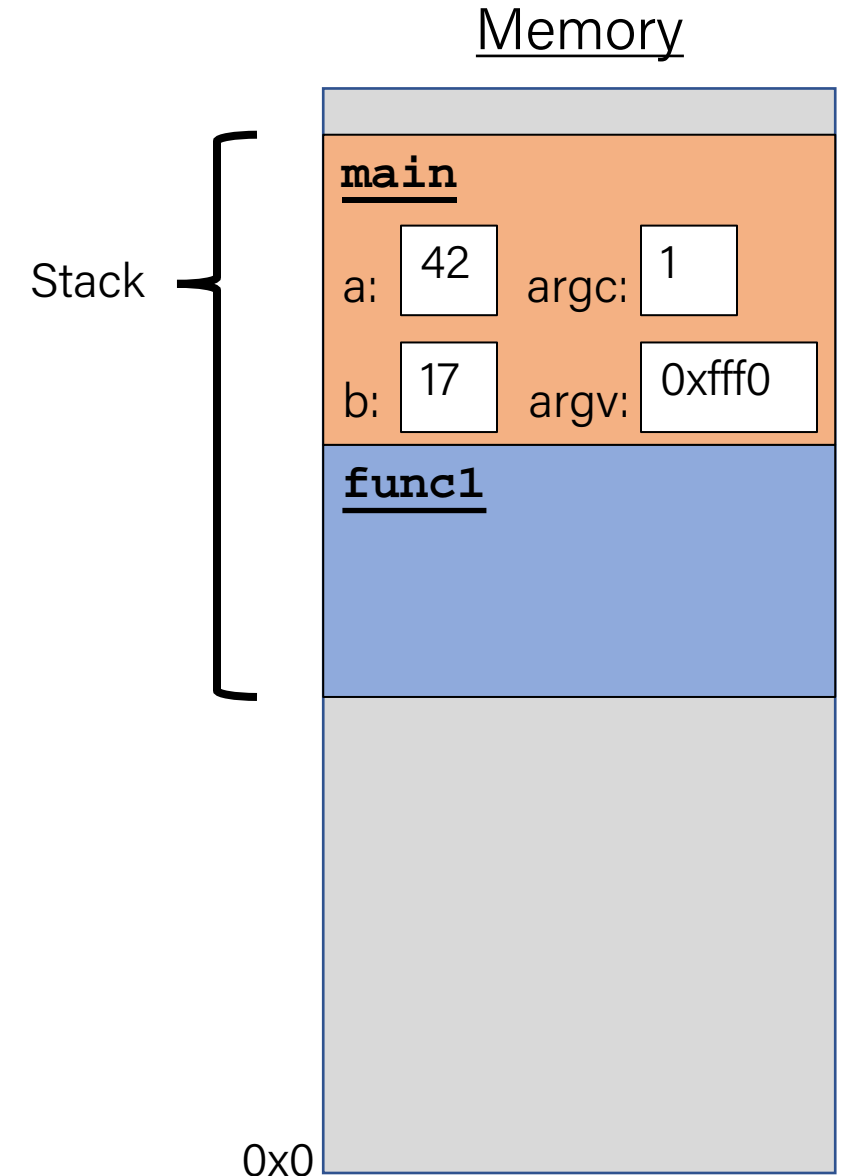


# The Stack

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

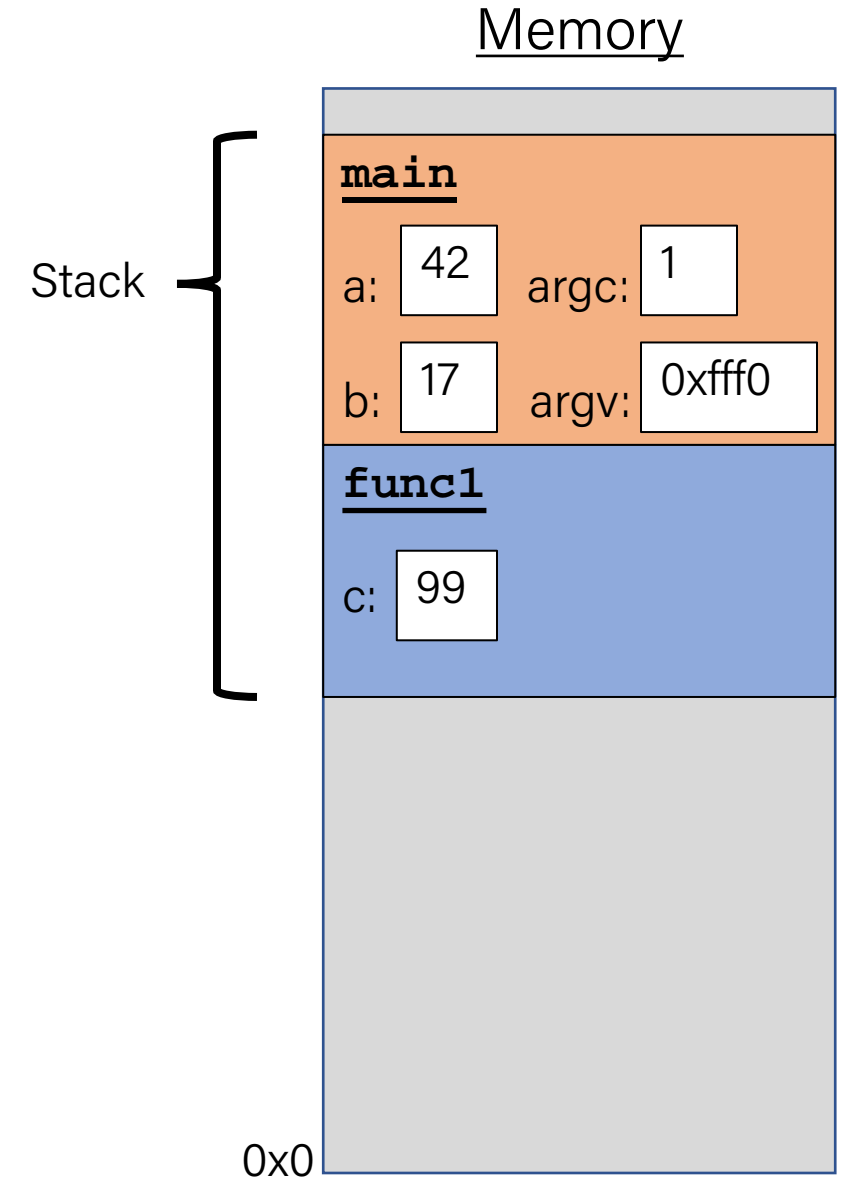
```
void func1() {  
    int c = 99;  
    func2();  
}
```

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



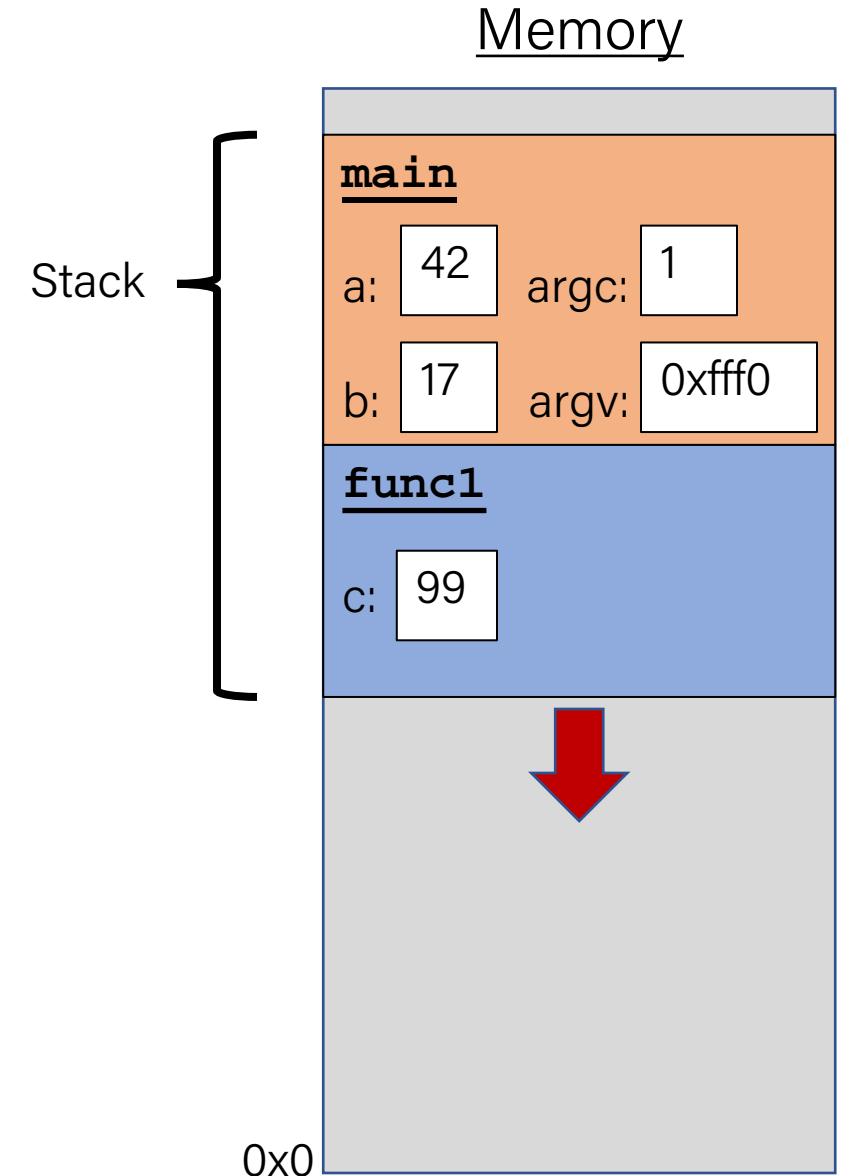
# The Stack

```
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;  
}
```



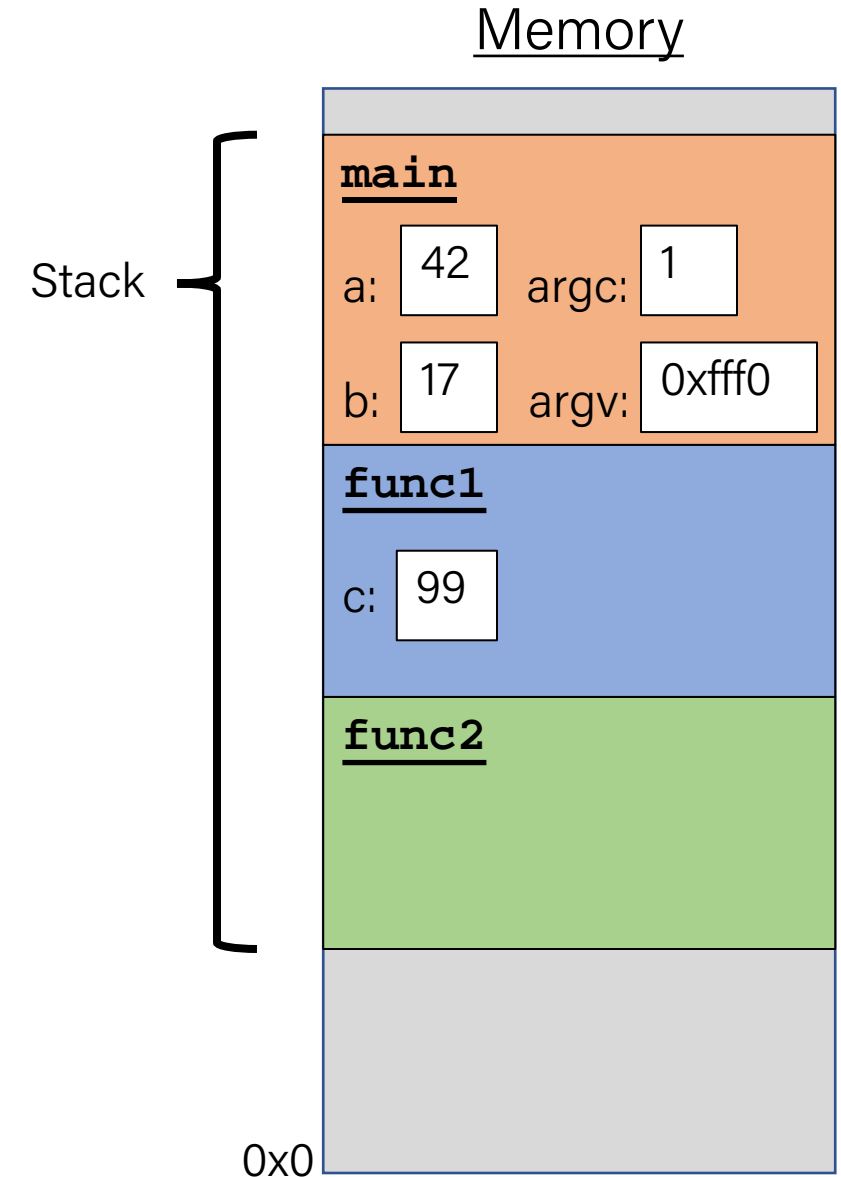
# The Stack

```
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;  
}
```

