CS107 Spring 2019, Lecture 7

Stack and Heap

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

Based on slides created by Marty Stepp, Cynthia Lee, Chris Gregg, and others.

Plan For Today

- Miscellaneous Useful Topics
  - **const**
  - Structs
  - Ternary
- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- Announcements
- `Realloc`
- **Practice:** Pig Latin Part 2

Plan For Today

- **Miscellaneous Useful Topics**
  - **const**
  - **Structs**
  - **Ternary**
- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- Announcements
- `Realloc`
- **Practice:** Pig Latin Part 2

Const

- Use **const** to declare global constants in your program. This indicates the variable cannot change after being created.

```
const double PI = 3.1415;
const int DAYS_IN_WEEK = 7;

int main(int argc, char *argv[]) {
	…
	if (x == DAYS_IN_WEEK) {
		…
	}
	…
}
```

Const

- Use **const** with pointers to indicate that the data that is pointed to cannot change.

```
char str[] = "Hello";
const char *s = str;

// Cannot use s to change characters it points to
```

~~`s[0] = 'h';`~~

Const

Sometimes we use **const** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```
// This function promises to not change str’s characters
int countUppercase(const char *str) {
     int count = 0;
     for (int i = 0; i < strlen(str); i++) {
          if (isupper(str[i])) {
               count++;
          }
     }
     return count;
}
```

Const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify.

```
// This function promises to not change str’s characters
int countUppercase(const char *str) {
    char *strToModify = str;
    strToModify[0] = …
}
```

Const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify. **Think of const as part of the variable type**.

```
// This function promises to not change str’s characters
int countUppercase(const char *str) {
	const char *strToModify = str;
	~~strToModify[0] = …~~
}
```

Const

const can be confusing to interpret in some variable types.

```
// cannot modify this char
const char c = 'h';

// cannot modify chars pointed to by str
const char *str = …

// cannot modify chars pointed to by *strPtr
const char **strPtr = …
```

Structs

A *struct* is a way to define a new variable type that is a group of other variables.

```
struct date {           // declaring a struct type
    int month;
    int day;            // members of each date structure
};
…

struct date today;                     // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31};   // shorter initializer syntax
```

Structs

Wrap the struct definition in a **typedef** to avoid having to include the word **struct** every time you make a new variable of that type.

```
typedef struct date {
    int month;
    int day;
} date;
…

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```
void advance_day(date d) {
     d.day++;
}

int main(int argc, char *argv[]) {
     date my_date = {1, 28};
     advance_day(my_date);
     printf("%d", my_date.day); // 28
     return 0;
}
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct. **Use a pointer to modify a specific instance.**

```
void advance_day(date *d) {
     (*d).day++;
}

int main(int argc, char *argv[]) {
     date my_date = {1, 28};
     advance_day(&my_date);
     printf("%d", my_date.day); // 29
     return 0;
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
     d->day++;
}

int main(int argc, char *argv[]) {
     date my_date = {1, 28};
     advance_day(&my_date);
     printf("%d", my_date.day); // 29
     return 0;
}
```

Structs

C allows you to return structs from functions as well. It returns whatever is contained within the struct.

```
date create_new_years_date() {
     date d = {1, 1};
    return d;          // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
     date my_date = create_new_years_date();
     printf("%d", my_date.day); // 1
     return 0;
}
```

Structs

sizeof gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```
typedef struct date {
      int month;
      int day;
  } date;

int main(int argc, char *argv[]) {
      int size = sizeof(date);   // 8
      return 0;
}
```

Arrays of Structs

You can create arrays of structs just like any other variable type.

```
typedef struct my_struct {
	int x;
	char c;
} my_struct;

…

my_struct array_of_structs[5];
```

Arrays of Structs

To initialize an entry of the array using short syntax, you must add a cast to confirm the type to C.

```
typedef struct my_struct {
	int x;
	char c;
} my_struct;

…

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Arrays of Structs

You can also set each field individually.

```
typedef struct my_struct {
     int x;
     char c;
} my_struct;

…
my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```

Ternary Operator

The ternary operator is a shorthand for using if/else to evaluate to a value.

```
condition ? expressionIfTrue : expressionIfFalse
```

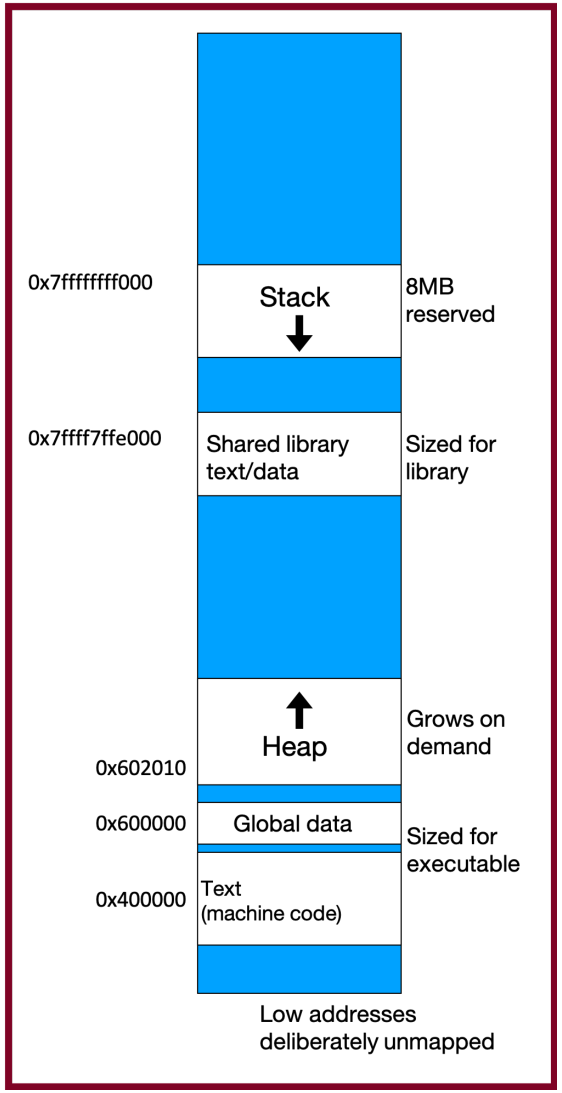
```
int x;
if (argc > 1) {
      x = 50;
} else {
      x = 0;
}

