CS107, Lecture 7
Stack and Heap

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

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
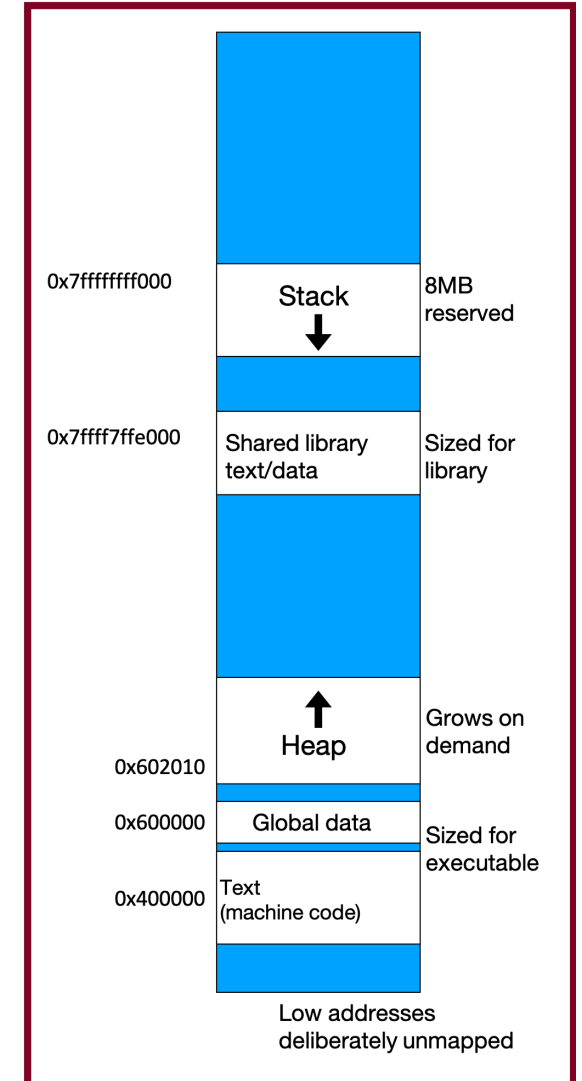- `realloc`
- **Practice:** Pig Latin Part 2

```
cp -r /afs/ir/class/cs107/lecture-code/lect7 .
```

Lecture Plan

- **The Stack**
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- `realloc`
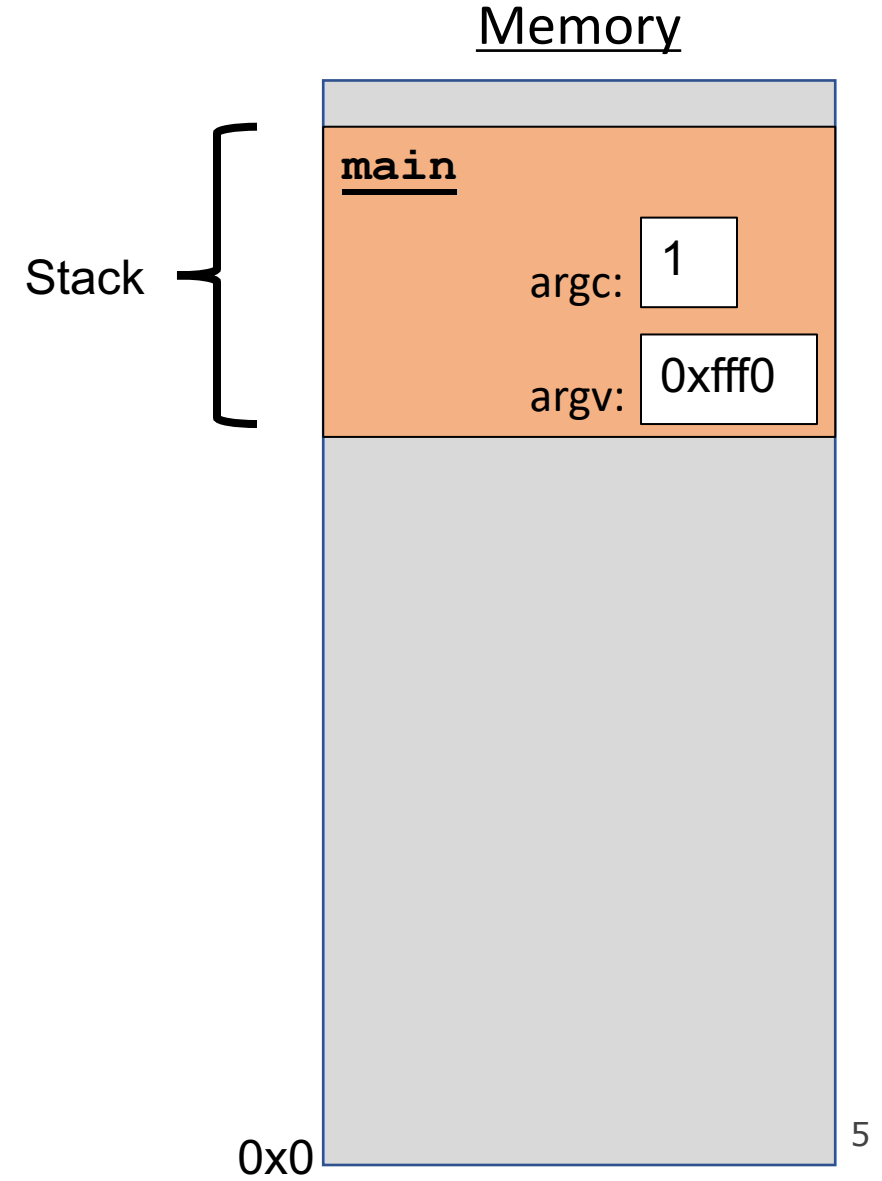- **Practice:** Pig Latin Part 2

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.

0x7fffffff000	Stack	8MB reserved
0x7ffff7ffe000	Shared library text/data	Sized for library
0x602010	Heap	Grows on demand
0x600000	Global data	Sized for executable
0x400000	Text (machine code)	
	Low addresses deliberately unmapped	

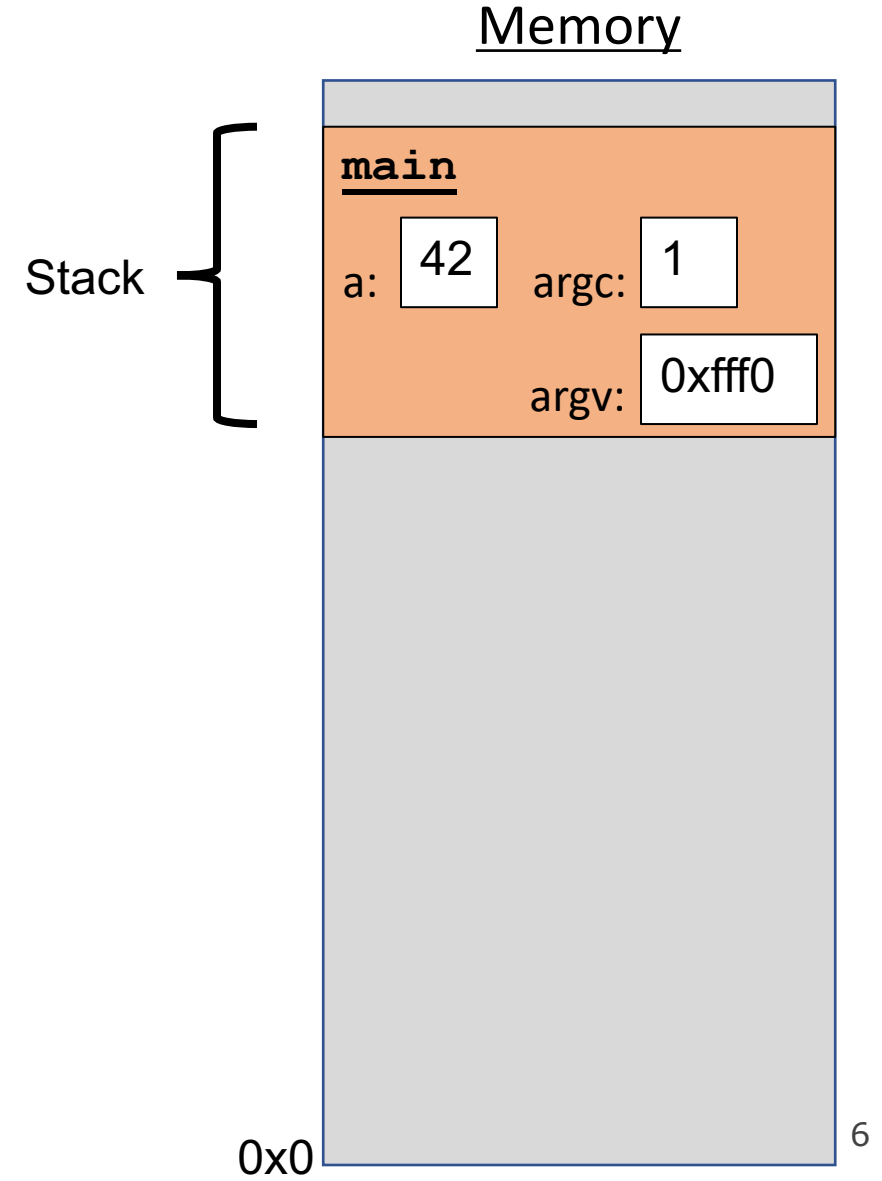# The Stack

```
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

Stack

main

argc: 1

argv: 0xfff0

0x0

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;
}
```
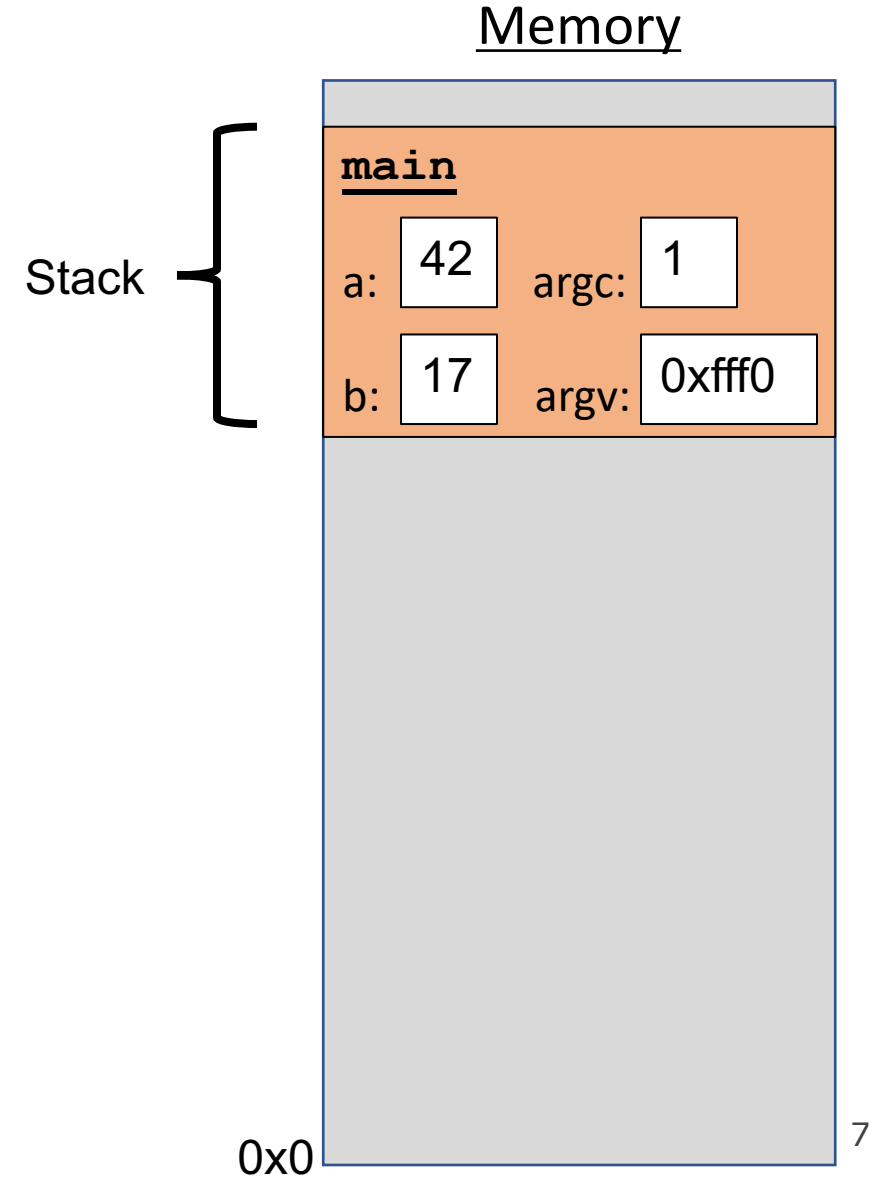
Memory

Stack

main

a: 42   argc: 1

argv: 0xfff0
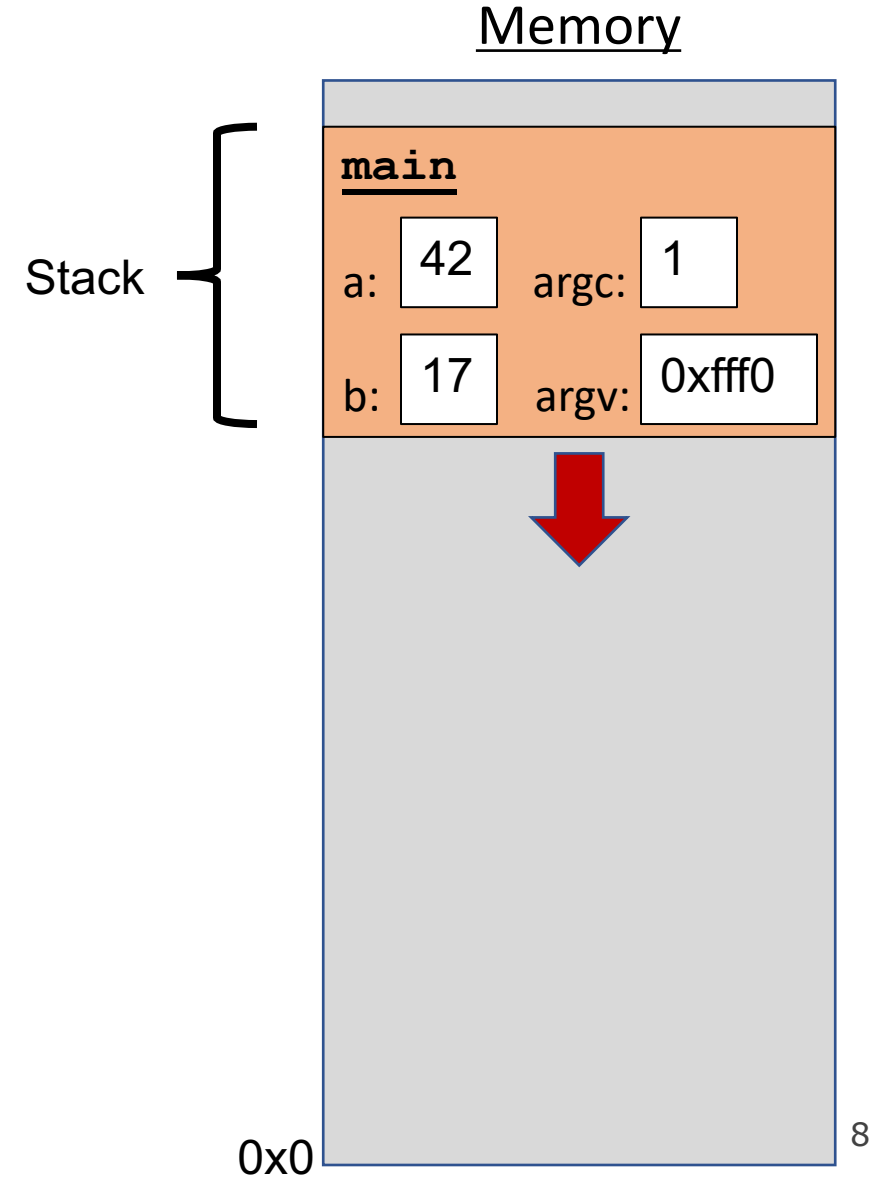
0x0

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

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

```
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;
}
```
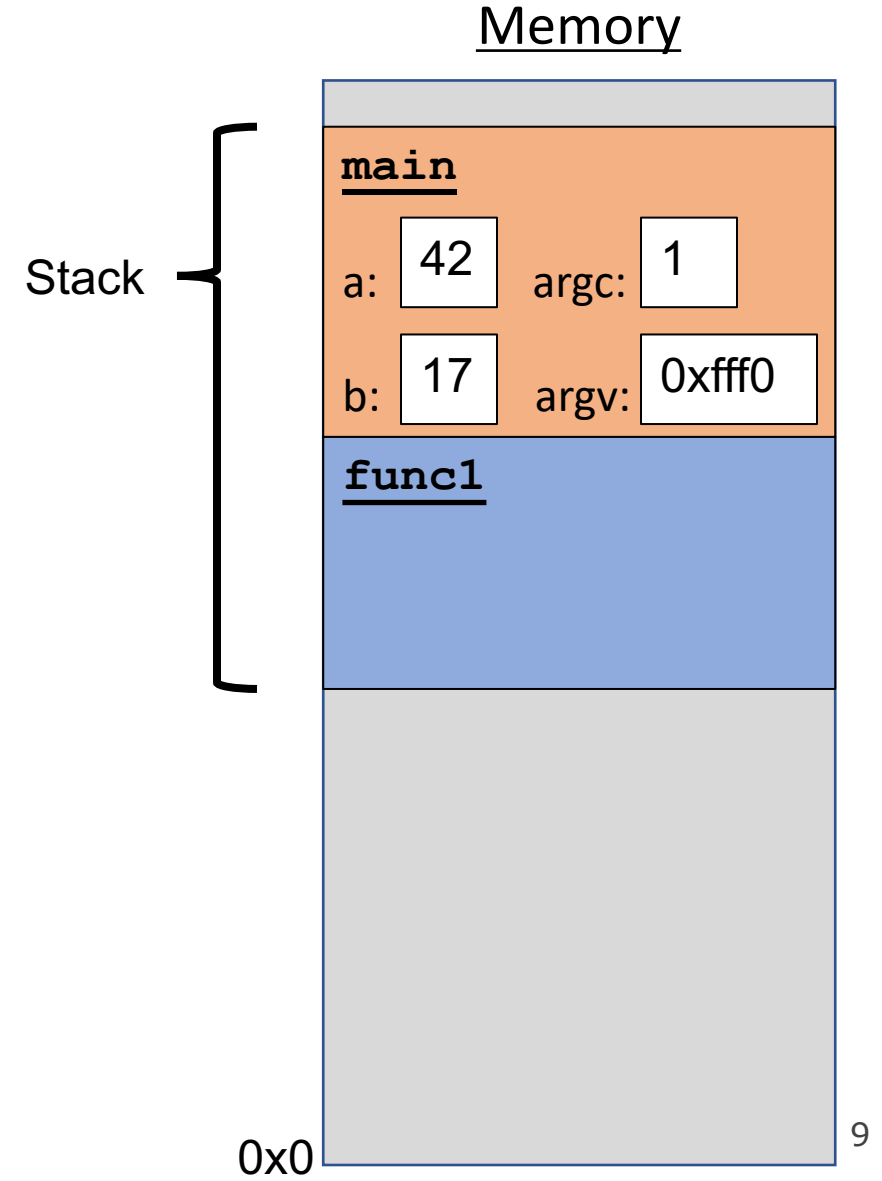
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