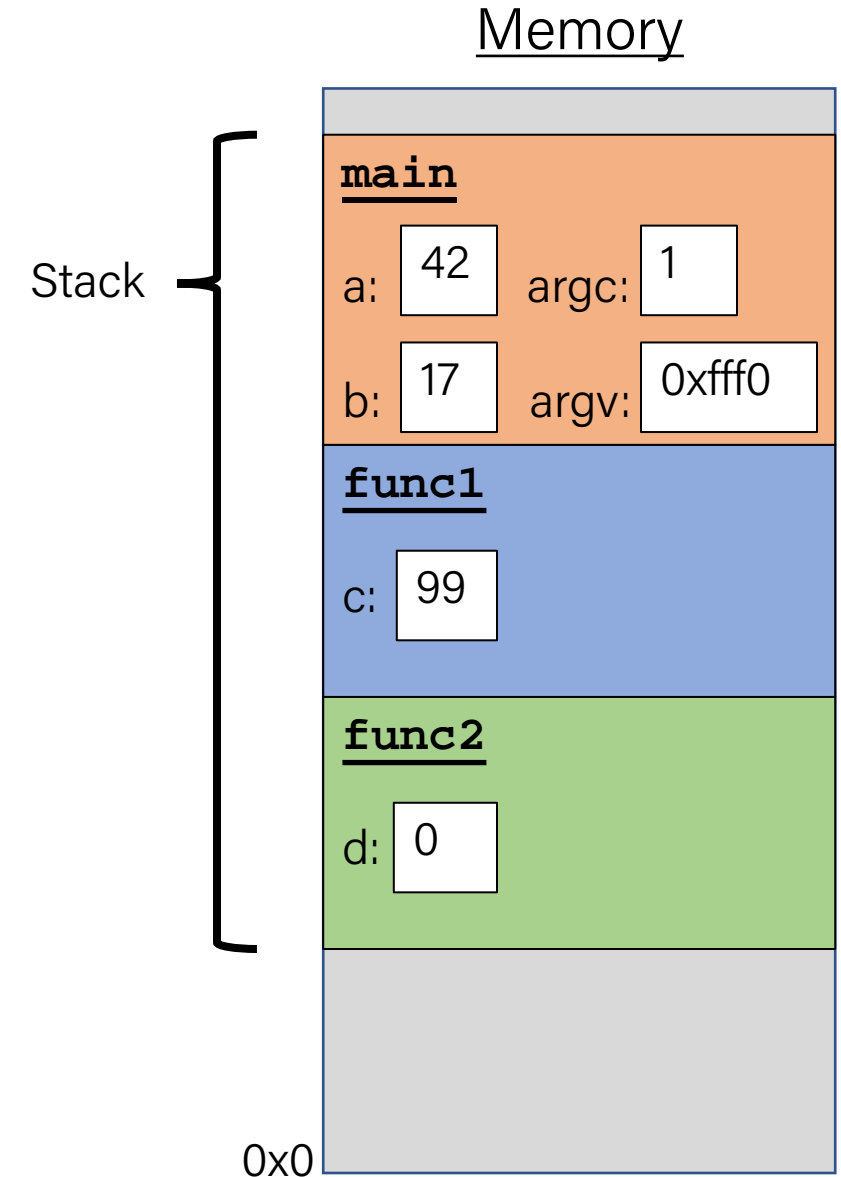


# The Stack

```
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;  
}
```

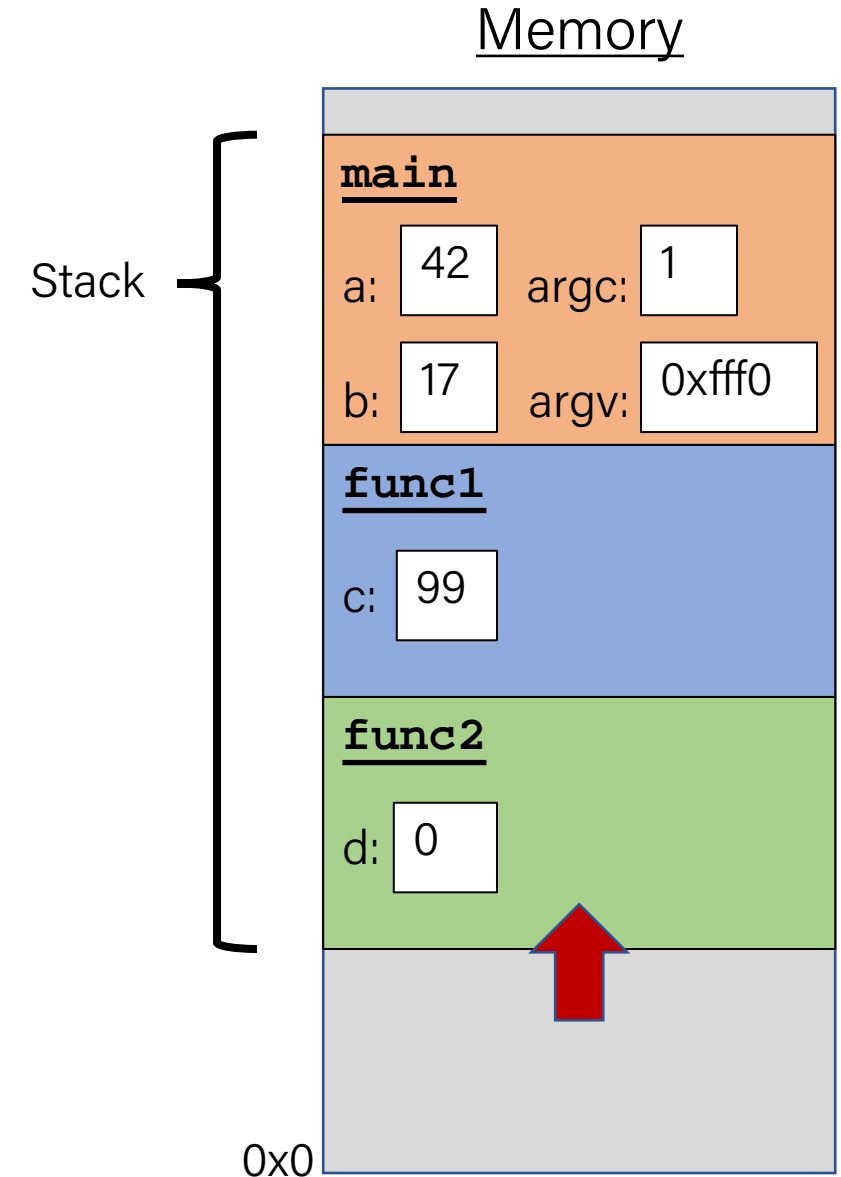


# The Stack

```
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;  
}
```

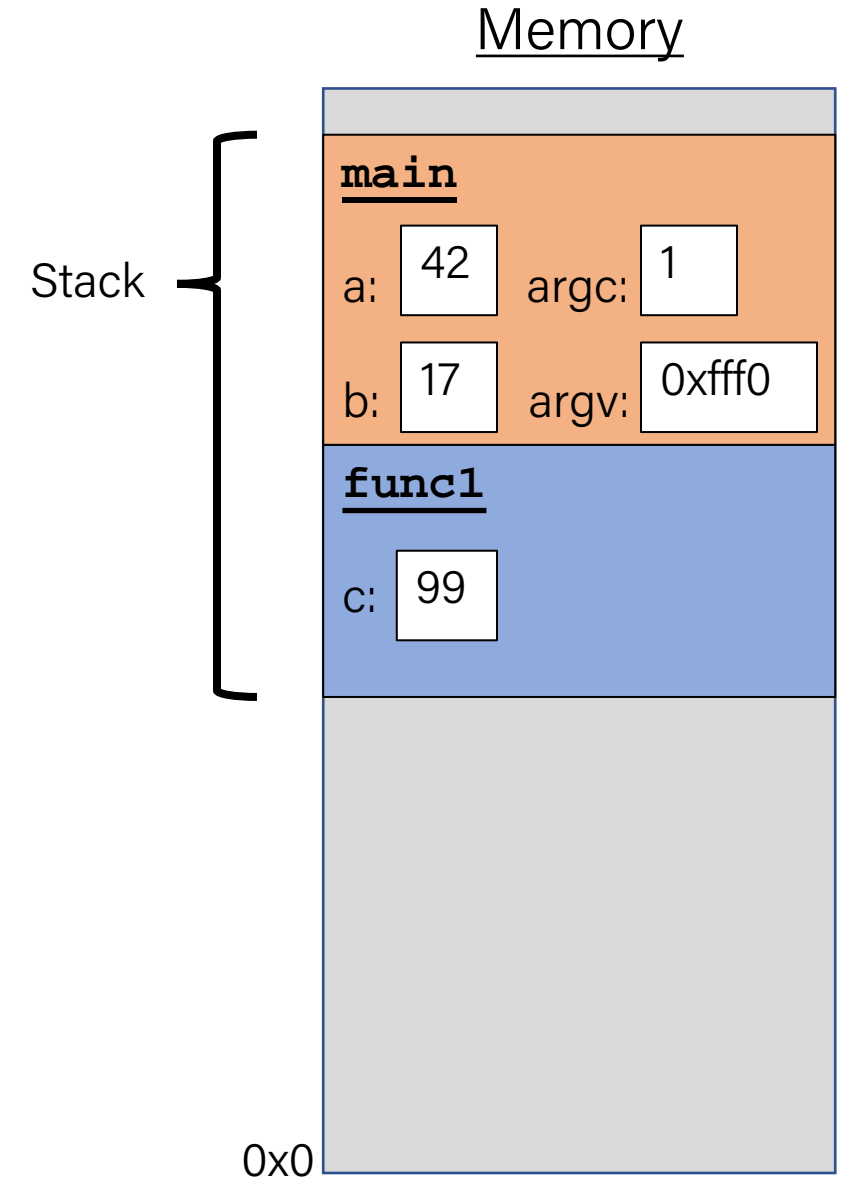


# The Stack

```
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;  
}
```

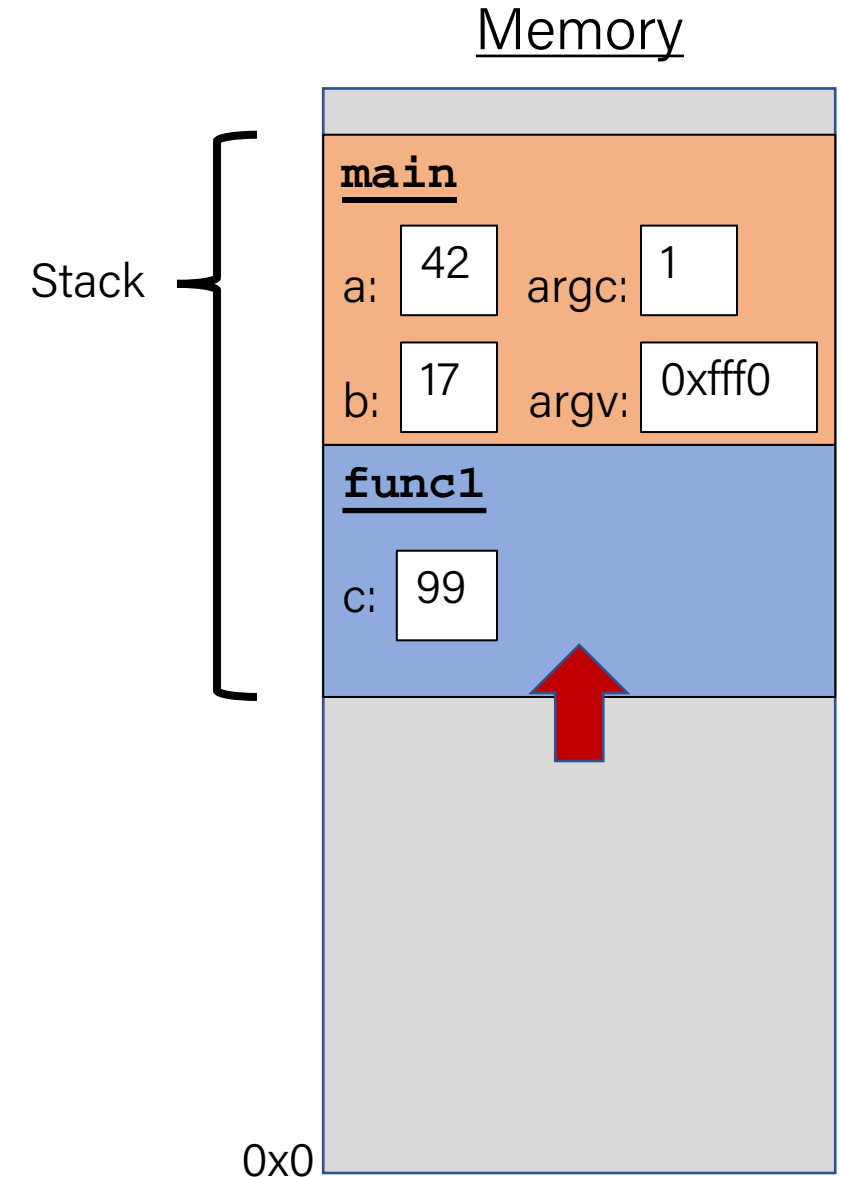


# The Stack

```
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;  
}
```



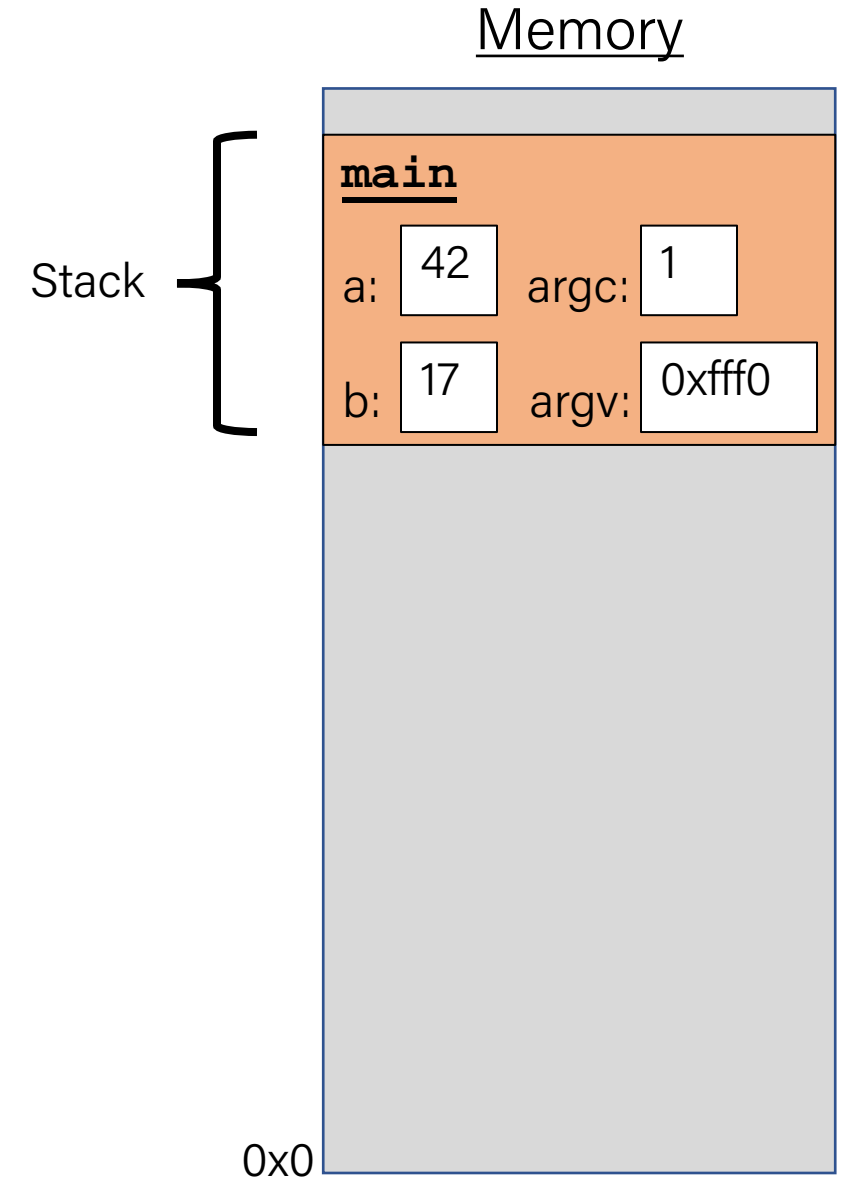


# The Stack

```
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;  
}
```

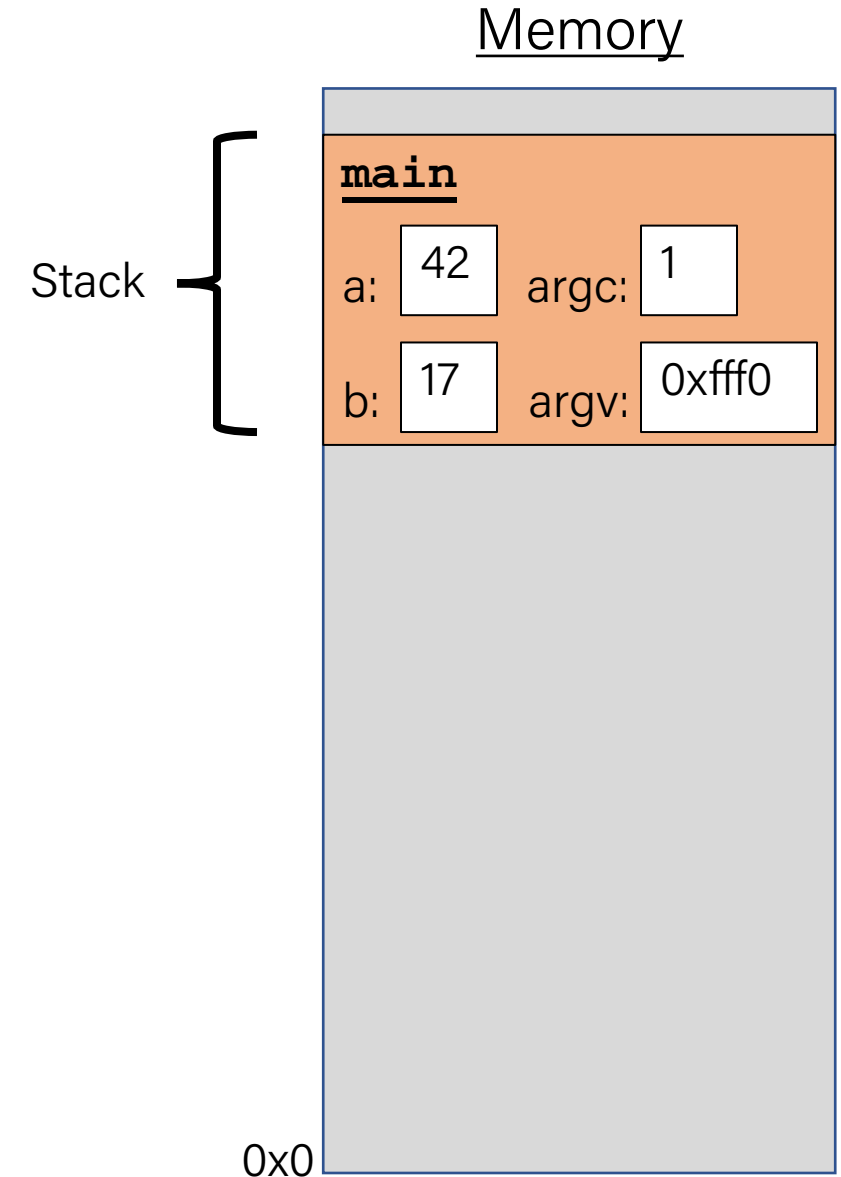


# The Stack

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

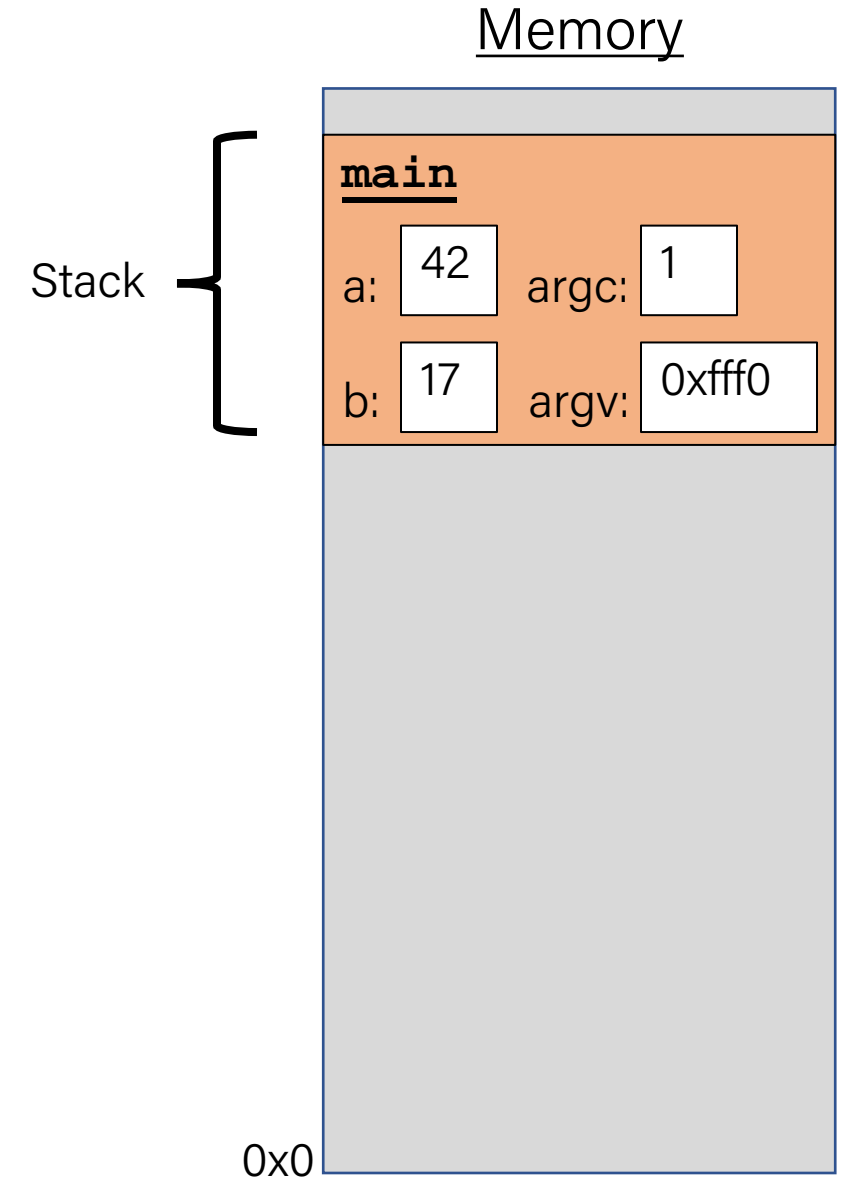
```
void func1() {  
    int c = 99;  
    func2();  
}
```

```
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;  
}
```

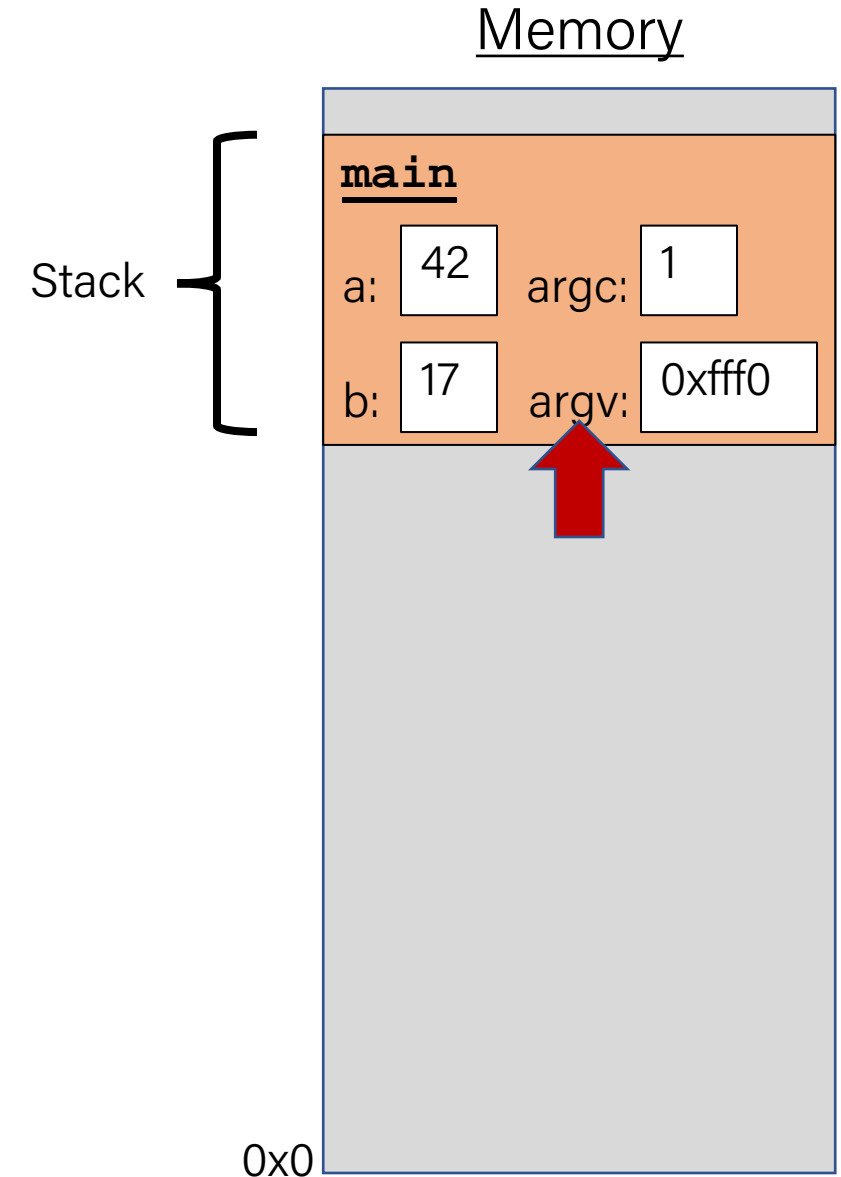


# The Stack

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

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

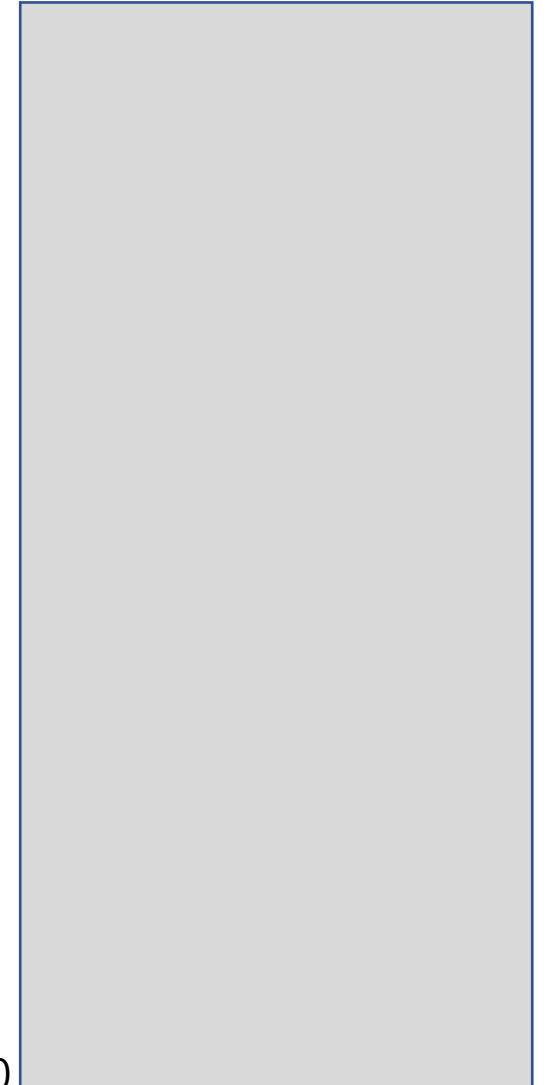
```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



# 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;  
}
```