// equivalent to
int x = argc > 1 ? 50 : 0;
```

Plan For Today

- Miscellaneous Useful Topics
  - **const**
  - Structs
  - Ternary
- **The Stack**
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- Announcements
- `Realloc`
- **Practice:** Pig Latin Part 2

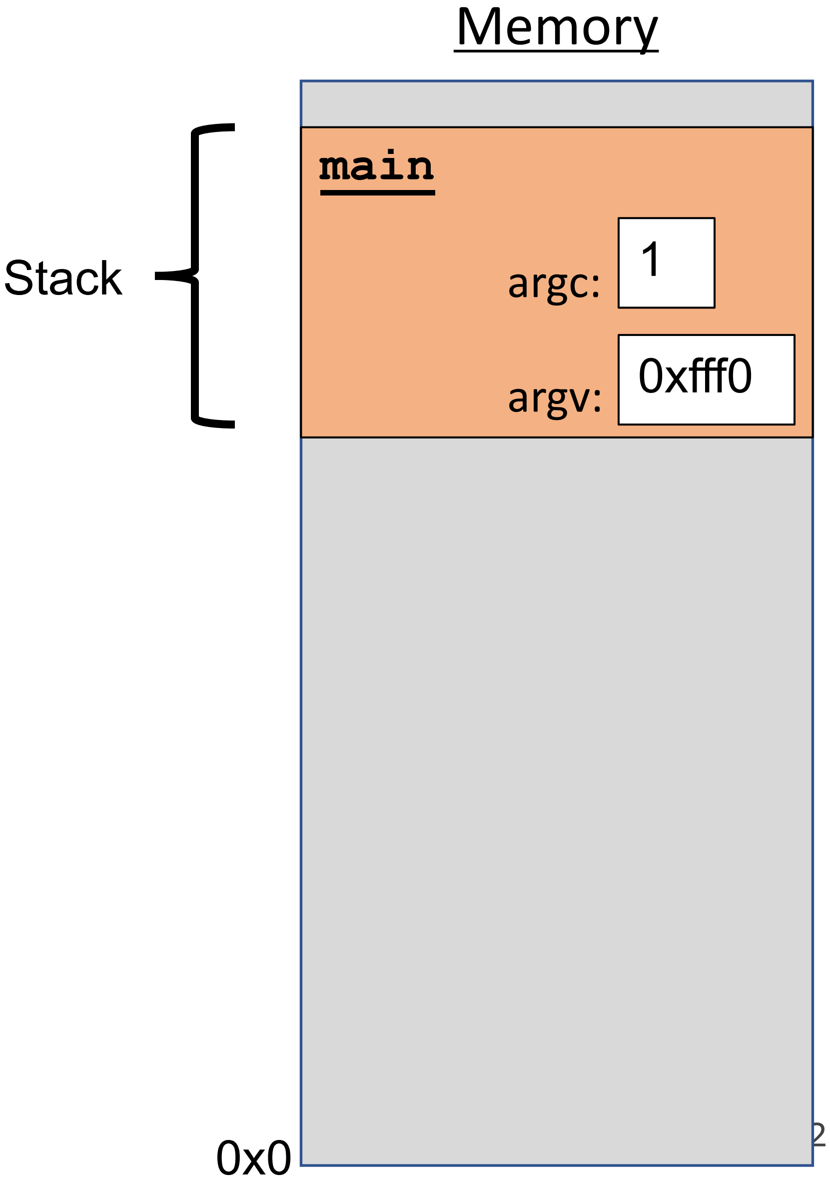
Memory Layout

- 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.



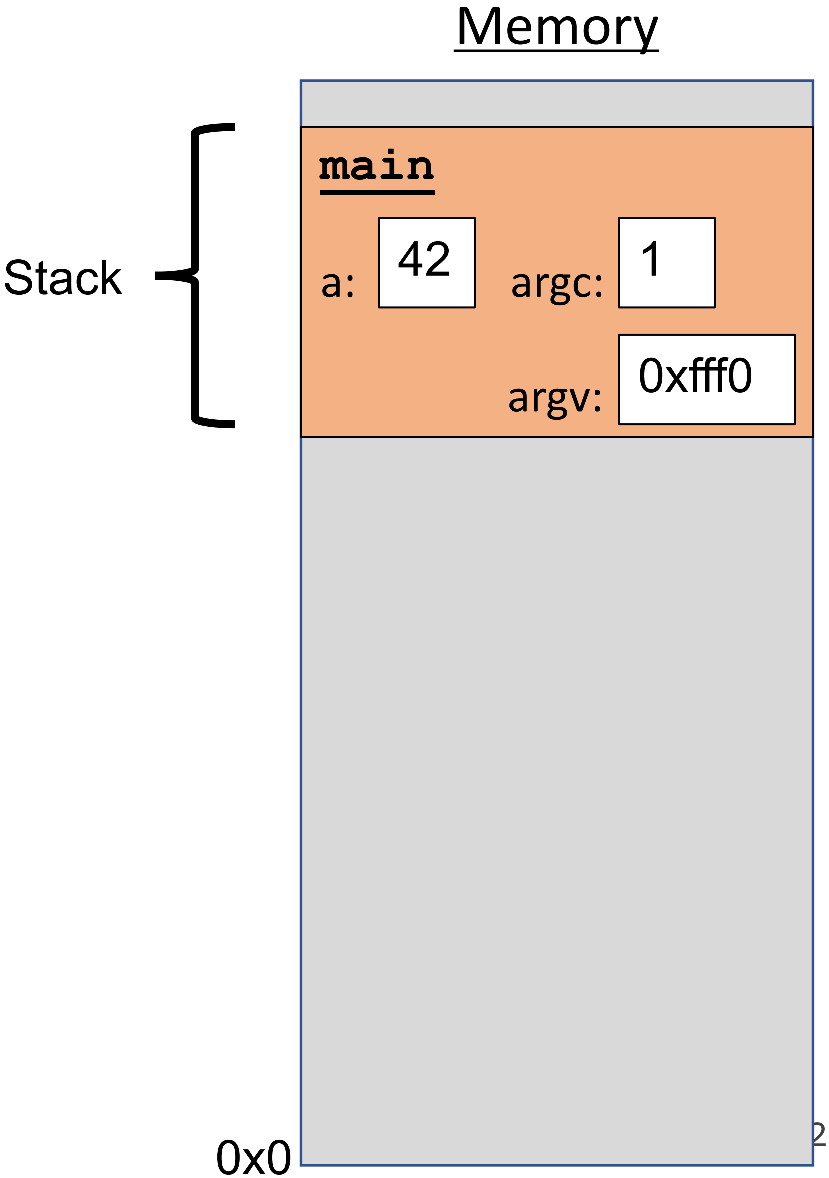
The Stack

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

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

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



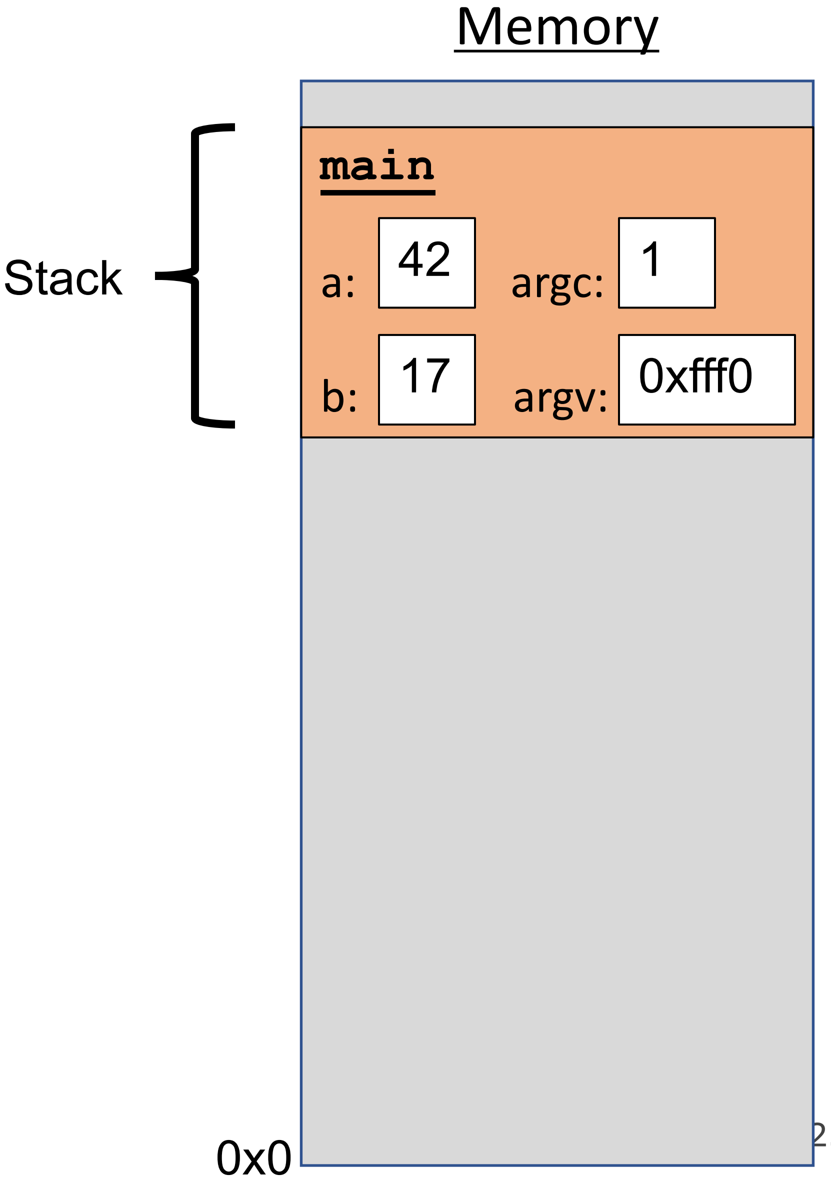
The Stack

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

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

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



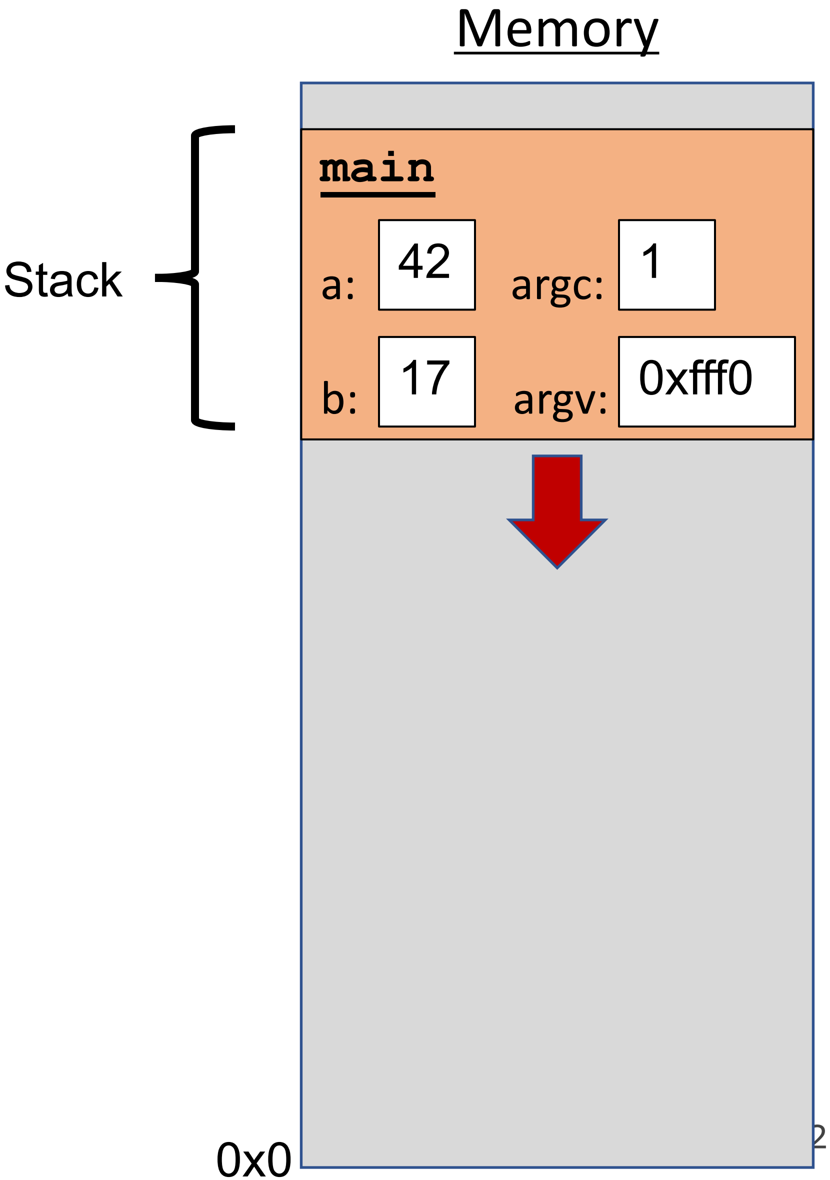
The Stack

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

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

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



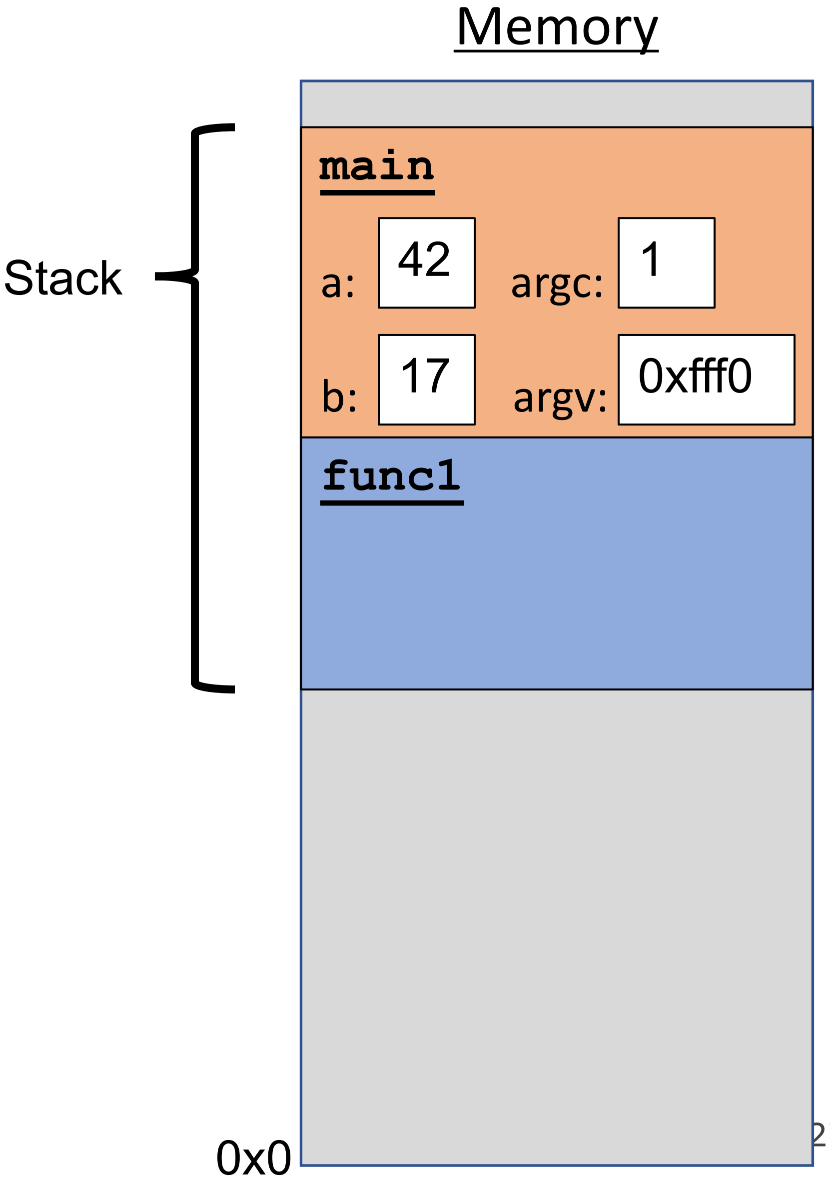
The Stack

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

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

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



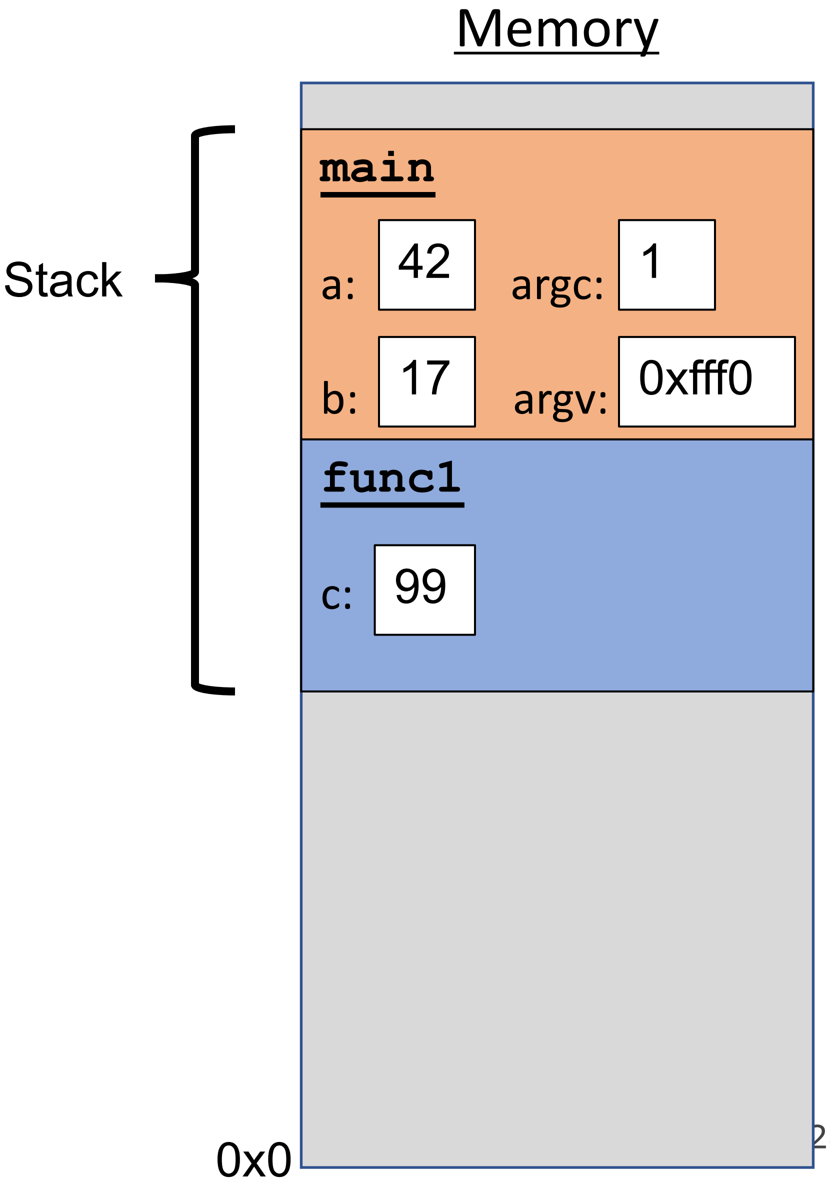
The Stack

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

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

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



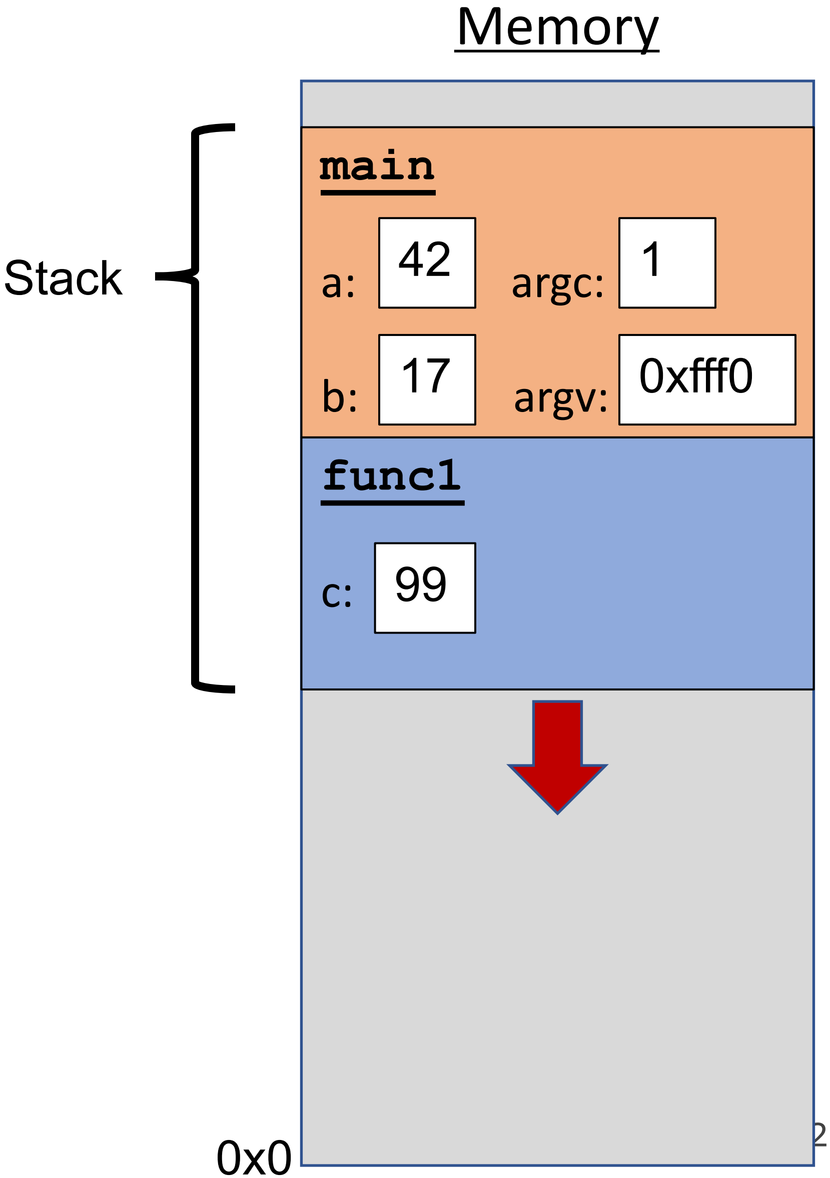
The Stack

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

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

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



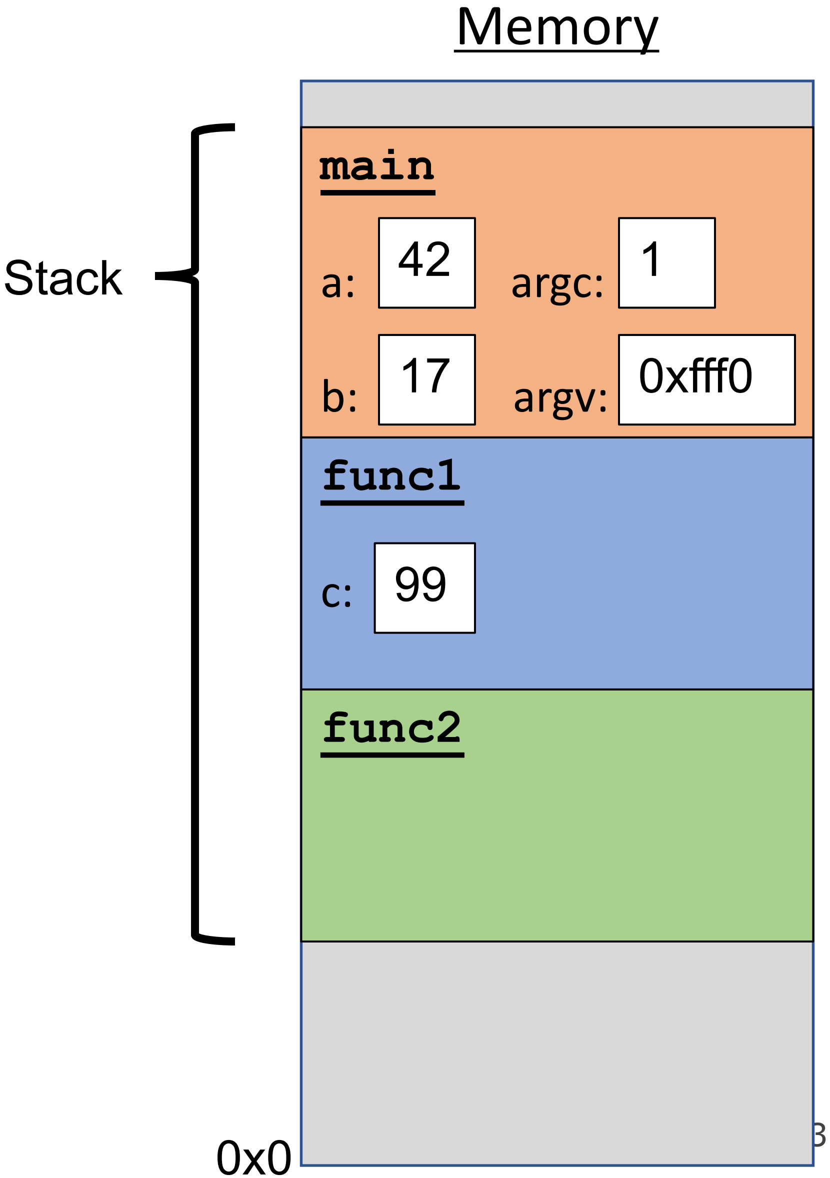
The Stack

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

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

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



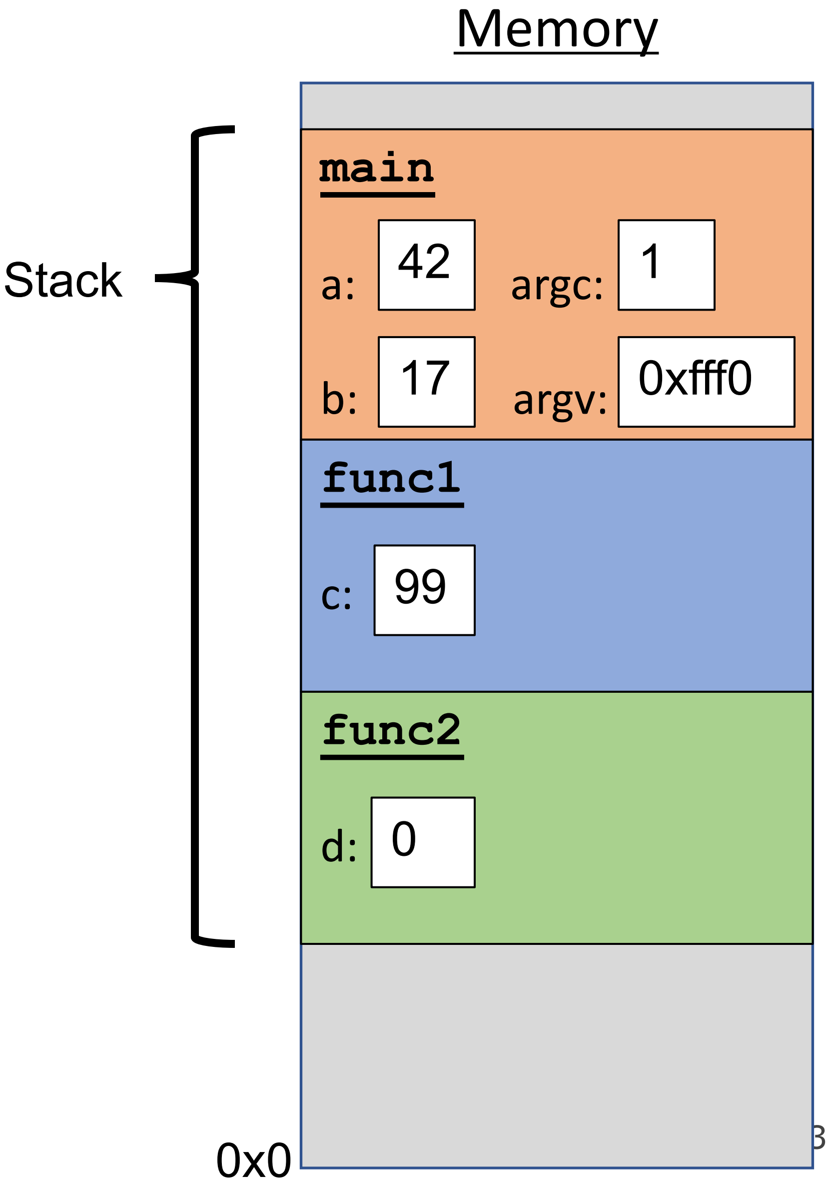
The Stack

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

