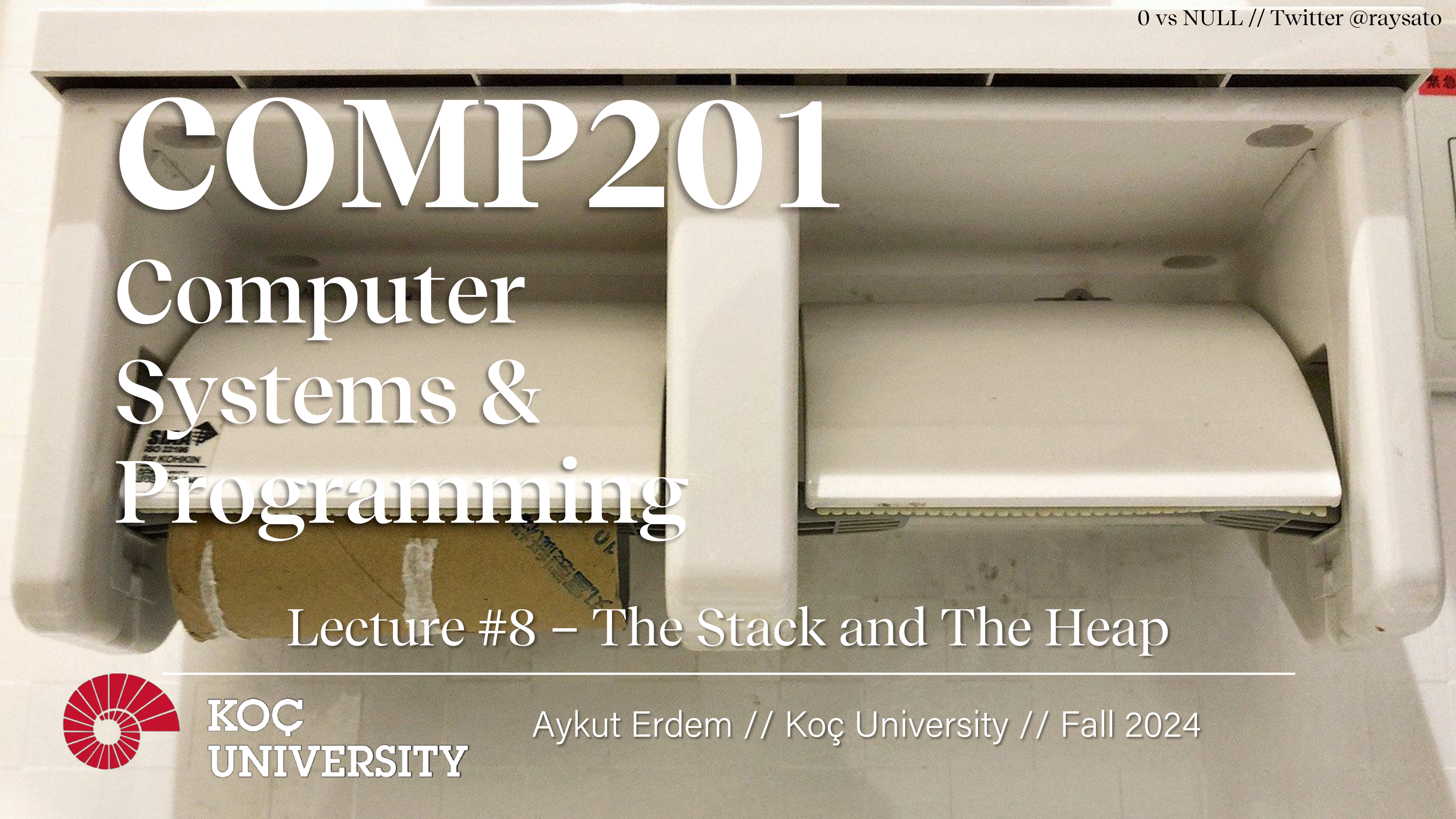


COMP201

Computer Systems & Programming

Lecture #8 – The Stack and The Heap

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2024

Pointers Practice

* Wars: Episode I (of 2)

In variable <u>**declaration**</u>, * creates a **pointer**.

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

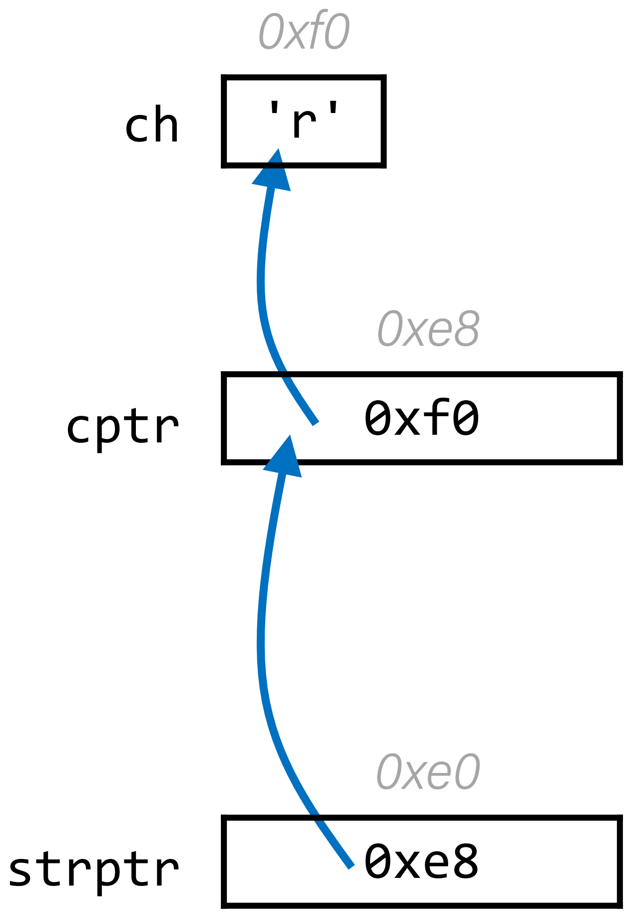
`ch` stores a `char`

```
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 *`)



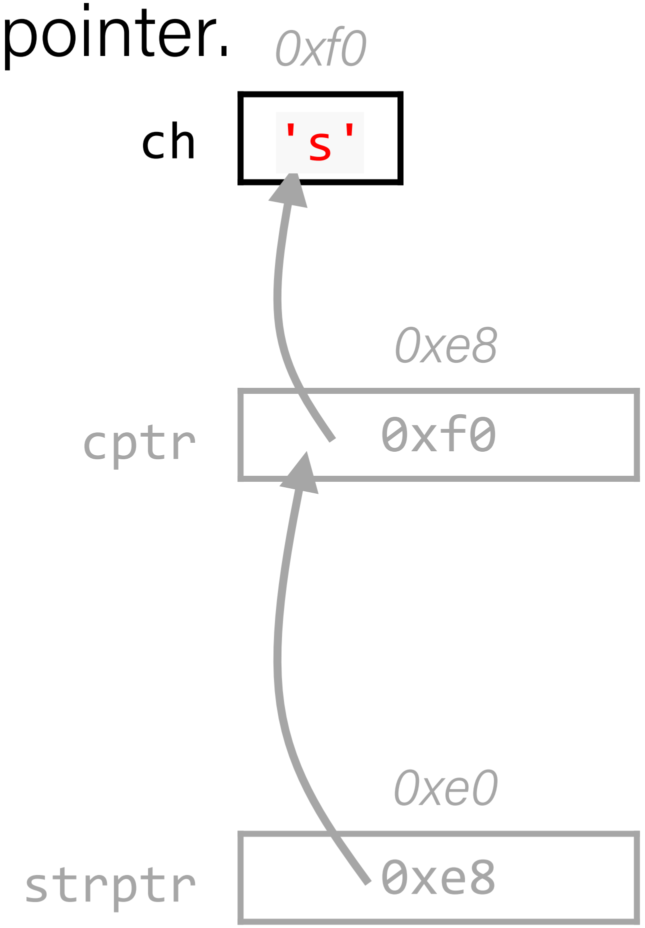
* Wars: Episode II (of 2)

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 <u>reading values from/storing values</u>, * **dereferences** a pointer.

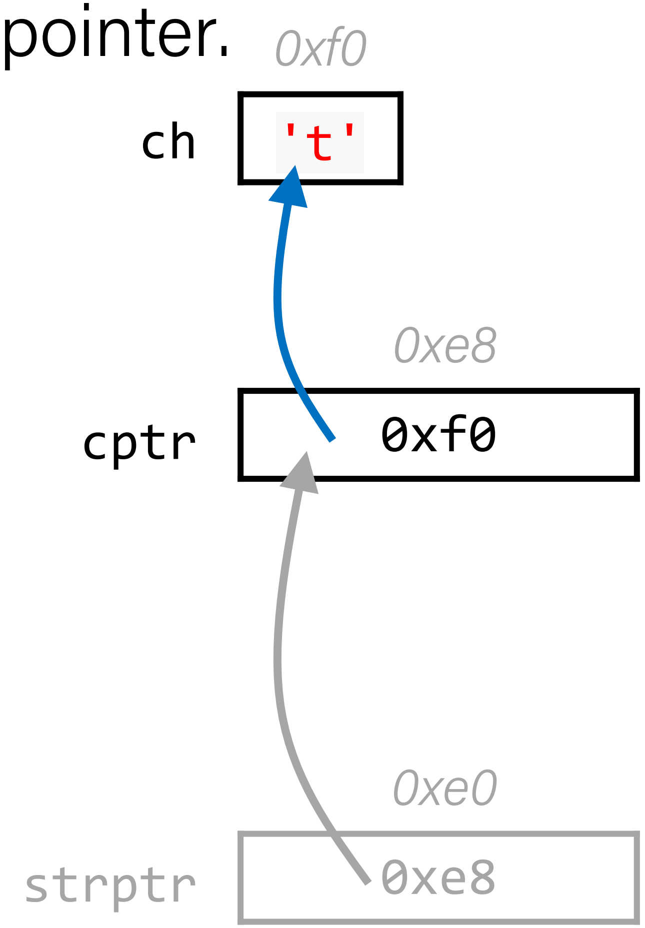
```
char ch = 'r';
ch = ch + 1;
```

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;
```

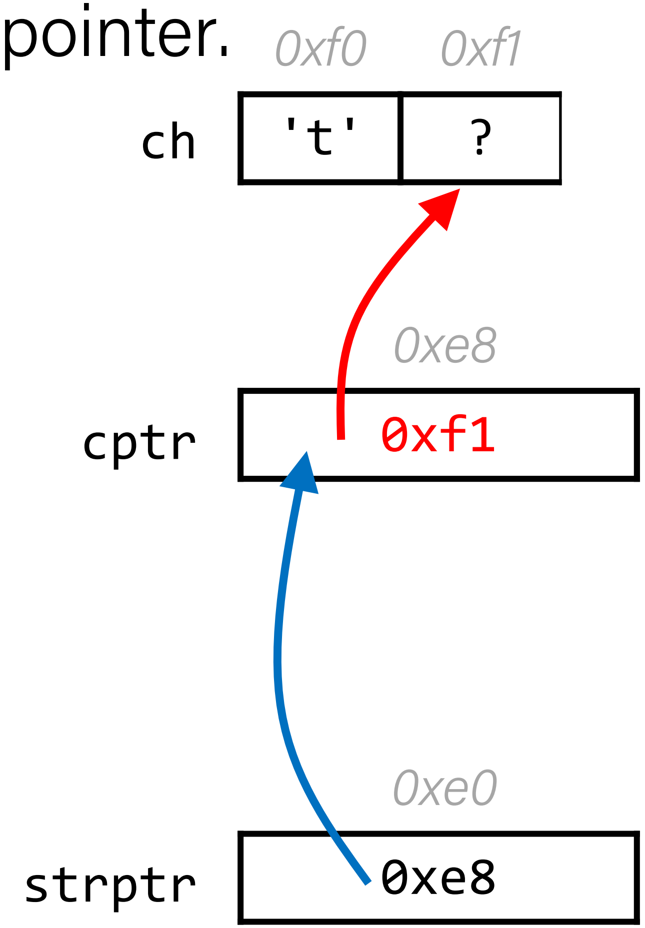
Increment value stored in ch

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

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

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

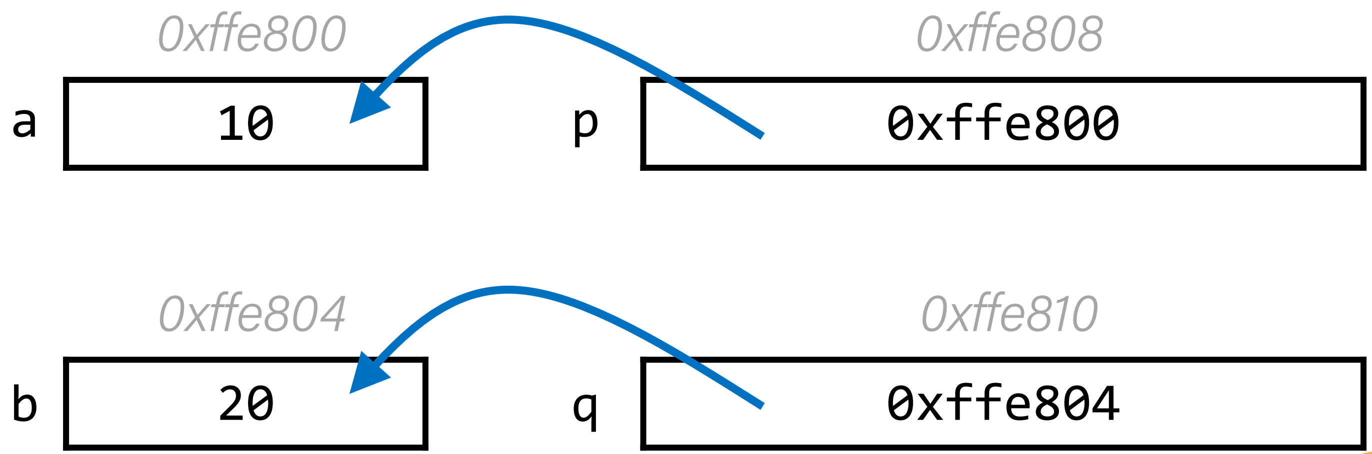
Increment value stored at memory address in cptr (increment address **pointed to**)



Pen and paper: A * Wars Story

```c
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 }
```

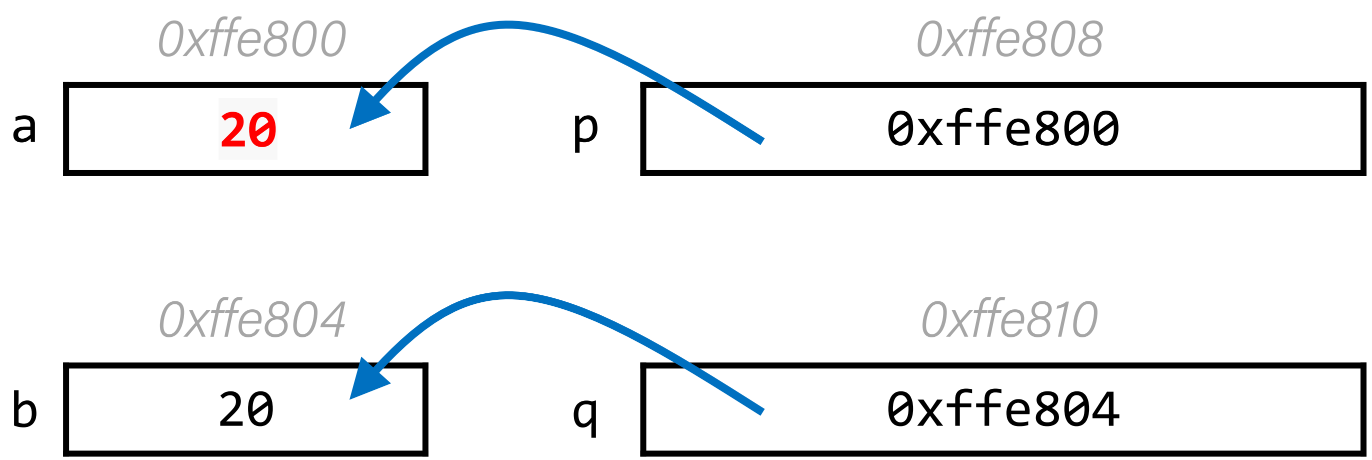
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



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 }
```

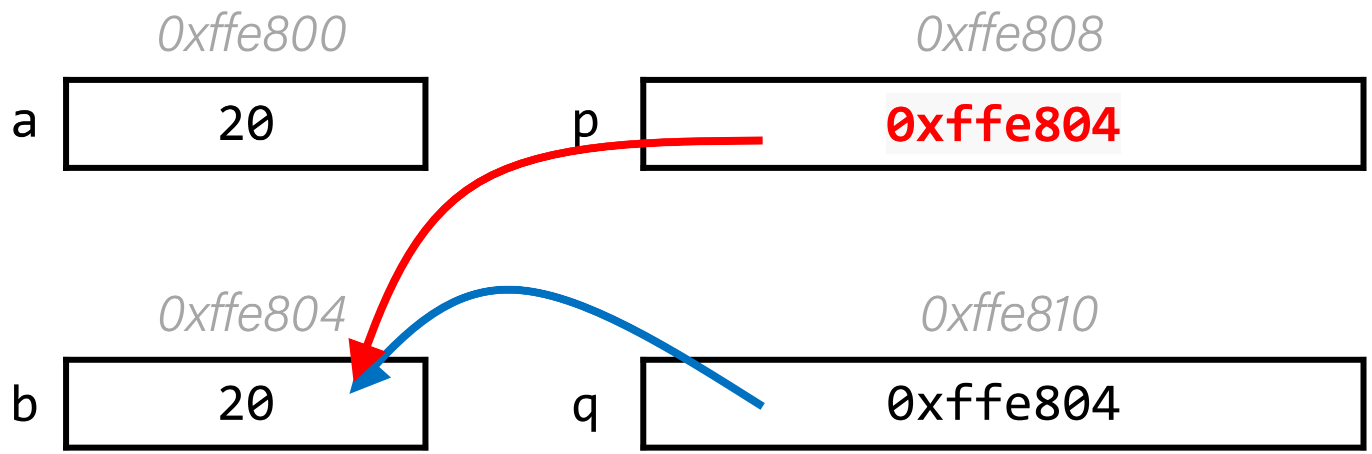
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.



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

- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Disclaimer: Slides for this lecture were borrowed from

—Nick Troccoli's Stanford CS107 class

Lecture Plan

- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

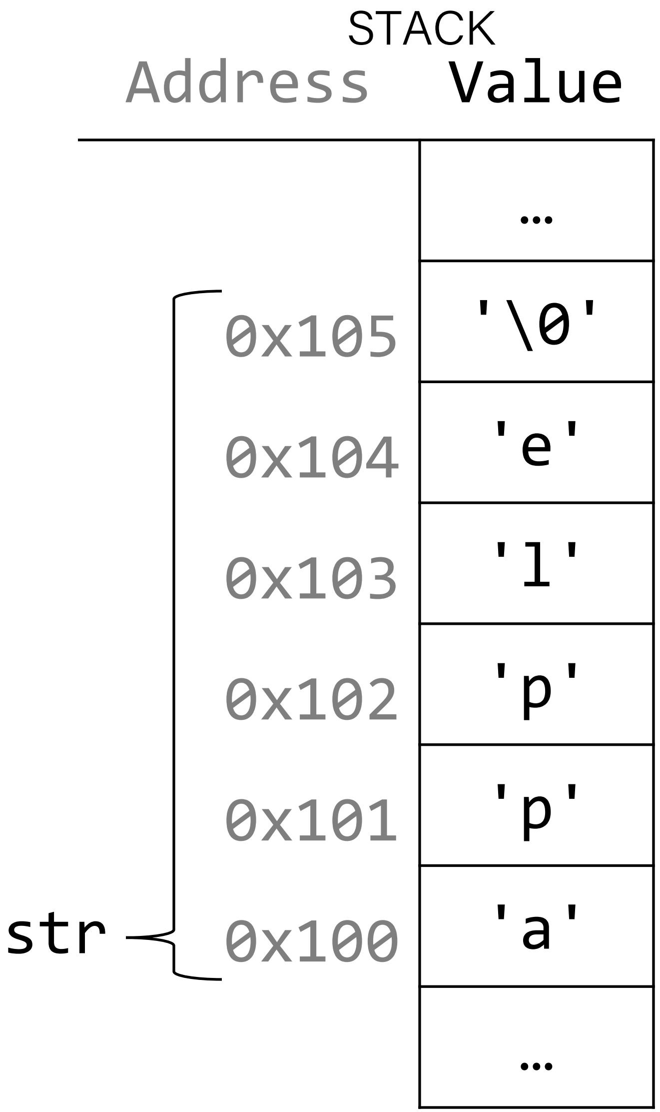
Arrays

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.

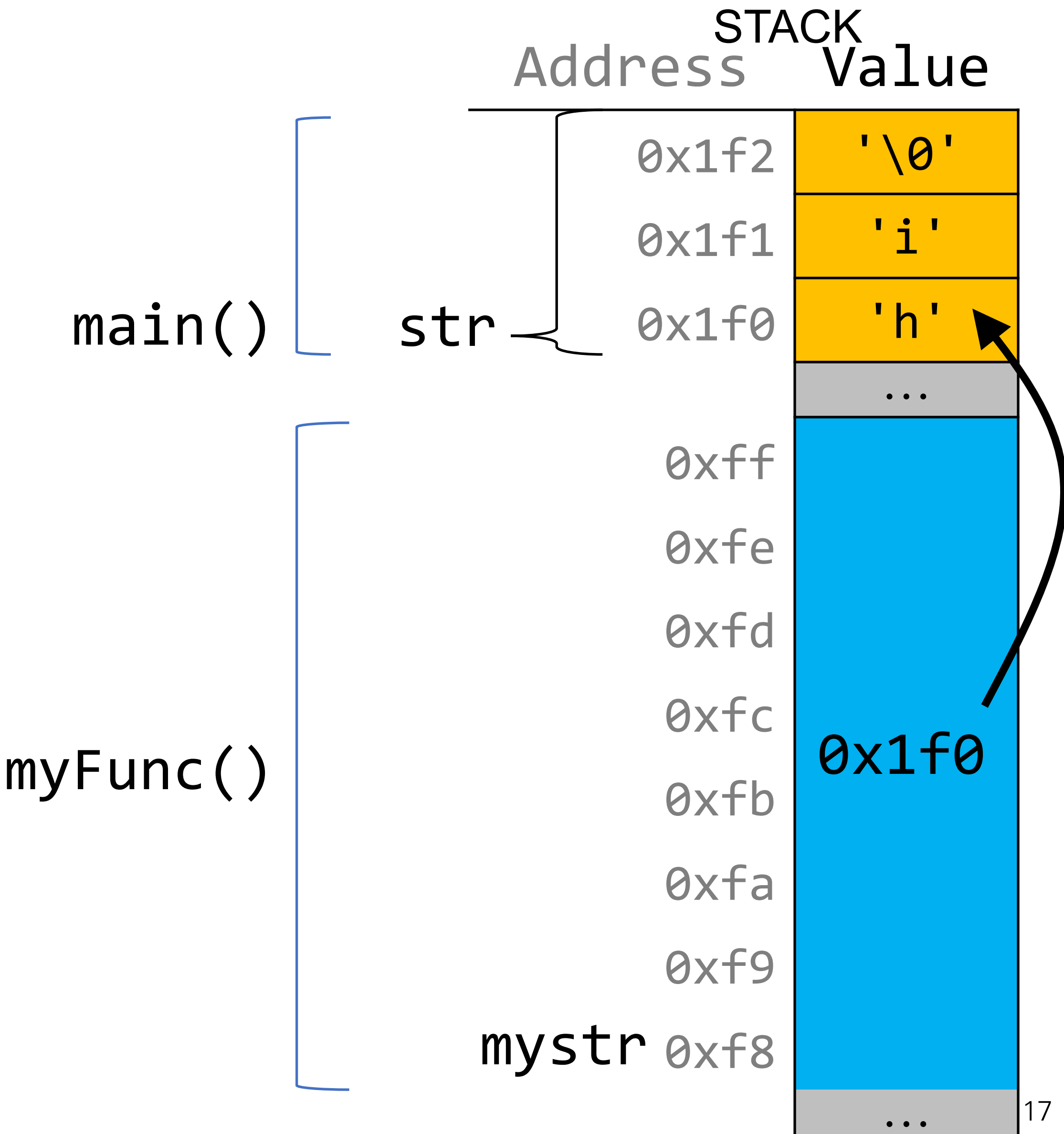
```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

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.