Memory



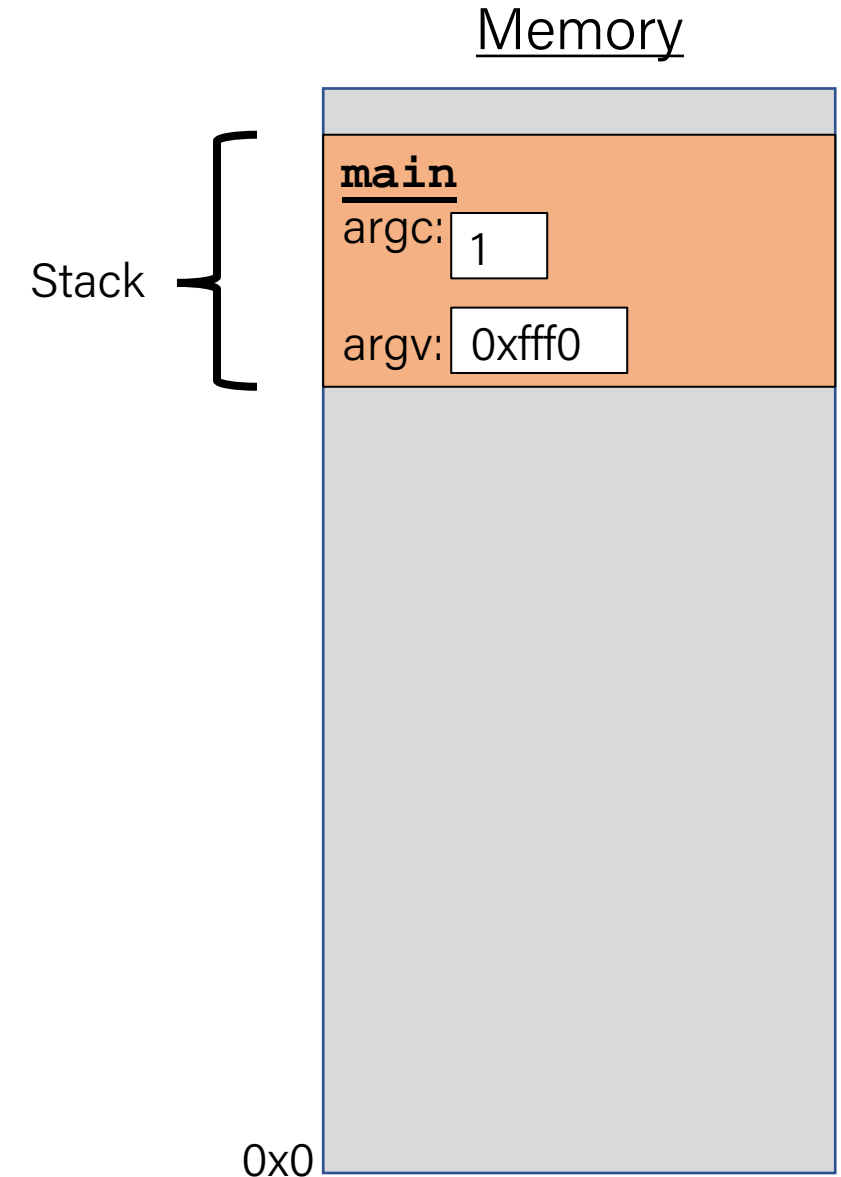
0x0

# 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;  
}
```

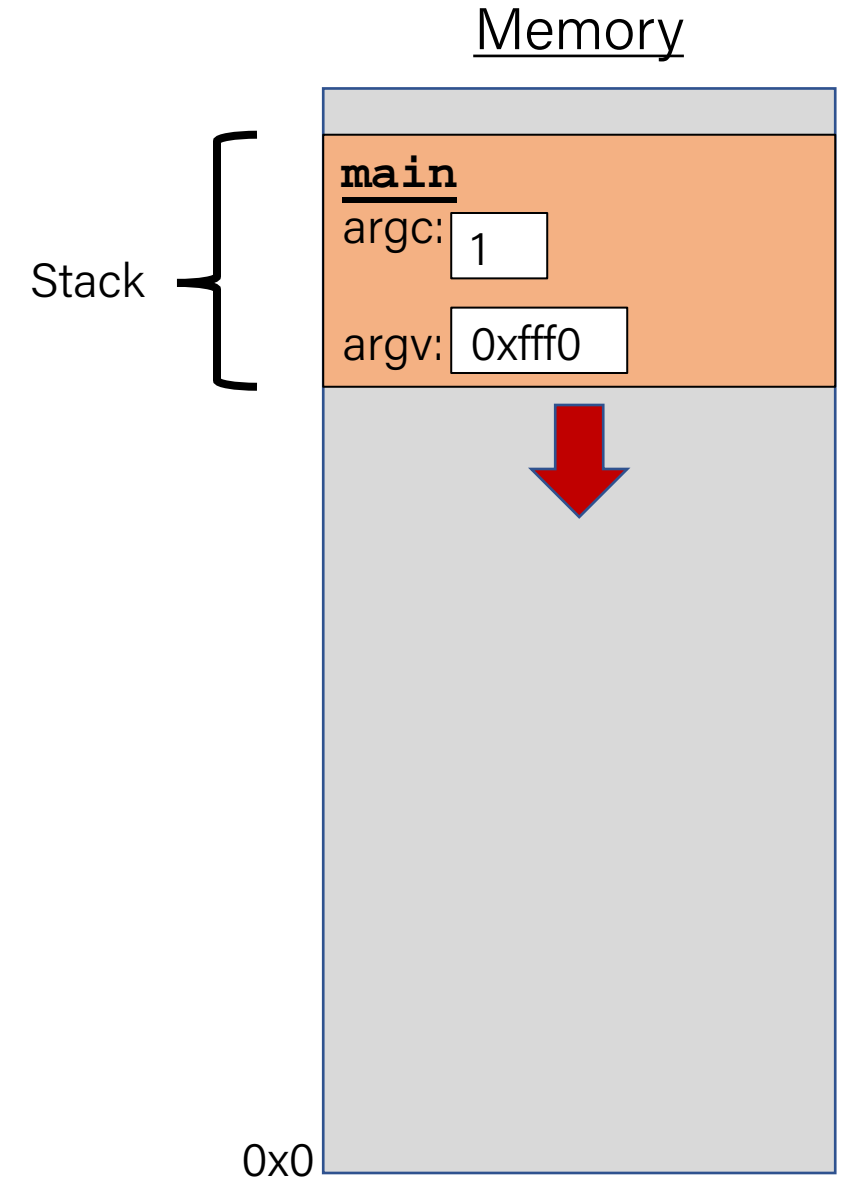


# The Stack

- 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;  
}
```

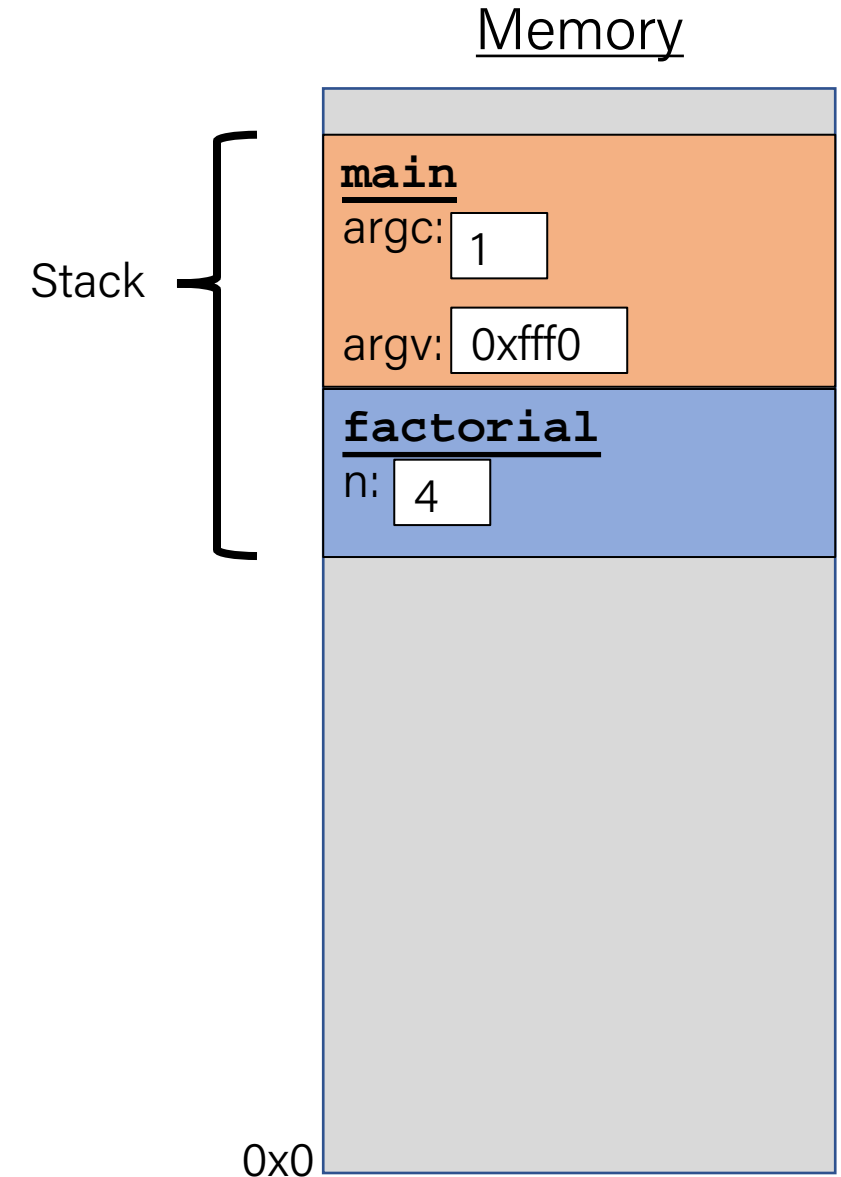


# The Stack

- 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;  
}
```



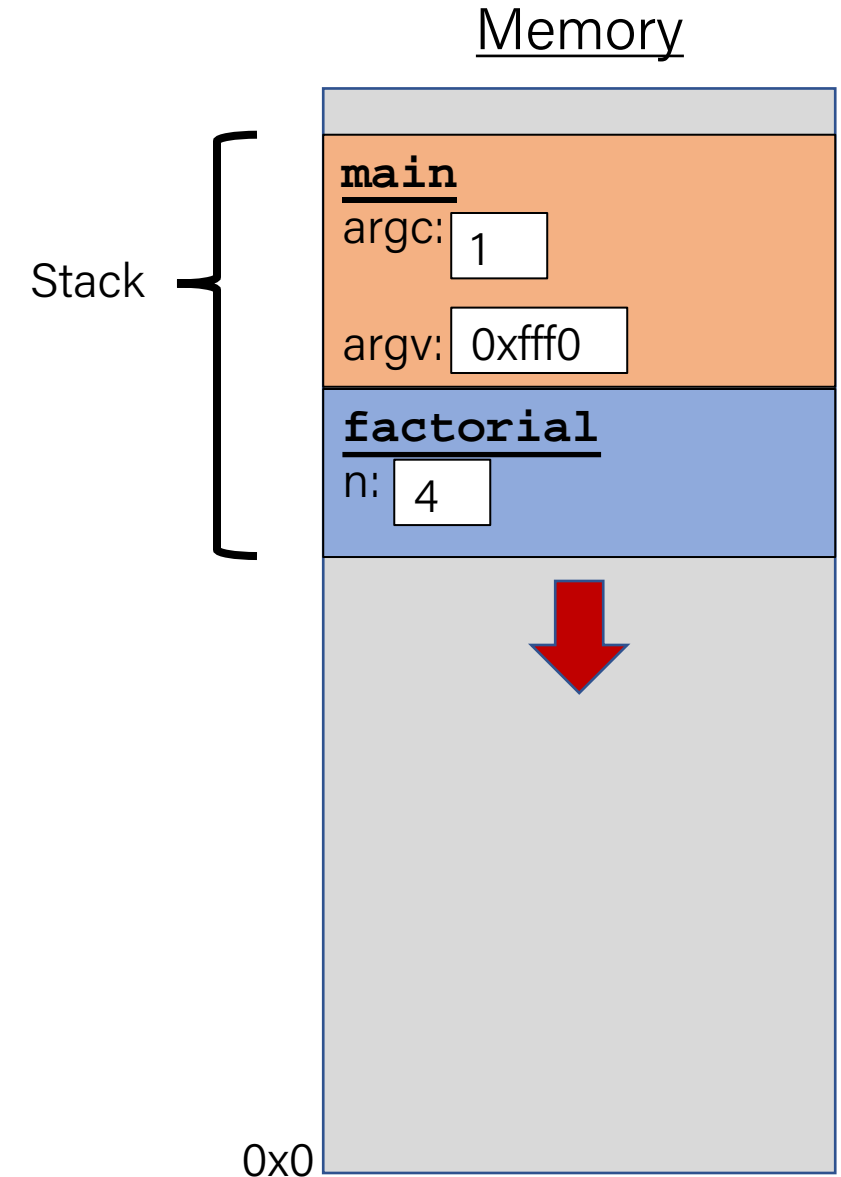


# The Stack

- 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);  
    }  
}
```