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

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



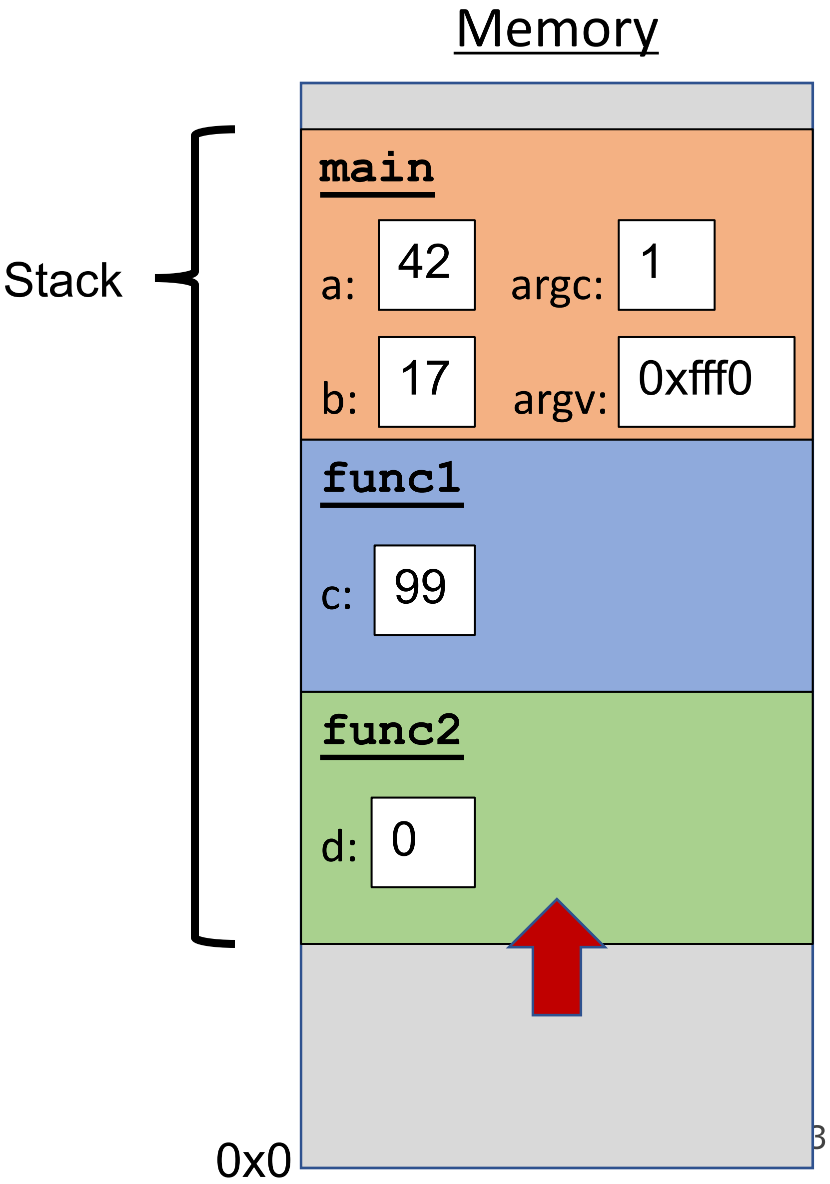
The Stack

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

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

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



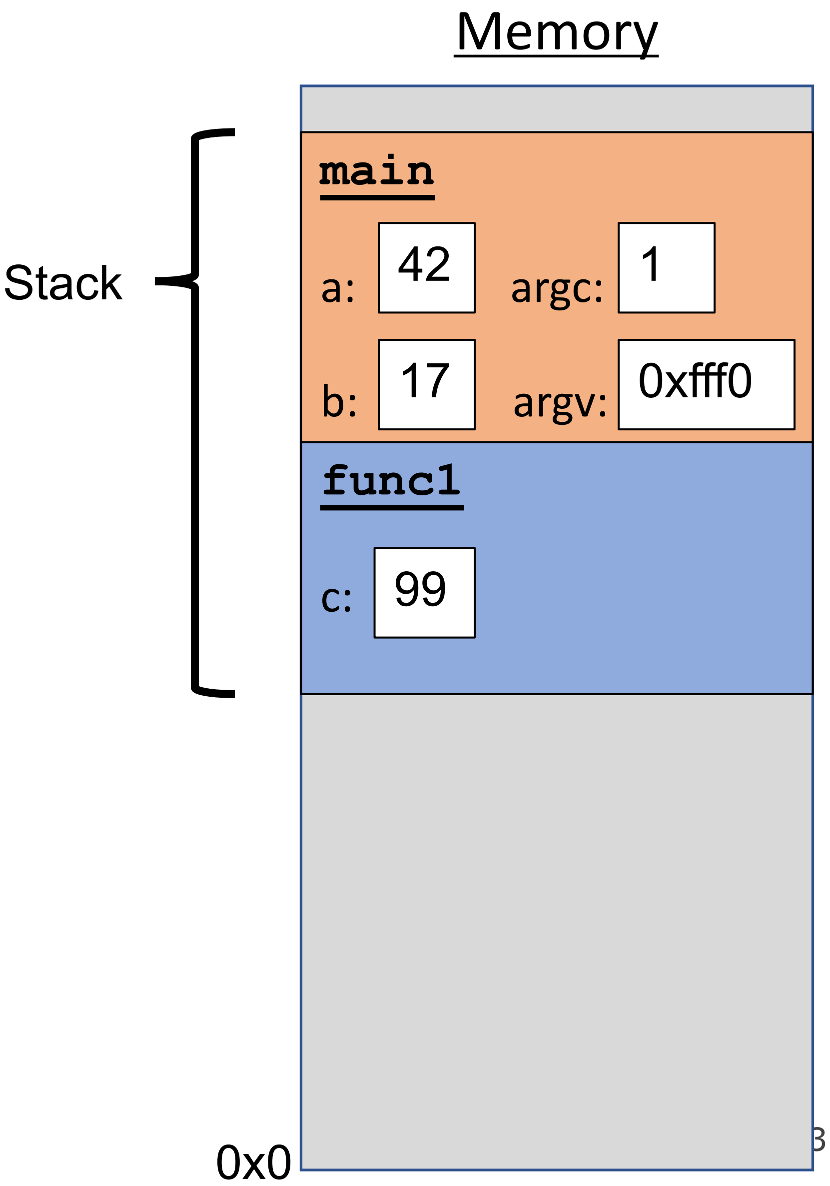
The Stack

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

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

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



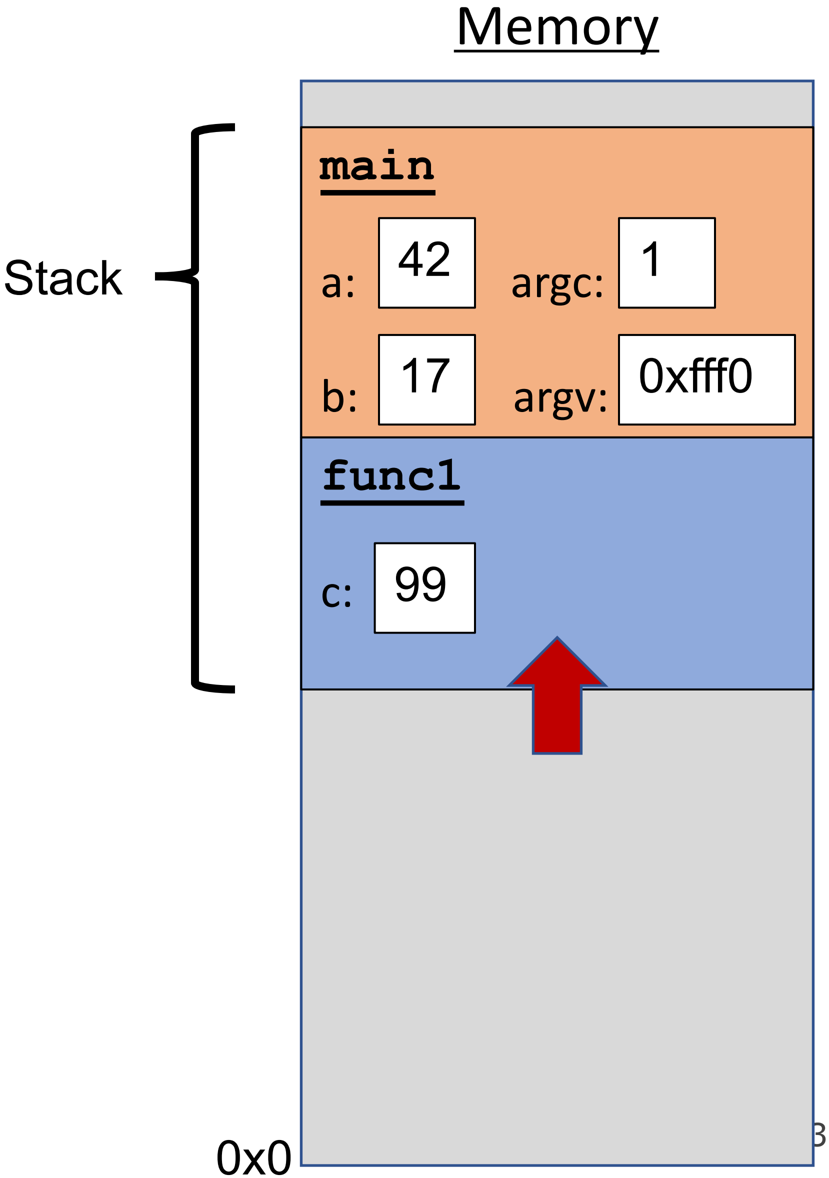
The Stack

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

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

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



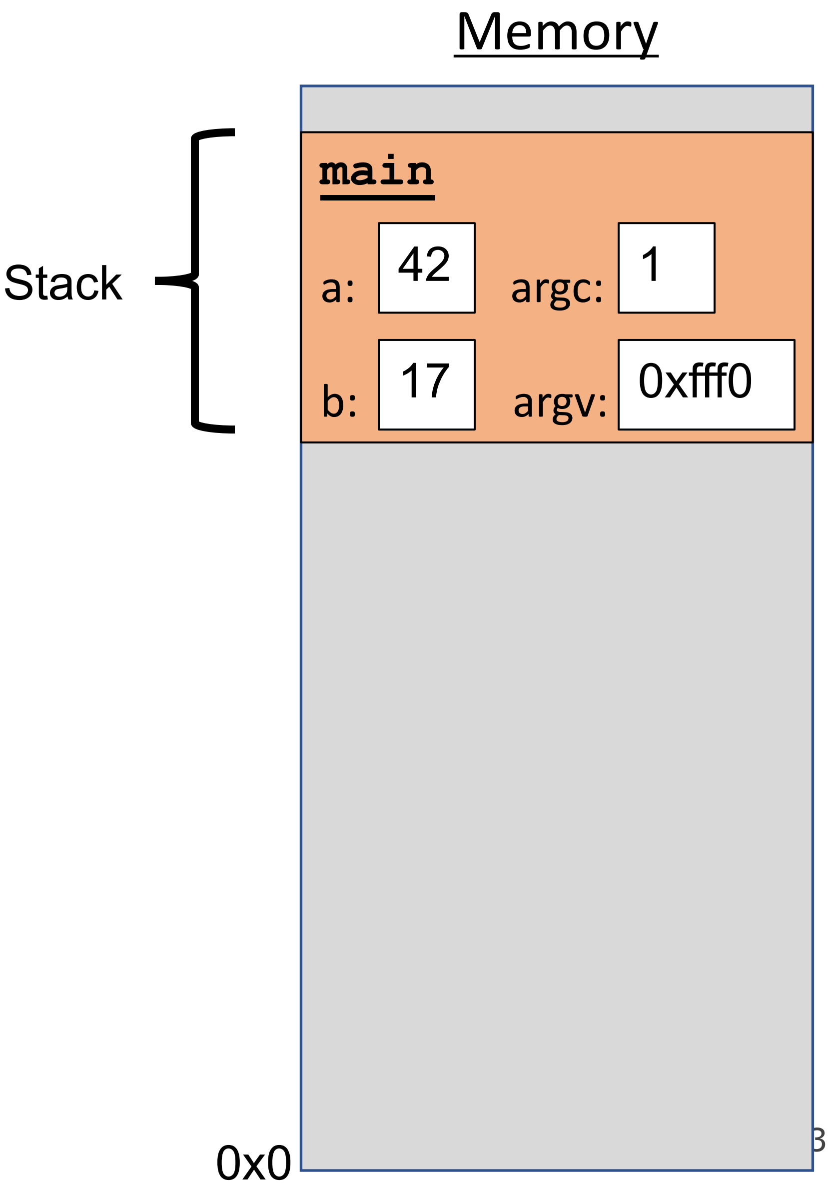
The Stack

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

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

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



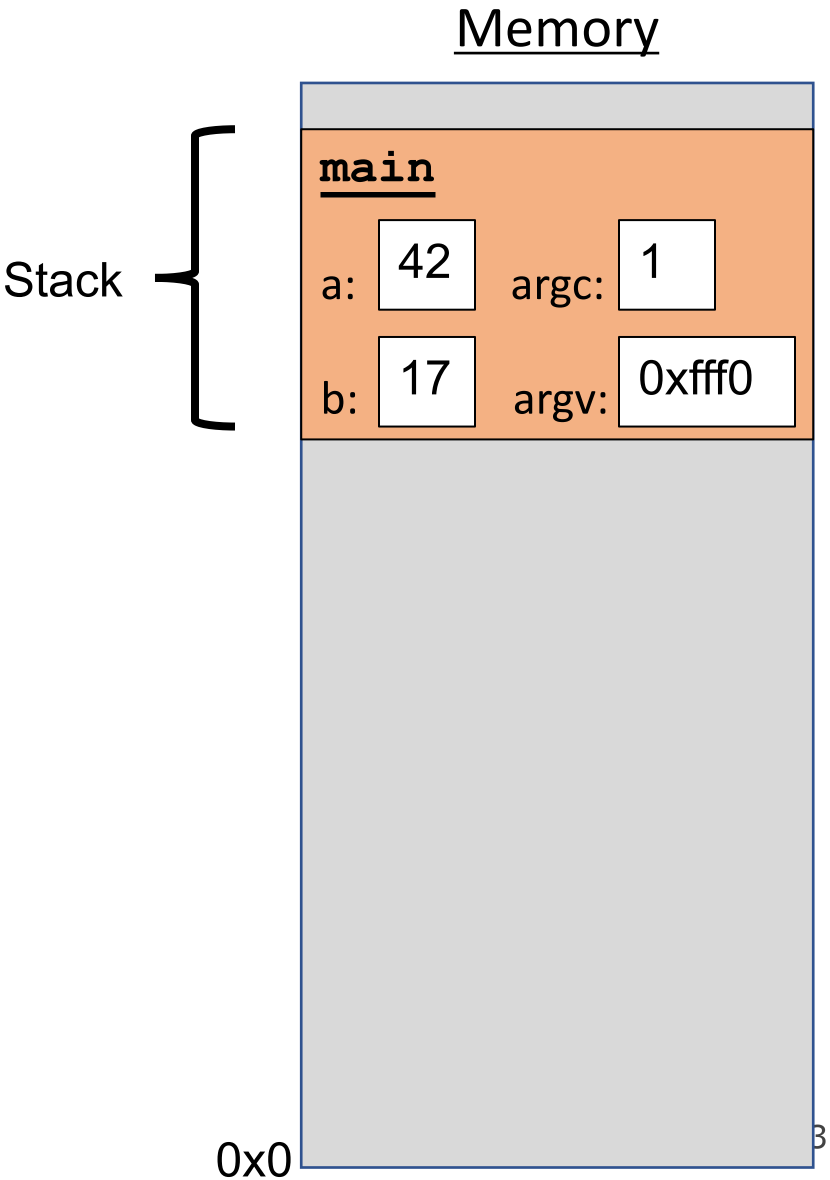
The Stack

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

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

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



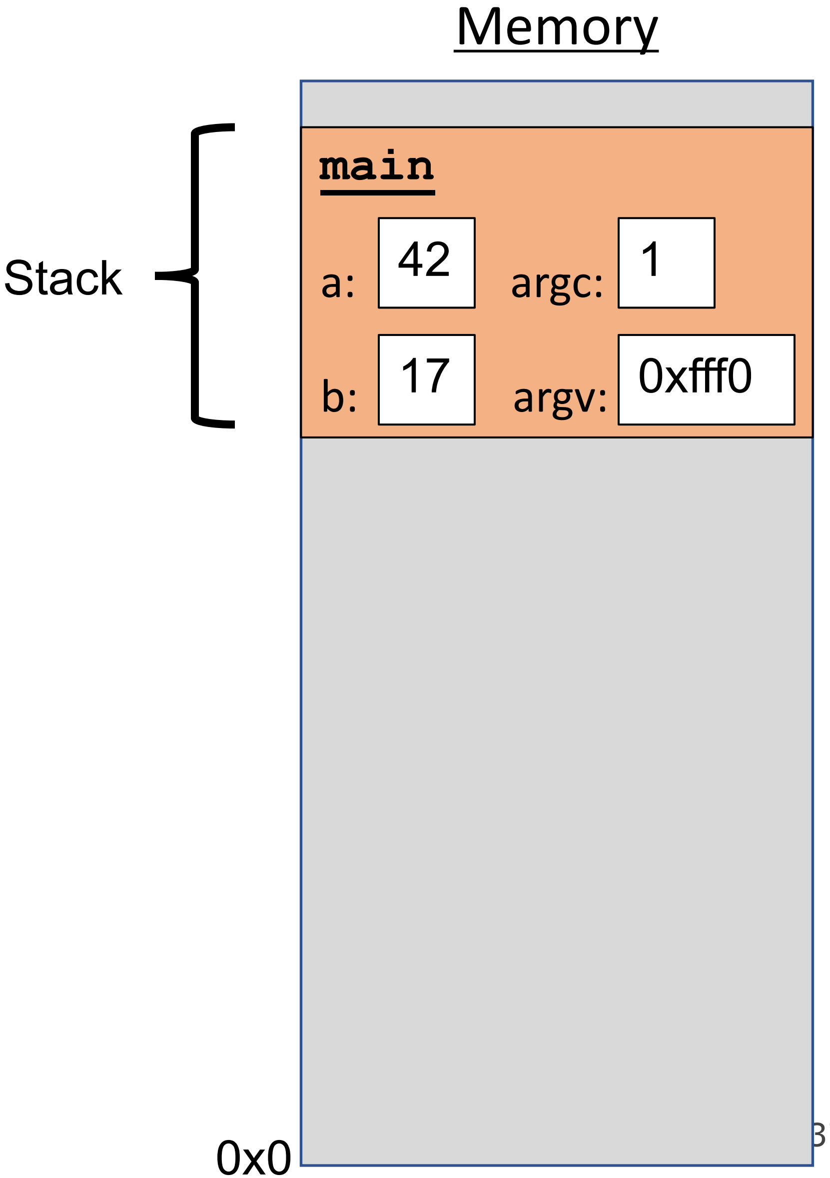
The Stack

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

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

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



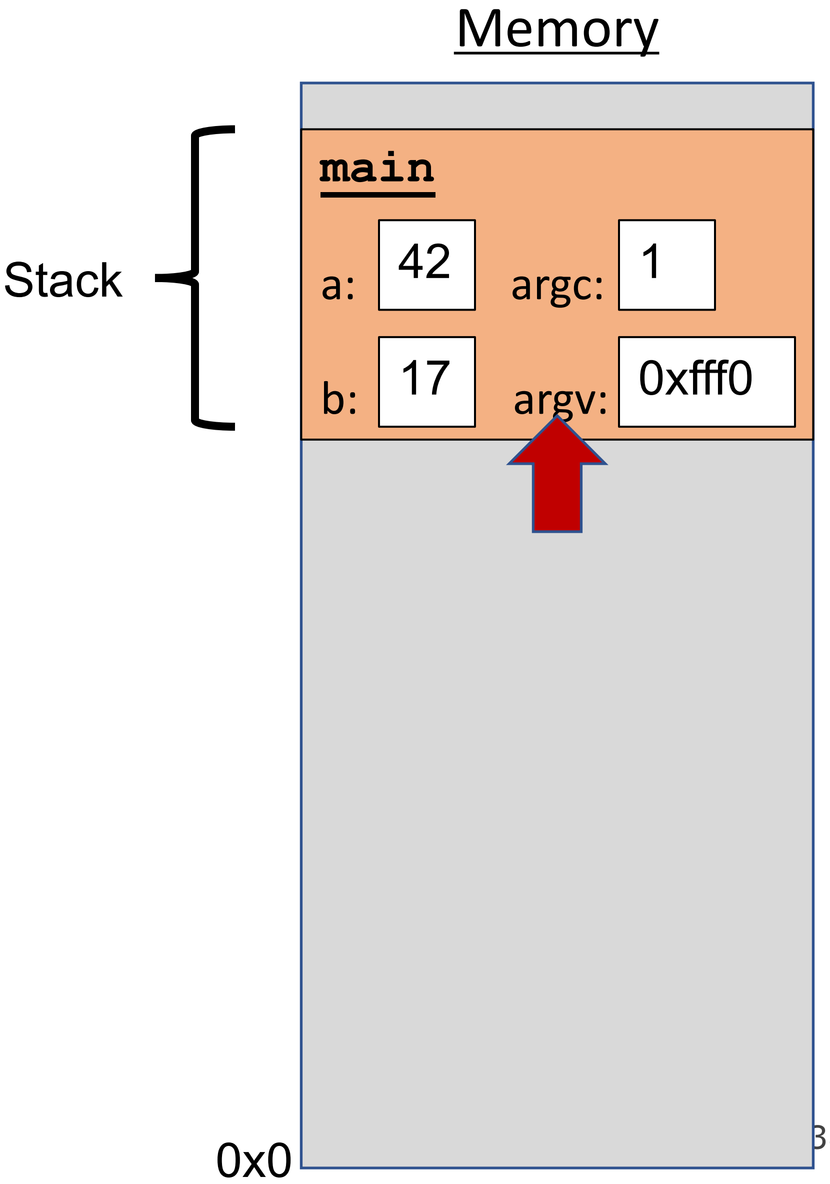
The Stack

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

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

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



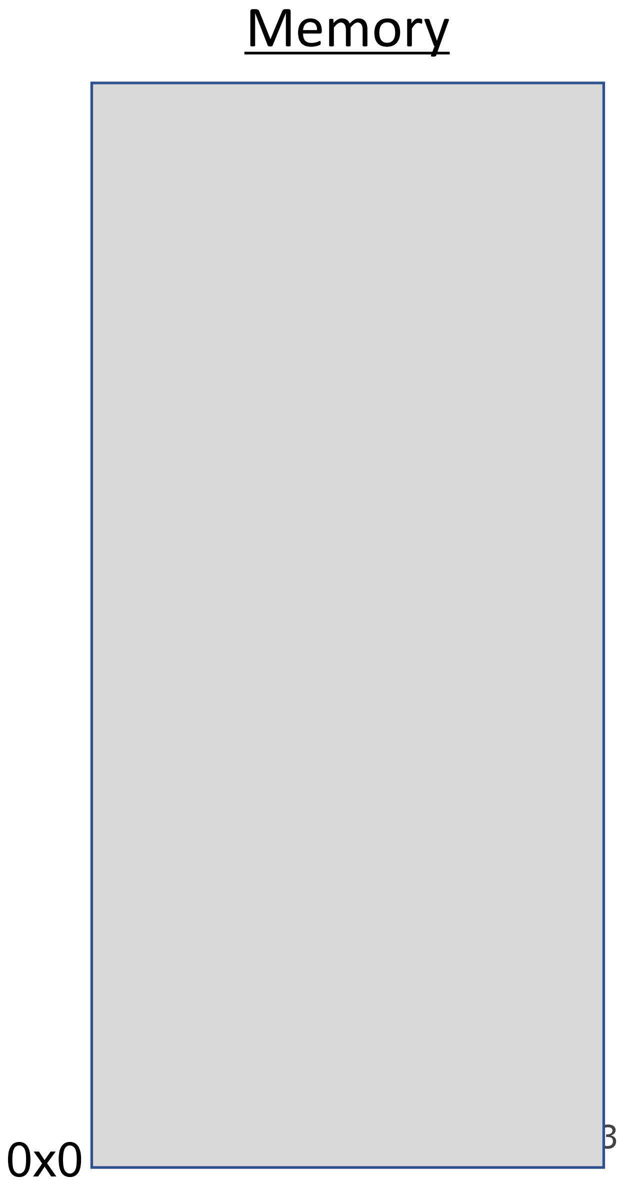
The Stack

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

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

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



The Stack

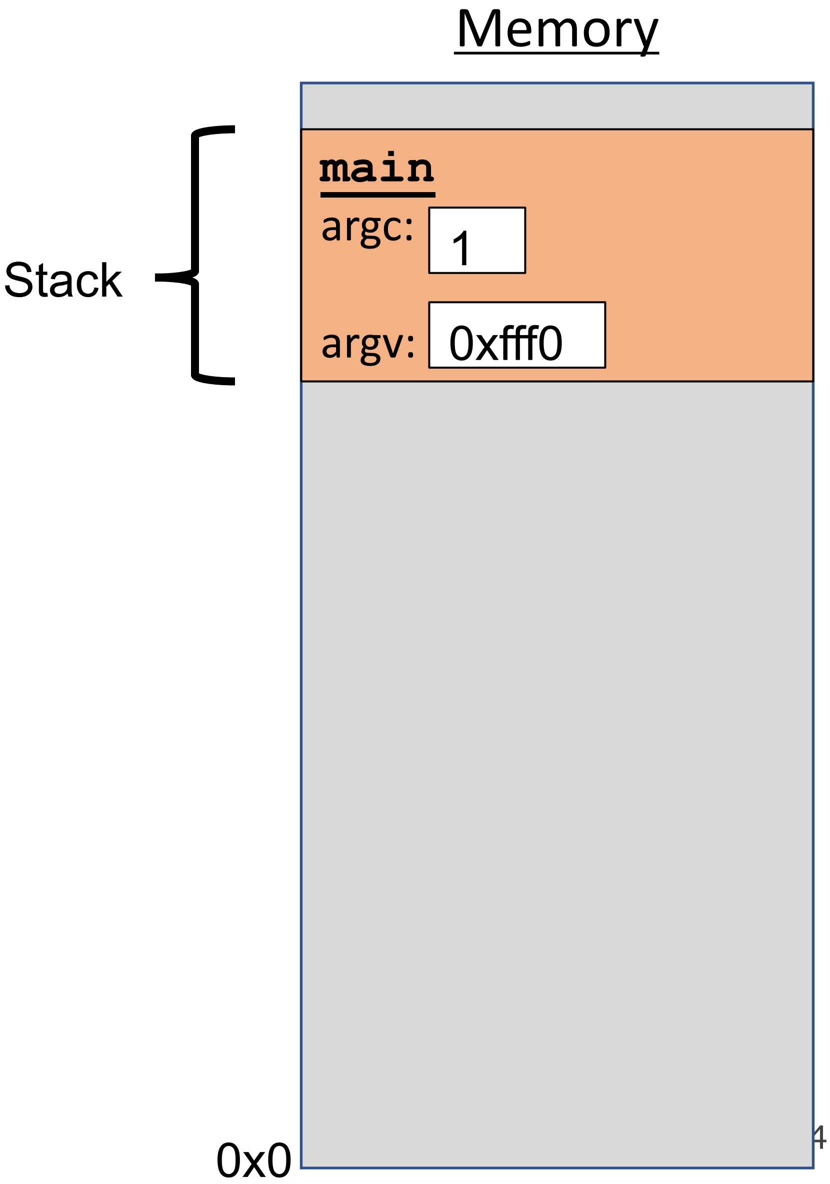
```
int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}