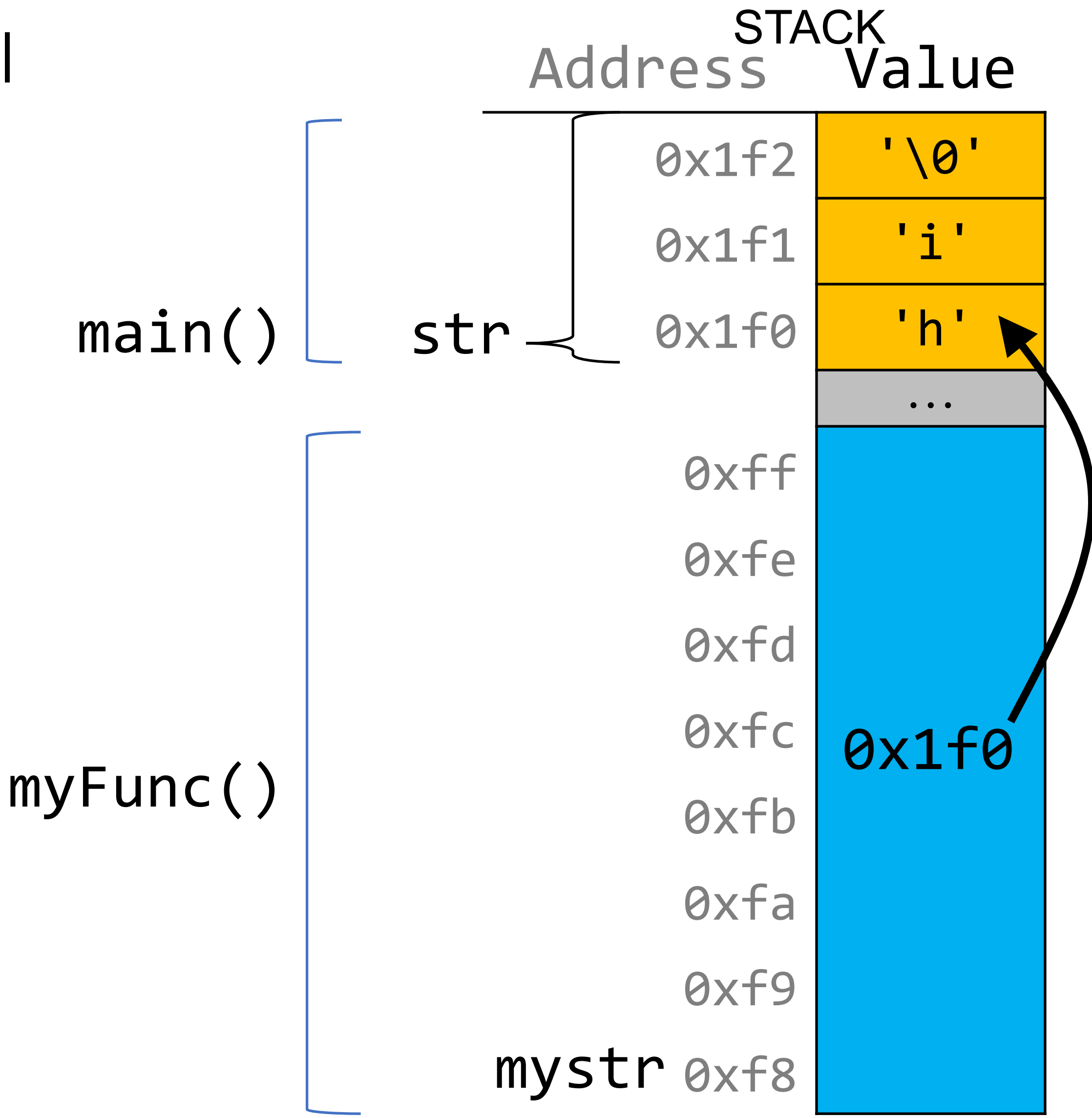
```
void myFunc(char *myStr) {
	...
}

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



Arrays as Parameters

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.

Lecture Plan

- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Arrays Of Pointers

You can make an array of pointers to e.g. group multiple strings together:

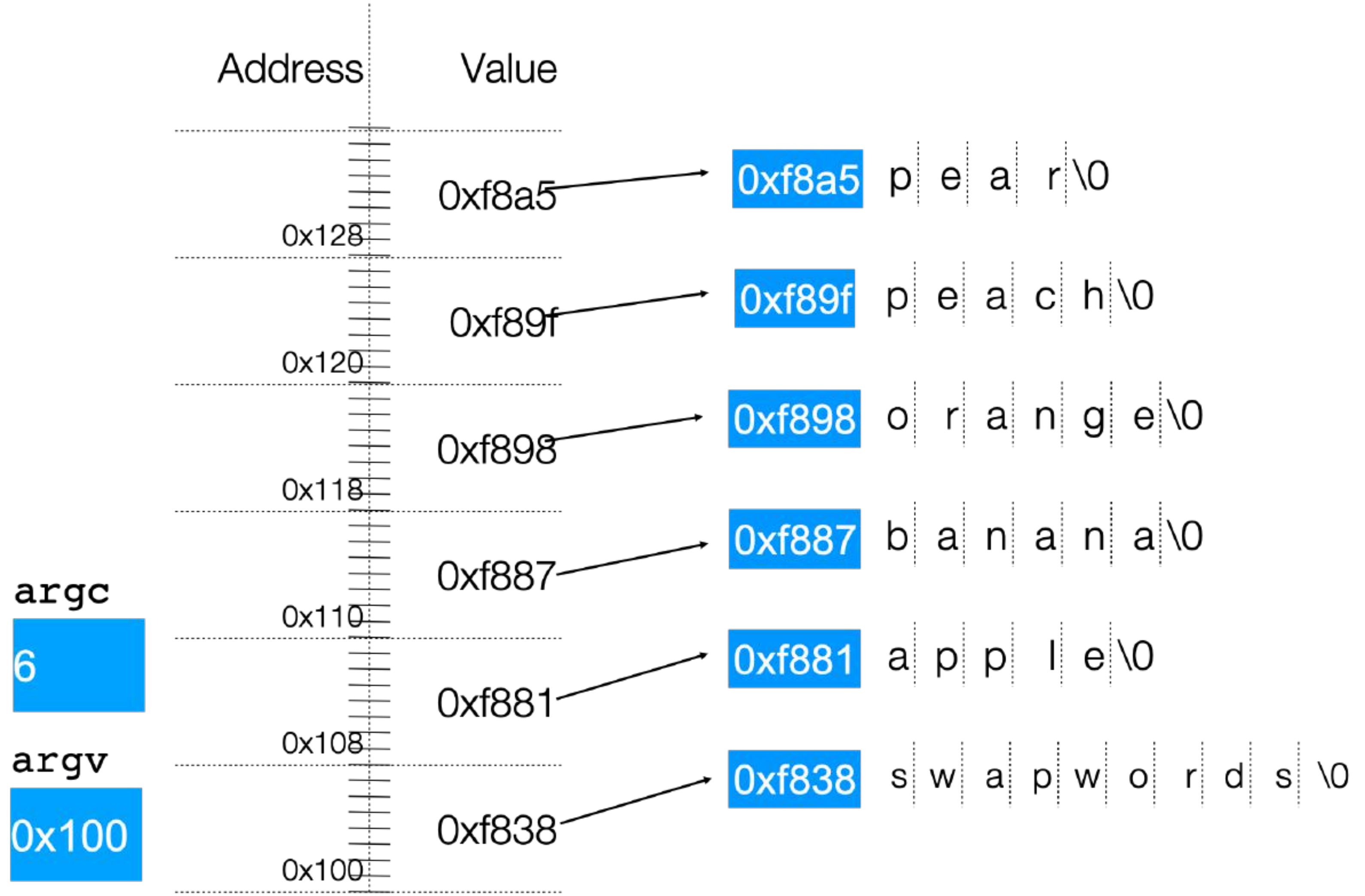
```
char *stringArray[5];   // space to store 5 char *s
```

This stores 5 **char *s**, *not* all of the characters for 5 strings!

```
char *str0 = stringArray[0];      // first char *
```

Arrays Of Pointers

```
./swapwords apple banana orange peach pear
```



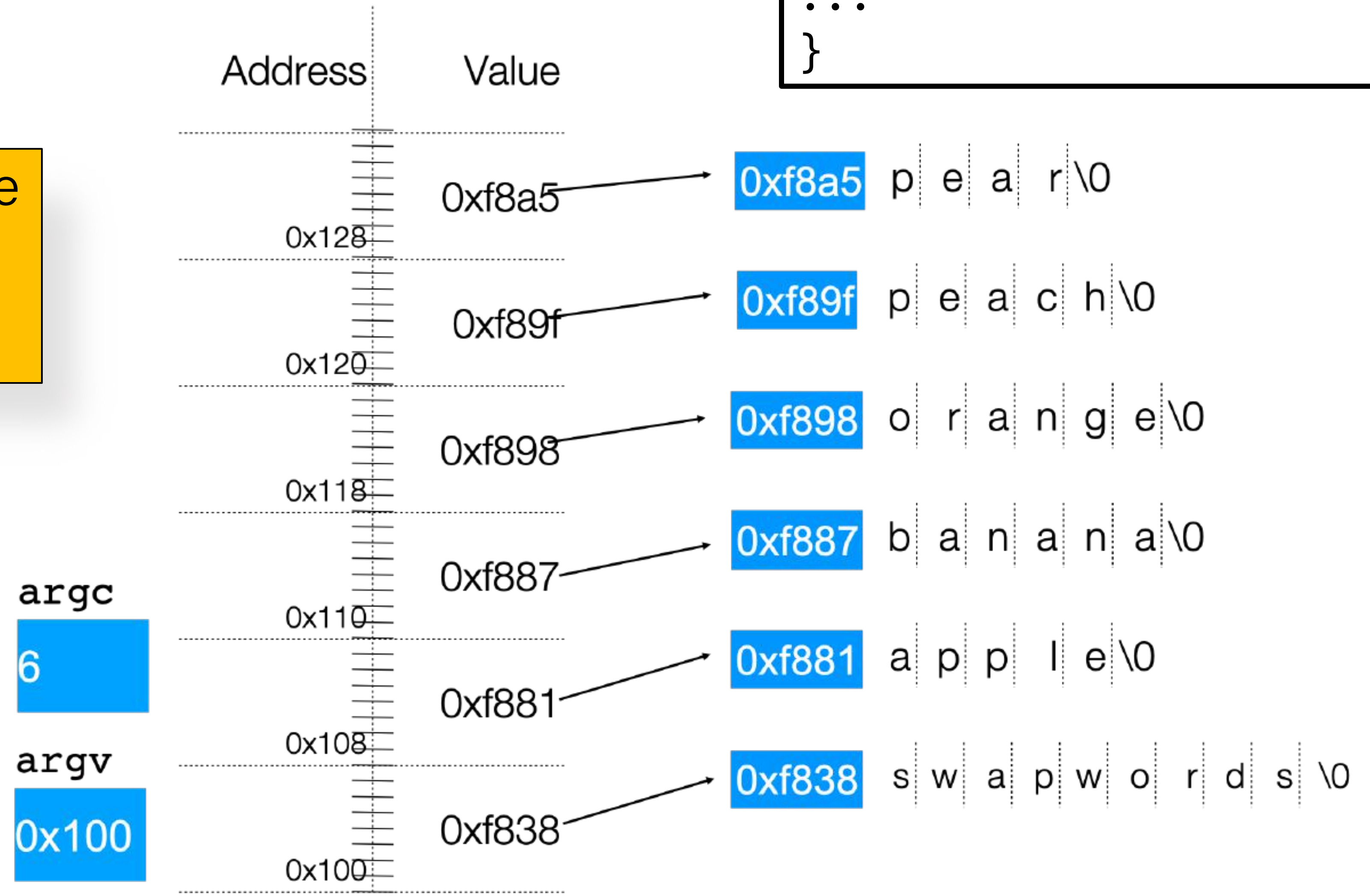
Arrays Of Pointers

```
./swapwords apple banana orange peach pear
```

```c
int main(int argc, char *argv[]) {
...
}
```

swapwords.c

What is the value of `argv[2]` in this diagram?



Lecture Plan

- Arrays in Memory
- Arrays of Pointers
- 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
```

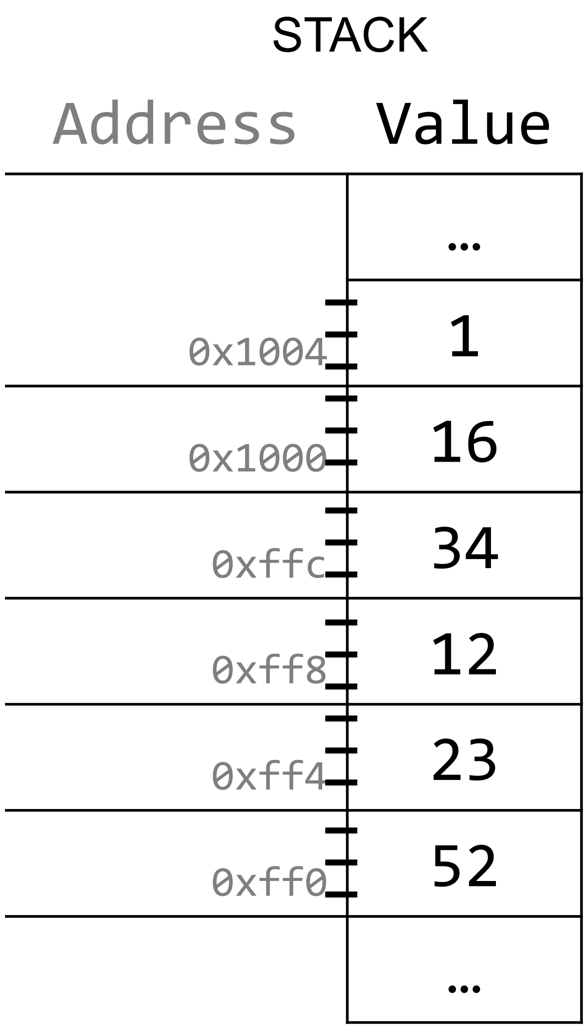
DATA SEGMENT

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
```



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'
```

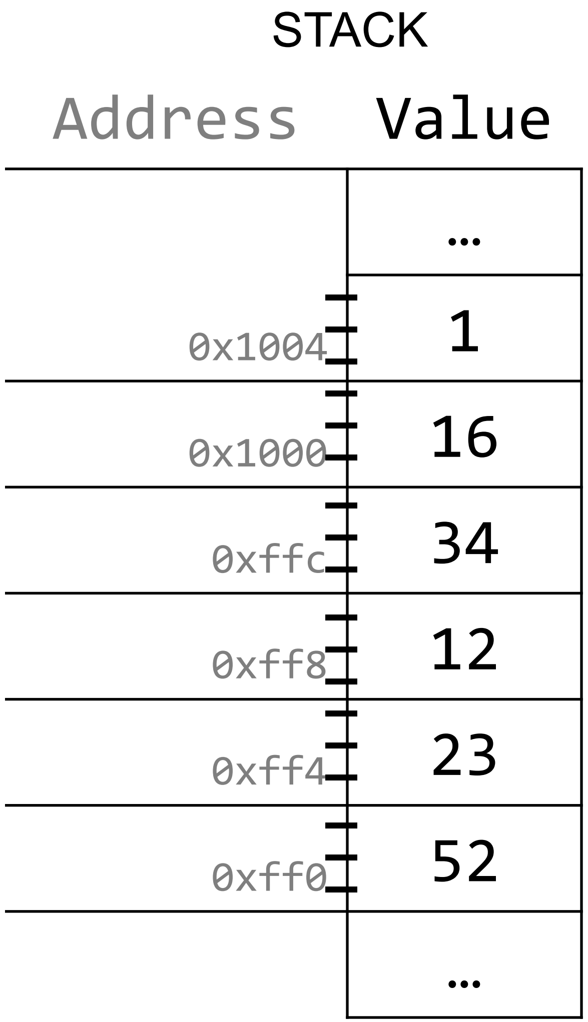
DATA SEGMENT

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
```



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?

```c
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:

```
ptr + i        ⇔    &ptr[i]
*(ptr + i)     ⇔    ptr[i]
```

⚠️ 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.

Example: Pointer arithmetic

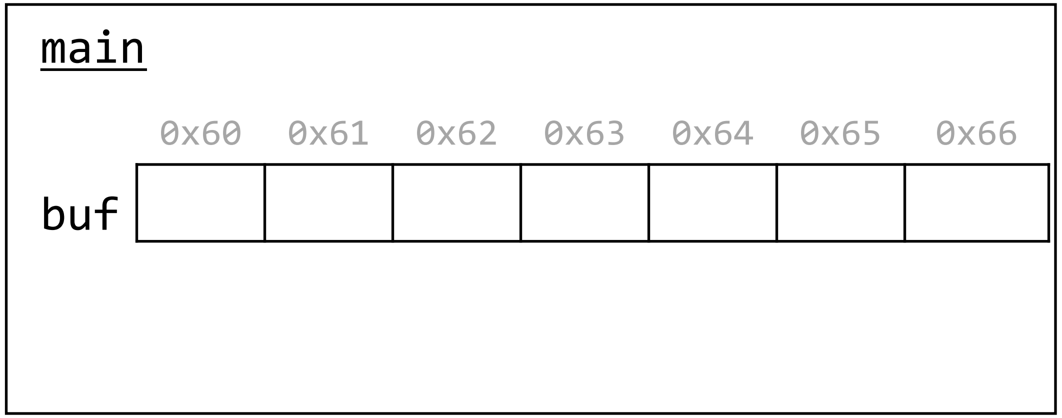
```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

int main(int argc, const char *argv[]) {
    char buf[] = "Monday";
    printf("before func: %s\n", buf);
    func(buf);
    printf("after  func: %s\n", buf);
    return 0;
}
```

- Will there be a compile error/segfault?
- If no errors, what is printed?

- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

Example: Pointer arithmetic

```c
void func(char *str) {
    str[0] = 'S';
    str++;
    *str = 'u';
    str = str + 3;
    str[-2] = 'm';
}

int main(int argc, const char *argv[]) {
    char buf[] = "Monday";
    printf("before func: %s\n", buf);
    func(buf);
    printf("after  func: %s\n", buf);
    return 0;
}
```



- **Draw memory diagrams!**
- **Pointers** store addresses! Make up addresses if it helps your mental model.

Code study: strncpy

```
STRCPY(3)                 Linux Programmer's Manual                 STRCPY(3)

DESCRIPTION
       The strncpy() function is similar, except that at most n bytes of src  are
       copied.  Warning: If there is no null byte among the first n bytes of src,
       the string placed in dest will not be null-terminated.

       If the length of src is less than  n,  strncpy()  writes  additional  null
       bytes to dest to ensure that a total of n bytes are written.

       A simple implementation of strncpy() might be:
```

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
        dest[i] = src[i];
    for ( ; i < n; i++)
         dest[i] = '\0';
    return dest;
}
```

What happens if we call `strncpy(buf, str, 5);`?

Code study: strncpy

```
STRCPY(3)                 Linux Programmer's Manual                 STRCPY(3)

DESCRIPTION
       The strncpy() function is similar, except that at most n bytes of src  are
       copied.  Warning: If there is no null byte among the first n bytes of src,
       the string placed in dest will not be null-terminated.

       If the length of src is less than  n,  strncpy()  writes  additional  null
       bytes to dest to ensure that a total of n bytes are written.

       A simple implementation of strncpy() might be:
```

	0x60	0x61	0x62	0x63	0x64	0x65	0x66
buf	'M'	'o'	'n'	'd'	'a'	'y'	'\0'

	0x58	0x59	0x5a	0x5b
str	'F'	'r'	'i'	'\0'

```c
char *strncpy(char *dest, const char *src, size_t n) {
    size_t i;
    for (i = 0; i < n && src[i] != '\0'; i++)
       dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

dest	
src	
n	5
i	

What happens if we call `strncpy(buf, str, 5);`?

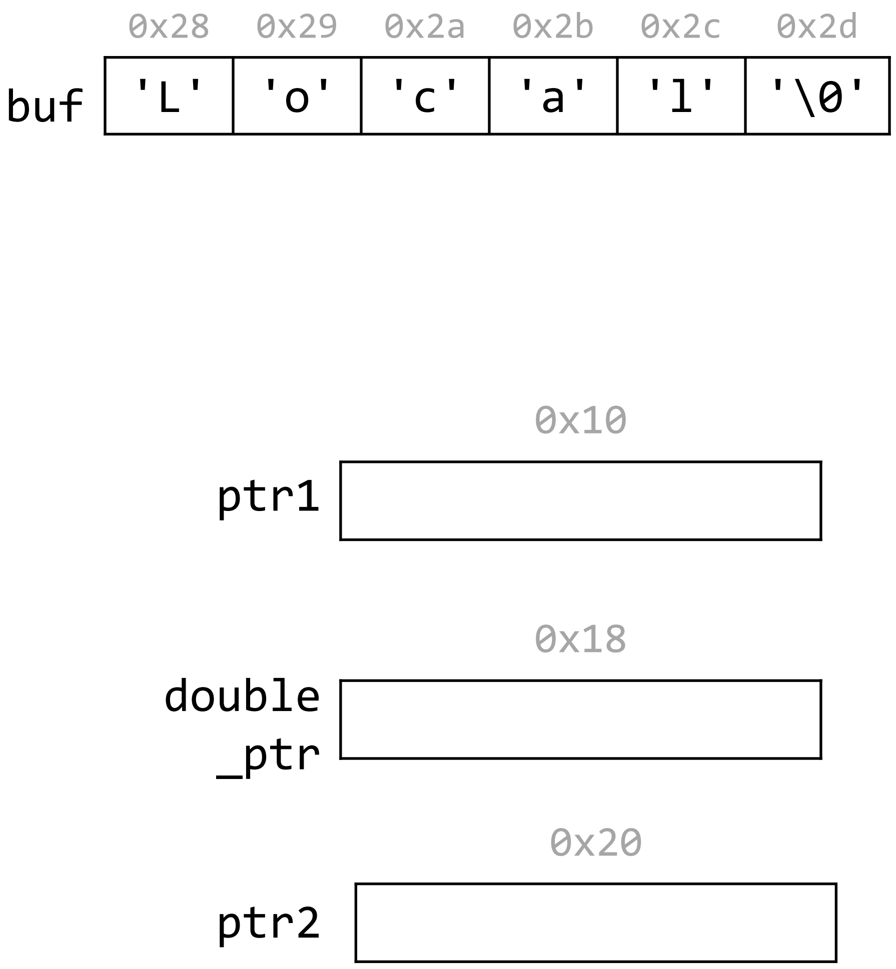
Bonus: Tricky addresses

```
void tricky_addresses() {
  char buf[] = "Local";
  char *ptr1 = buf;
  char **double_ptr = &ptr1;
  printf("ptr1's value:     %p\n", ptr1);
  printf("ptr1’s deref:     %c\n", *ptr1);
  printf("        address:   %p\n", &ptr1);
  printf("double_ptr value: %p\n", double_ptr);
  printf("buf's address:    %p\n", &buf);

  char *ptr2 = &buf;
  printf("ptr2's value:     %s\n", ptr2);
}
```

What is stored in each variable?

Bonus: Tricky addresses

```
1  void tricky_addresses() {
2    char buf[] = "Local";
3    char *ptr1 = buf;
4    char **double_ptr = &ptr1;
5    printf("ptr1's value:     %p\n", ptr1);
6    printf("ptr1’s deref:     %c\n", *ptr1);
7    printf("       address:   %p\n", &ptr1);
8    printf("double_ptr value: %p\n", double_ptr);
9    printf("buf's address:    %p\n", &buf);

10   char *ptr2 = &buf;
11   printf("ptr2's value:     %s\n", ptr2);
12 }
```



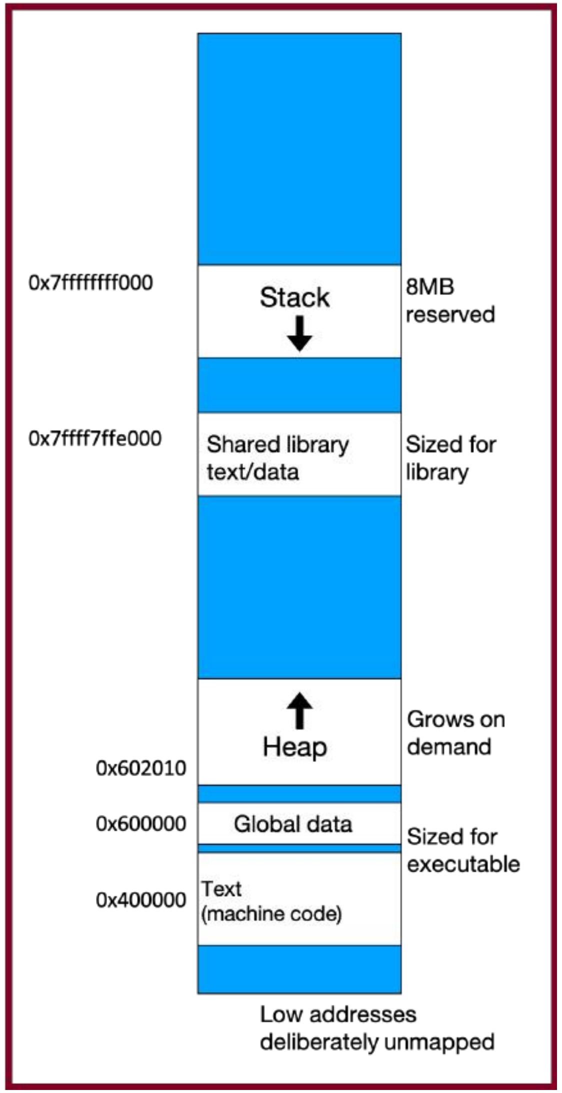
While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

Lecture Plan

- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

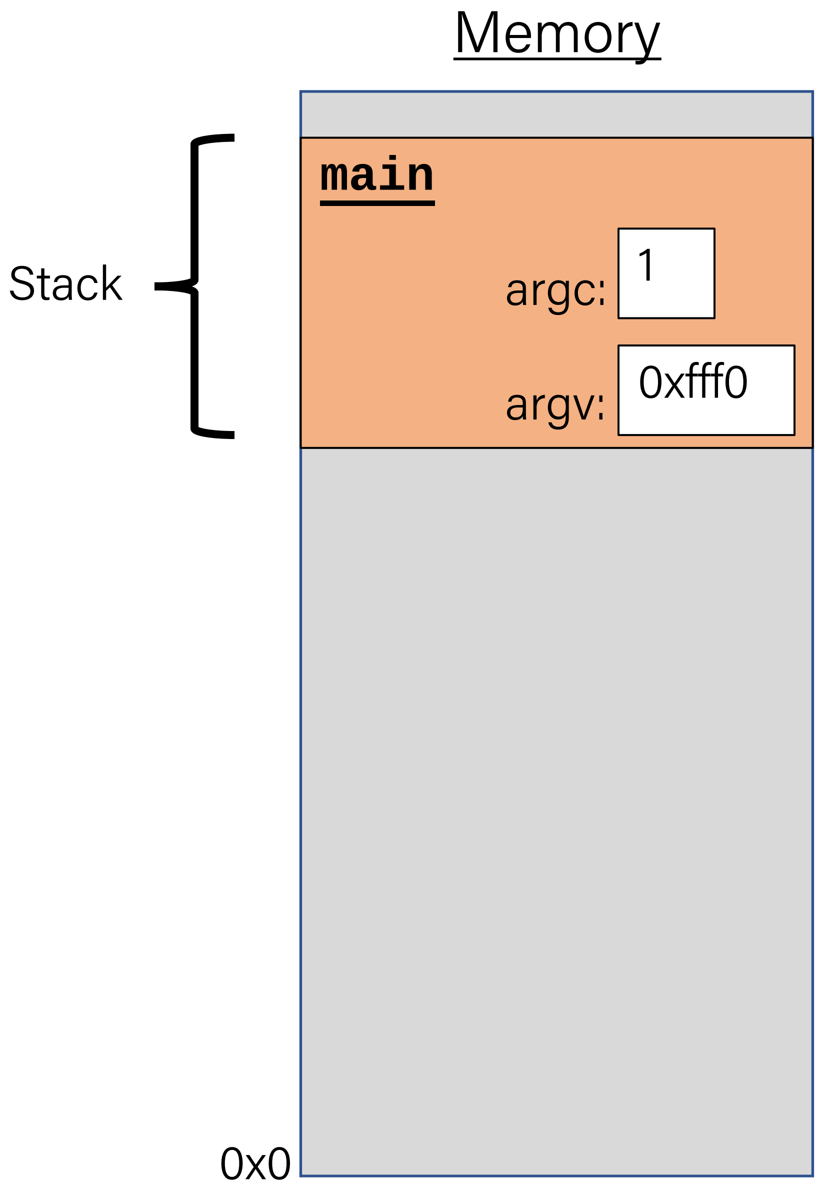
Memory Layout

- 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.