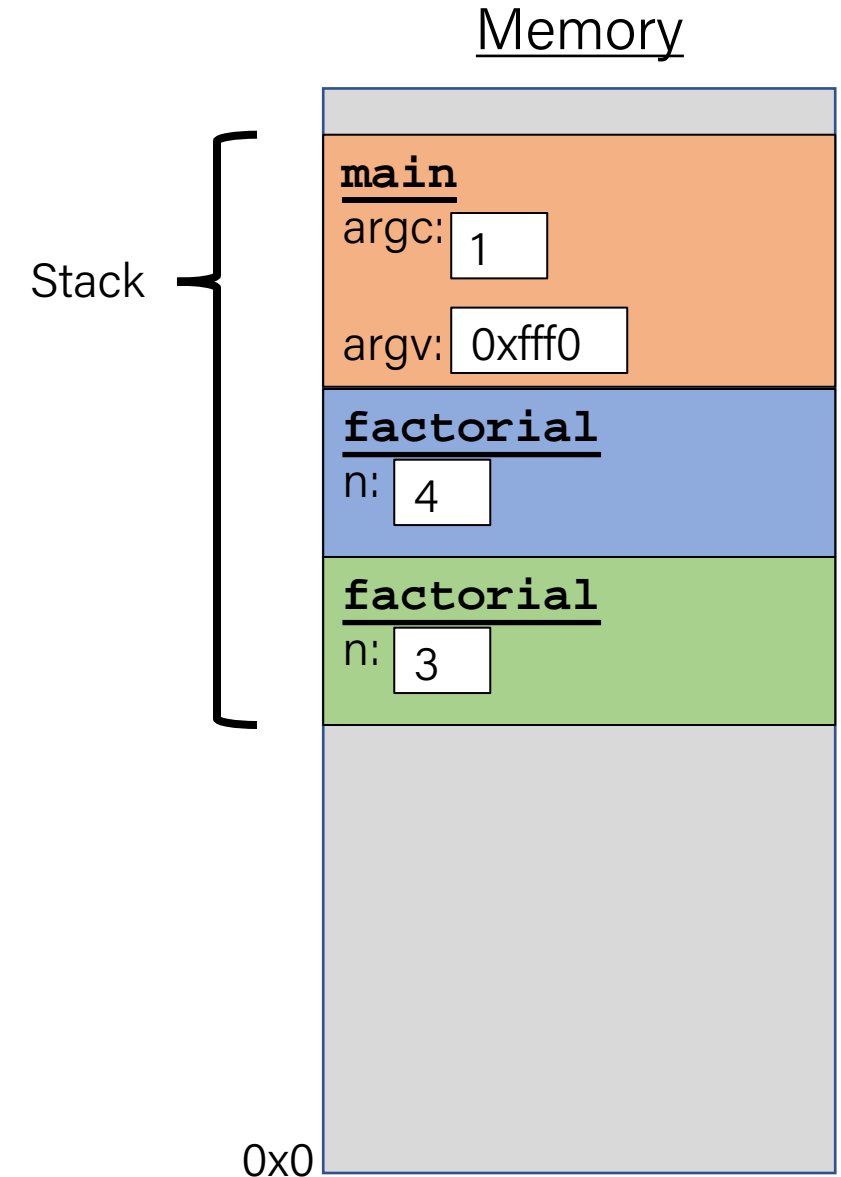
```
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

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

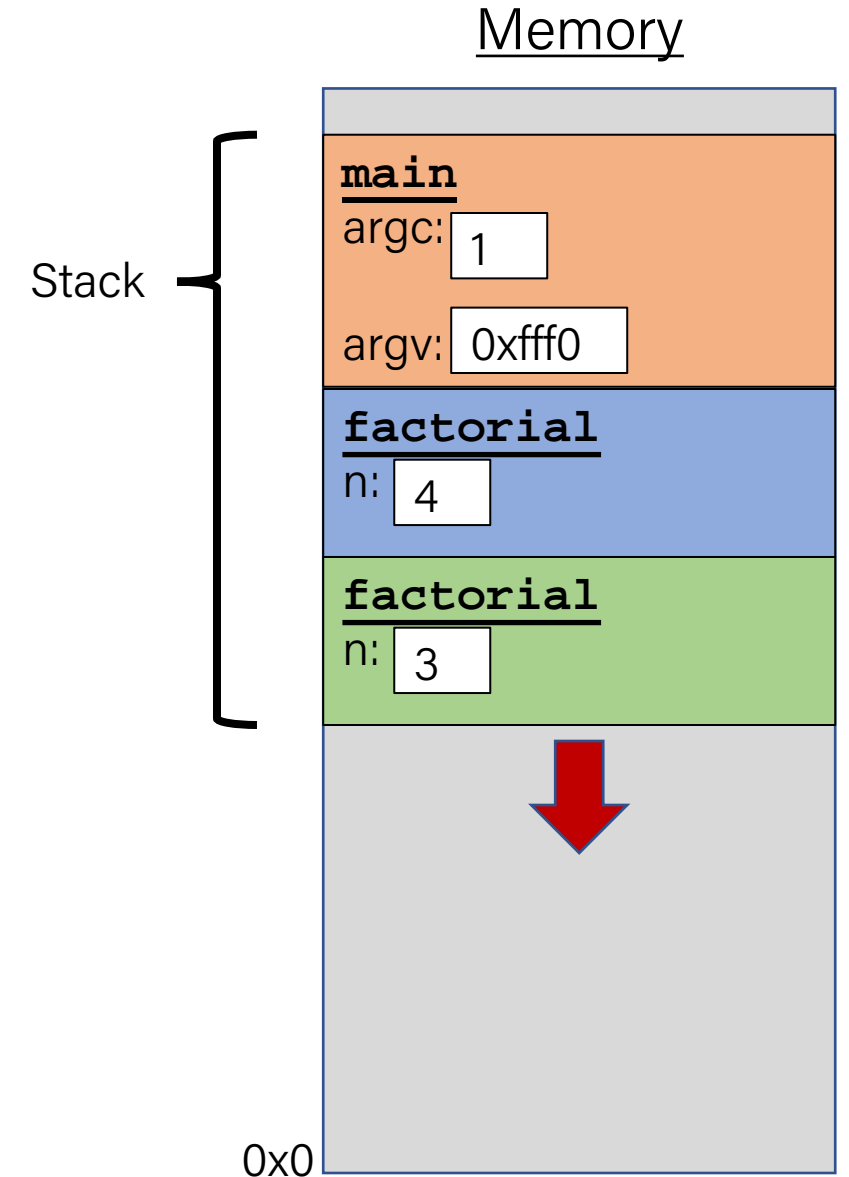
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

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

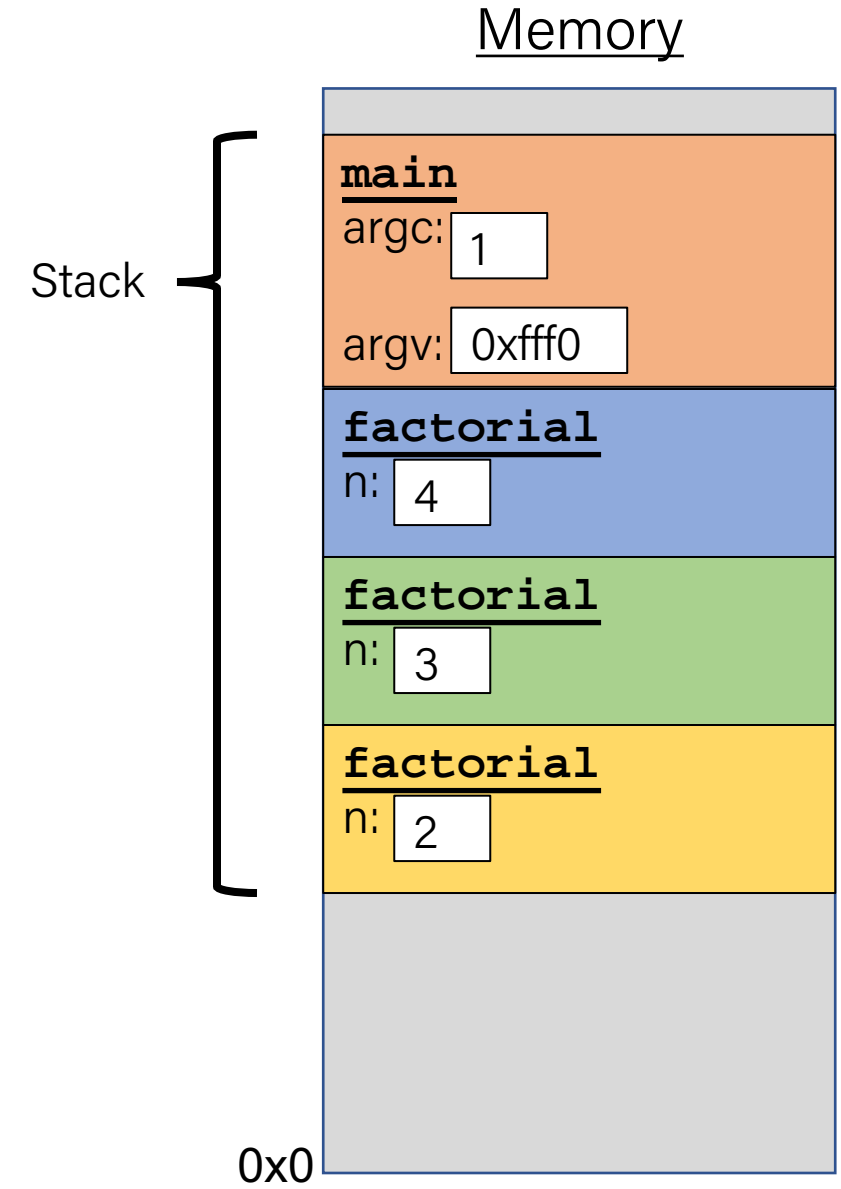
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

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

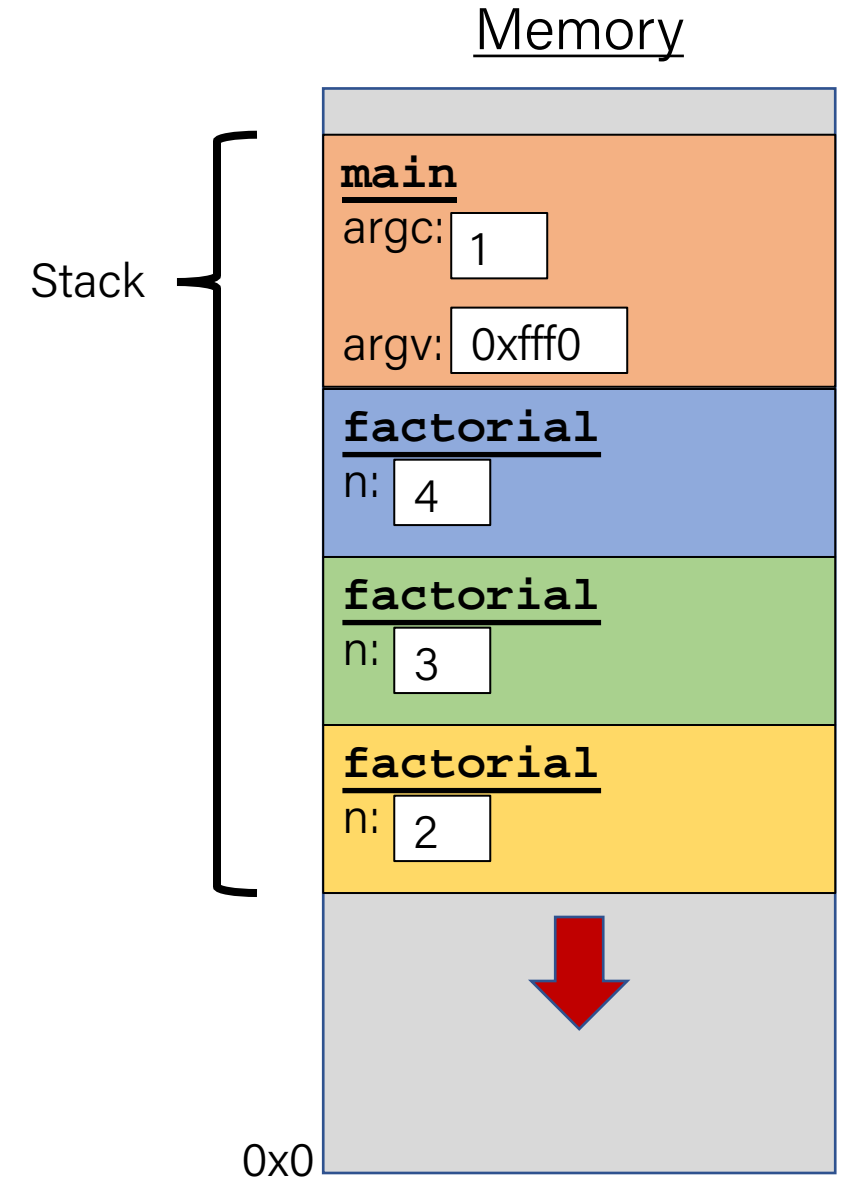
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# The Stack

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

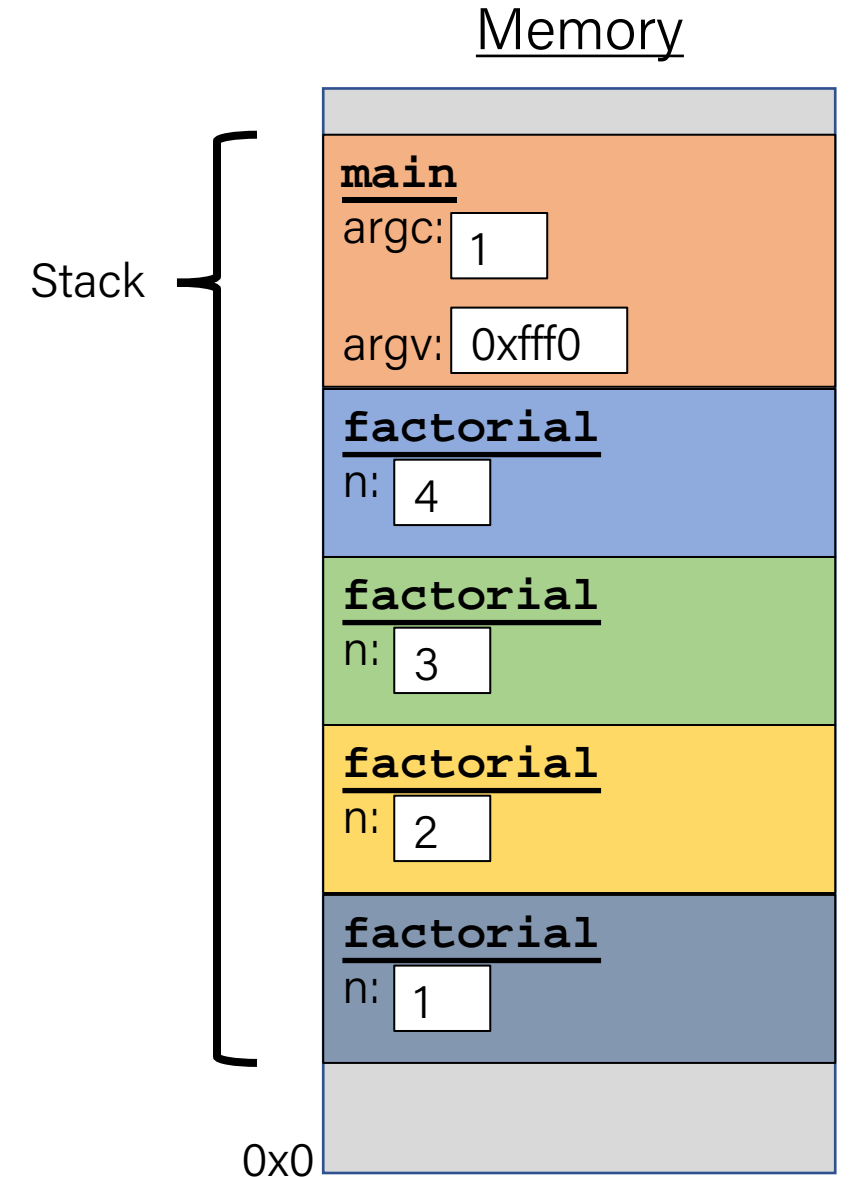
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# 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;  
}
```

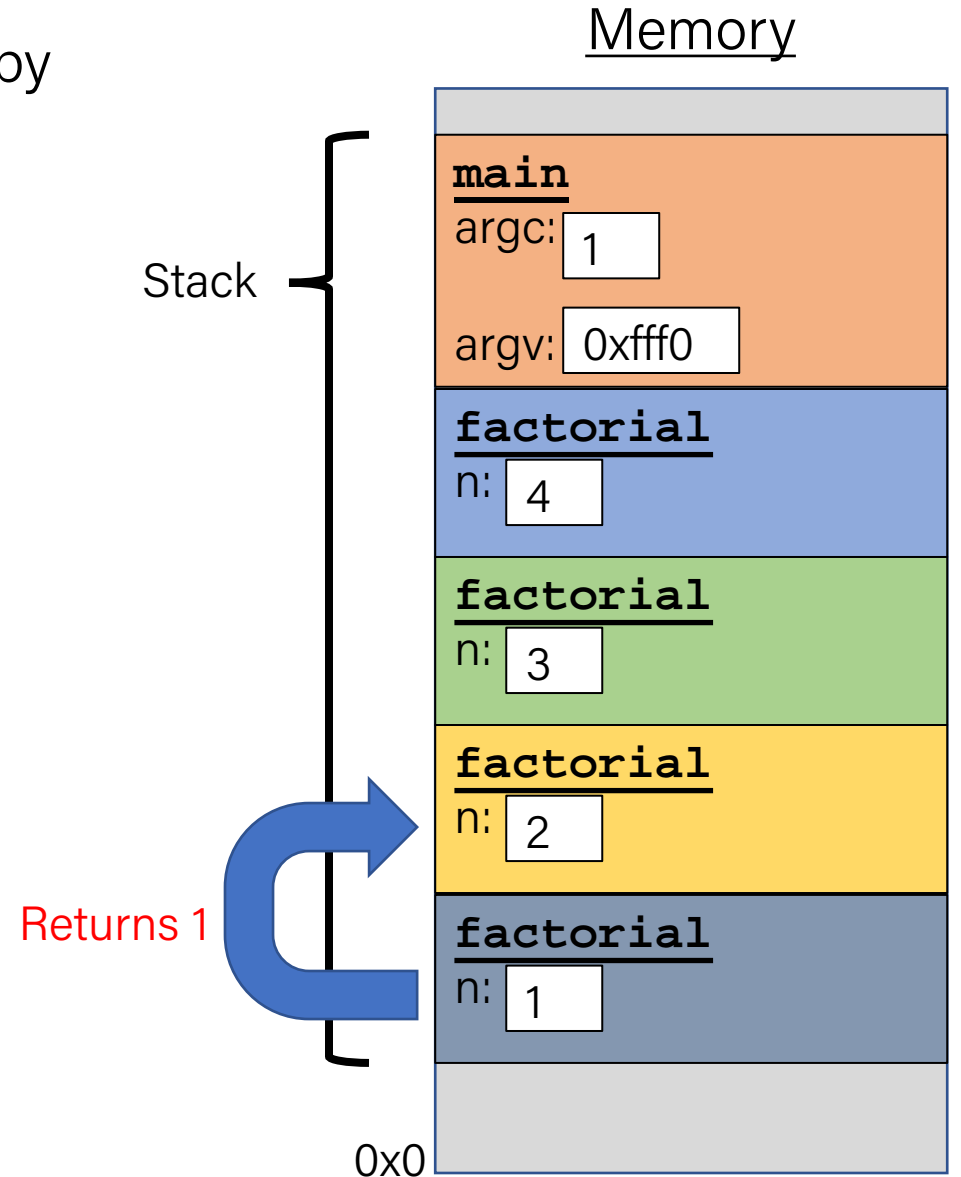


# The Stack

- 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);  
    }  
}
```

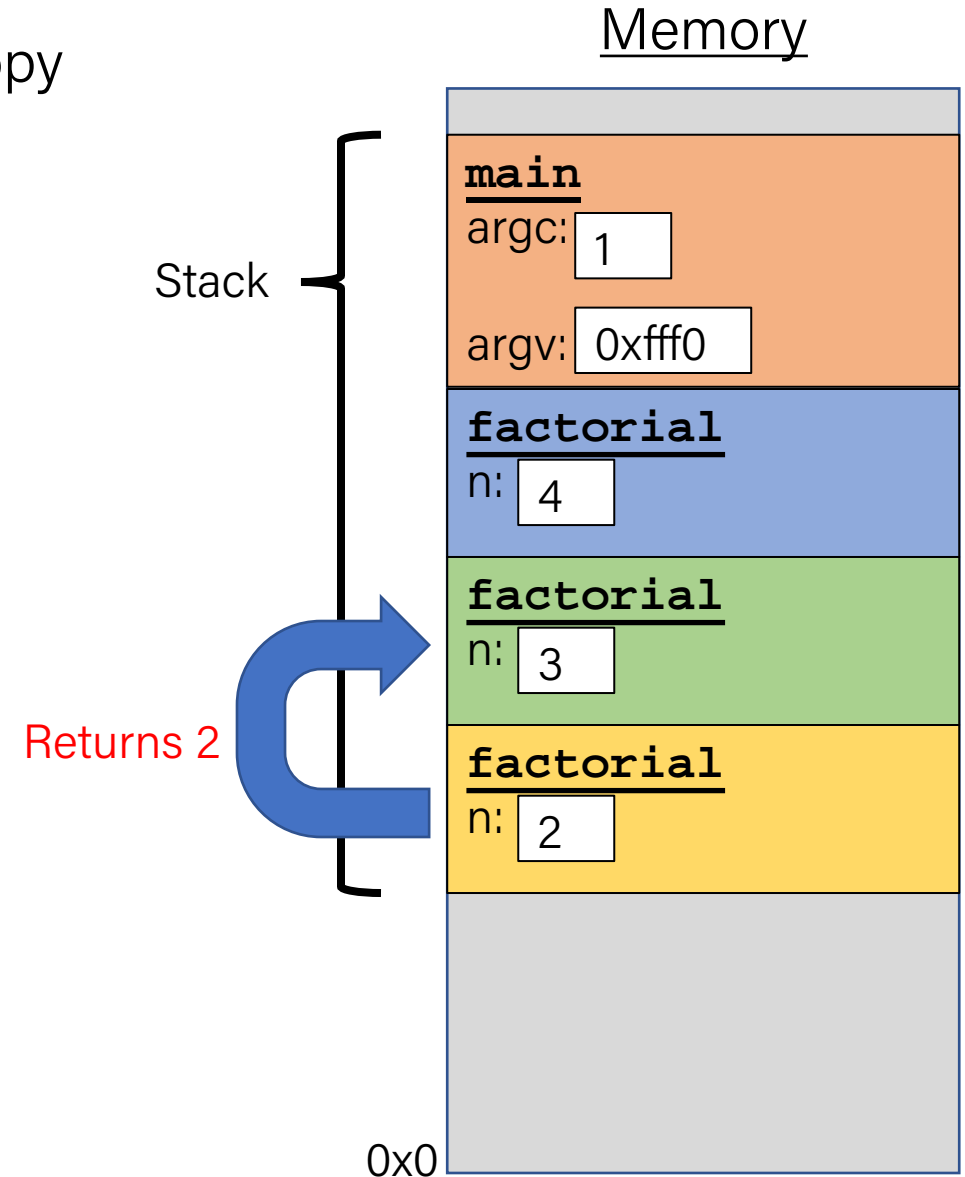
```
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



