

CS107, Lecture 10

Stack and Heap

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

CS107 Topic 3

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

Why is answering this question important?

- Shows us how we can pass around data efficiently with pointers (last time)
- Introduces us to the heap and allocating memory that we manually manage (today)
- Helps us better understand use-after-free vulnerabilities, a common bug (next time)

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

Learning Goals

- Learn about the differences between arrays and pointers
- Learn about the differences between the stack and the heap and when to use each one
- Become familiar with the **malloc** function for allocating memory on the heap

Lecture Plan

- **Recap:** Pointers So Far
- Arrays in Memory
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Lecture Plan

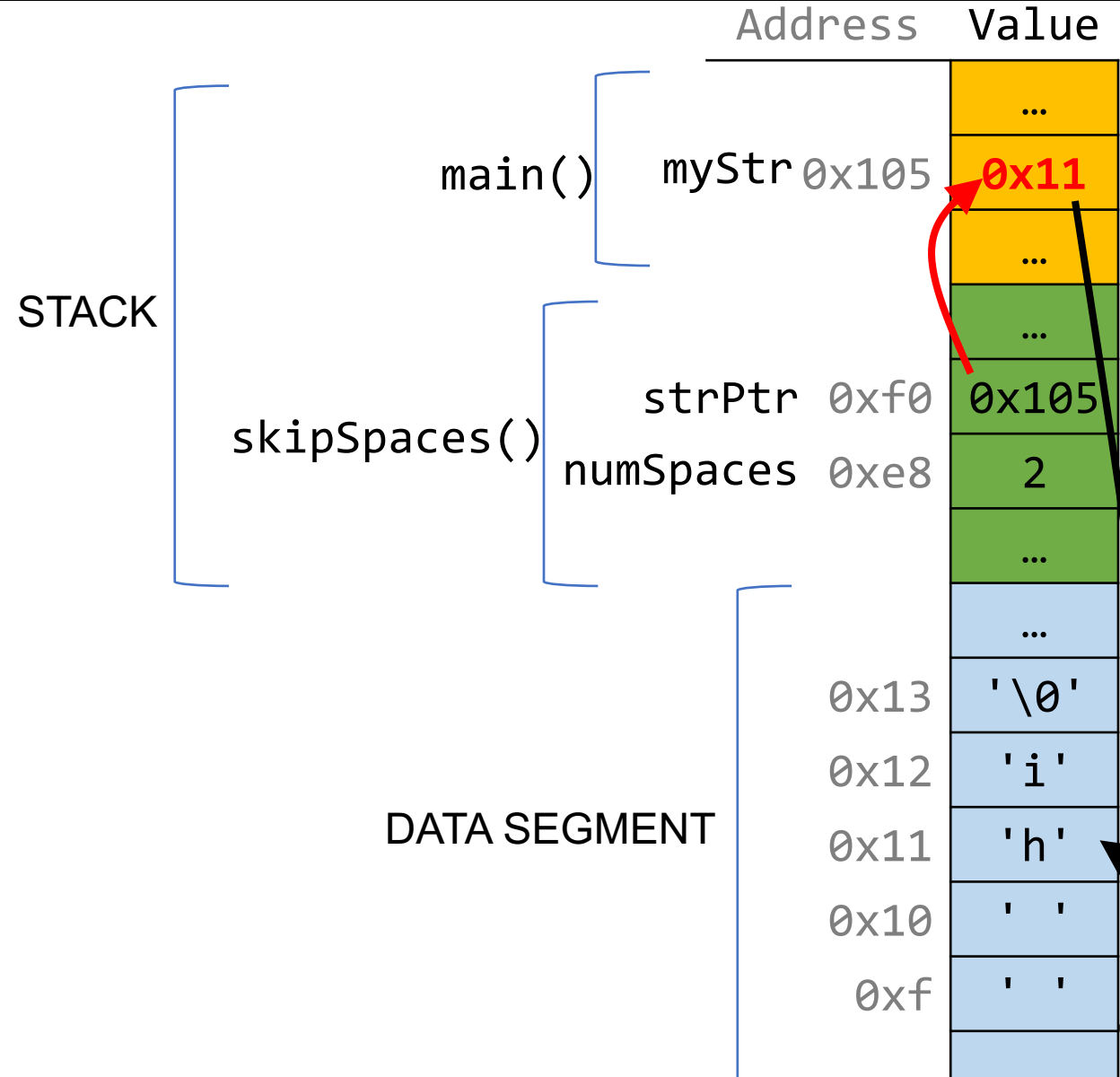
- **Recap: Pointers So Far**
- Arrays in Memory
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Pointers Summary

- If you are performing an operation with some input and do not care about any changes to the input, **pass the data type itself**.
- If you are modifying a specific instance of some value, **pass the location** of what you would like to modify.
- If a function takes an address (pointer) as a parameter, it can *go to* that address if it needs the actual value.
- Strings are one application of pointers, and they can help us understand pointers more generally.

Pointers to Strings

```
void skipSpaces(char **strPtr) {  
    int numSpaces = strspn(*strPtr, " ");  
    *strPtr += numSpaces;  
}  
  
int main(int argc, char *argv[]) {  
    char *myStr = " hi";  
    skipSpaces(&myStr);  
    printf("%s\n", myStr);           // hi  
    return 0;  
}
```



Lecture Plan

- **Recap:** Pointers So Far
- **Arrays in Memory**
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

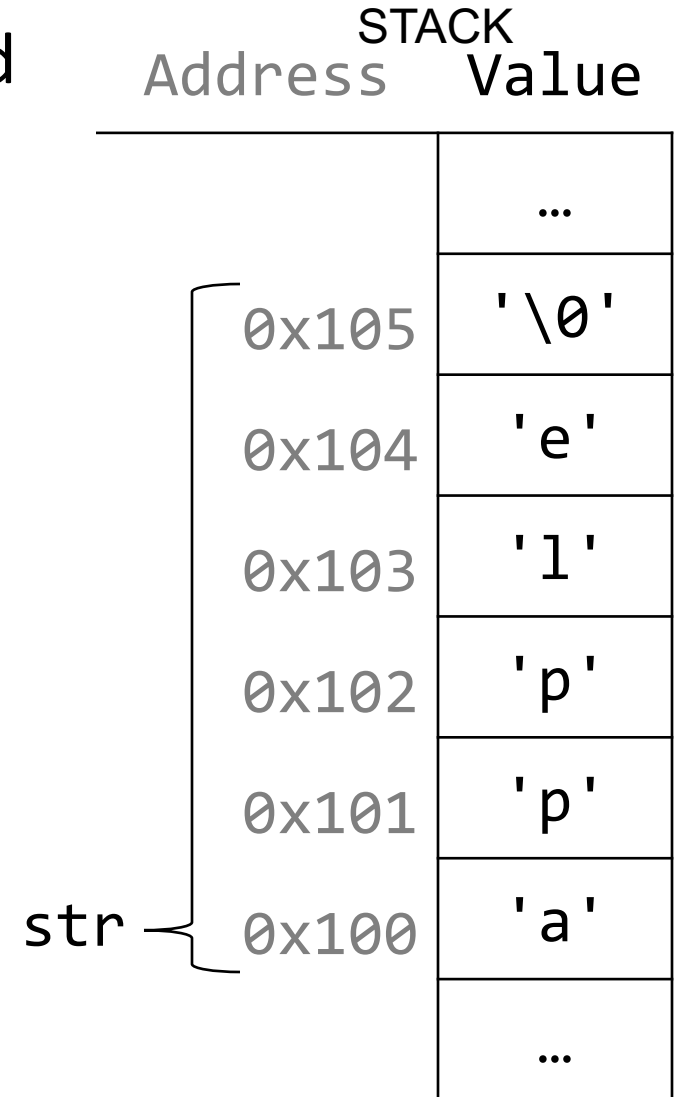
Arrays

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

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

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

```
int arrayBytes = sizeof(str); // 6
```



Arrays

An array variable refers to an entire block of memory. You cannot reassign an existing array to be equal to a new array.

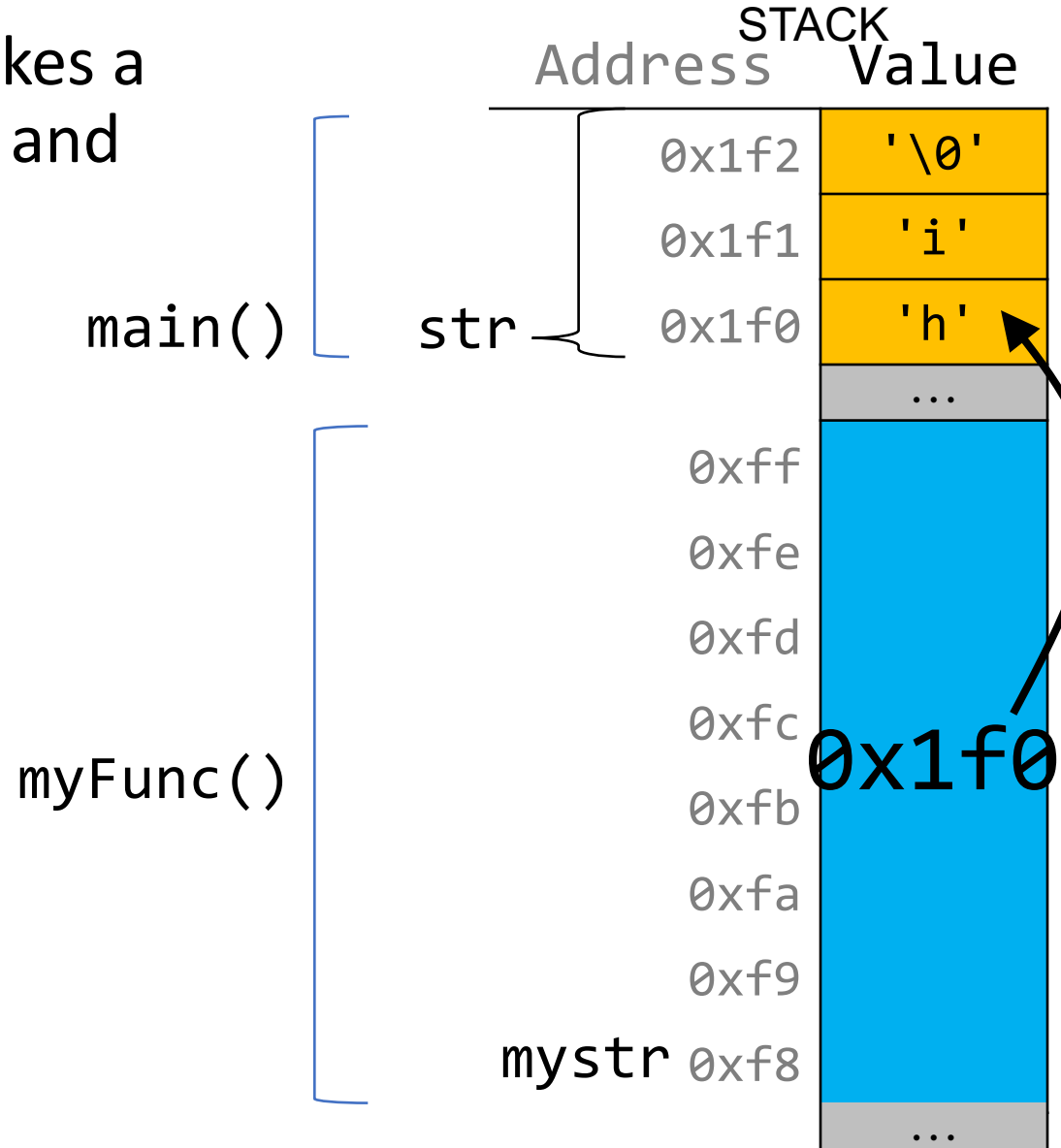
```
int nums[] = {1, 2, 3};  
int nums2[] = {4, 5, 6, 7};  
nums = nums2; // not allowed!
```

An array's size cannot be changed once you create it; you must create another new array instead.

Arrays as Parameters

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

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



Arrays as Parameters

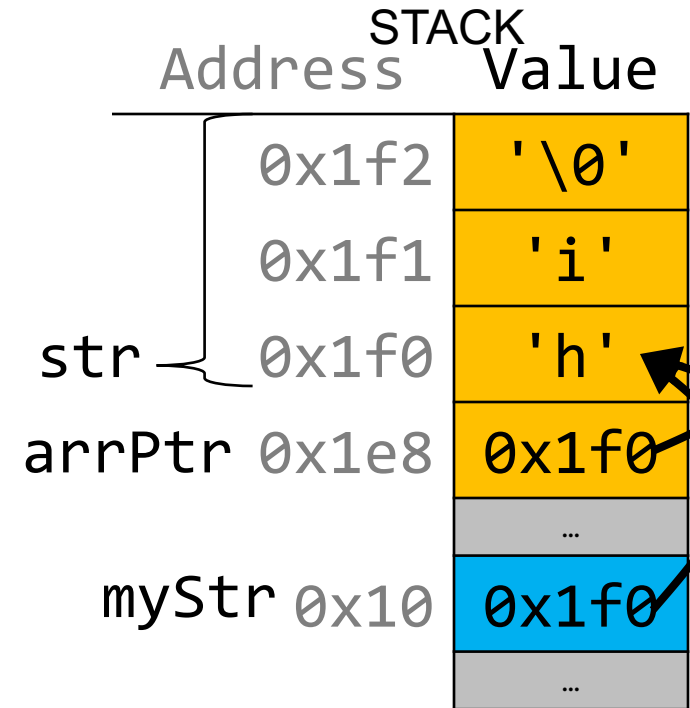
When you pass an **array** as a parameter, C makes a *copy of the address of the first array element* and passes it (a pointer) to the function.

```
void myFunc(char *myStr) {  
    ...  
}
```

```
int main(int argc, char *argv[]) {  
    char str[3];  
    strcpy(str, "hi");  
    // equivalent  
    char *arrPtr = str;  
    myFunc(arrPtr);  
    ...  
}
```

main()

myFunc()

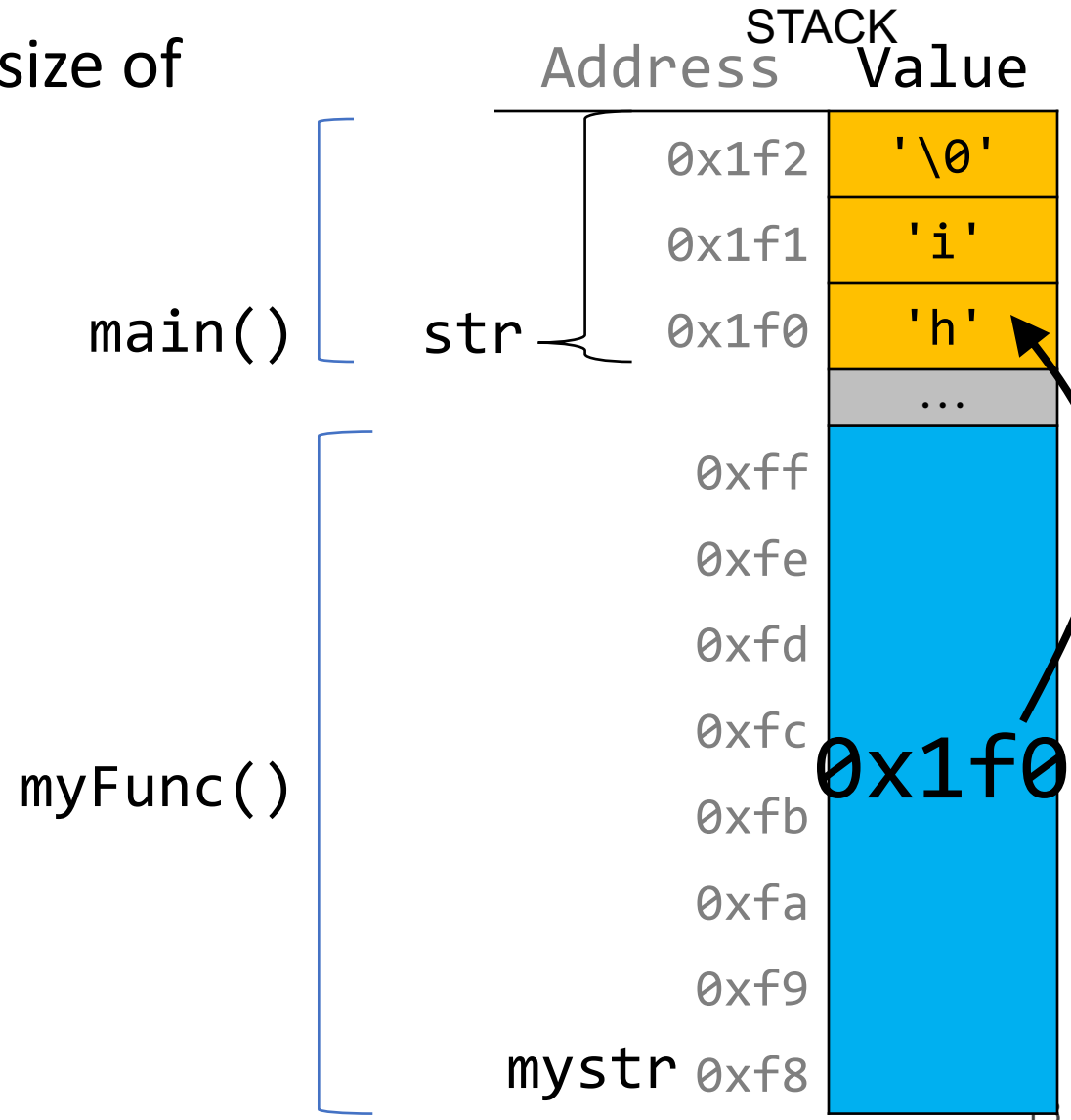


Arrays as Parameters

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

```
void myFunc(char *myStr) {  
    int size = sizeof(myStr); // 8  
}
```

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



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

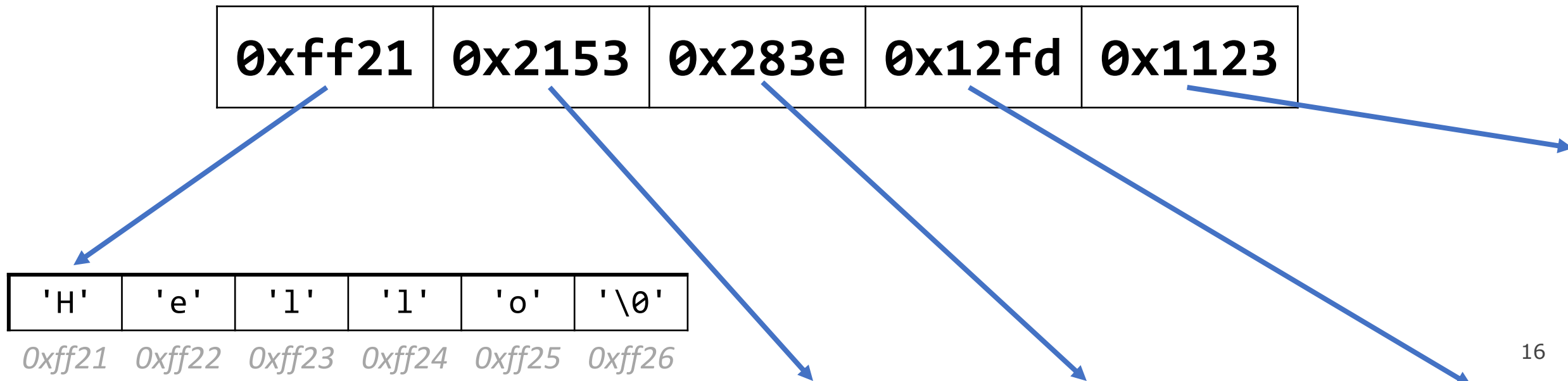
Arrays vs. Pointers

- When you create an array, you are making space for each element in the array.
- When you create a pointer, you are making space for an 8 byte address.
- Arrays "decay to pointers" when you perform arithmetic or pass as parameters.
- You can set a pointer equal to an array; that pointer will point to the array's first element
- `&arr` does nothing on arrays, but `&ptr` on pointers gets its address
- `sizeof(arr)` gets the size of an array in bytes, but `sizeof(ptr)` is always 8

Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together – this stores 5 pointers, *not* all of the characters for 5 strings!:

```
char *stringArray[5]; // space to store 5 char *s
```



Lecture Plan

- **Recap:** Pointers So Far
- Arrays in Memory
- **Pointer Arithmetic**
- The Stack
- The Heap and Dynamic Memory

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

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 *nums3 = nums + 3;   // e.g. 0xffc
int *nums2 = nums3 - 1;  // e.g. 0xff8

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

STACK

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

Pointer Arithmetic

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

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

// both of these add 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
	...

Pointer Arithmetic

How does the code know how many bytes it should look at once it visits an address? At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.

```
int x = 2;
int *xPtr = &x;           // e.g. 0xff0

// C knows 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? At compile time, C can figure out the sizes of different data types, and the sizes of what they point to.

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

```
// C knows to add 4 bytes here
```

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

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

```
strcpy(str, "CS107");
```

```
// C knows to add 1 byte here
```

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

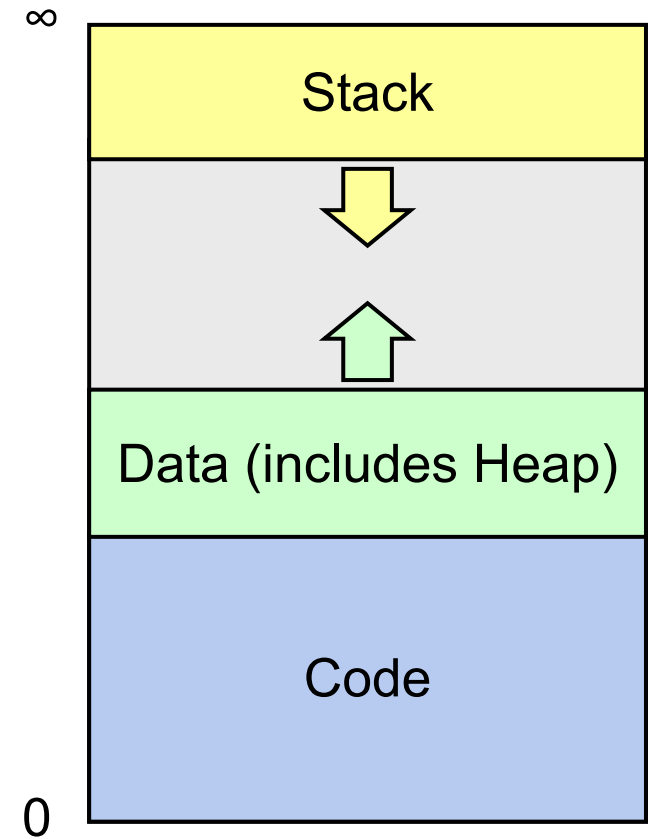

Lecture Plan

- **Recap:** Pointers So Far
- Arrays in Memory
- Pointer Arithmetic
- **The Stack**
- The Heap and Dynamic Memory

Memory Layout

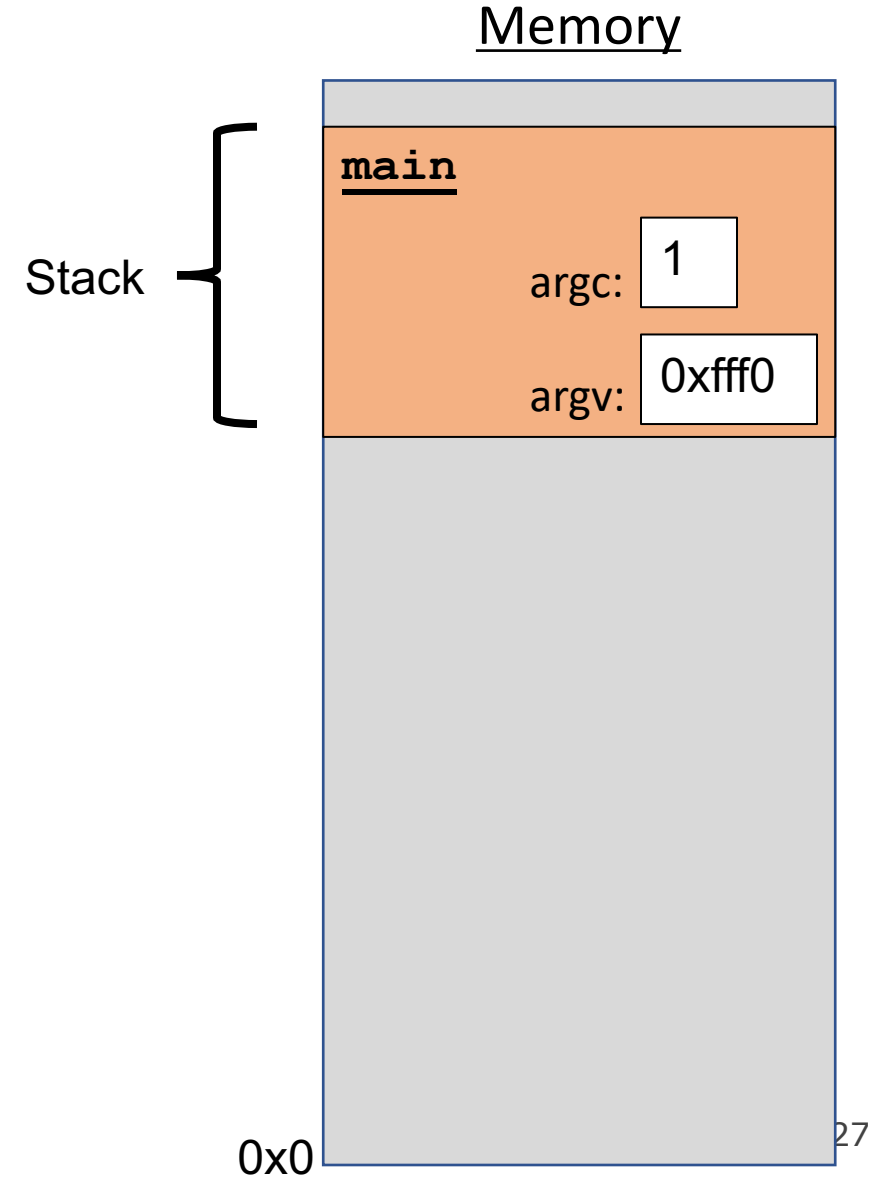
Let's dive deeper into different areas of memory used by our programs. What does memory look like?

- A process's memory is a collection of *segments* (sections)
- **Code** ("text") – program code
- **Data** – constants, heap
- **Stack** – stack frames for functions (local variables, parameters). A function's stack frame goes away when the function returns, and the stack shrinks upwards.
- Stack grows down, heap grows up as more space is needed



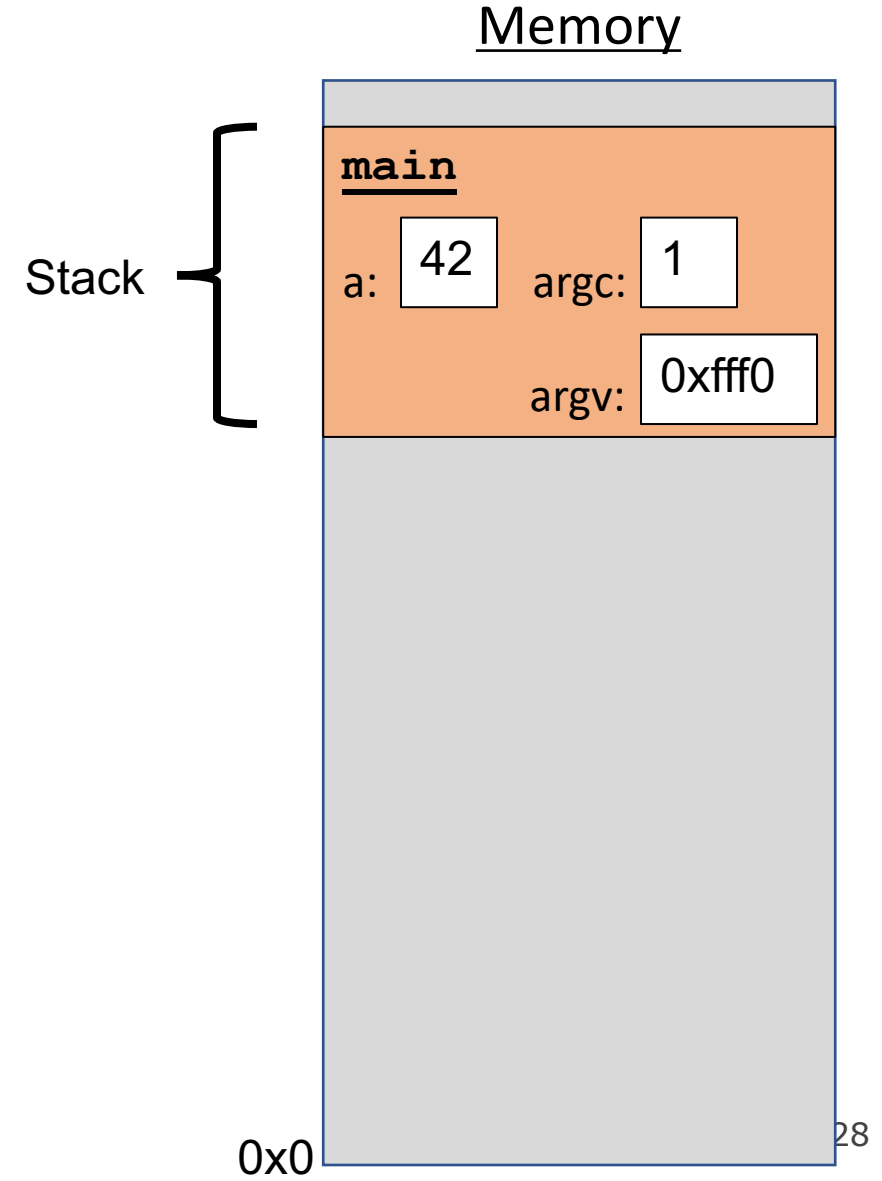
The Stack