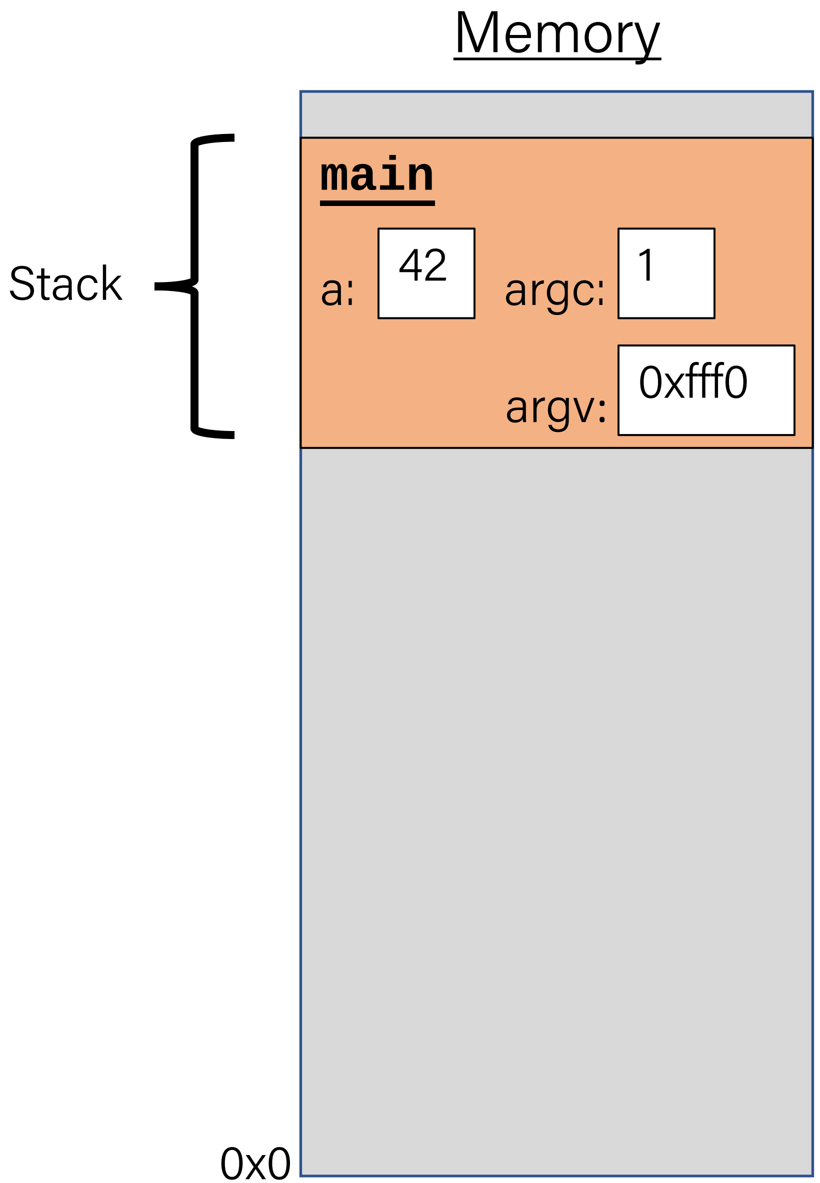
The Stack

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

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

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



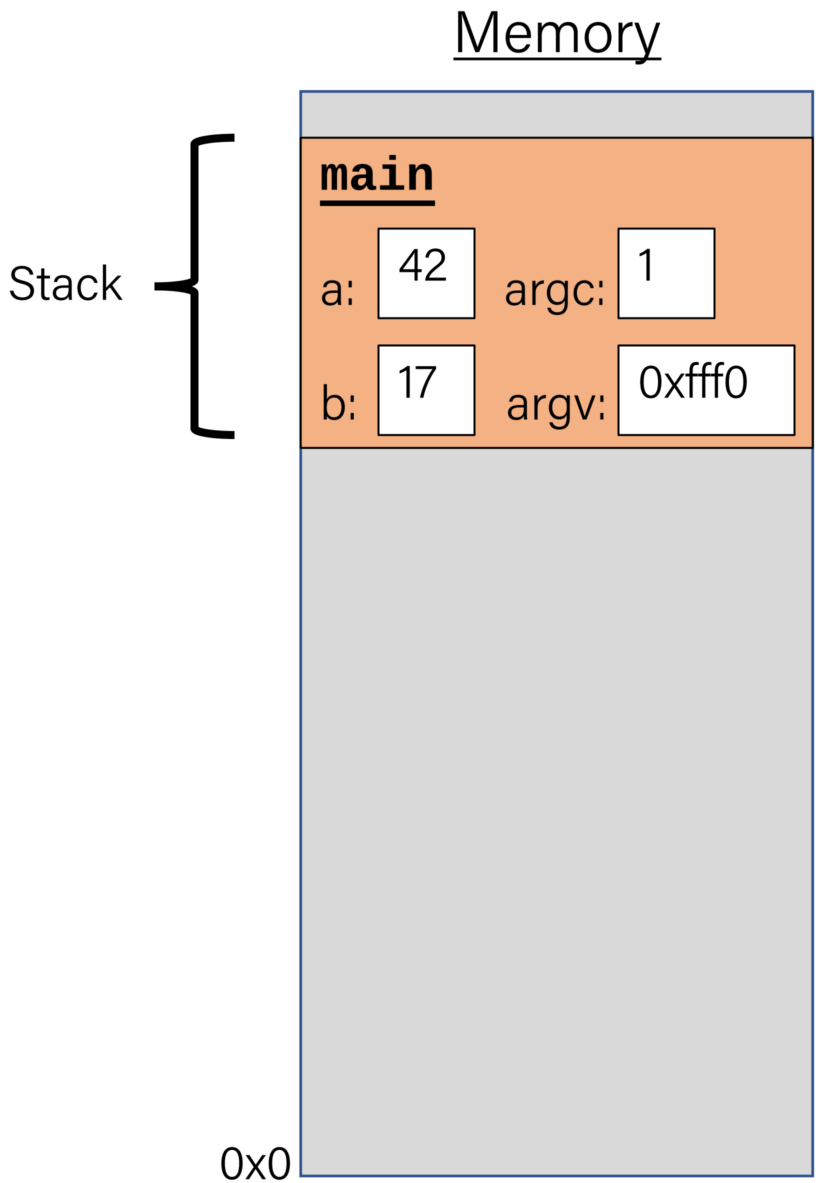
The Stack

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

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

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



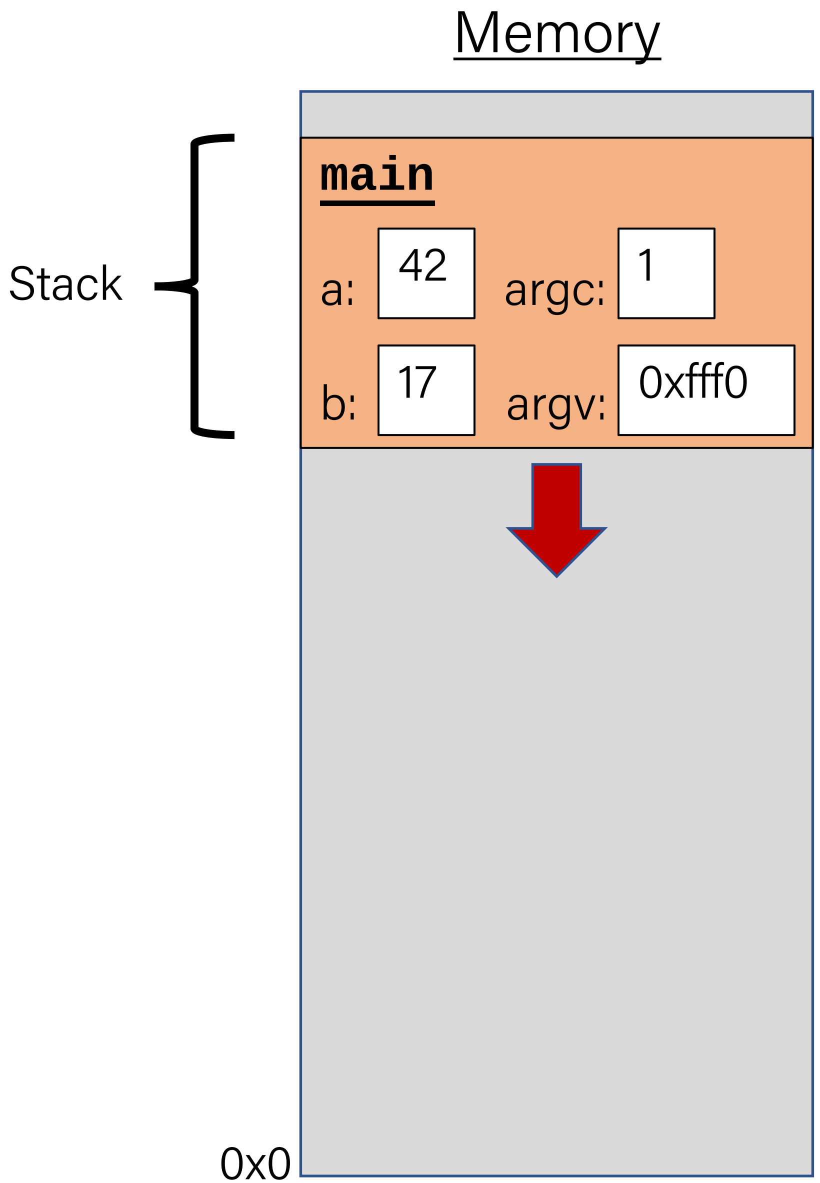
The Stack

```
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;
}
```



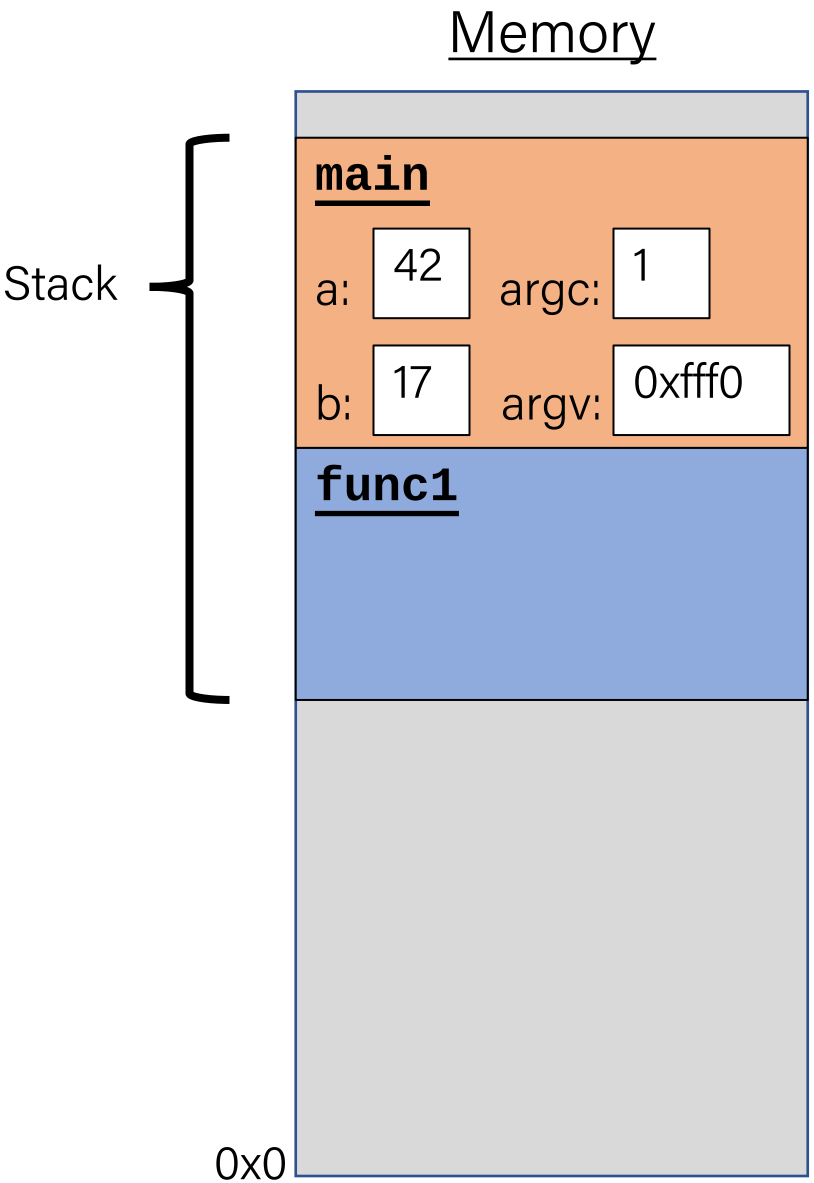
The Stack

```
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;
}
```



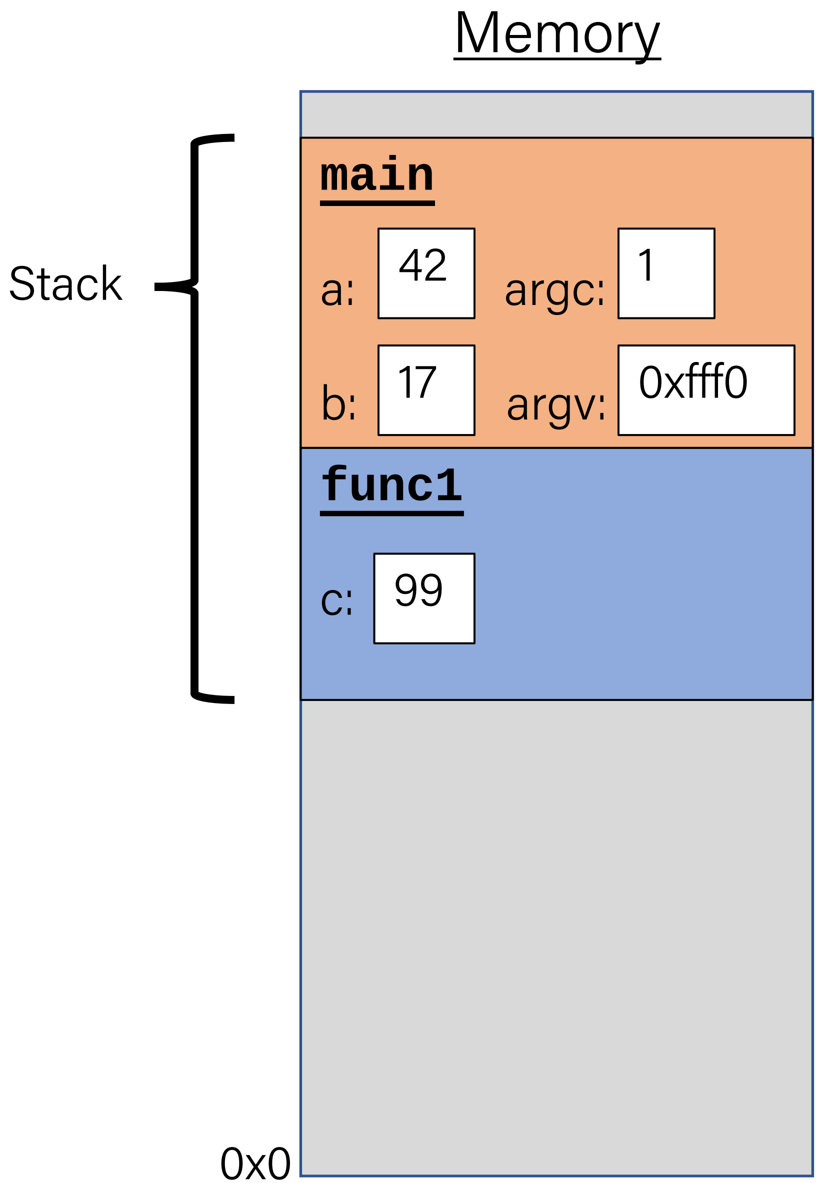
The Stack

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

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

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



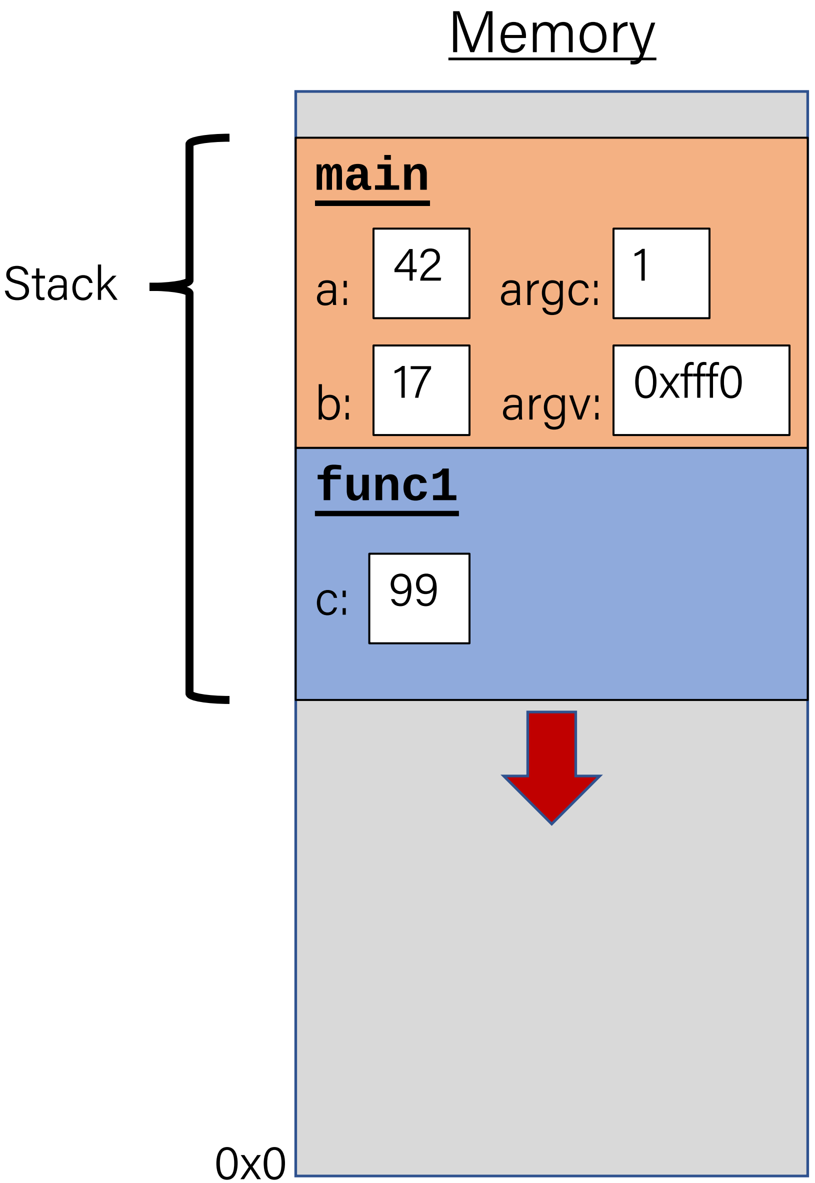
The Stack

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

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

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



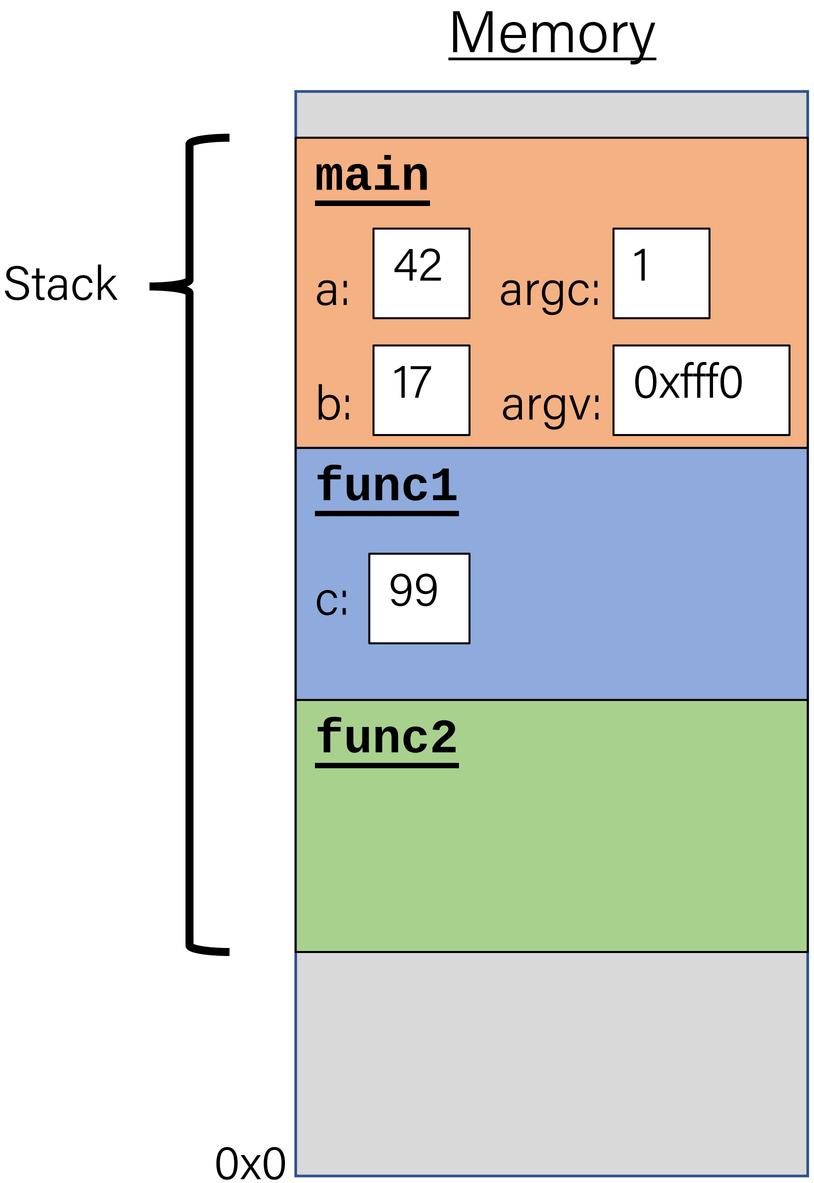
The Stack

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

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

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



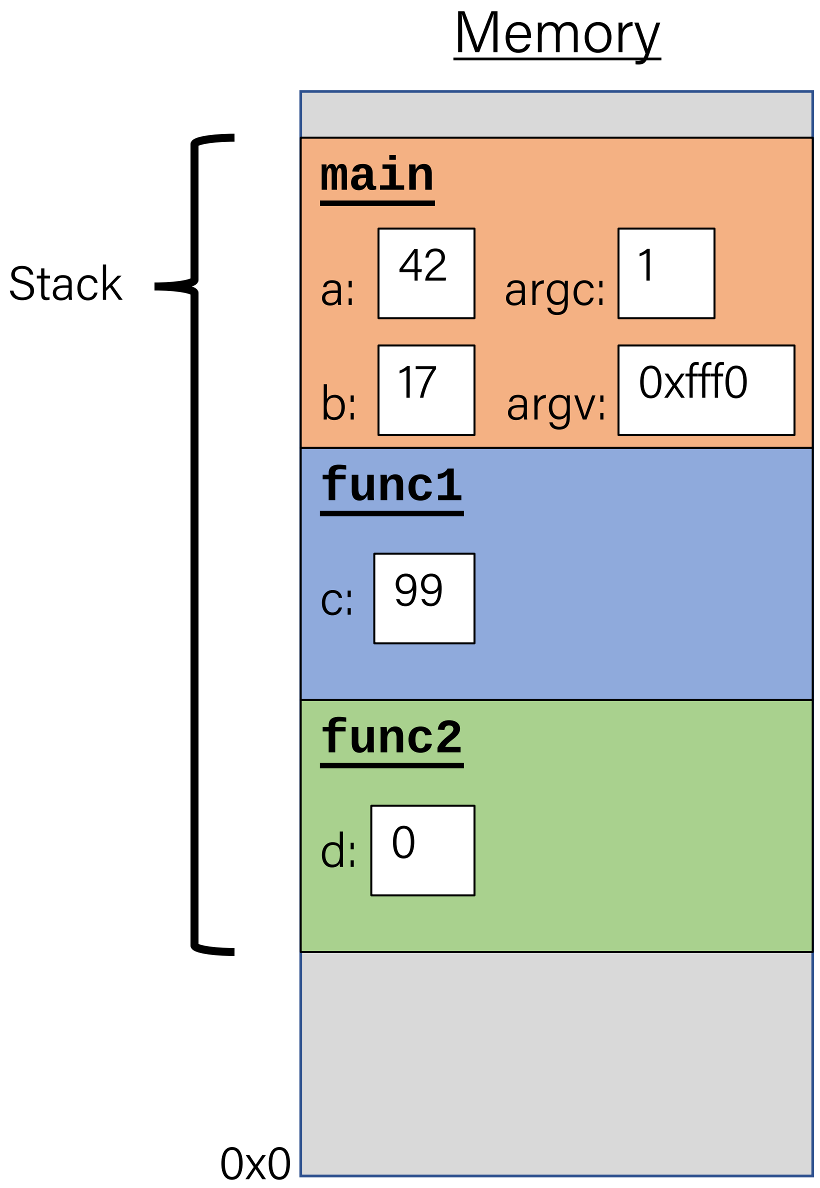
The Stack

```
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;
}
```



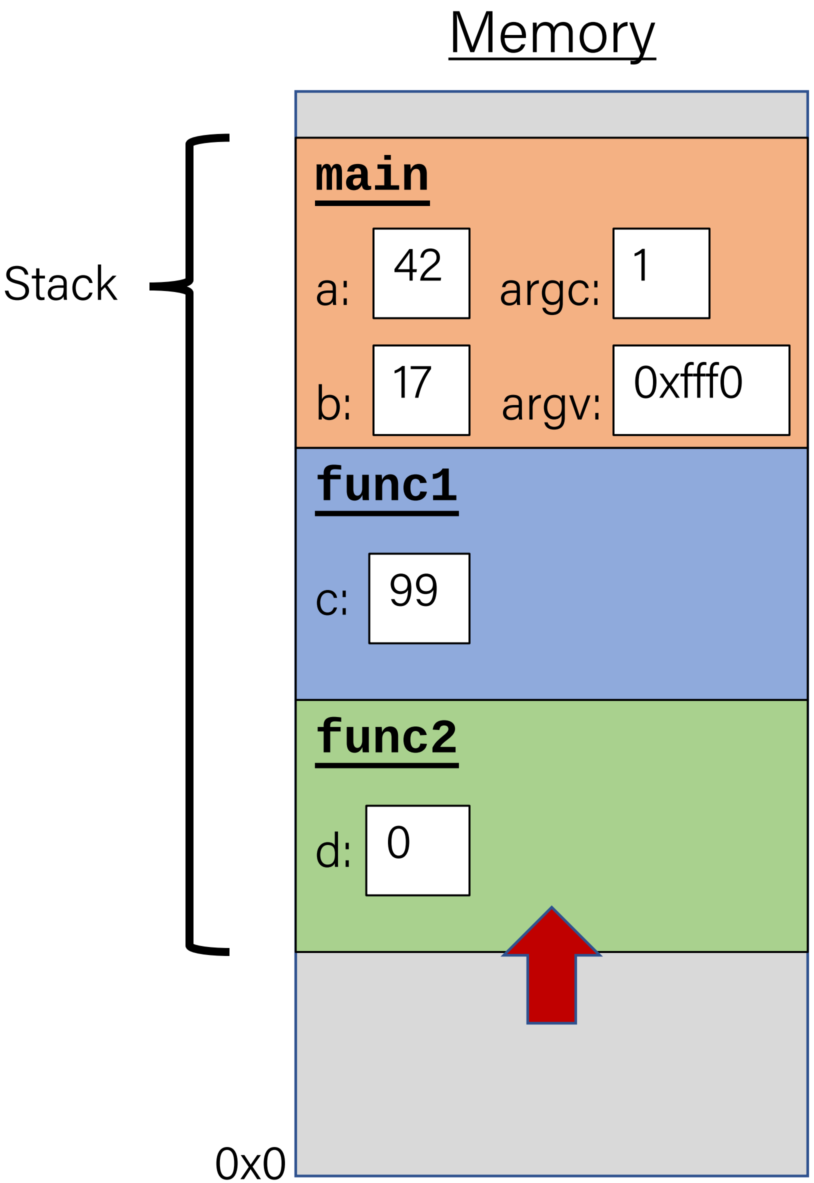
The Stack

```
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;
}
```