# 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;  
}
```

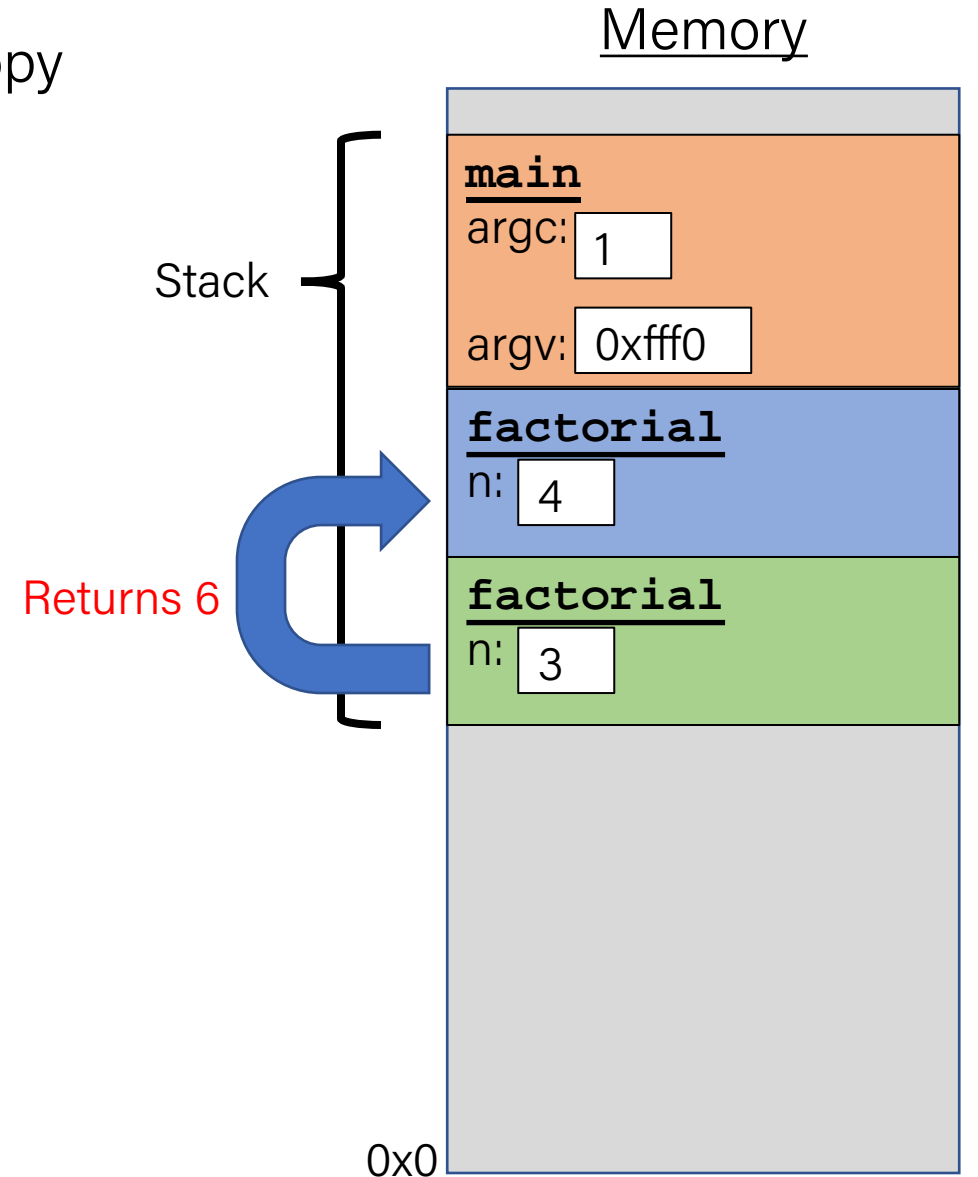




# The Stack

- 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;  
}
```

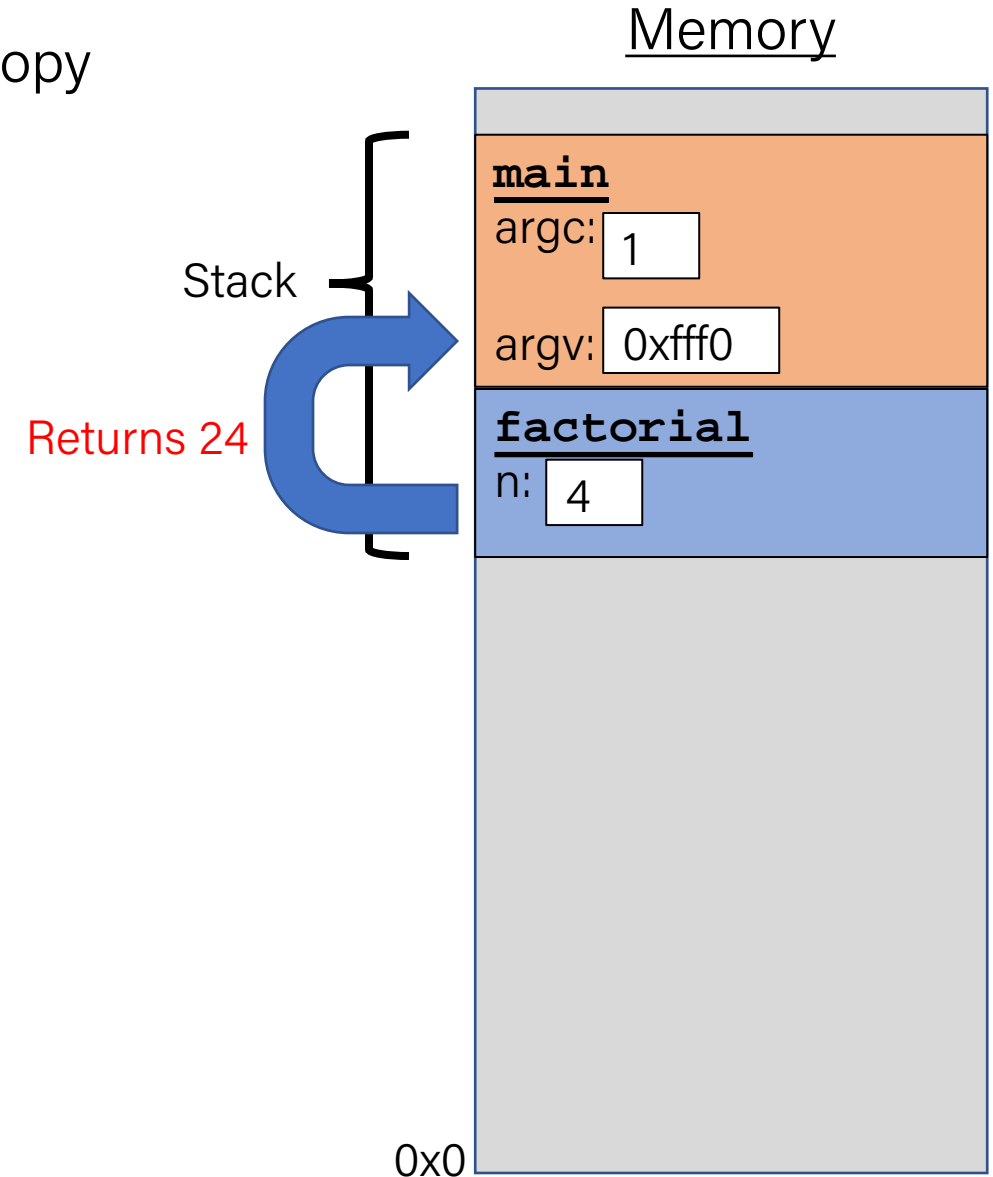


# The Stack

- 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;  
}
```

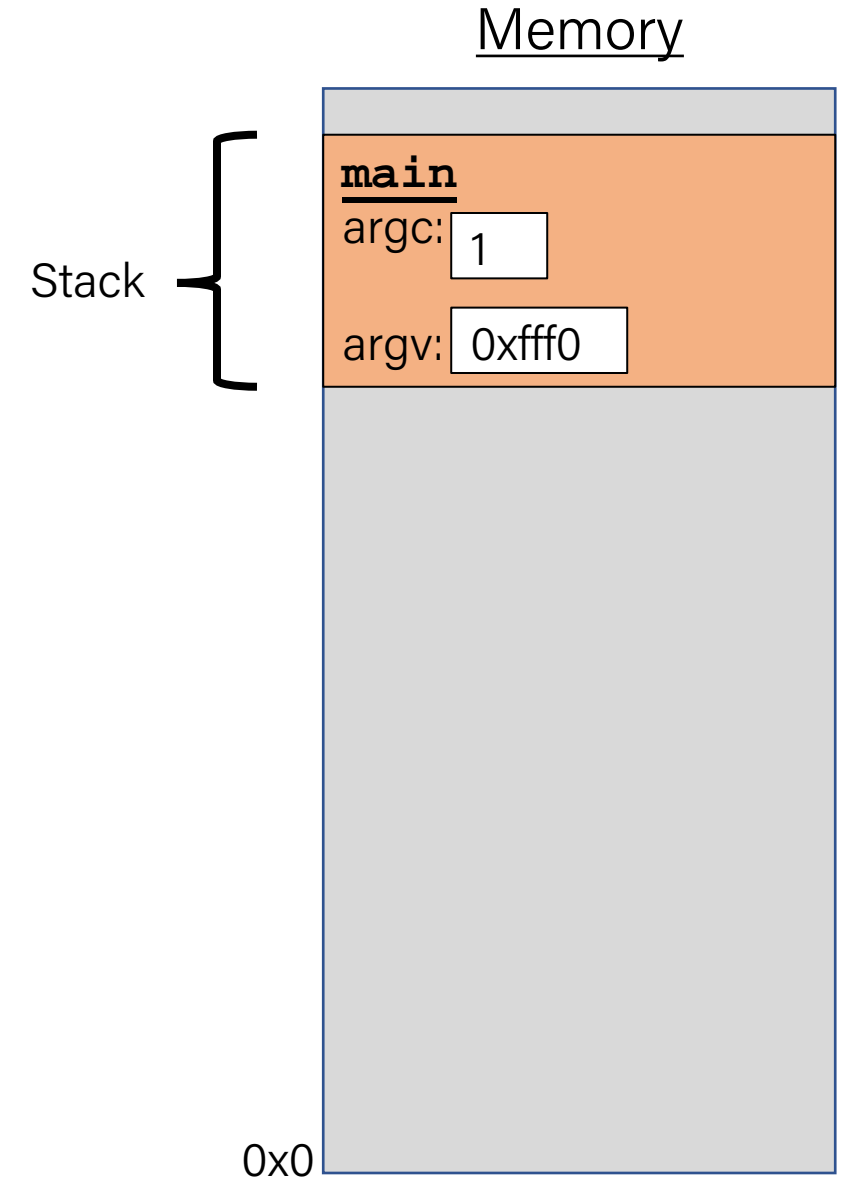


# The Stack

- 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;  
}
```

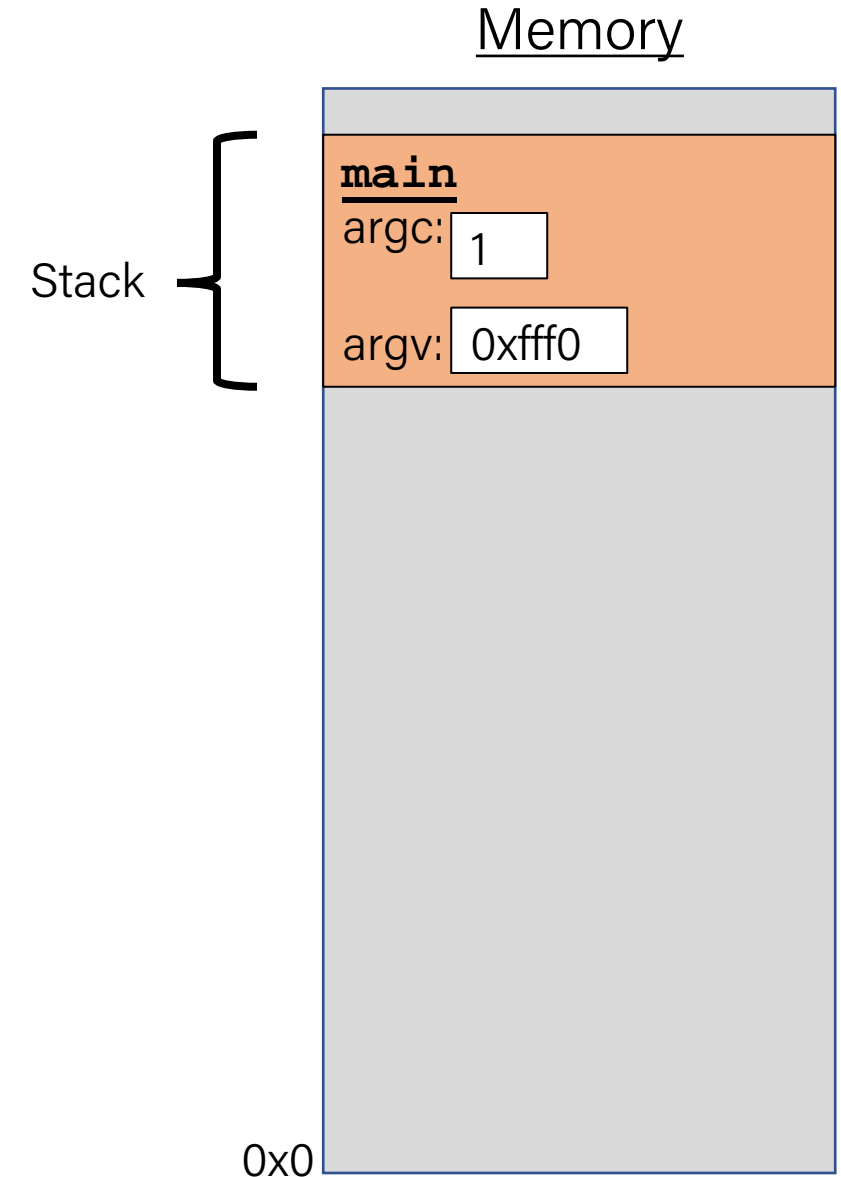


# The Stack

- 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;  
}
```



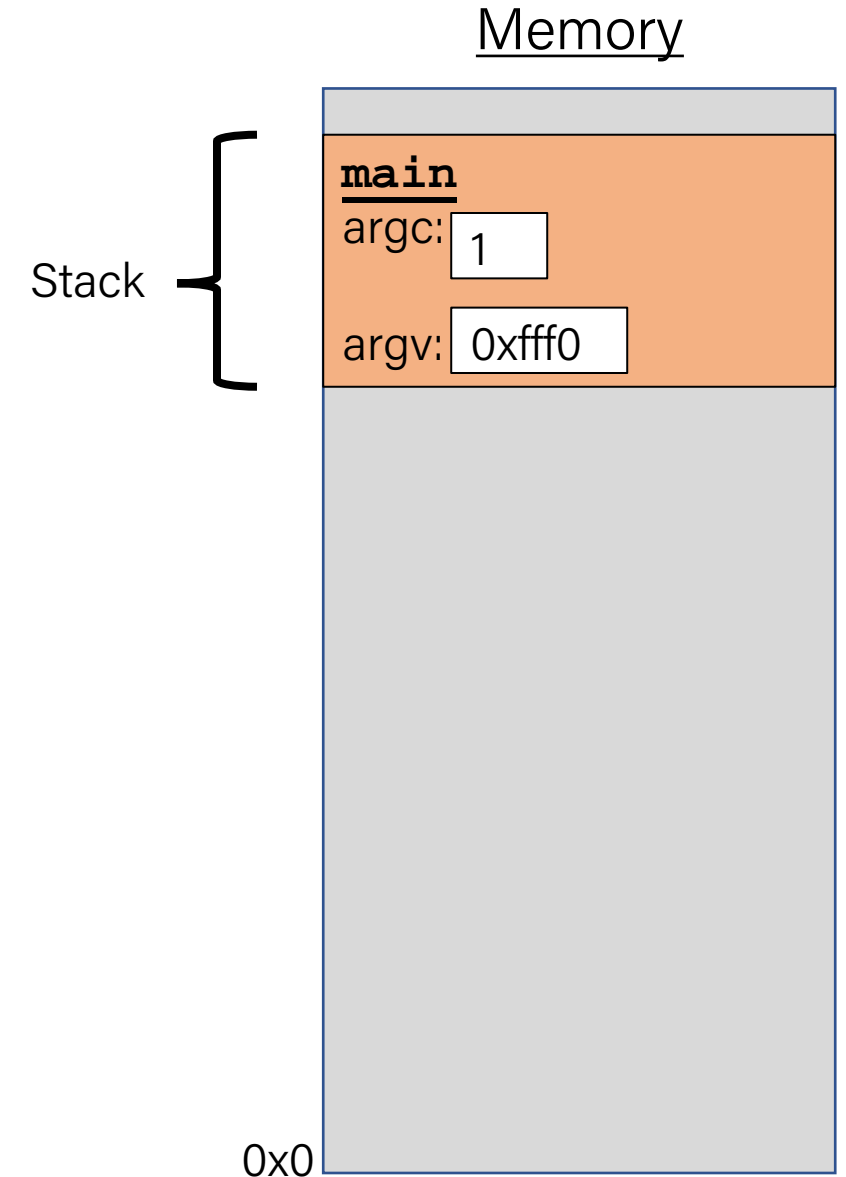
# The Stack

- 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!
- A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.
- What are the limitations of the stack?

# The Stack

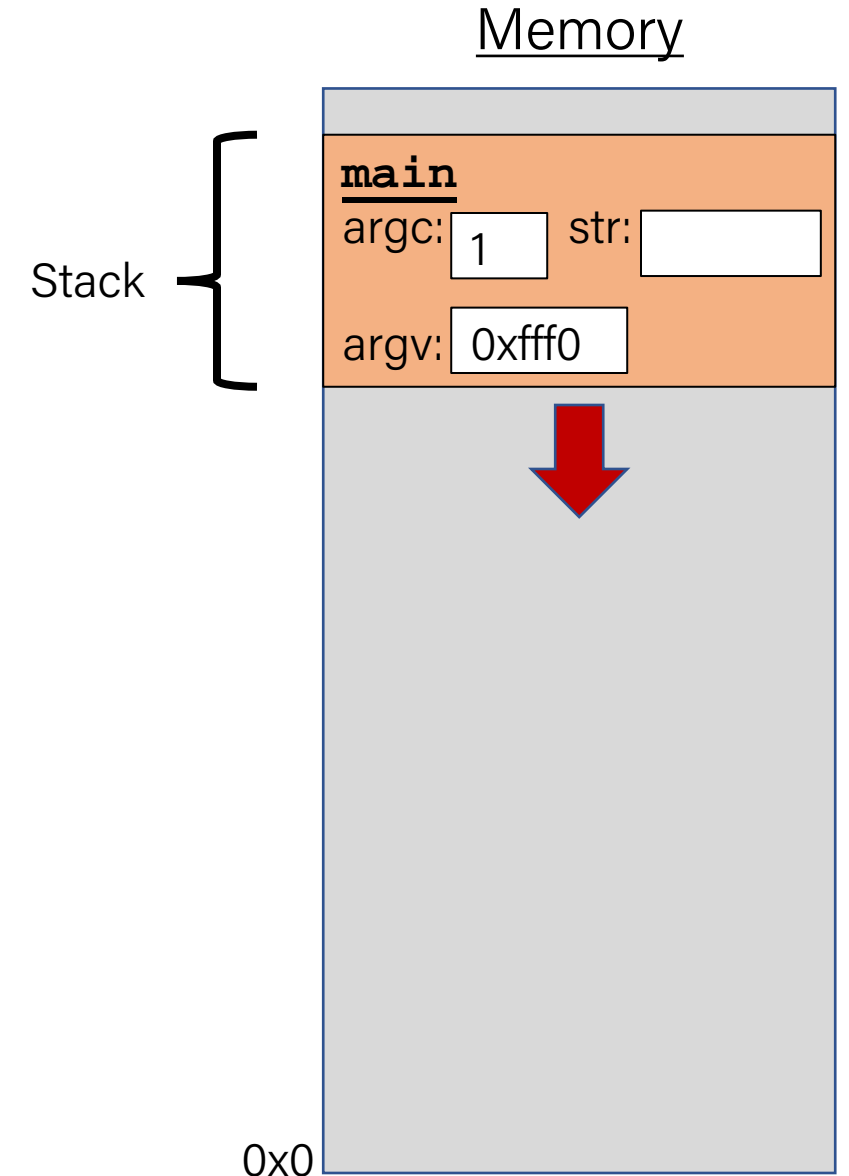
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}
```