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

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

0x0

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;
}
```
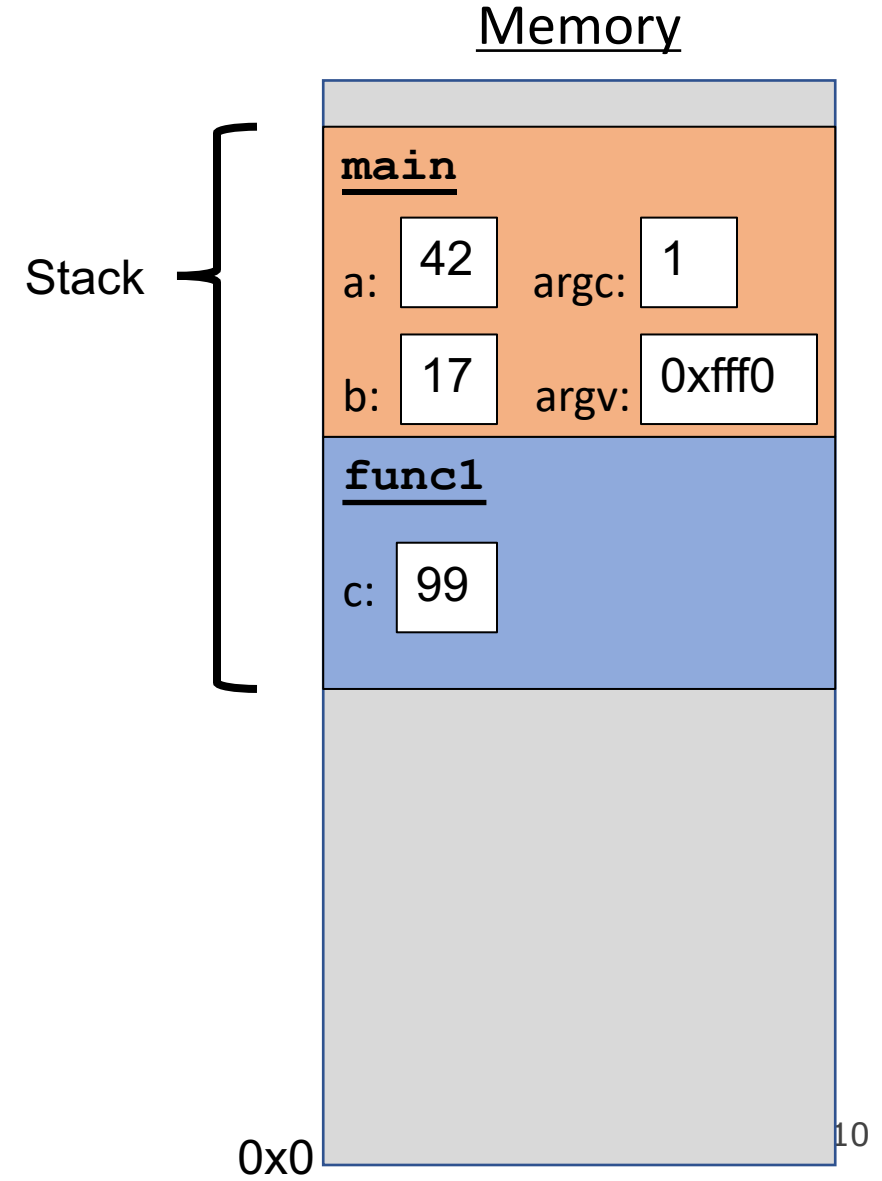
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

0x0

10

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;
}
```
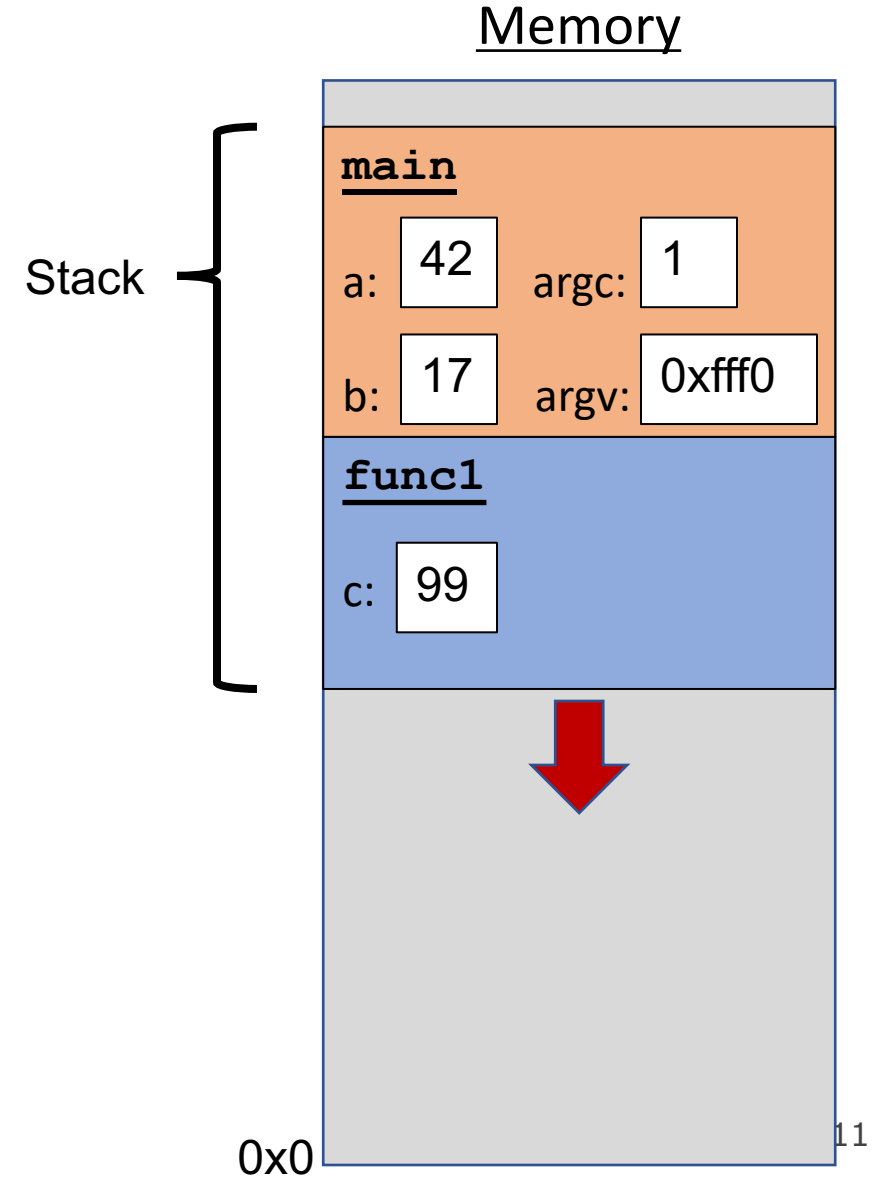
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

0x0

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;
}
```
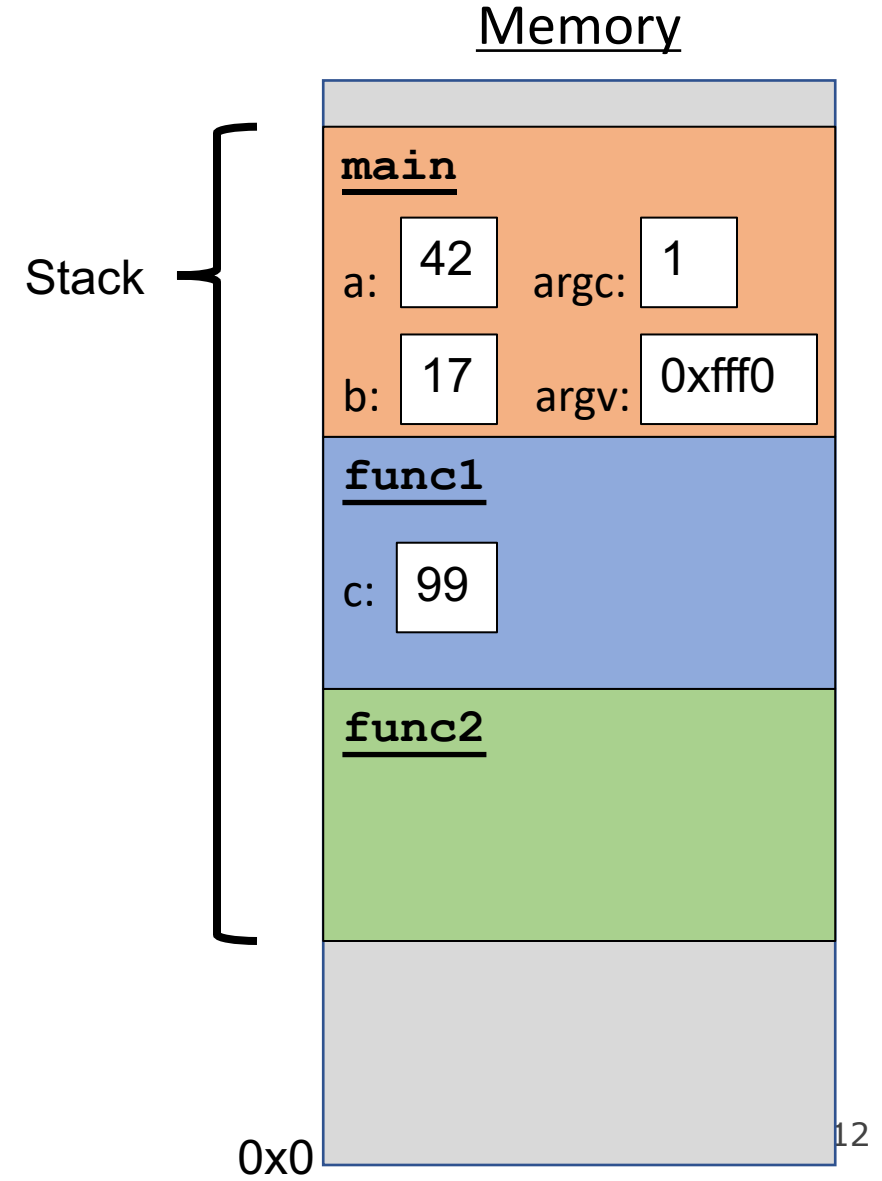
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

0x0

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;
}
```
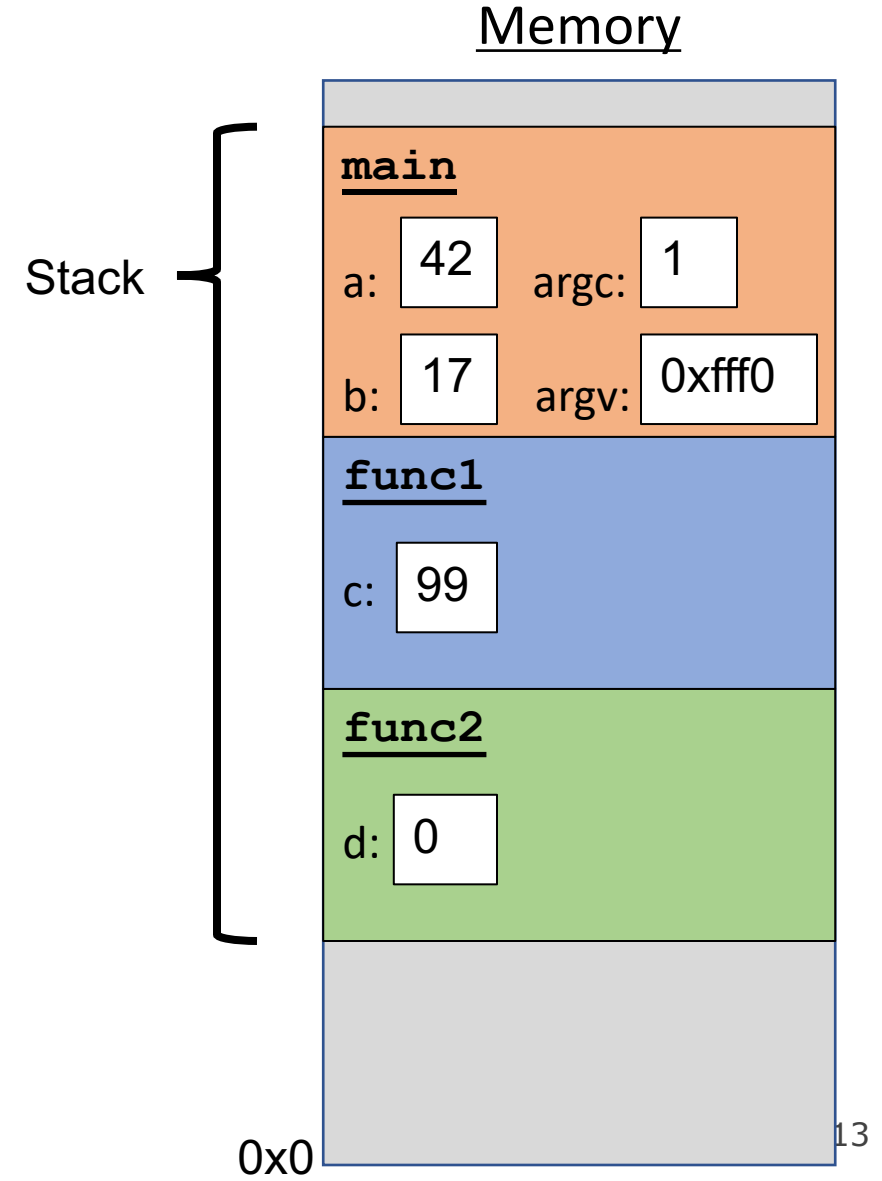
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

d: 0

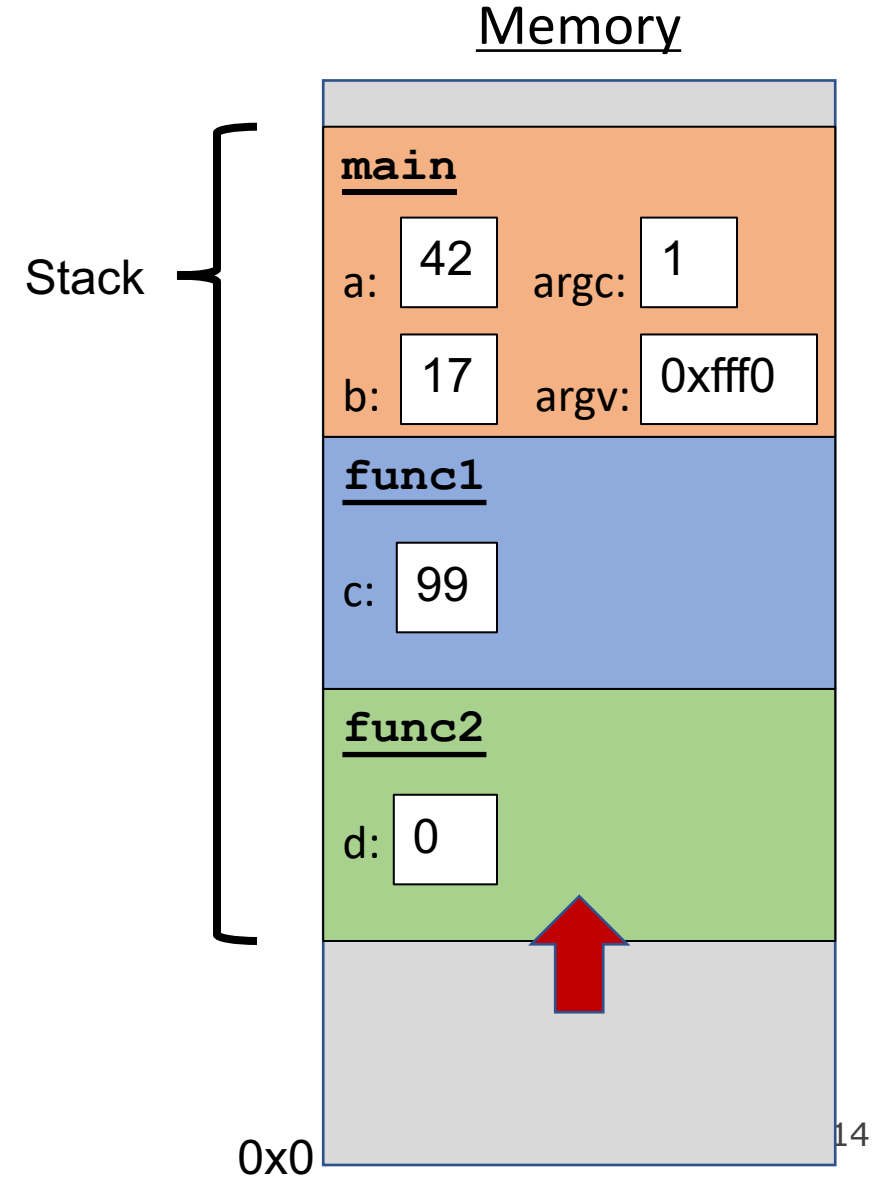0x0

13

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

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

d: 0

0x0

14

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;
}
```
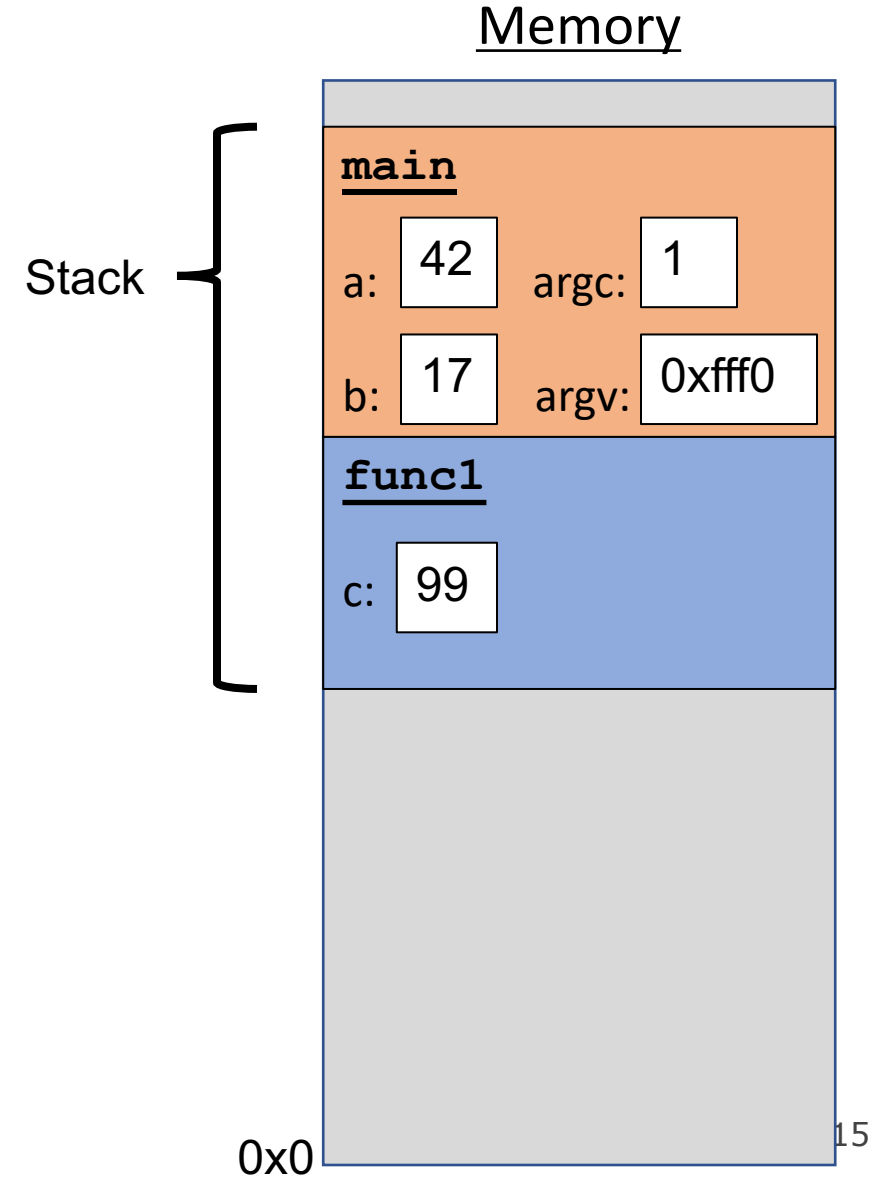
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99
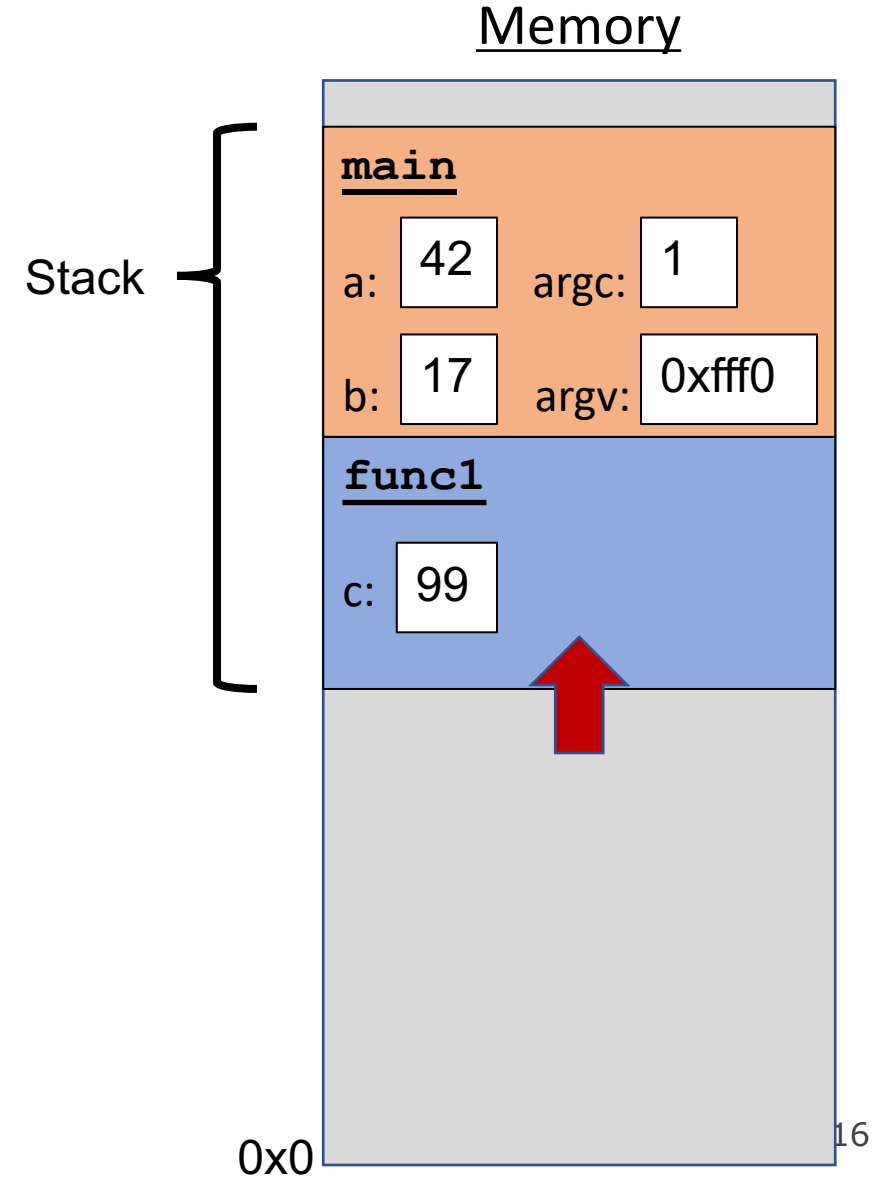
0x0

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

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

0x0

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

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

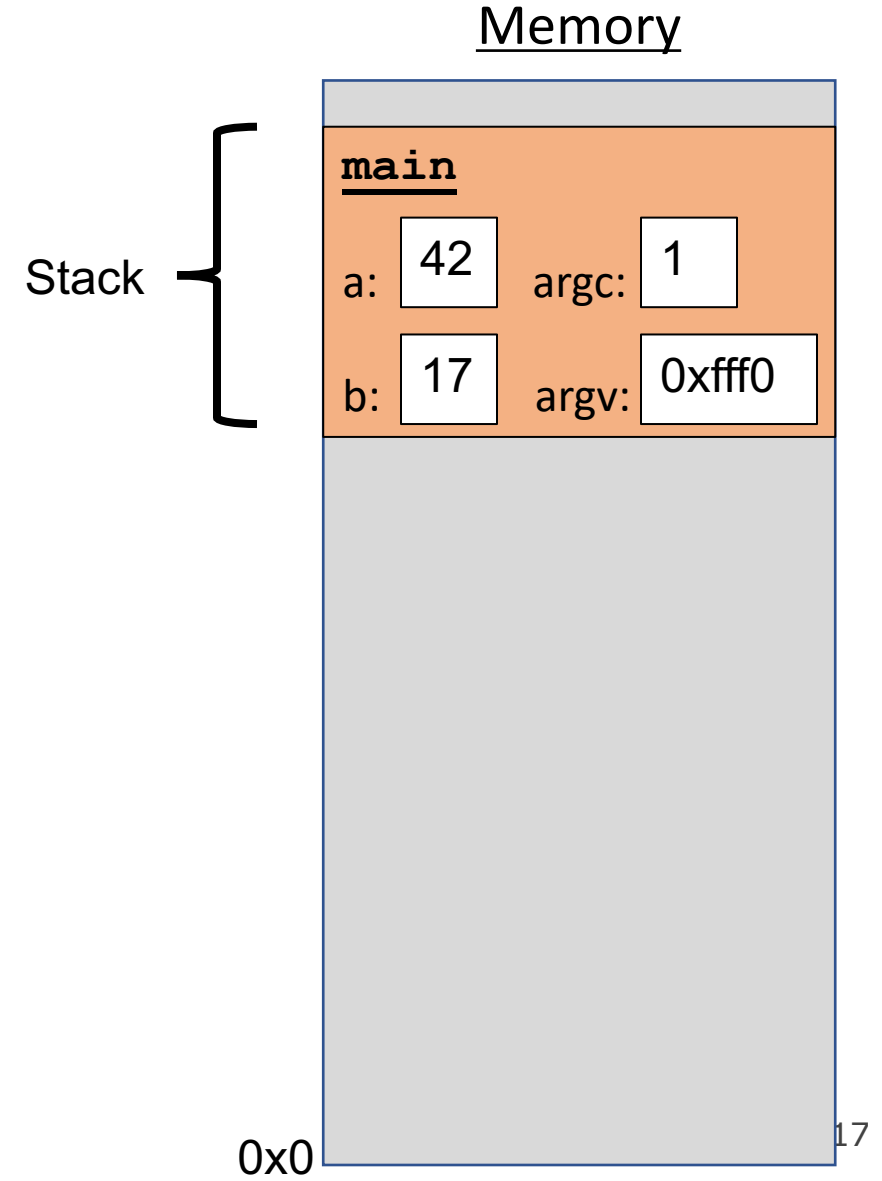# The Stack