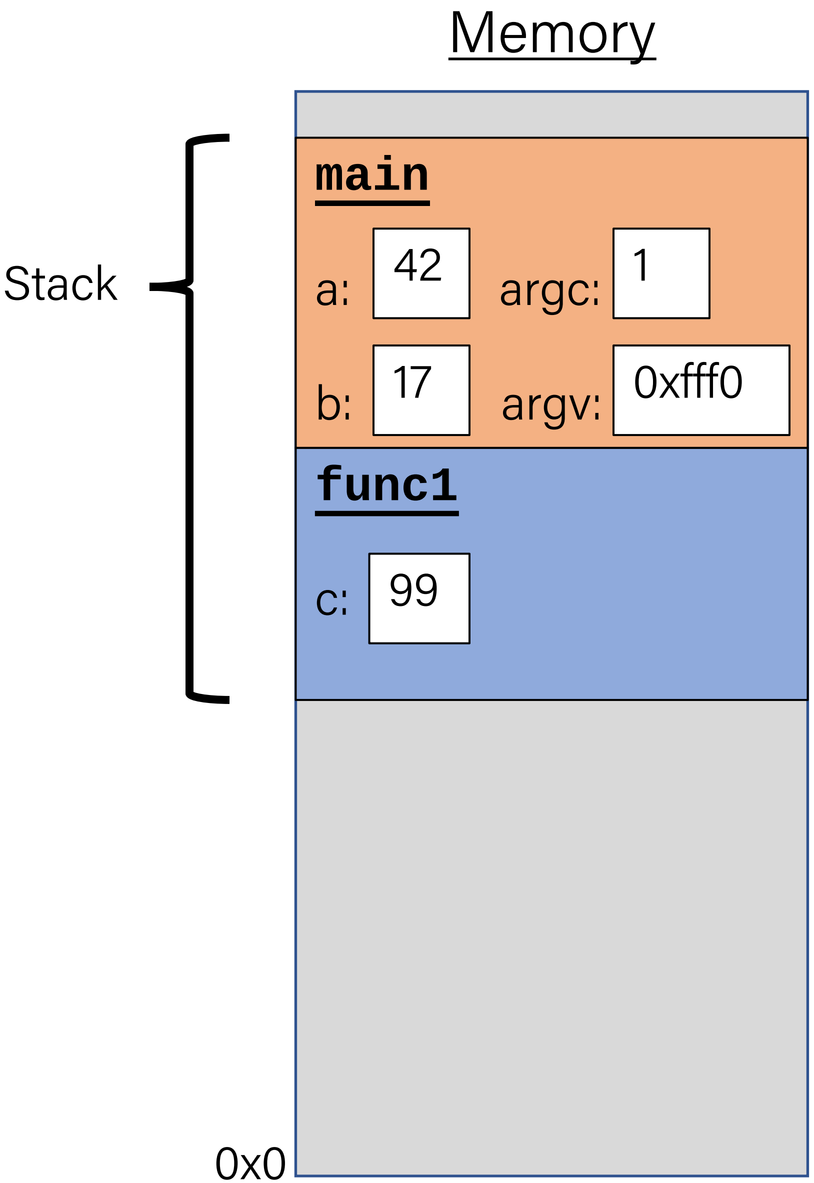
The Stack

```
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;
}
```



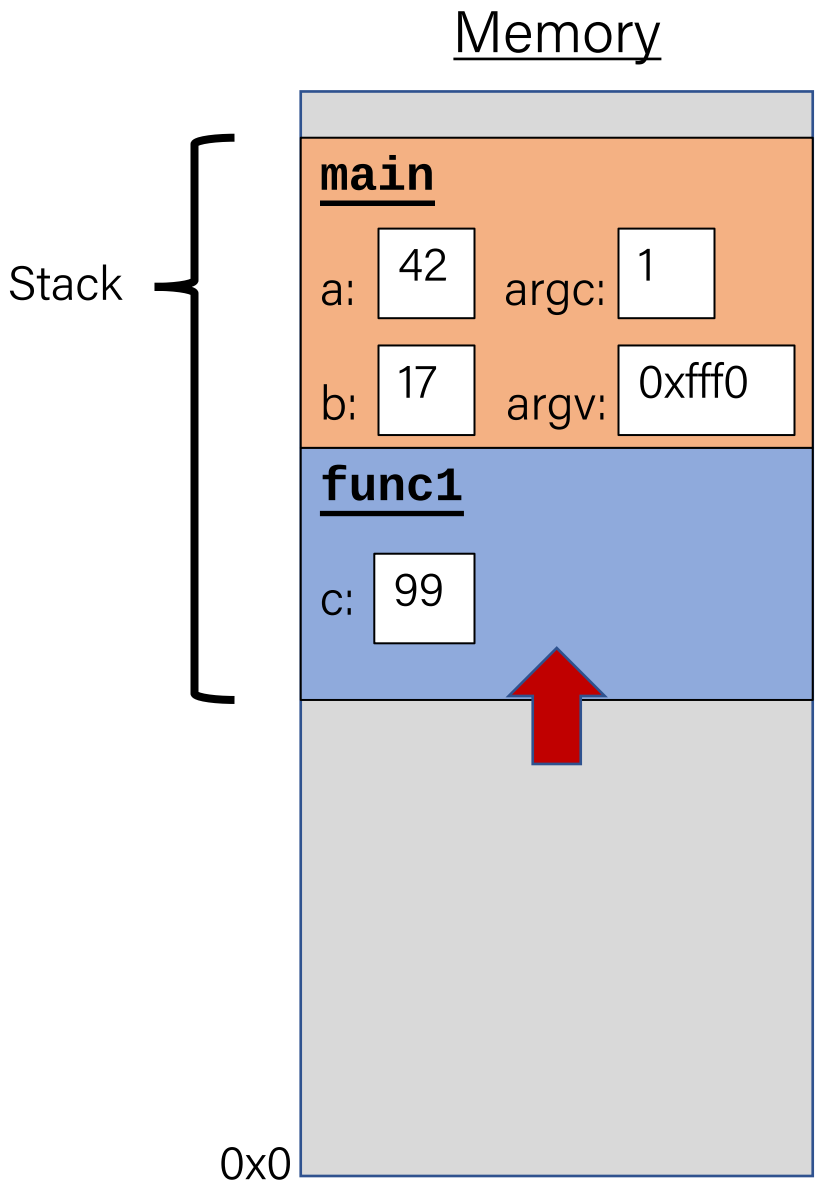
The Stack

```
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;
}
```



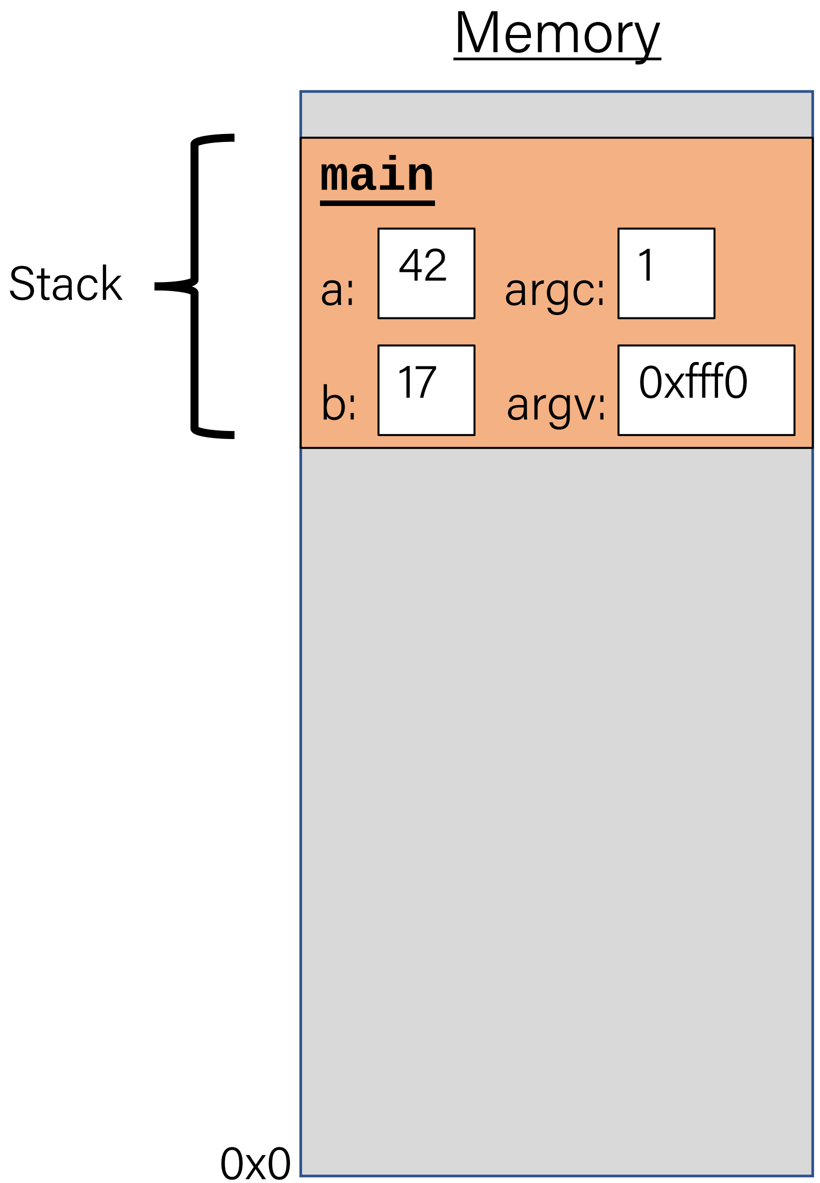
The Stack

```
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;
}
```



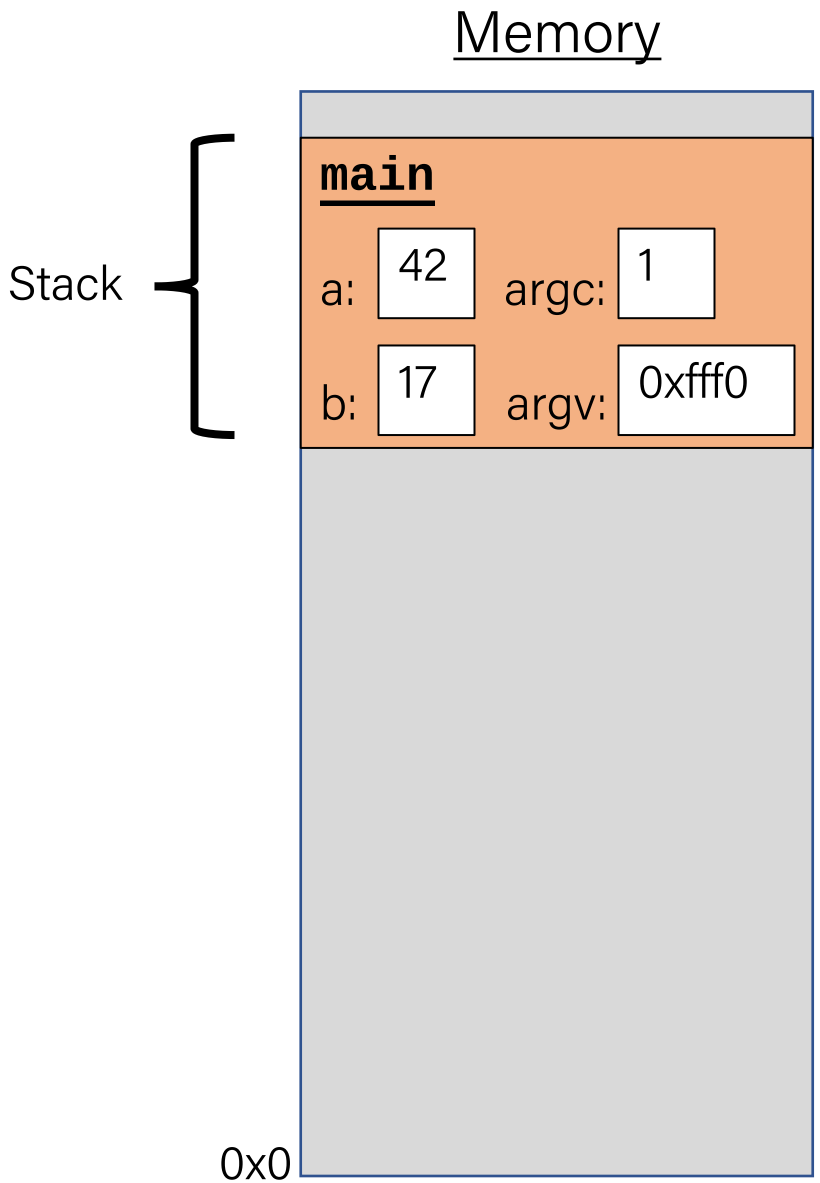
The Stack

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

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

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



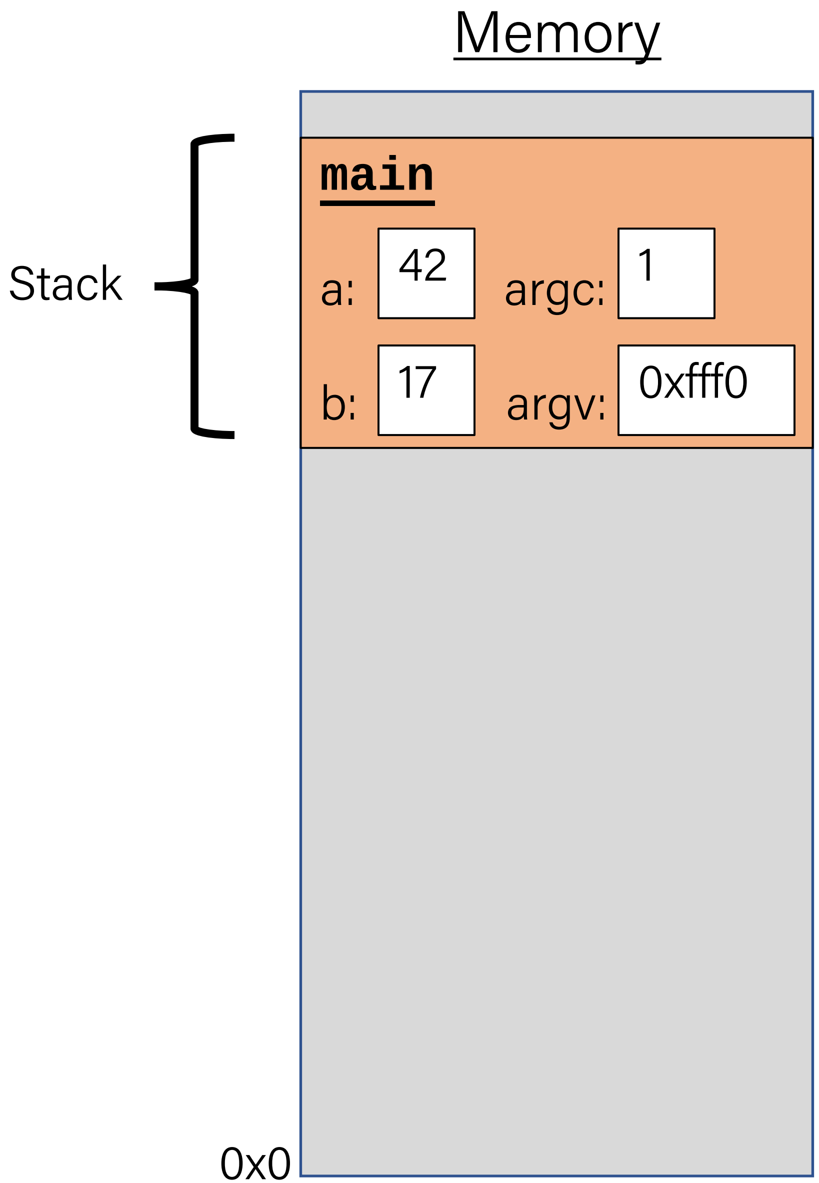
The Stack

```
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;
}
```



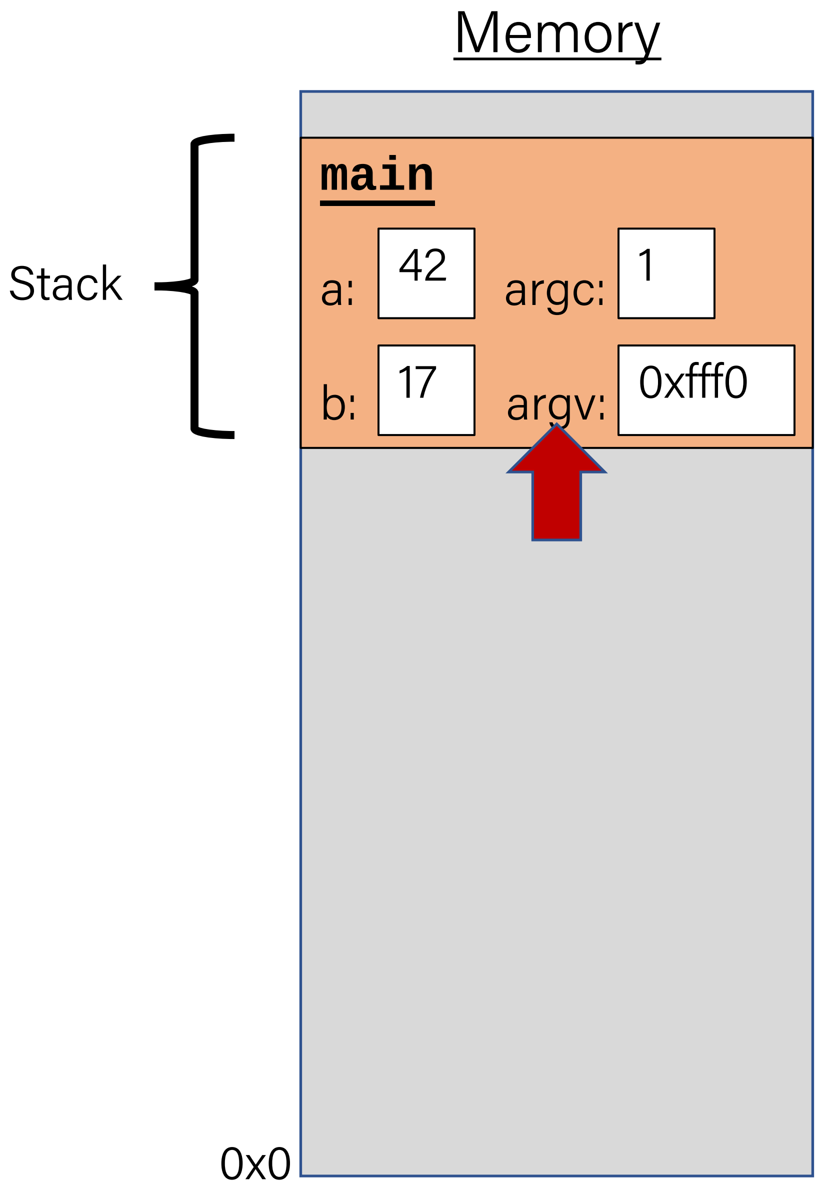
The Stack

```
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;
}
```