```
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;  
}
```



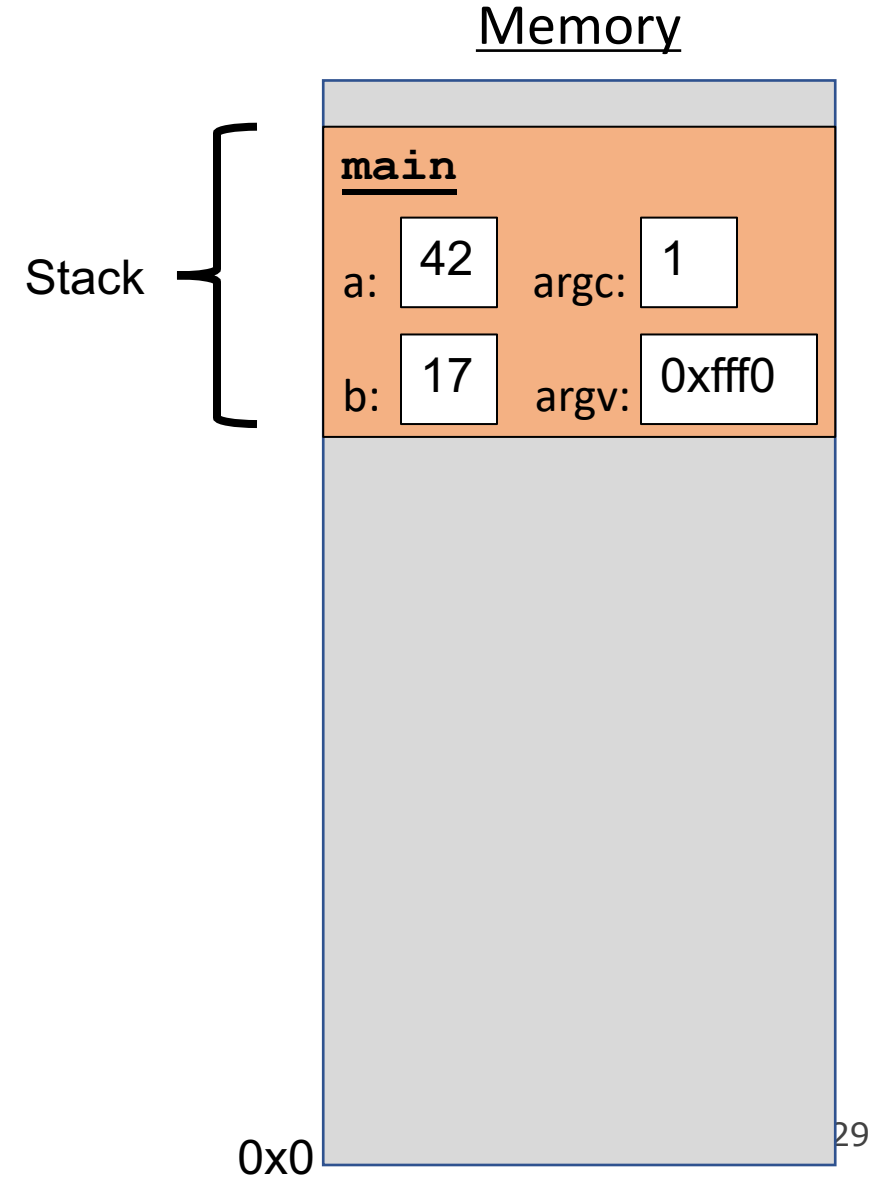
The Stack

```
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;  
}
```



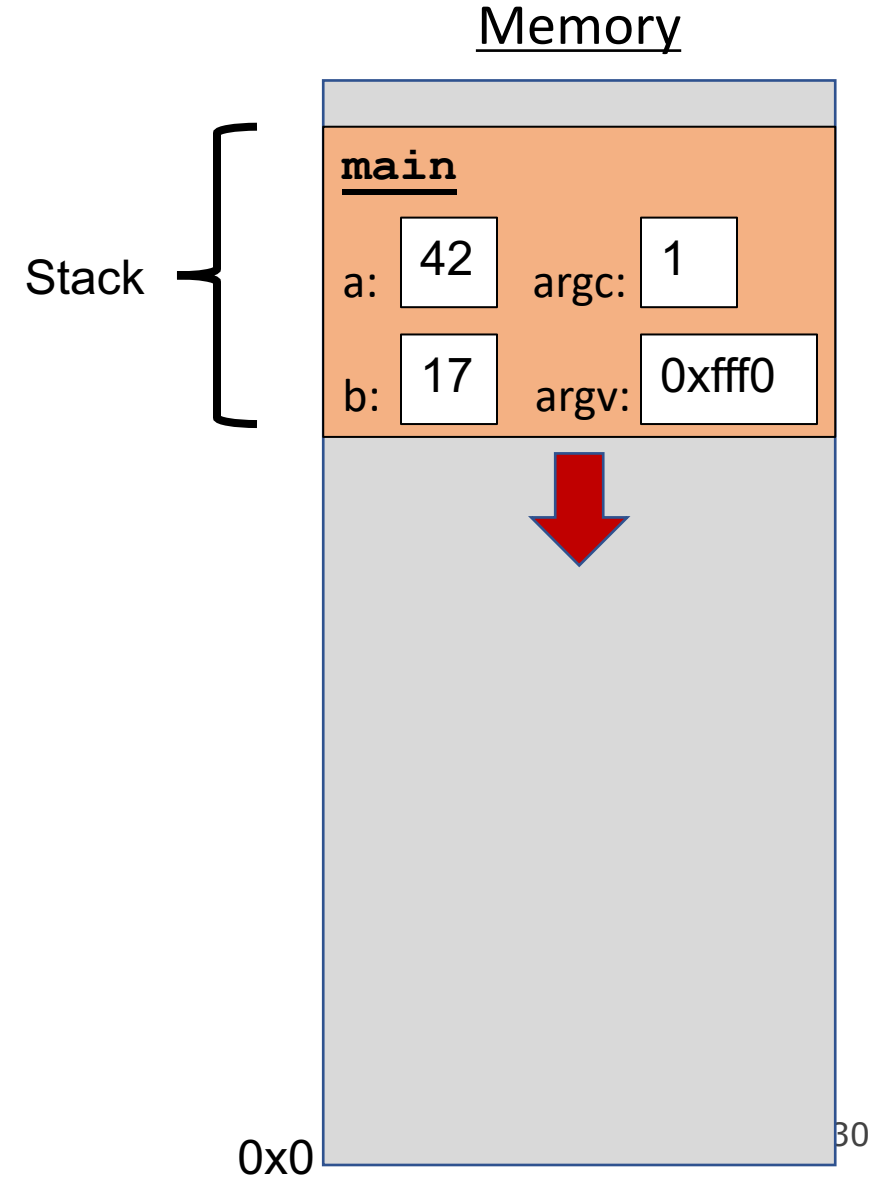
The Stack

```
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;  
}
```



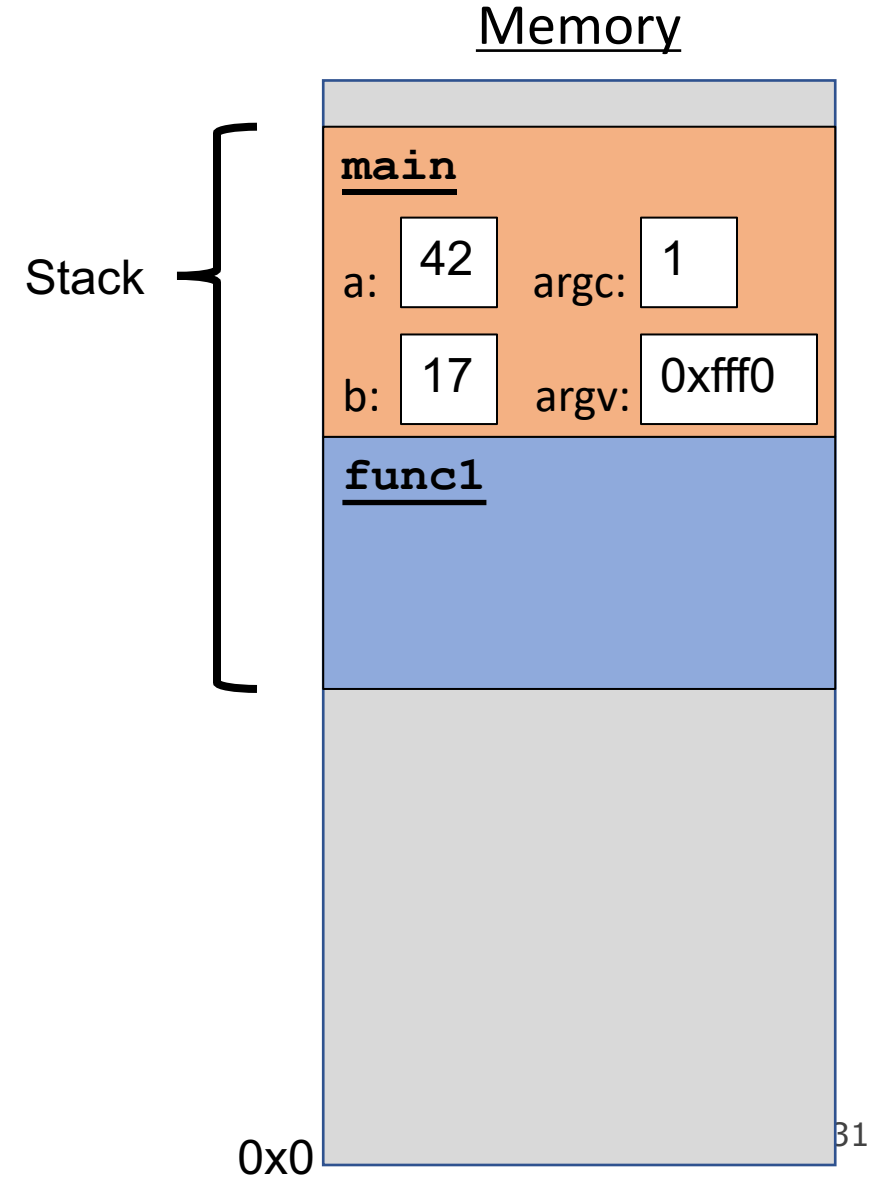
The Stack

```
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;  
}
```



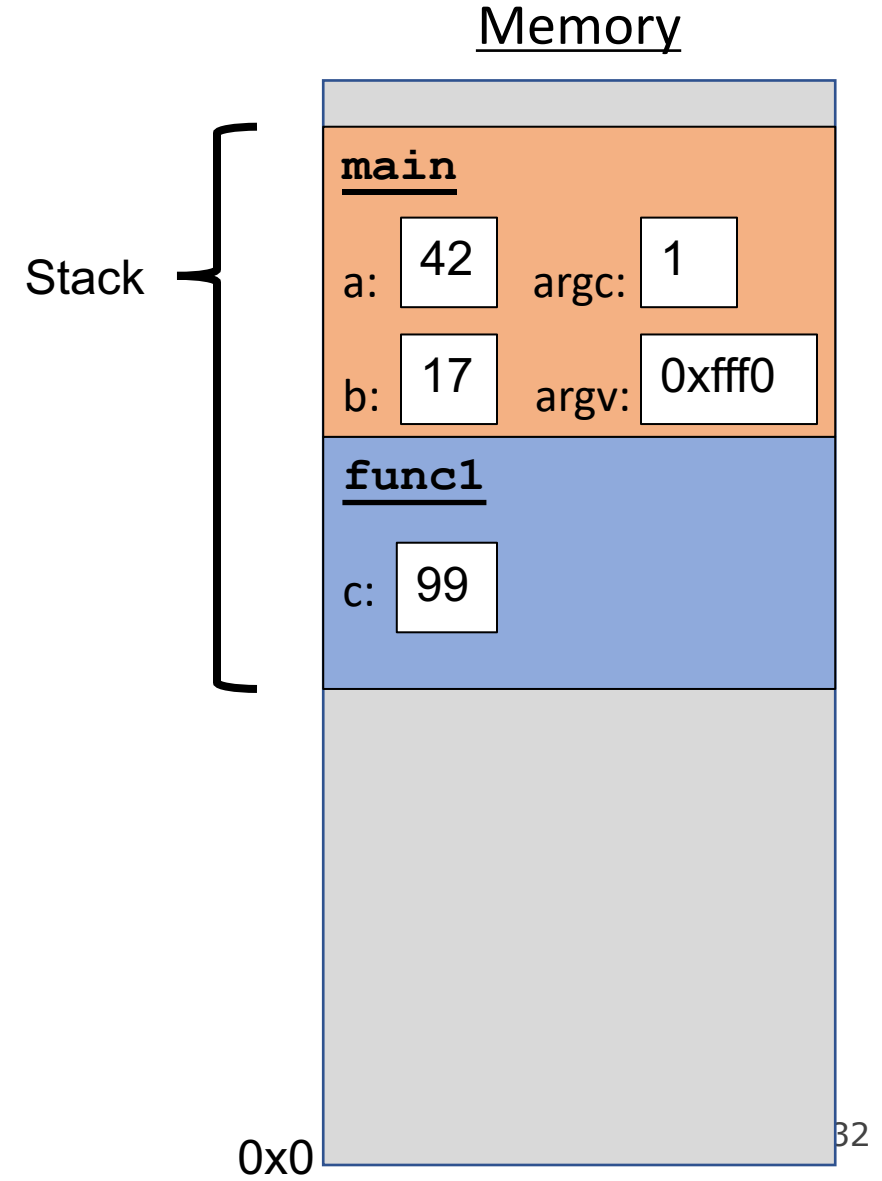
The Stack

```
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;  
}
```



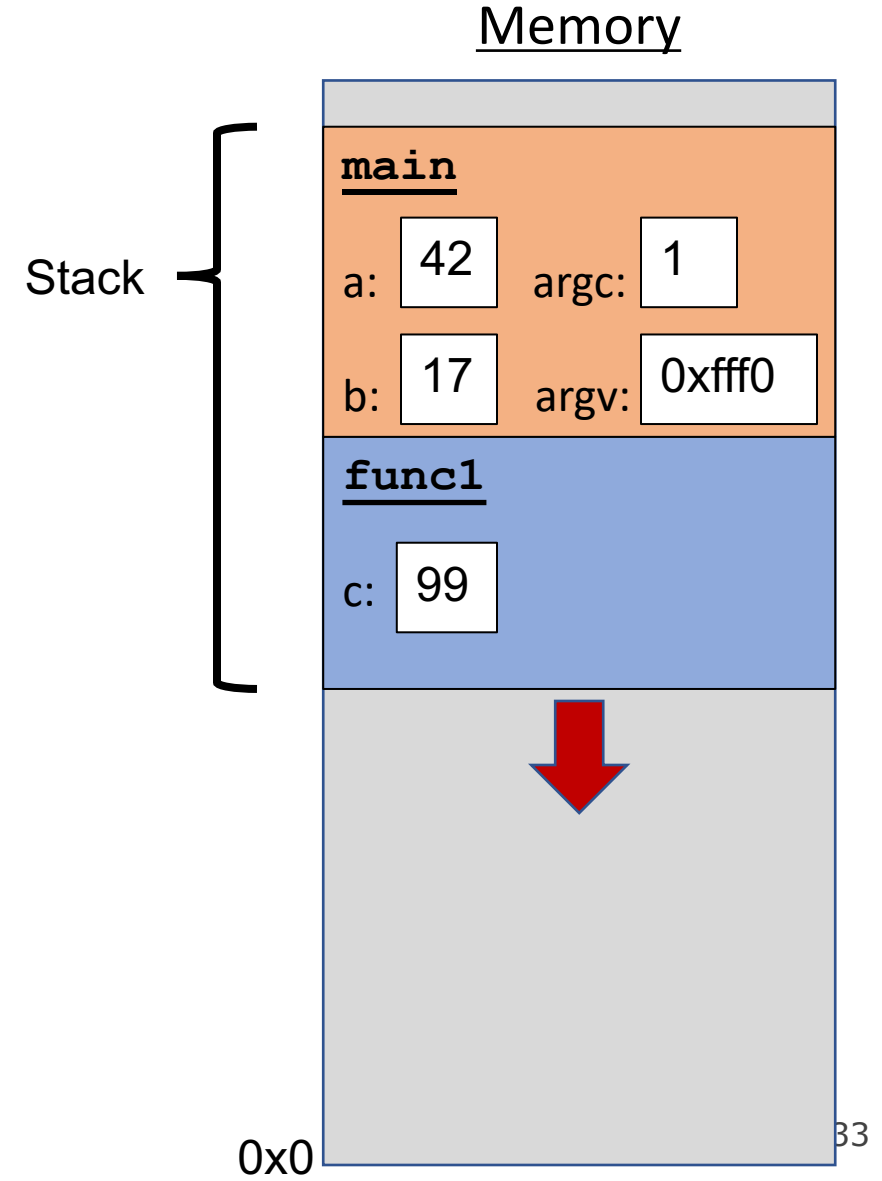
The Stack

```
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;  
}
```



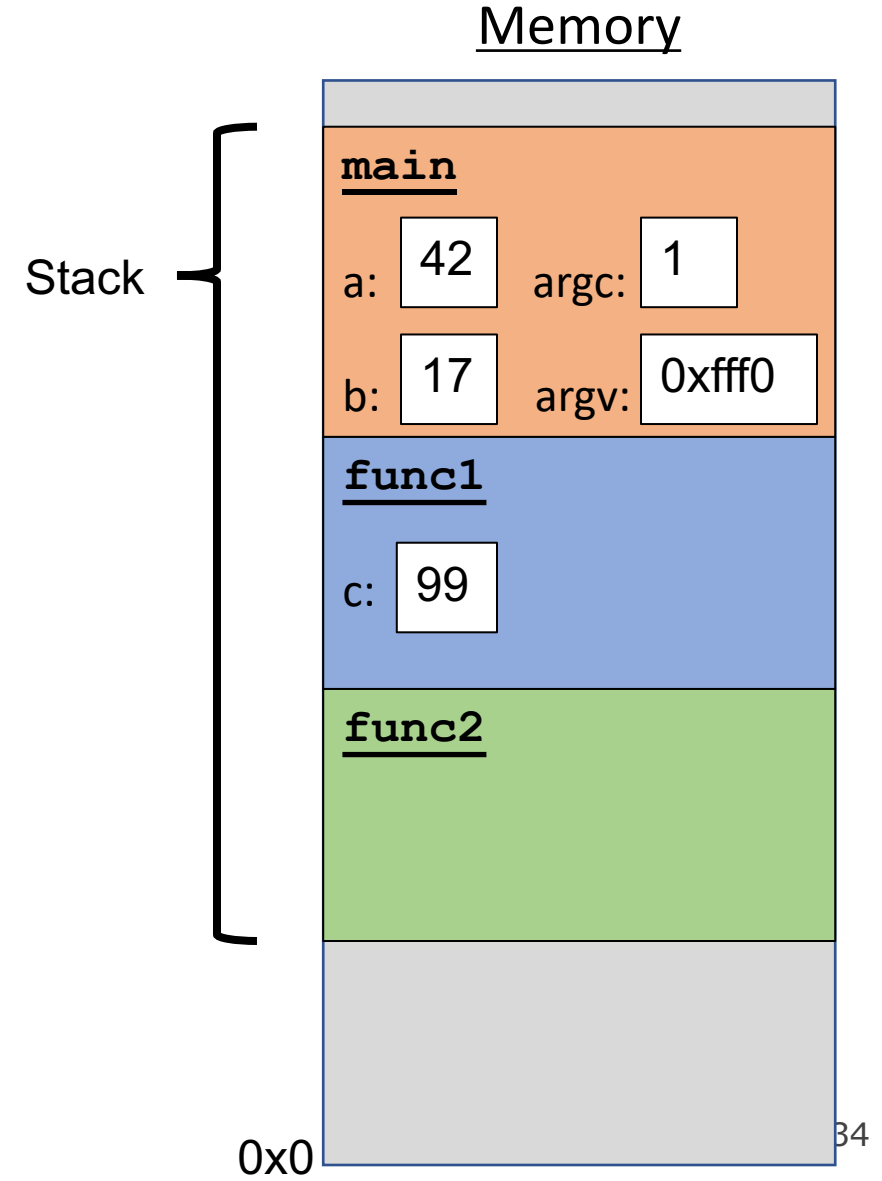
The Stack