```
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

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

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;
}
```
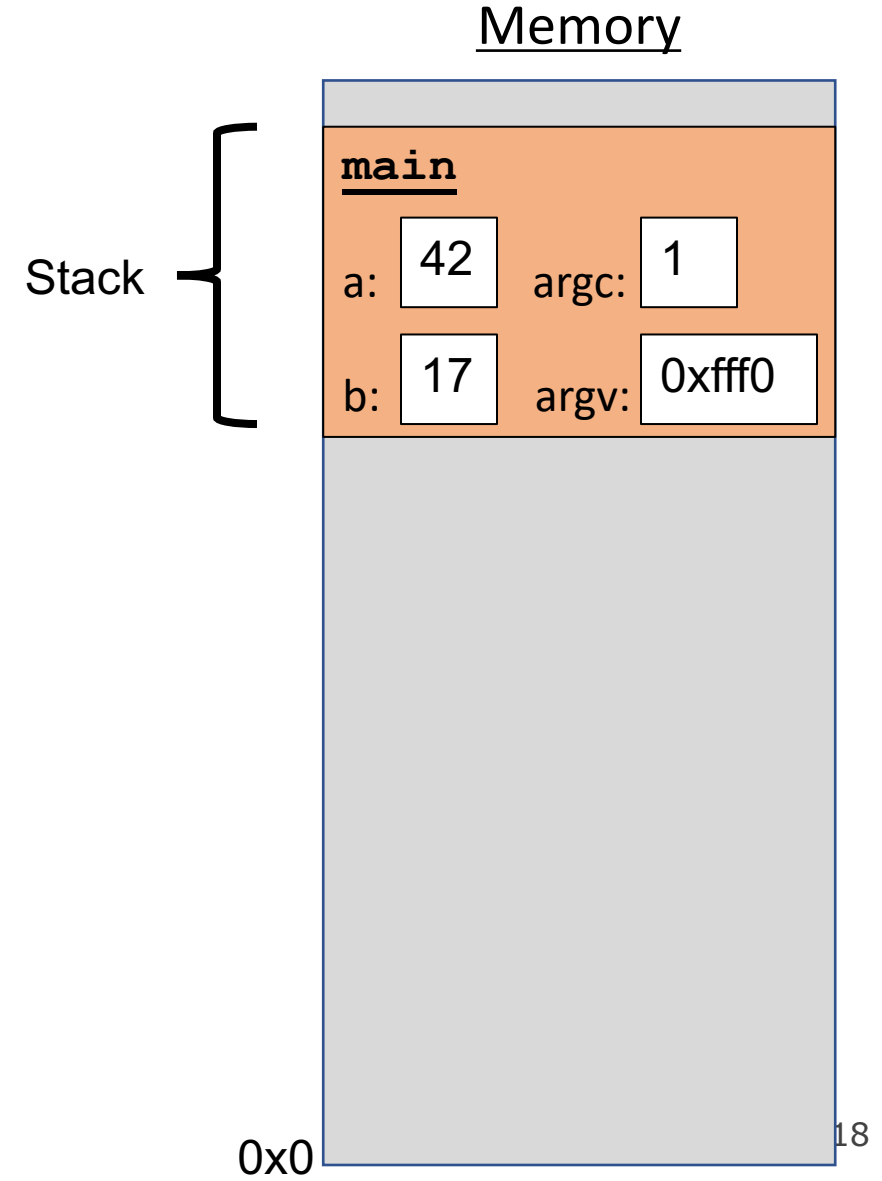
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

0x0

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;
}
```
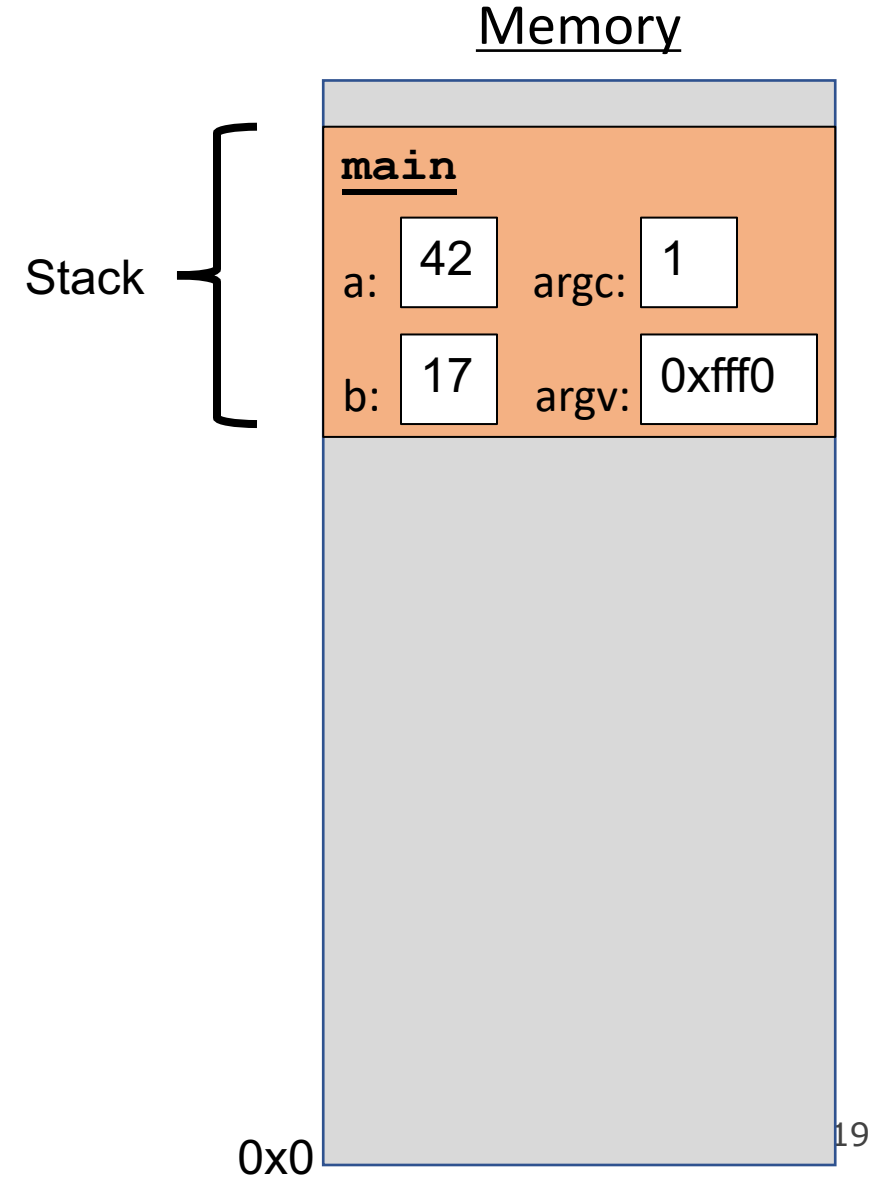
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0
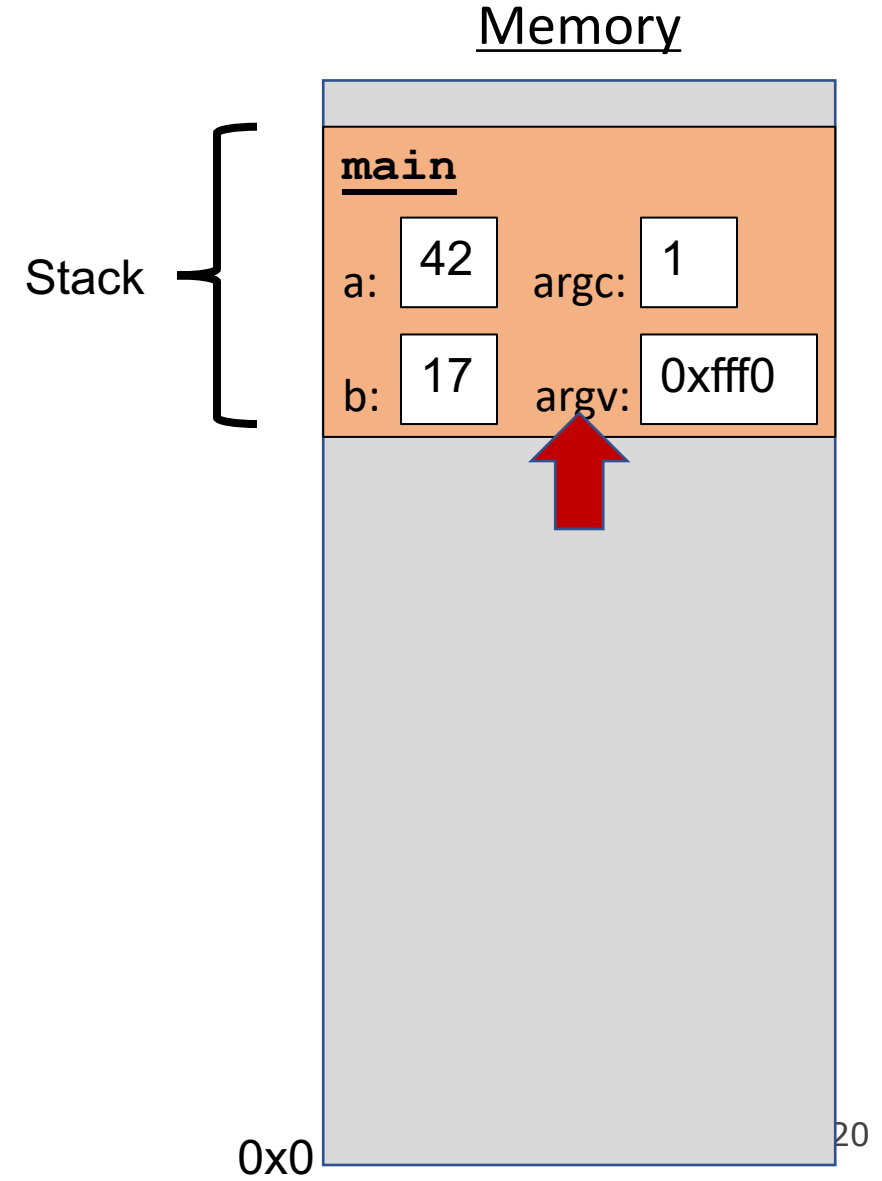
0x0

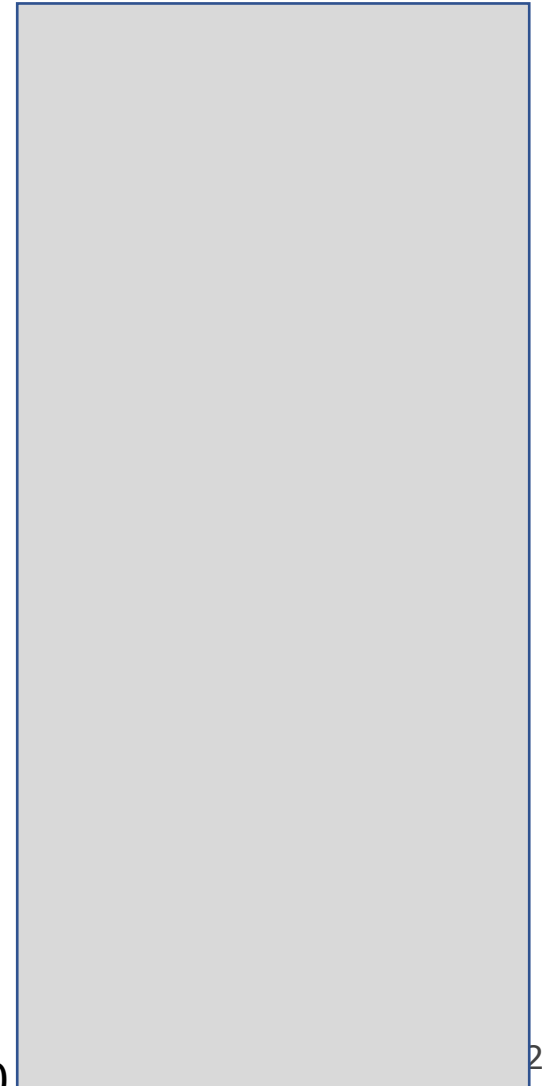# The Stack

```
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;
}
```

0x0

The Stack

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

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

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

Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

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;
}
```

Memory

Stack
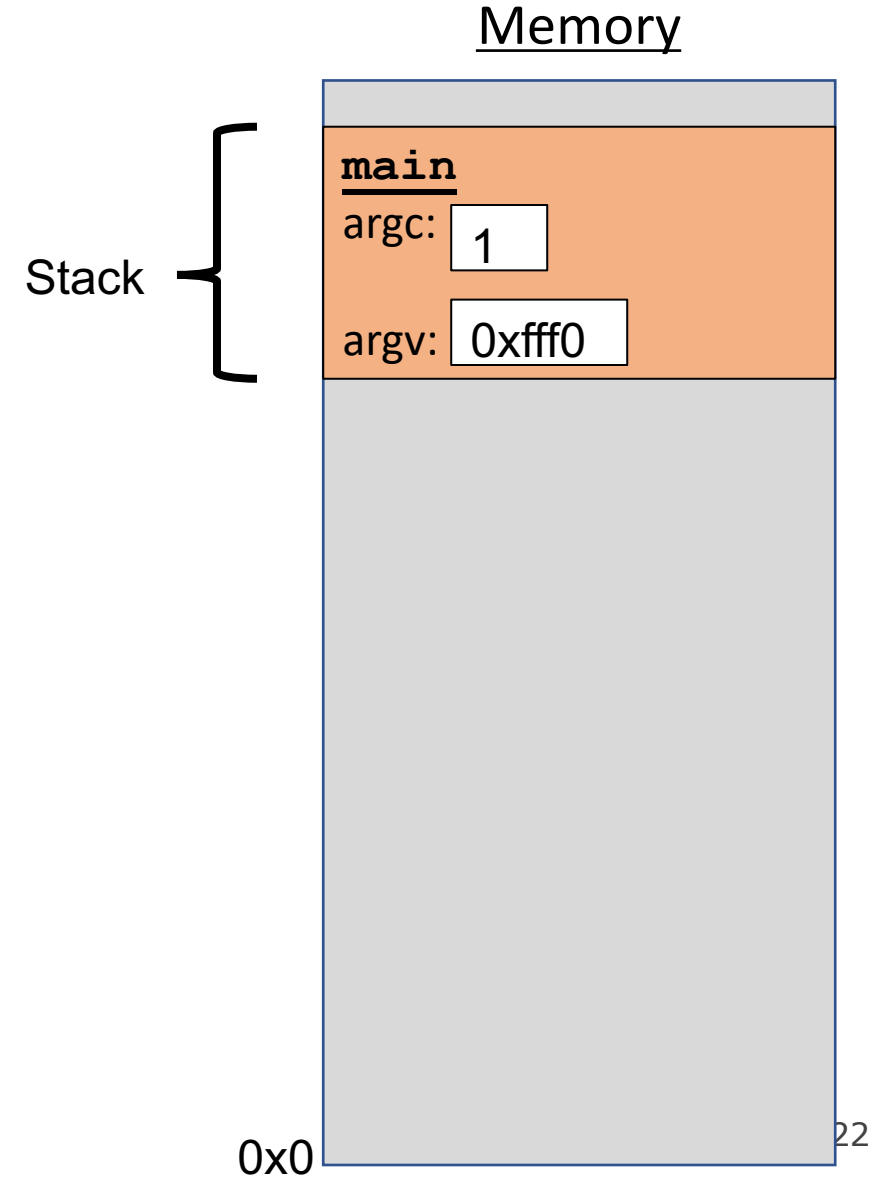
main

argc: 1

argv: 0xfff0

0x0

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;
}
```

Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

0x0

The Stack

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

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

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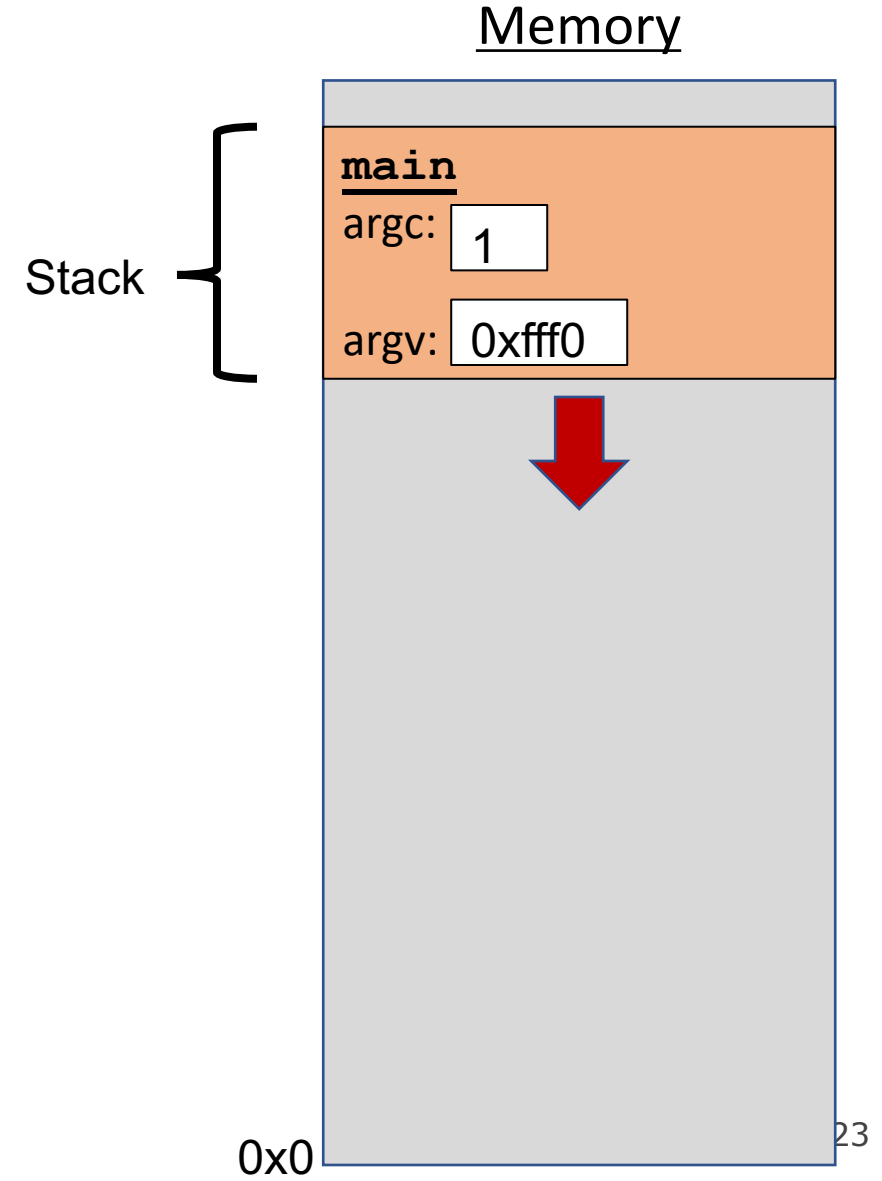
Stack

main

argc: 1

argv: 0xfff0

factorial

n: 4

0x0

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
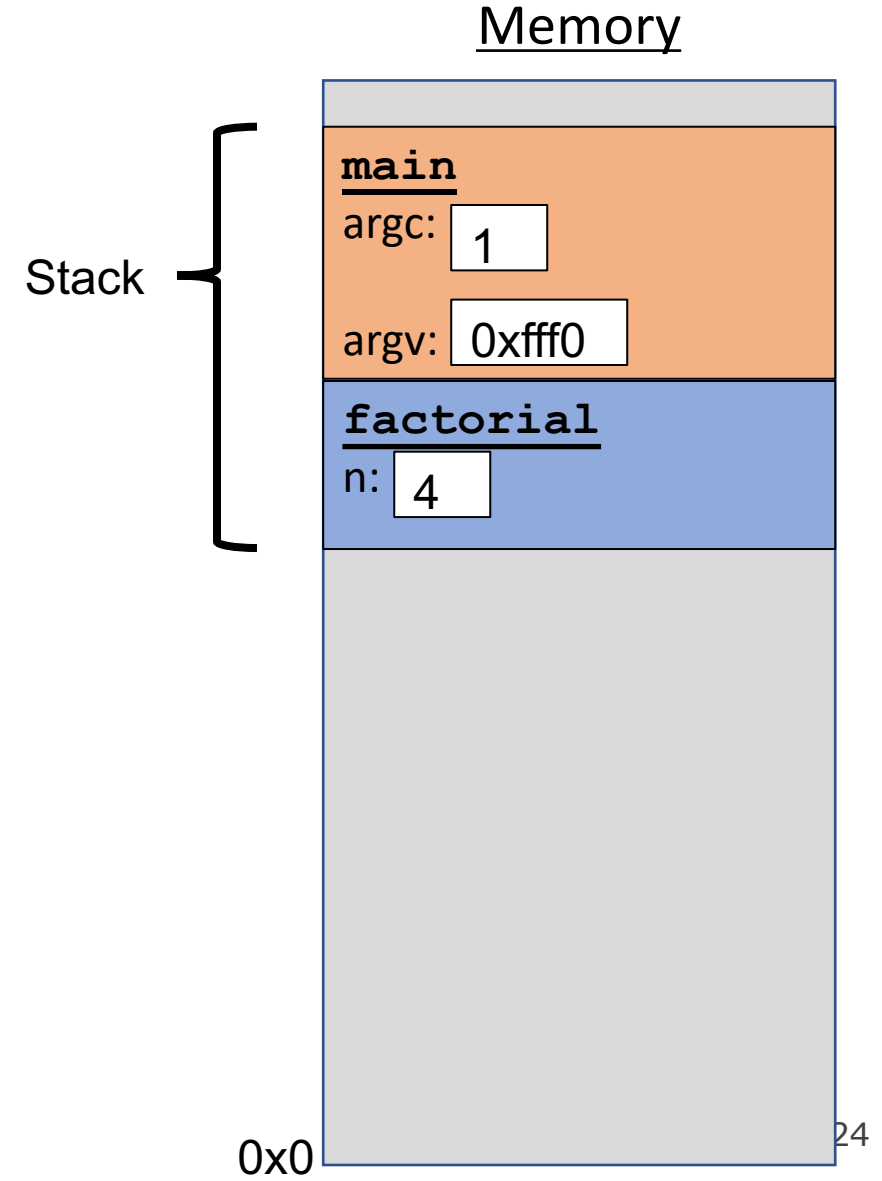
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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


Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
n: 3

0x0

The Stack

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

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

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

Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
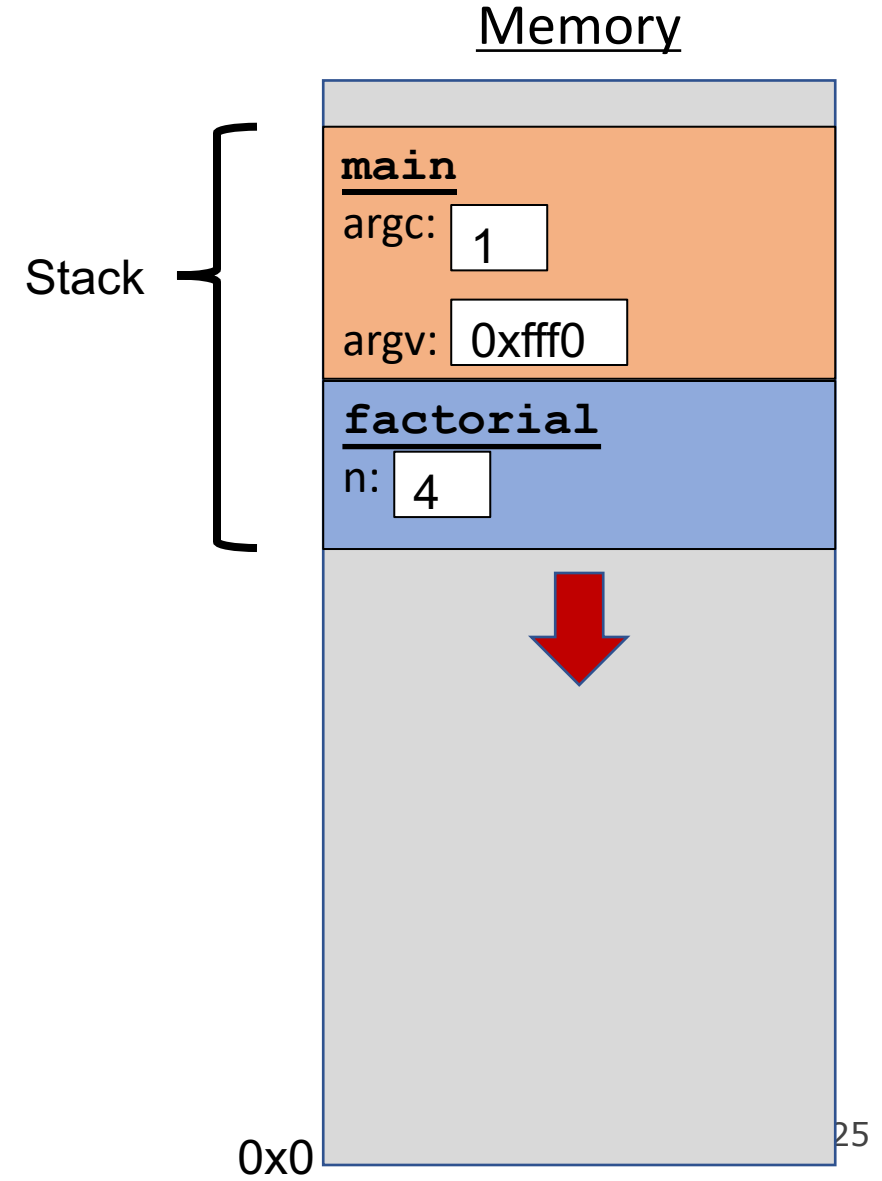n: 3