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

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



The Stack

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

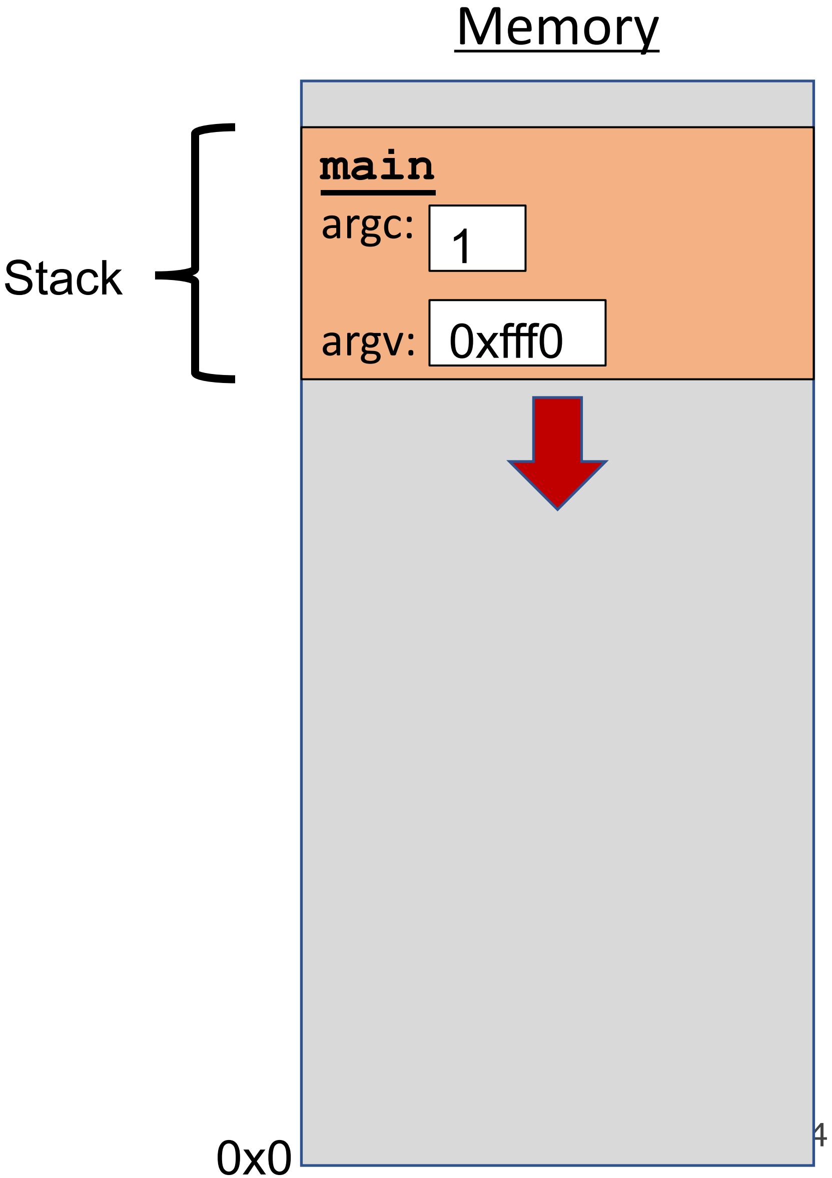
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

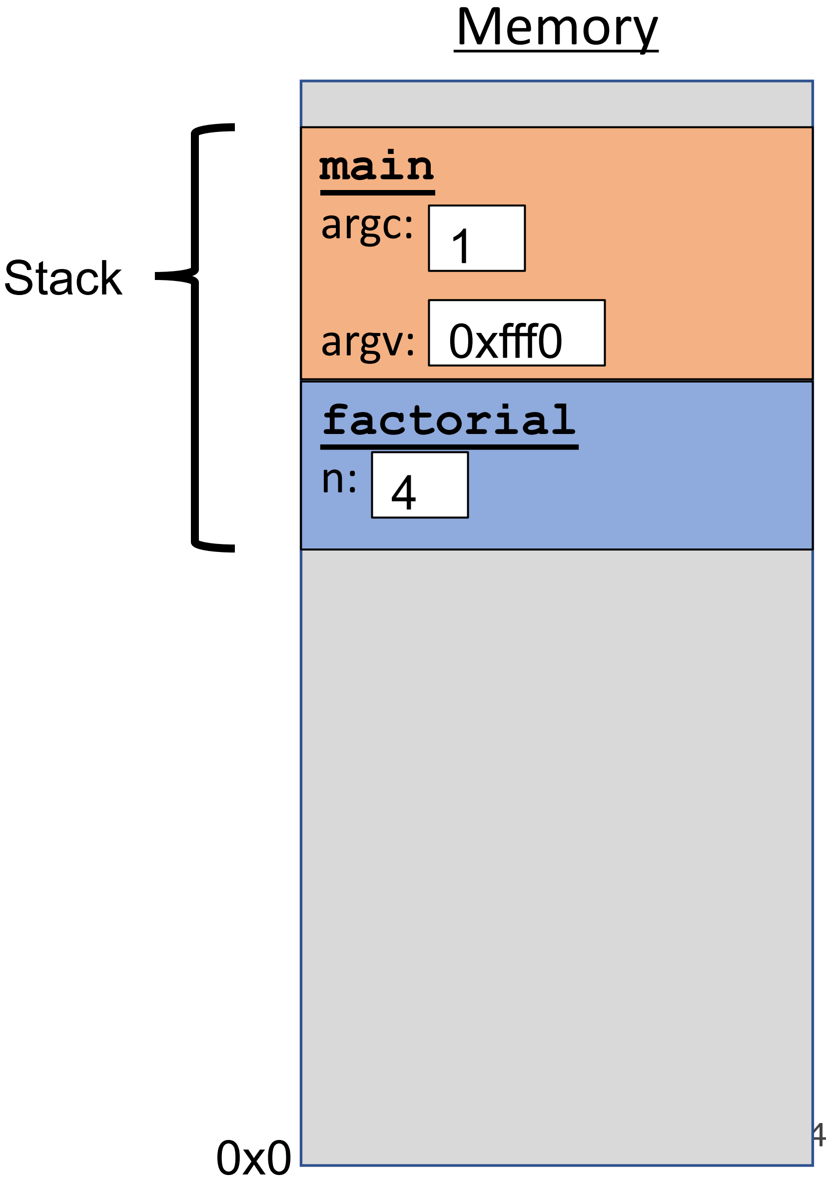
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

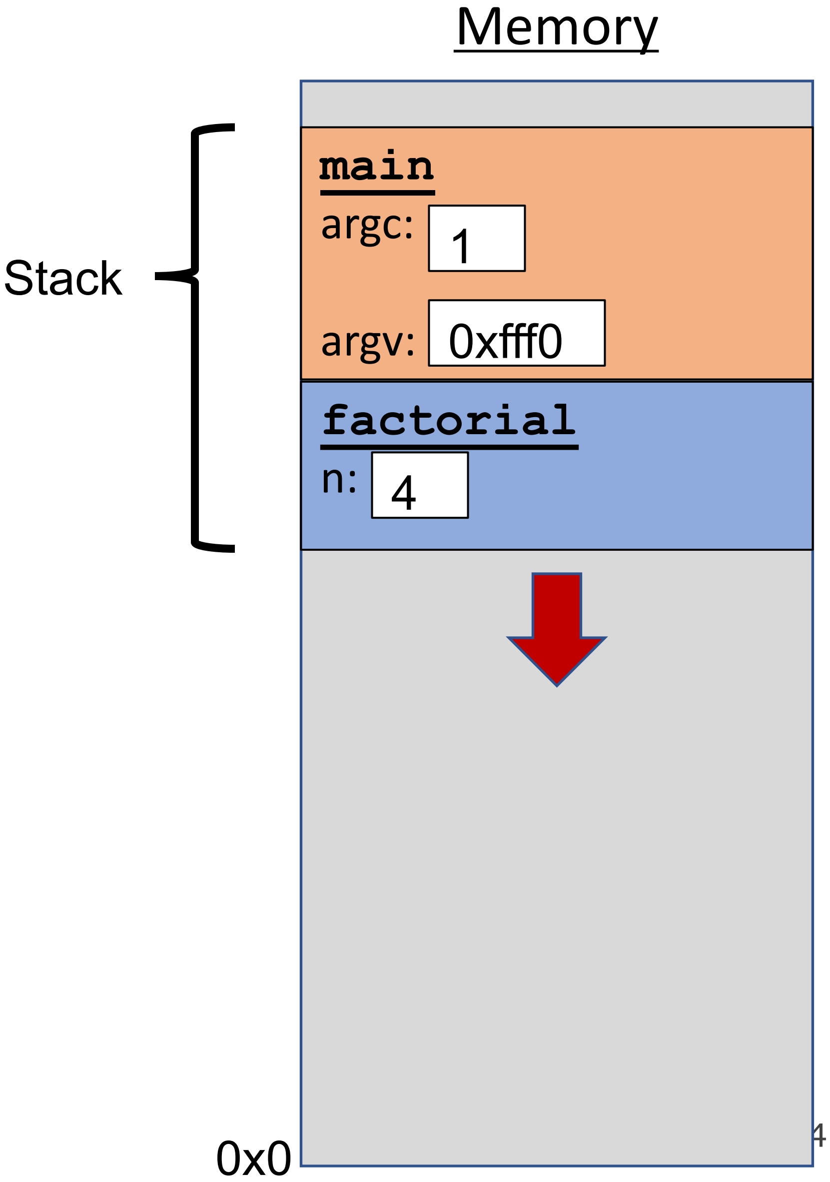
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

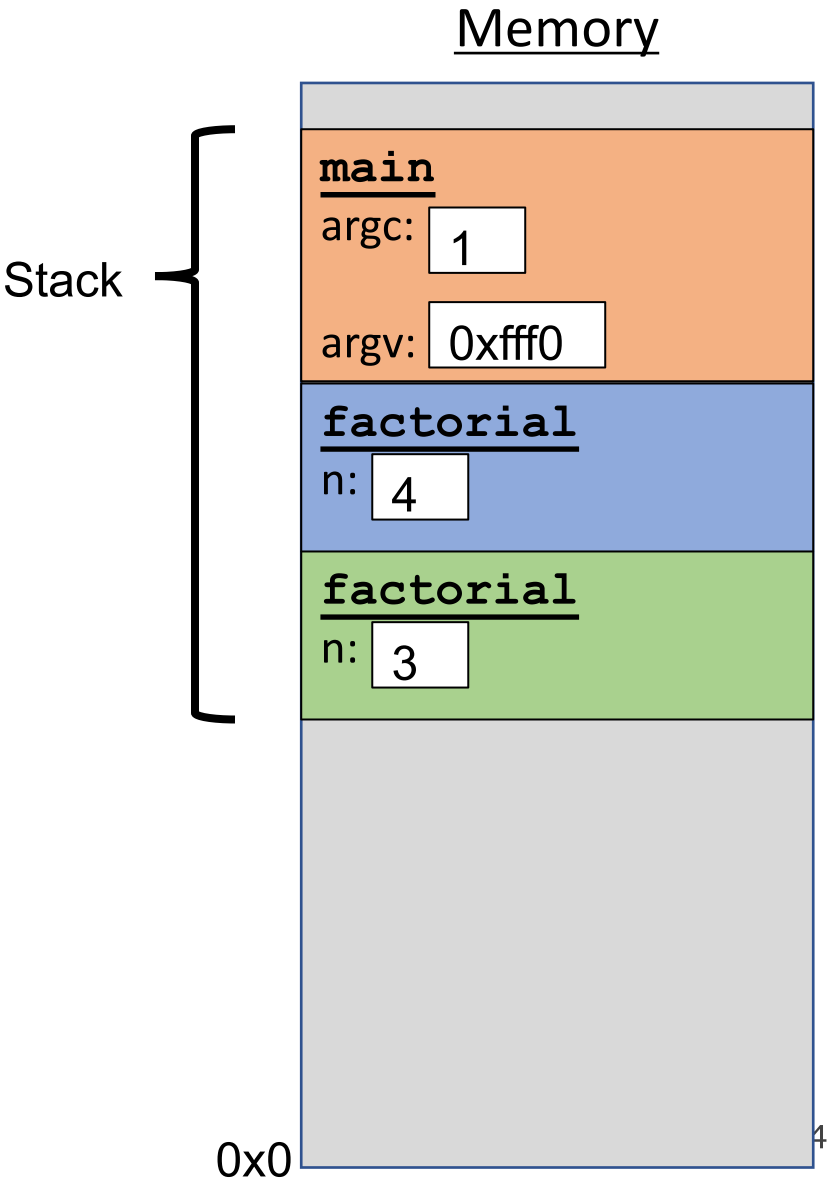
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

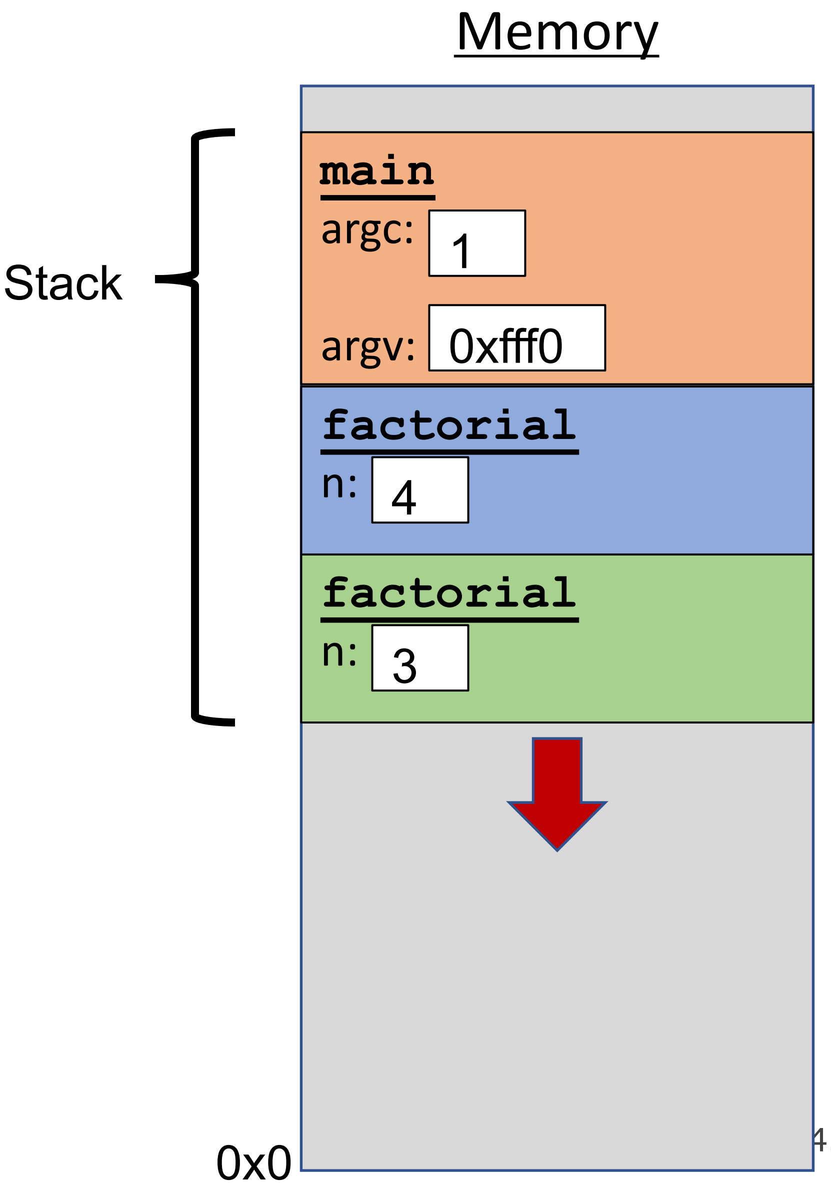
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