```
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;  
}
```



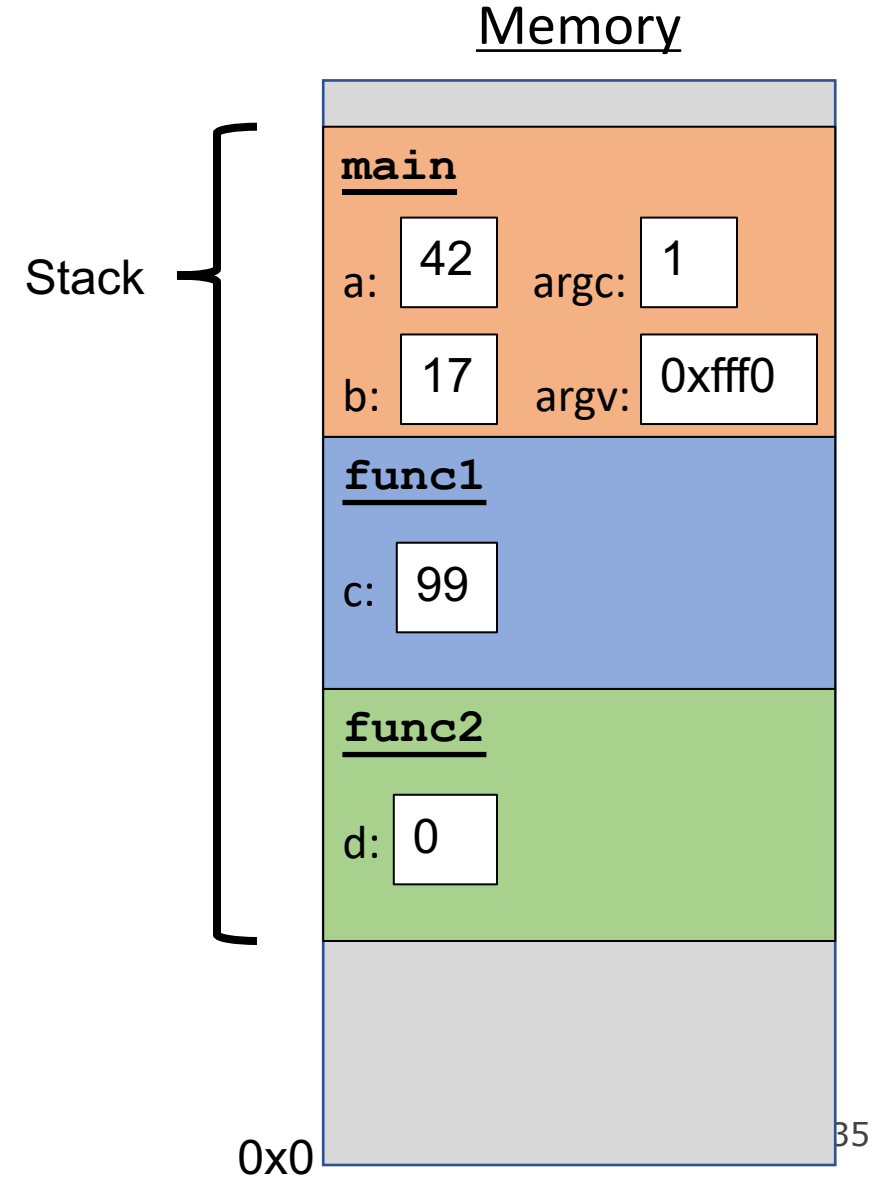
The Stack

```
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;  
}
```



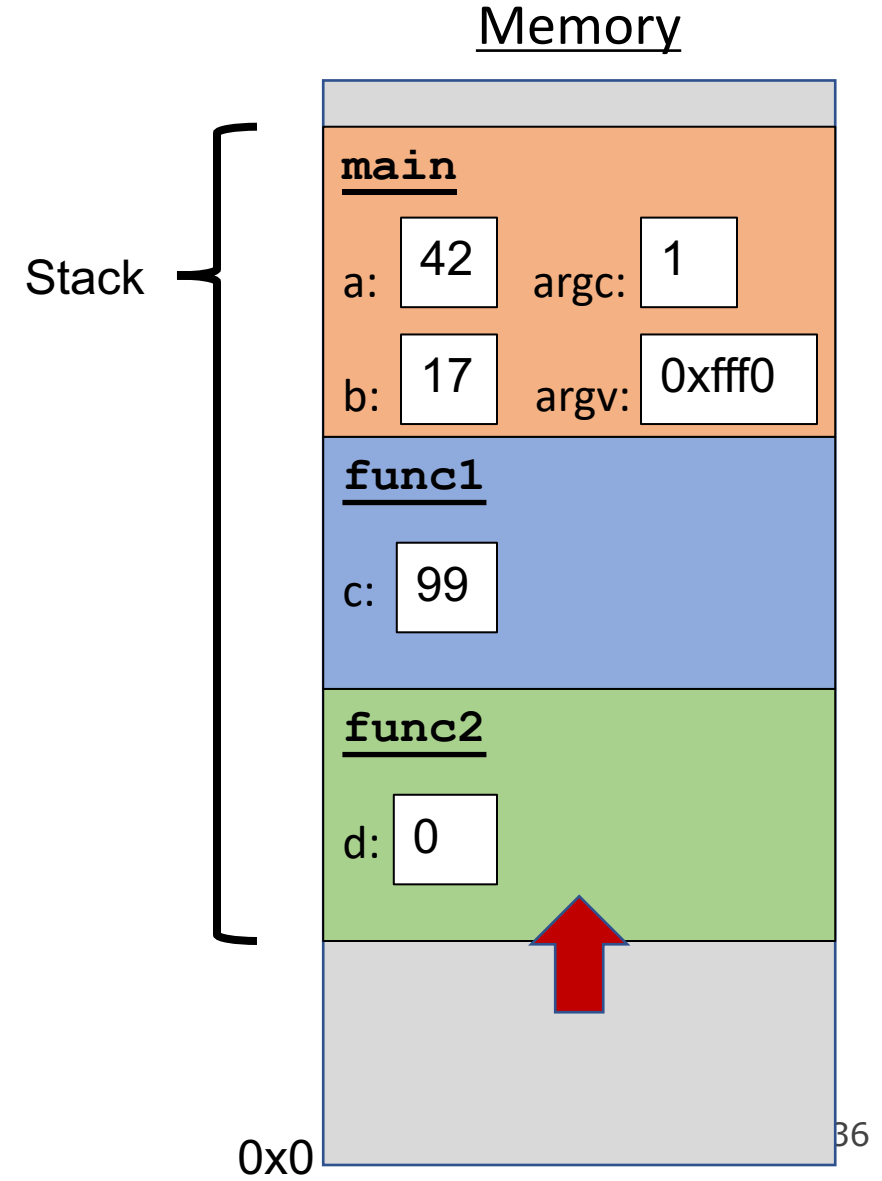
The Stack

```
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;  
}
```



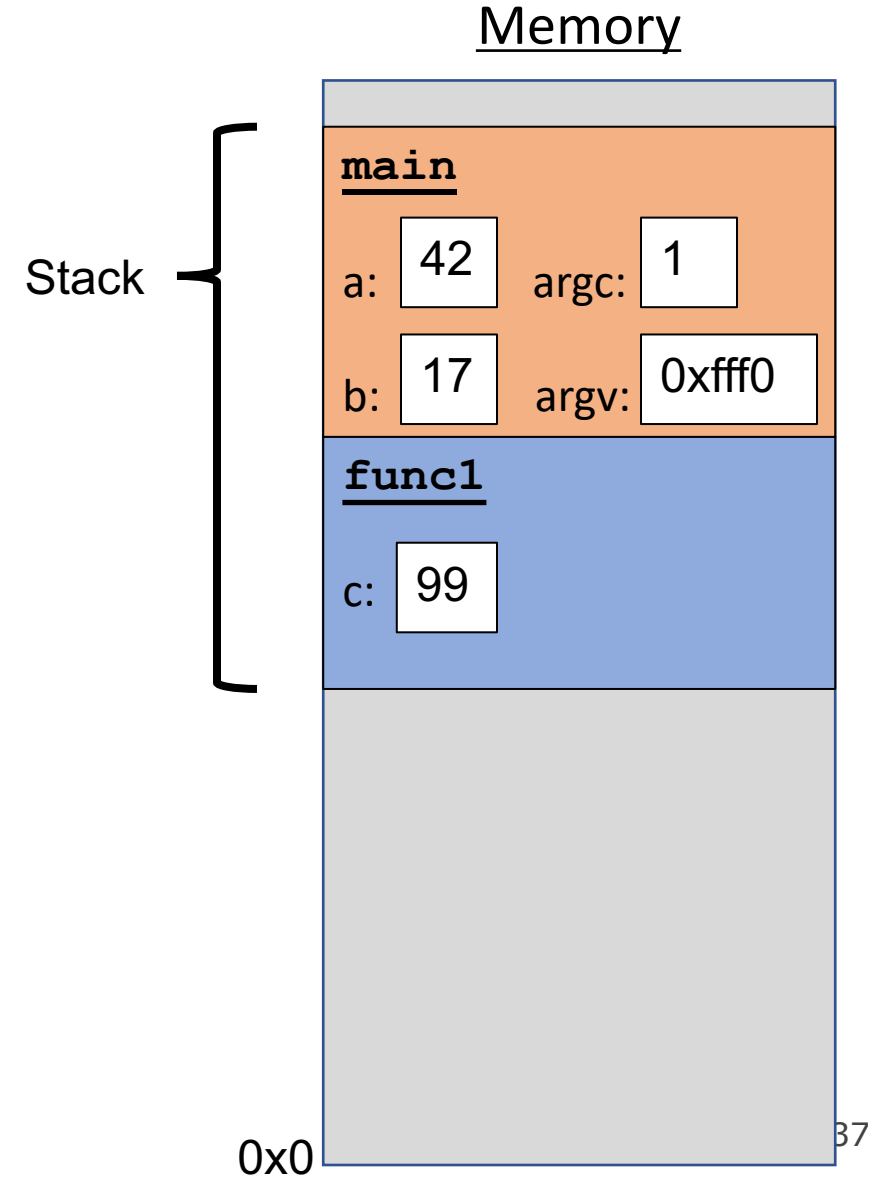
The Stack

```
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;  
}
```



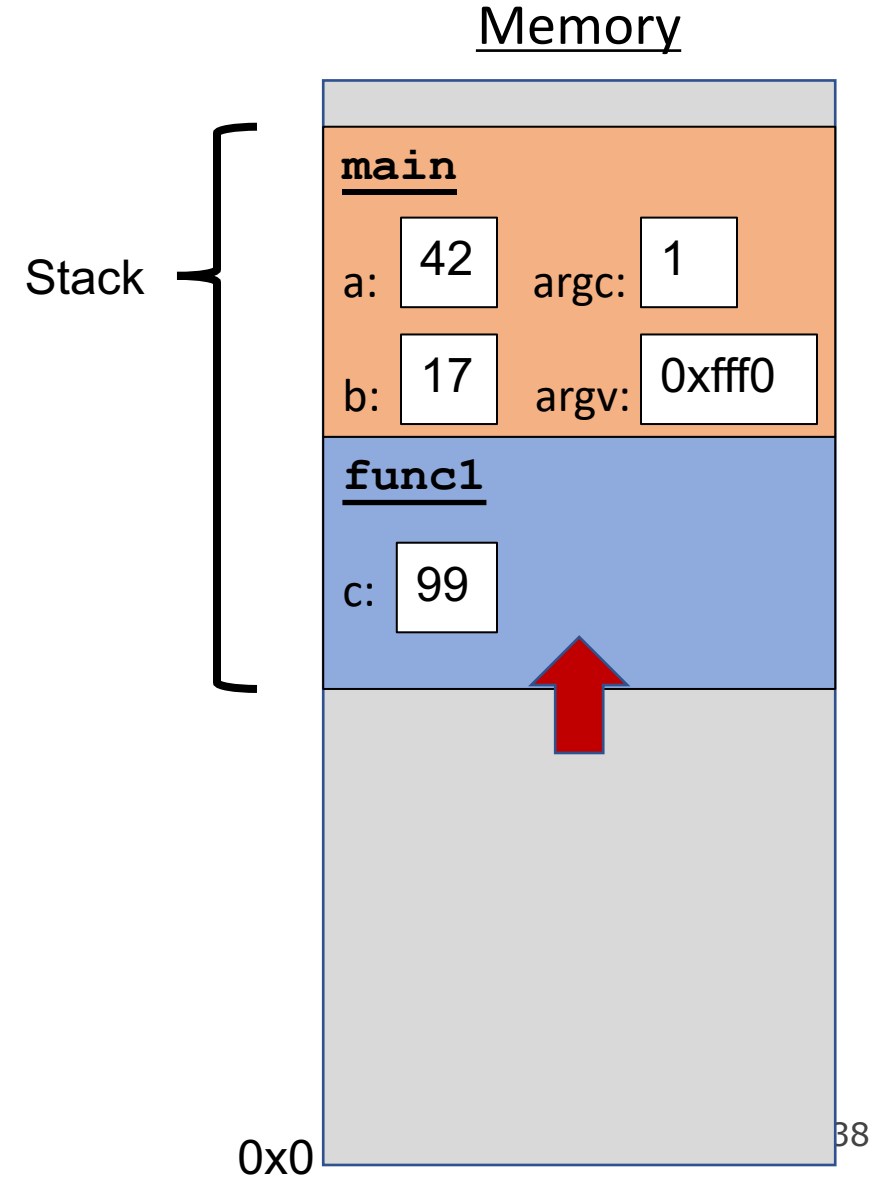
The Stack

```
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;  
}
```



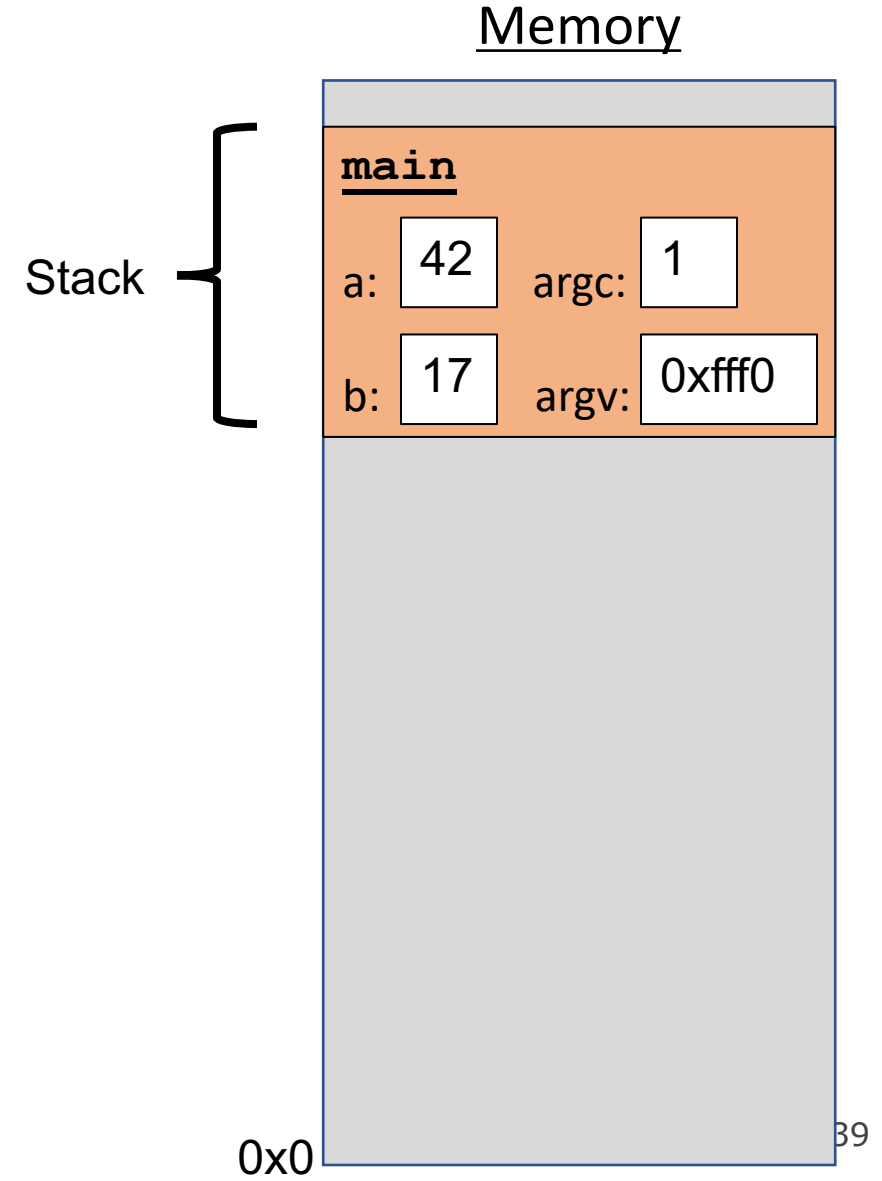
The Stack

```
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;  
}
```



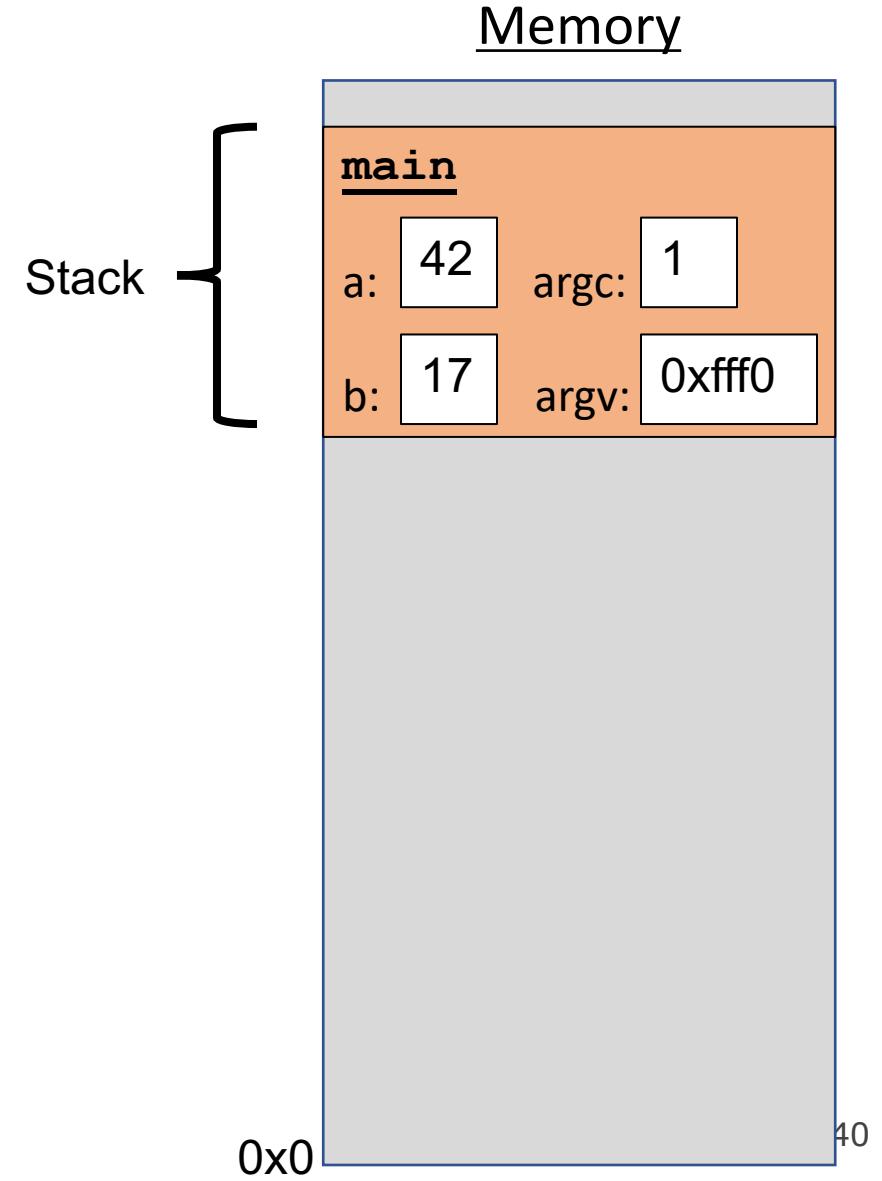
The Stack

```
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;  
}
```



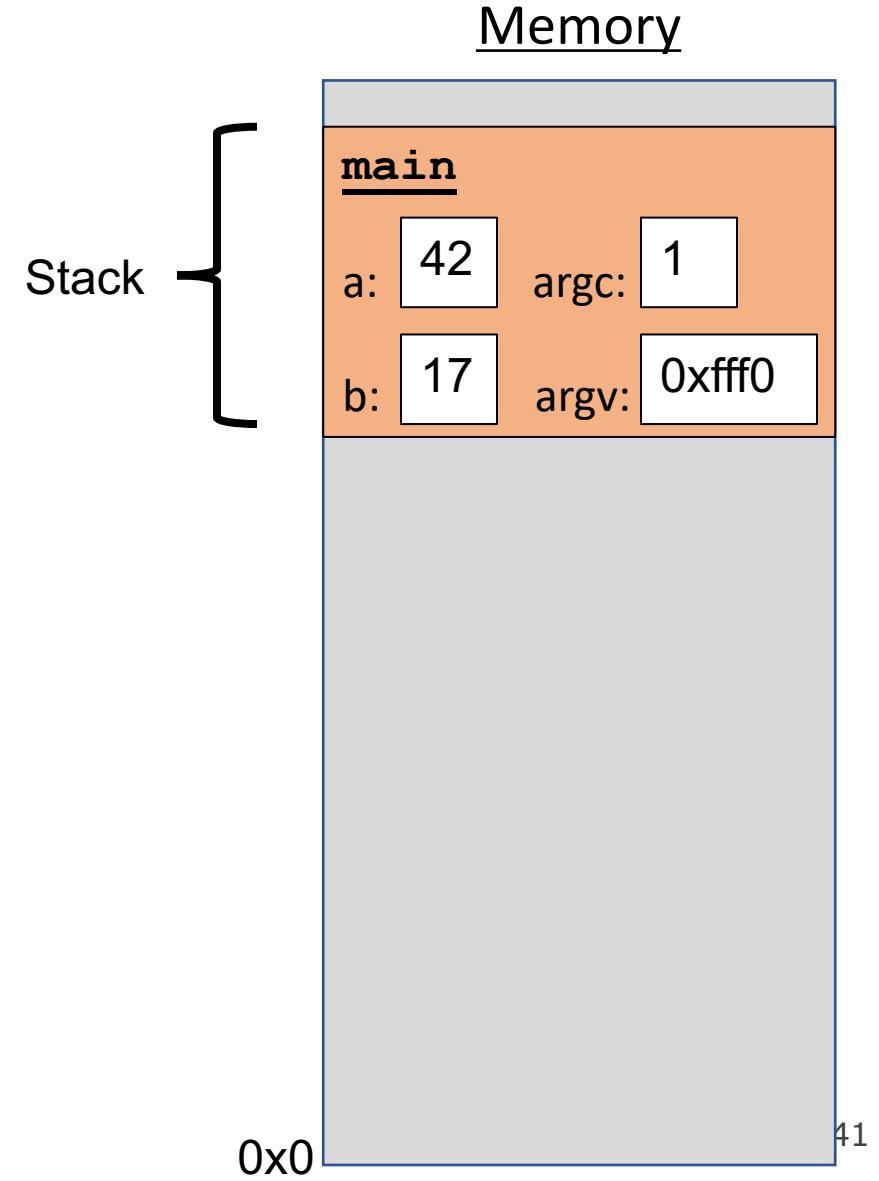
The Stack

```
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;  
}
```



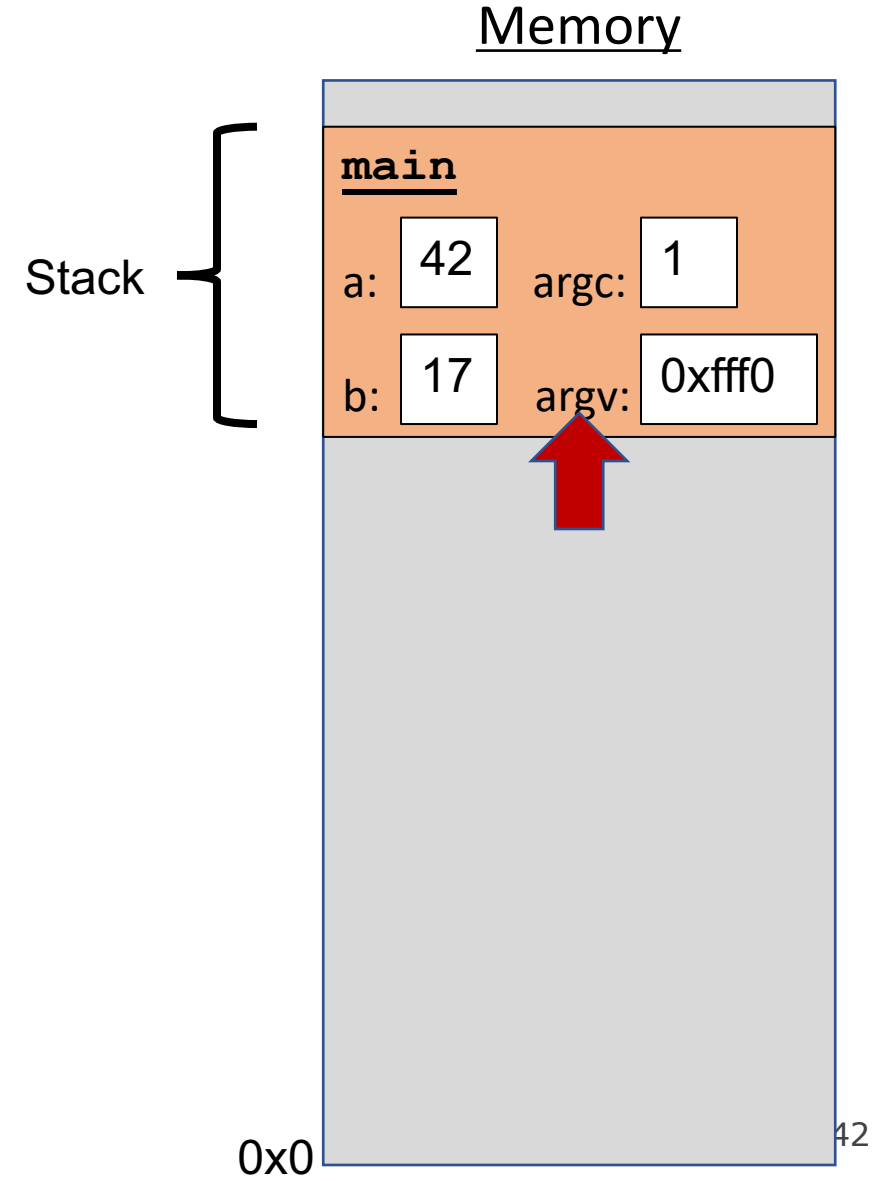
The Stack