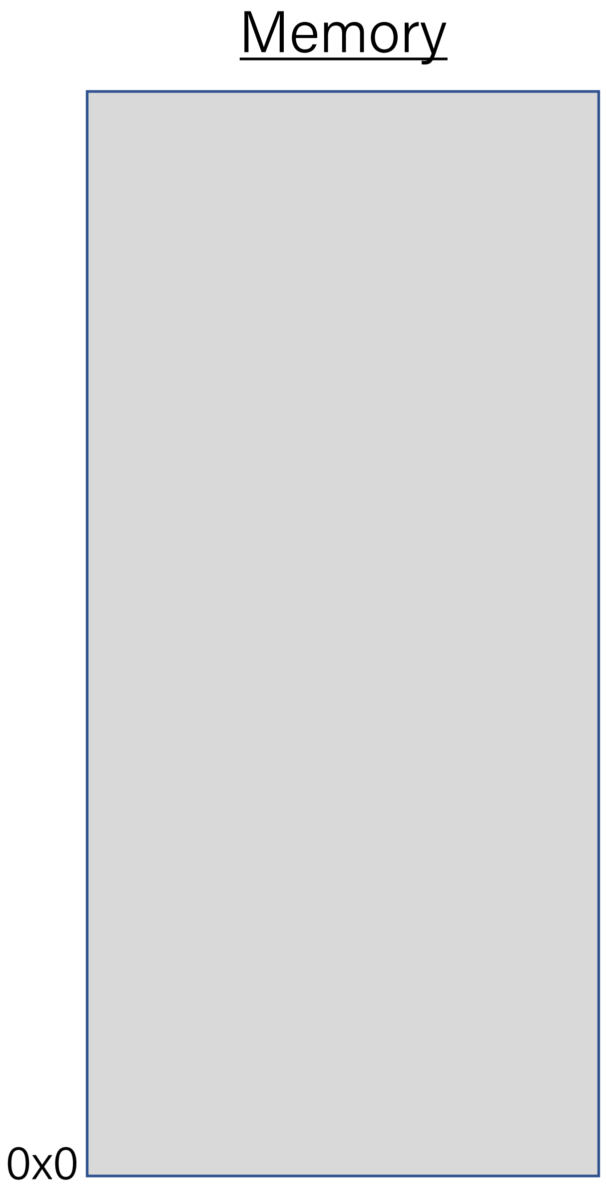
The Stack

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

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

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



The Stack

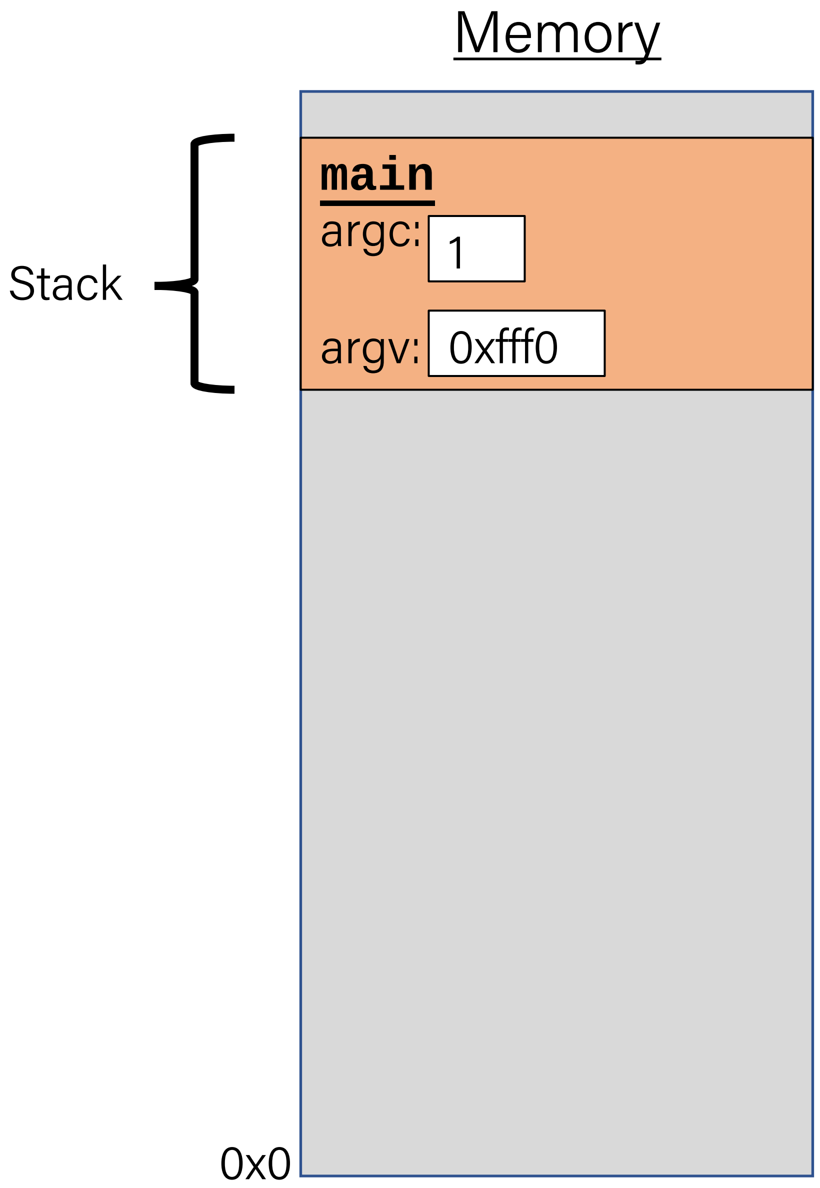
```c
void func2() {
    int d = 0;
}

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

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



The Stack

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

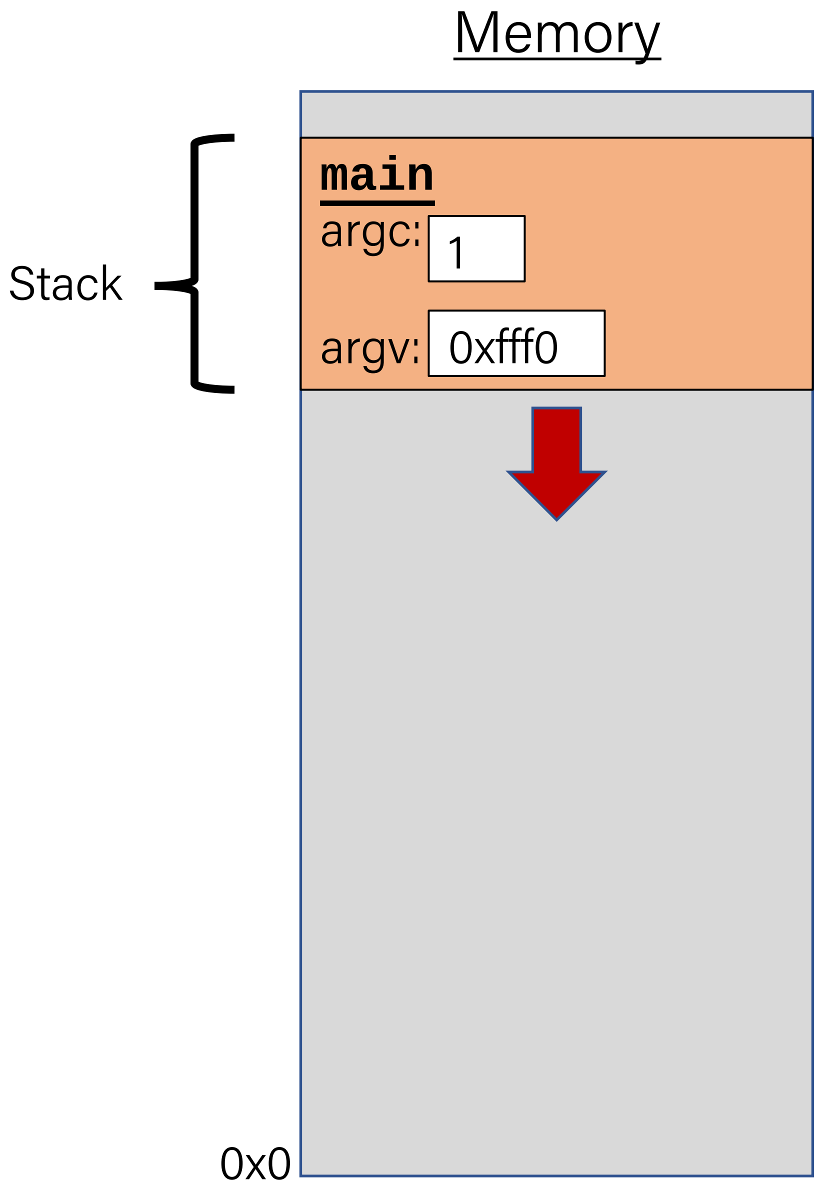
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

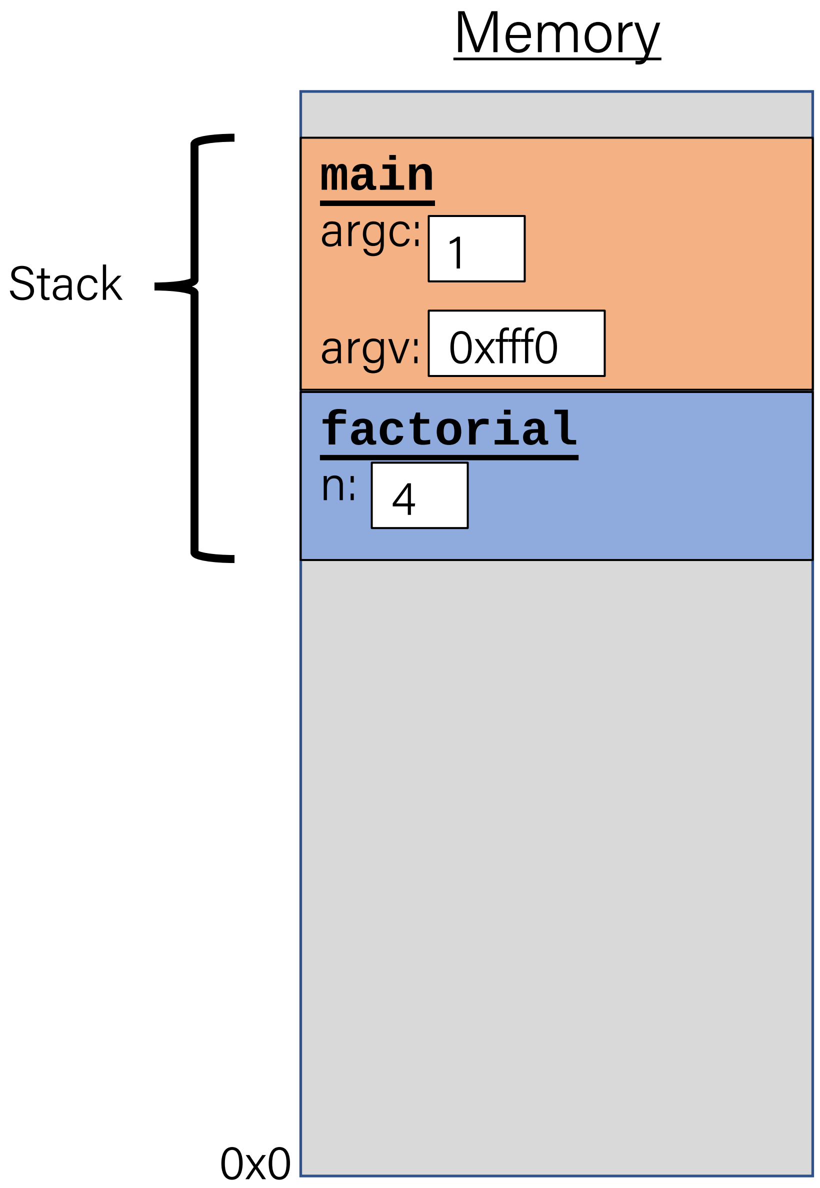
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

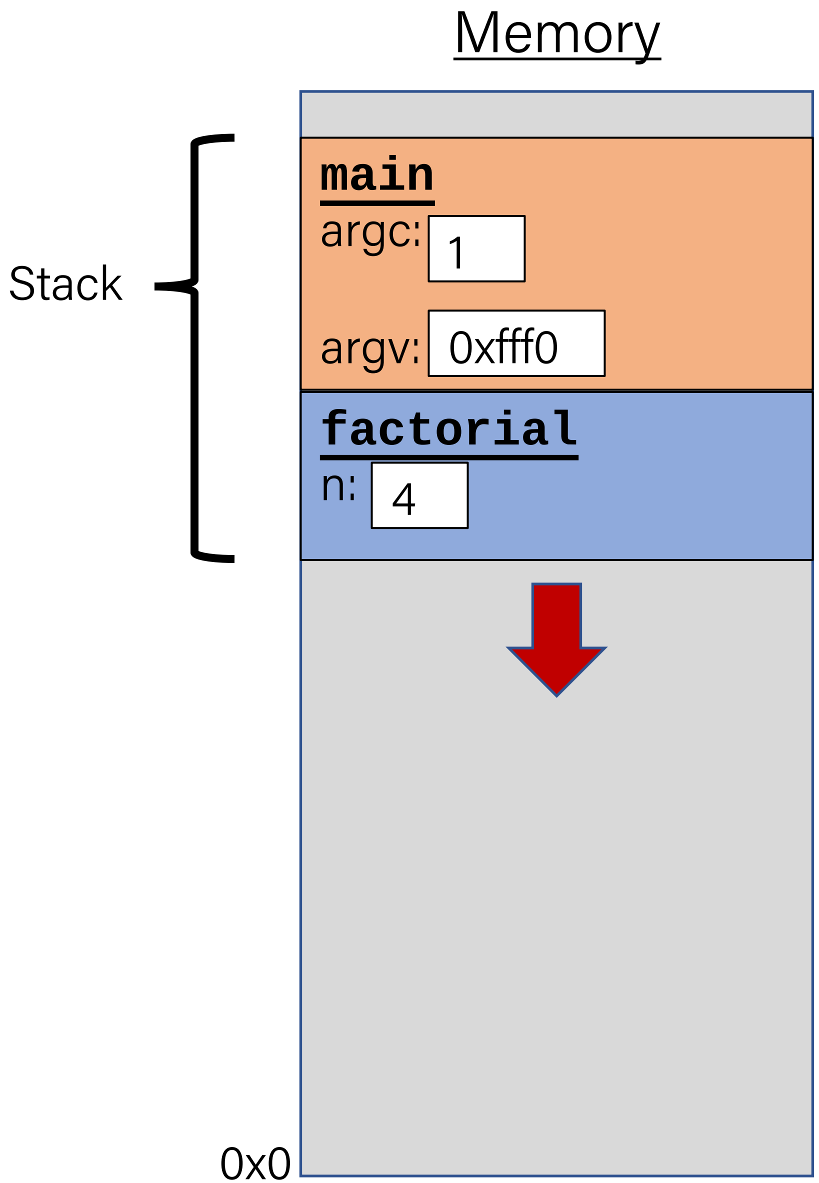
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

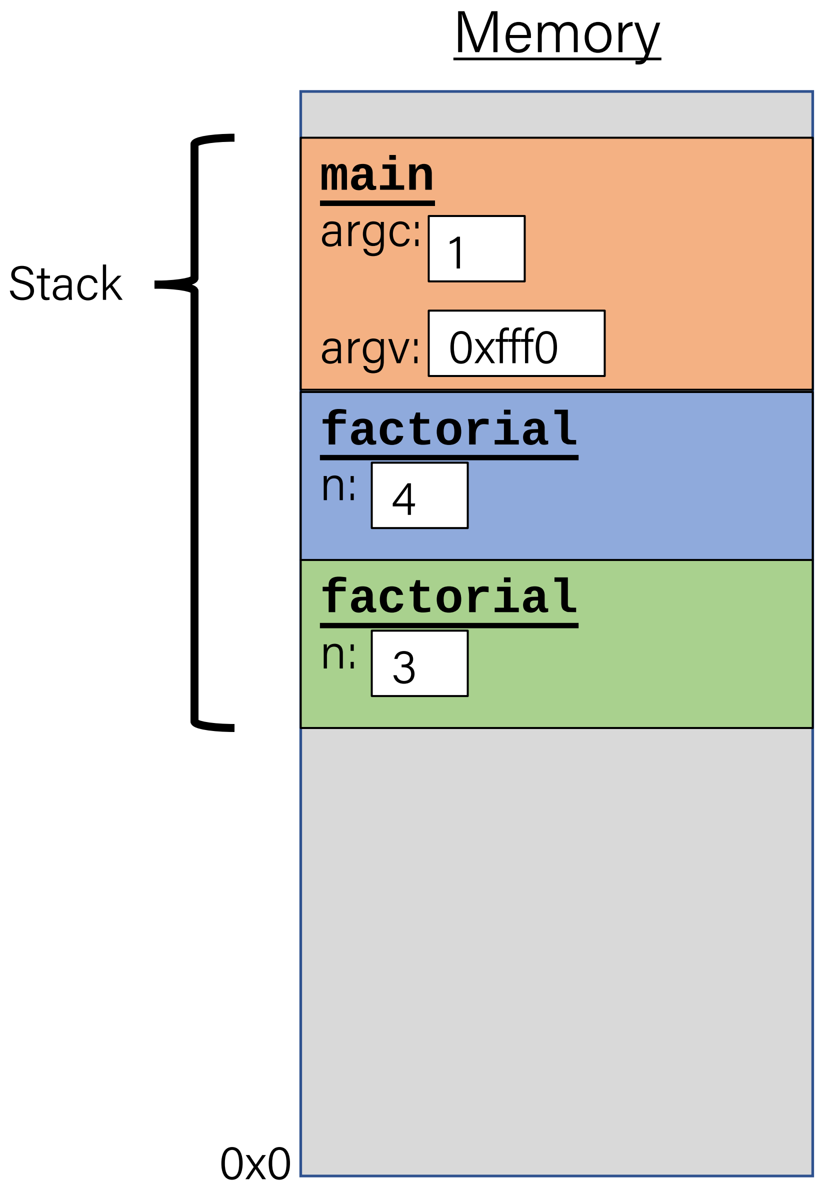
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

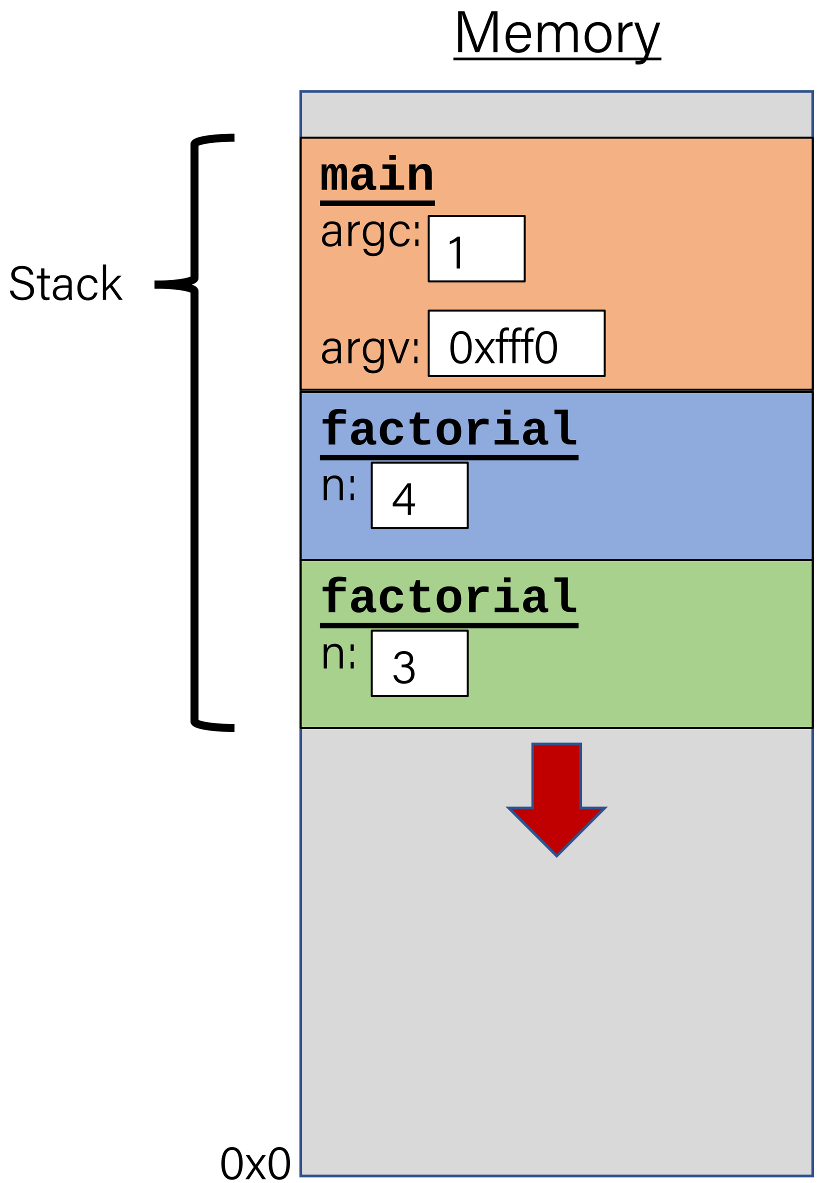
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

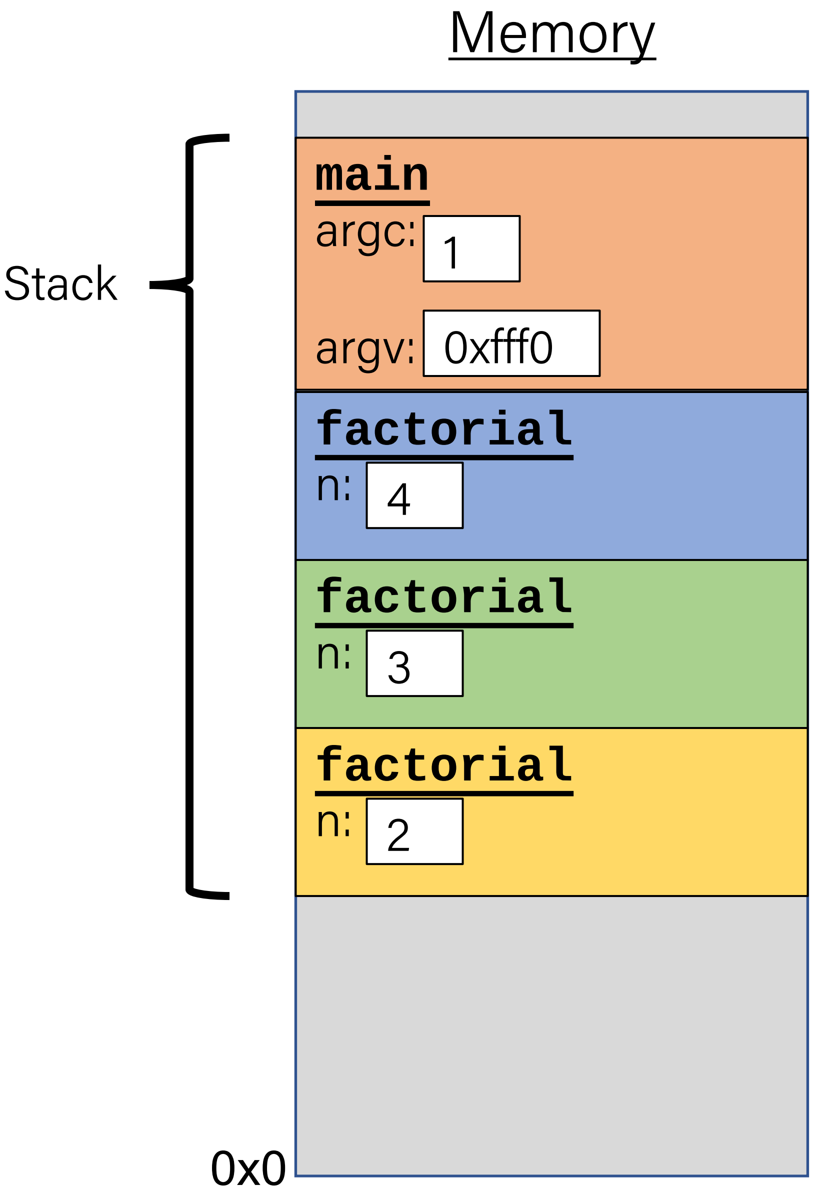
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

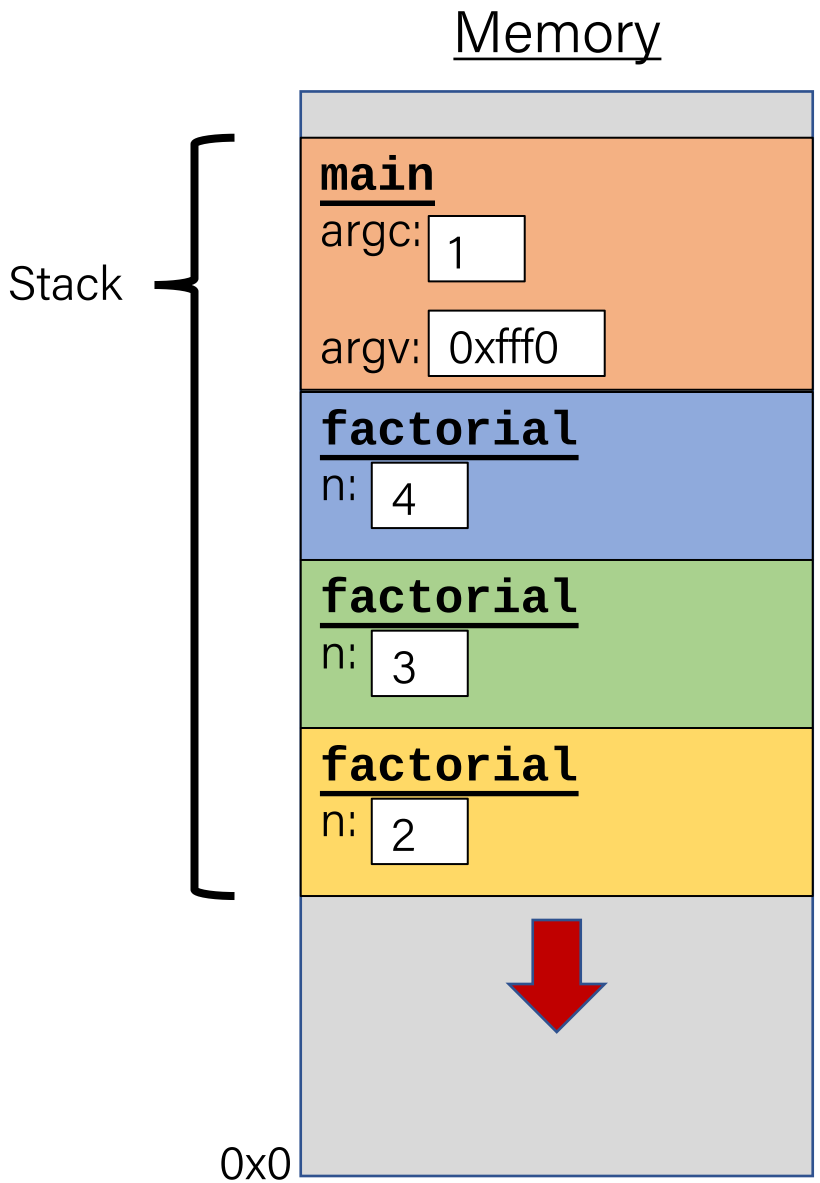
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

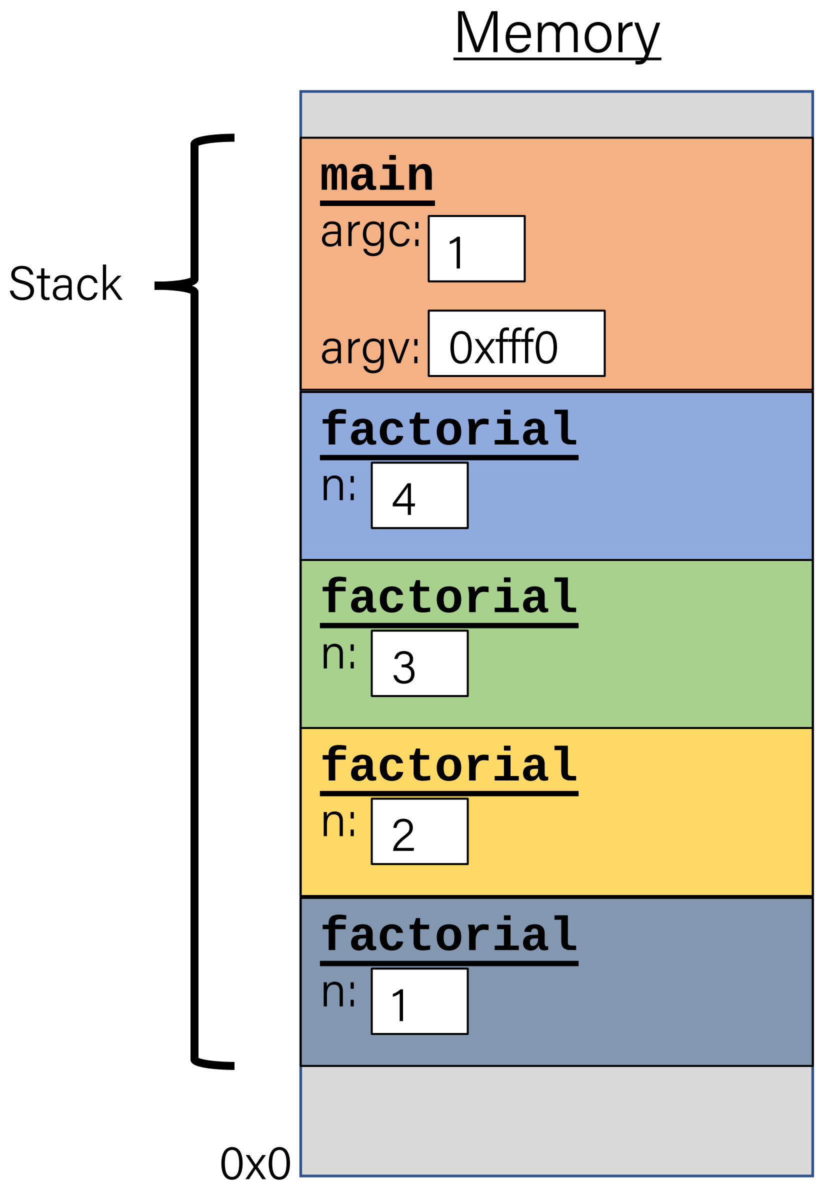
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

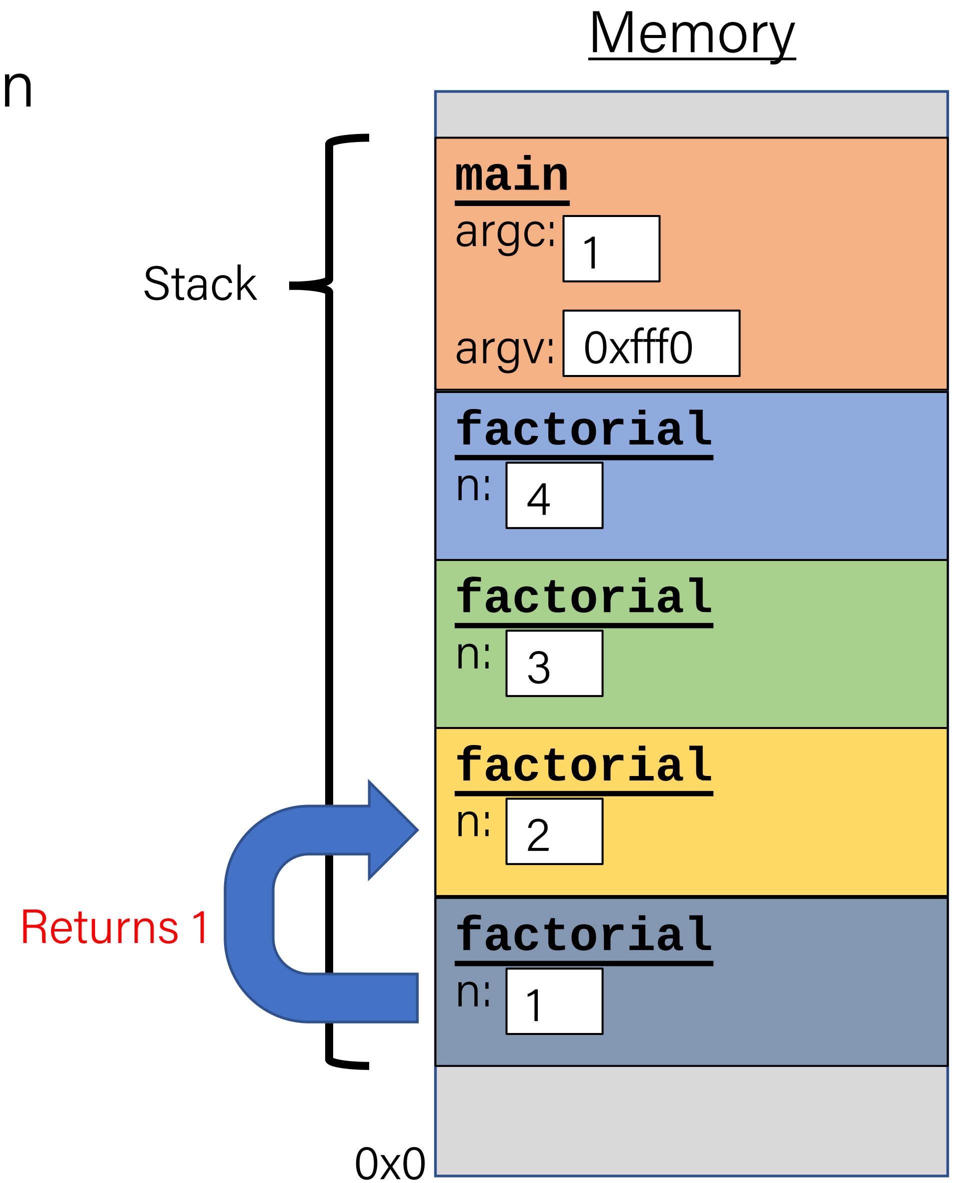
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

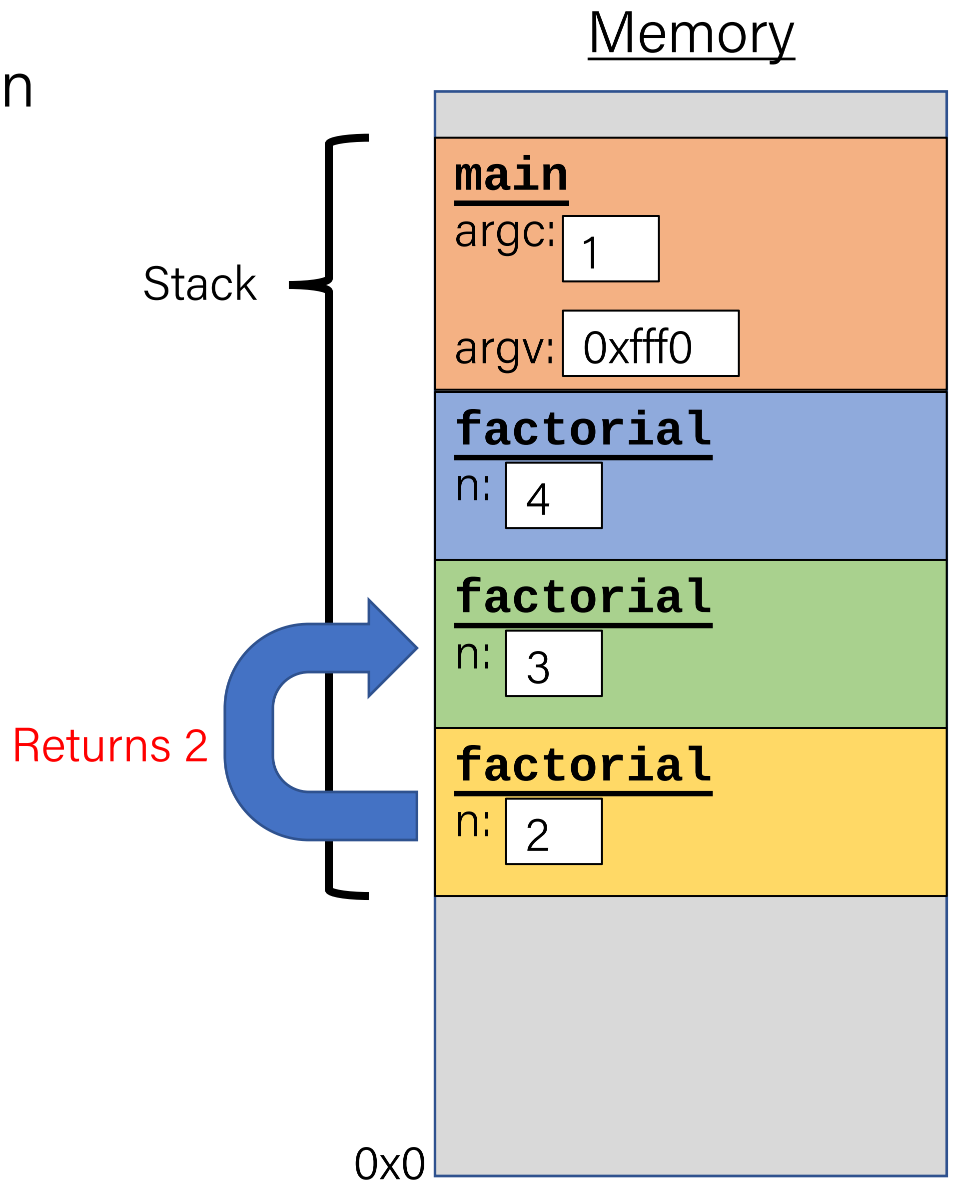
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

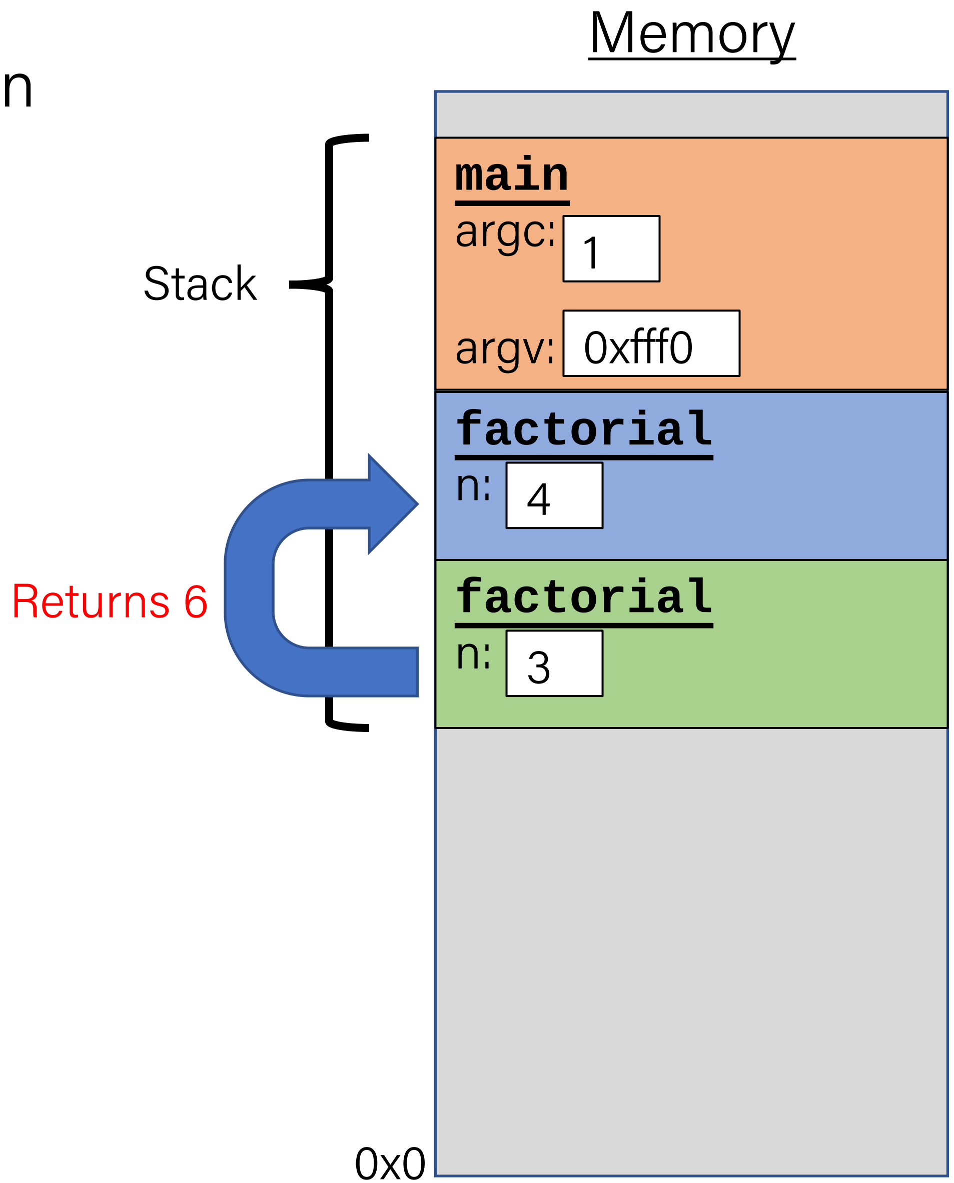
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

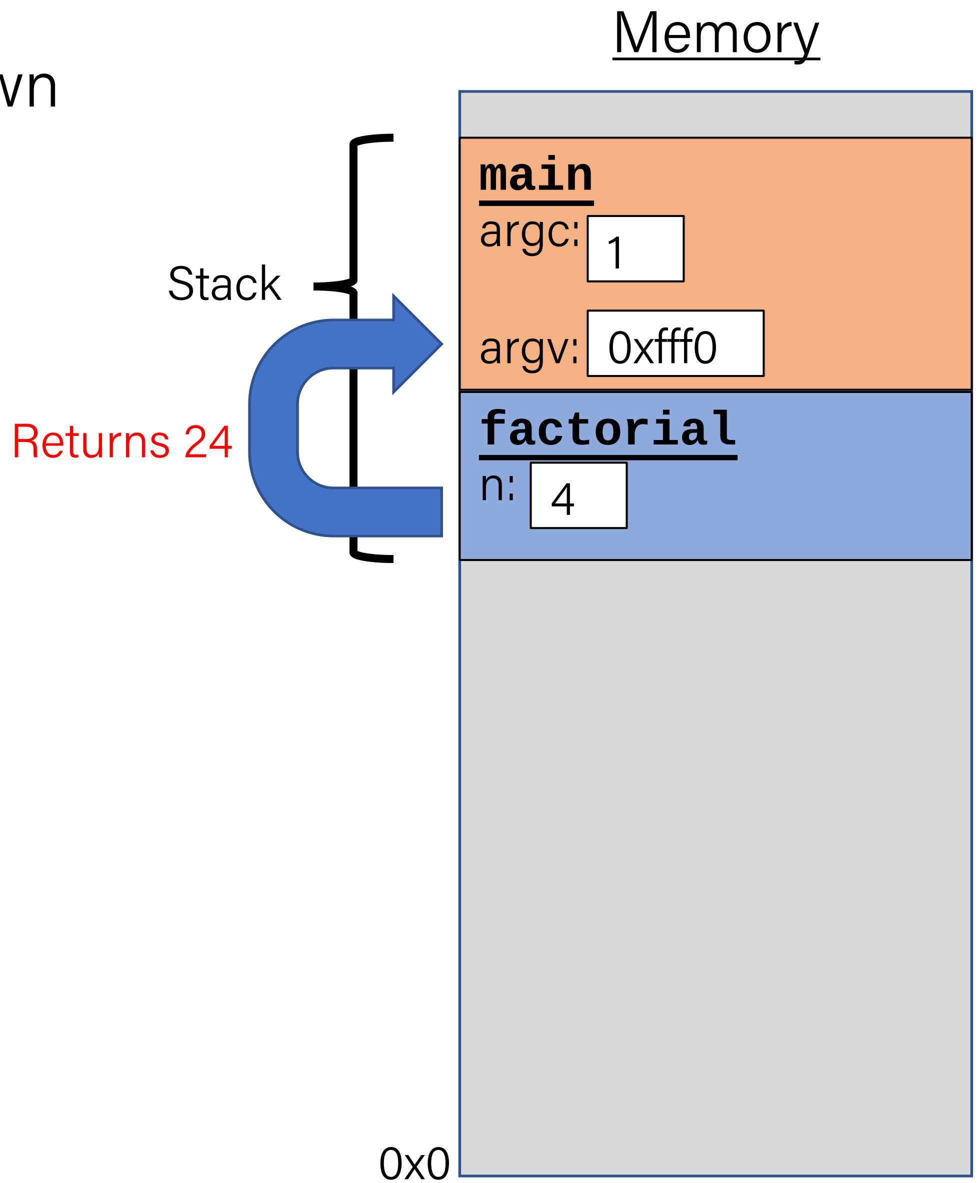
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

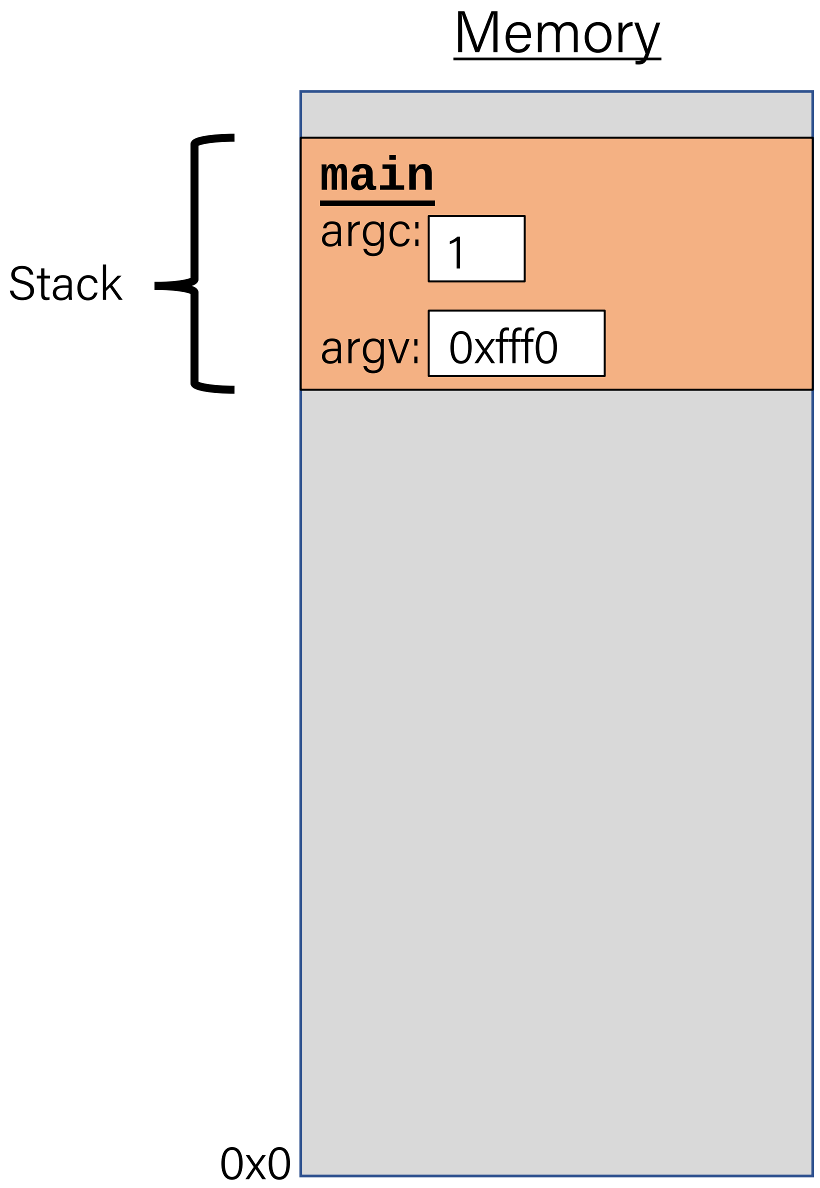
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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.

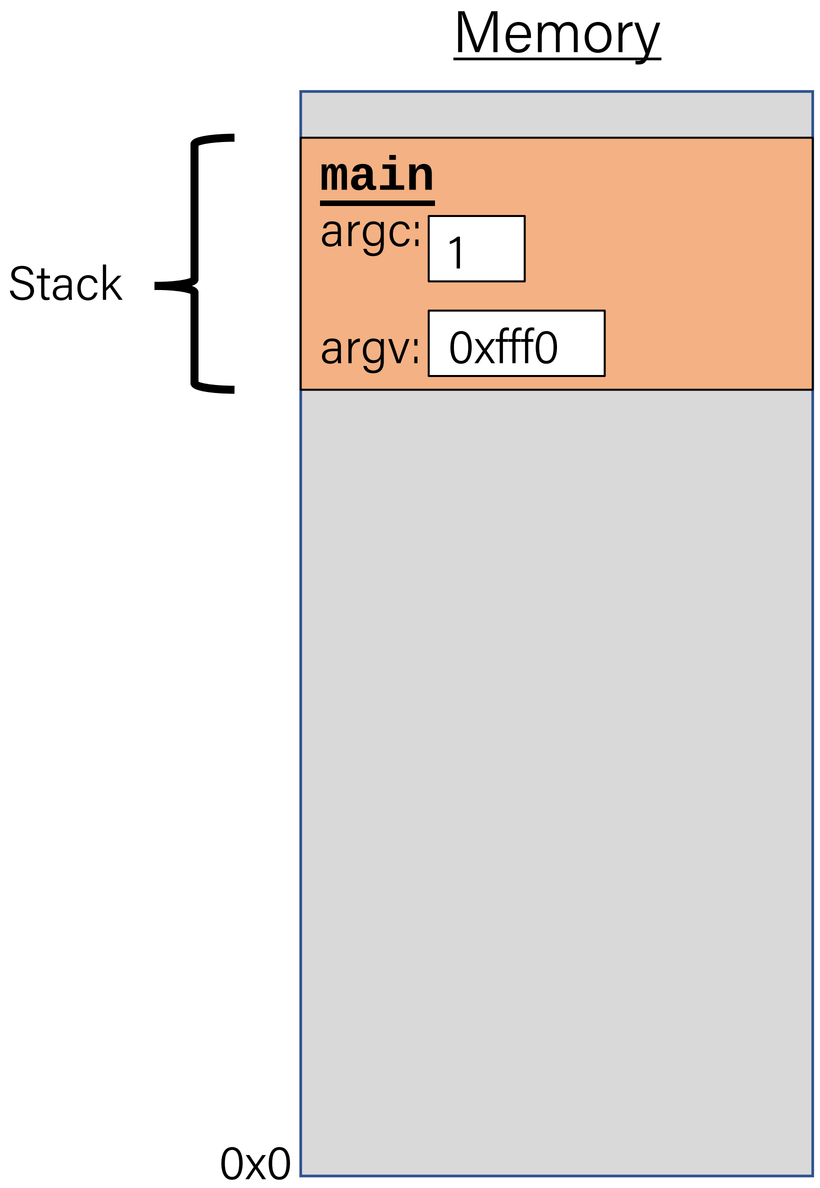
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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;
}
```