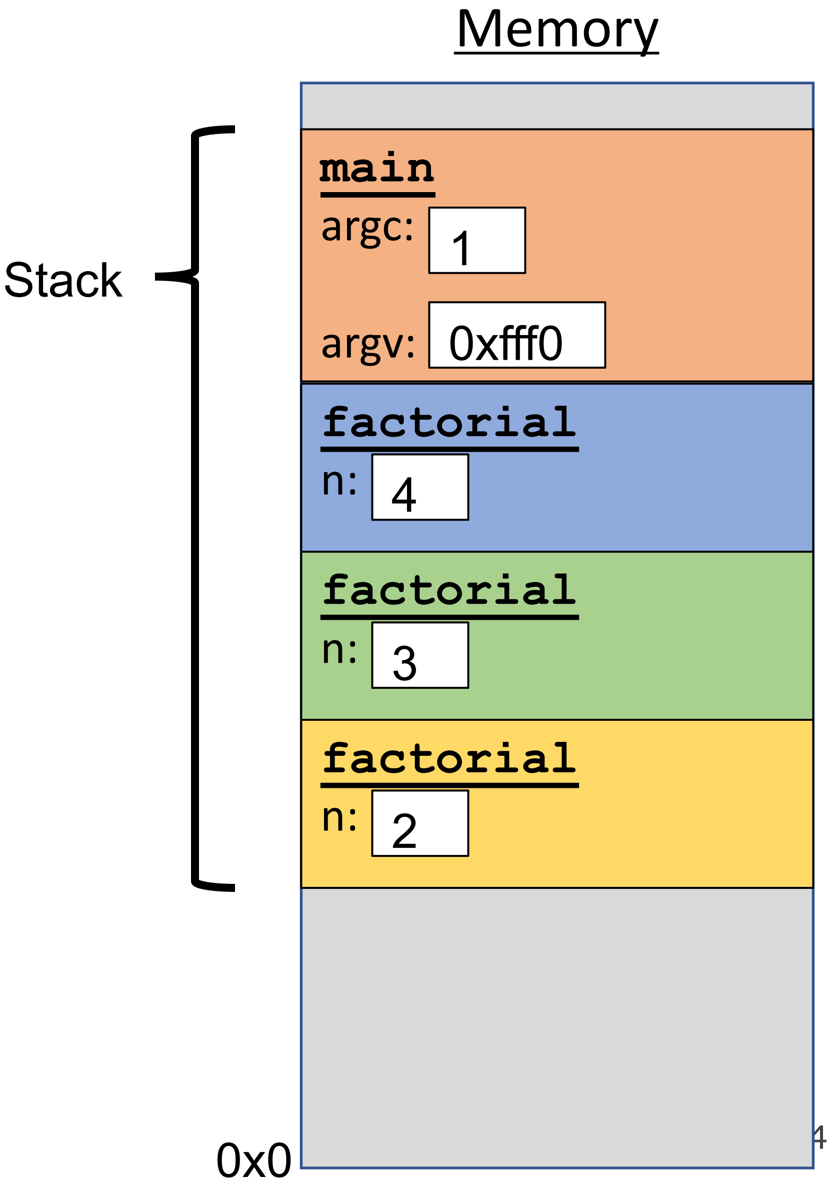
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

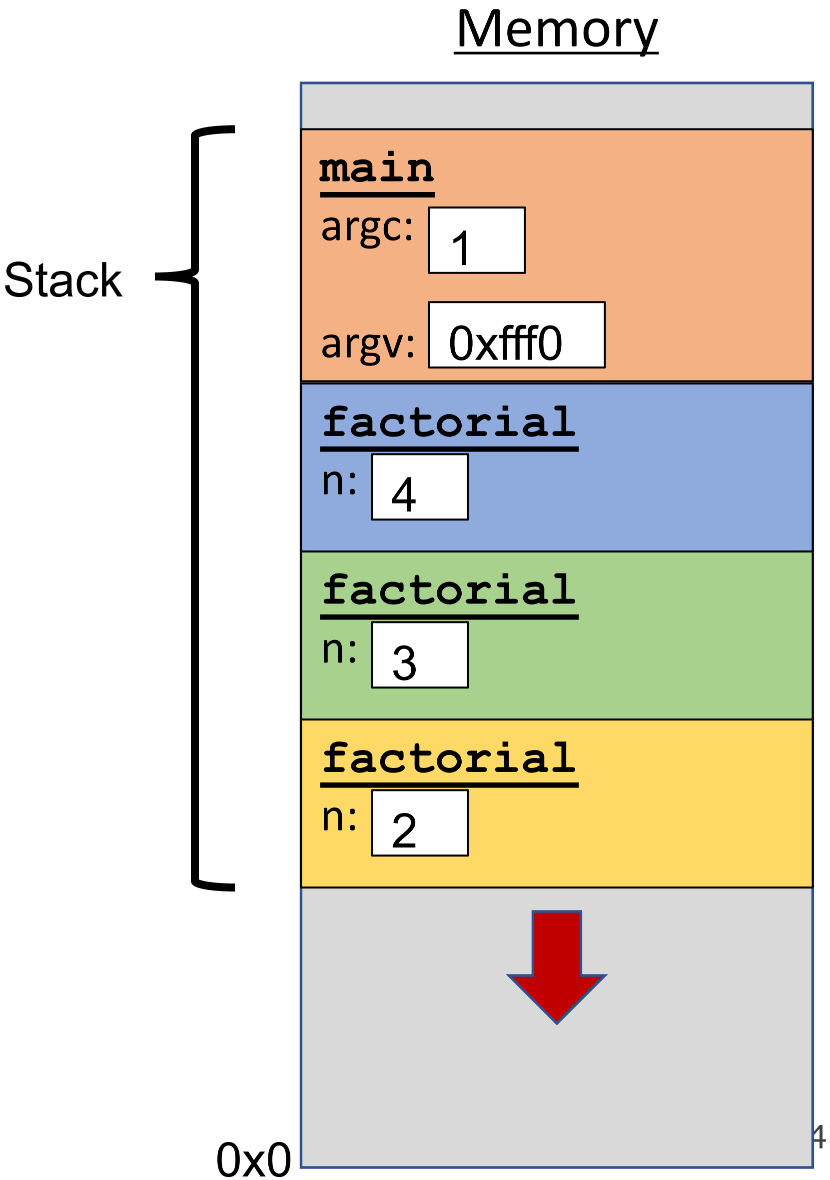
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

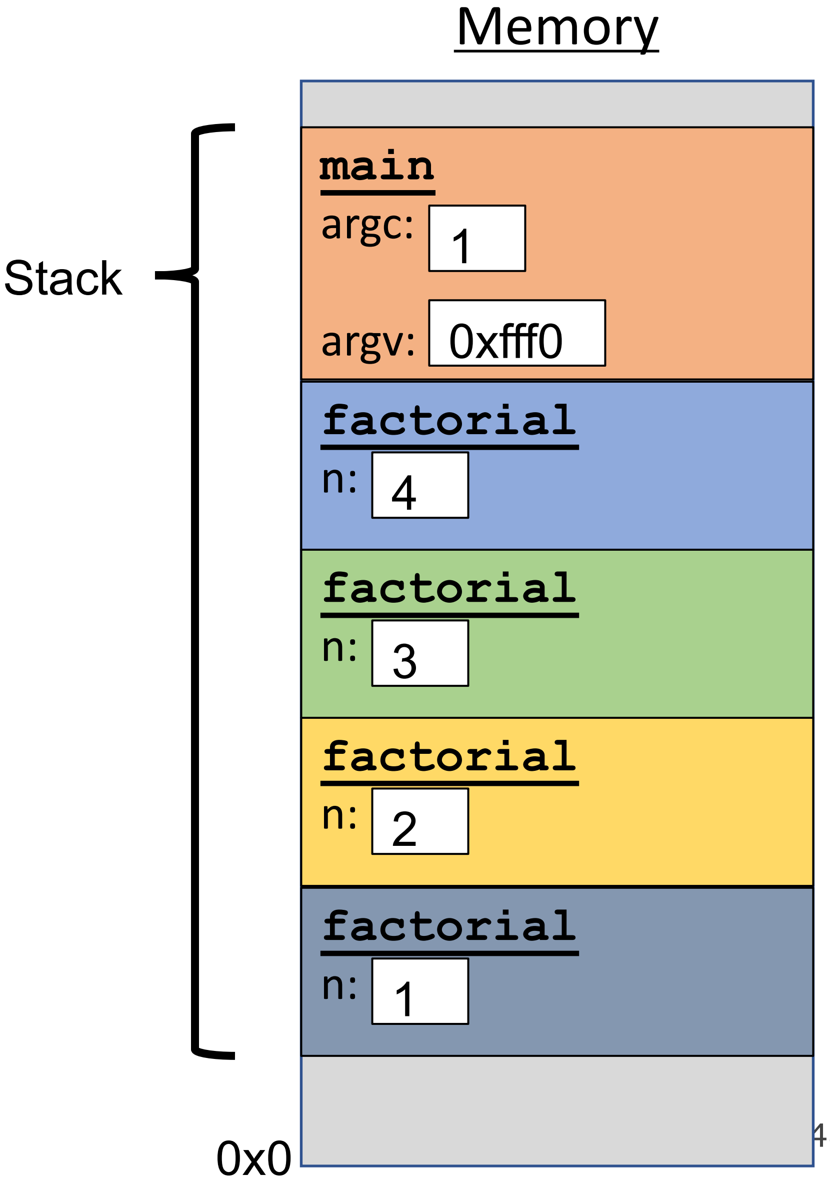
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

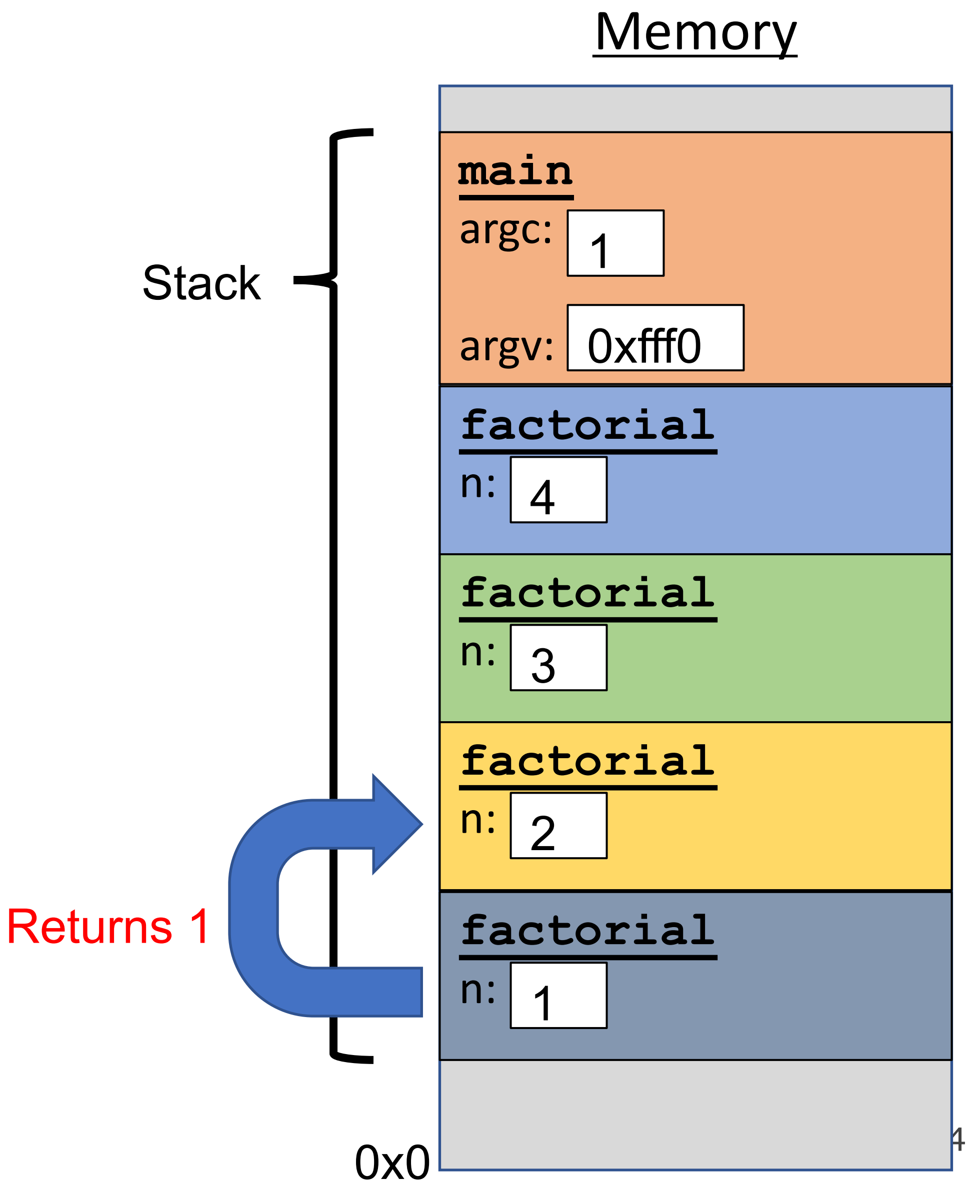
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

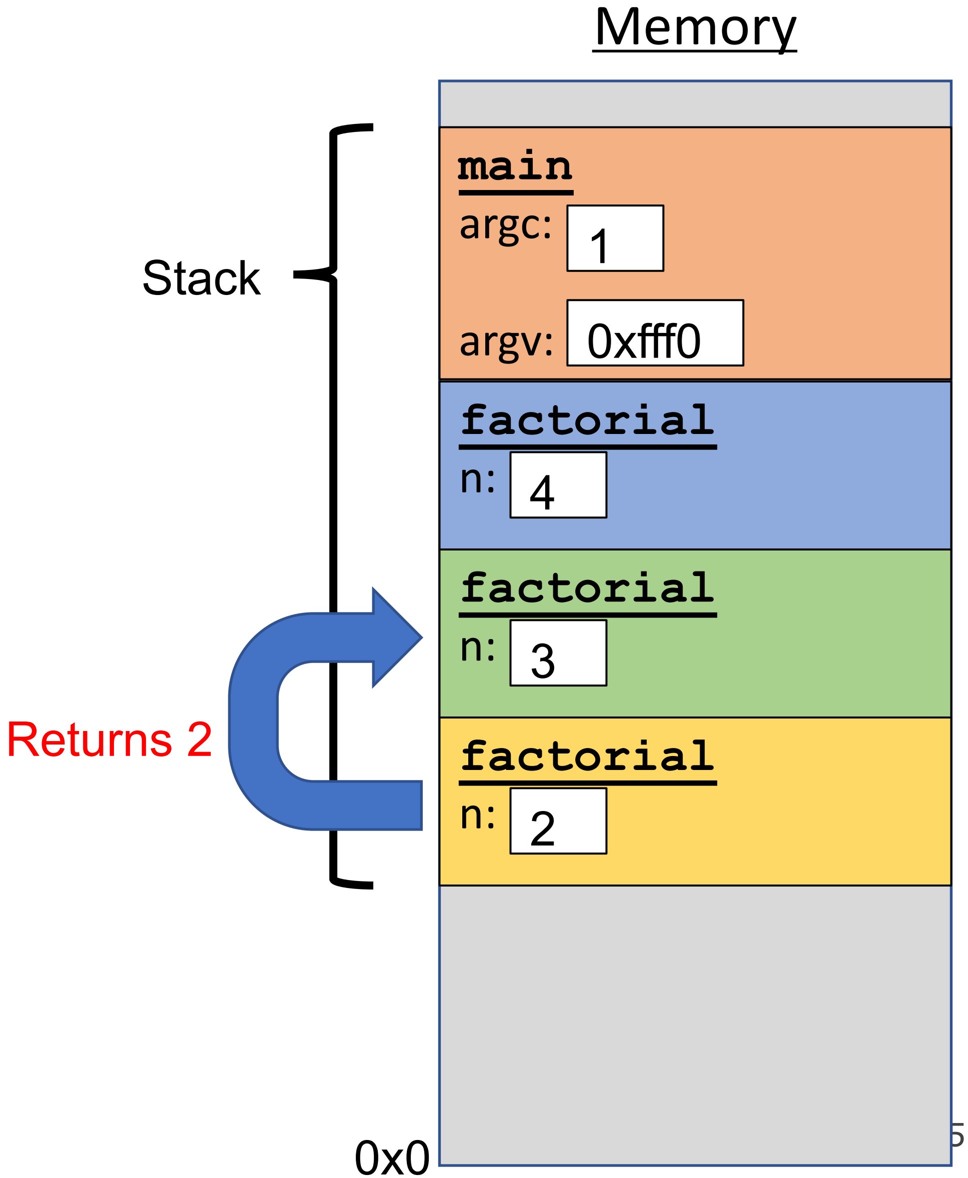
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

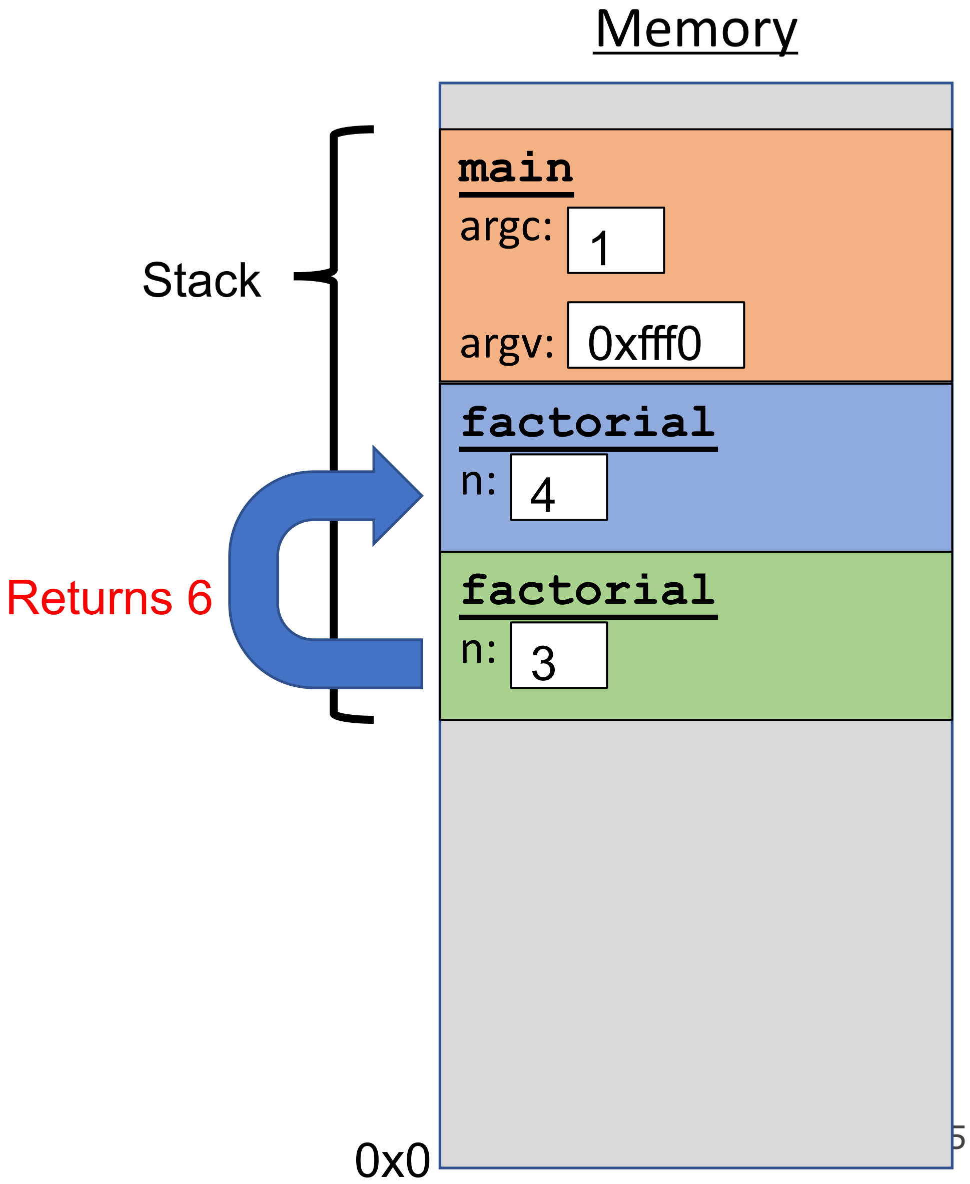
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