```
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

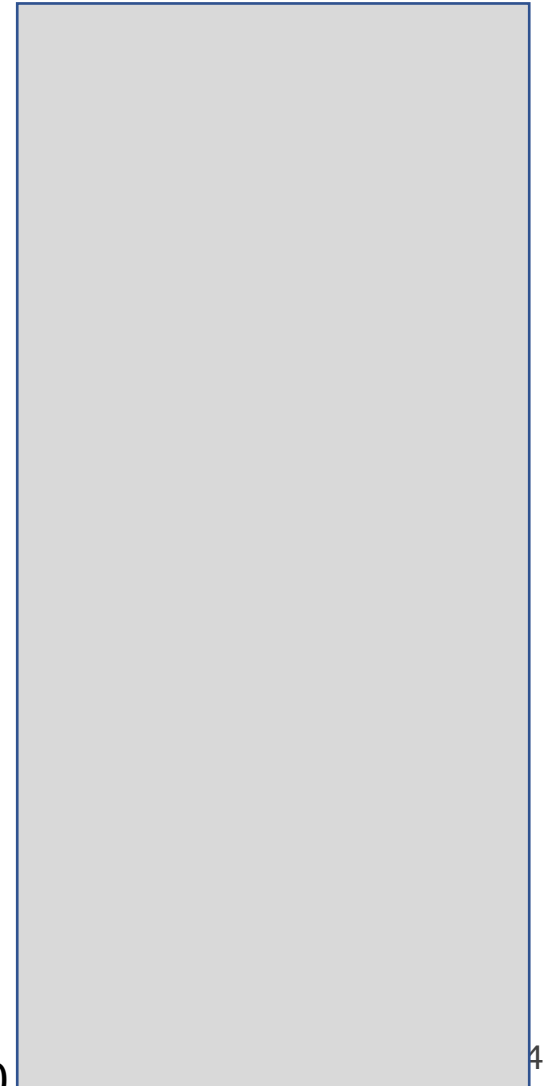
```
void func2() {  
    int d = 0;  
}  
  
void func1() {  
    int c = 99;  
    func2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    func1();  
    printf("Done.");  
    return 0;  
}
```