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



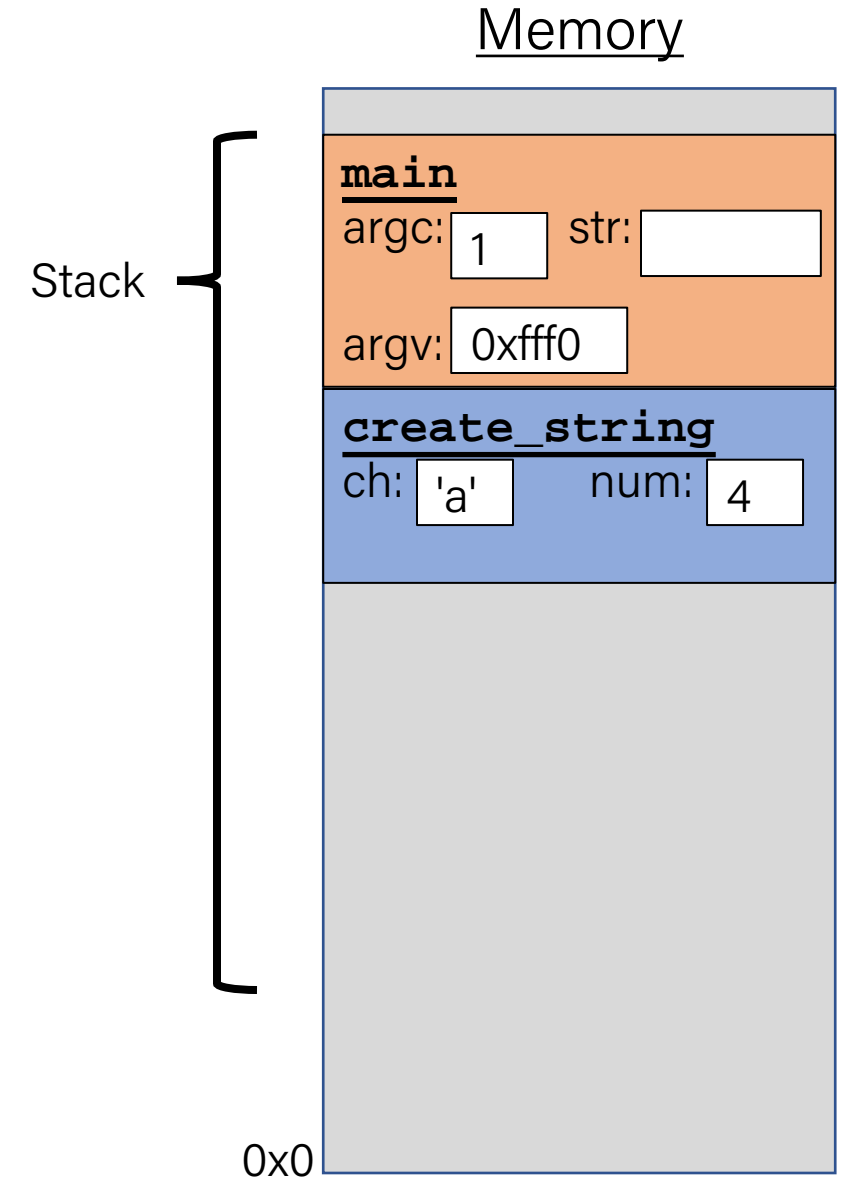
# The Stack

```
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

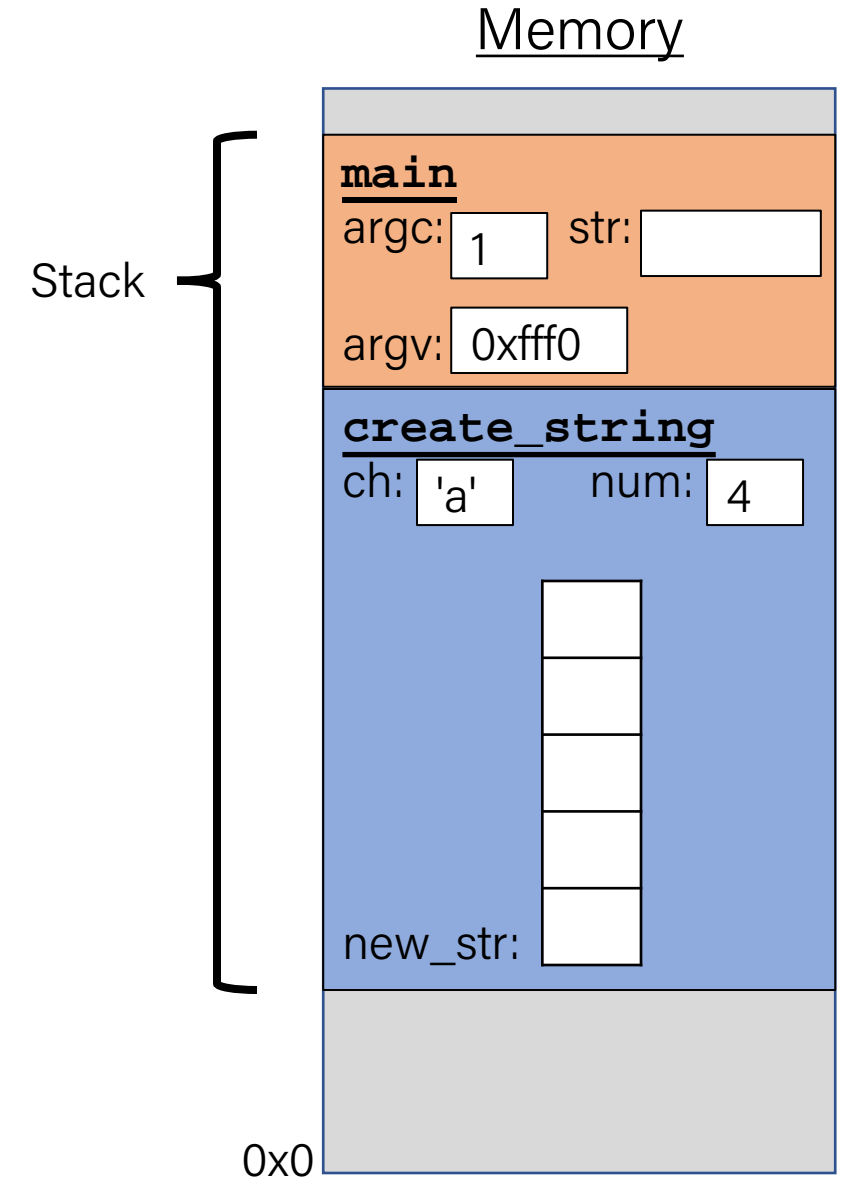
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str);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





# The Stack

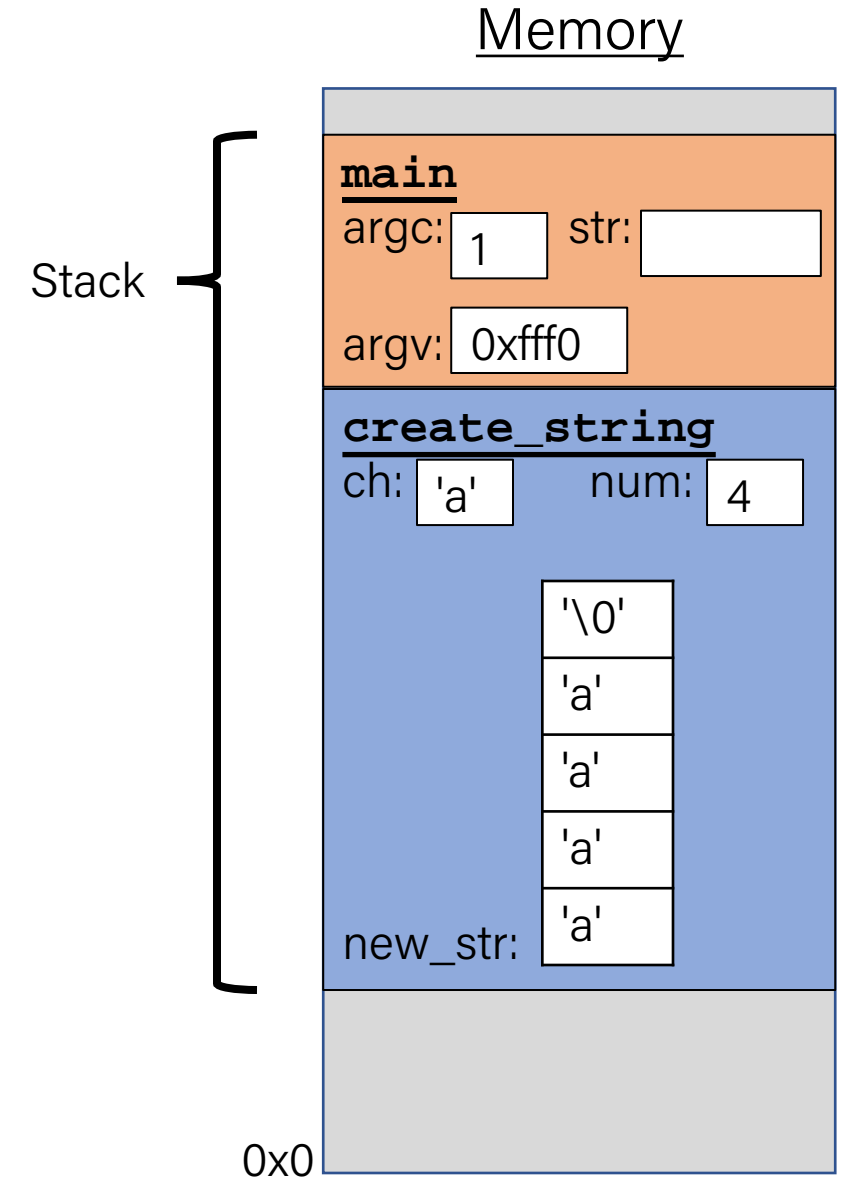
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str);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

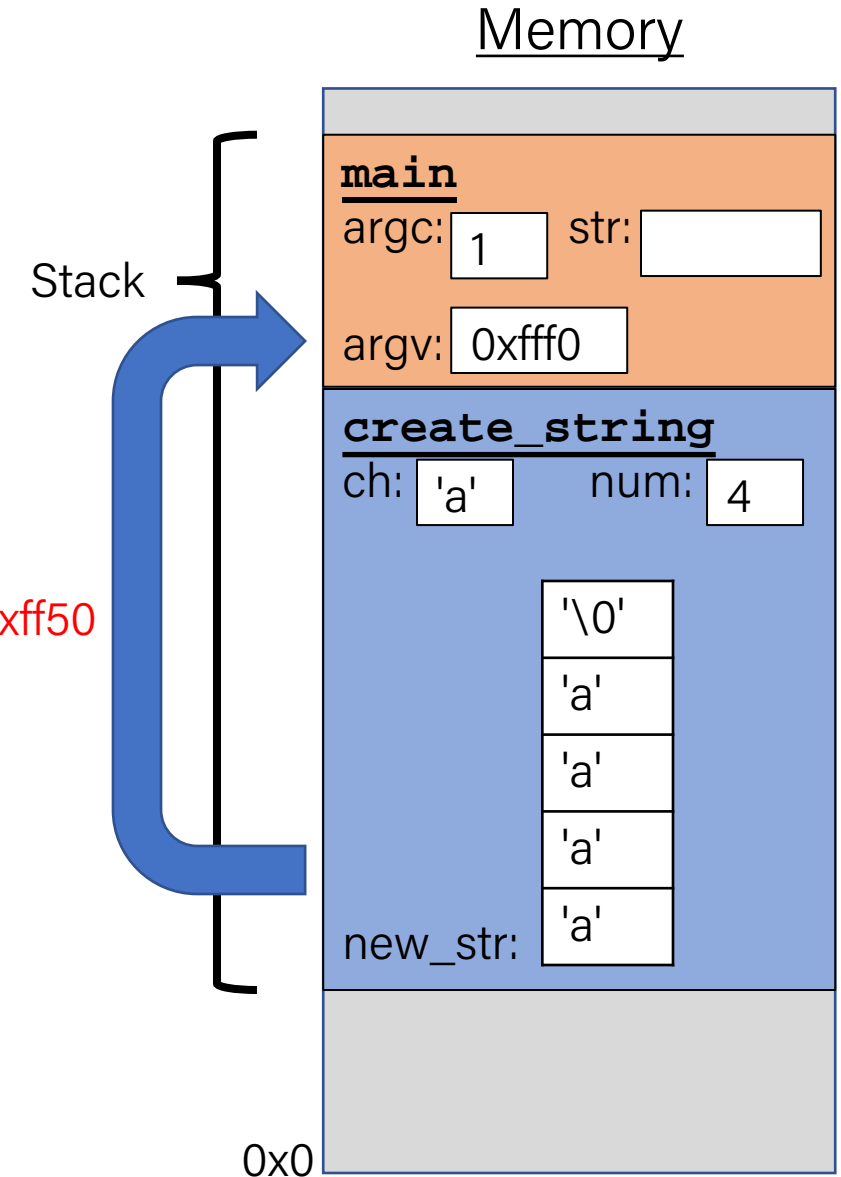


# The Stack

```
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;  
}
```

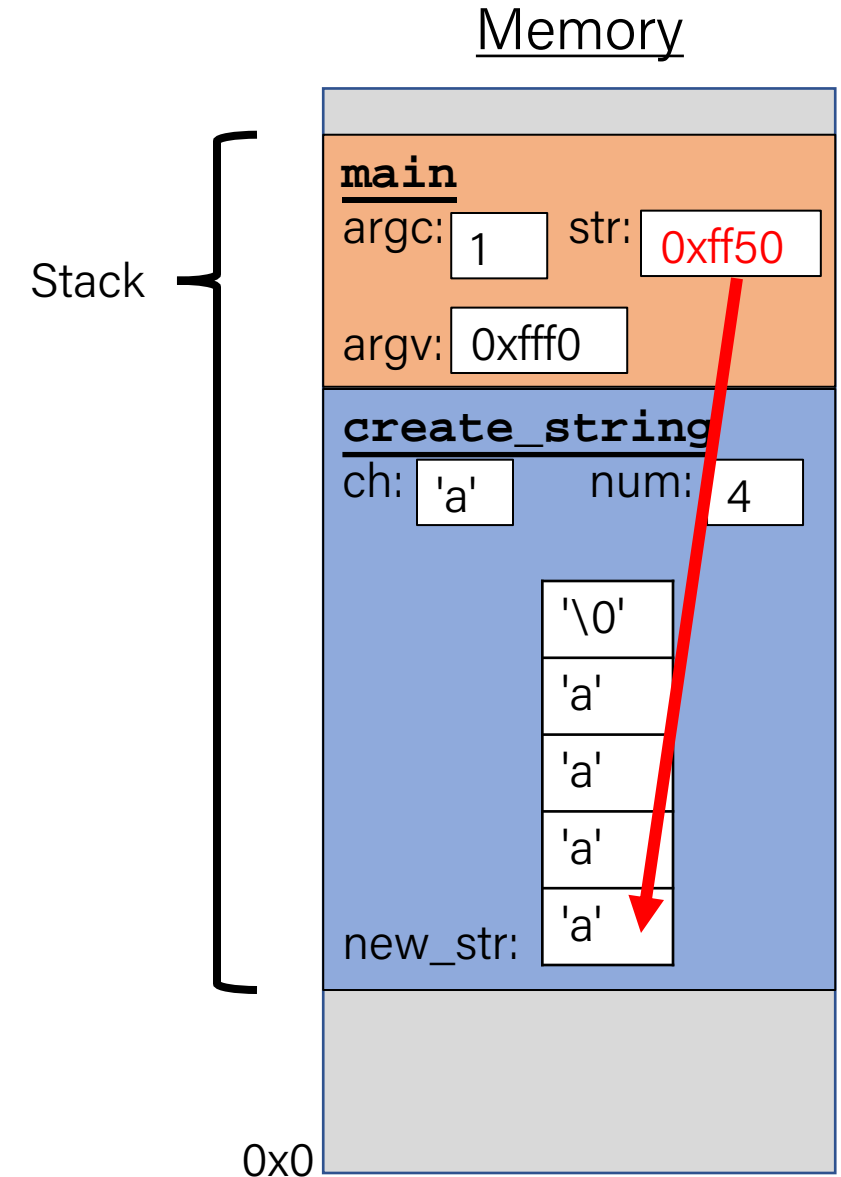
Returns e.g. 0xff50



# 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;  
}
```