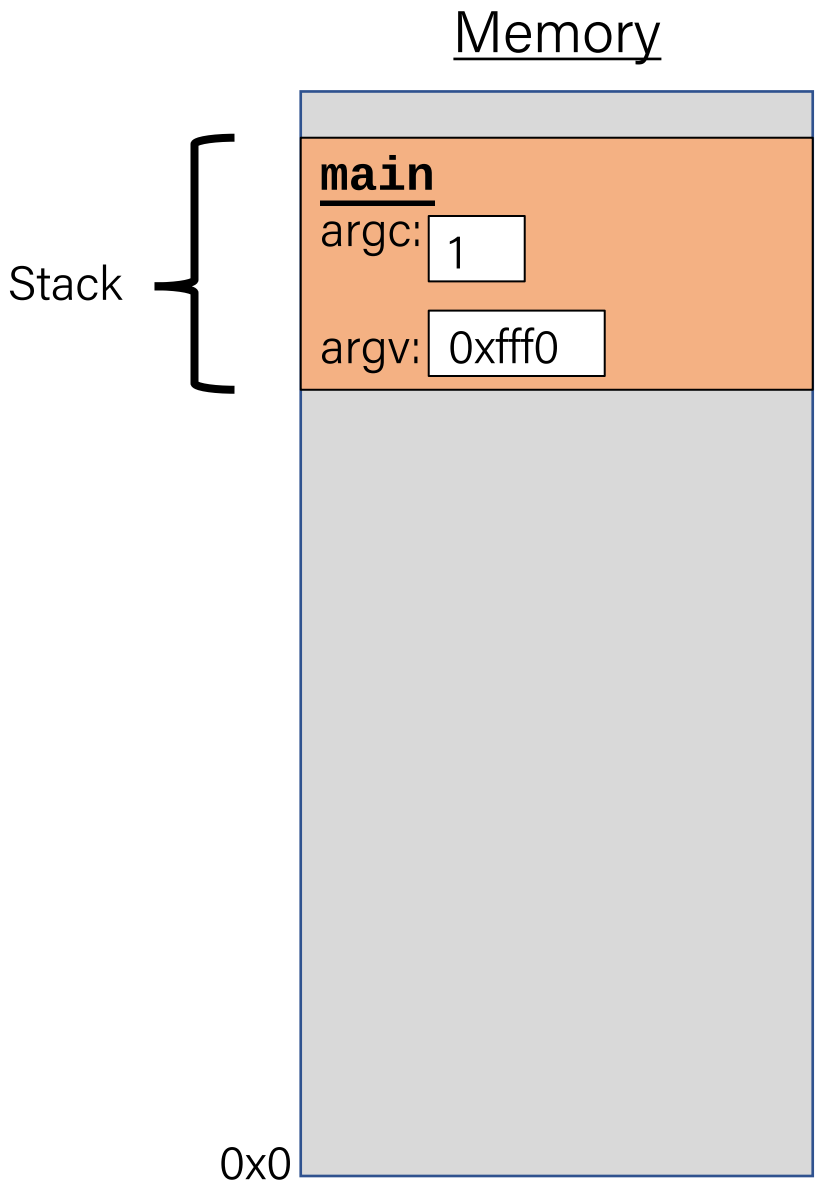


The Stack

- 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?

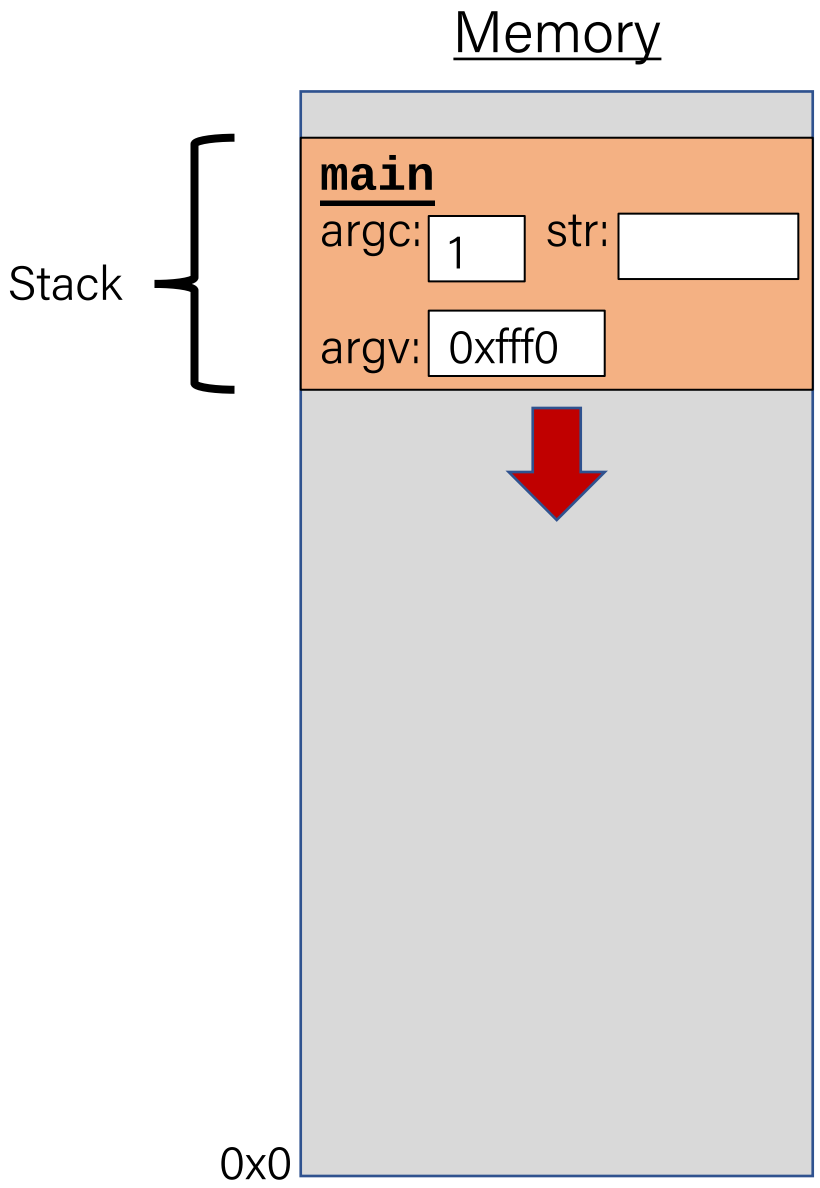
The Stack

```
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;
}
```



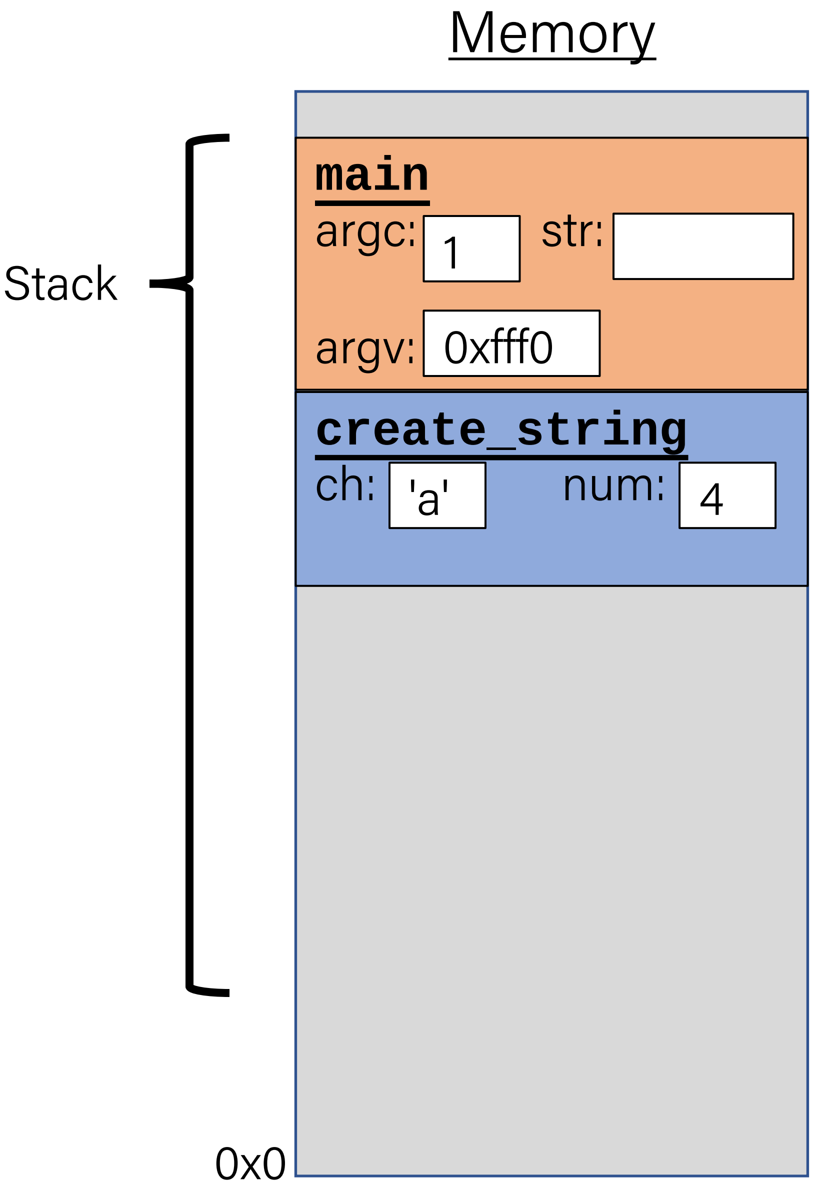
The Stack

```
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;
}
```