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

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



Memory

Stack

main

argc: 1

argv: 0xfff0

factorial

n: 4

factorial

n: 3

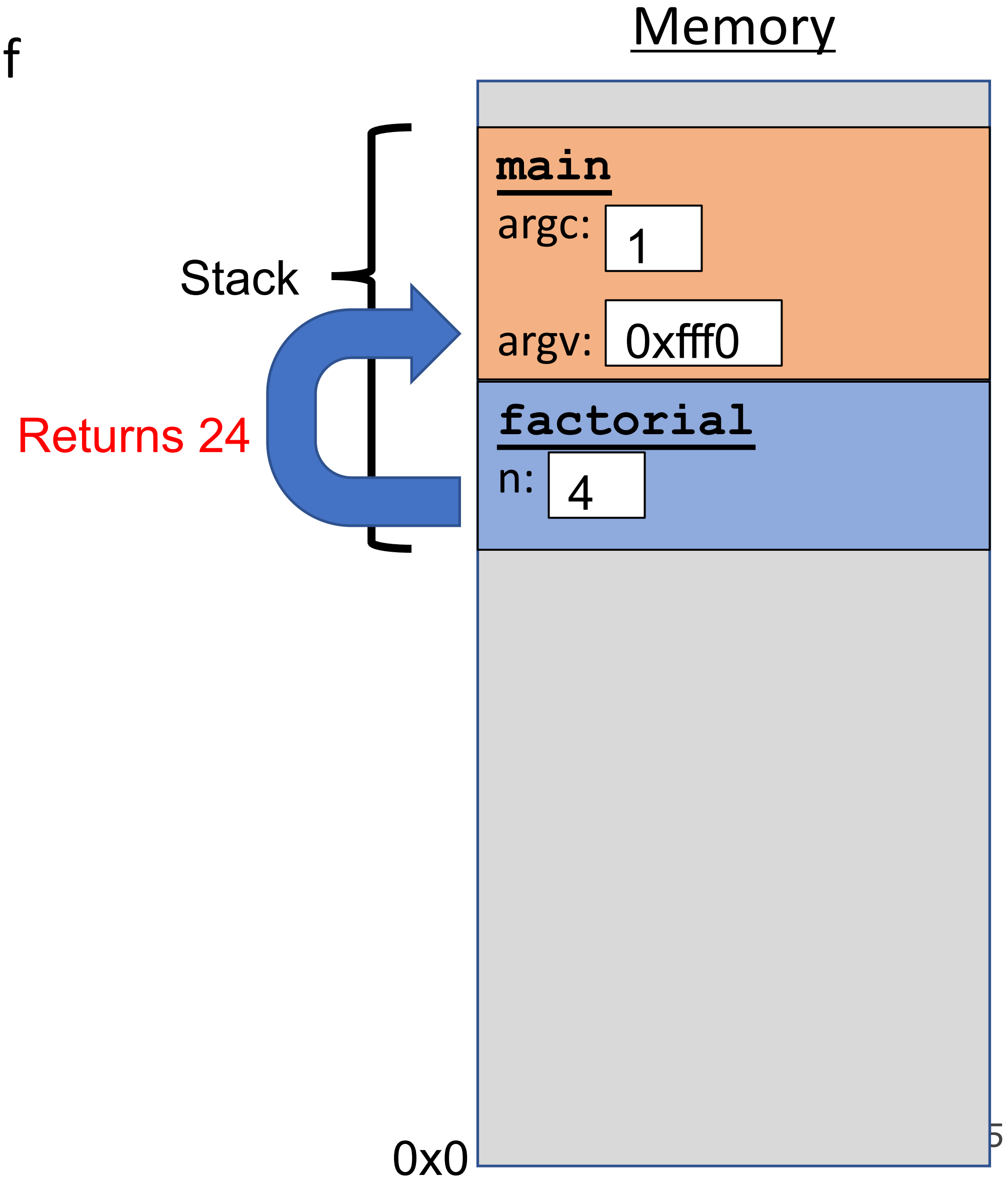
Returns 6

0x0

The Stack

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

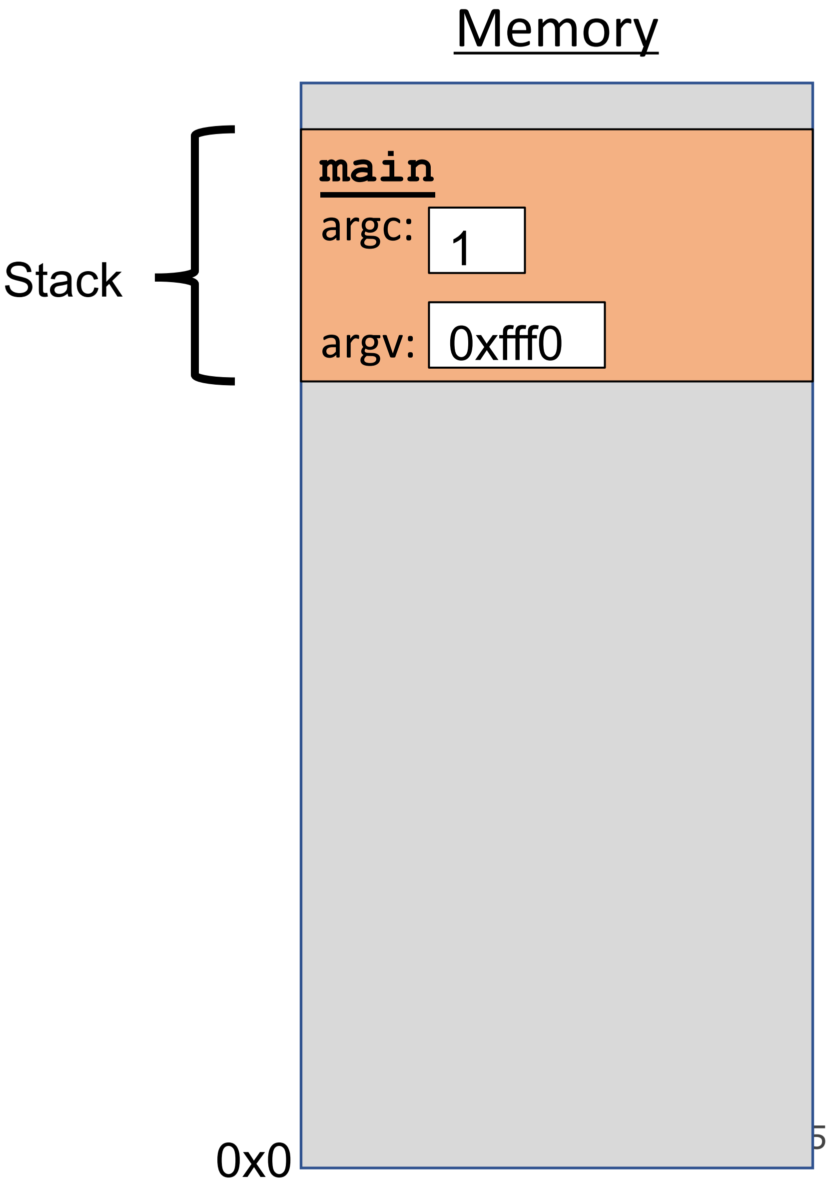
```
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

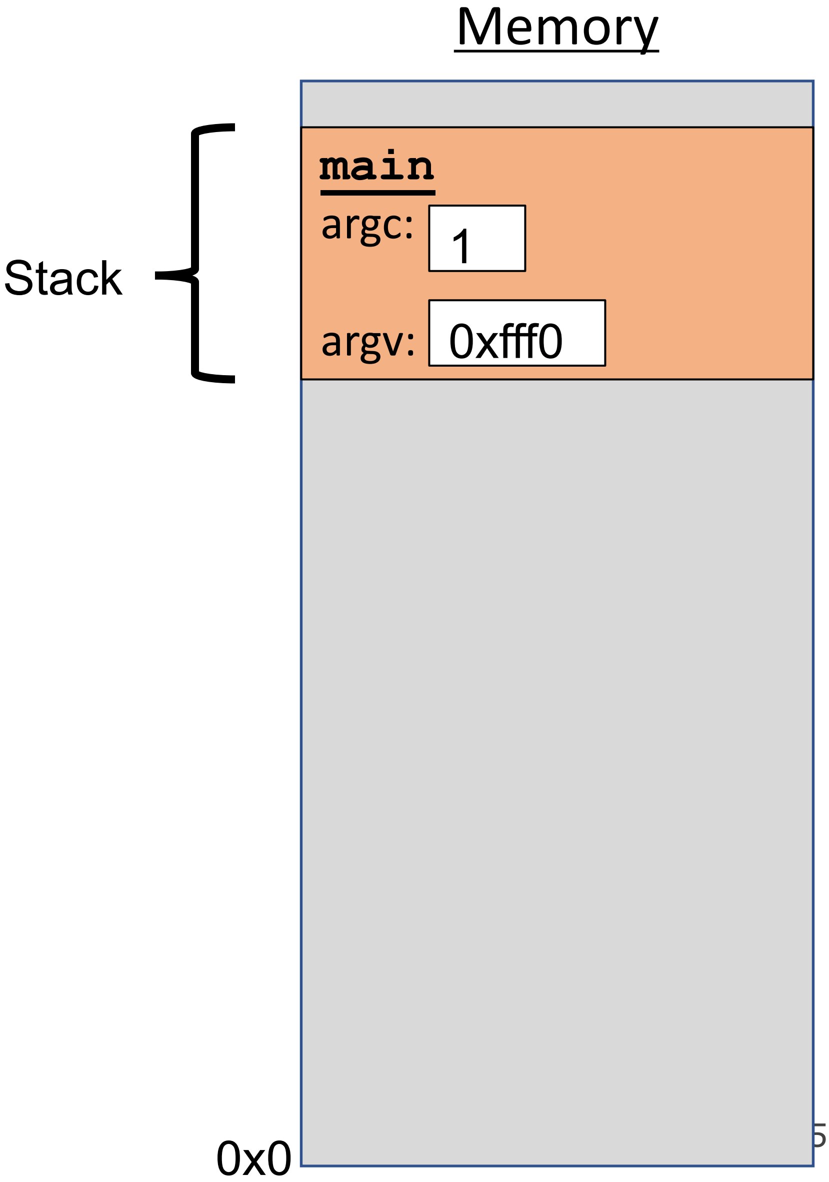
```c
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}

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



The Stack

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

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

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

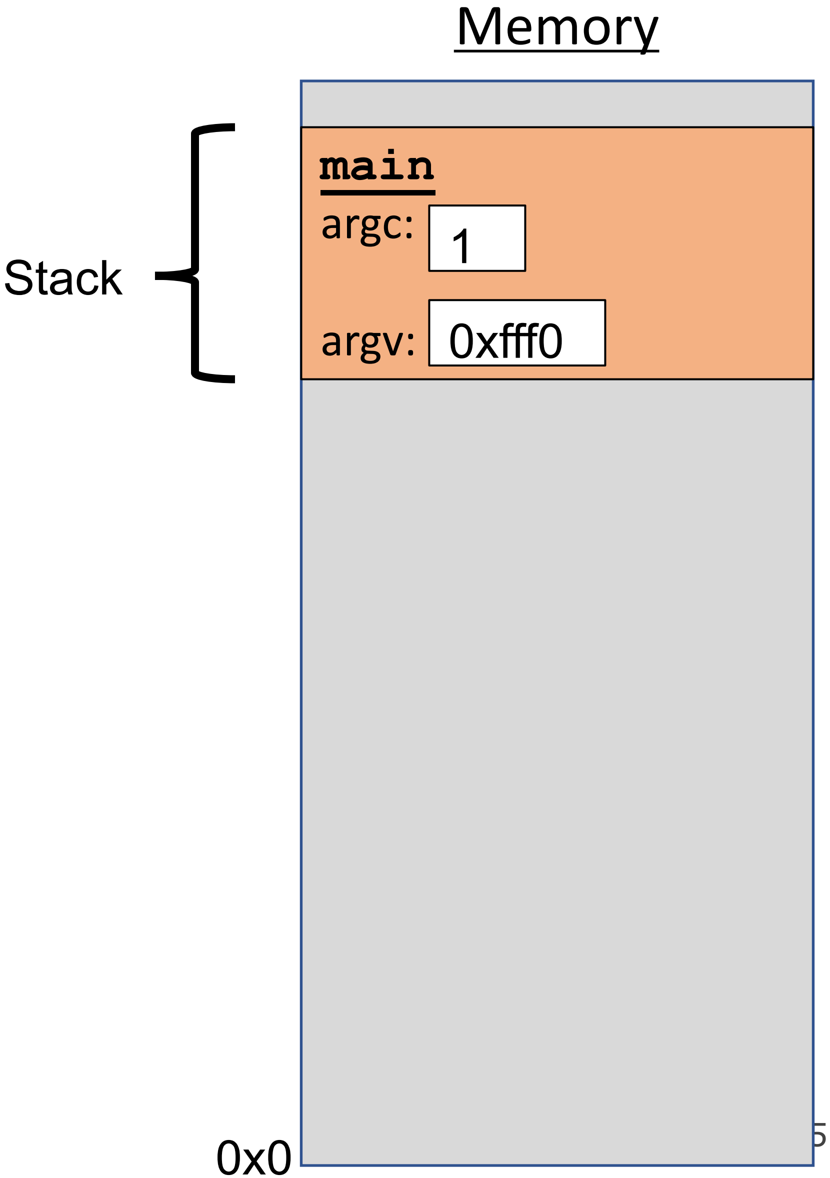


The Stack

- 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?

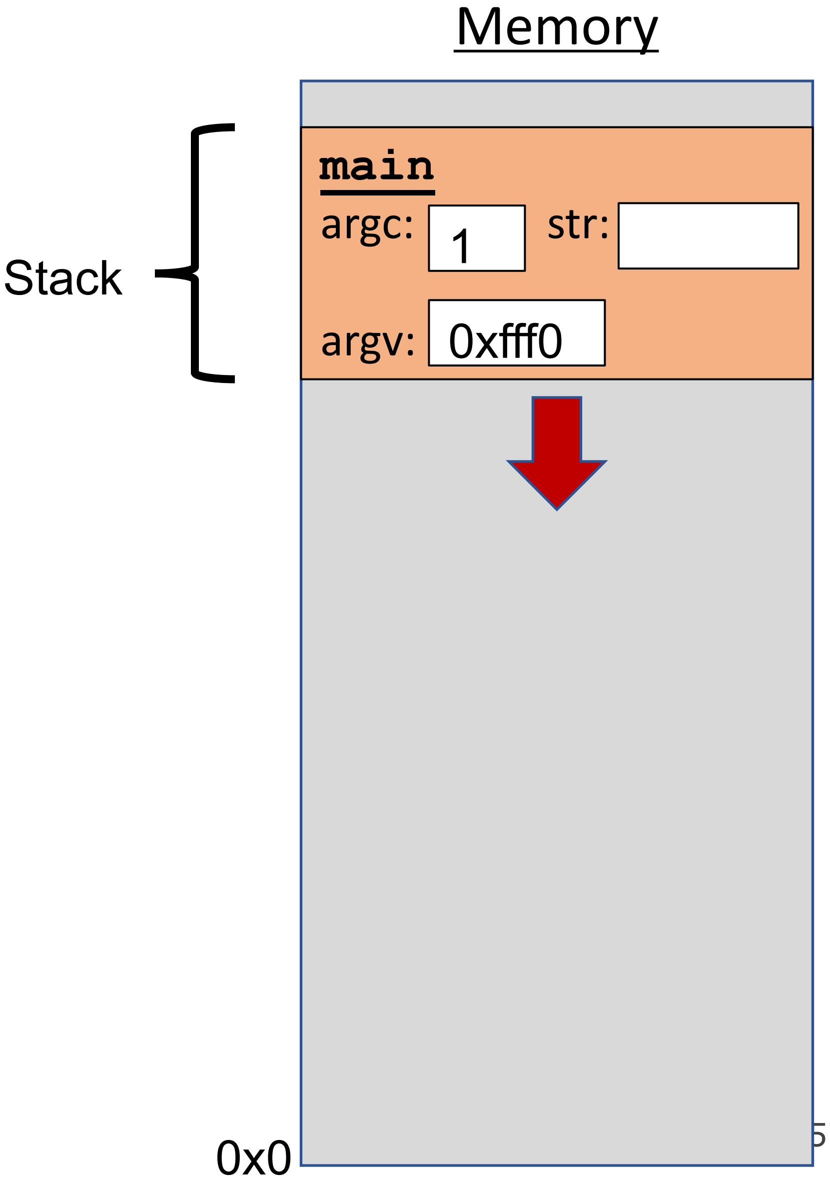
The Stack

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

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;
}
```



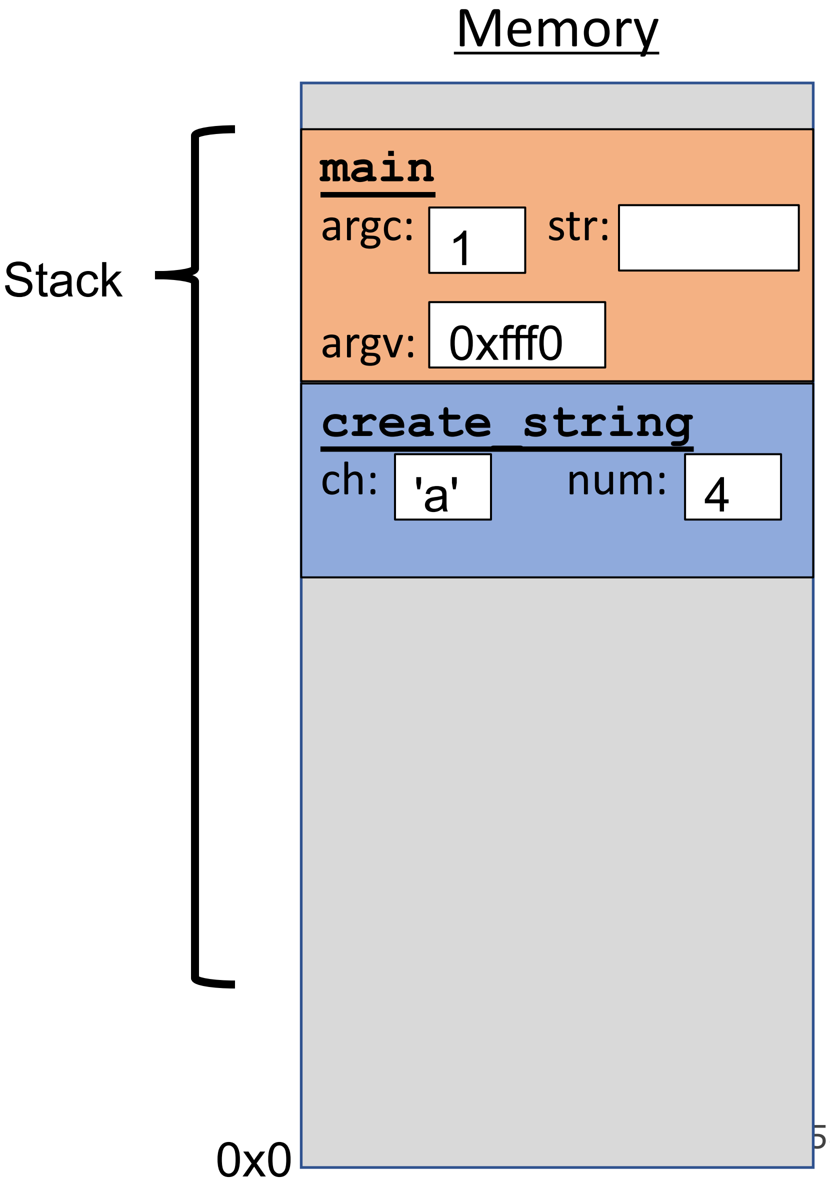
The Stack

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

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;
}
```