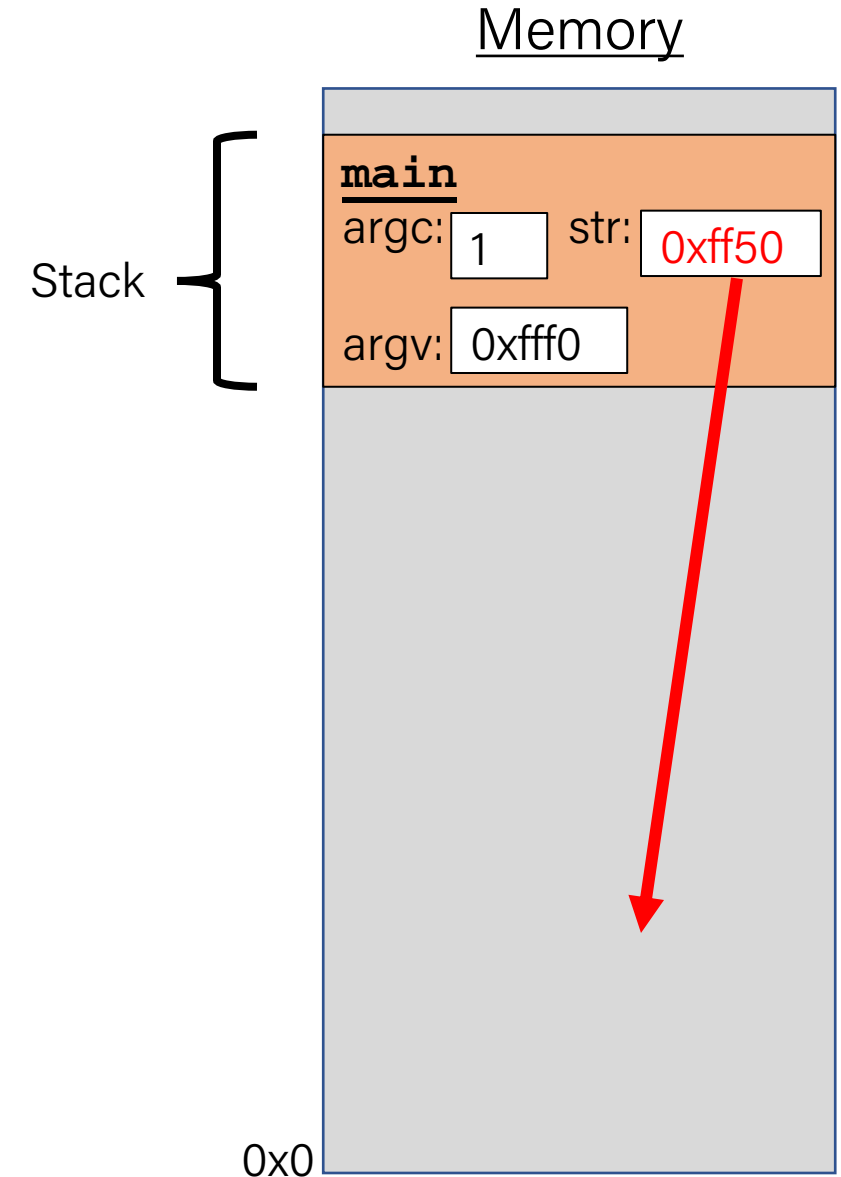
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```



# 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;  
}
```

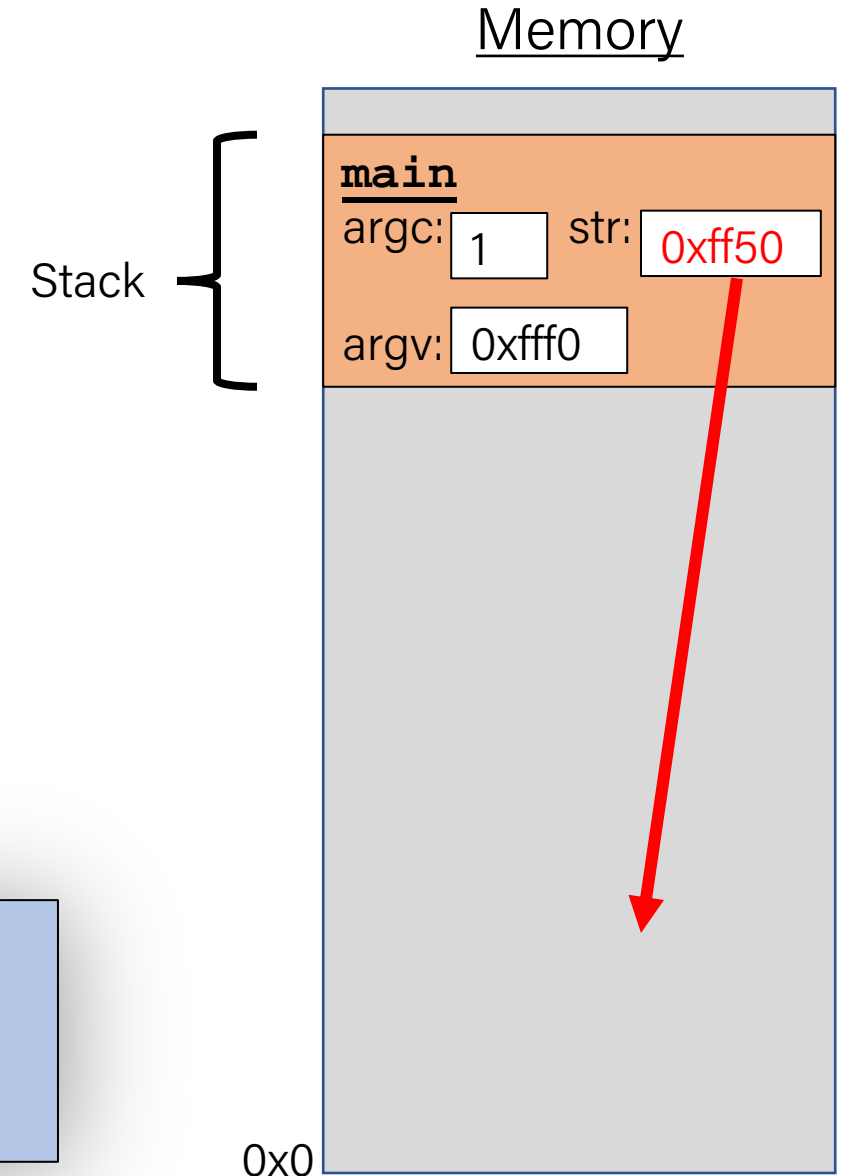


# The Stack

```
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;  
}
```

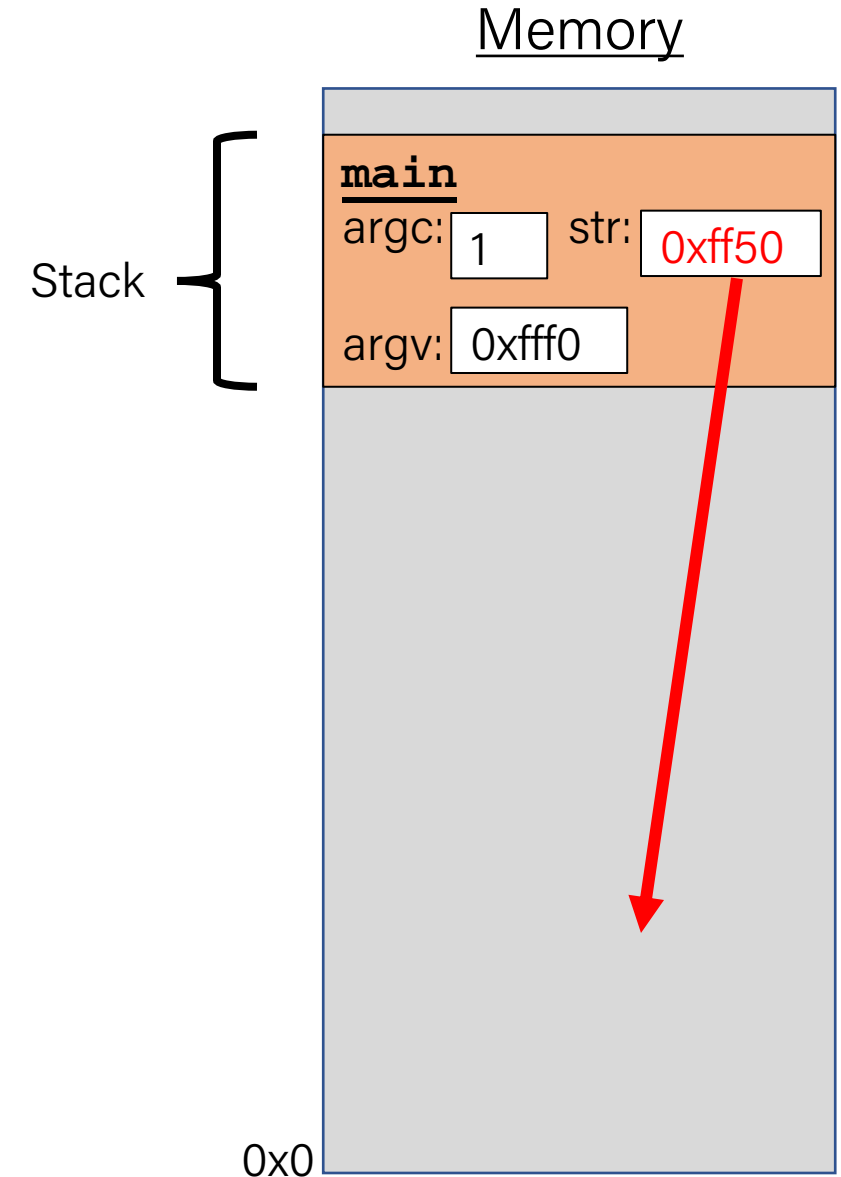
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```

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

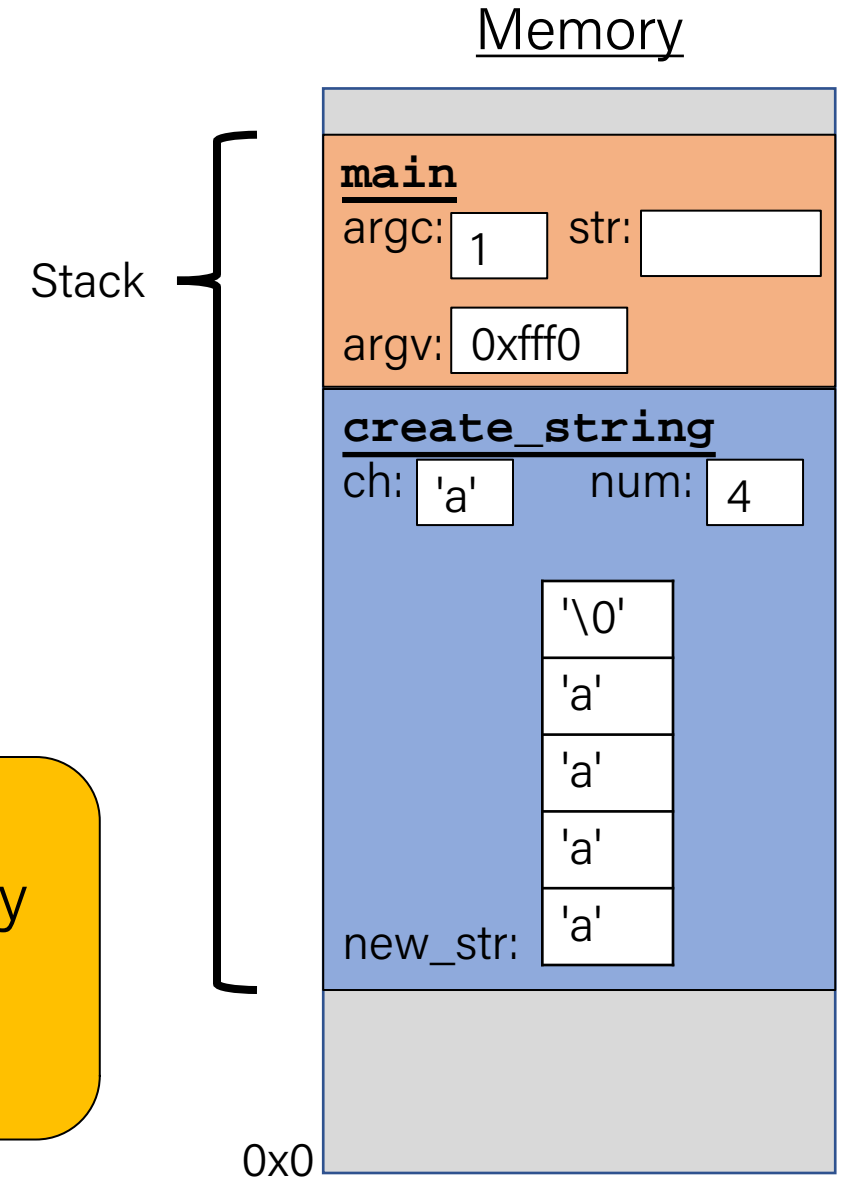
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory
- `realloc`

# 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);  
    return 0;  
}
```

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

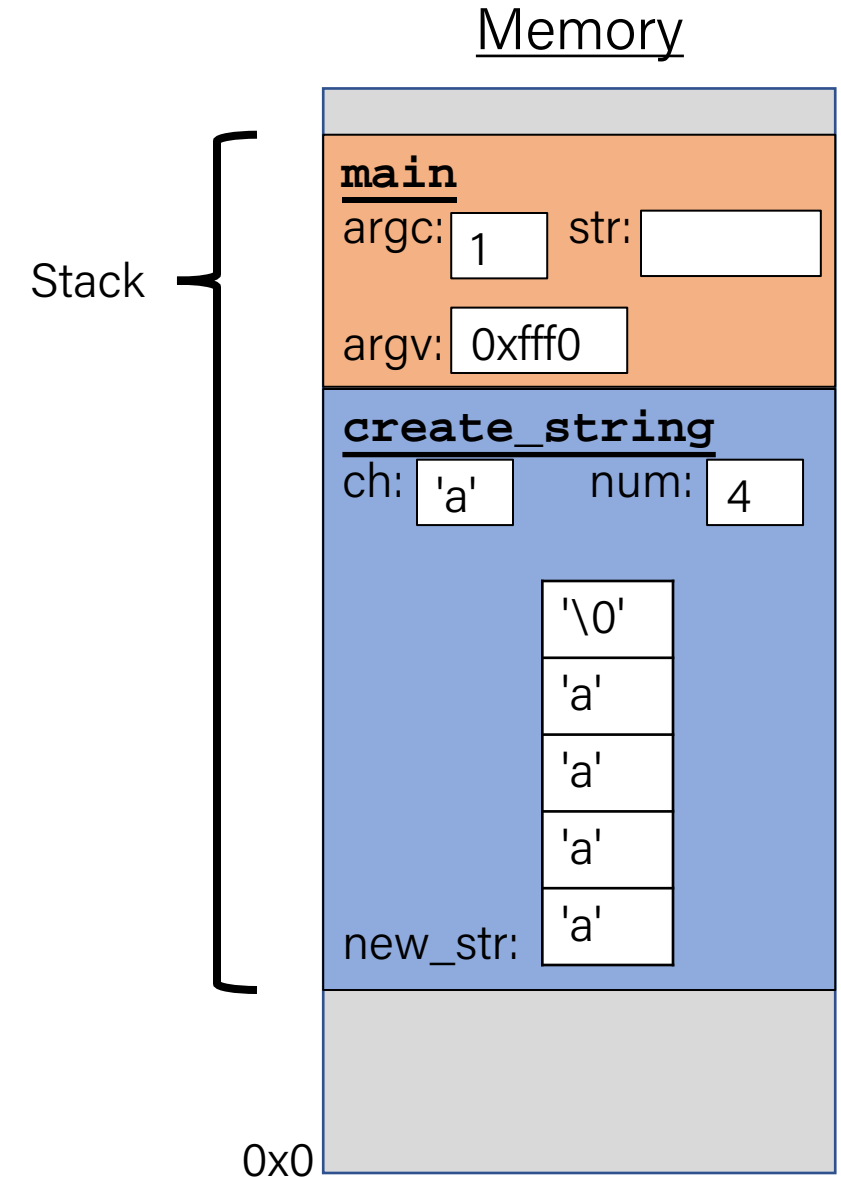


# The Heap

```
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;  
}
```

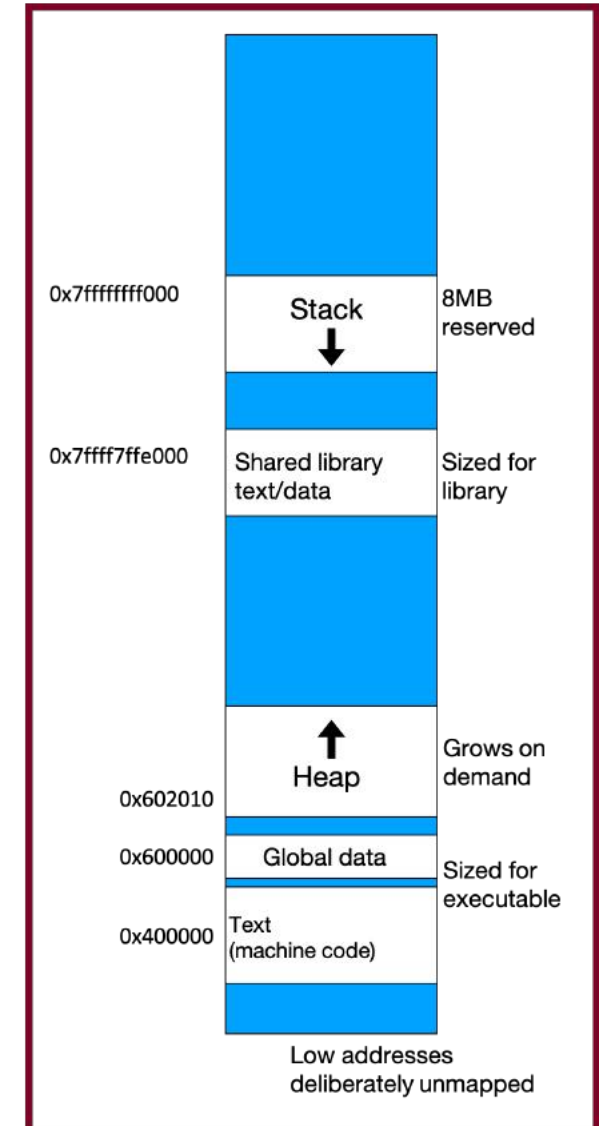
**C:** sure, but since I don't know when to clean it up anymore, it's your responsibility...