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



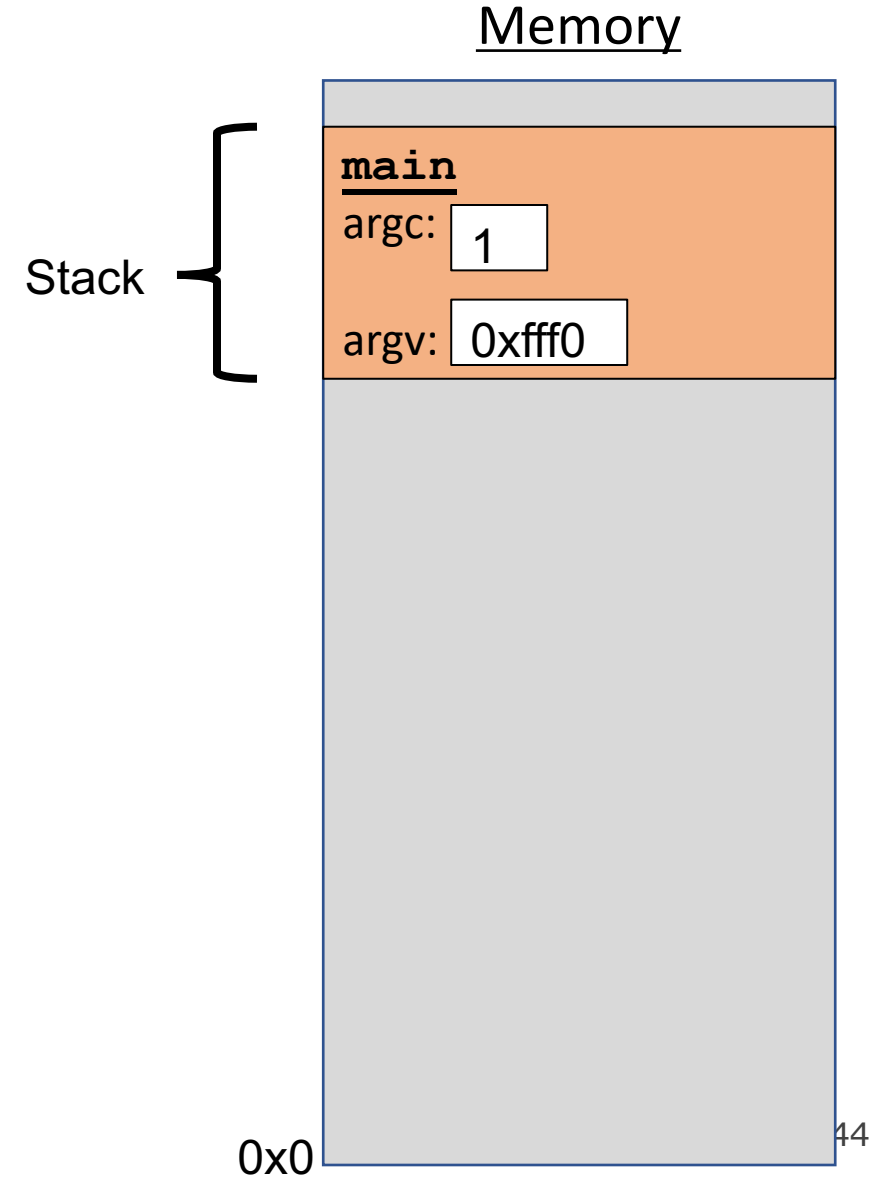
0x0

43

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;  
}
```

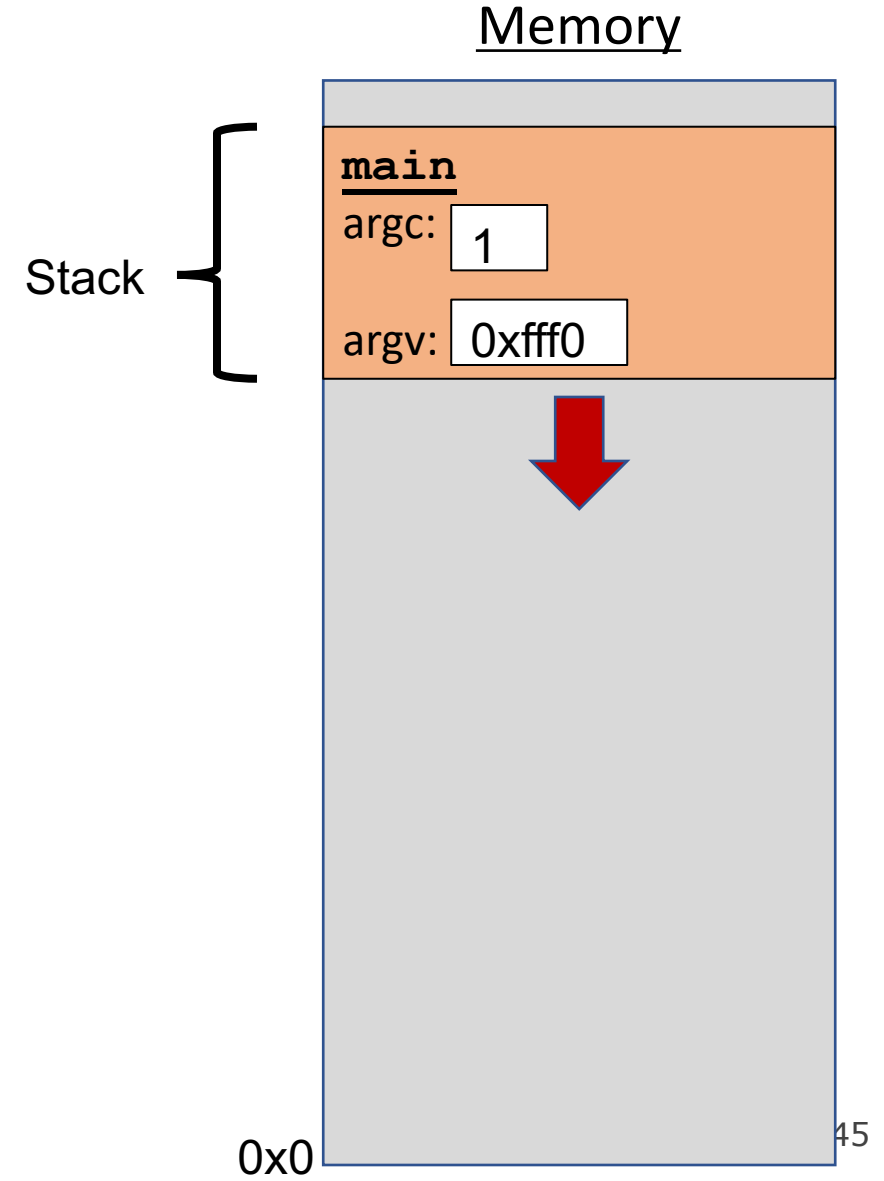


The Stack

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);
    }
}

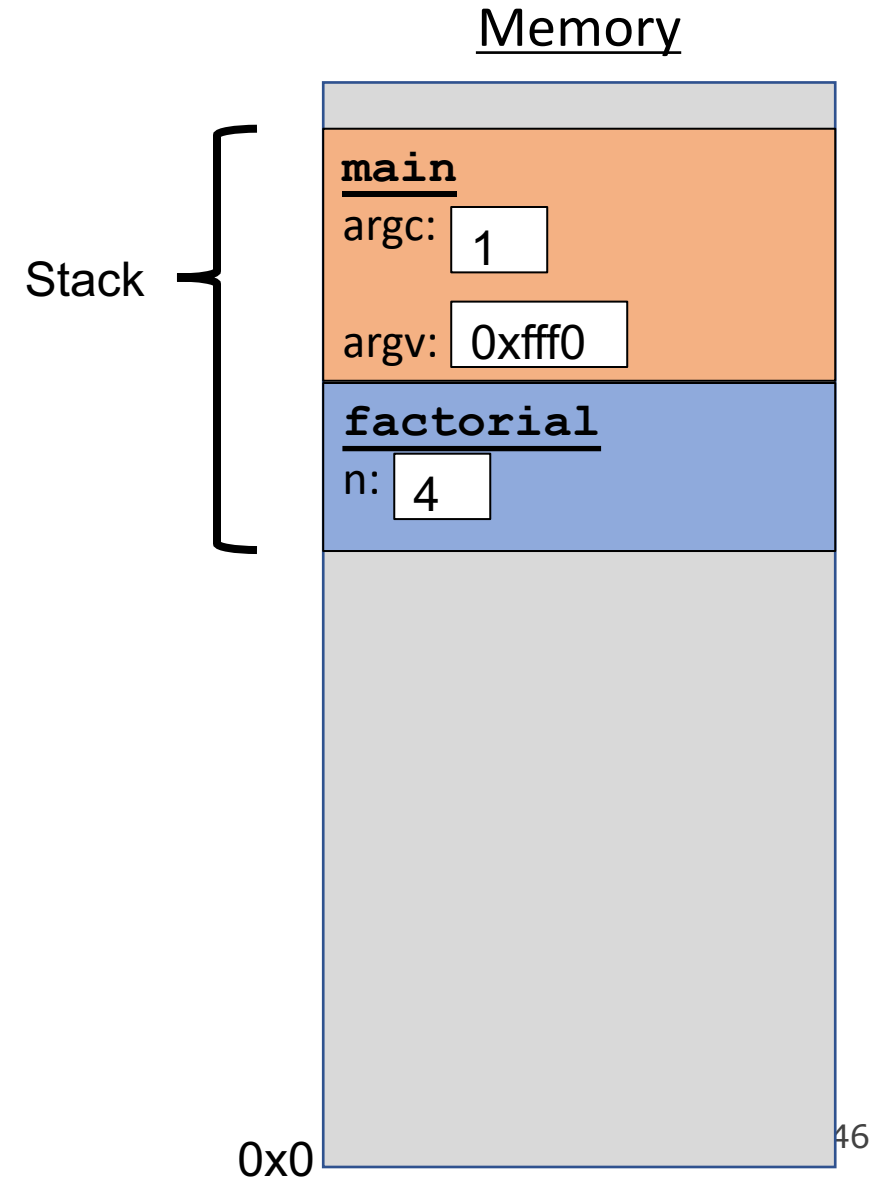
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

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

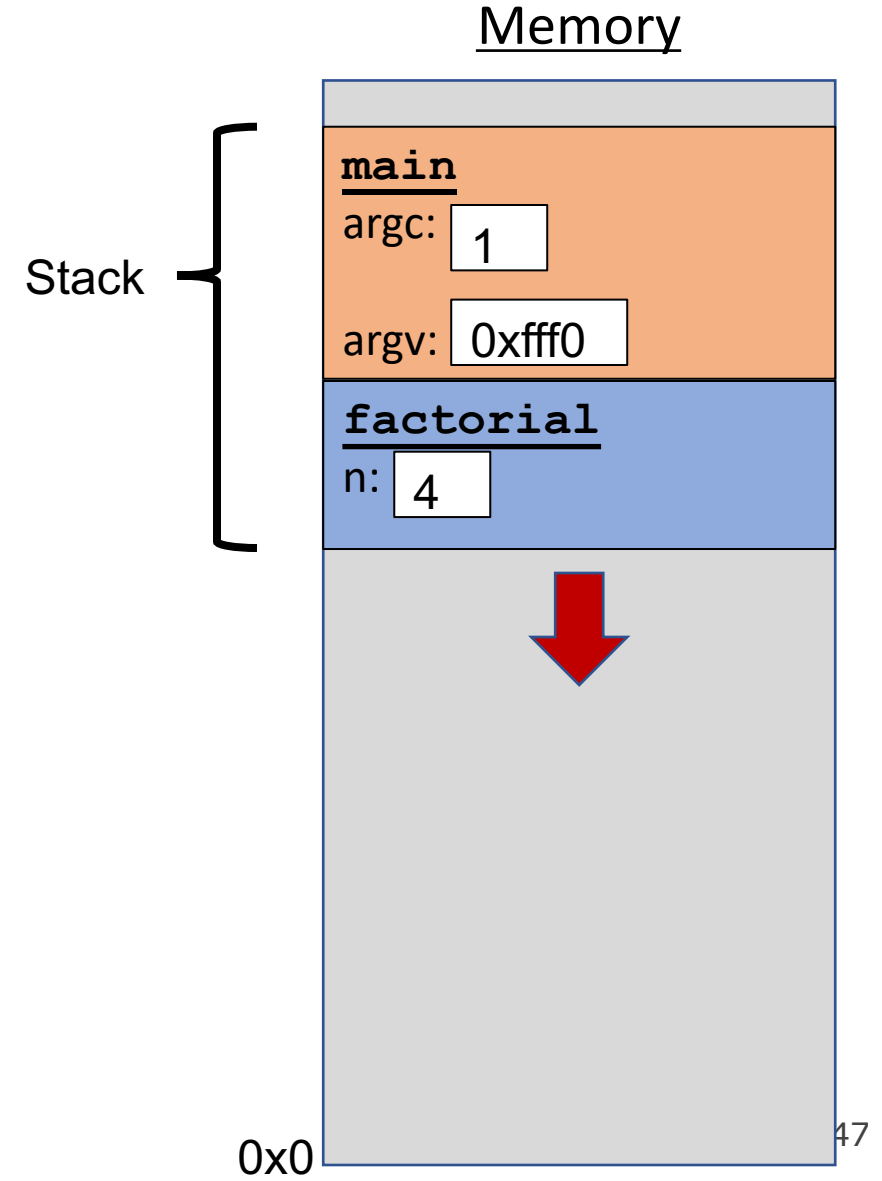
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

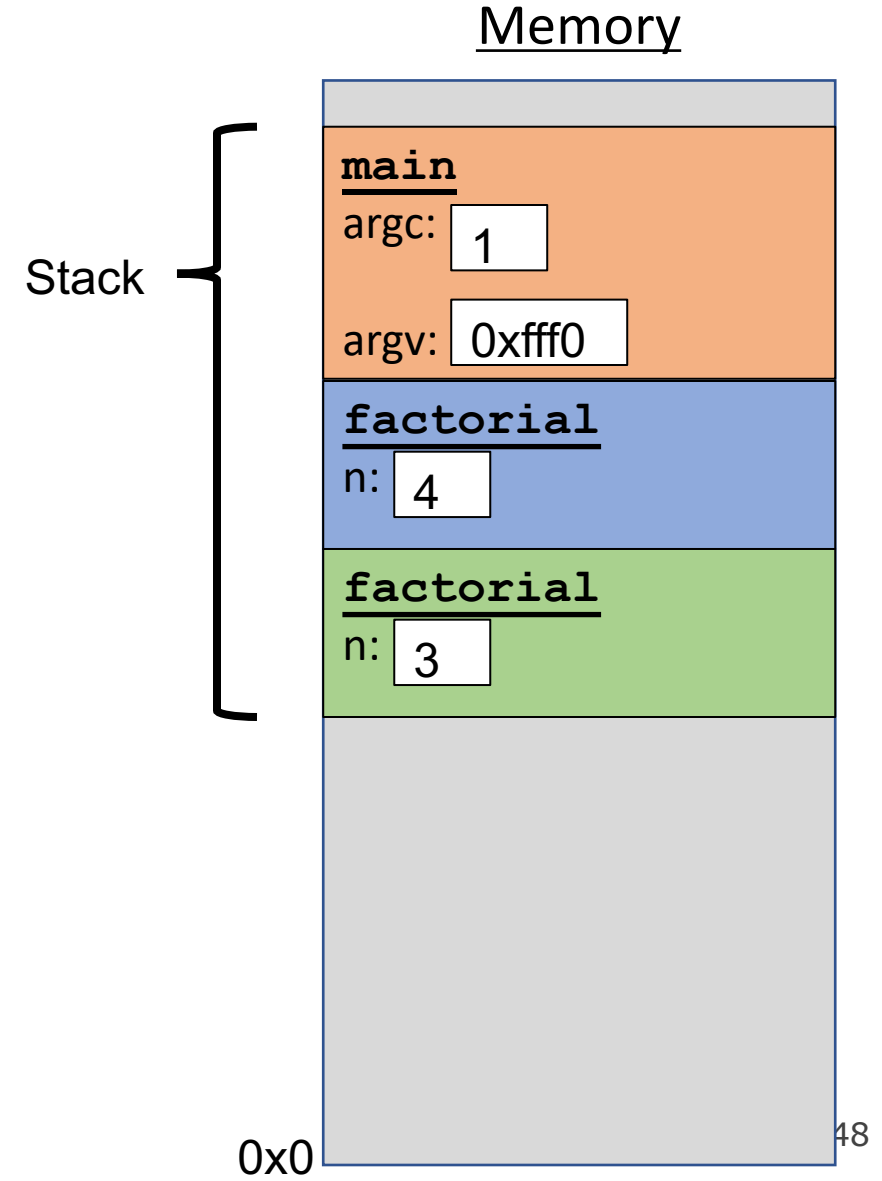
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

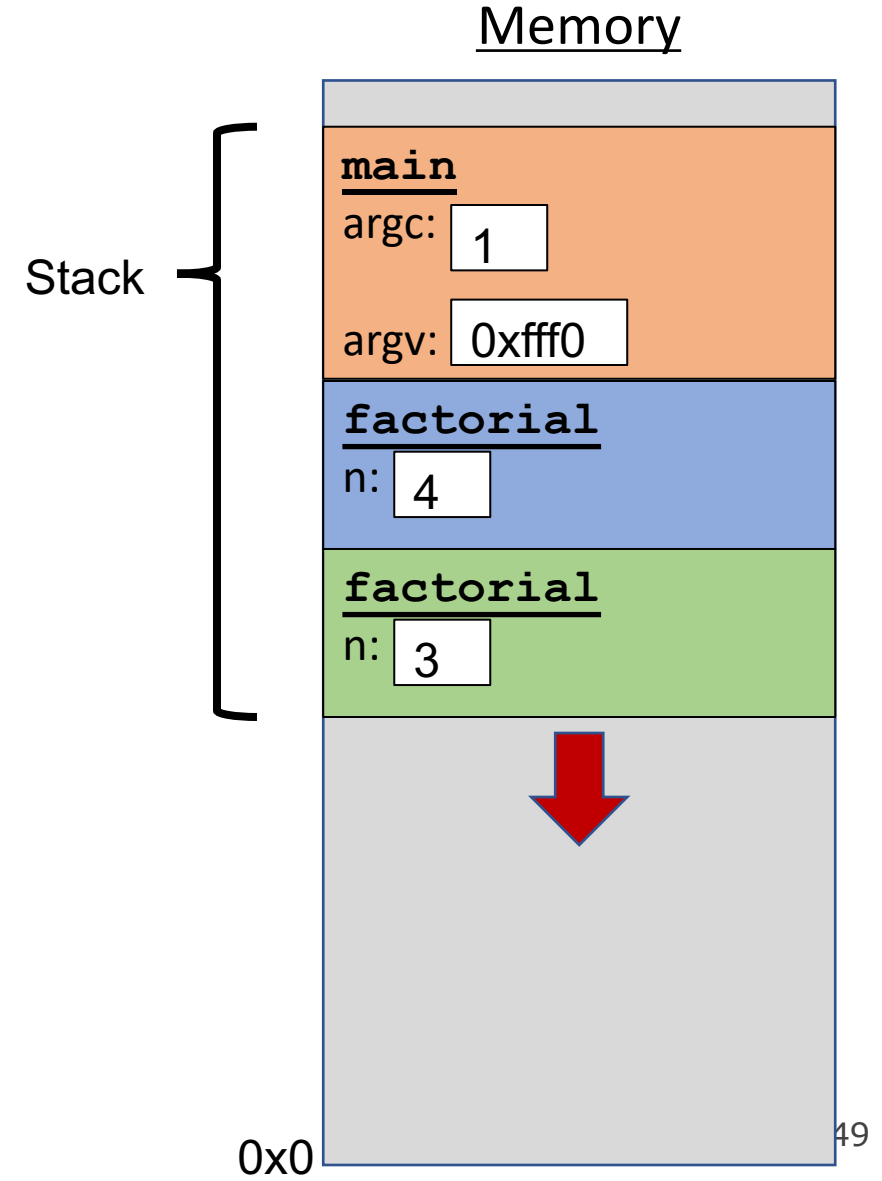
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

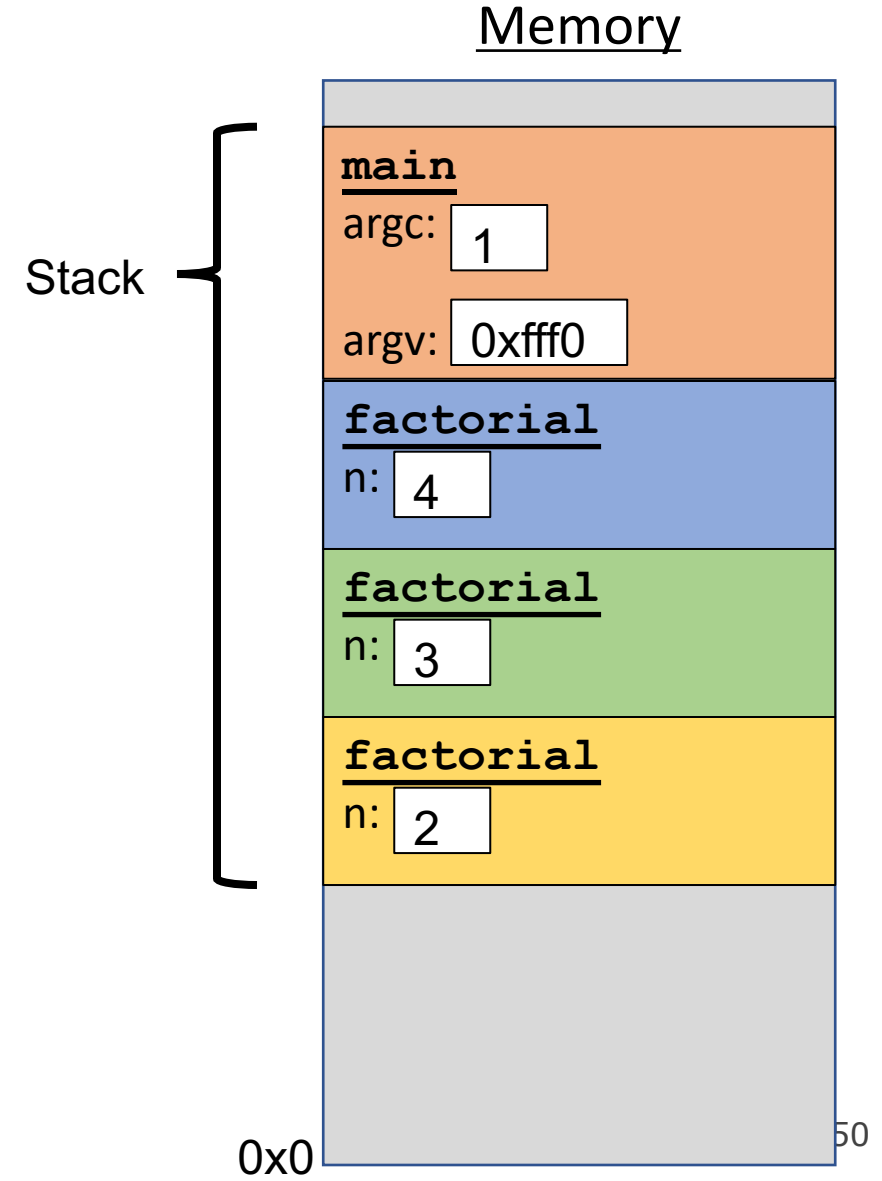
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

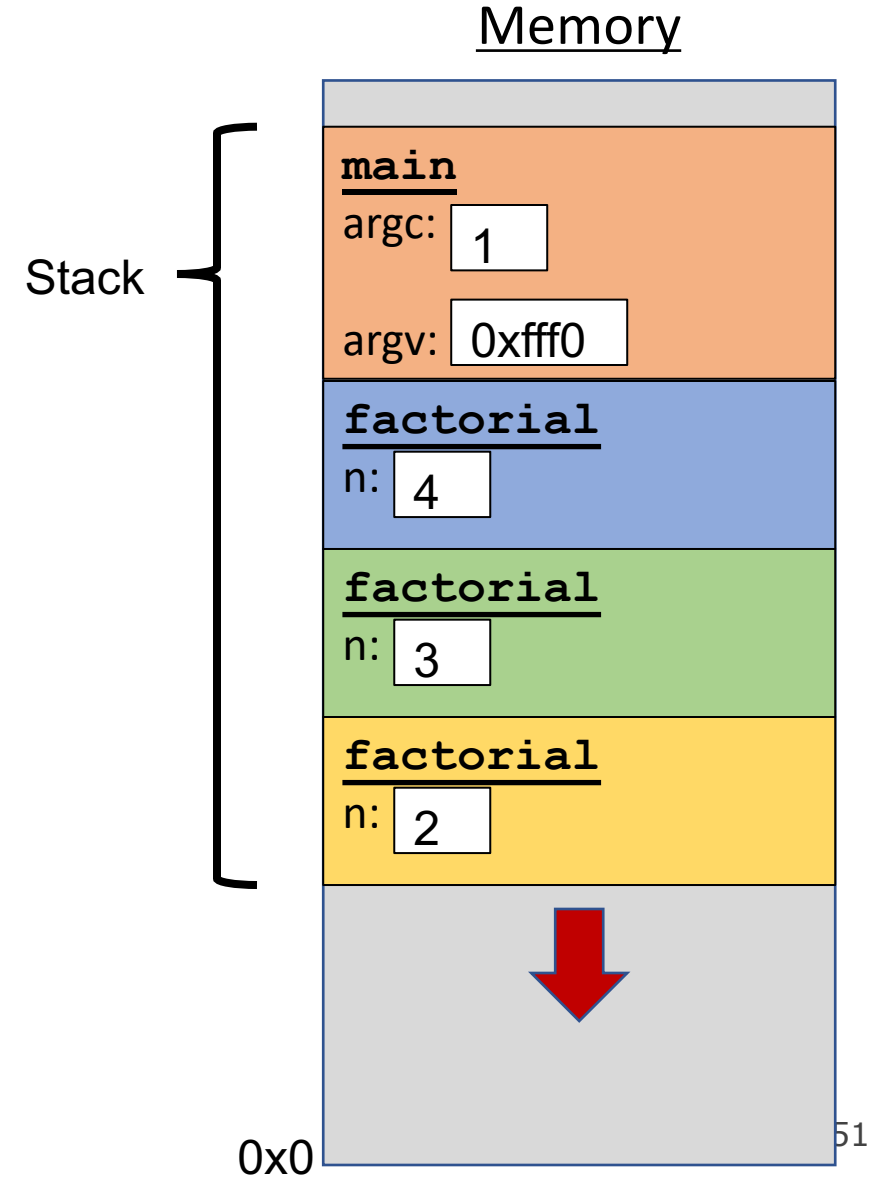
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

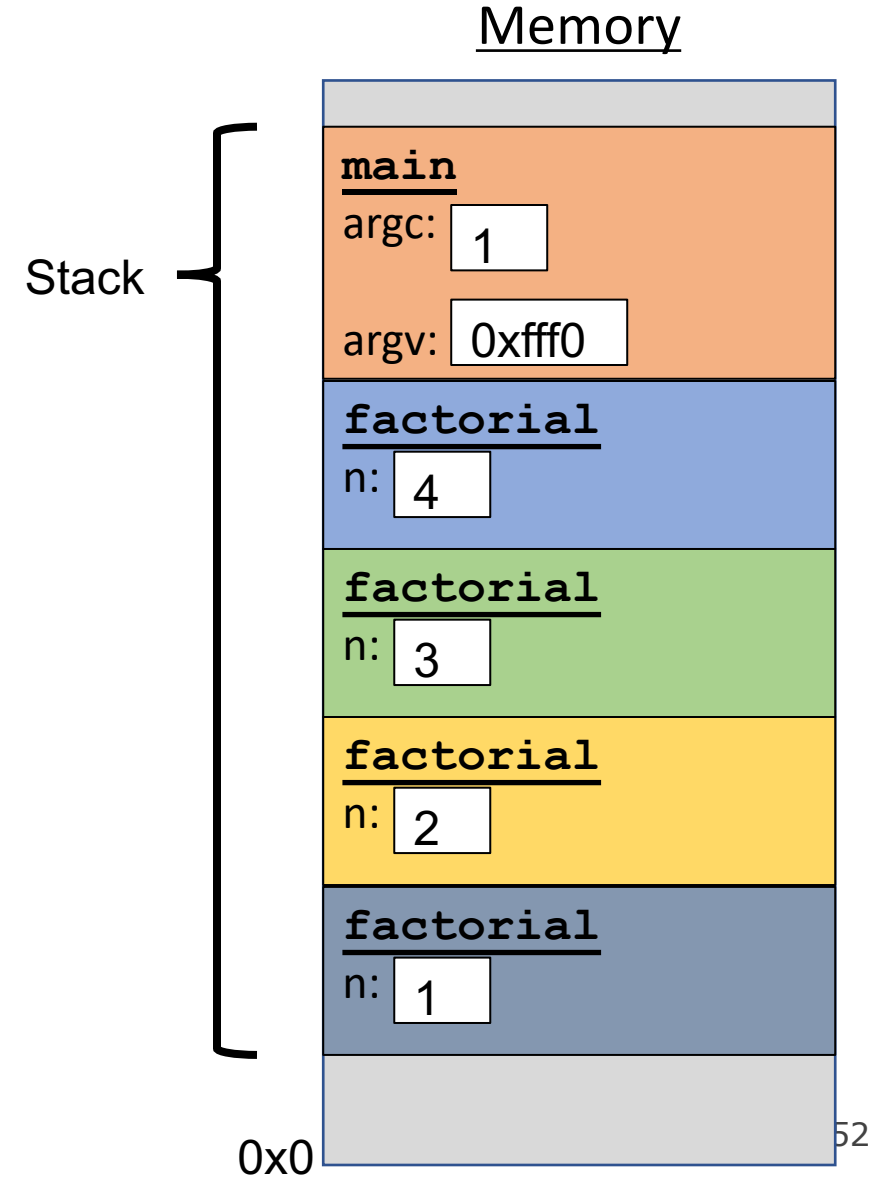
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

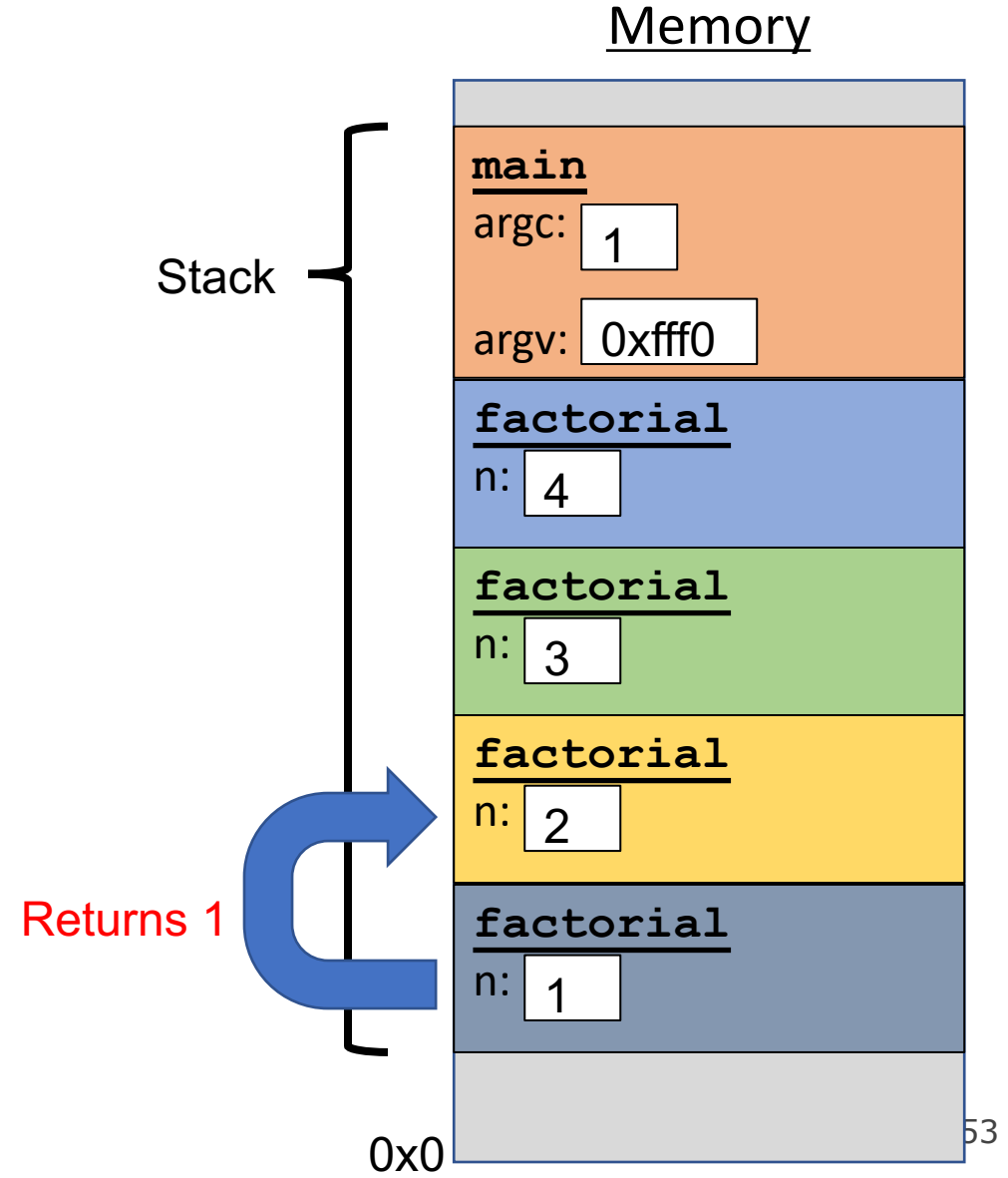
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

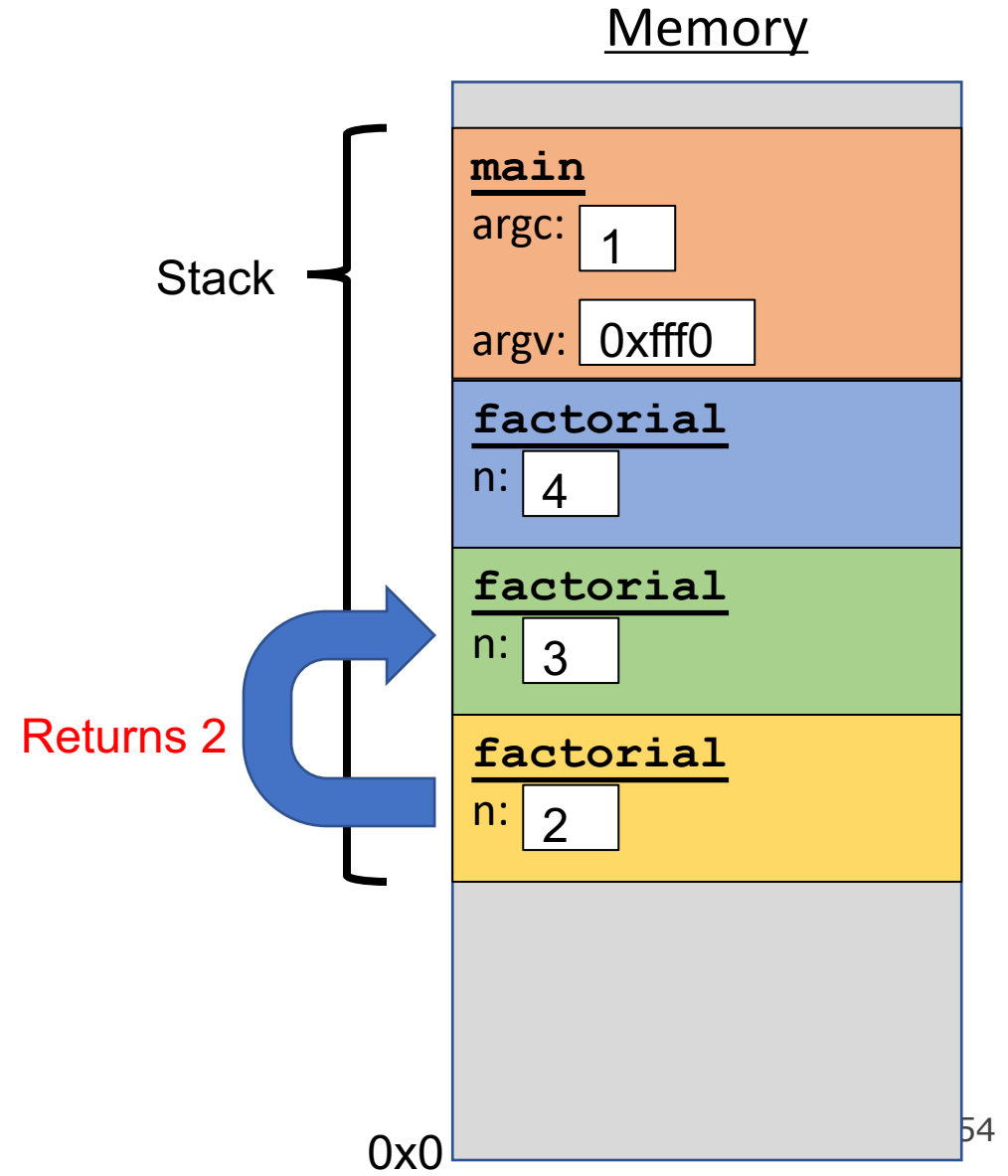
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

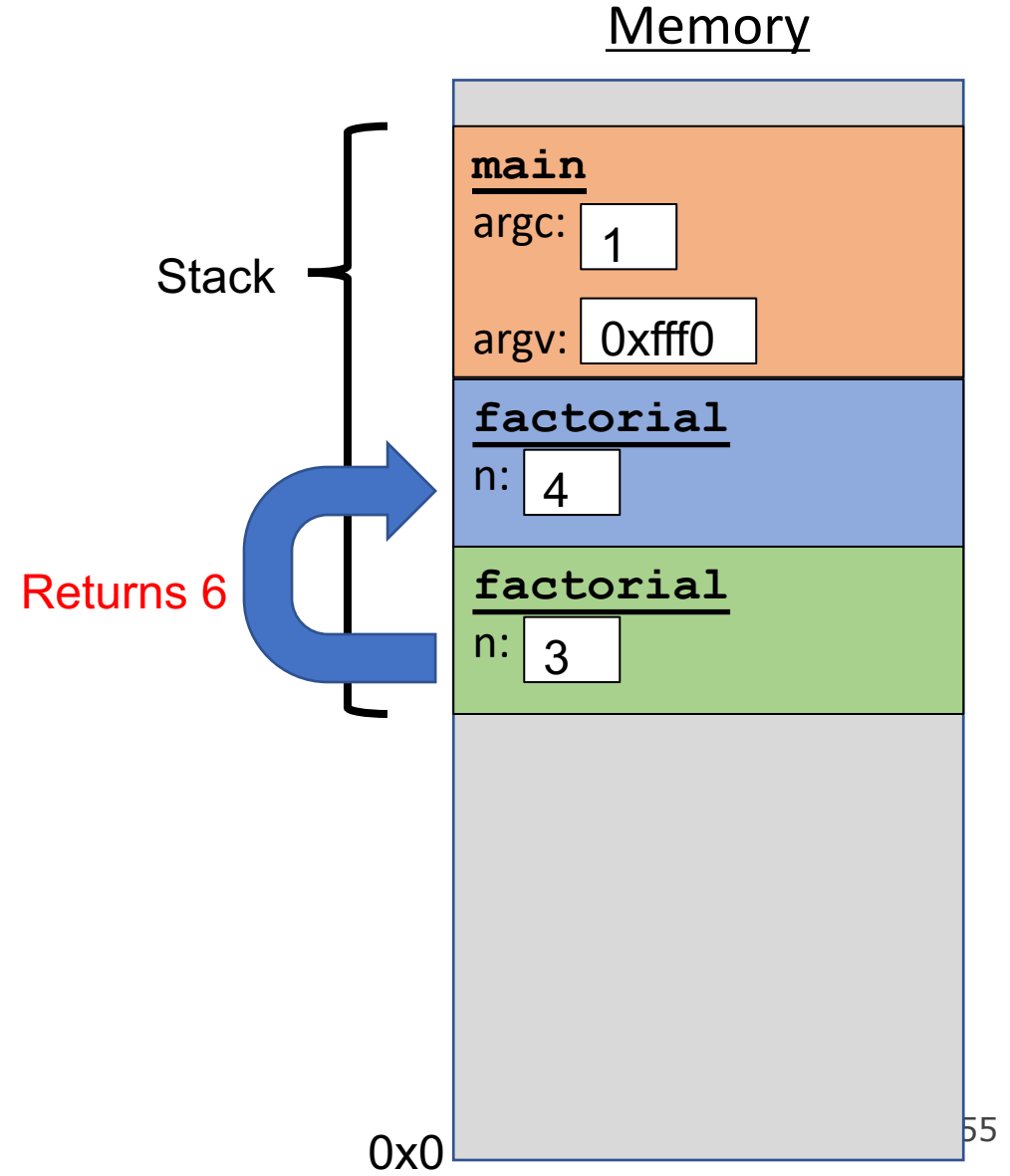
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack

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

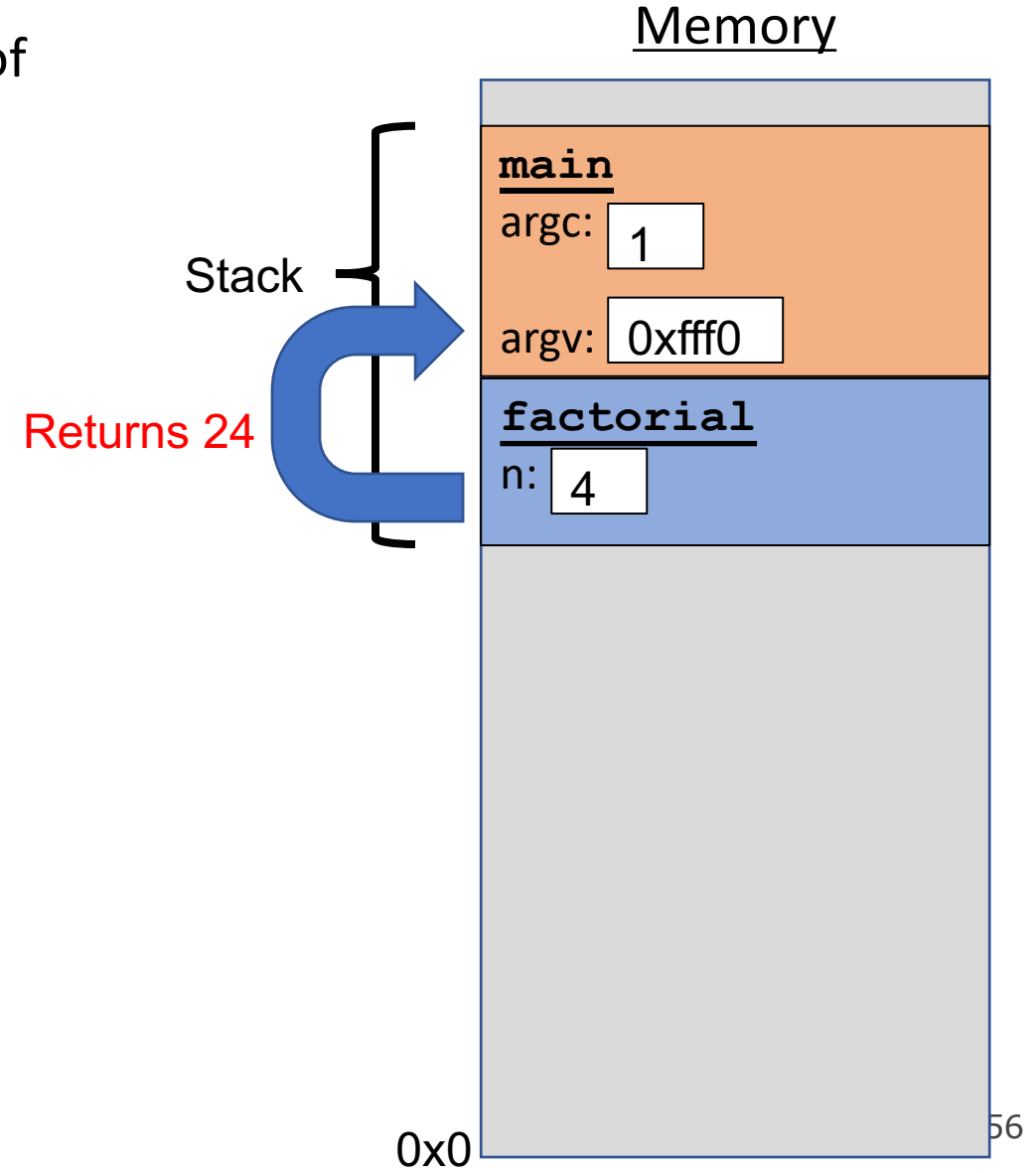
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



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;  
}
```

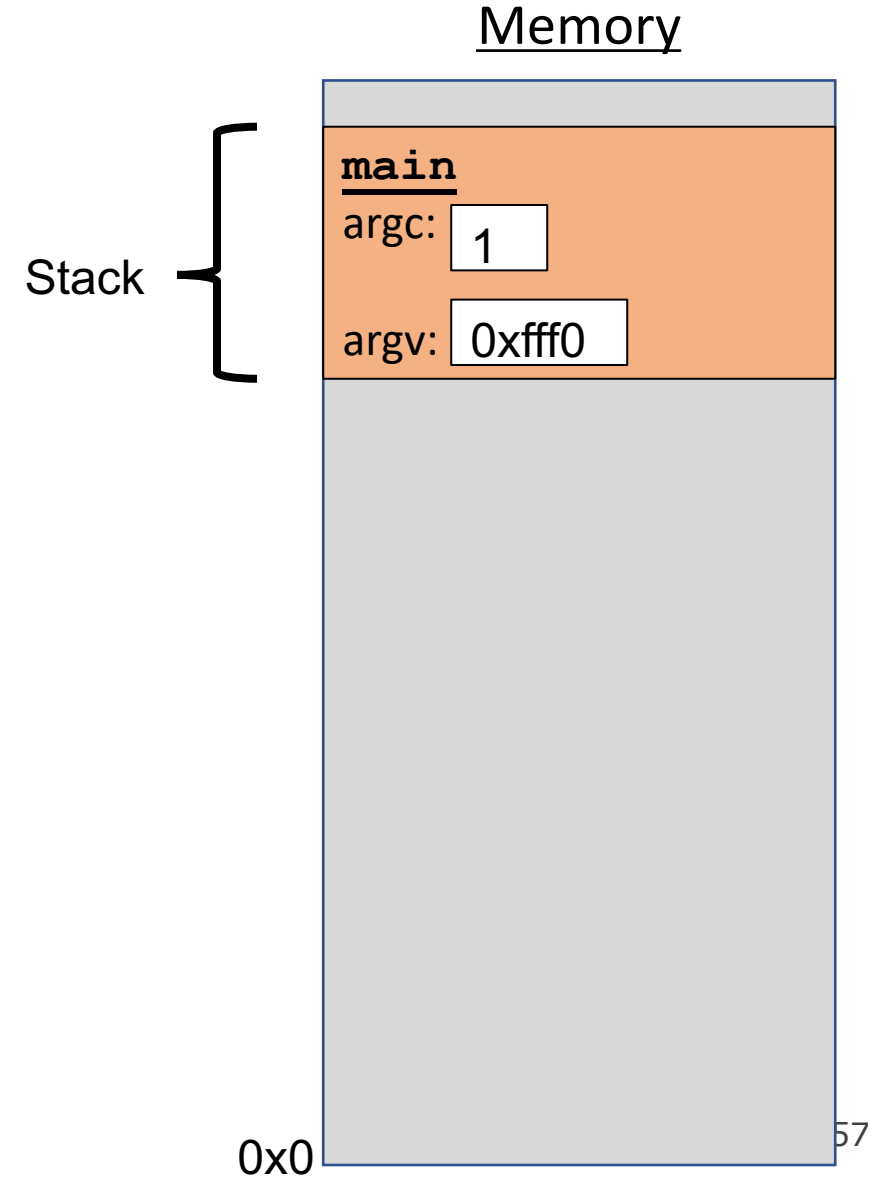


The Stack

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);
    }
}

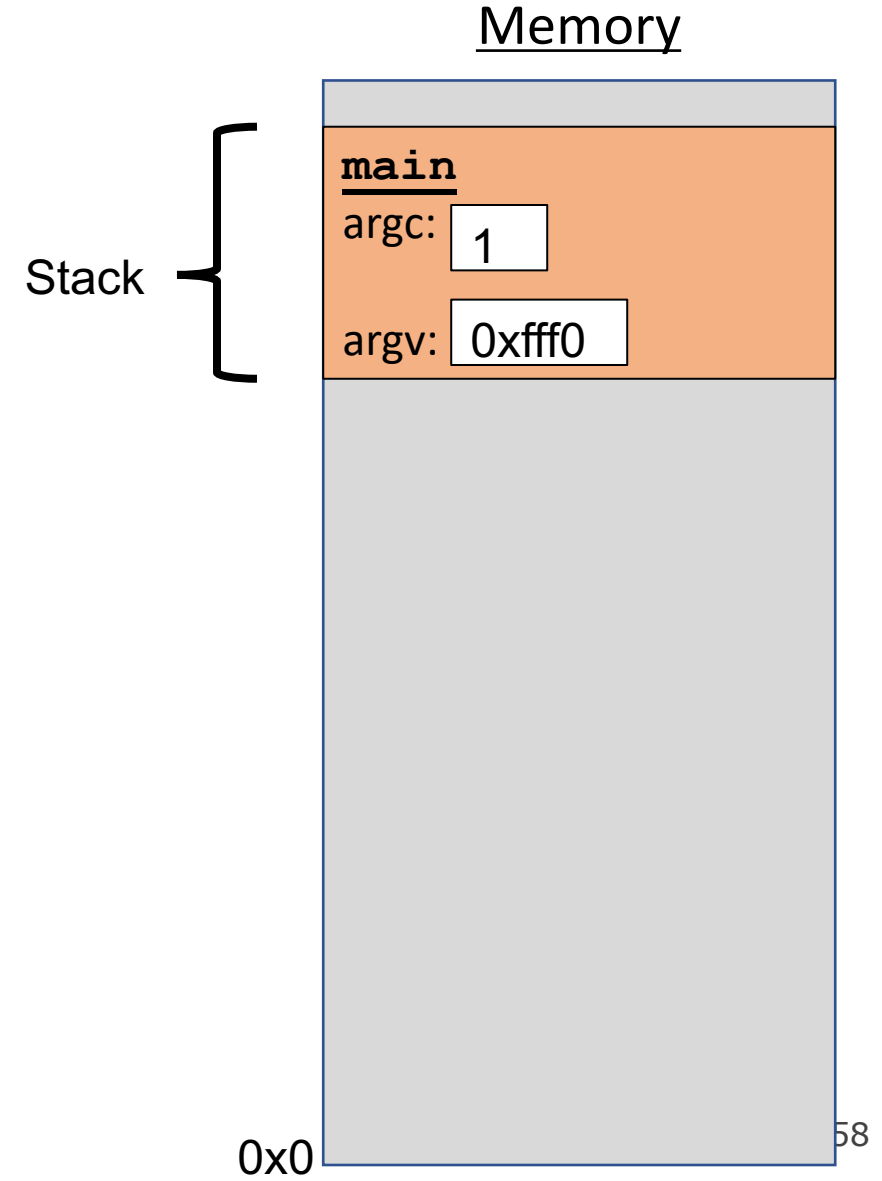
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