The Stack

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

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;
}
```



Memory

Stack

main

argc: 1

str:

argv: 0xfff0

create_string

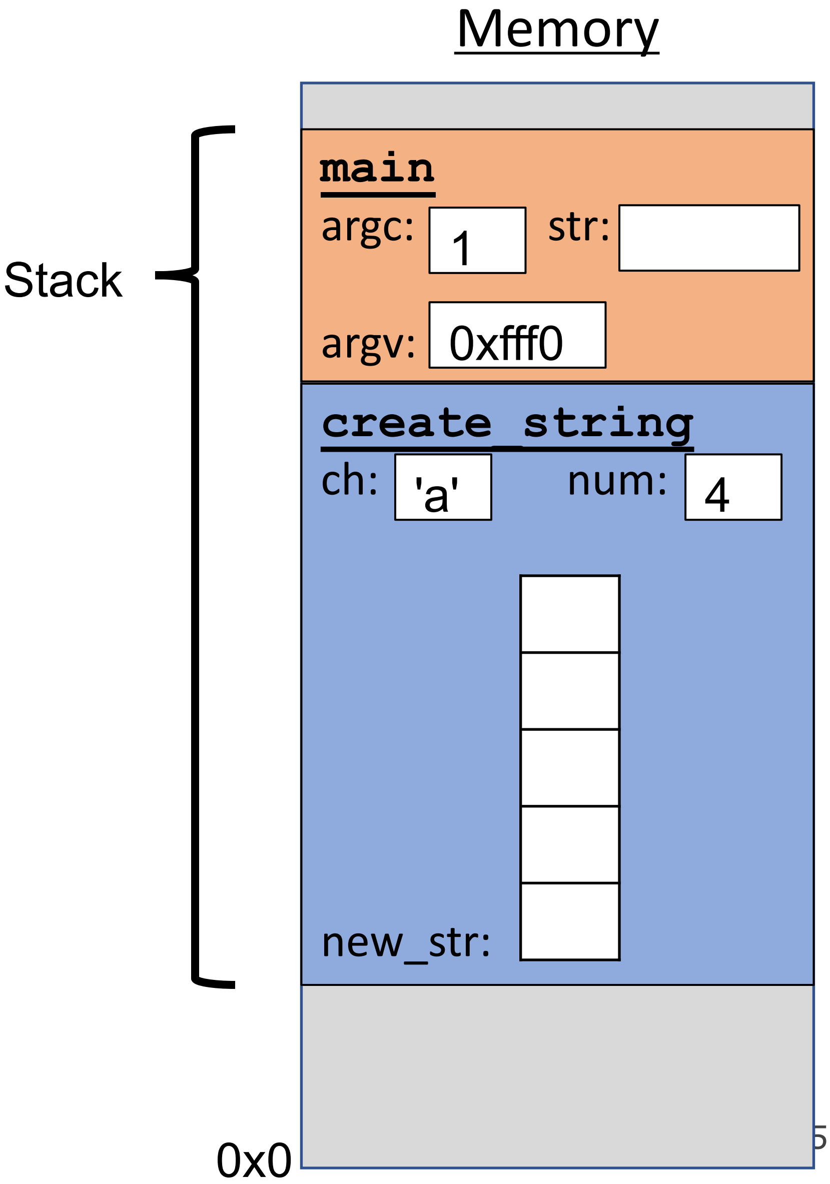
ch: 'a'

num: 4

0x0

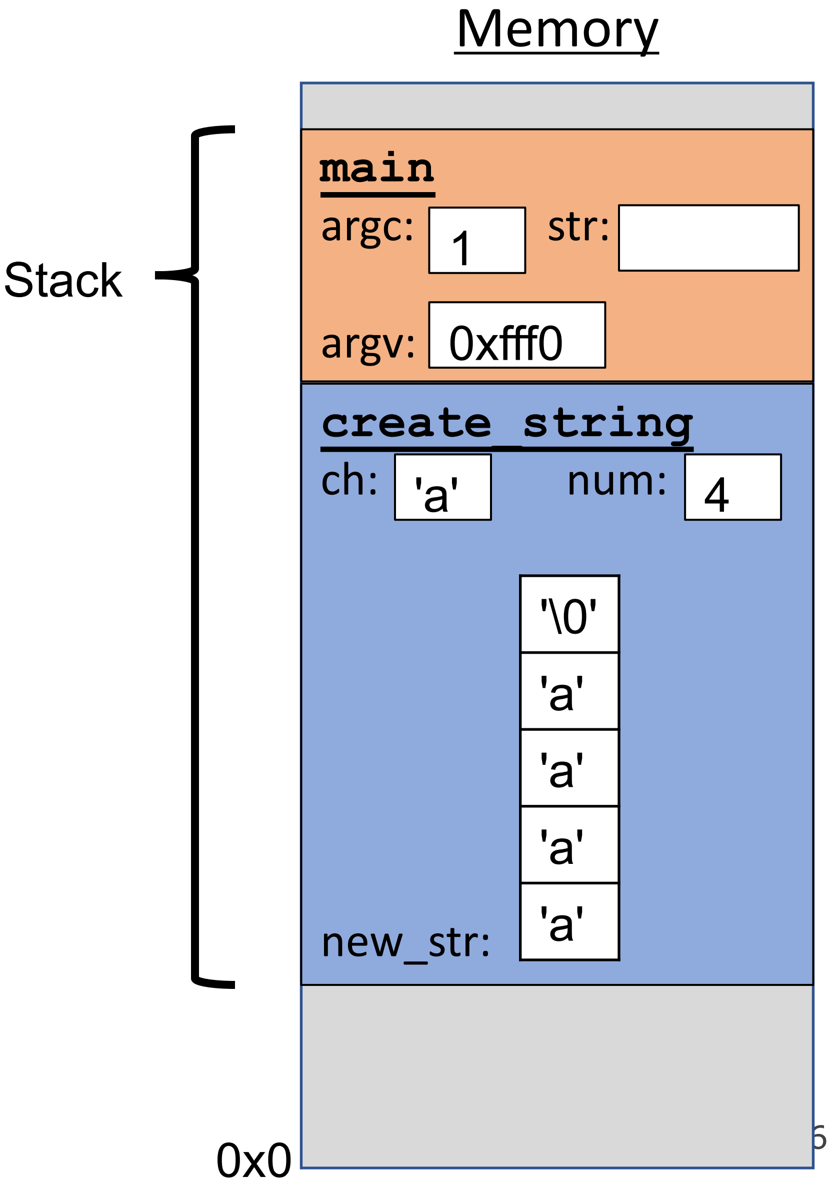
The Stack

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

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;
}
```



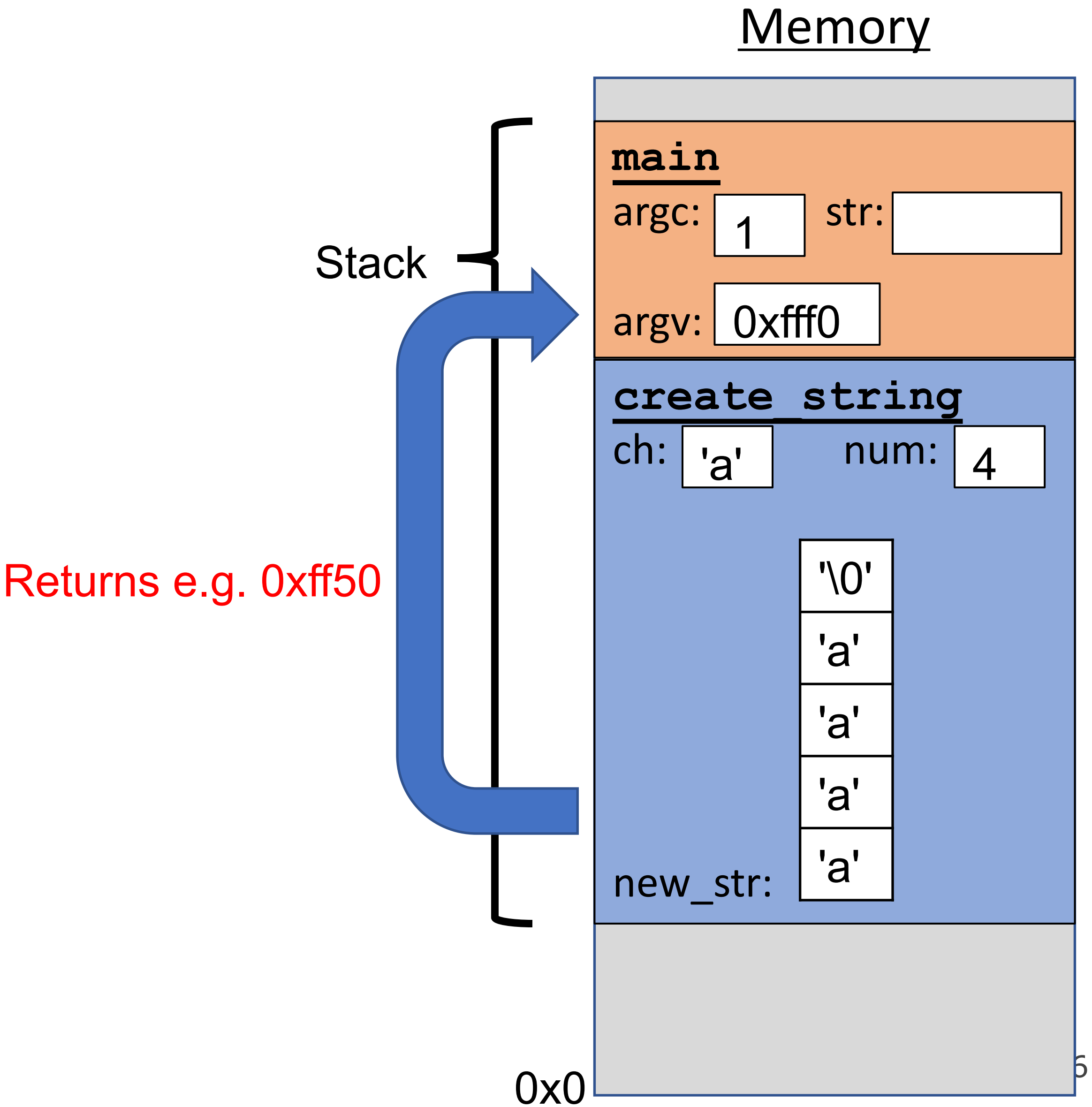
The Stack

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

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;
}
```



The Stack

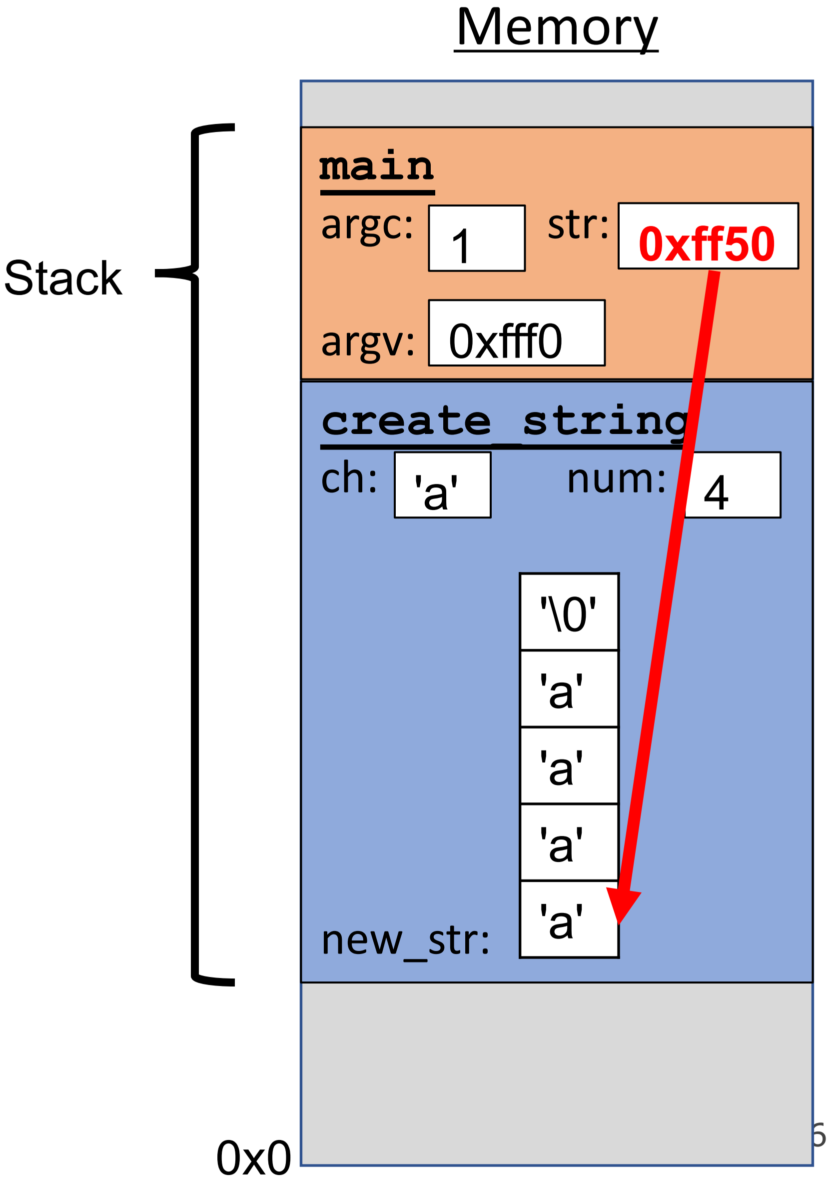
```
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}

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;
}
```



Returns e.g. 0xff50

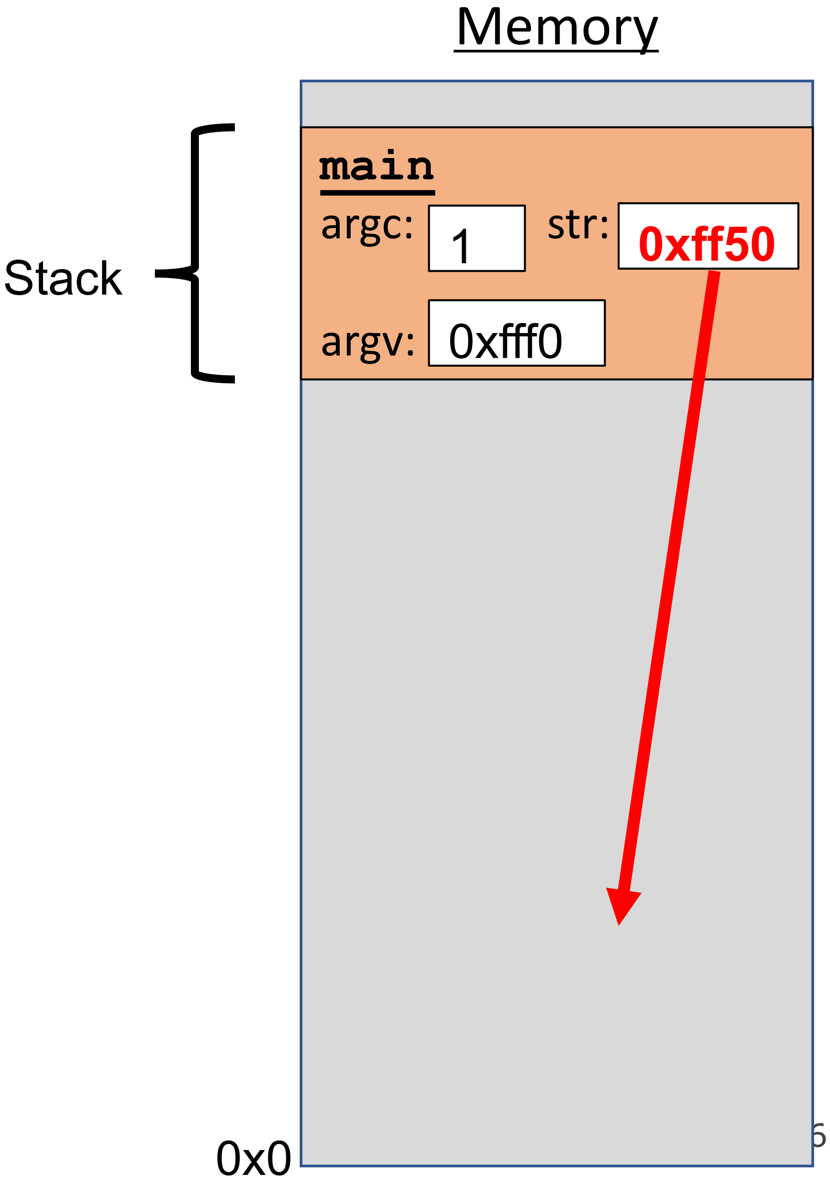
The Stack

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

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;
}
```



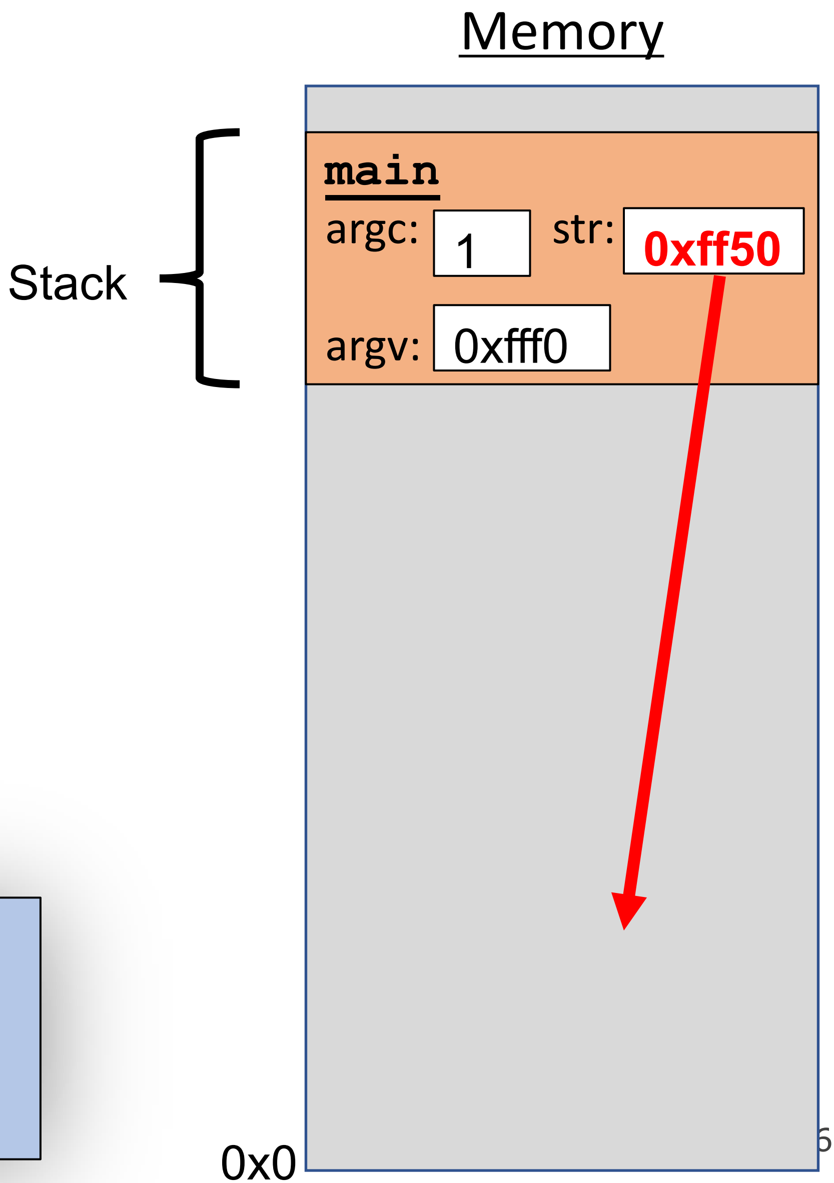
The Stack

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

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;
}
```



The Stack

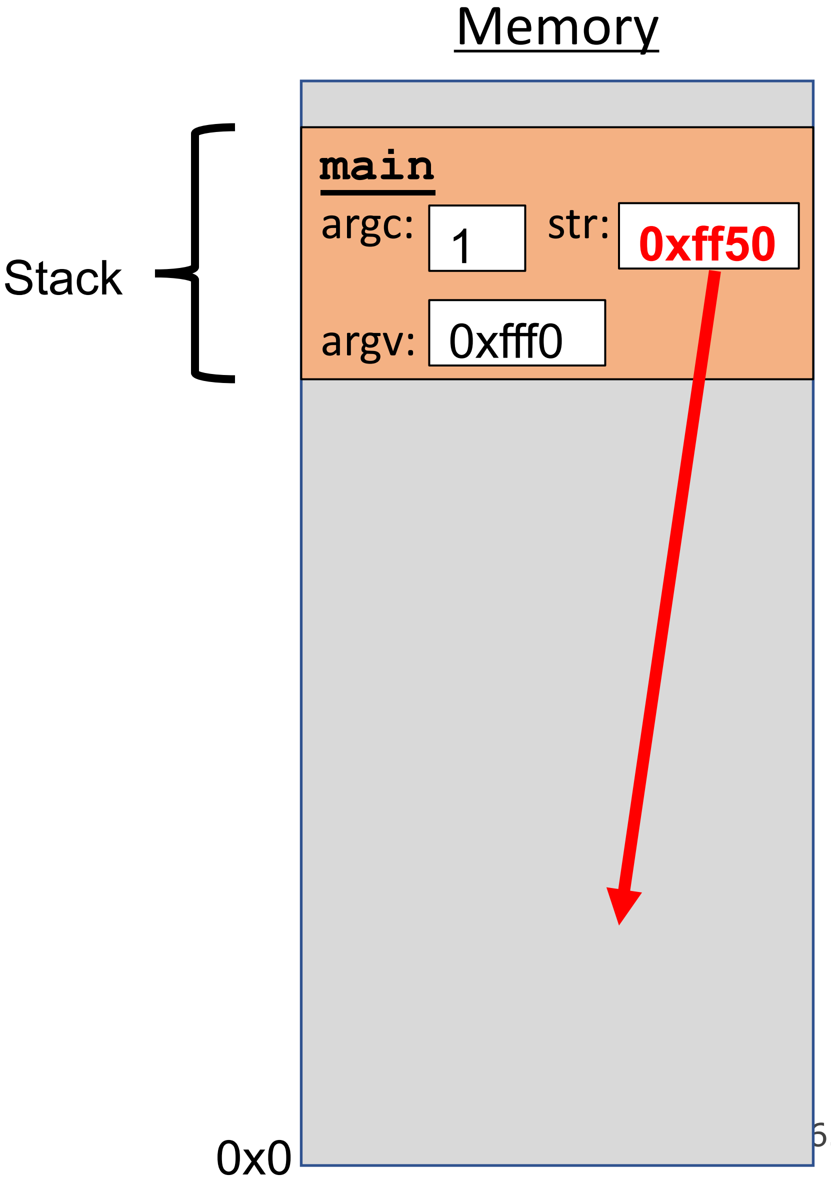
```
int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}