0x0

27

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;
}
```
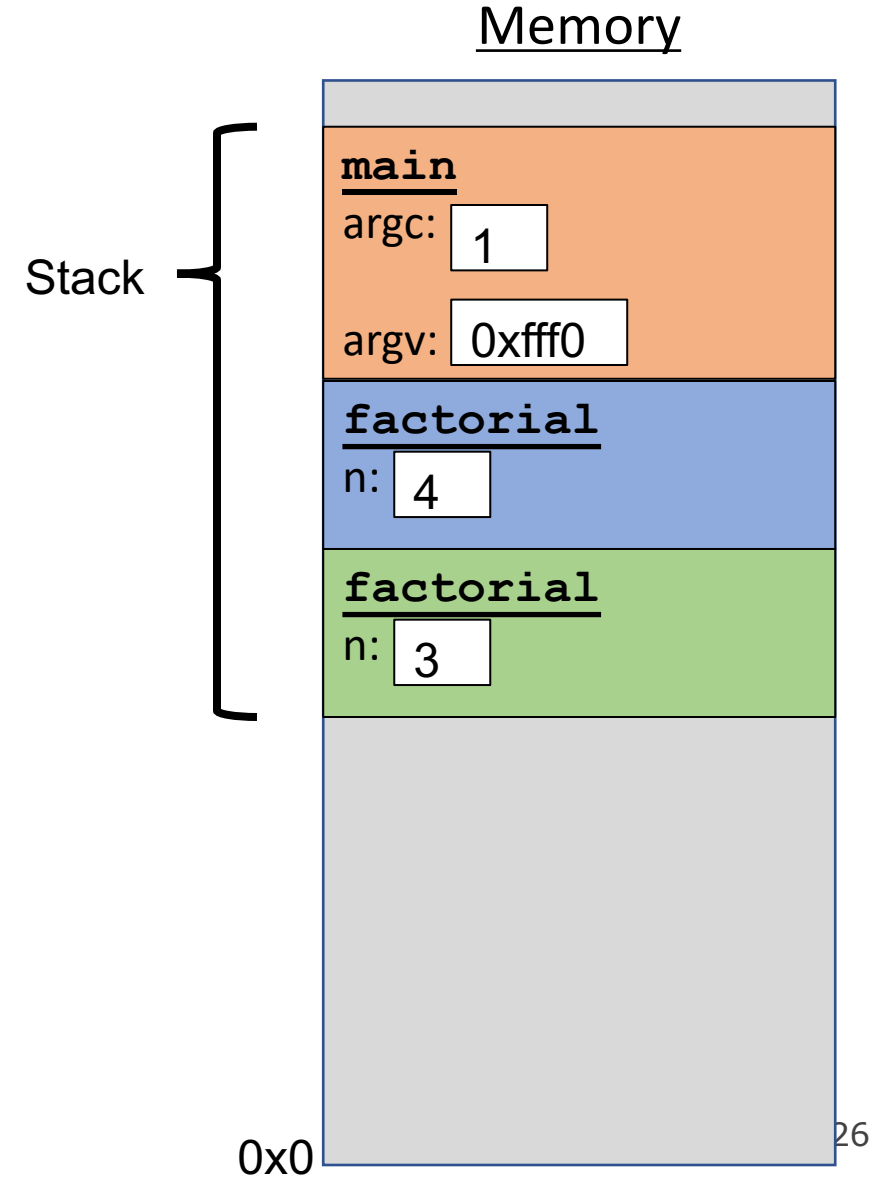
Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
n: 3

factorial
n: 2

0x0

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
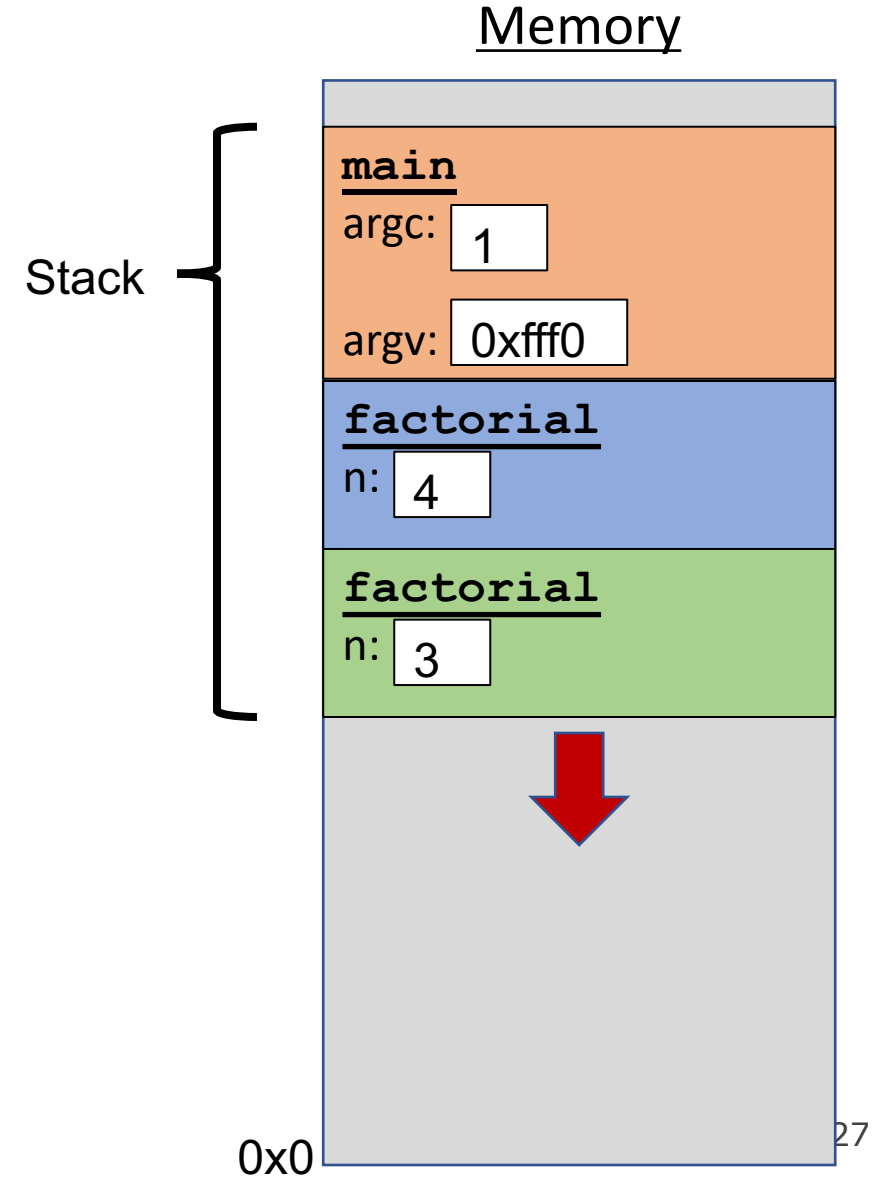
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
n: 3

factorial
n: 2

0x0

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
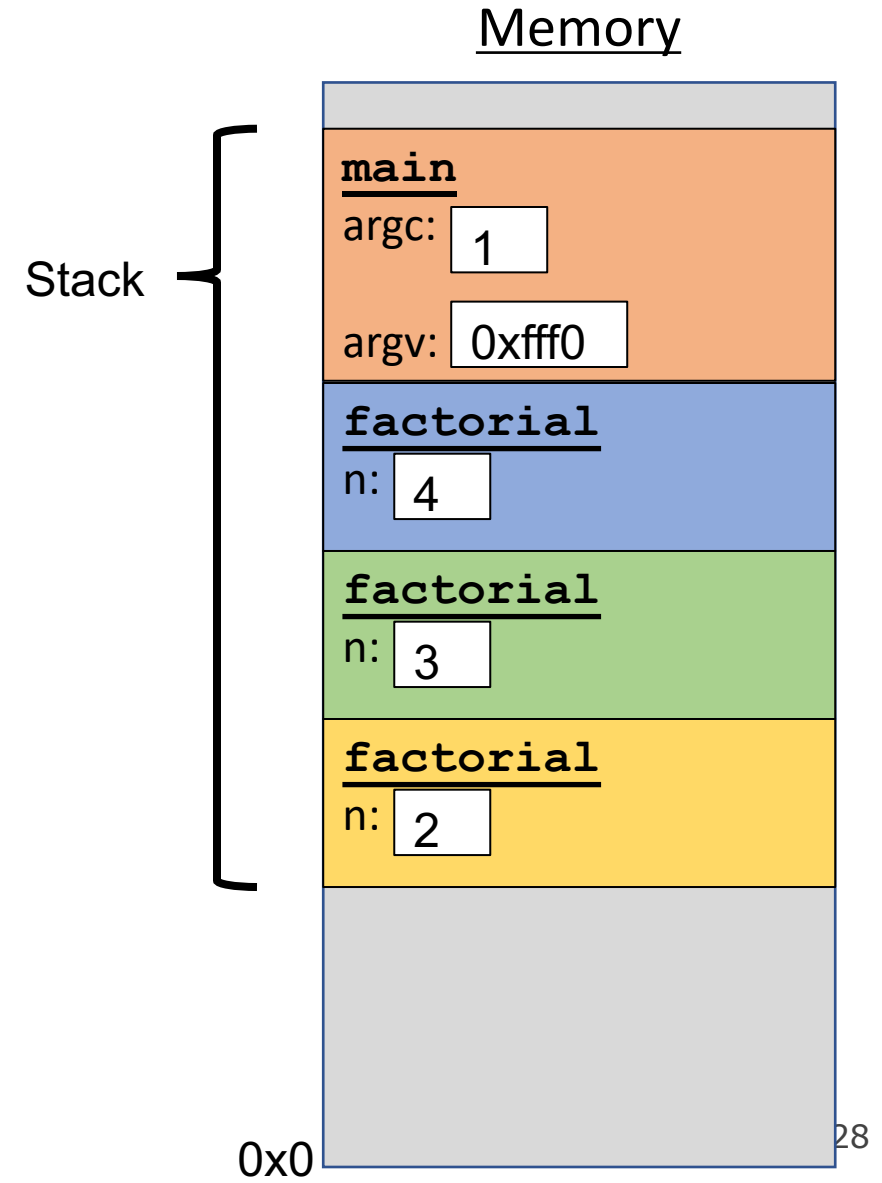
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
n: 3

factorial
n: 2

factorial
n: 1

0x0

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
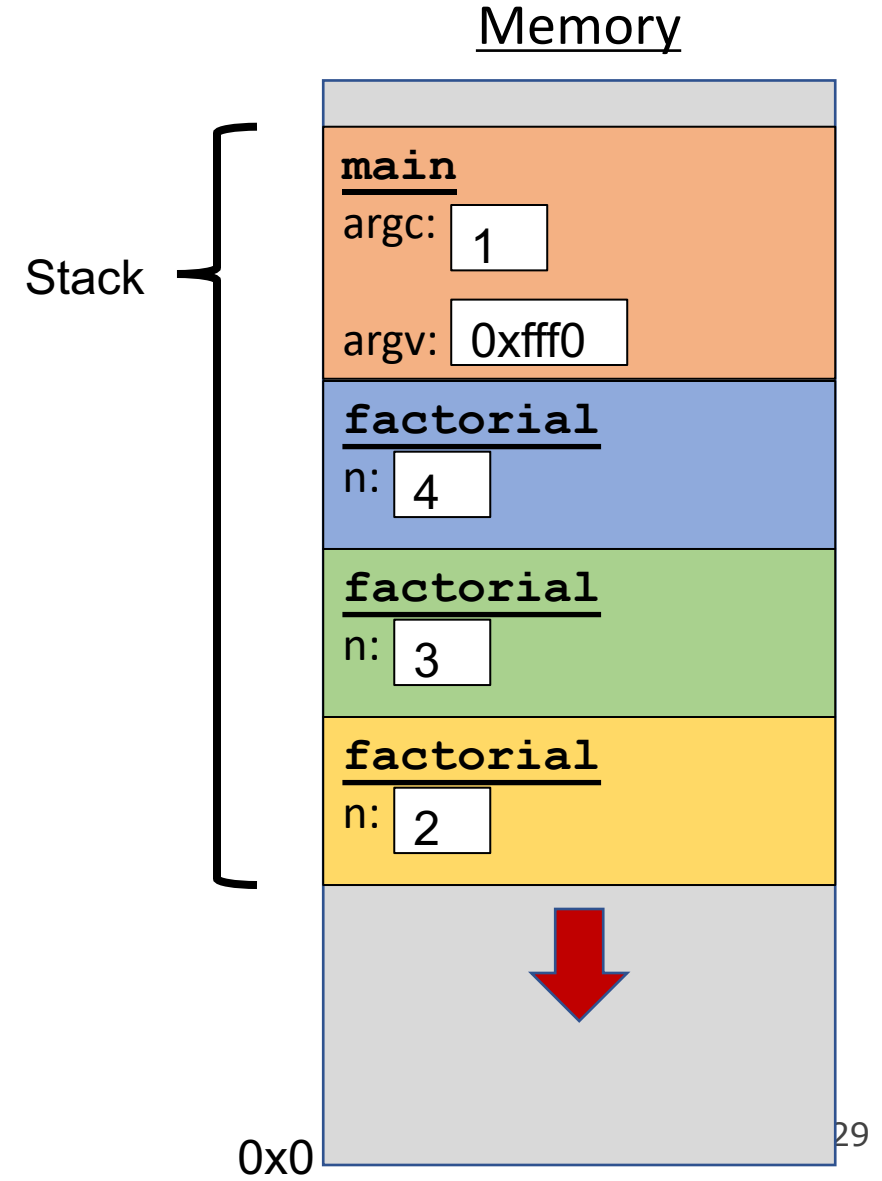
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

Memory

Stack

main	
argc:	1
argv:	0xfff0

factorial	
n:	4

factorial	
n:	3

factorial	
n:	2

factorial	
n:	1

Returns 1

0x0

31

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
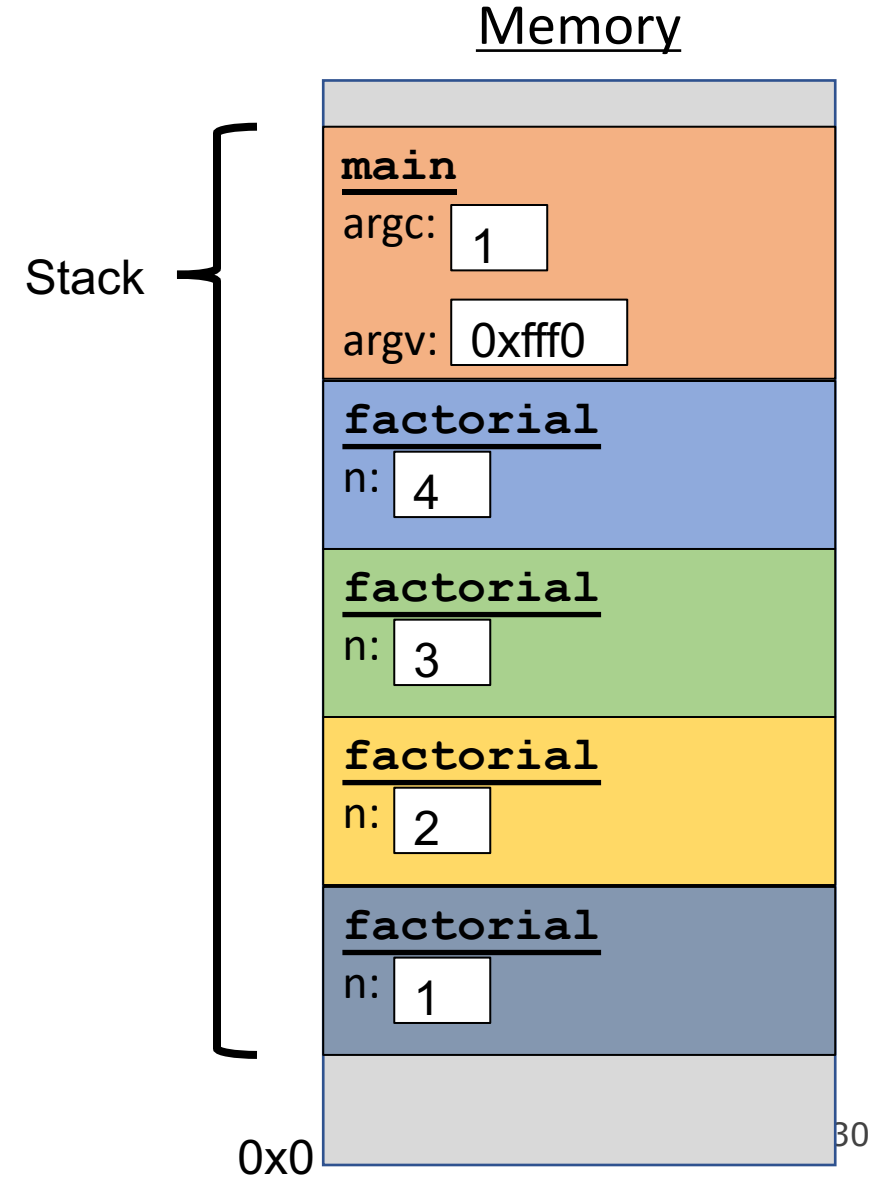
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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


Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
n: 3
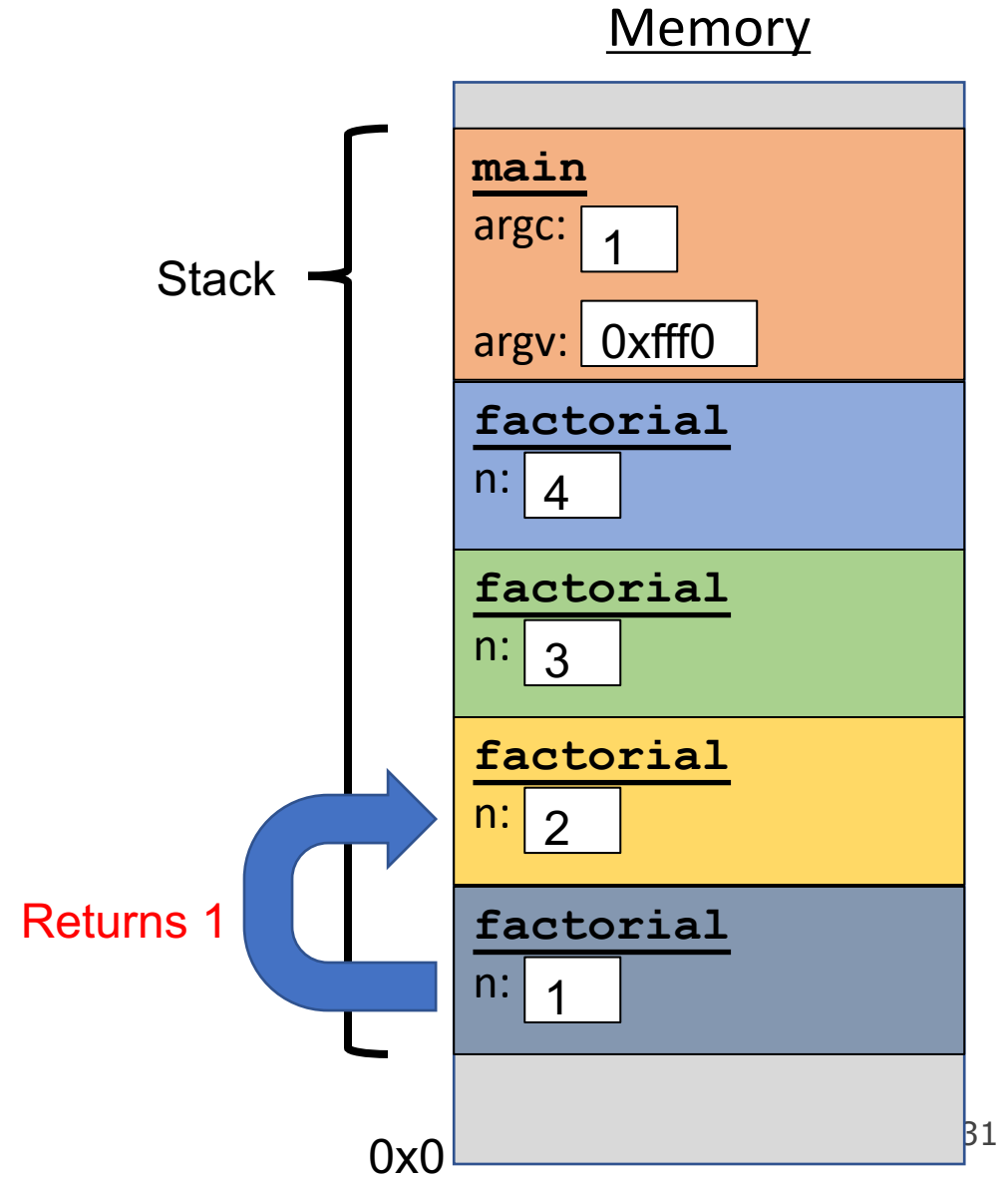
Returns 2

factorial
n: 2

0x0

The Stack

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

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

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

Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

Returns 6

factorial
n: 3

0x0

The Stack

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

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

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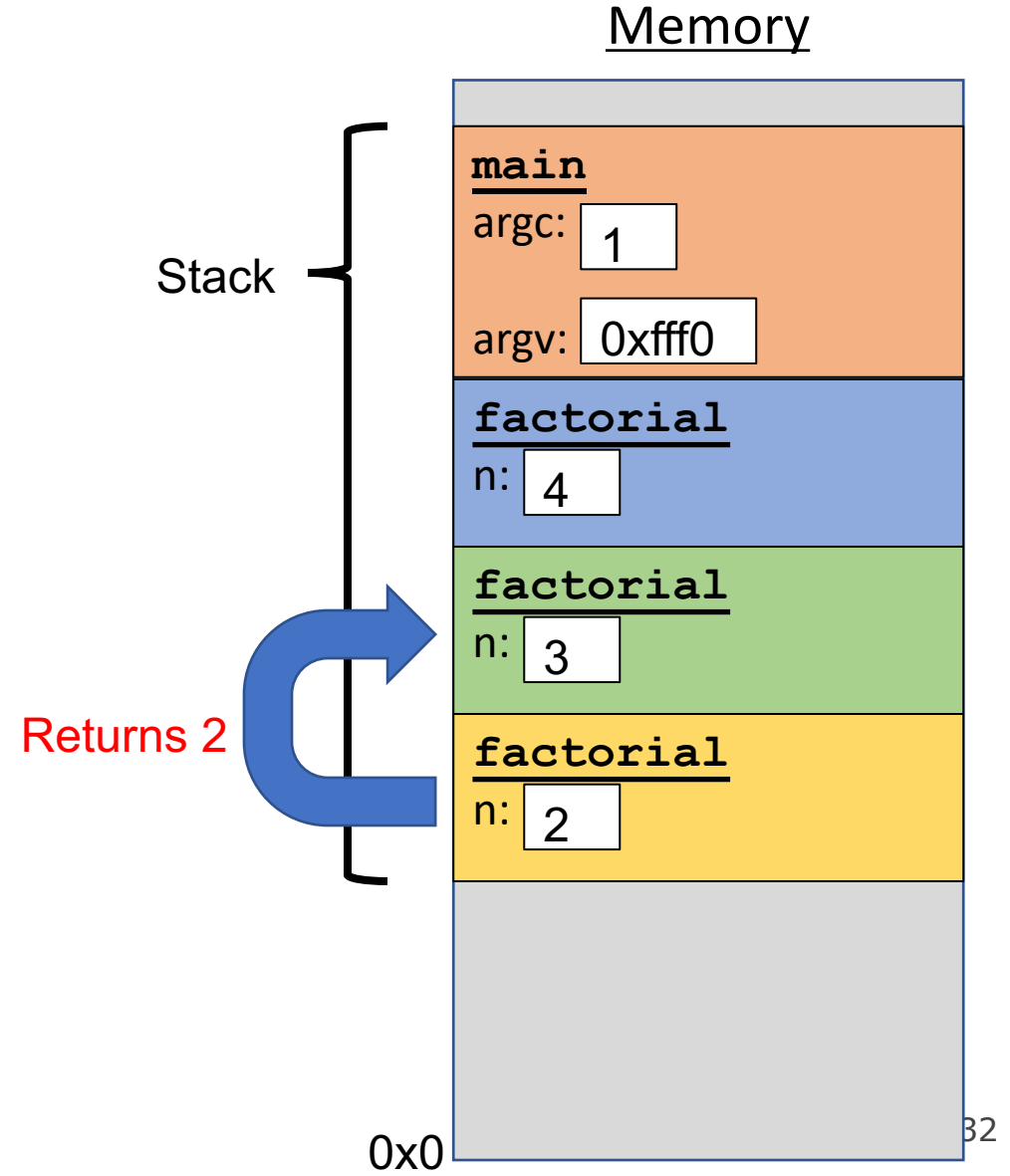
Memory

Stack

Returns 24

main

argc: 1

argv: 0xfff0

factorial

n: 4

0x0

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
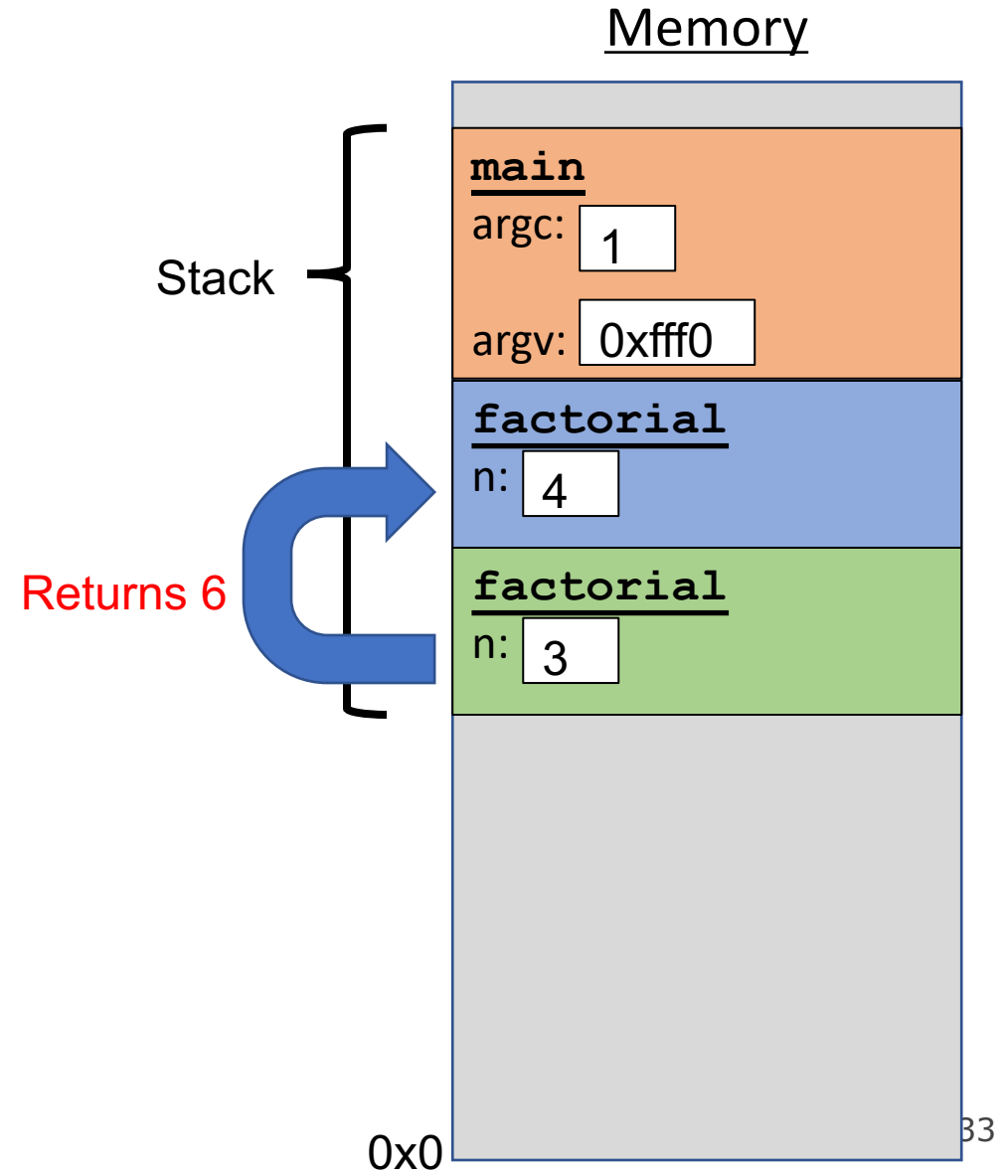
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

35

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.
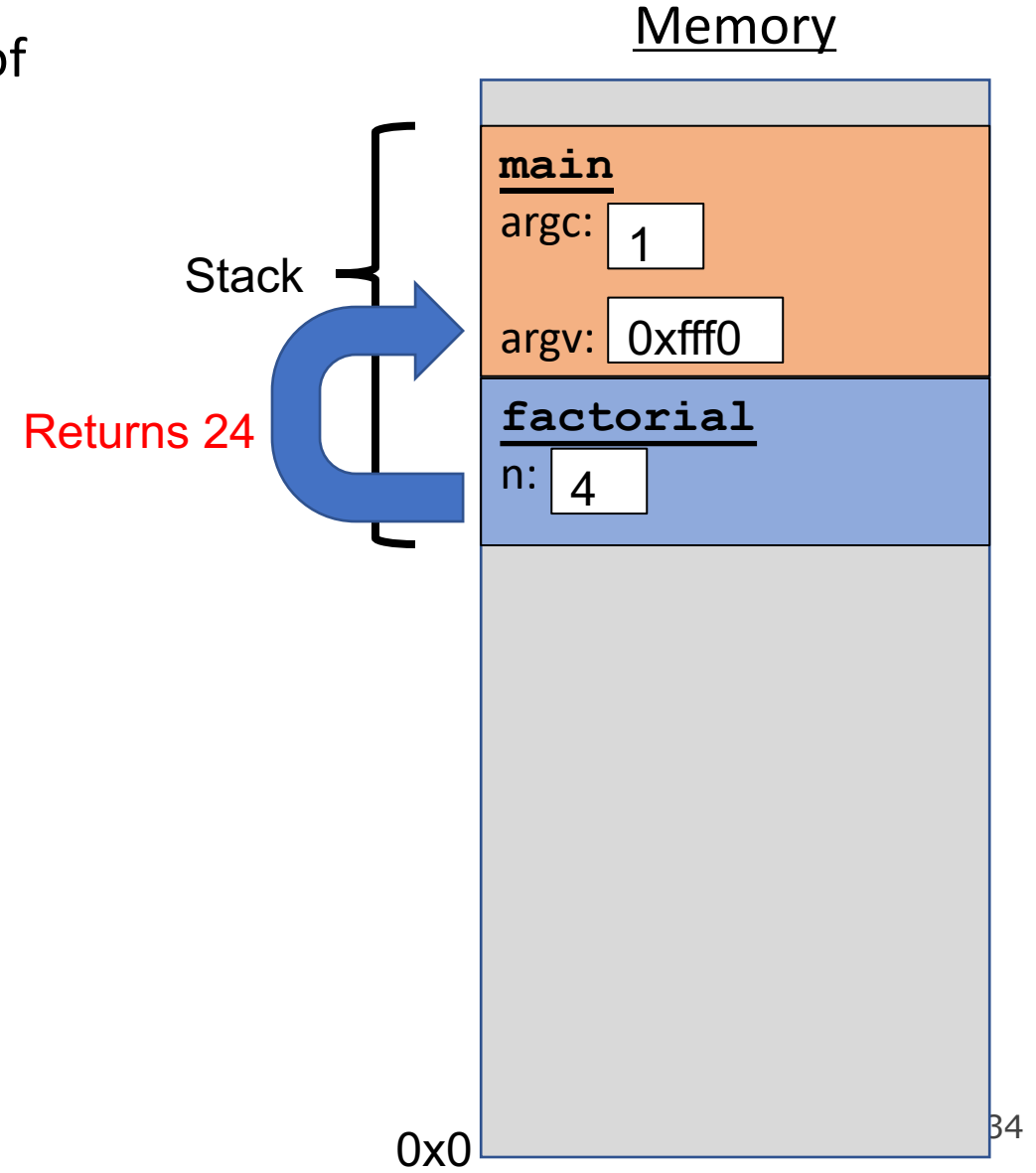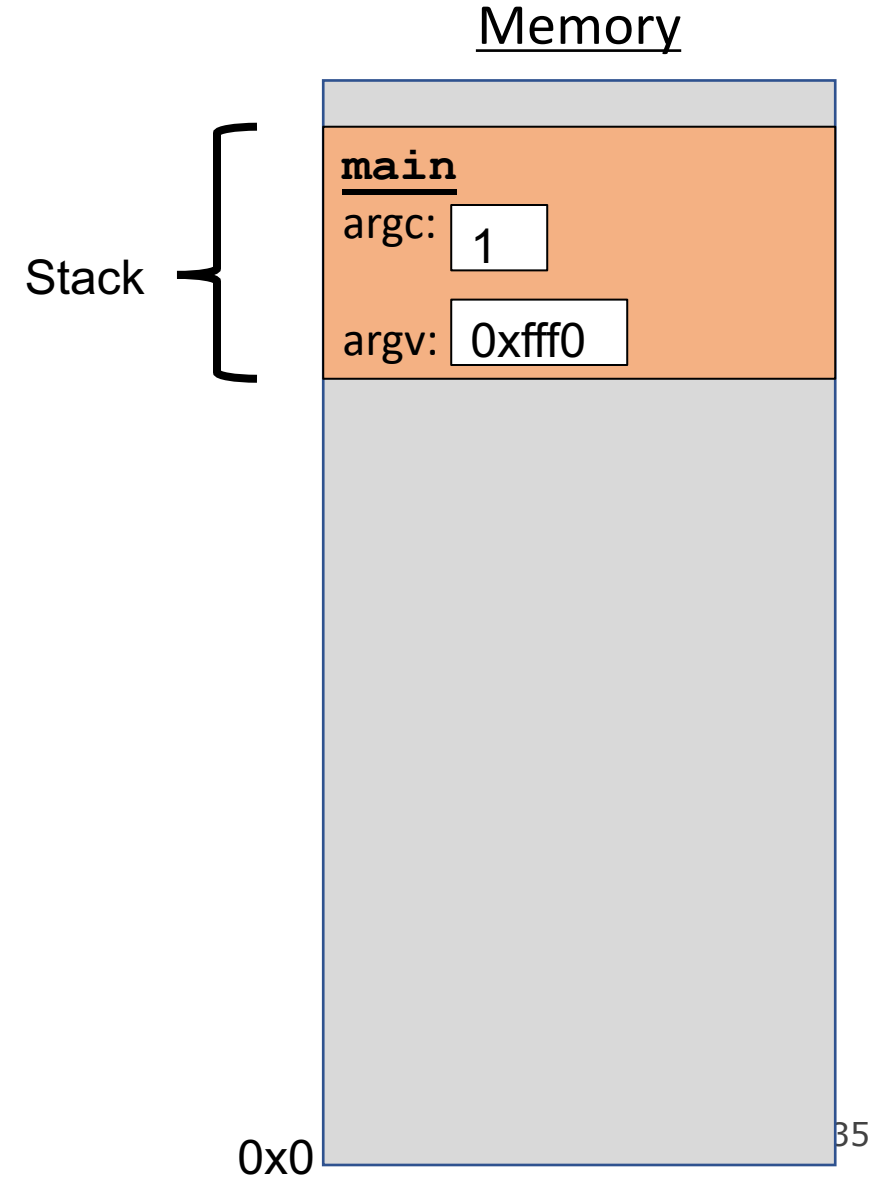
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

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?