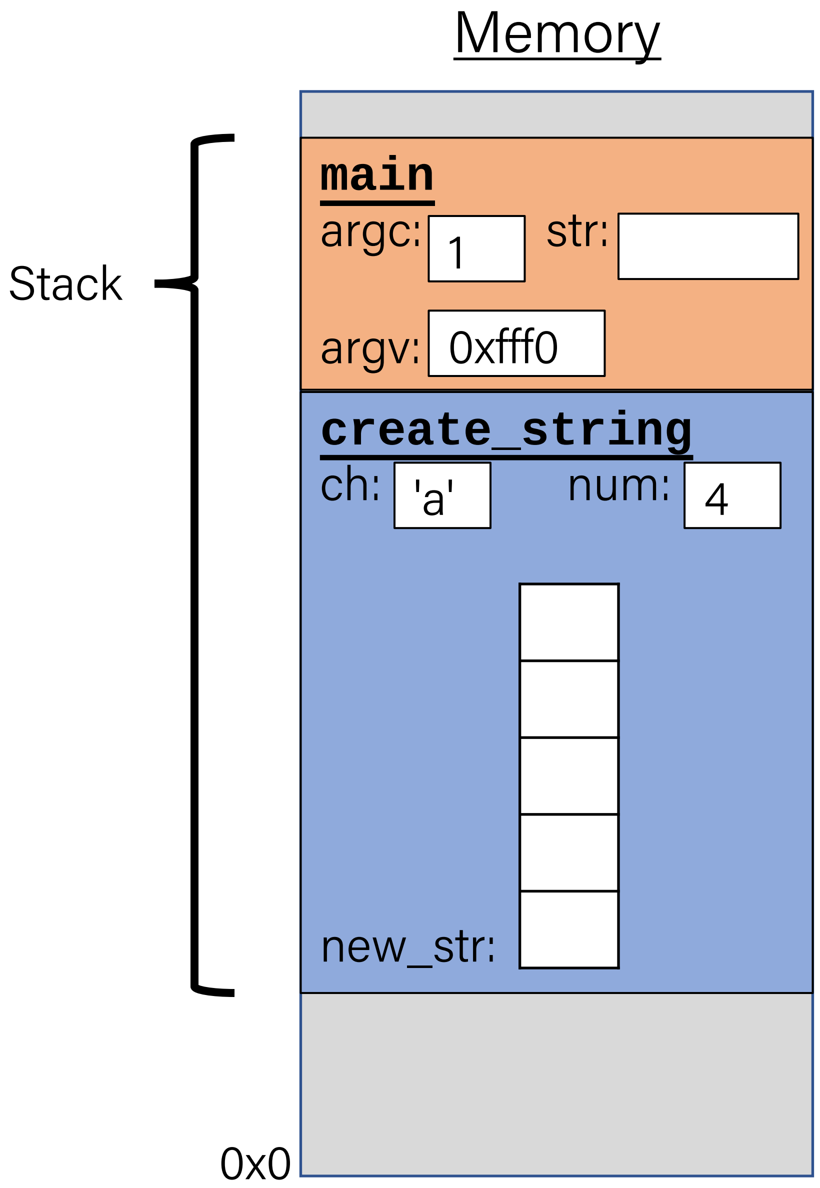
The Stack

```
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;
}
```



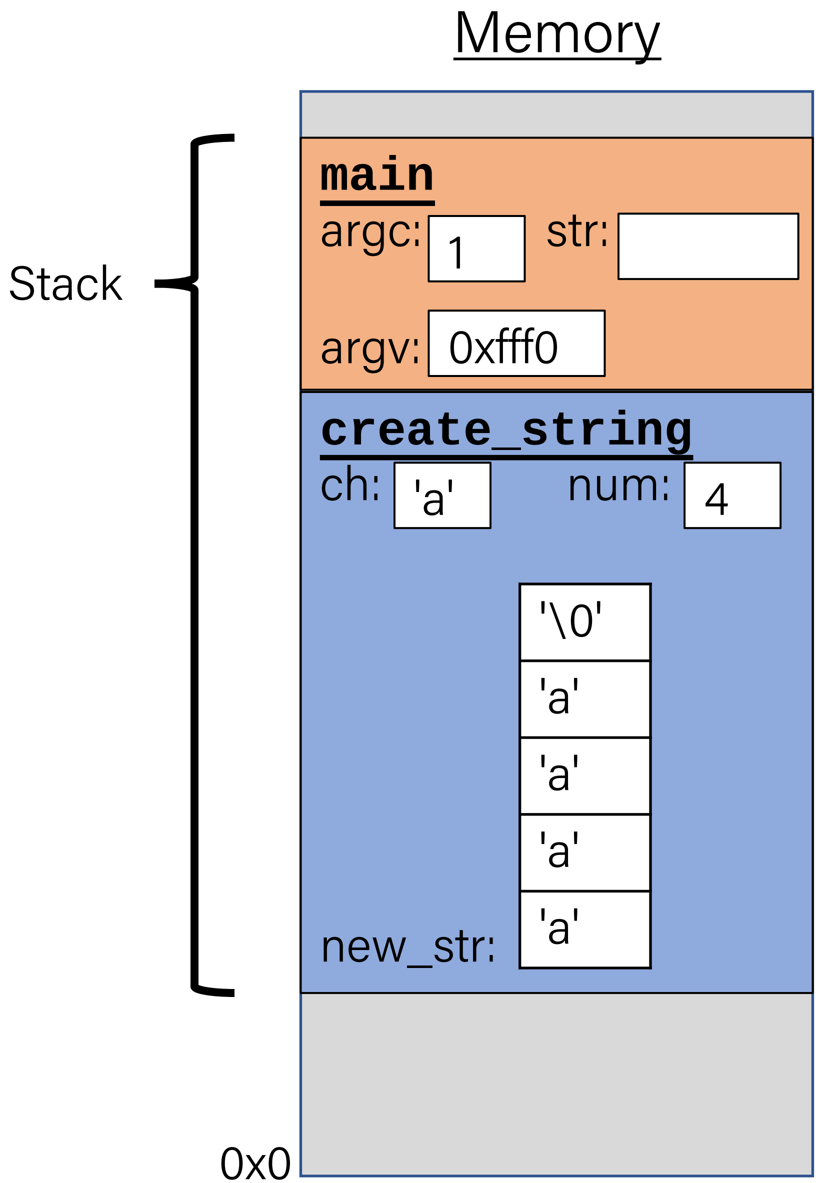
The Stack

```
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;
}
```



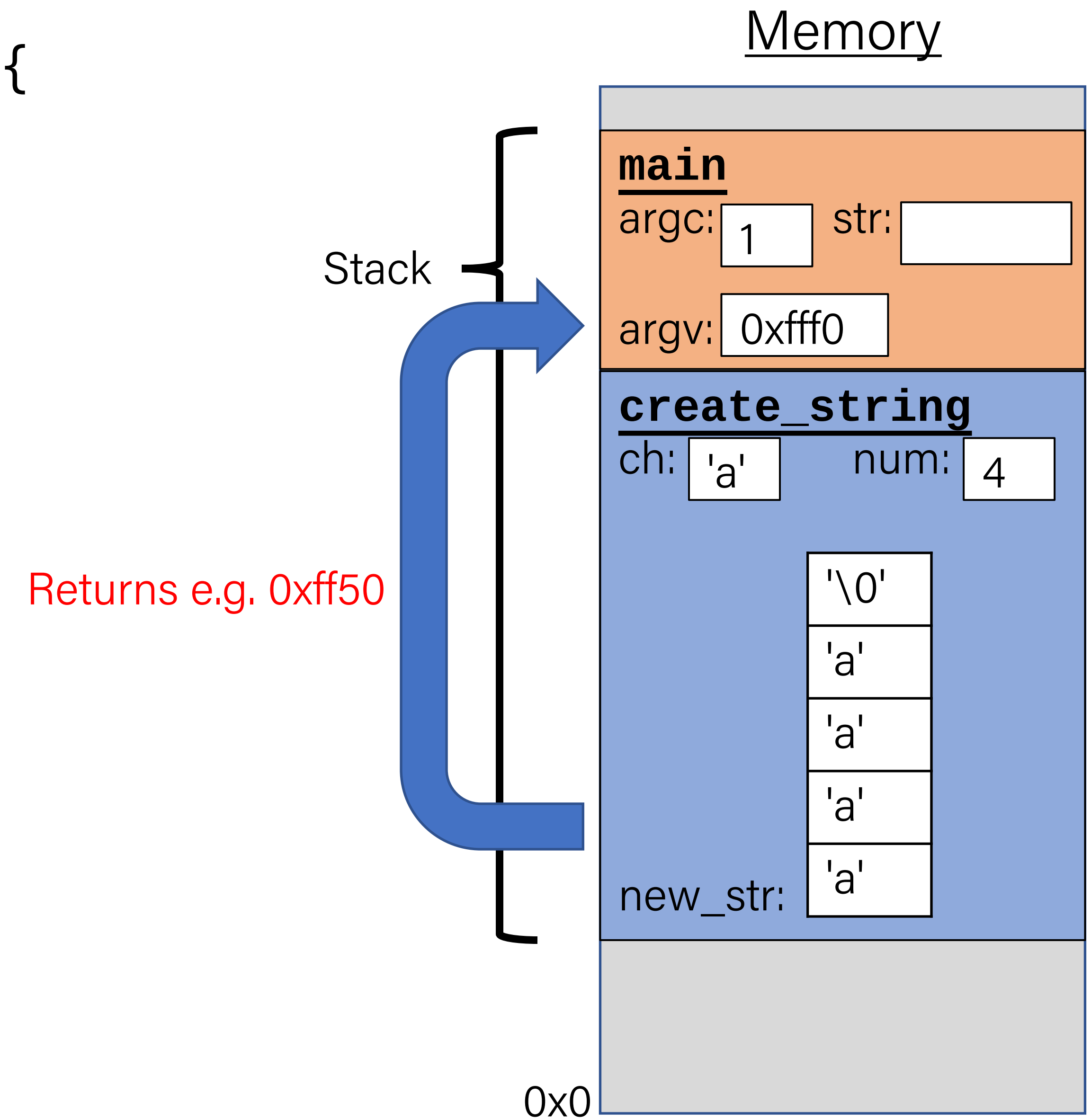
The Stack

```
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;
}
```



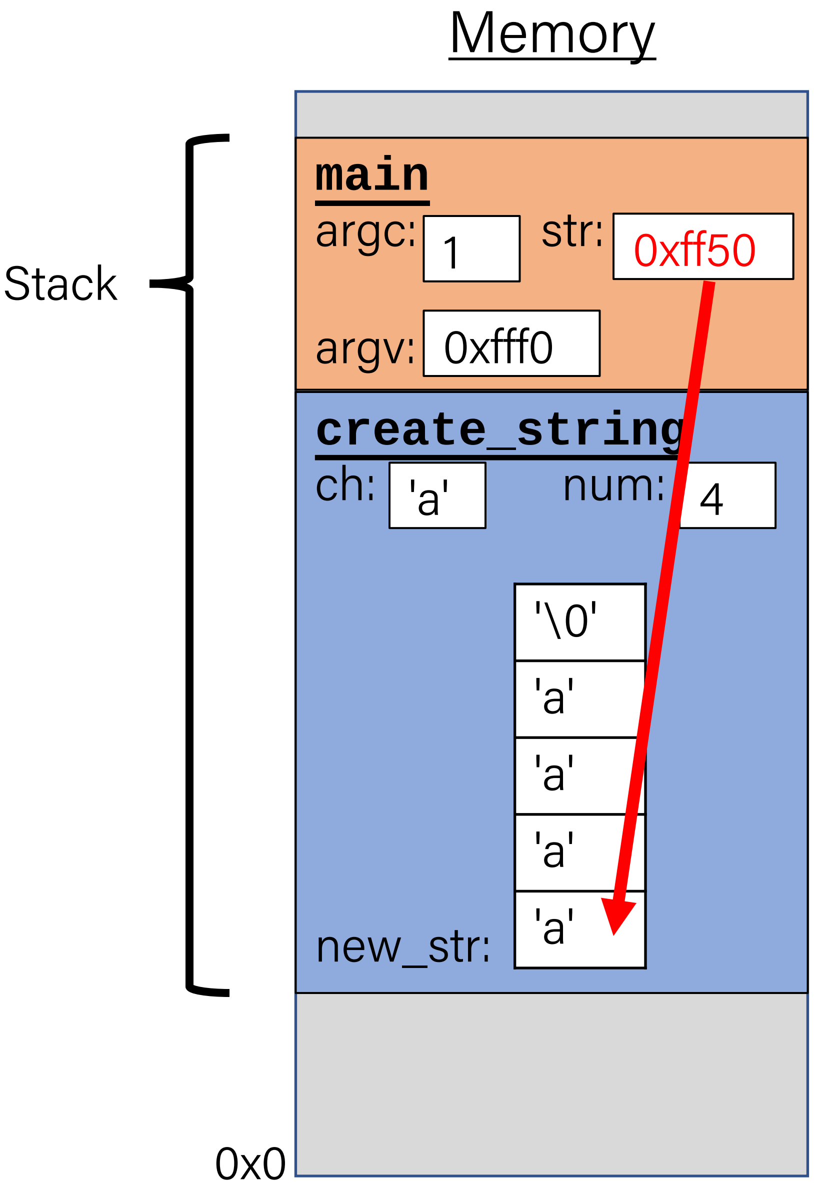
The Stack

```
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;
}
```



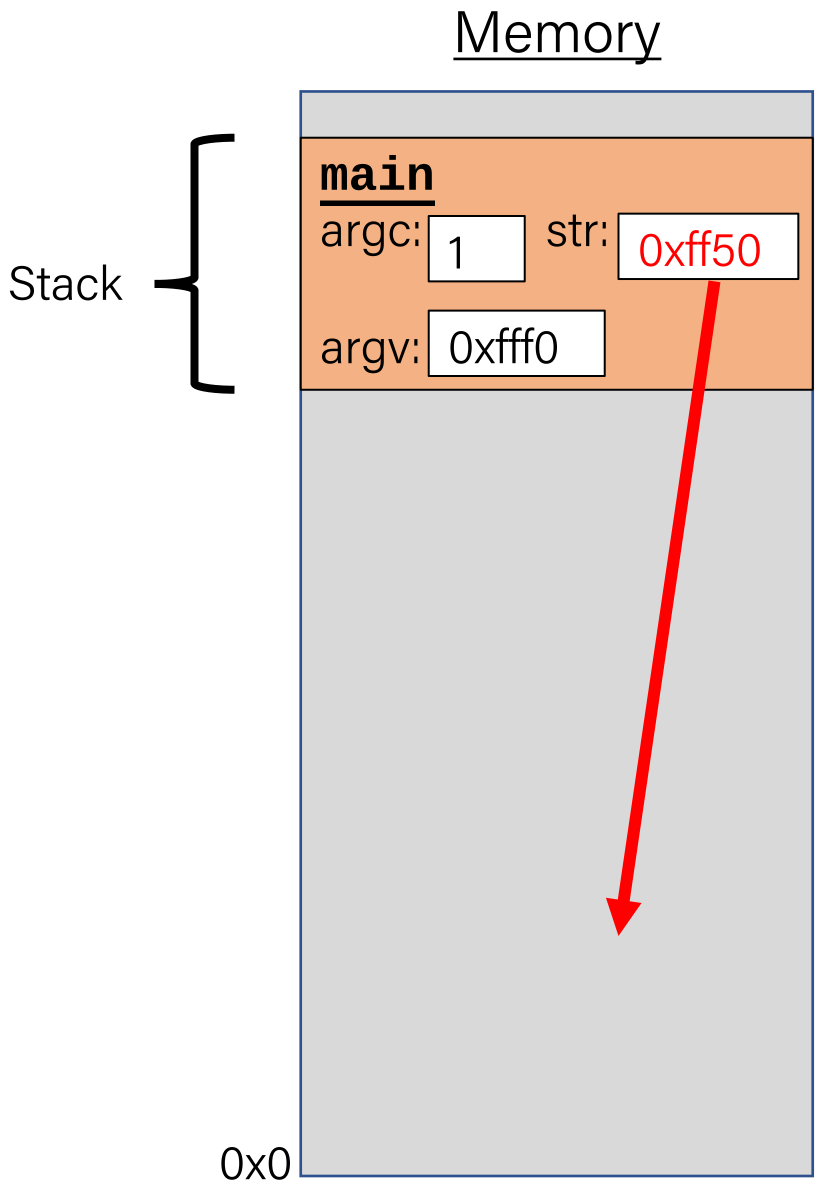
The Stack

```
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

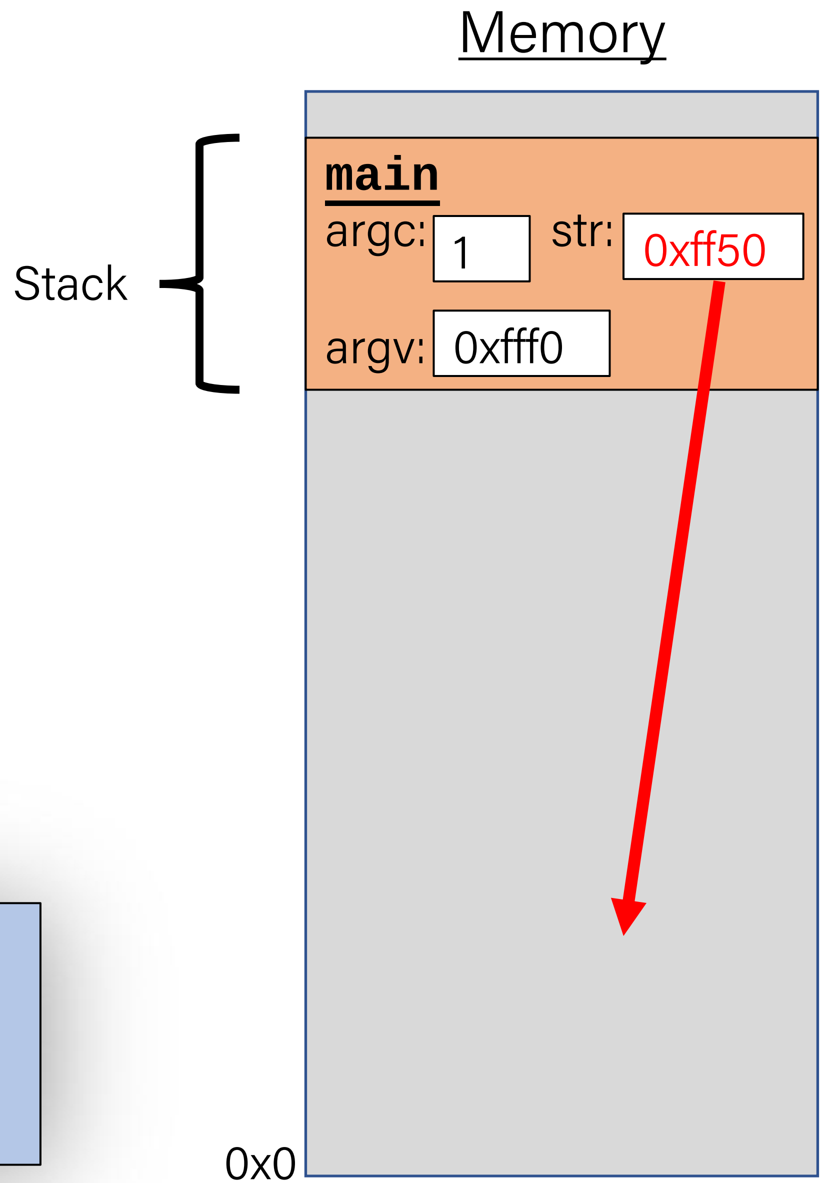
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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



The Stack

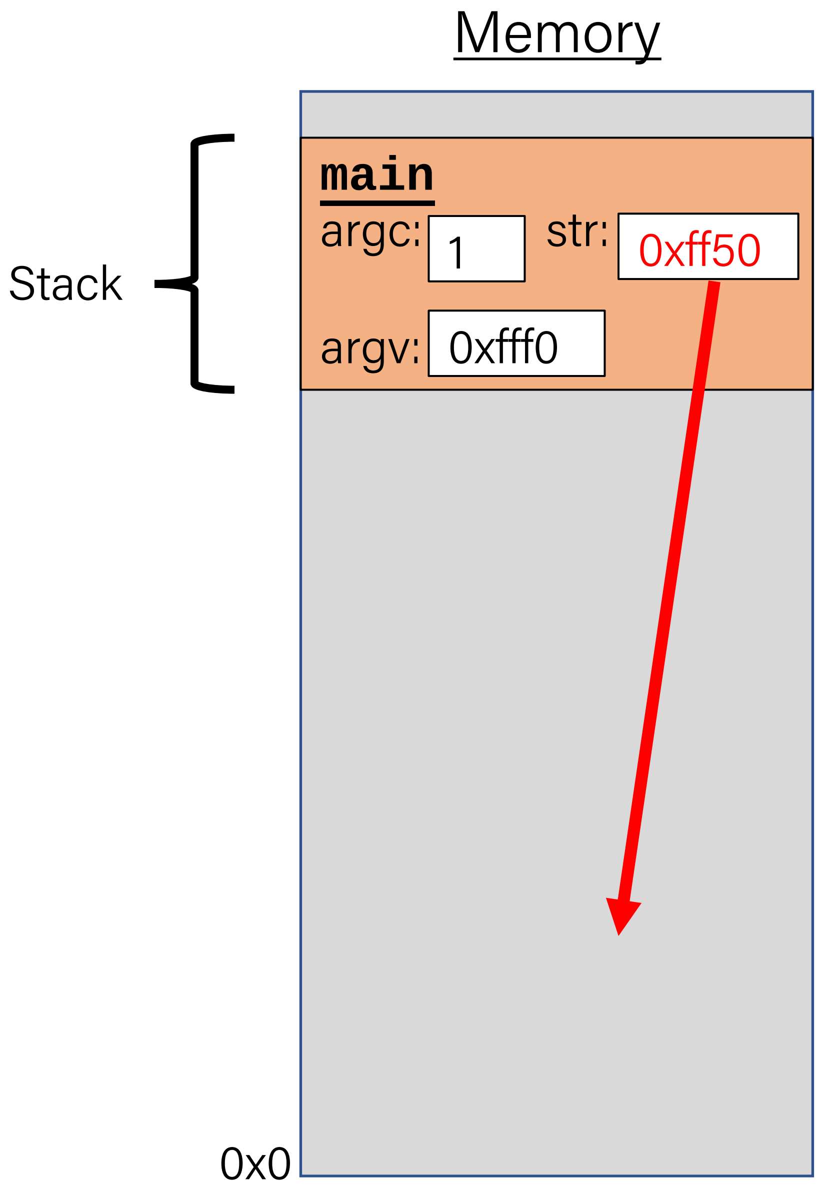
```
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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.