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;  
}
```



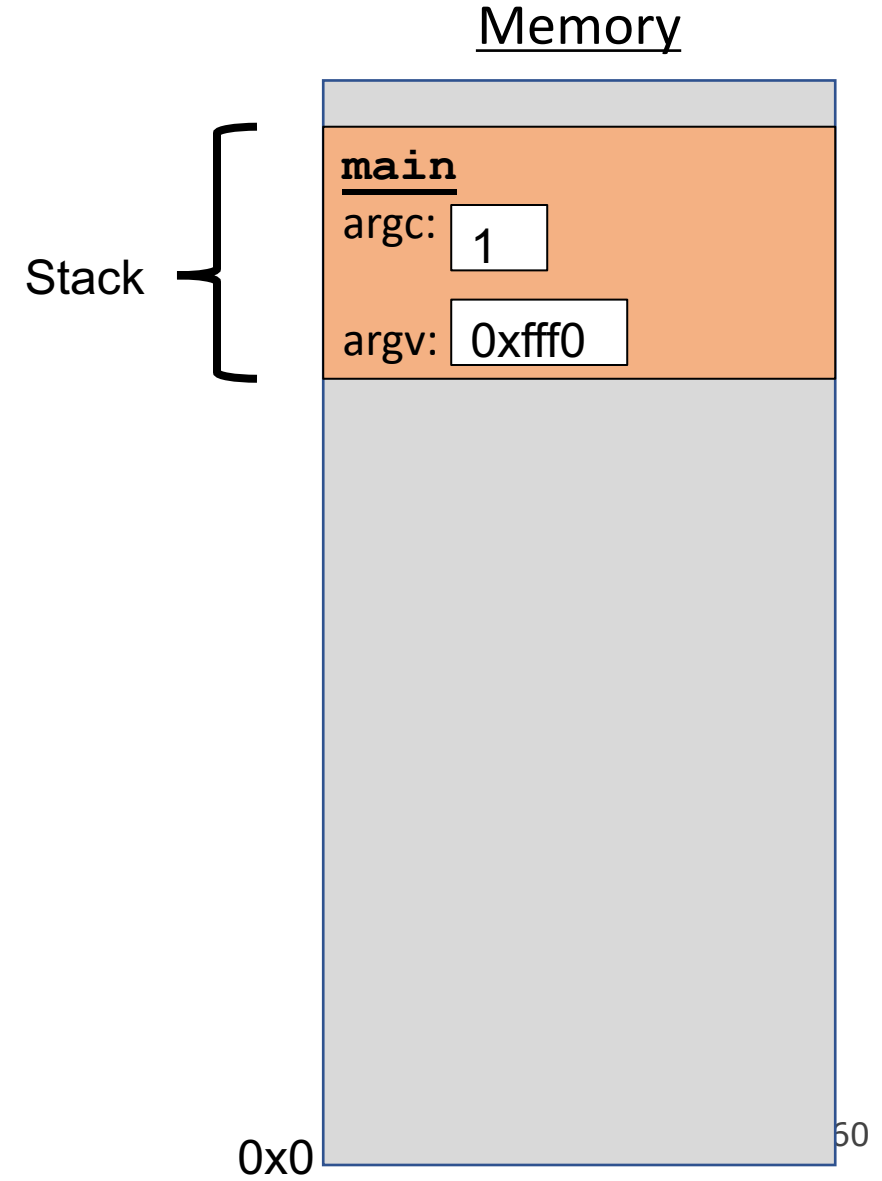
The Stack

- 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

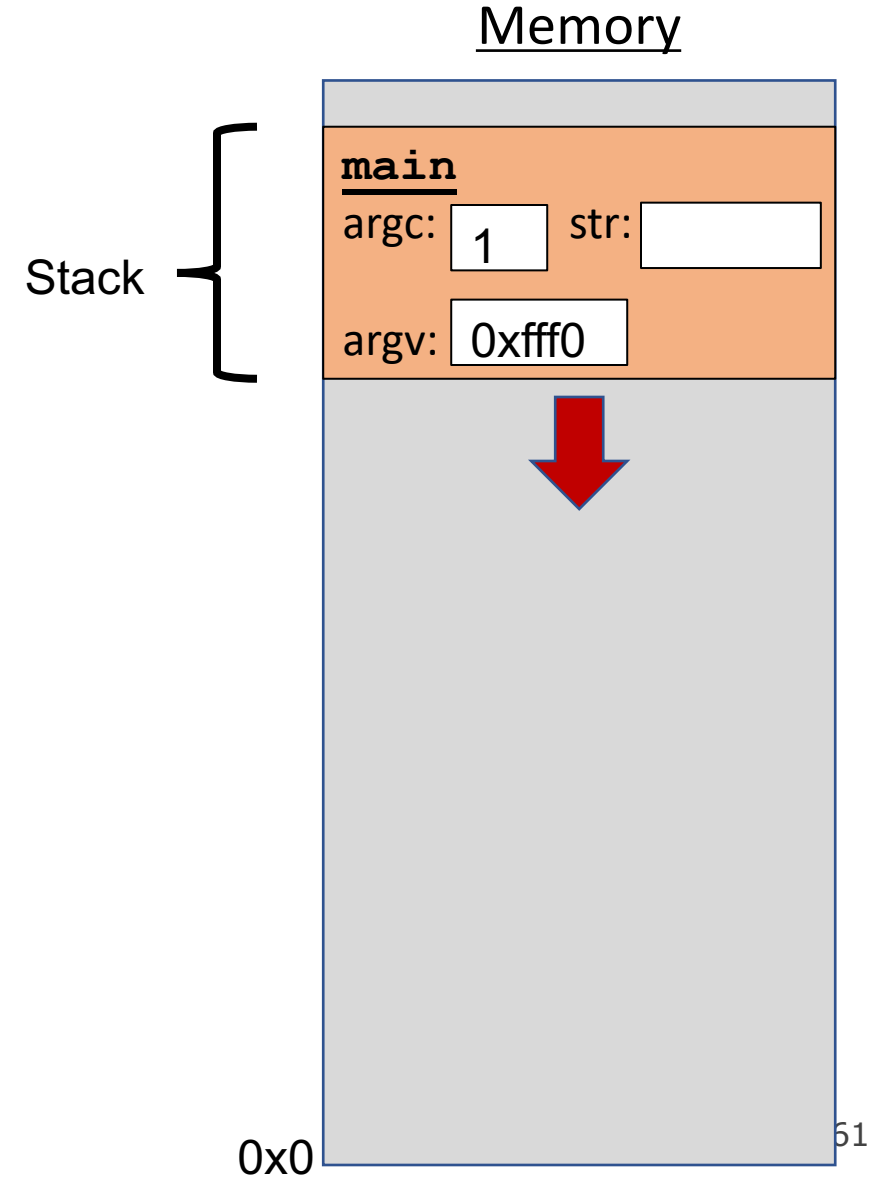
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}
```

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



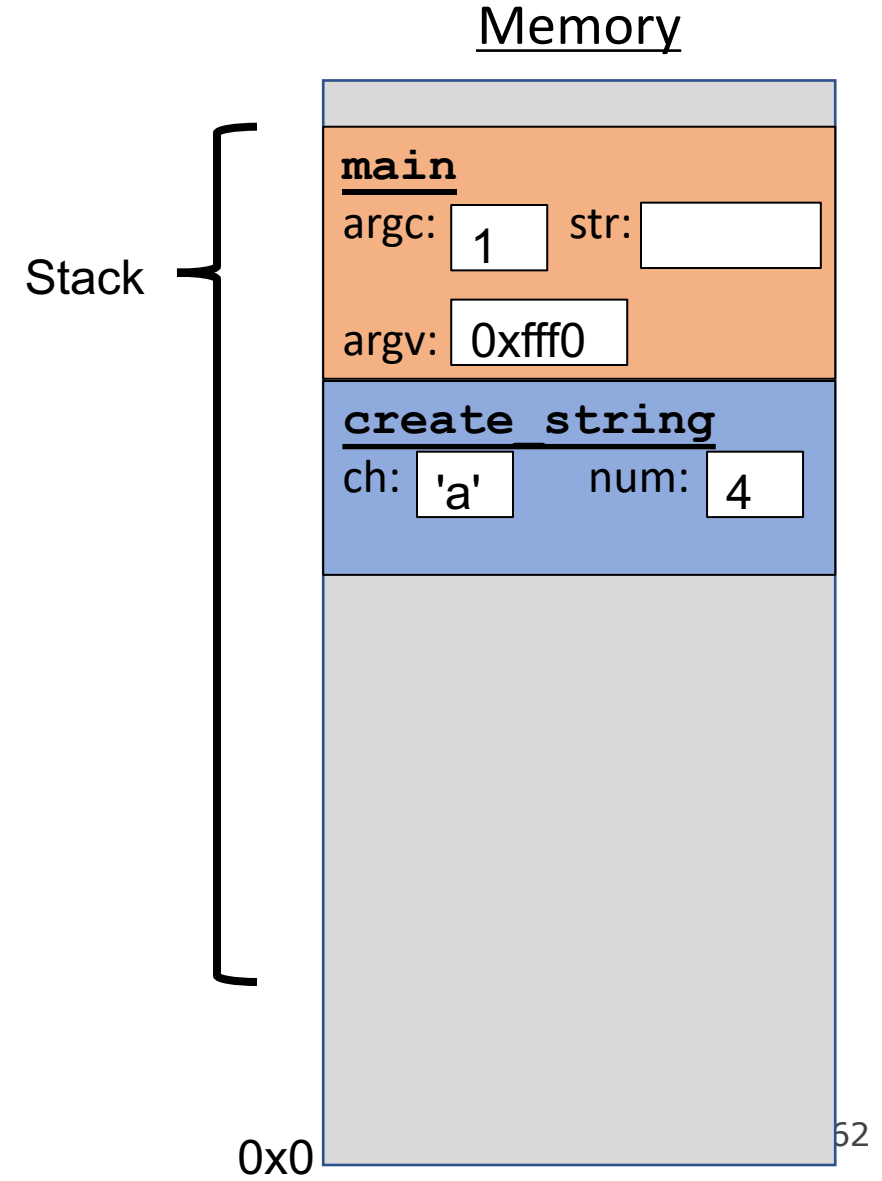
The Stack

```
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;  
}
```



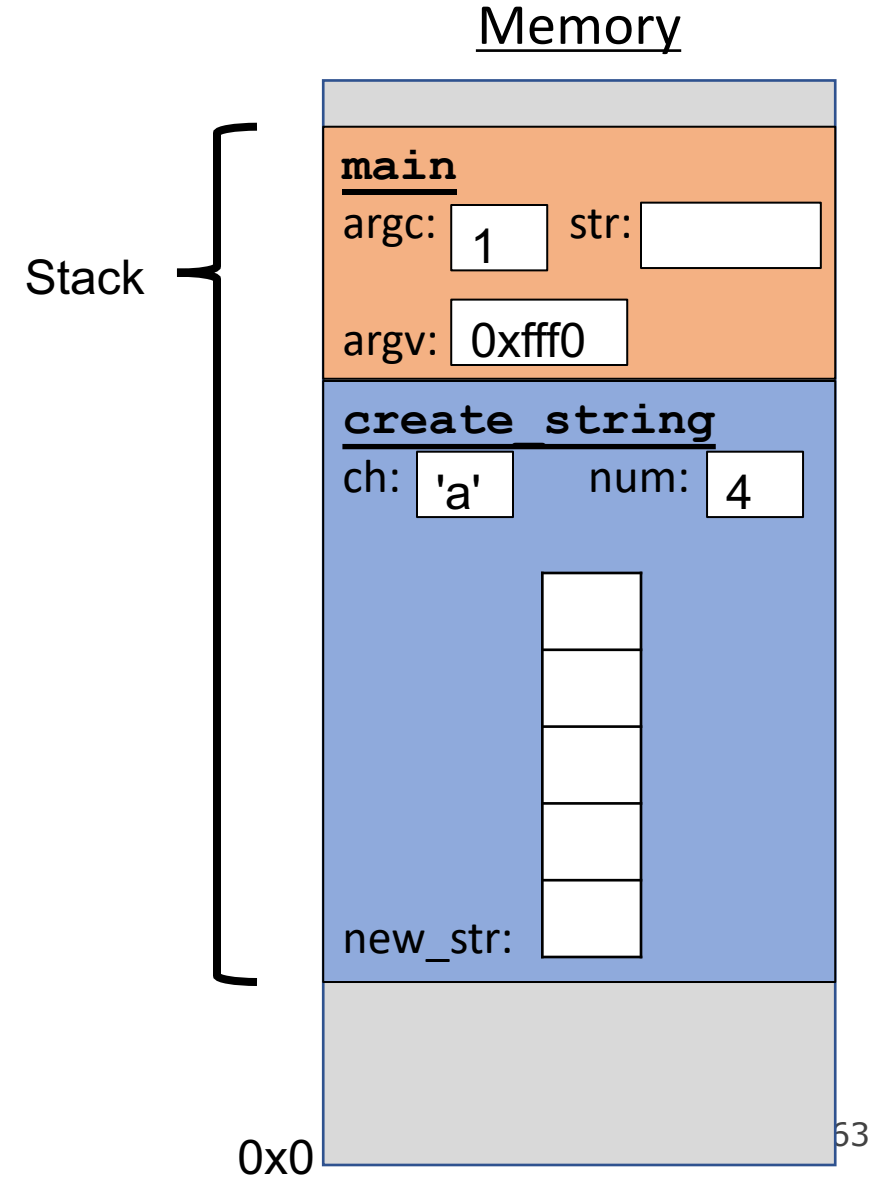
The Stack

```
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;  
}
```



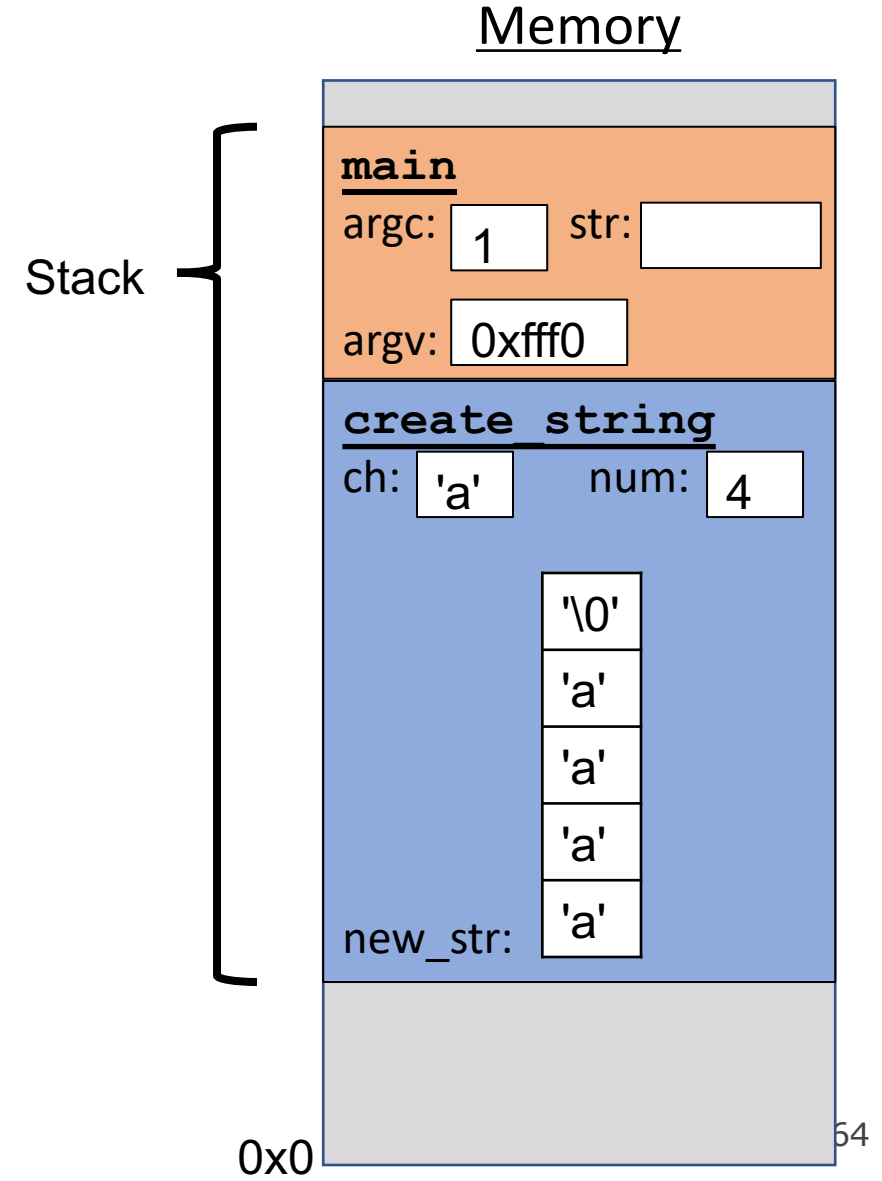
The Stack

```
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;  
}
```