```
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;
}
```

Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

38

```
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;
}
```

Memory

Stack

main

argc: 1   str:

argv: 0xfff0

0x0

The Stack

```c
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;
}
```

Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

0x0

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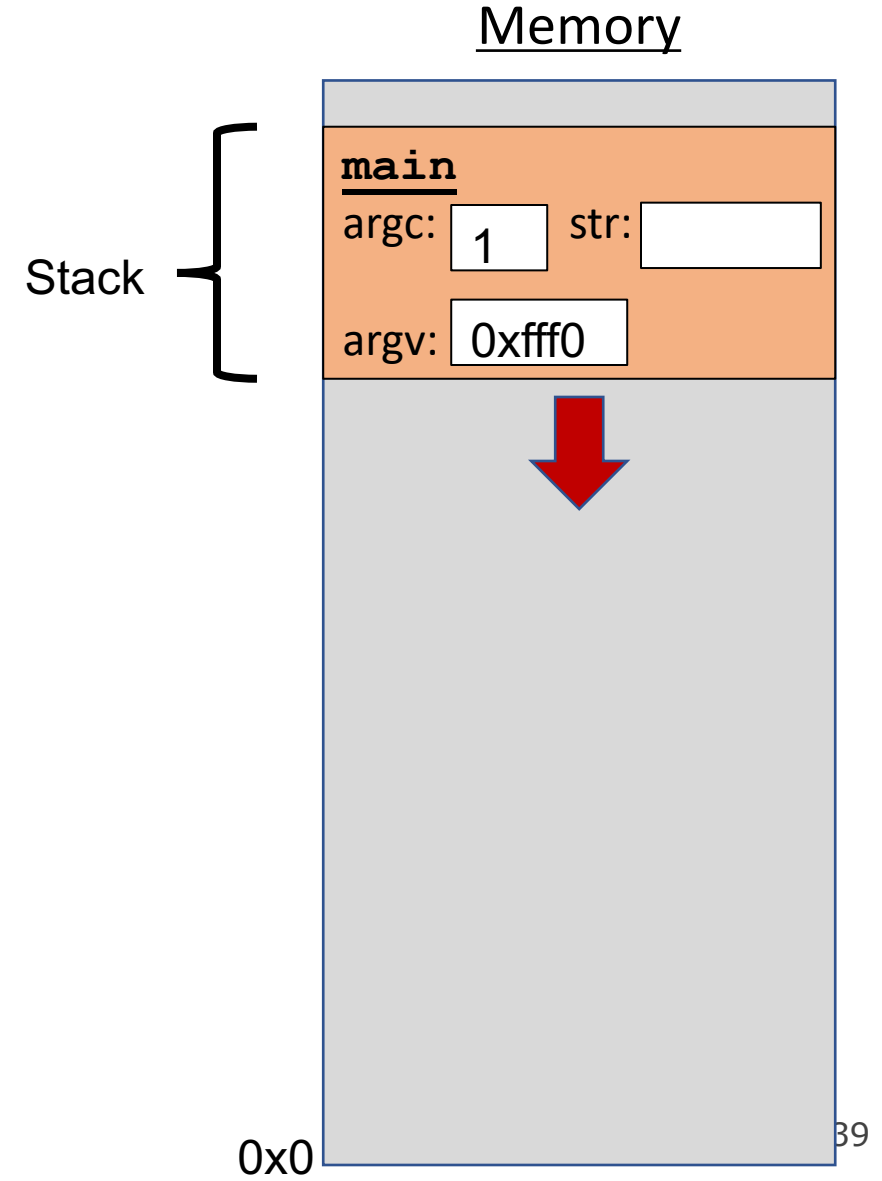
```c
char *create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

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

Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str:
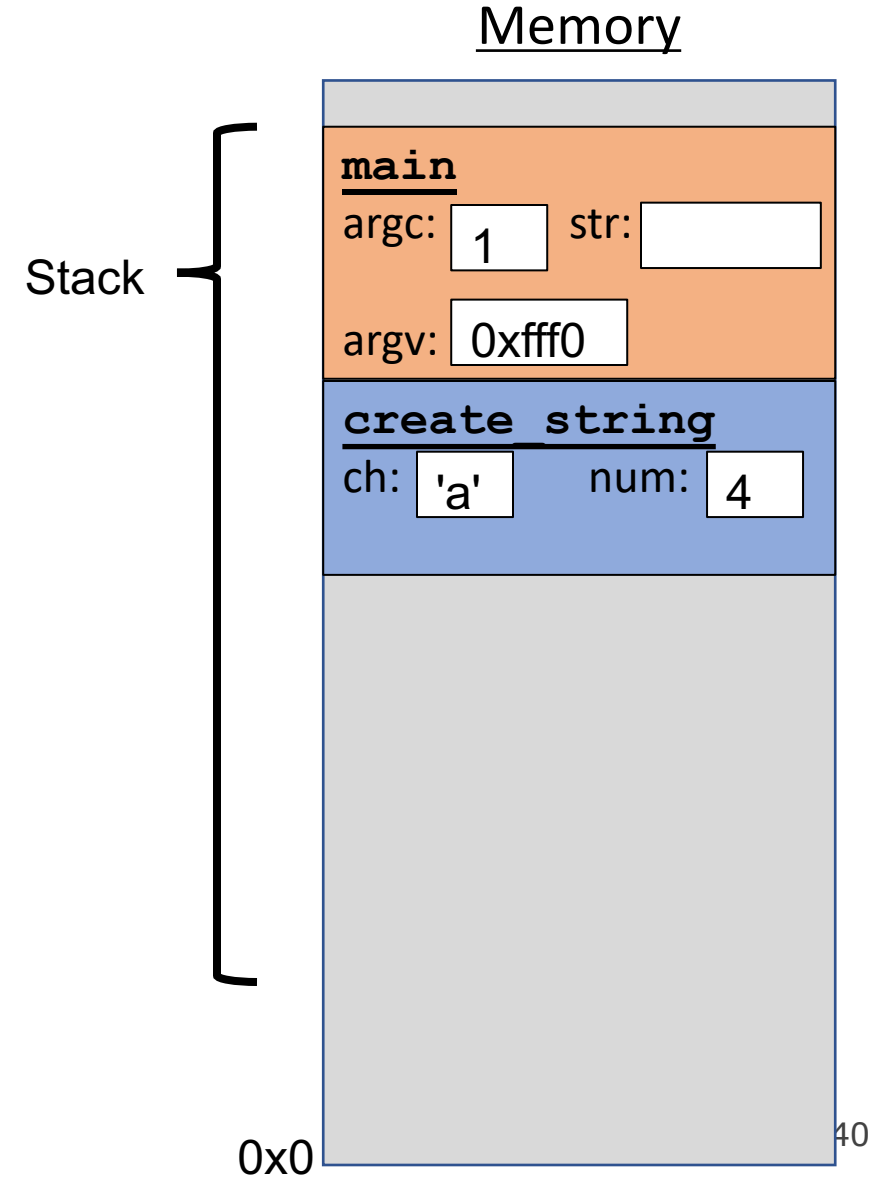
0x0

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;
}
```

Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

new_str:
'\0'
'a'
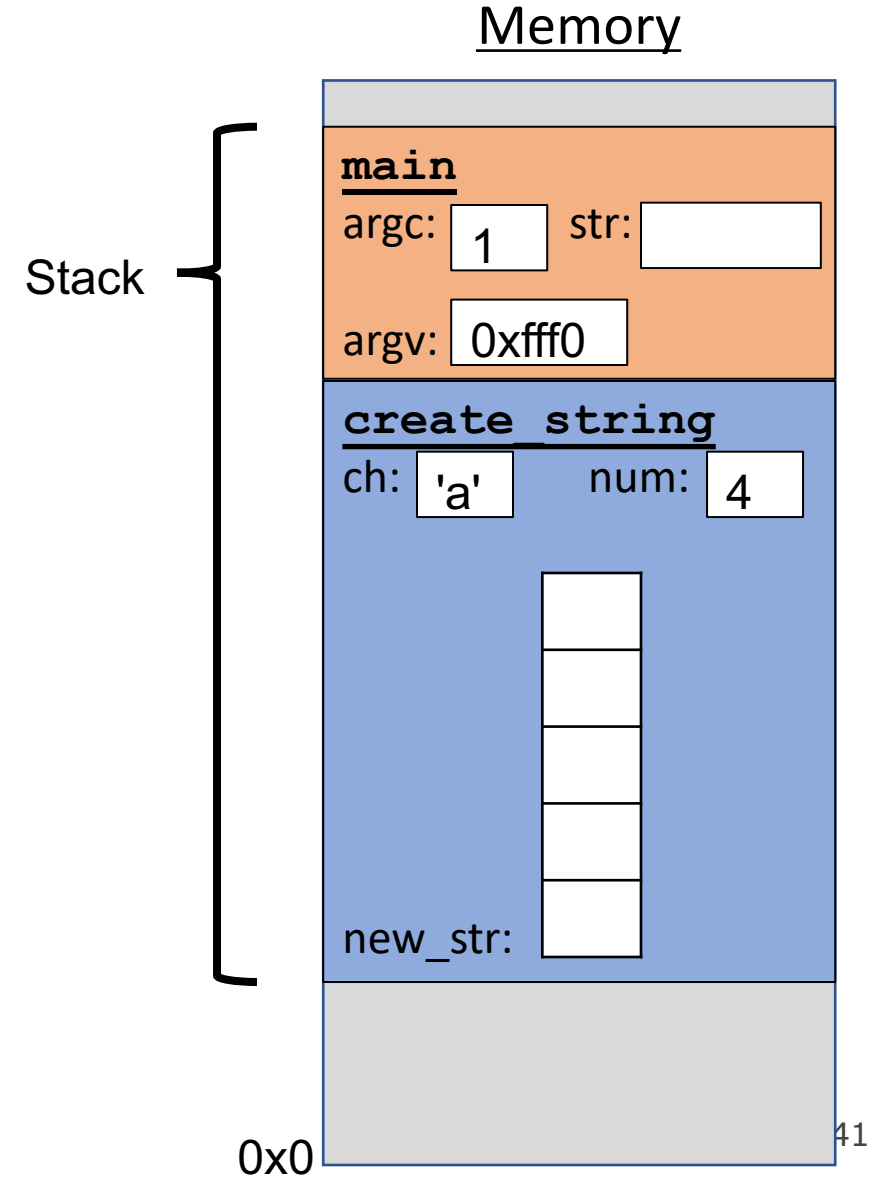'a'
'a'
'a'

0x0

42

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;
}
```

Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

Returns e.g. 0xff50

new_str:
'\0'
'a'
'a'
'a'
'a'

0x0
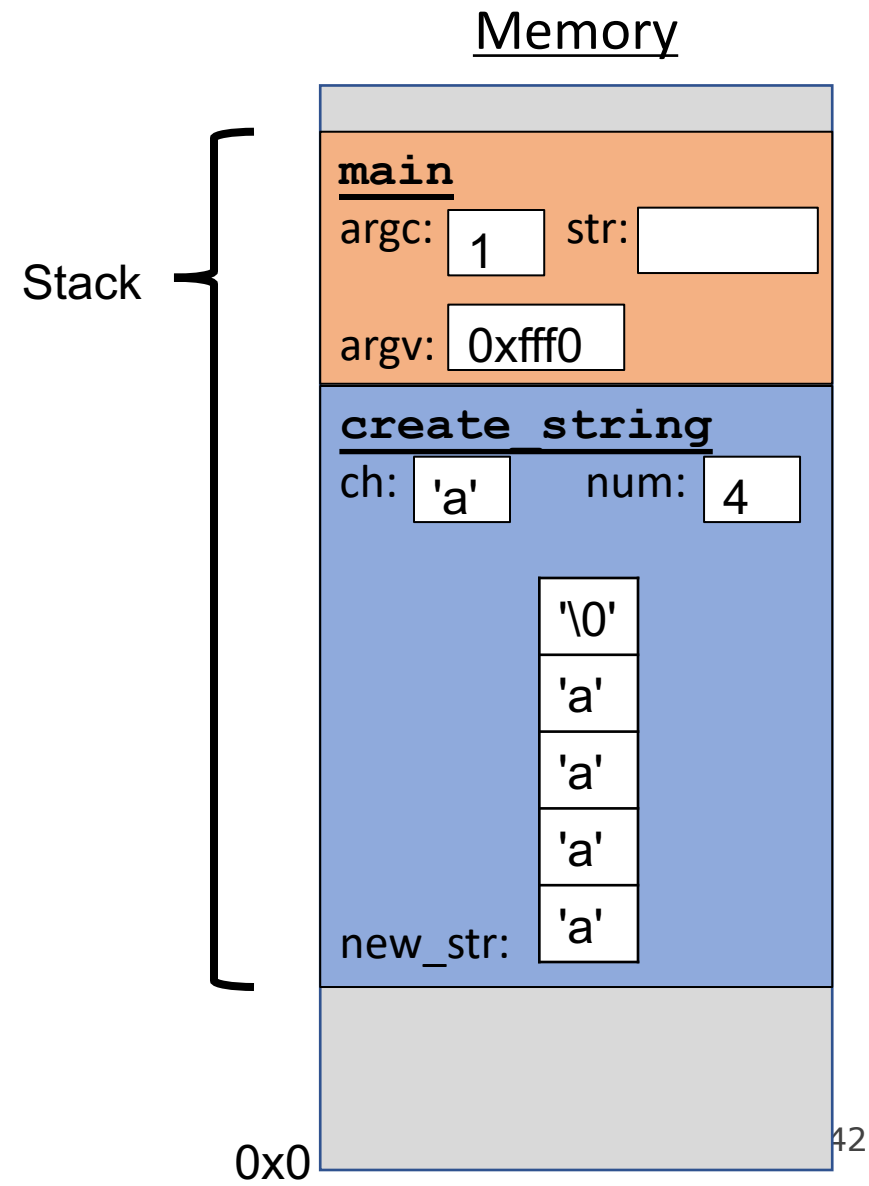
The Stack

```c
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;
}
```

Memory

Stack

main

argc: 1   str: **0xff50**

argv: 0xfff0

create_string

ch: 'a'   num: 4

'\0'

'a'

'a'

'a'

new_str: 'a'

0x0

The Stack

```c
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;
}
```
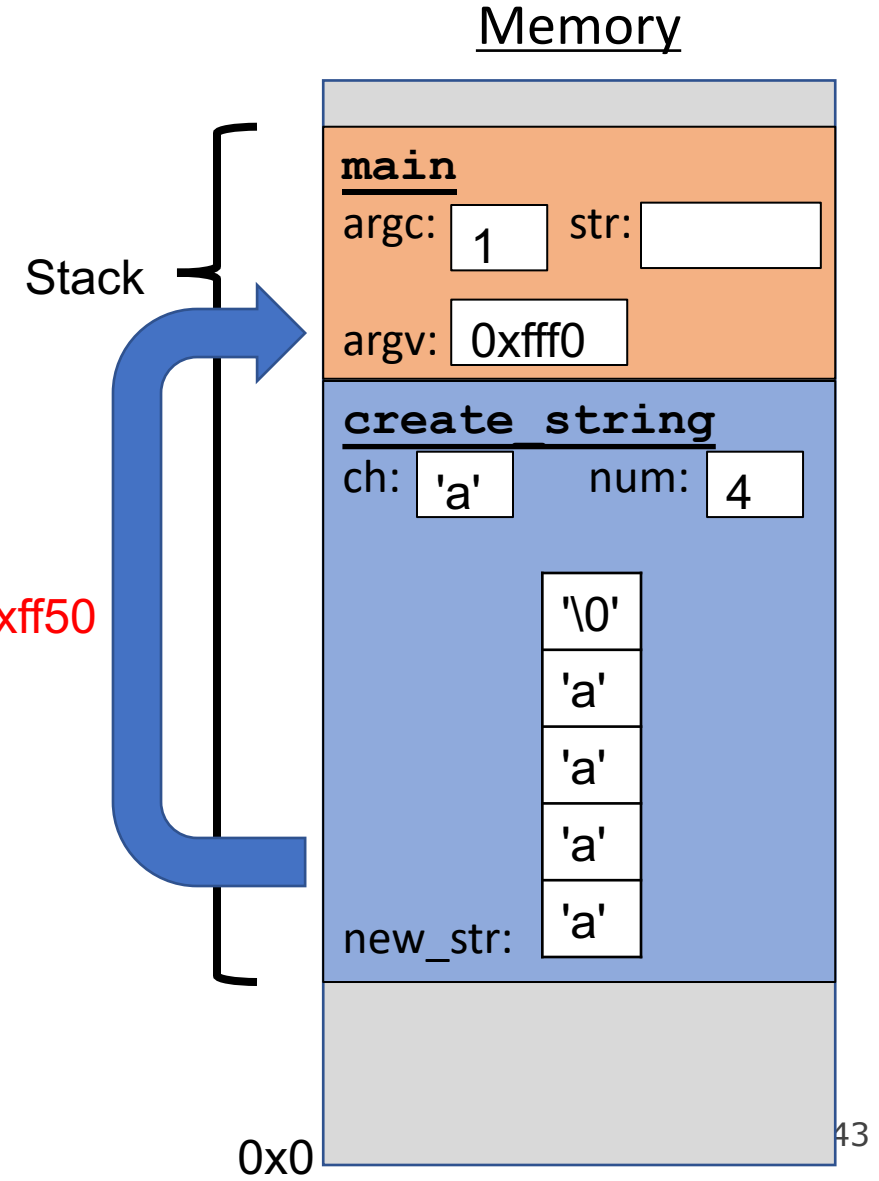
Memory

Stack

main

argc: 1   str: **0xff50**

argv: 0xfff0

0x0

45

```c
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;
}
```
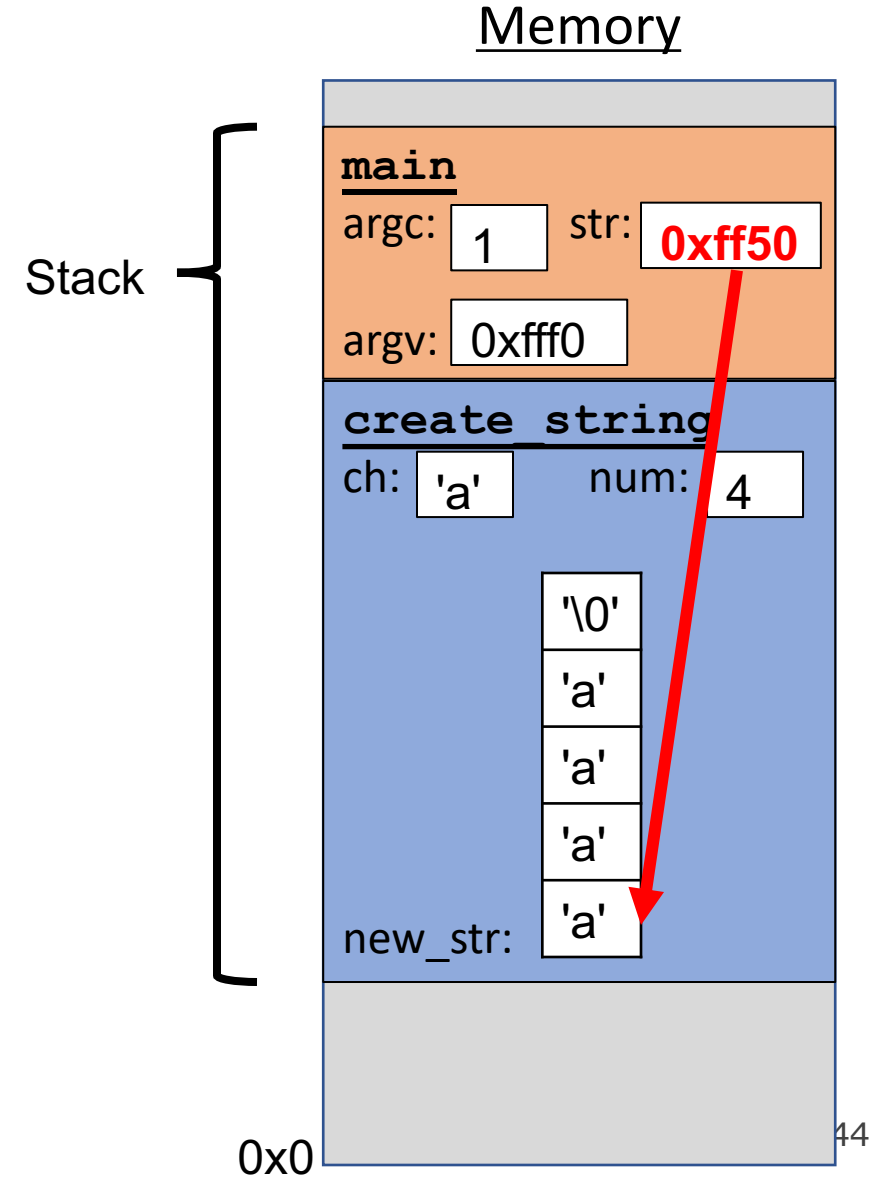
Memory