# 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

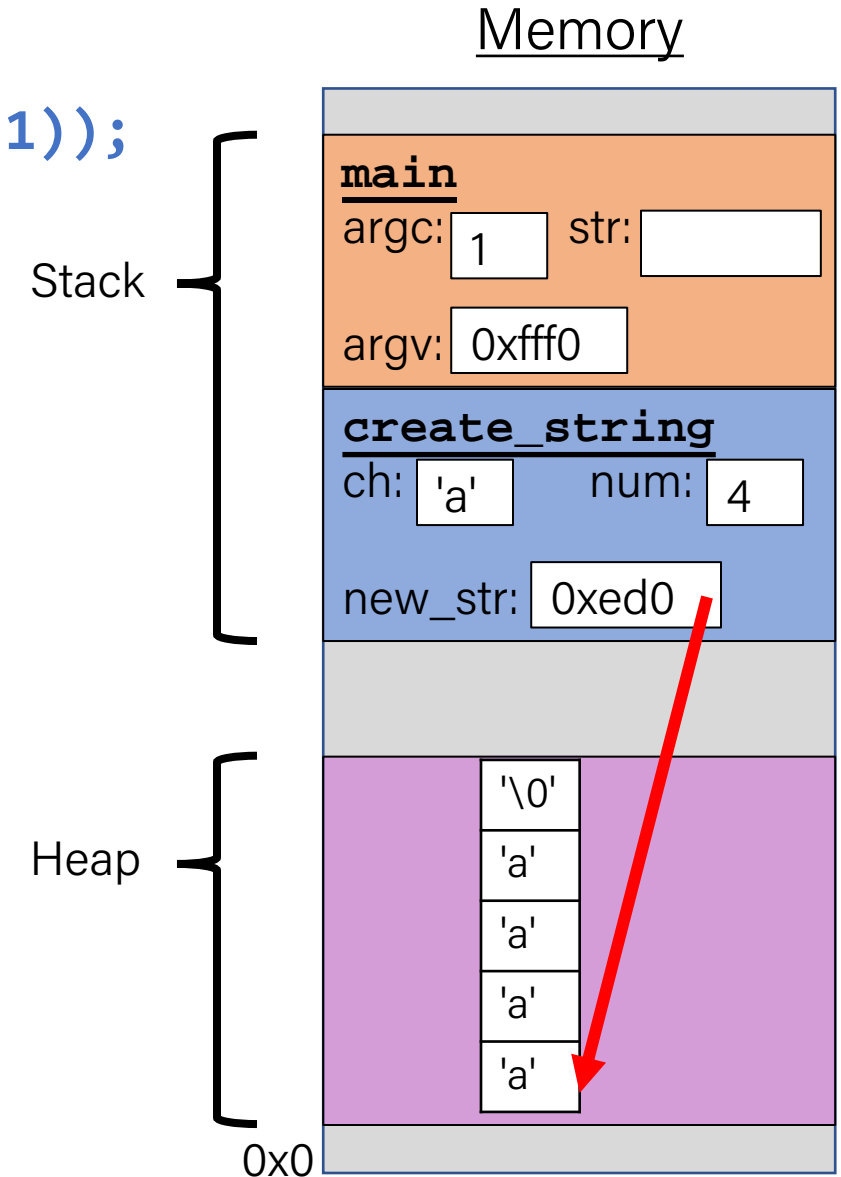
```
void *malloc(size_t size);
```

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

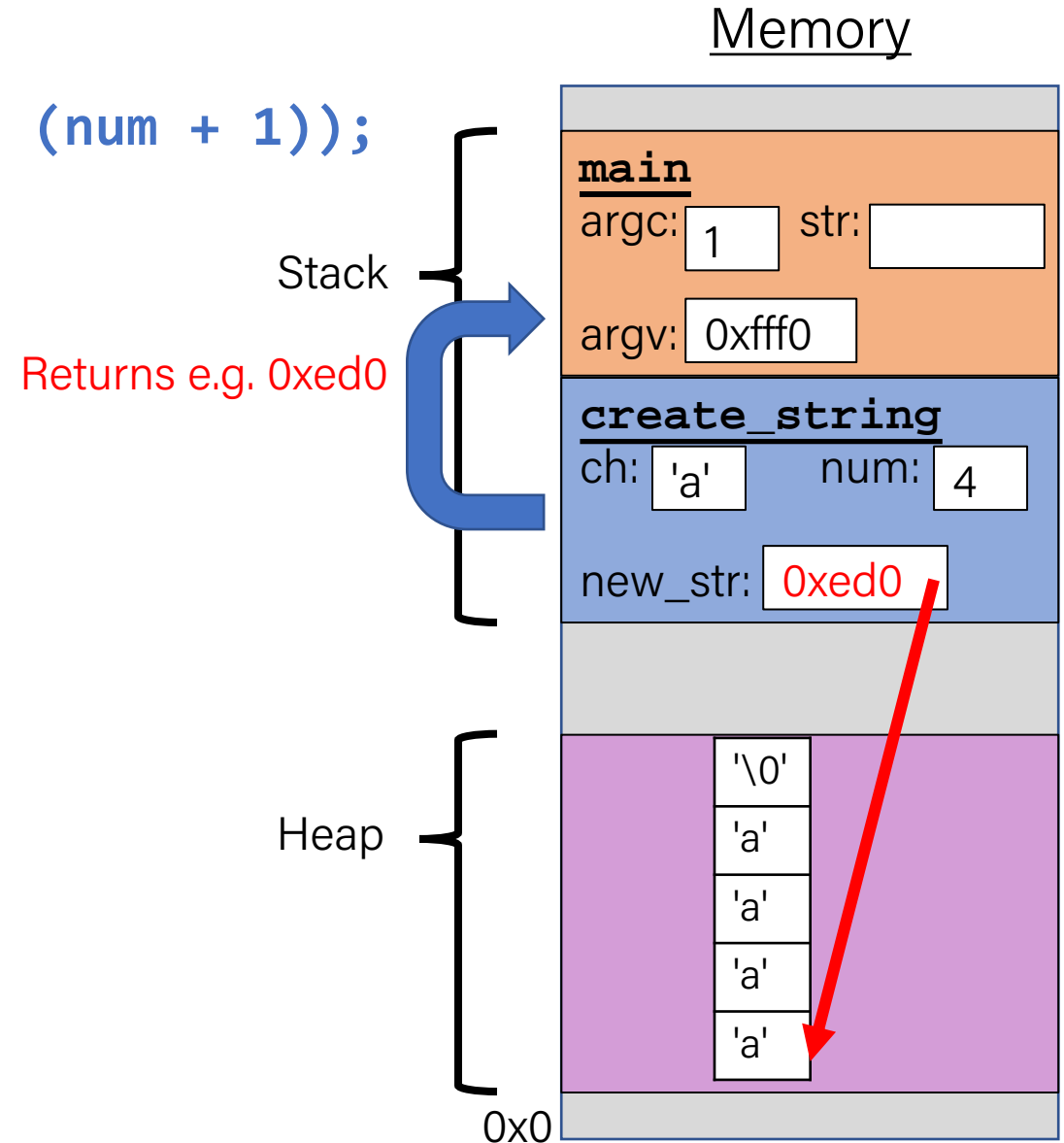
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a',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



# The Heap

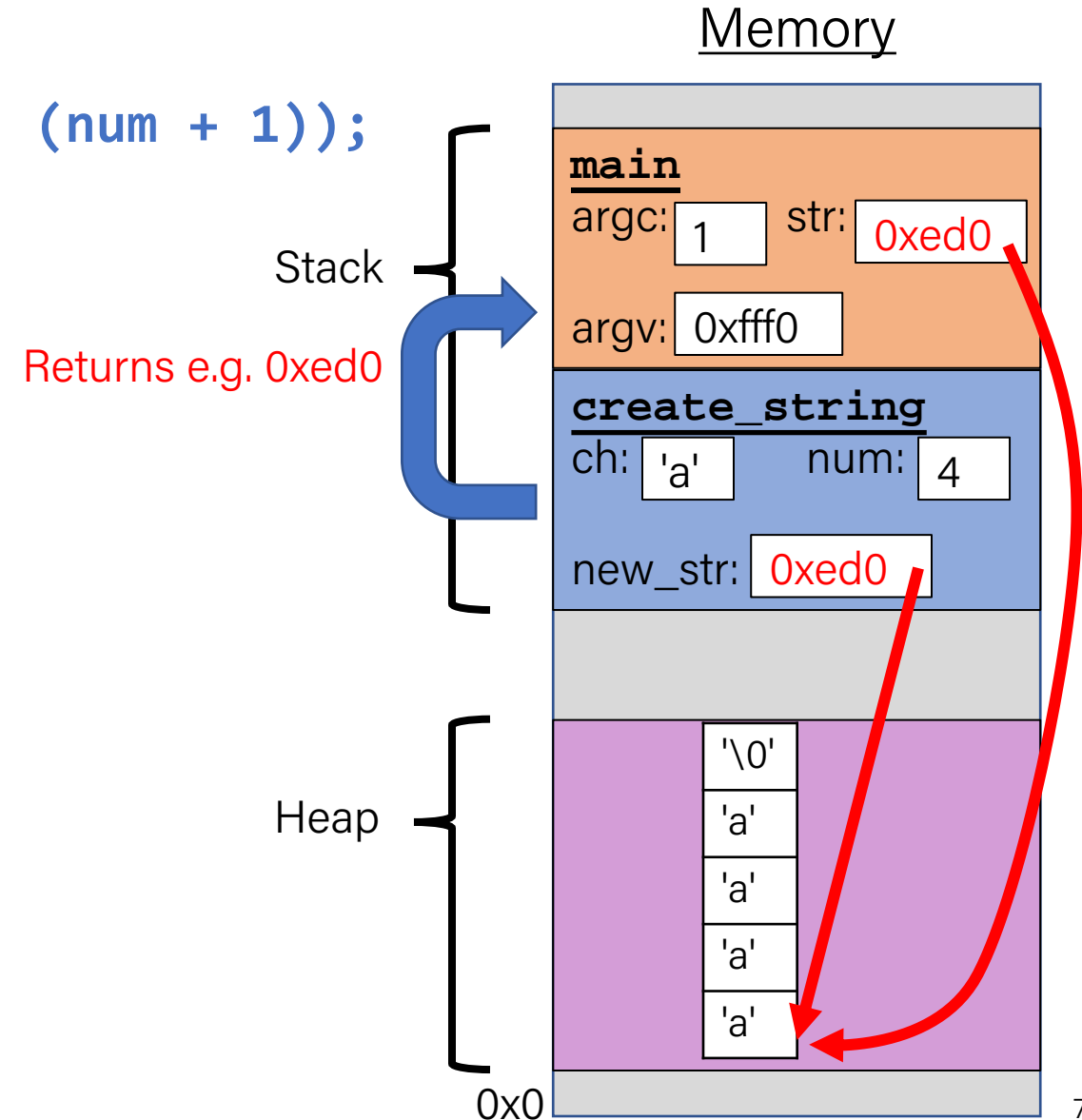
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}
```

```
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;  
}
```

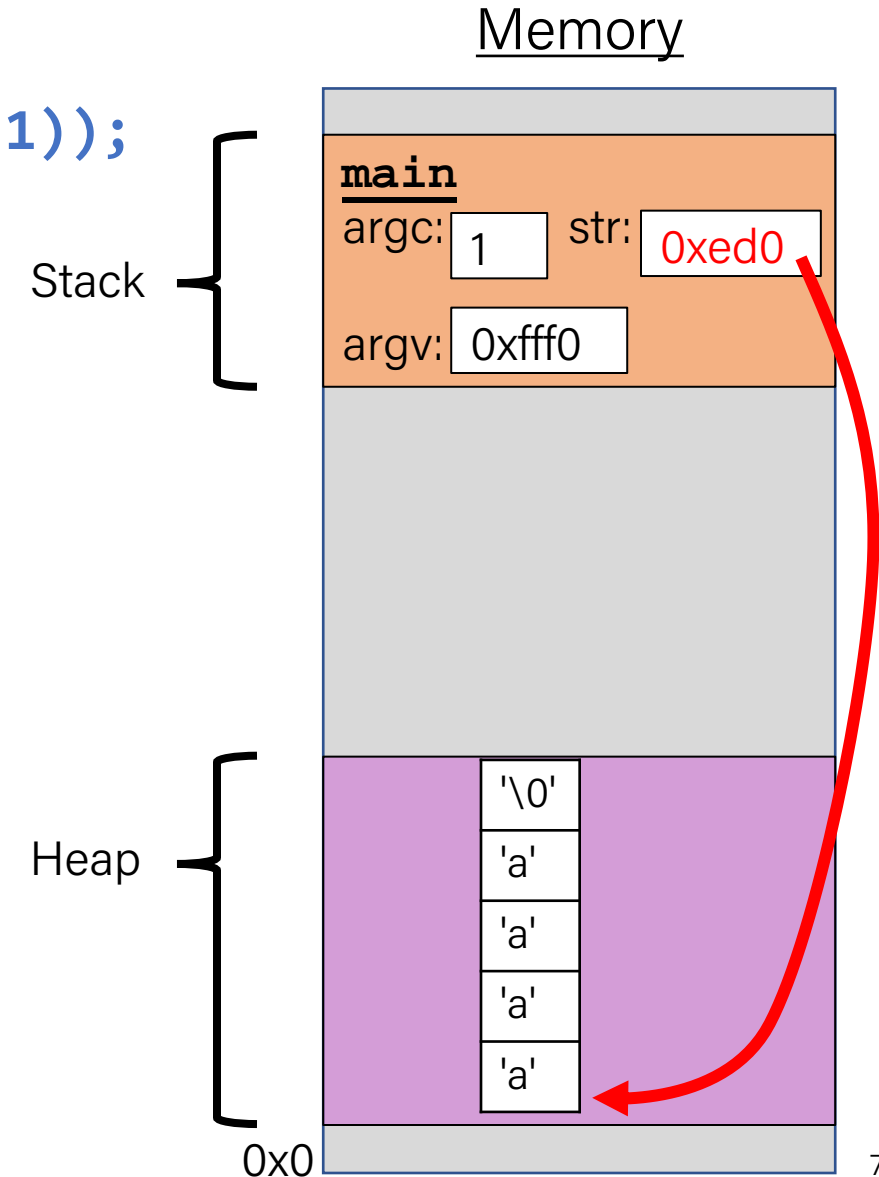




# The Heap

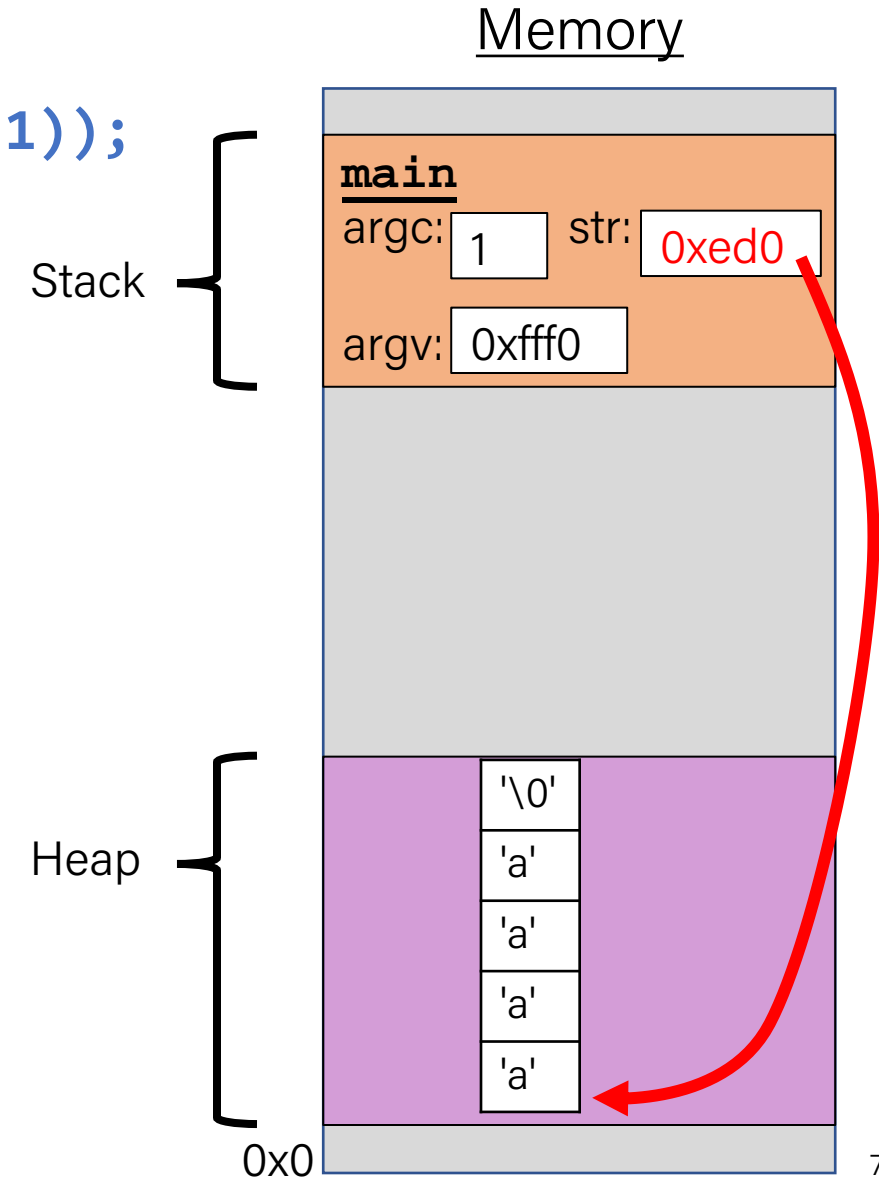
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}
```

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



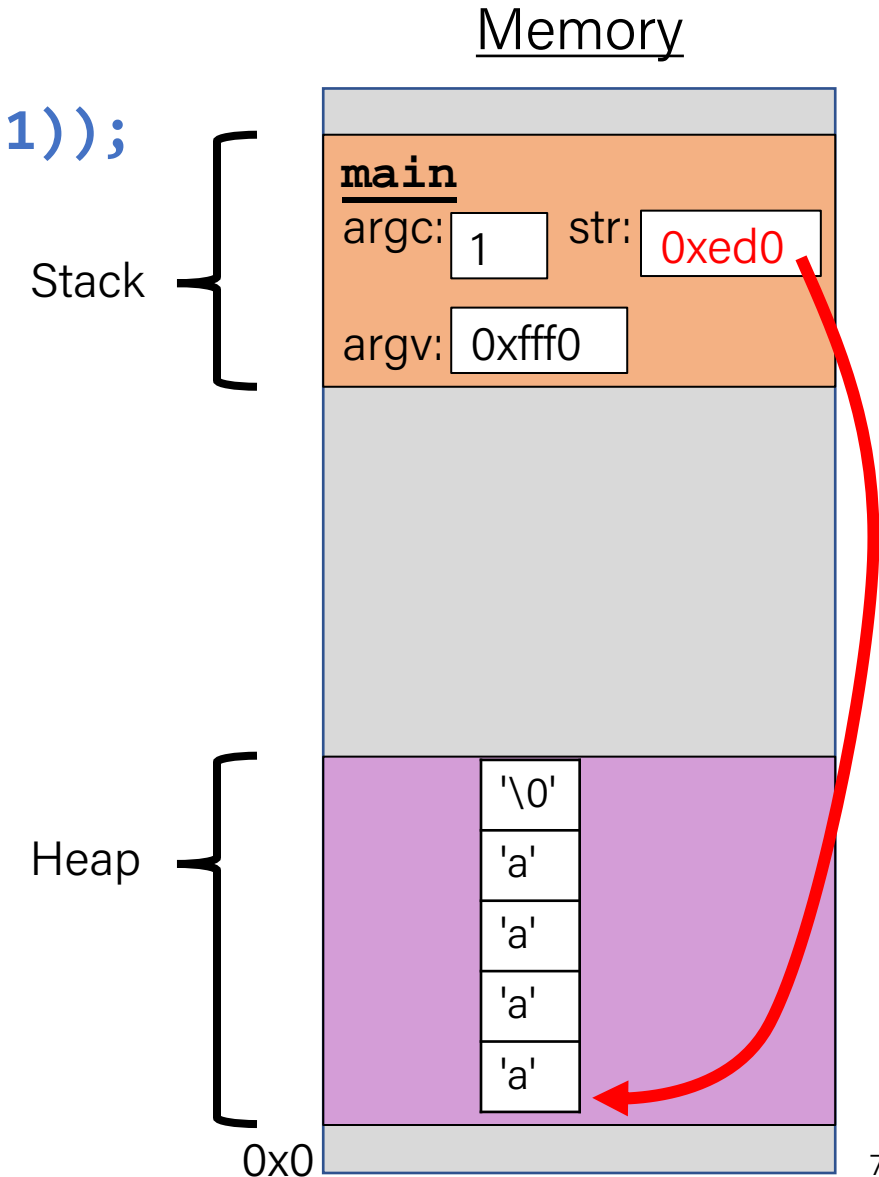
# The Heap

```
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**.

```
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));`
- E. Something else



# 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));`
- 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);  
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 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';
```



# 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 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);
```





**✗** 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);
```

# 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!
```

# 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!

# Demo: Pig Latin



```
pig_latin.c
```

# Lecture Plan

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory
- **realloc**

# 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?



# Demo: Pig Latin Part 2



```
pig_latin.c
```

# 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; 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 ("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

# Recap

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory
- `realloc`

Next time: C Generics