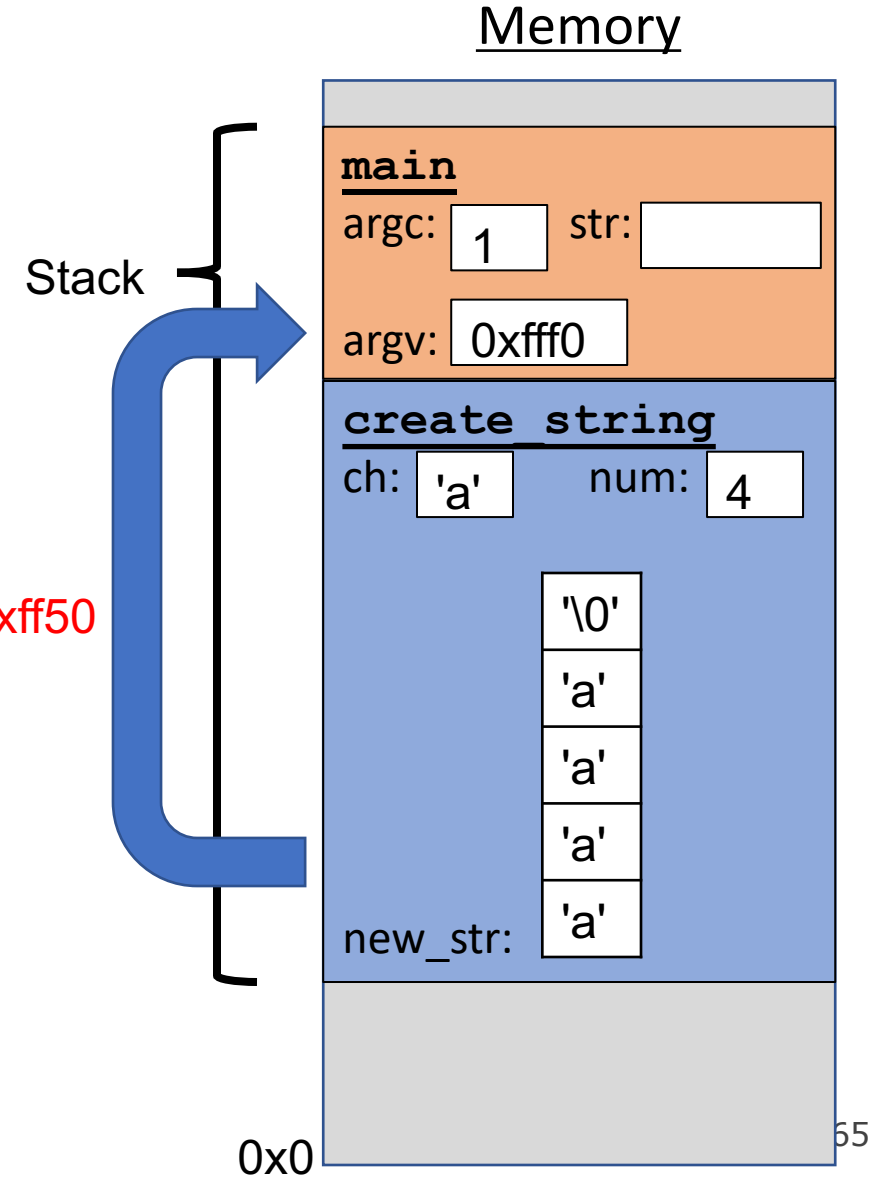


The Stack

```
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;  
}
```

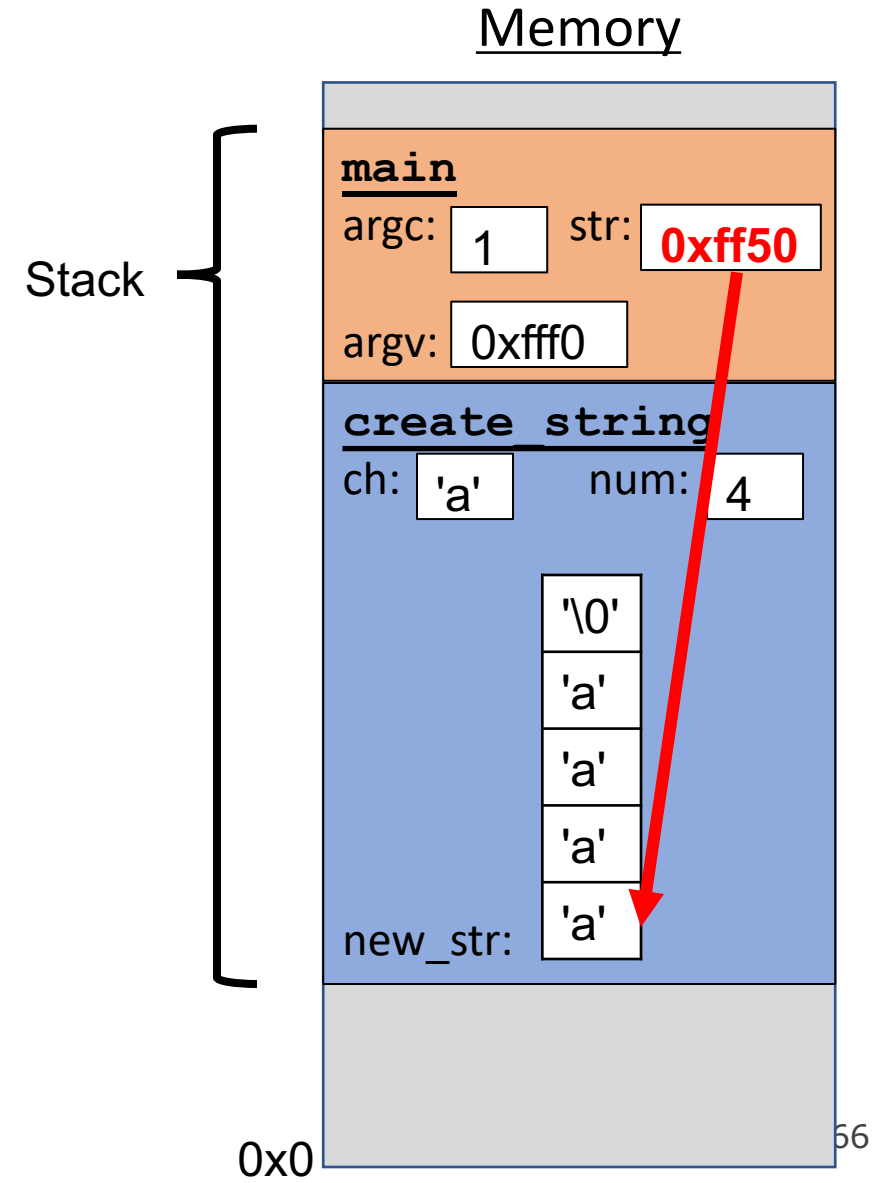
```
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```

Returns e.g. 0xff50



The Stack

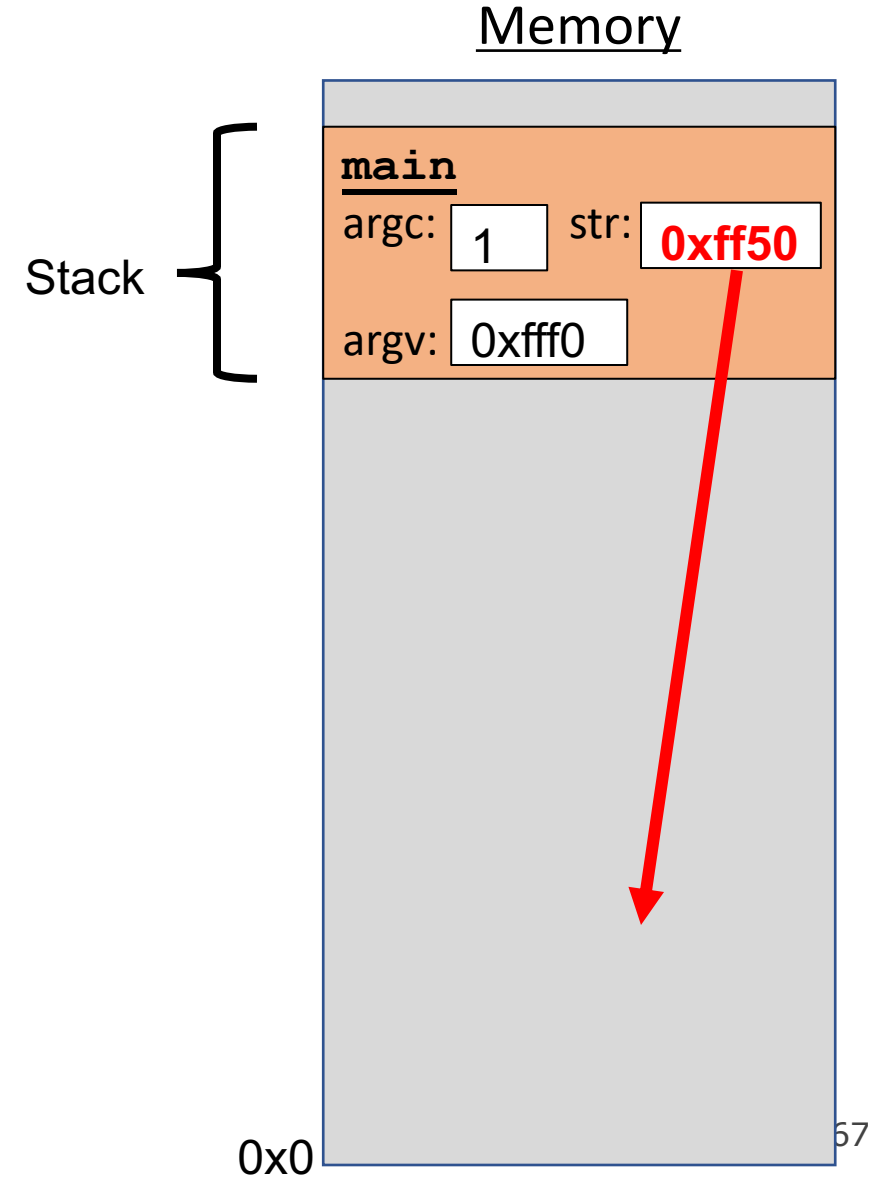
```
char *create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a', 4);  
    printf("%s", str);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

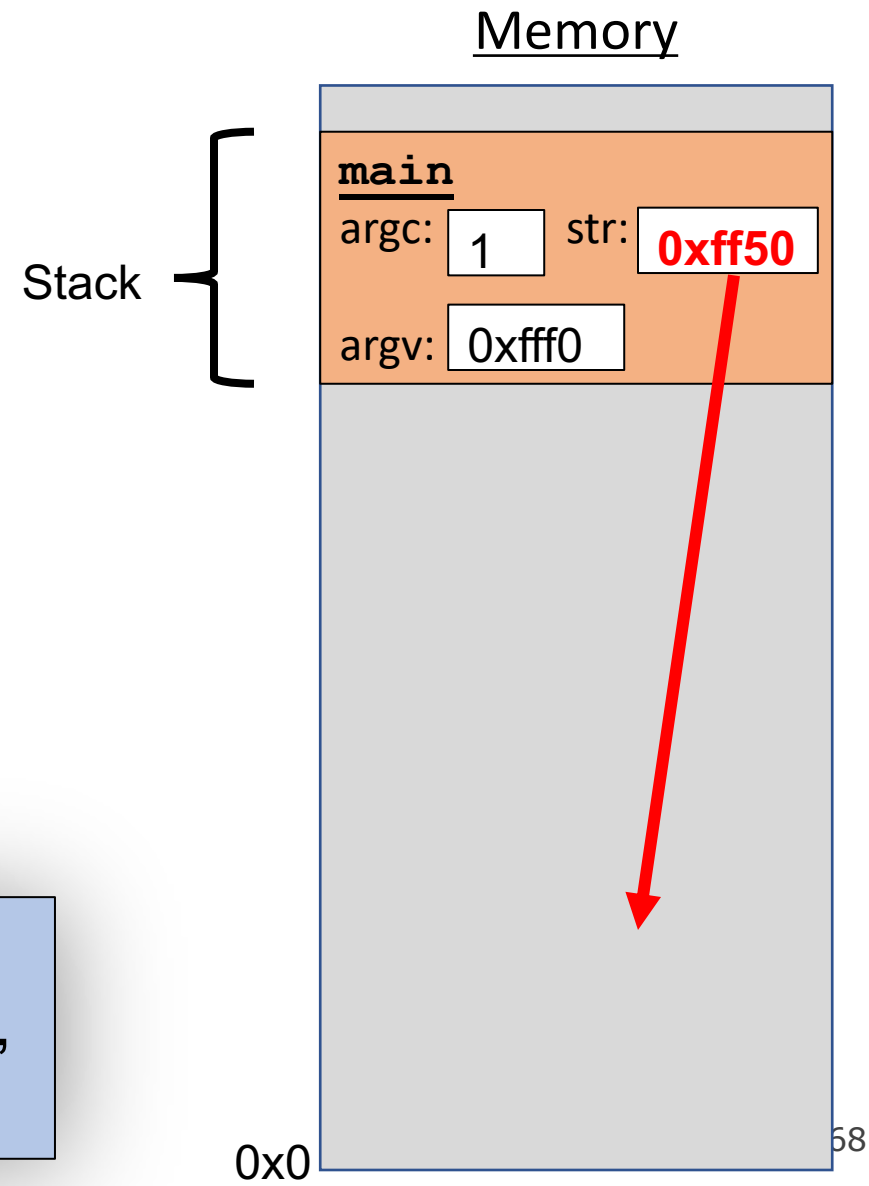


The Stack

```
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!

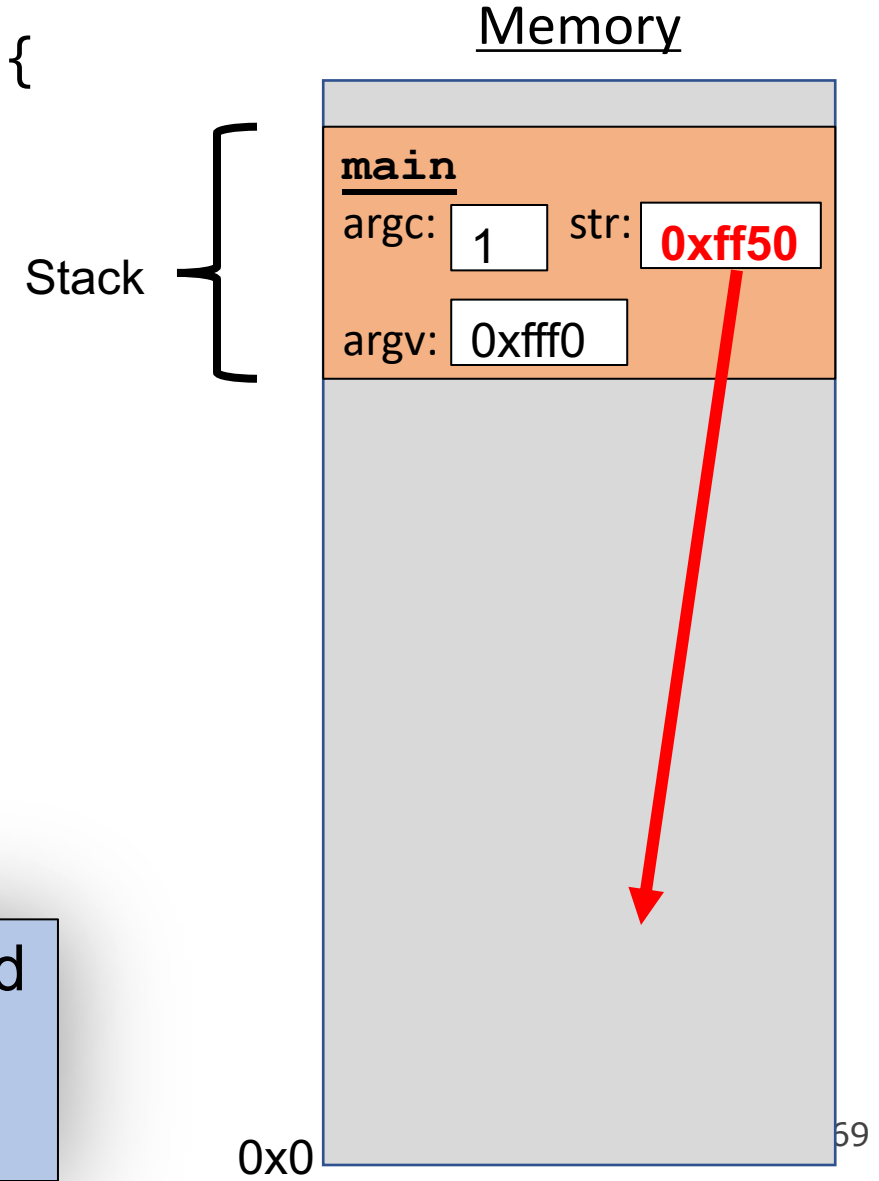


The Stack

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

```
int main(int argc, char *argv[]) {  
    char buf[5];  
    char *str = create_string(buf, 'a', 4);  
    printf("%s", str); // want "aaaa"  
    return 0;  
}
```

Sometimes, we can make the array in the caller and pass it as a parameter. But this isn't always possible if the size isn't known in advance.



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

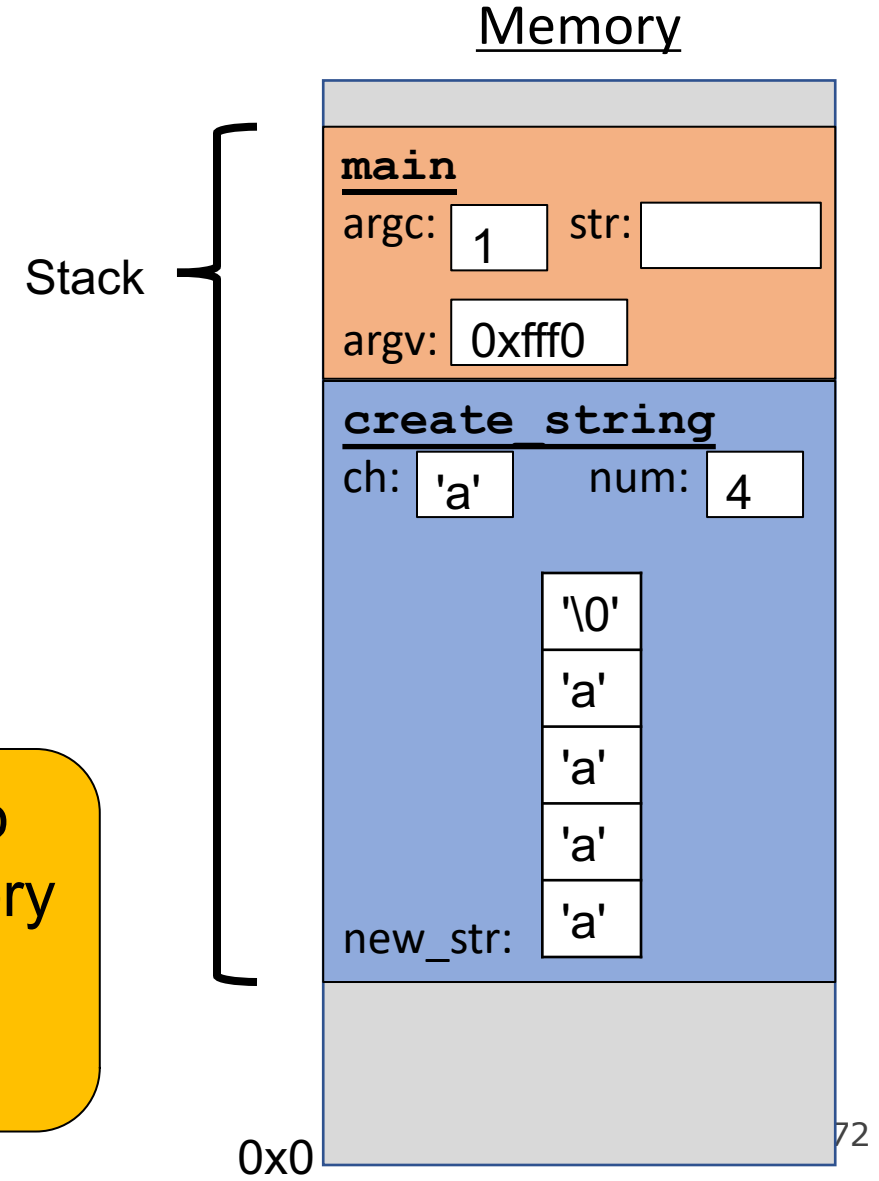
- **Recap:** Pointers So Far
- Arrays in Memory
- Pointer Arithmetic
- The Stack
- **The Heap and Dynamic Memory**

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?

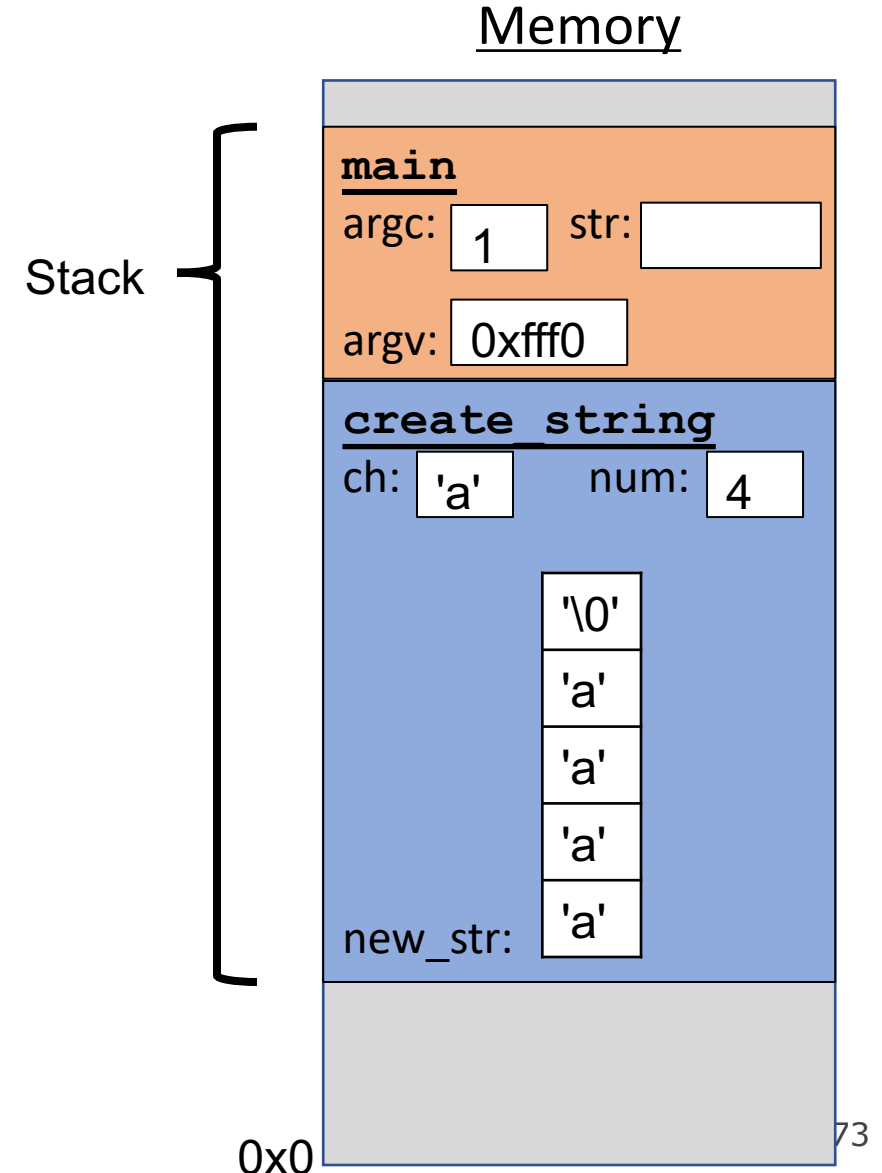


The Heap