Stack

main

argc: 1   str: **0xff50**

argv: 0xfff0

Problem: local variables go away when a function finishes. These characters will thus no longer exist, and the address will be for unknown memory!
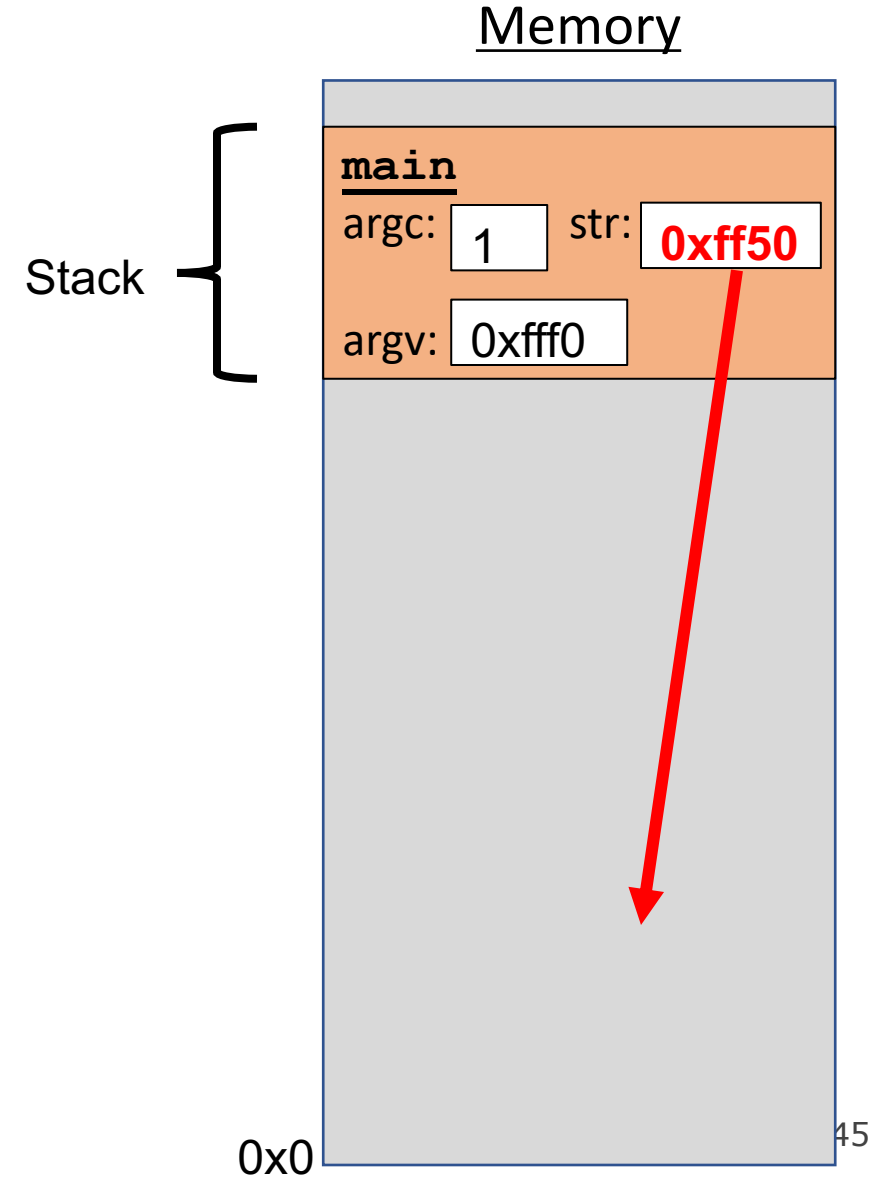
0x0

The Stack

```c
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;
}
```

Memory

Stack

main

argc: 1   str: **0xff50**

argv: 0xfff0

0x0

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

Lecture Plan

- The Stack
- **The Heap and Dynamic Memory**
- **Practice:** Pig Latin
- `realloc`
- **Practice:** Pig Latin Part 2
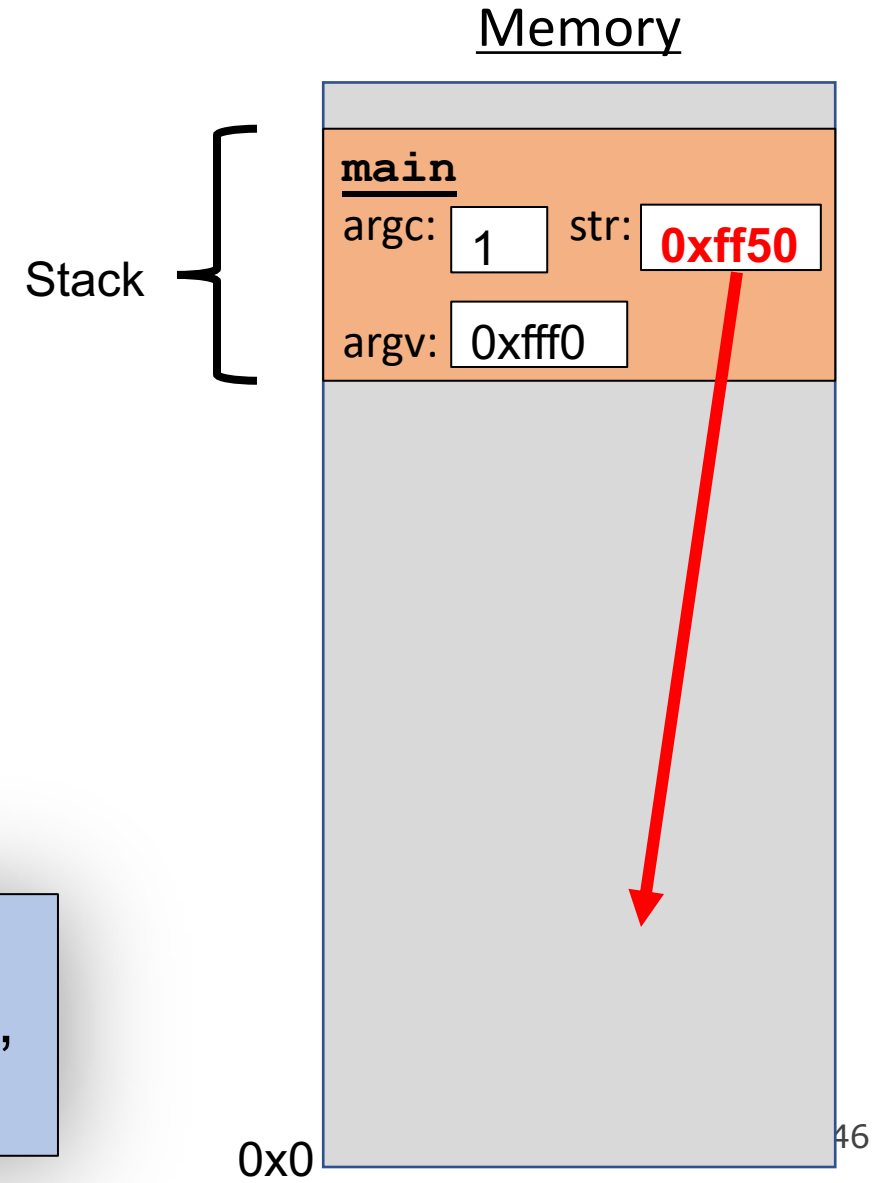
The Heap

```c
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;
}
```

Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

'\0'
'a'
'a'
'a'
new_str: 'a'

Us: hey C, is there a way to make this variable in memory that isn't automatically cleaned up?
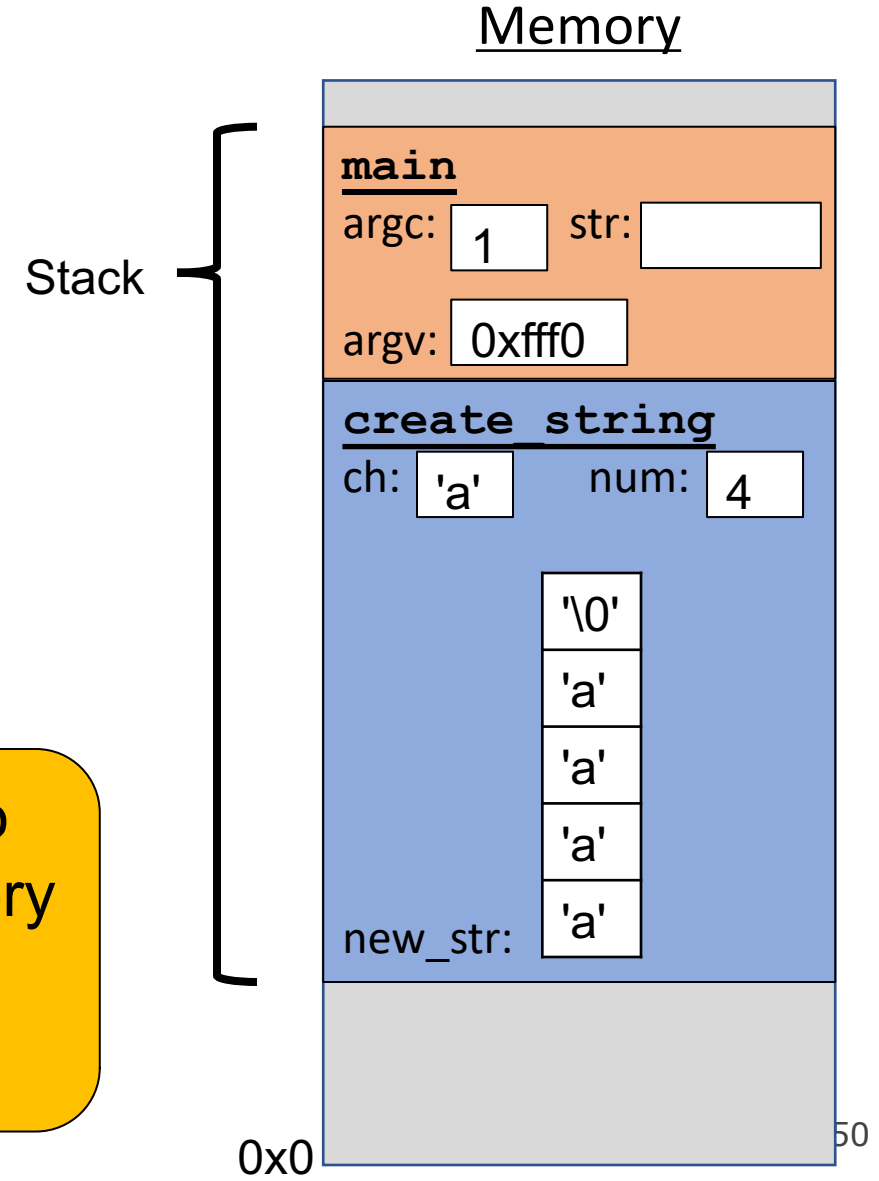
0x0

The Heap

```c
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;
}
```
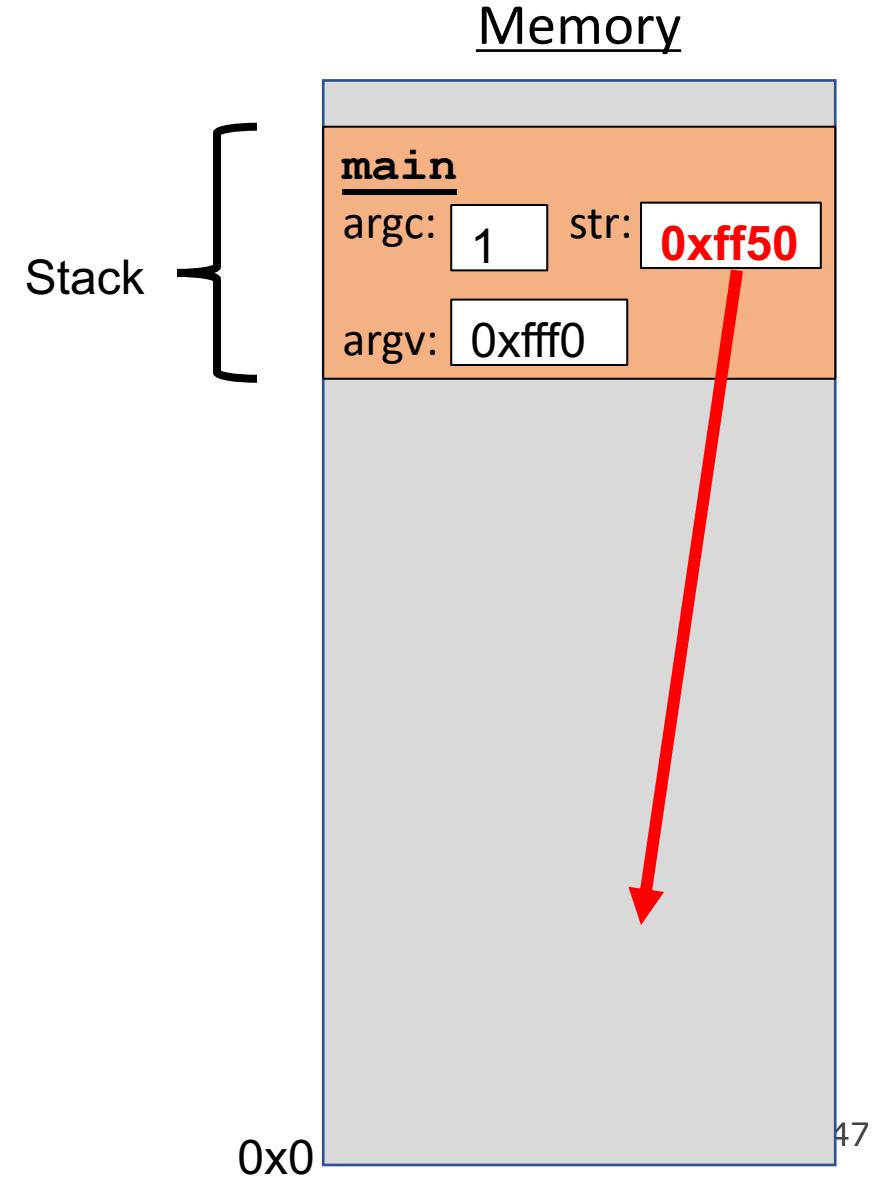
Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

new_str:
'\0'
'a'
'a'
'a'
'a'

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

0x0
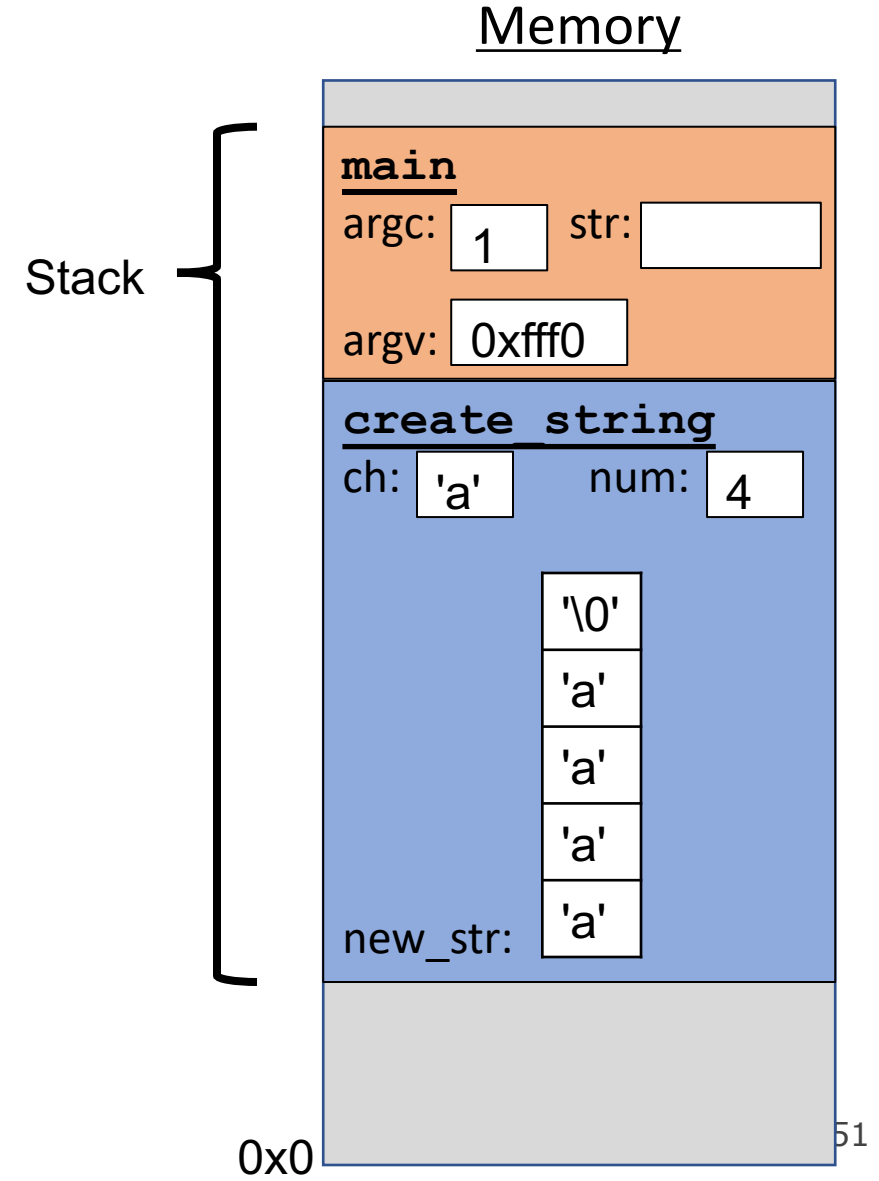
The Heap

- 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**.

0x7fffffffff000 — Stack — 8MB reserved

0x7ffff7ffe000 — Shared library text/data — Sized for library

Heap — Grows on demand

0x602010

0x600000 — Global data — Sized for executable

0x400000 — Text (machine code)

Low addresses deliberately unmapped

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;
}
```
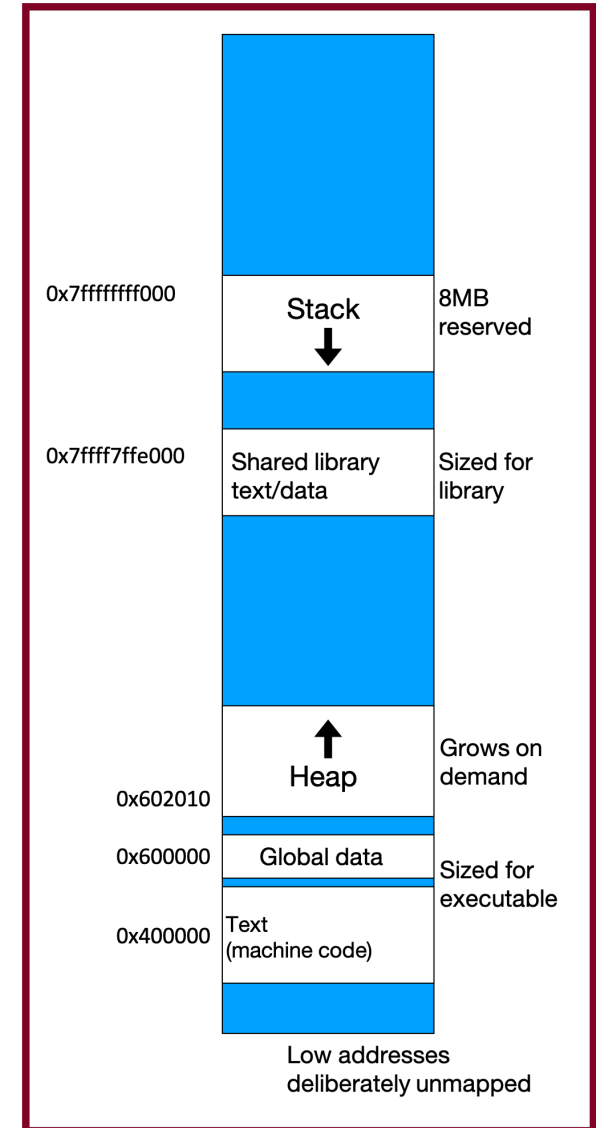
Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

new_str: 0xed0

Heap

'\0'
'a'
'a'
'a'
'a'

0x0

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;
}
```

Memory

Stack

Returns e.g. 0xed0

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str: **0xed0**

Heap

'\0'

'a'

'a'

'a'

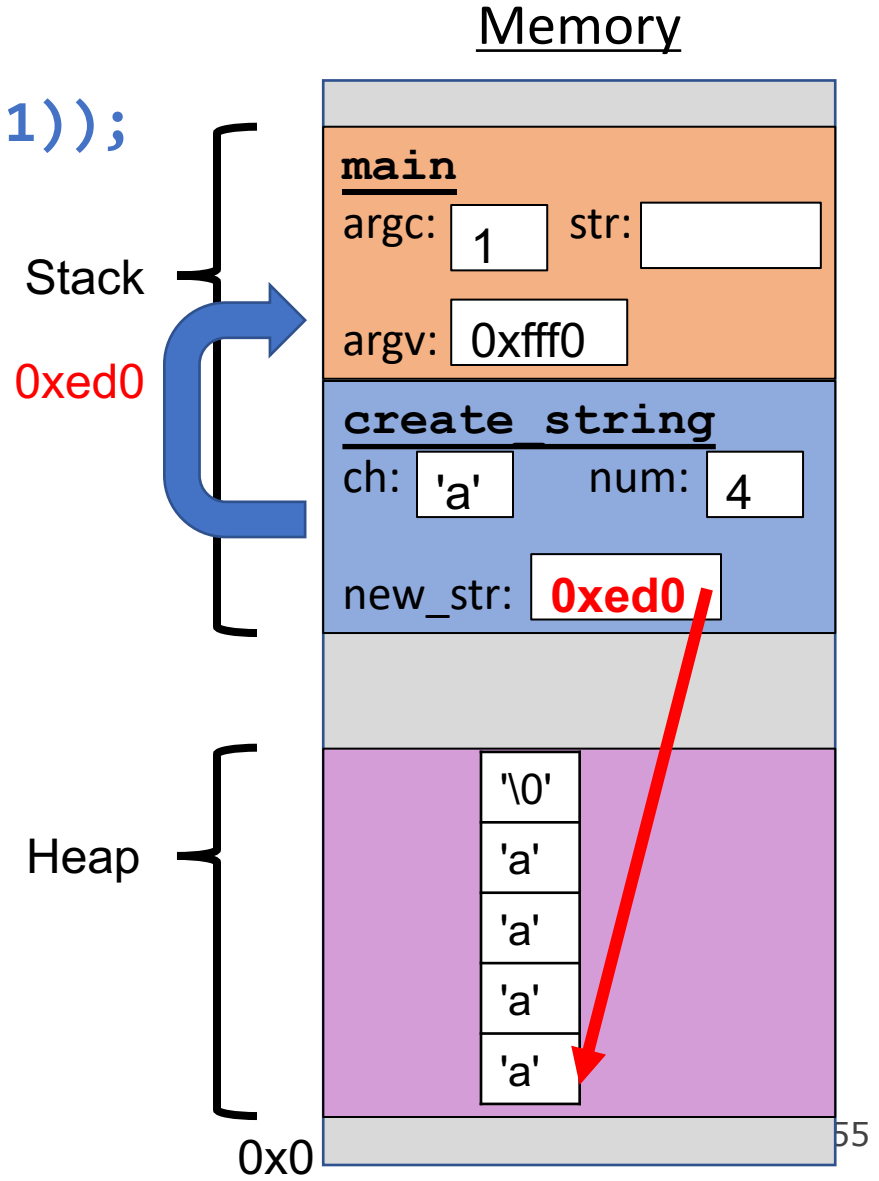'a'