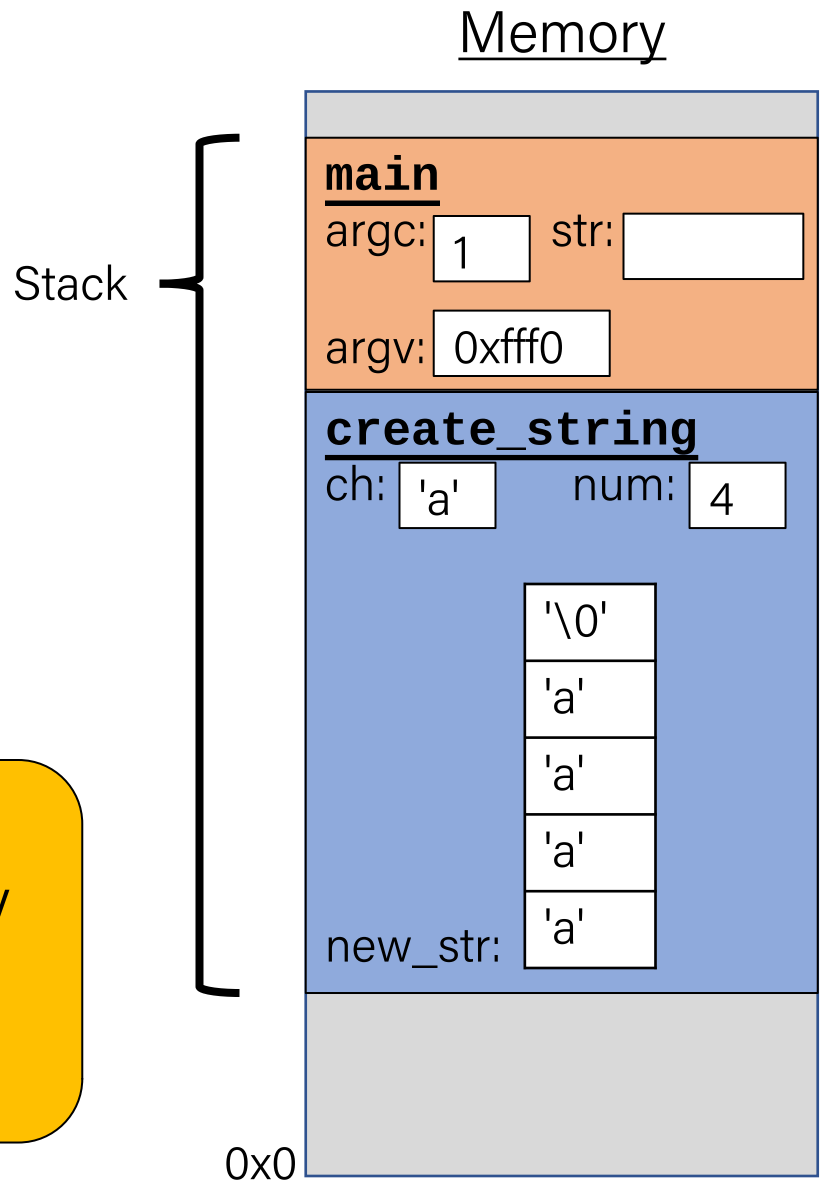
Lecture Plan

- Arrays in Memory
- Arrays of Pointers
- 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;
}
```

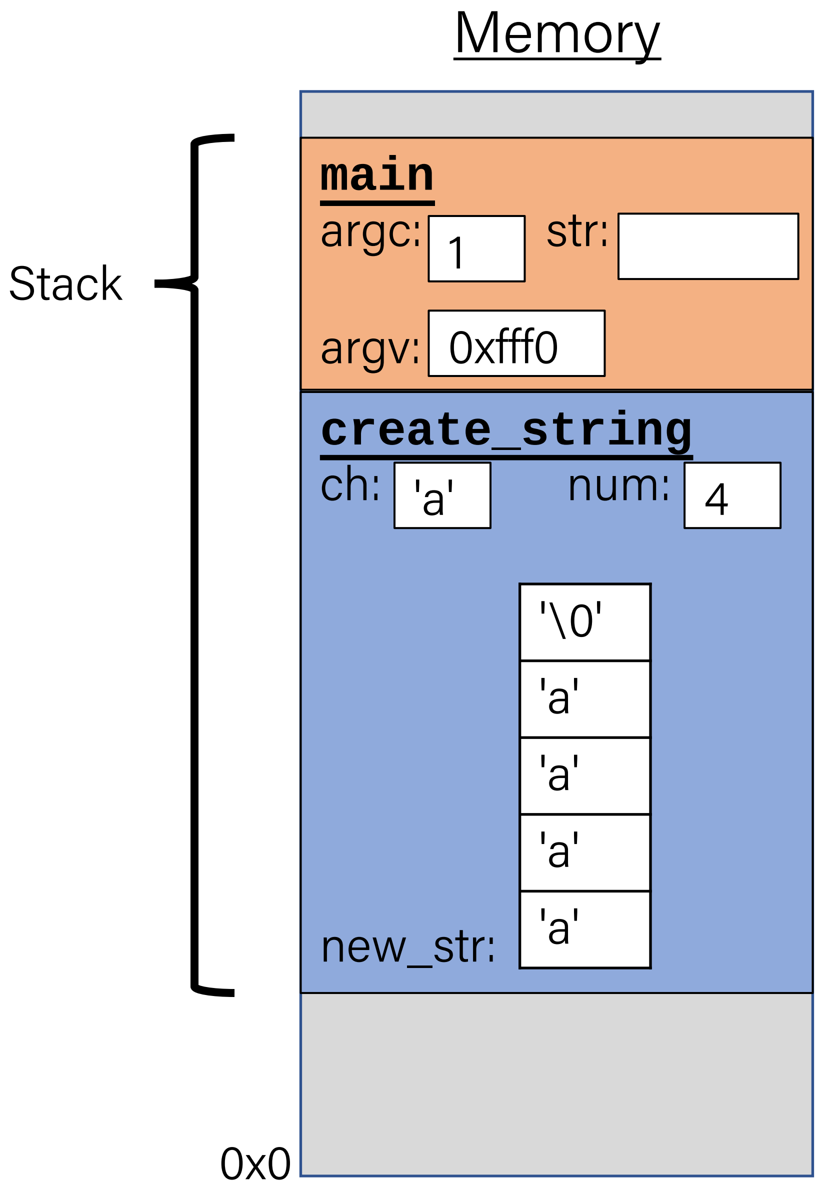
Us: hey C, is there a way to make this variable in memory that isn't automatically cleaned up?



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;
}
```

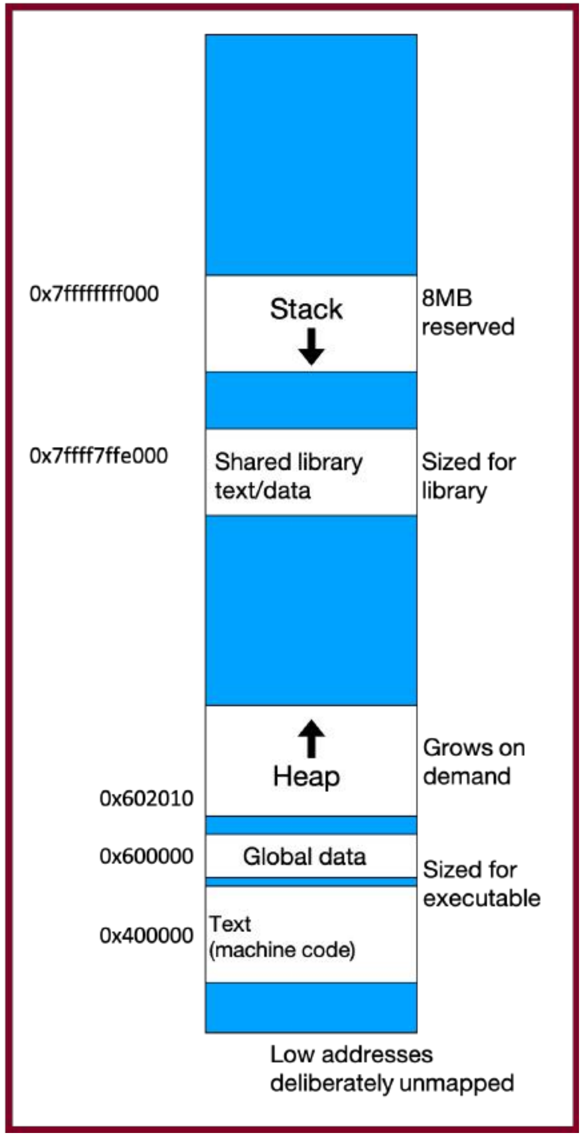
C: sure, but since I don't know when to clean it up anymore, it's your responsibility…



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

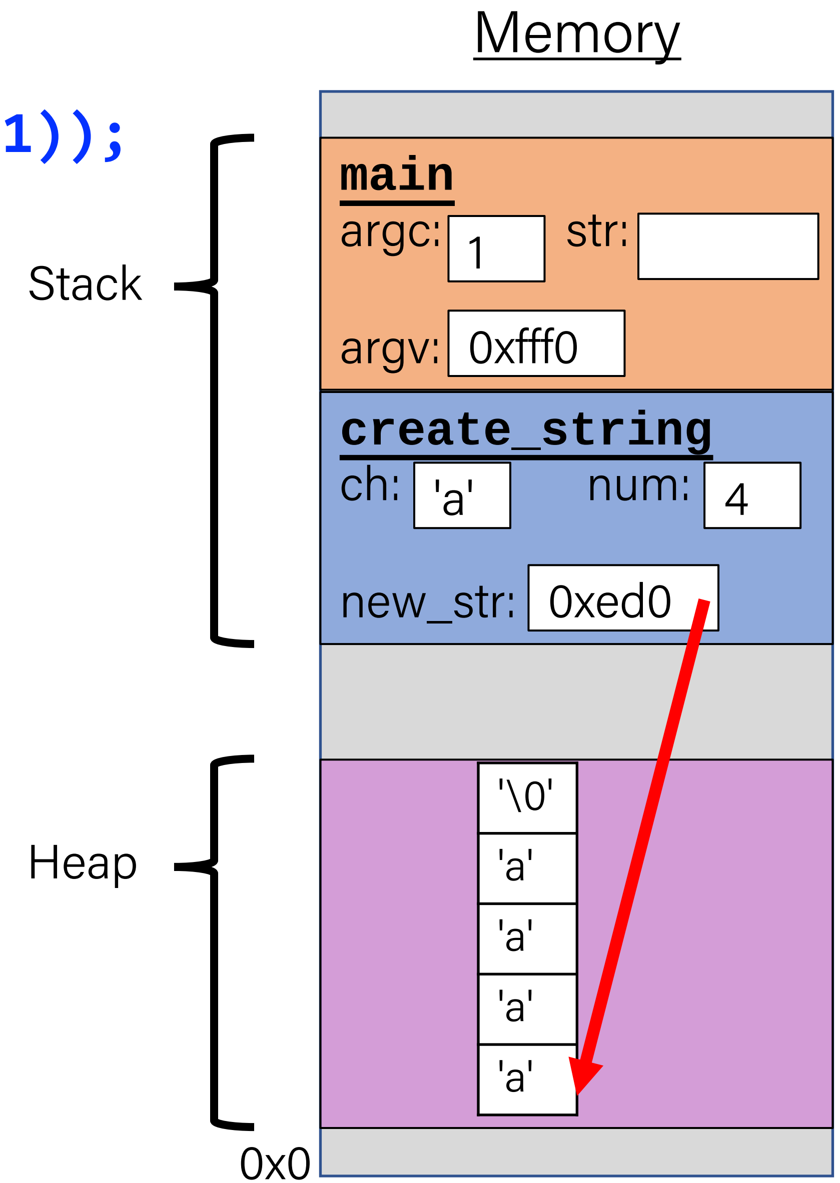
```c
void *malloc(size_t size);
```

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.

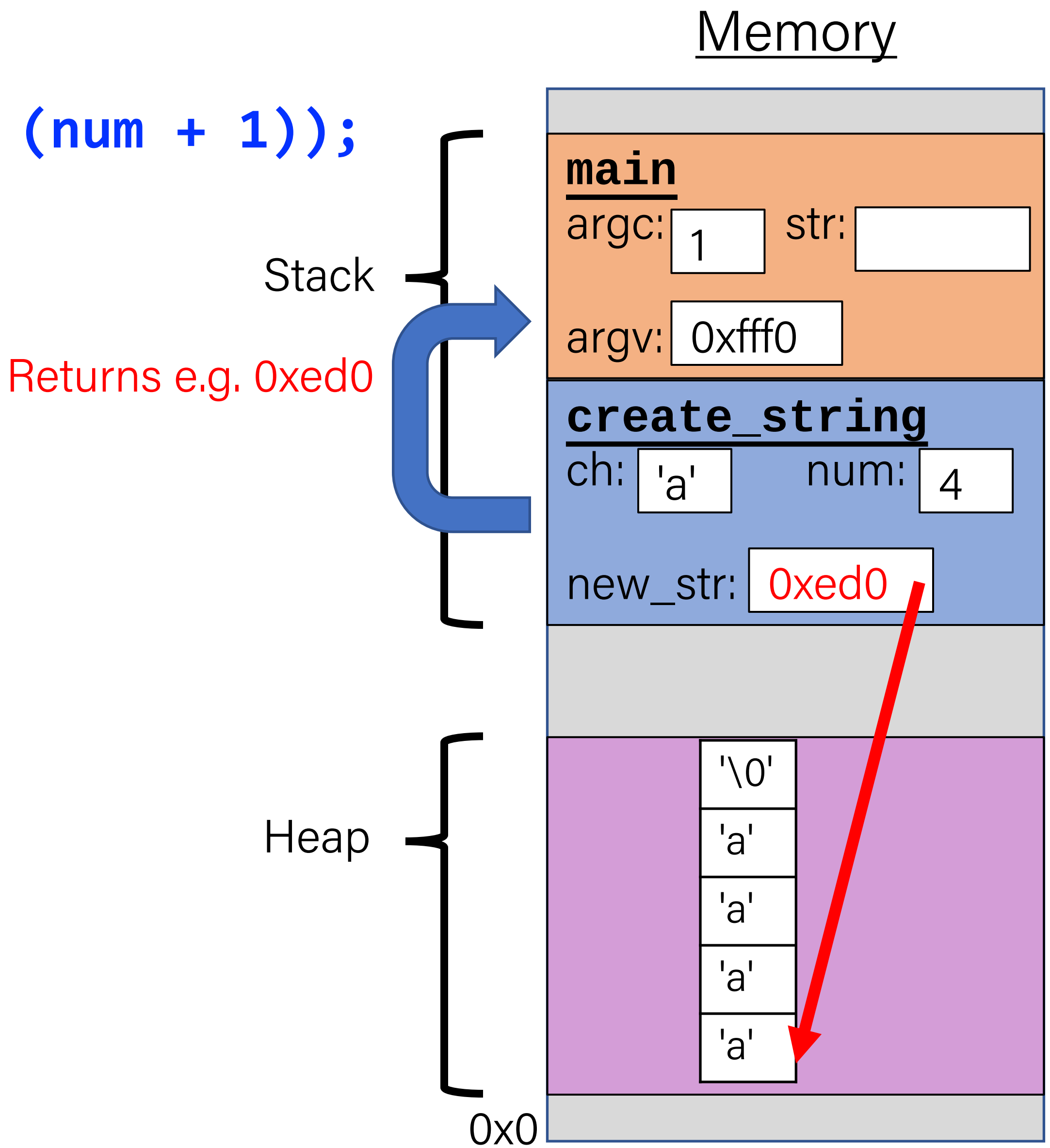
The Heap

```
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;
}
```



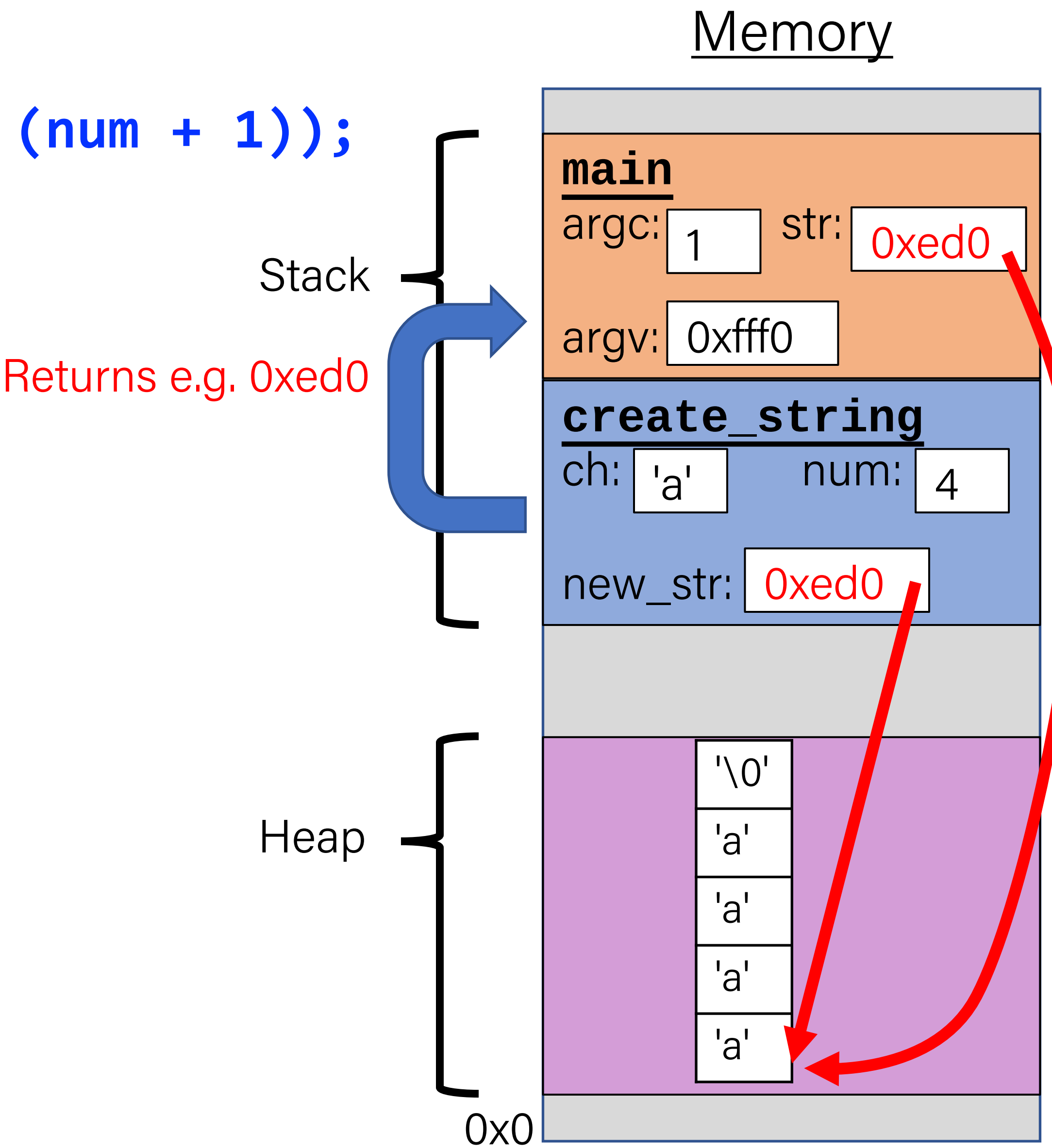
The Heap

```
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;
}
```



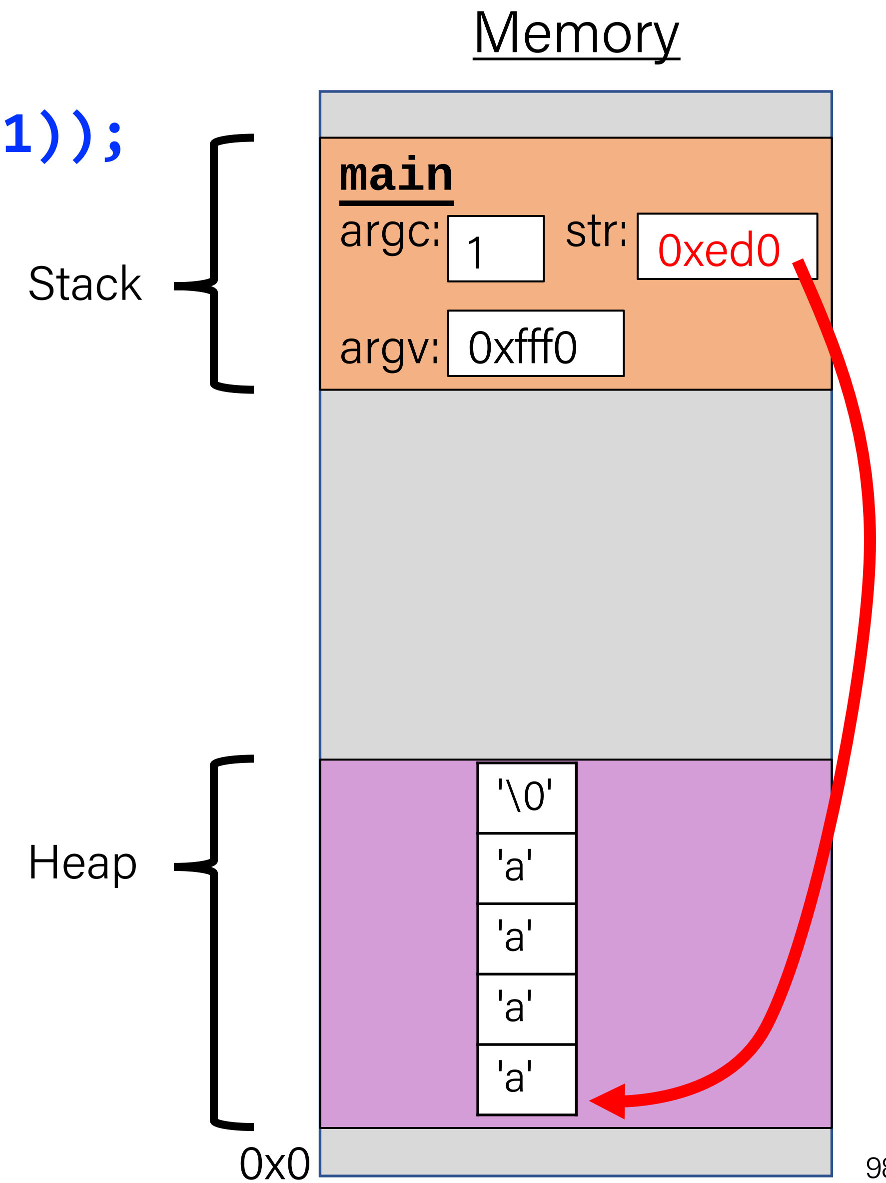
The Heap

```
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;
}
```



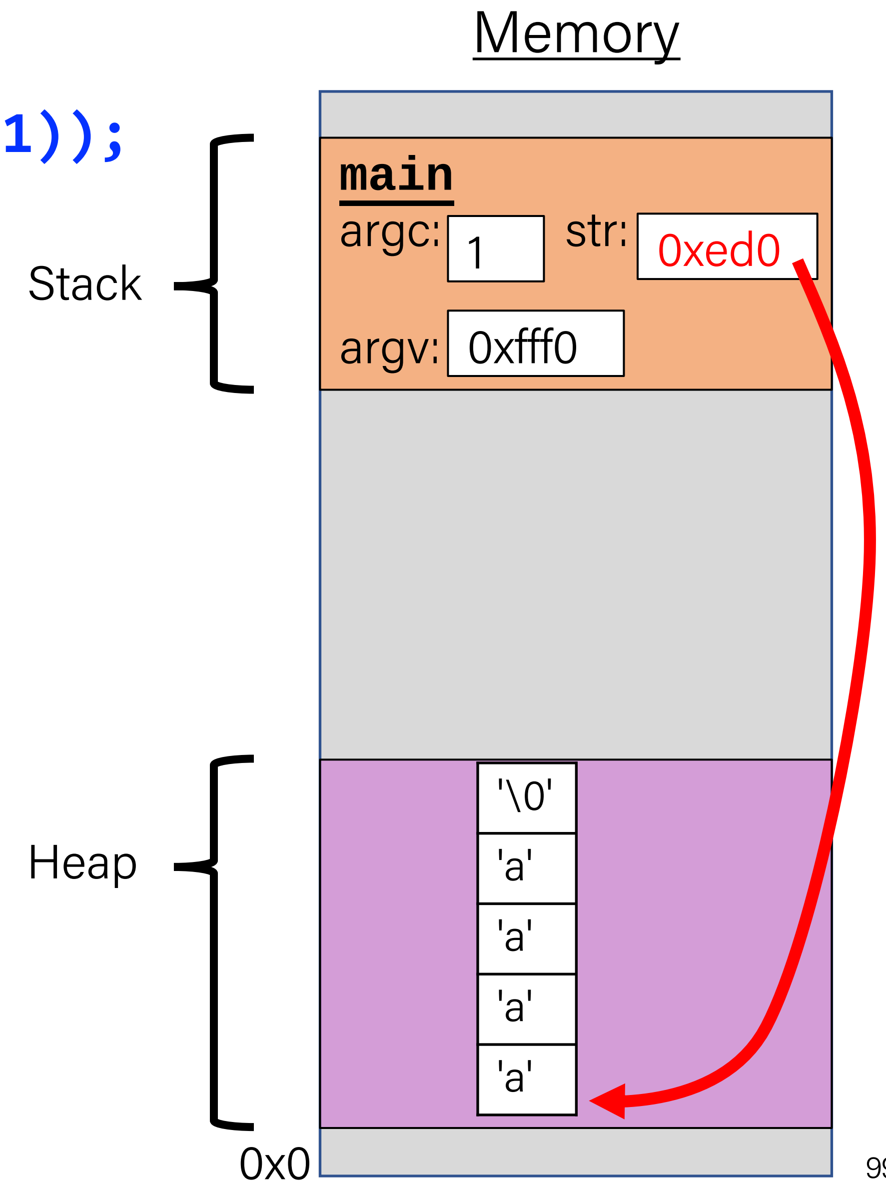
The Heap

```
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;
}
```



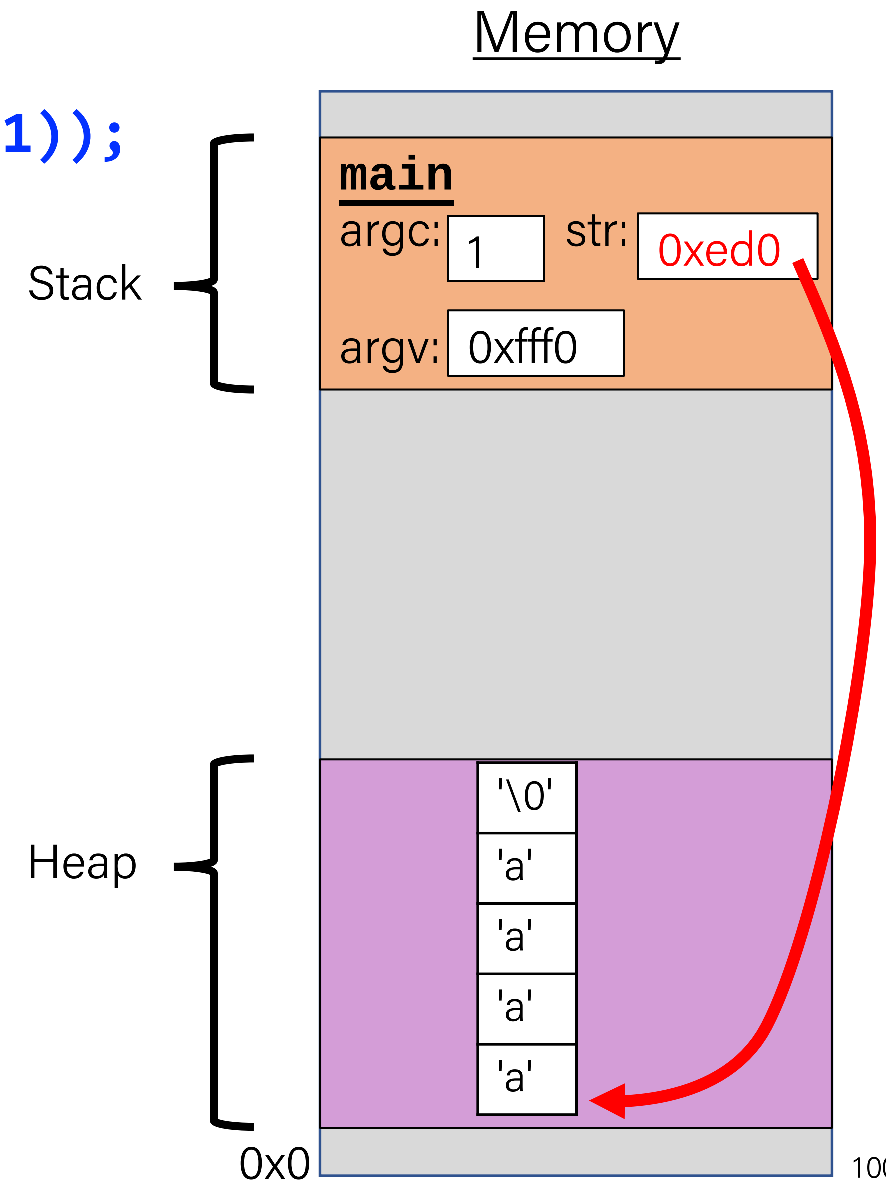
The Heap

```
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

slido

Please download and install the Slido app on all computers you use

How should we declare arr?

ⓘ Start presenting to display the poll results on this slide.

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**.

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

- 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';
```

Implementing strdup

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

```
char *myStrdup(char *str) {
    char *heapStr = malloc(strlen(str) + 1);
    assert(heapStr != NULL);
    strcpy(heapStr, str);
    return heapStr;
}
```

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!

More on Valgrind in Lab 4!

free Practice

Freeing Memory

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

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
}
printf("%s\n", str);
```

Freeing Memory

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

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
     int *num = malloc(sizeof(int));
     assert(num != NULL);
     *num = i;
     printf("%s %d\n", ptr, *num);
     free(num);
}
printf("%s\n", str);
free(str);
```

Recap

- Arrays in Memory
- Arrays of Pointers
- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Next time: *`realloc`, Memory bugs*