```
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;  
}
```

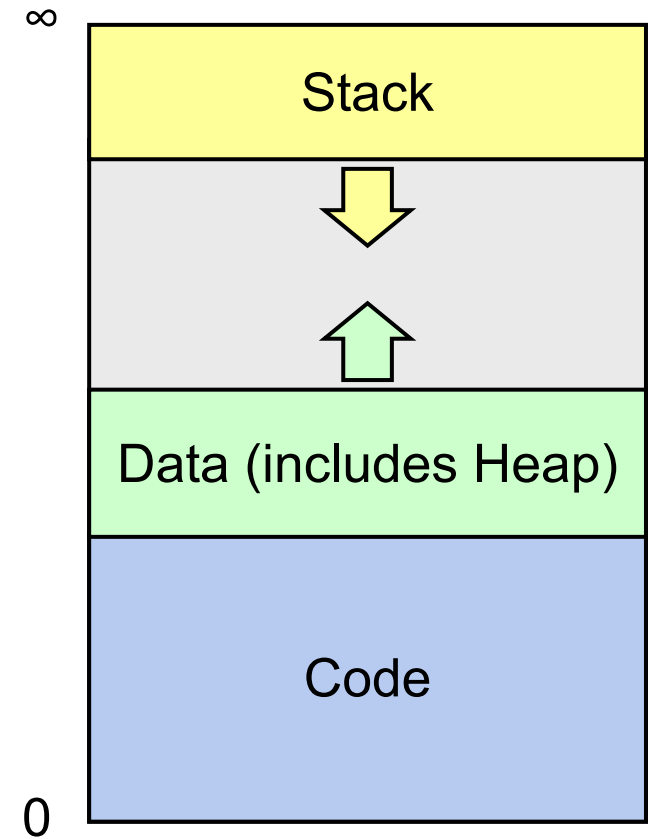
C: sure, but since I don't know when to clean it up anymore, it's your responsibility...



Memory Layout

- 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**.



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 (more later)
3. Free when done (more later)

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

malloc

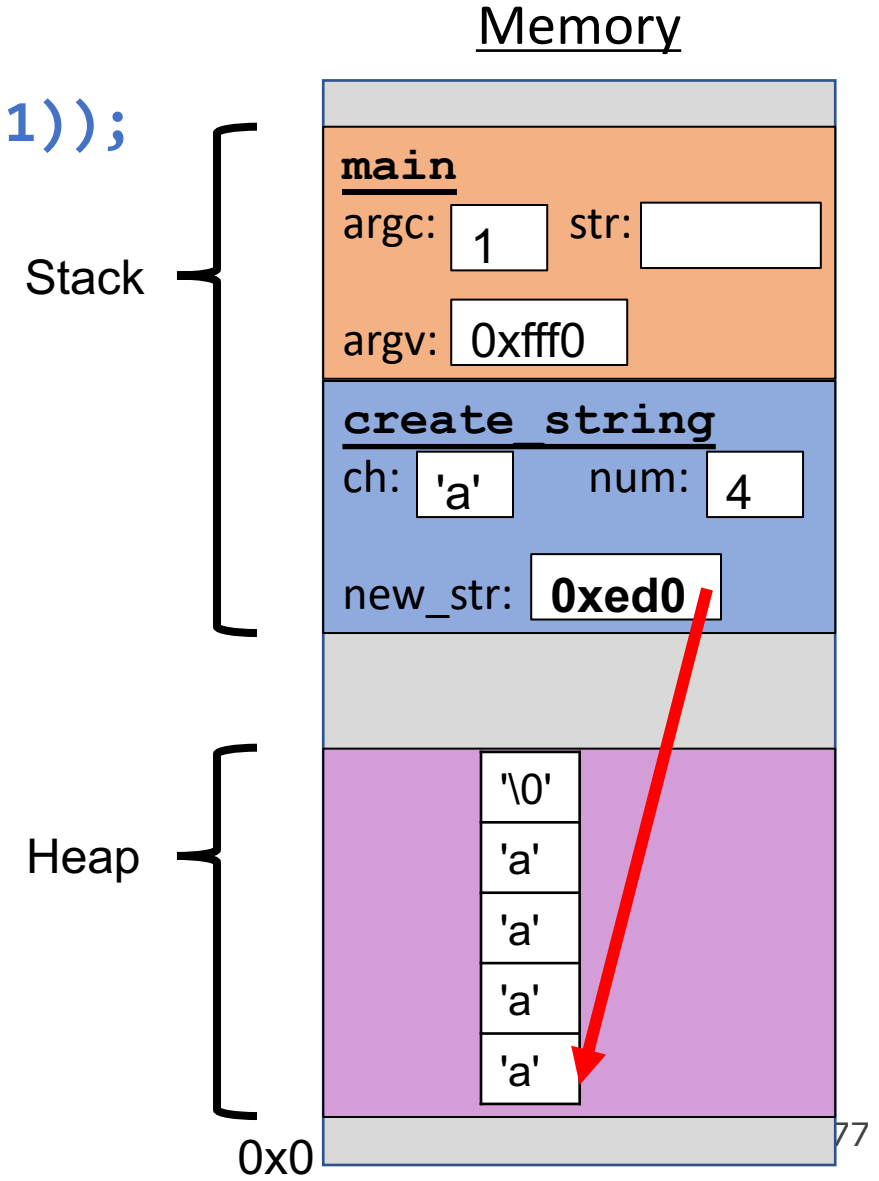
```
void *malloc(size_t size);
```

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

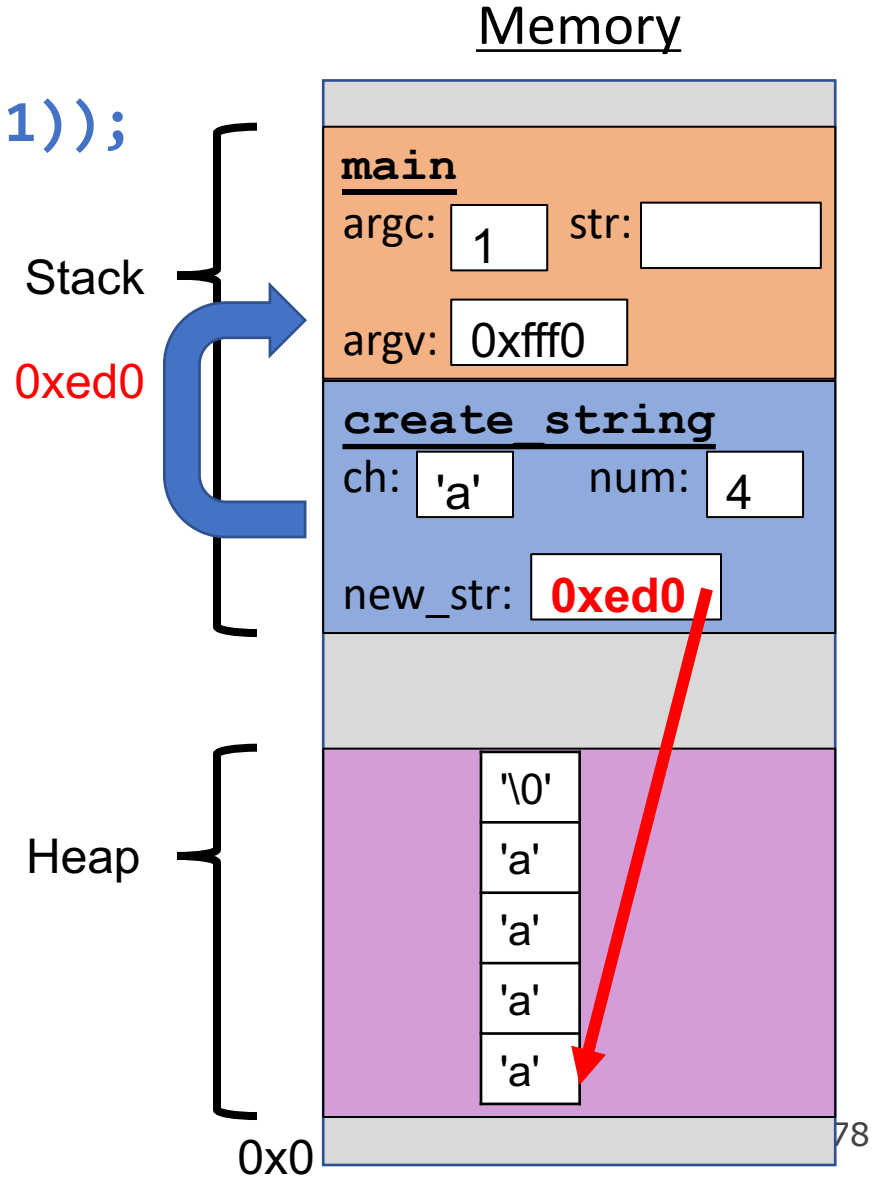
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a',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



The Heap

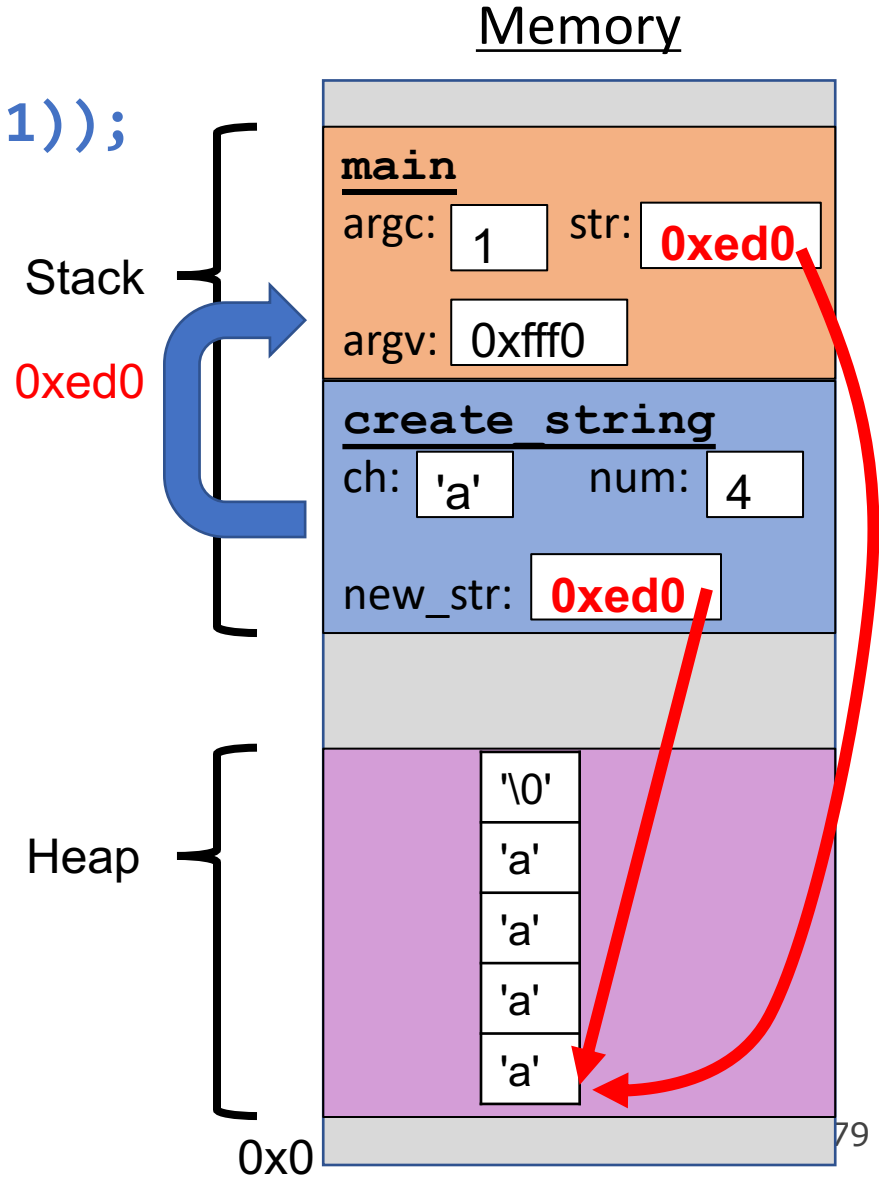
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a',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

Returns e.g. 0xed0



The Heap

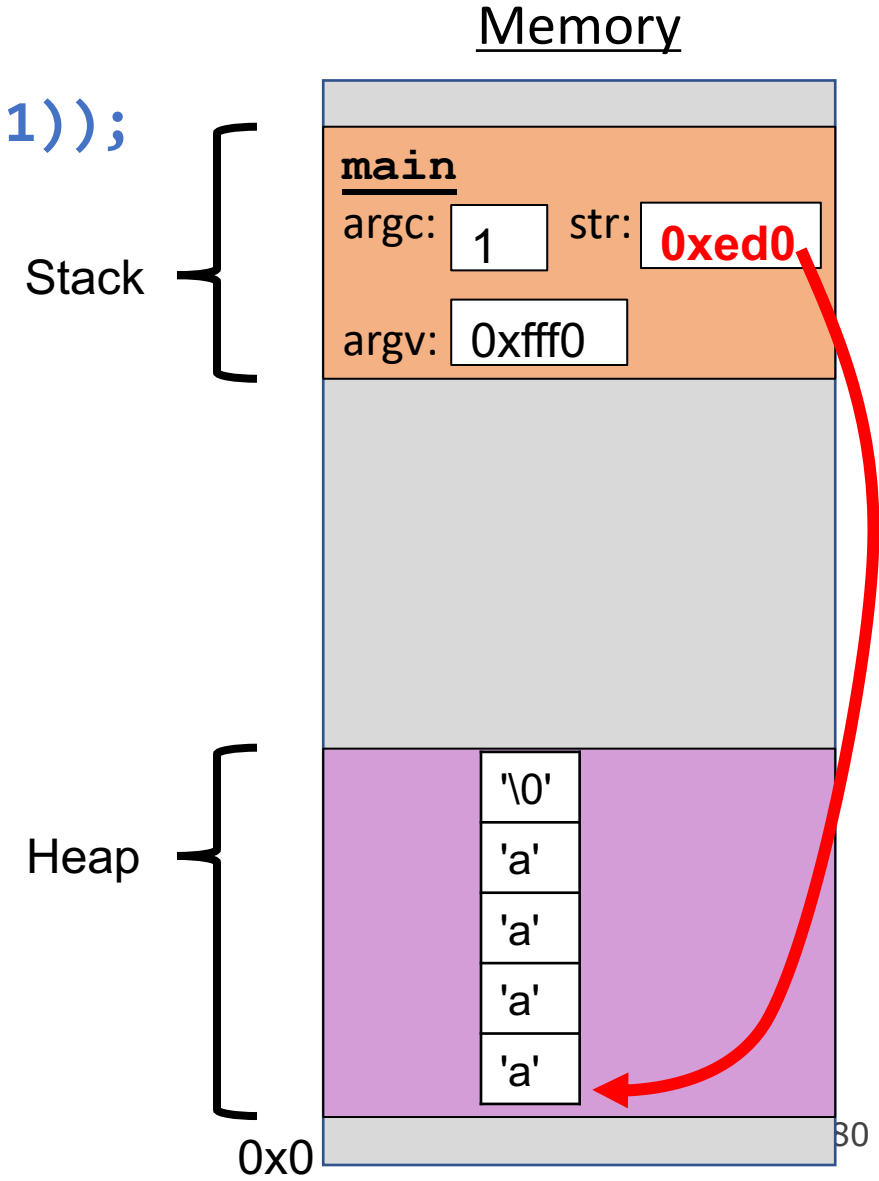
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('a',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



The Heap

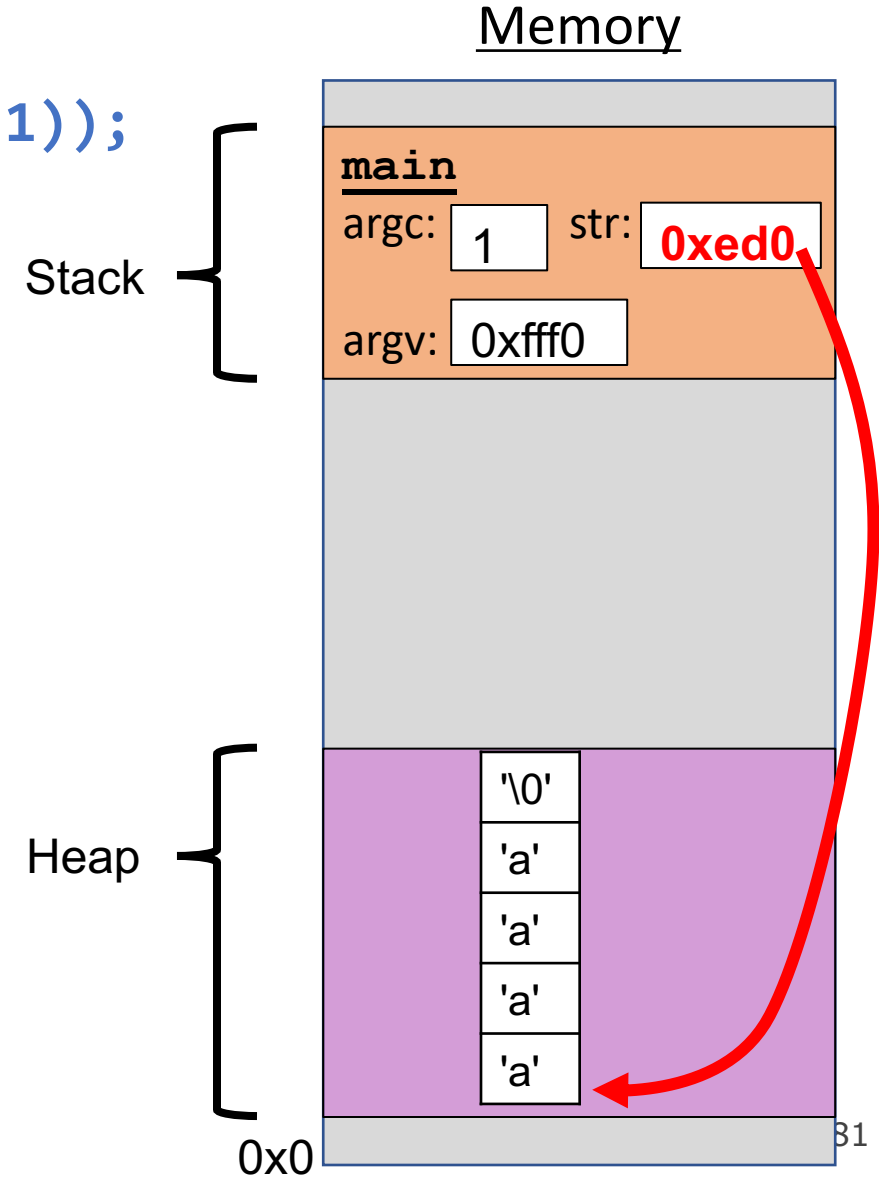
```
char *create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char) * (num + 1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}
```

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



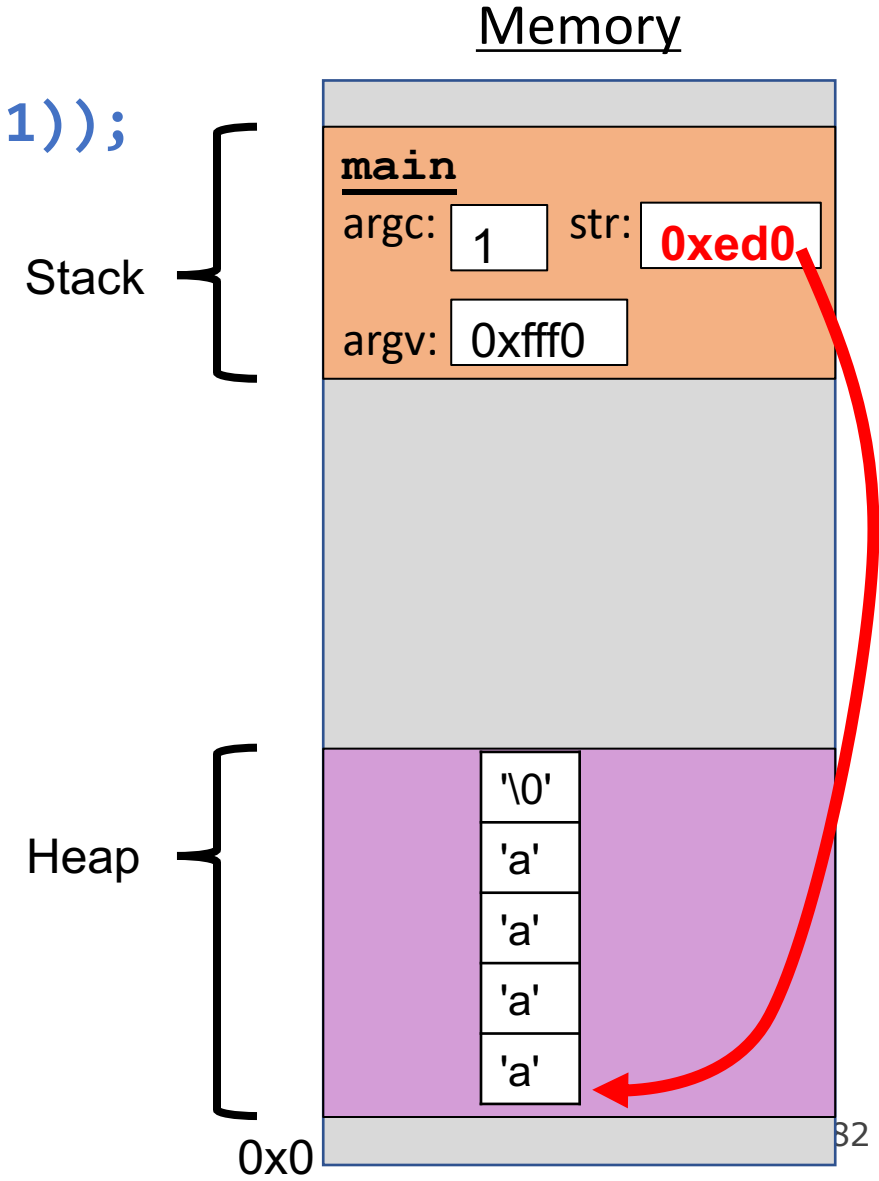
The Heap

```
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 }
```

Respond on PolEv:

pollev.com/cs107

or text CS107 to 22333

once to join.



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. `int *arr = malloc(len);`



How should we declare *arr*?

```
int arr[len];
```

0%

```
int arr[] = malloc(sizeof(int));
```

0%

```
int *arr = malloc(sizeof(int) * len);
```

0%

```
int *arr = malloc(sizeof(int) * (len + 1));
```

0%

```
int *arr = malloc(len);
```

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 }
```

- Use a pointer to store the address returned by malloc.
- Malloc's argument is the **number of bytes** to allocate, not # of elements.
- Arrays other than strings don't have special terminating elements/values.

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. `int *arr = malloc(len);`

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 (more later)

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

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';
```

You could imagine **strdup** might be implemented in terms of **malloc** + **strcpy**.

Recap

- **Recap:** Pointers So Far
- Arrays in Memory
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Lecture 10 takeaway: We can allocate memory on the heap to manage it ourselves. We manipulate heap memory via pointers.

Extra Practice

Translating C into English

If **declaration**: "pointer"

ex: `int *` is "pointer to an int"

*

If **operation**: "dereference/the value at address"

ex: `*num` is "the value at address num"

& "address of"

<ptr
name> address

<arr
name> address
(except sizeof)

```
int arr[] = {3, 4, -1, 2};
```

1. `int *ptr0 = arr;`
2. `int *elt0 = *arr;`
3. `int elt = *(arr + 3);`
4. `int **ptr1 = &ptr;`

```
// initializes stack array  
// with 4 ints
```

Type check with a diagram!



Translating C into English

If **declaration**: "pointer"

ex: `int *` is "pointer to an int"

*

If **operation**: "dereference/the value at address"

ex: `*num` is "the value at address num"

& "address of"

<ptr
name> address

<arr address
name> (except sizeof)

```
int arr[] = {3, 4, -1, 2};
```

1. `int *ptr0 = arr;`
2. `int *elt0 = *arr;`
3. `int elt = *(arr + 3);`
4. `int **ptr1 = &ptr;`

```
// initializes stack array  
// with 4 ints
```

Address `arr`

Value at address `arr`

The value at address `<3 ints
after address arr>`

address of `ptr`

Type check with a diagram!

Translating C into English

If **declaration**: "pointer"

ex: `int *` is "pointer to an int"

*

If **operation**: "dereference/the value at address"

ex: `*num` is "the value at address num"

& "address of"

<ptr
name> address

<arr
name> address
(except sizeof)

```
int arr[] = {3, 4, -1, 2};
```

1. `int *ptr0 = arr;`
2. `int *elt0 = *arr;`
3. `int elt = *(arr + 3);`
4. `int **ptr1 = &ptr;`

```
// initializes stack array  
// with 4 ints
```

Type check with a diagram!

* Wars: Episode I (of 2)

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

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

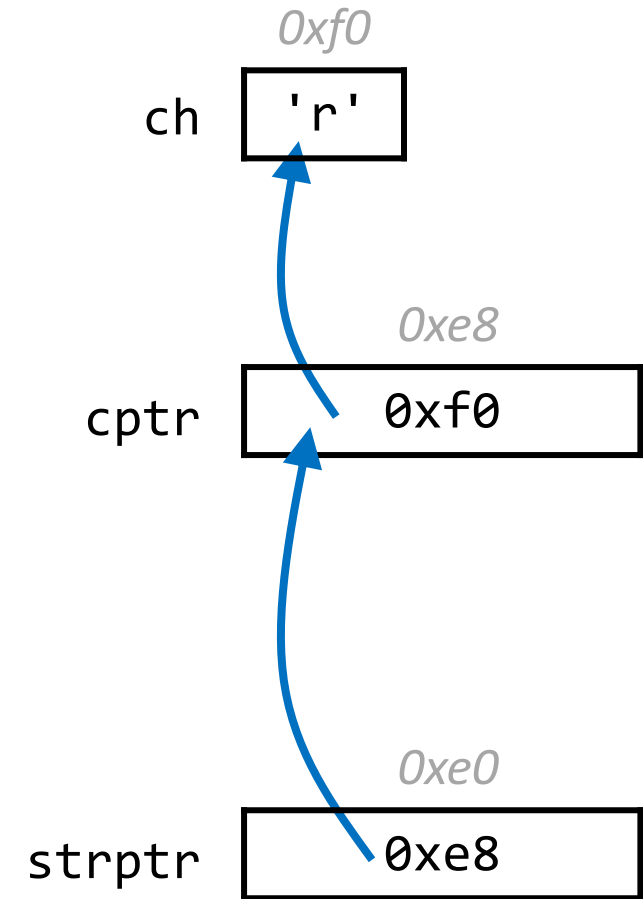
ch stores a char

```
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)

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

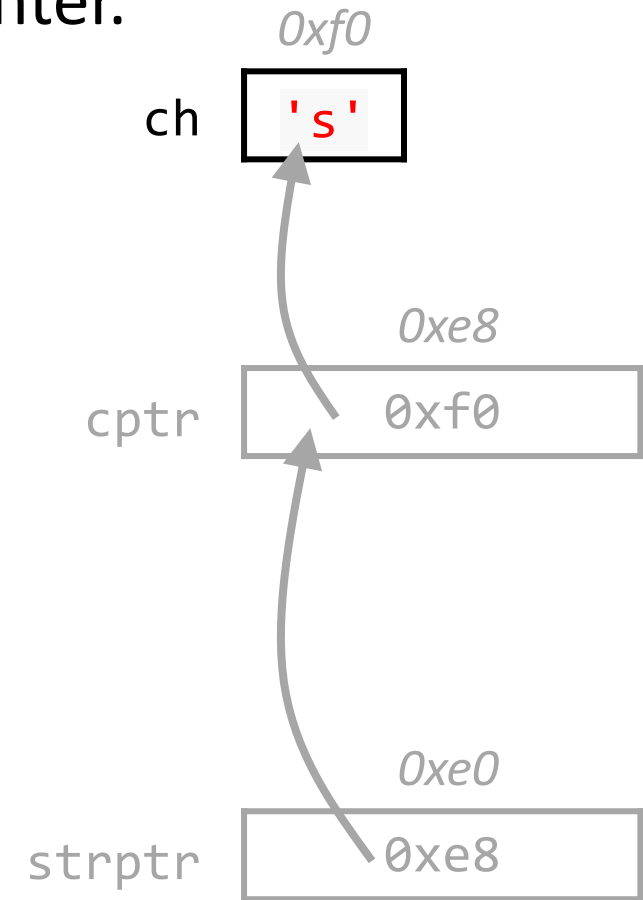
```
char ch = 'r';
```

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

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

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

Increment value stored in ch



* Wars: Episode II (of 2)

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

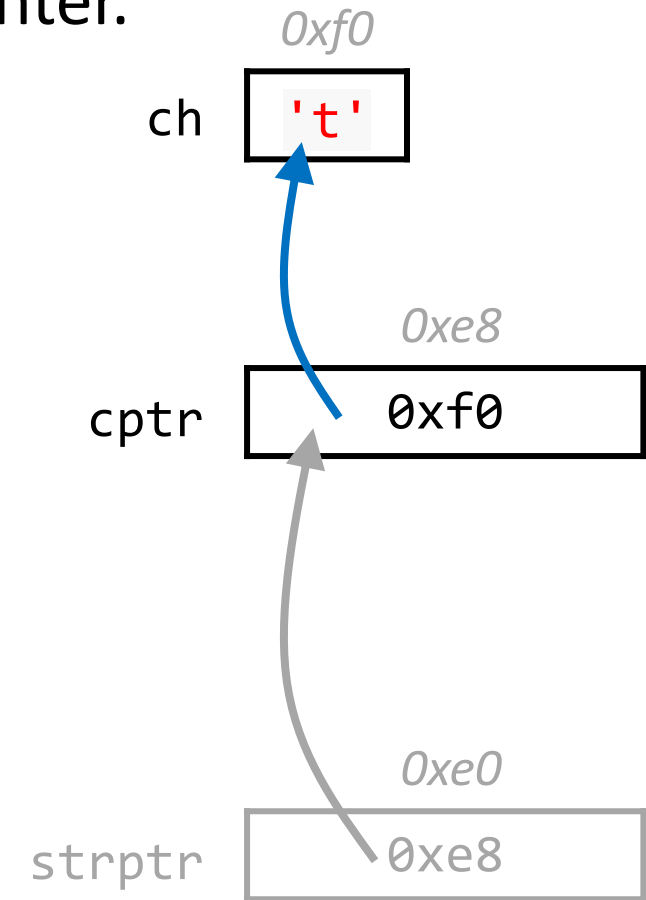
```
char ch = 'r';  
ch = ch + 1;
```

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)

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

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

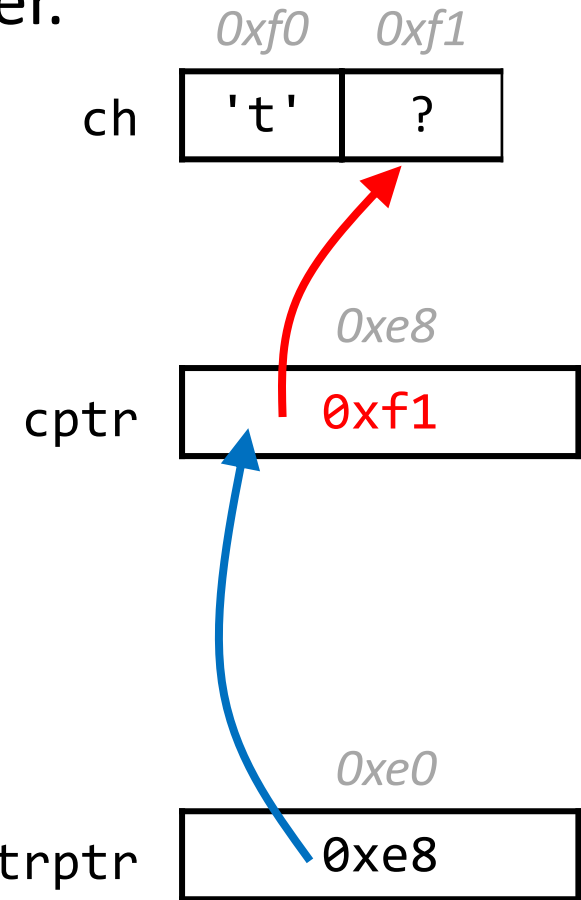
Increment value stored in ch

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

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

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

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



Exercise: Implementation

The below function sums up the string lengths of the num strings in strs.

- Try both **1. array [] syntax** and **2. pointer arithmetic**!

```
1 size_t get_total_strlen(char *strs[], size_t num) {
2     size_t total_length = 0;
3     for (int i = 0; i < num; i++) {
4         // fill this in
5     }
6     return total_length;
7 }
```



Exercise: Implementation

The below function sums up the string lengths of the num strings in strs.

- Try both **1. array [] syntax** and **2. pointer arithmetic**!

```
1 size_t get_total_strlen(char *strs[], size_t num) {
2     size_t total_length = 0;
3     for (int i = 0; i < num; i++) {
4         // TODO: fill this in two ways
5     }
6     return total_length;
7 }
```

Equivalent:

1. `total_length += strlen(strs[i]);`
2. `total_length += strlen(*(strs + i));`

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)