0x0
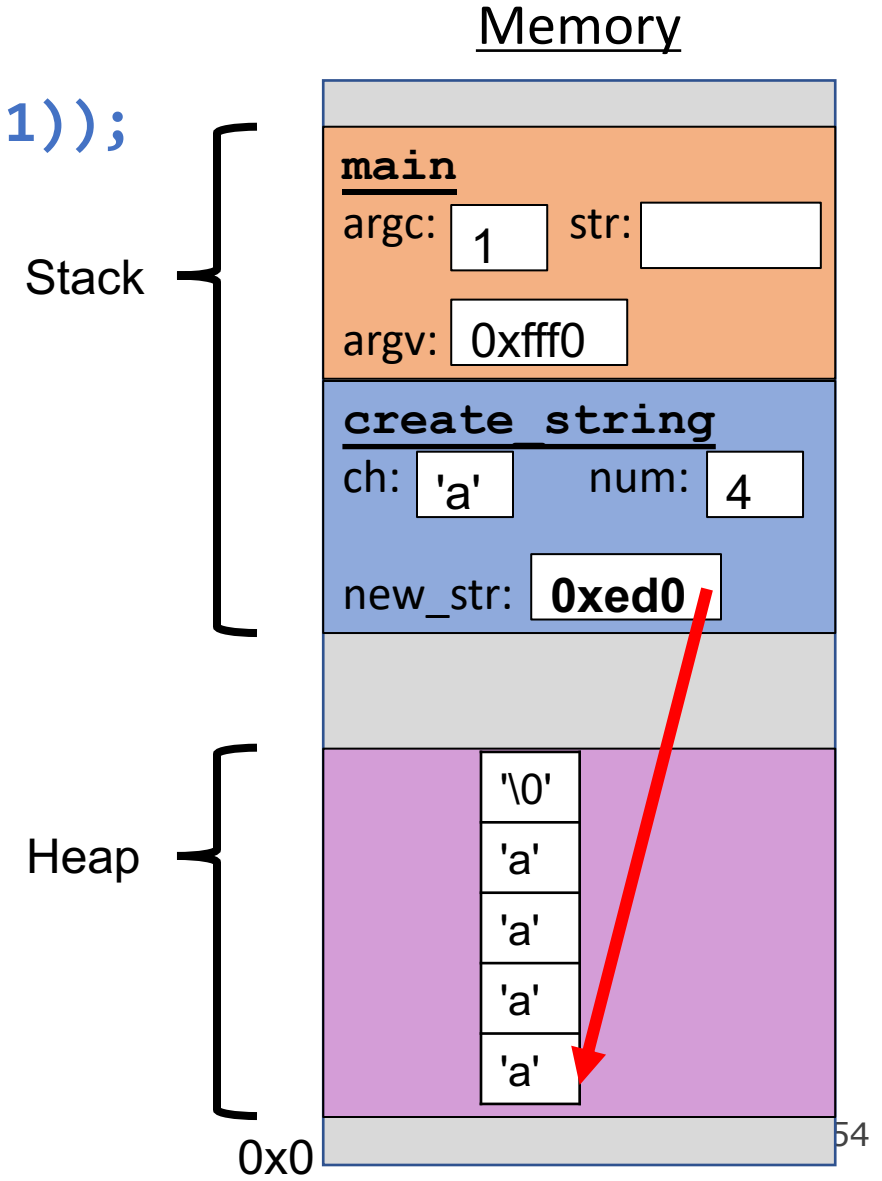
The Heap

```
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

Memory

Stack

main
argc: 1   str: 0xed0
argv: 0xfff0

create_string
ch: 'a'   num: 4
new_str: 0xed0

Heap

'\0'
'a'
'a'
'a'
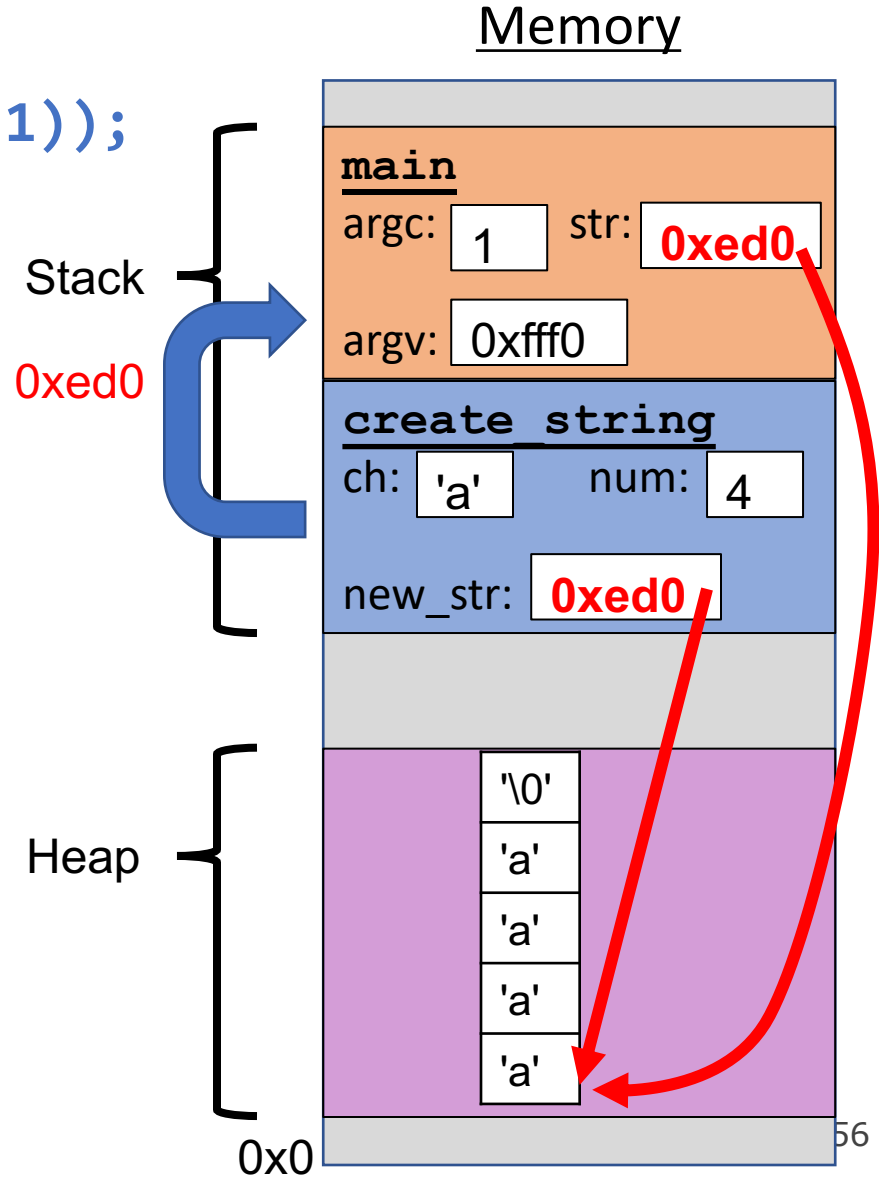'a'

0x0

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;
}
```


Memory

Stack

main
argc: 1   str: 0xed0
argv: 0xfff0

Heap

'\0'
'a'
'a'
'a'
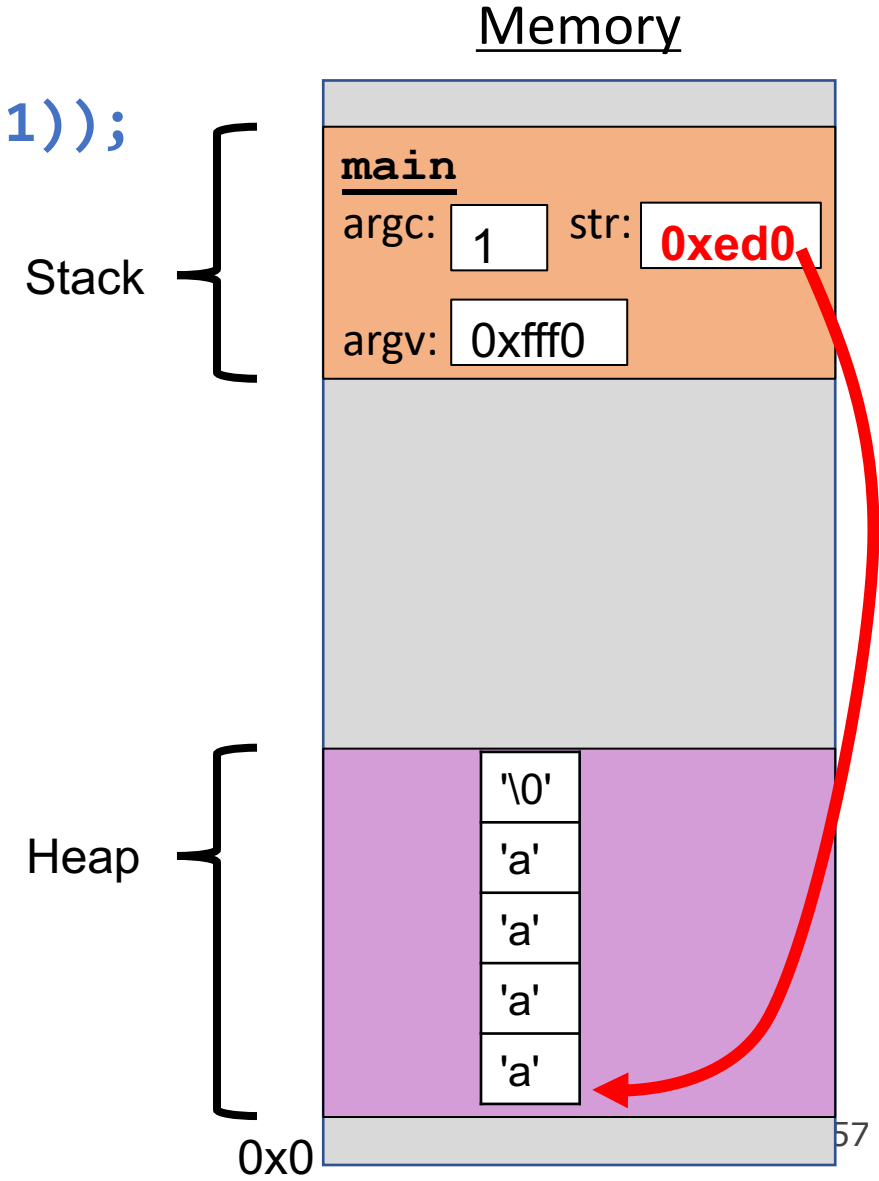'a'

0x0

The Heap

```c
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;
}
```

Memory

Stack

main

argc: 1   str: **0xed0**

argv: 0xfff0

Heap

'\0'

'a'

'a'

'a'

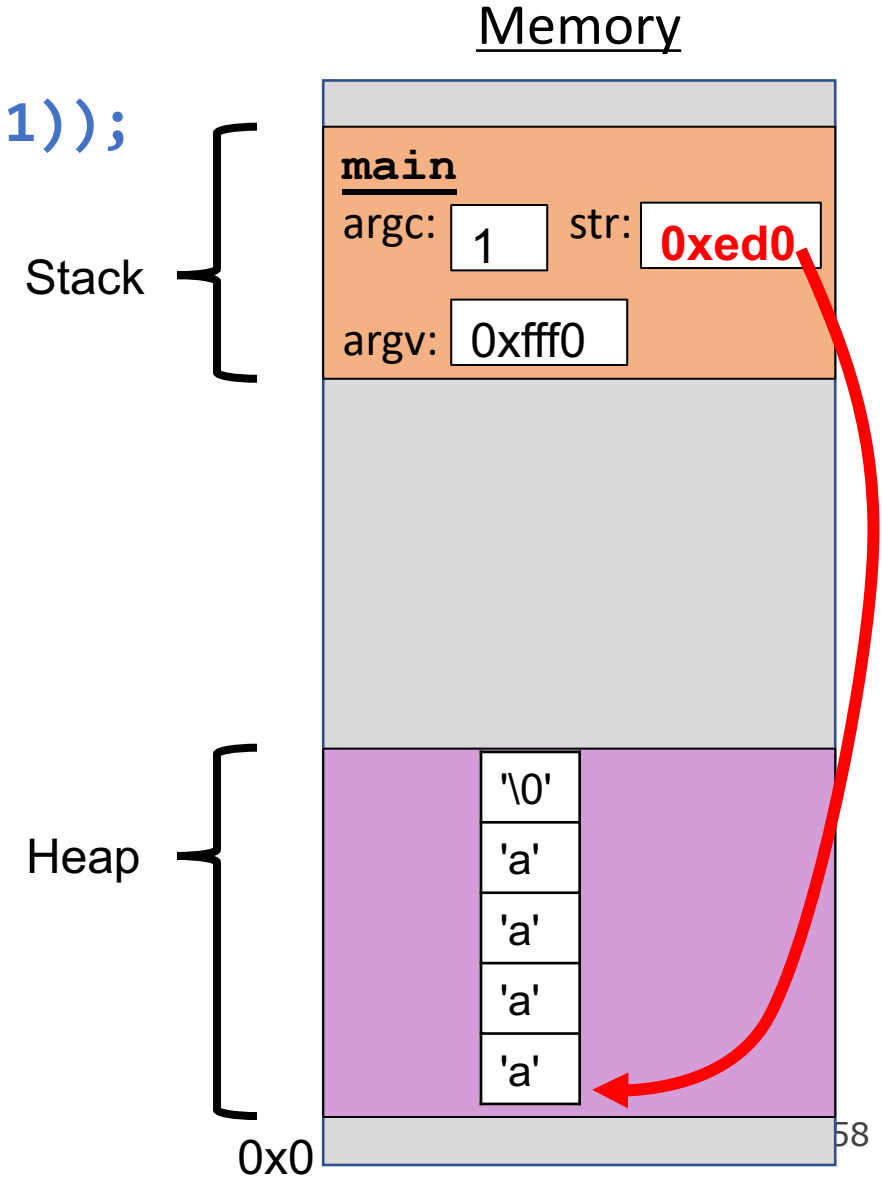'a'

0x0

```c
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;
}
```
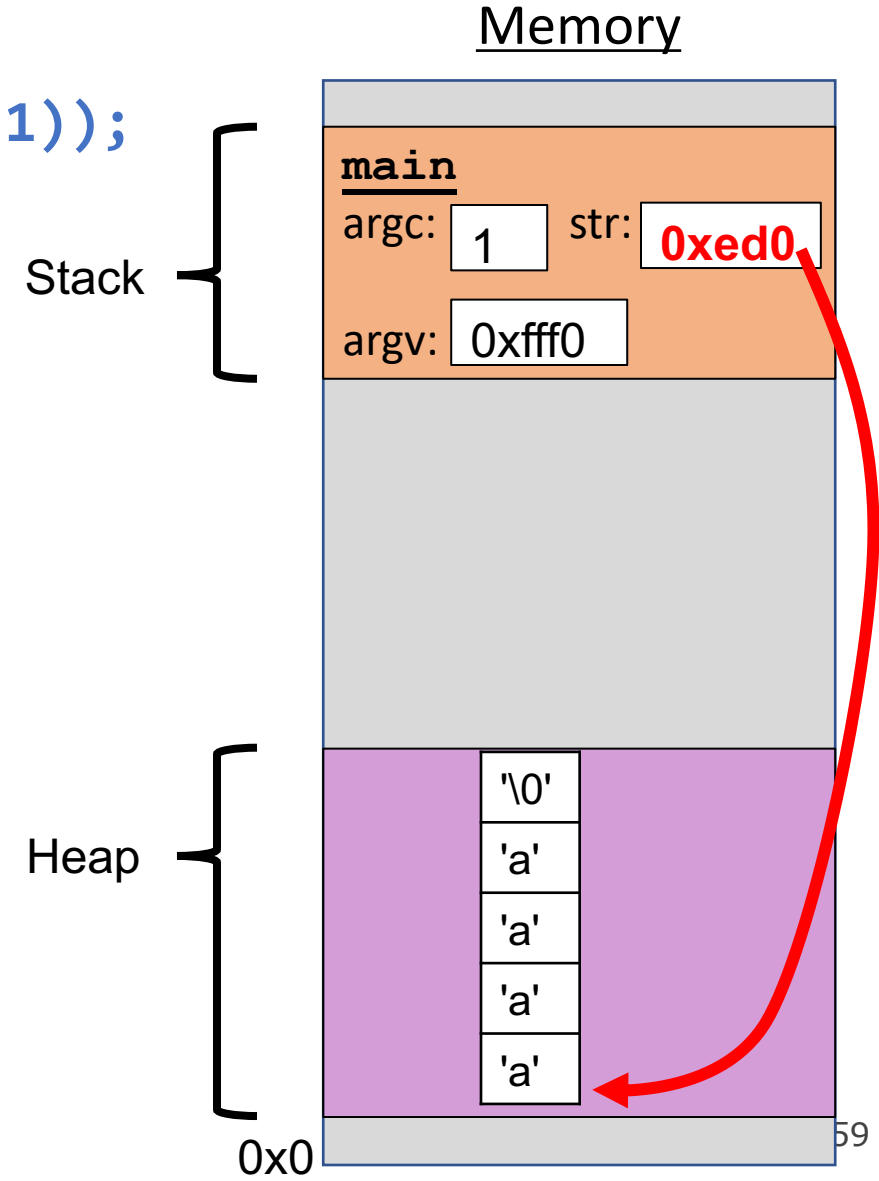
Memory

main

argc: 1   str: 0xed0

argv: 0xfff0

Stack

Heap

'\0'
'a'
'a'
'a'
'a'

0x0

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

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

```c
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
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!

```
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';
```

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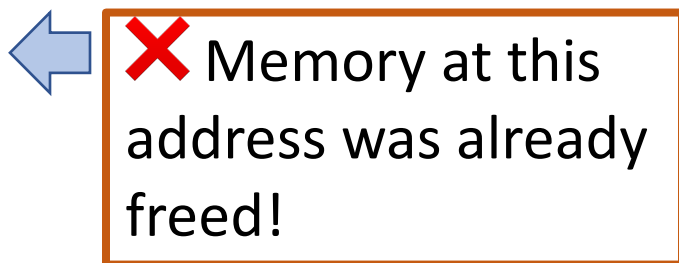
Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

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 = bytes;
…
free(bytes);   ⬅ ✅
…
free(ptr);     ⬅ ❌ Memory at this
                    address was already
                    freed!
```

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);    ⬅ ✅
…
free(ptr + 1);  ⬅ ❌
```

66

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!

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- **Practice: Pig Latin**
- `realloc`
- **Practice:** Pig Latin Part 2

Demo: Pig Latin

`pig_latin.c`

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- **realloc**
- **Practice:** Pig Latin Part 2

```
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.

- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.

- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

```c
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

realloc

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

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc`—the previous (smaller) one was already reclaimed by `realloc`.

```c
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Request memory by size (`malloc`)

- Receive room key to first of connecting rooms

Need more room? (`realloc`)

- Extend into connecting room if available
- If not, trade for new digs, employee moves your stuff for you

Check out when done (`free`)

- You remember your room number though

Errors! What happens if you...

- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes...
- Return to room after checkout?

Lecture Plan

- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- `realloc`
- **Practice: Pig Latin Part 2**

Demo: Pig Latin Part 2

pig_latin.c

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

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

🤔

Heap allocation interface: A summary

```c
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- NULL on failure, so check with `assert`

- Memory is contiguous; it is not recycled unless you call free

- `realloc` preserves existing data

- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)

- If you use after `free`, or if `free` is called twice on a location.

- If you `realloc/free` non-heap address

Engineering principles: stack vs heap

Stack ("local variables")   **Heap** (dynamic memory)

- **Fast**
Fast to allocate/deallocate; okay to oversize

- **Convenient**.
Automatic allocation/ deallocation;
declare/initialize in one step

- **Reasonable type safety**
Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope
dictated by control flow in/out of functions

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize

- **Convenient**.
  Automatic allocation/ deallocation;
  declare/initialize in one step

- **Reasonable type safety**
  Thanks to the compiler

- ⚠️ **Not especially plentiful**
  Total stack size fixed, default 8MB

- ⚠️ **Somewhat inflexible**
  Cannot add/resize at runtime, scope
  dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful**.
  Can provide more memory on demand!

- **Very flexible.**
  Runtime decisions about how much/when to
  allocate, can resize easily with realloc

- **Scope under programmer control**
  Can precisely determine lifetime

- ⚠️ **Lots of opportunity for error**
  Low type safety, forget to allocate/free
  before done, allocate wrong size, etc.,
  Memory leaks (much less critical)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.

- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Recap

- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- `realloc`
- **Practice:** Pig Latin Part 2

Next time: C Generics

Question Break

Post on Ed with any questions you have for today's lecture!

Extra Practice

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;
}
```

Freeing Memory

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

```
1    char *str = strdup("Hello");
2    assert(str != NULL);
3    char *ptr = str + 1;
4    for (int i = 0; i < 5; i++) {
5        int *num = malloc(sizeof(int));
6         assert(num != NULL);
7        *num = i;
8        printf("%s %d\n", ptr, *num);
9    }
10    printf("%s\n", str);
```

Freeing Memory

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

```
1     char *str = strdup("Hello");
2     assert(str != NULL);
3     char *ptr = str + 1;
4     for (int i = 0; i < 5; i++) {
5          int *num = malloc(sizeof(int));
6          assert(num != NULL);
7          *num = i;
8          printf("%s %d\n", ptr, *num);
9          free(num);
10    }
11  printf("%s\n", str);
12  free(str);
```