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;
}
```



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

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

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;
}
```



Stacked Against Us

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

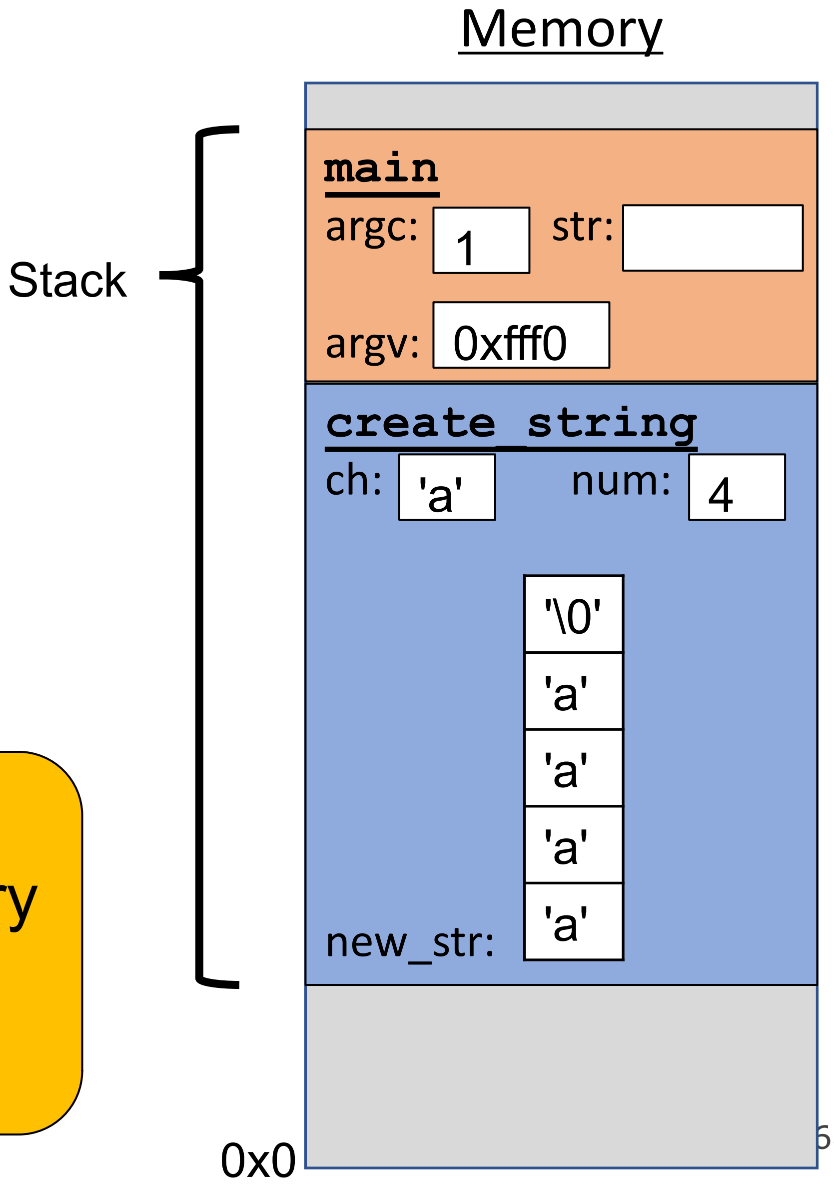
Plan For Today

- Miscellaneous Useful Topics
  - **const**
  - Structs
  - Ternary
- The Stack
- **The Heap and Dynamic Memory**
- **Practice:** Pig Latin
- Announcements
- `Realloc`
- **Practice:** Pig Latin Part 2

The Heap

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

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;
}
```

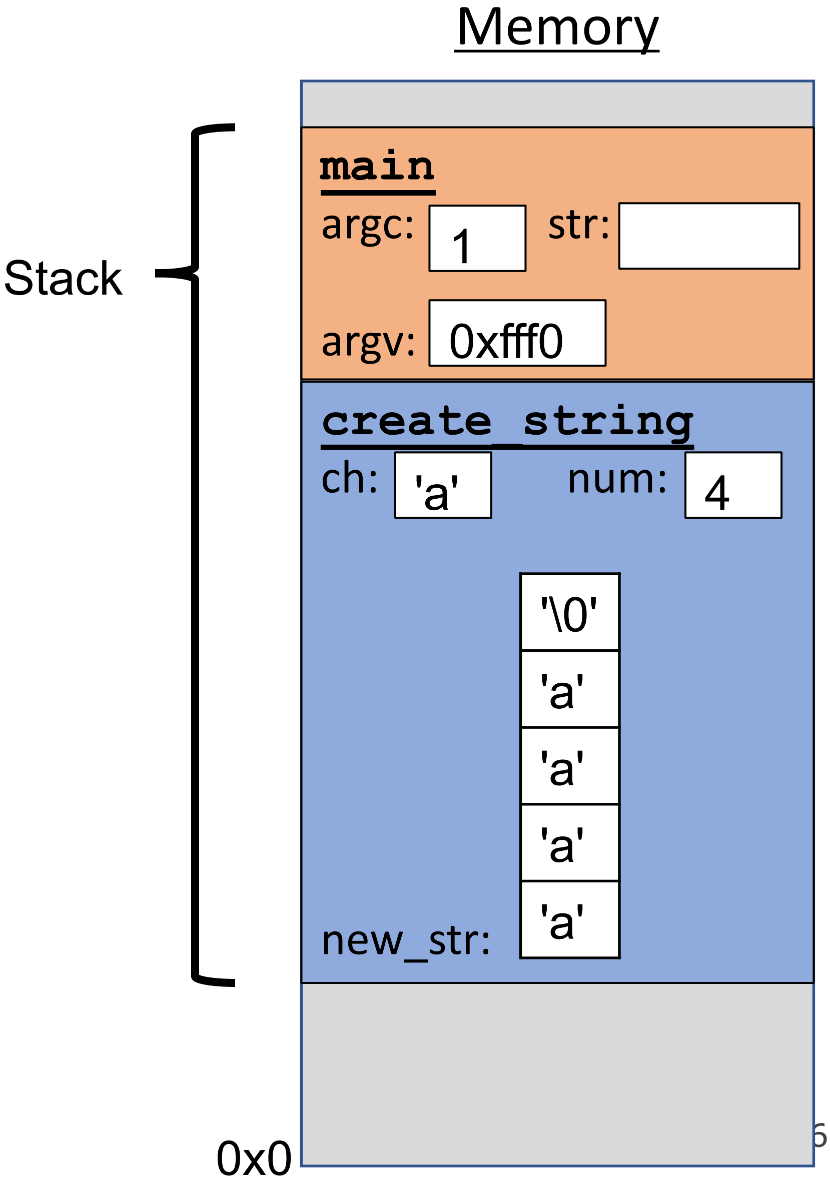
Us: hey C, is there a way to make this variable in memory that isn't automatically cleaned up?



The Heap

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

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;
}
```

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



Memory

Stack

main

argc: 1

str:

argv: 0xfff0

create_string

ch: 'a'

num: 4

'\0'

'a'

'a'

'a'

'a'

new_str:

0x0

The Heap

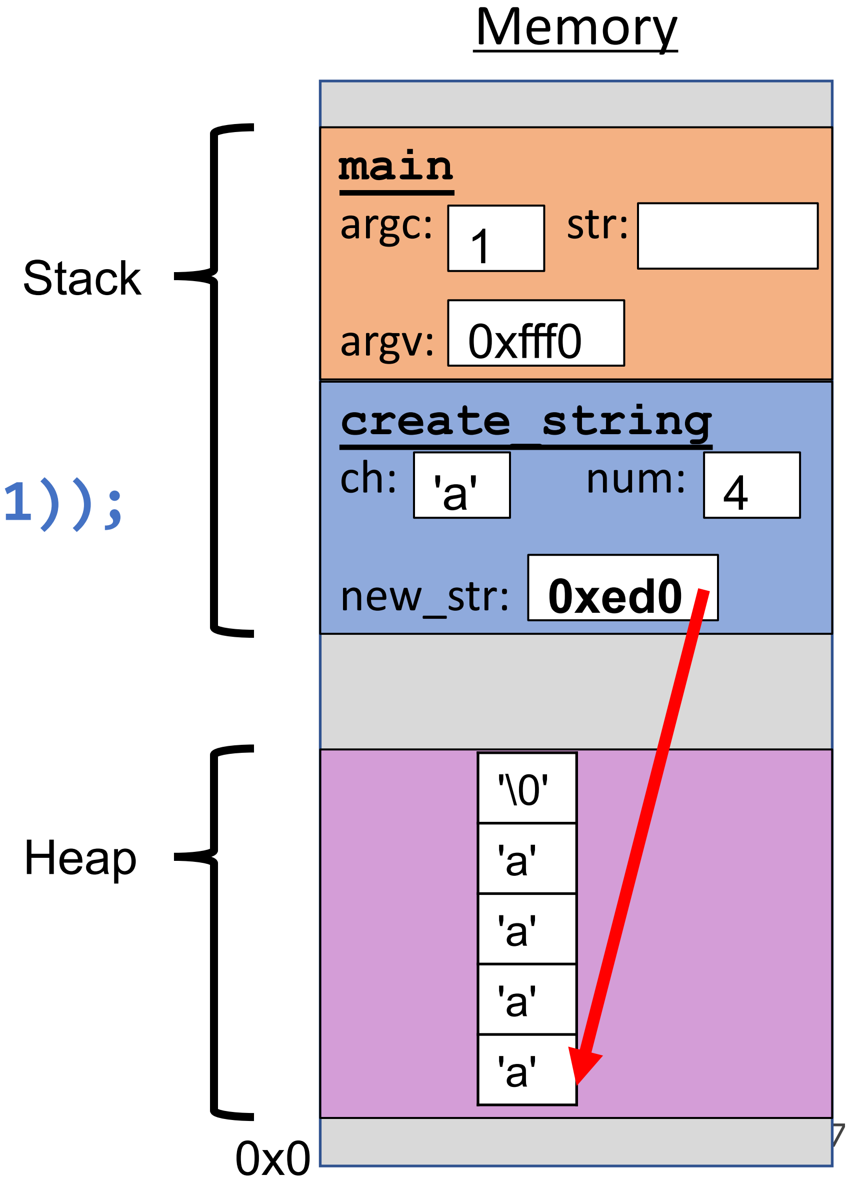
- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- Unlike the stack, the heap grows **upwards** as more memory is allocated.
- To allocate memory on the heap, use the **`malloc`** function ("memory allocate") and specify the number of bytes you'd like. This function returns *the address on the heap of the new memory*.

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

- **`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!

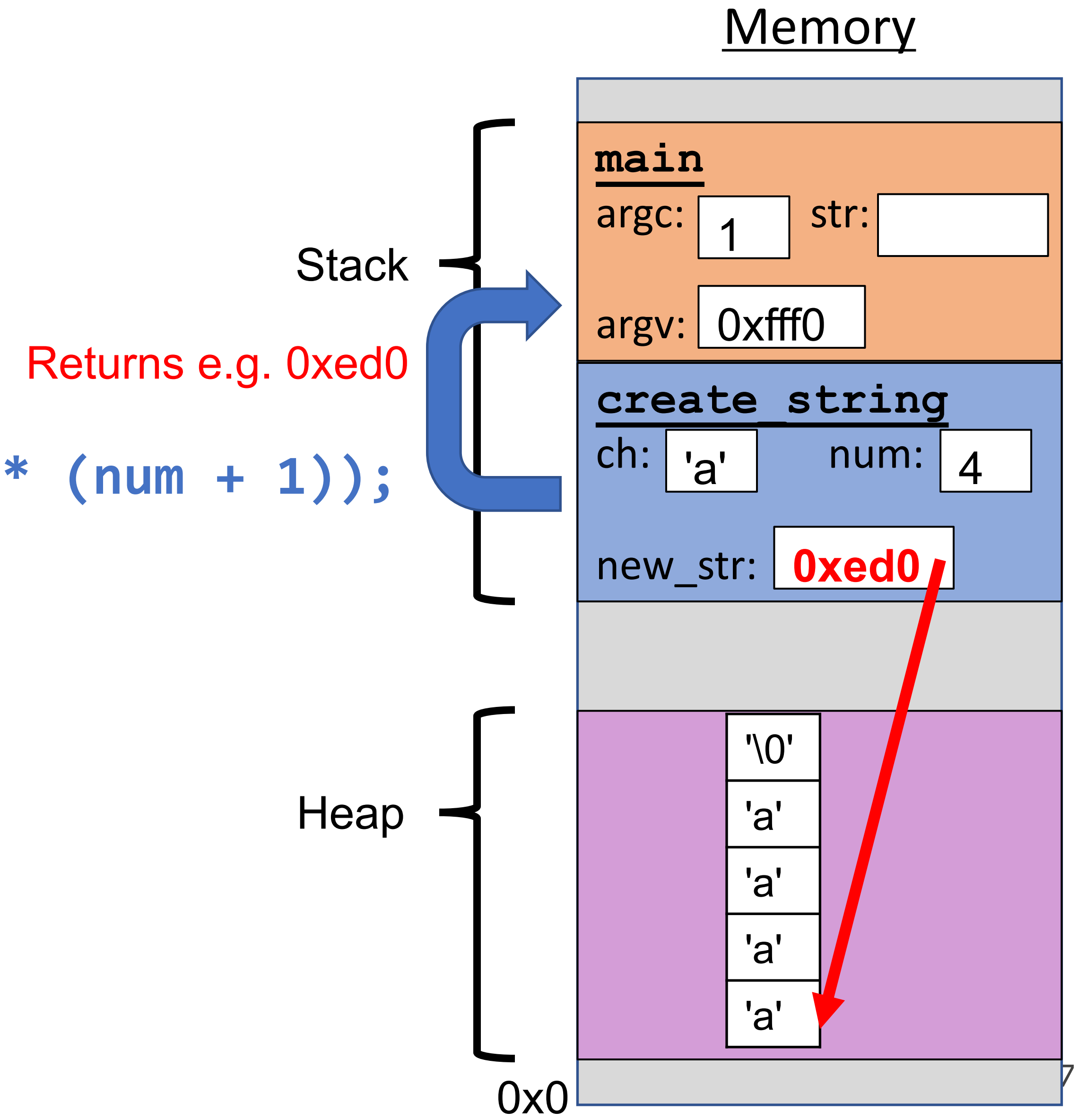
The Heap

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

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;
}
```



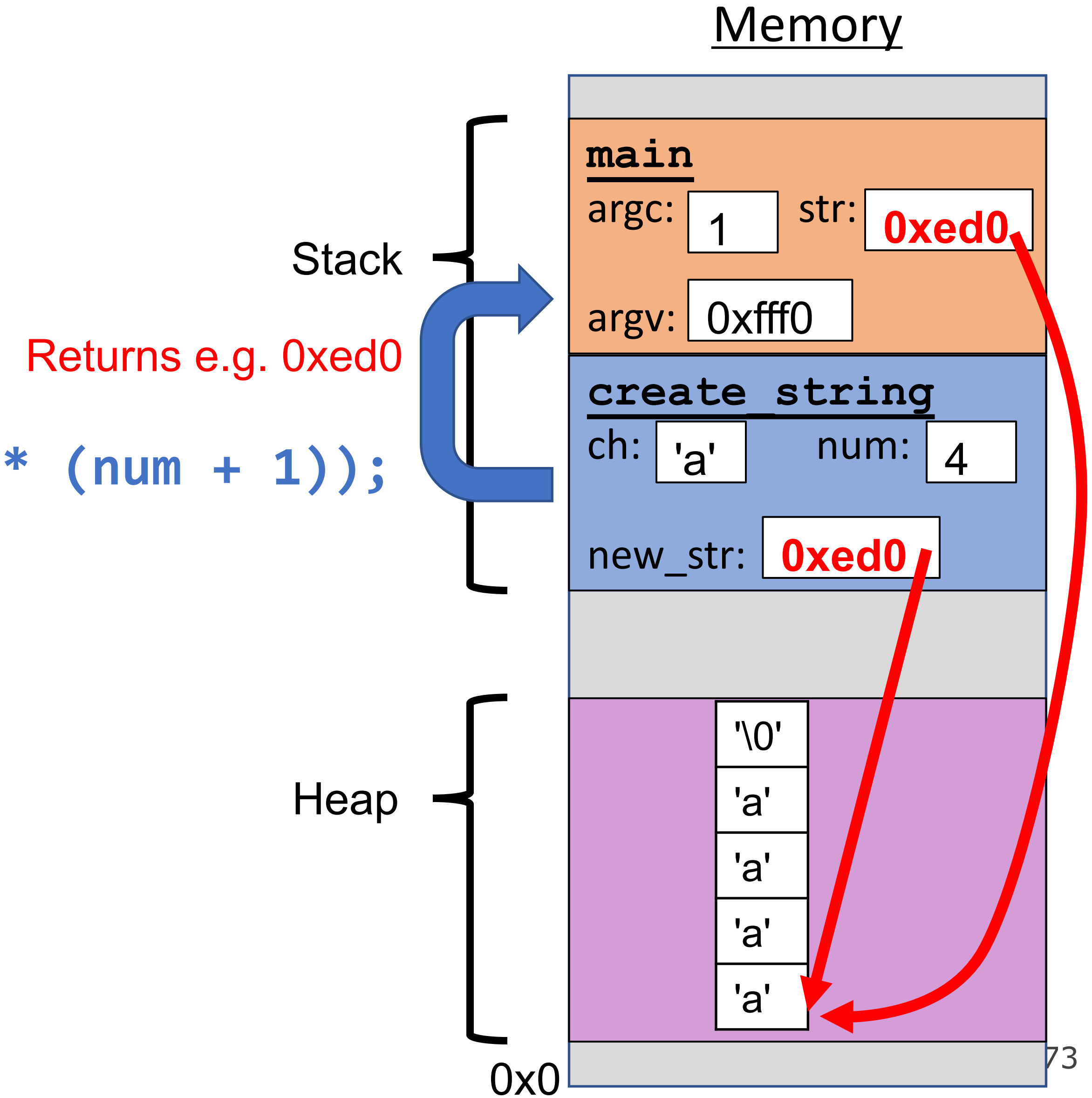
The Heap

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

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;
}
```



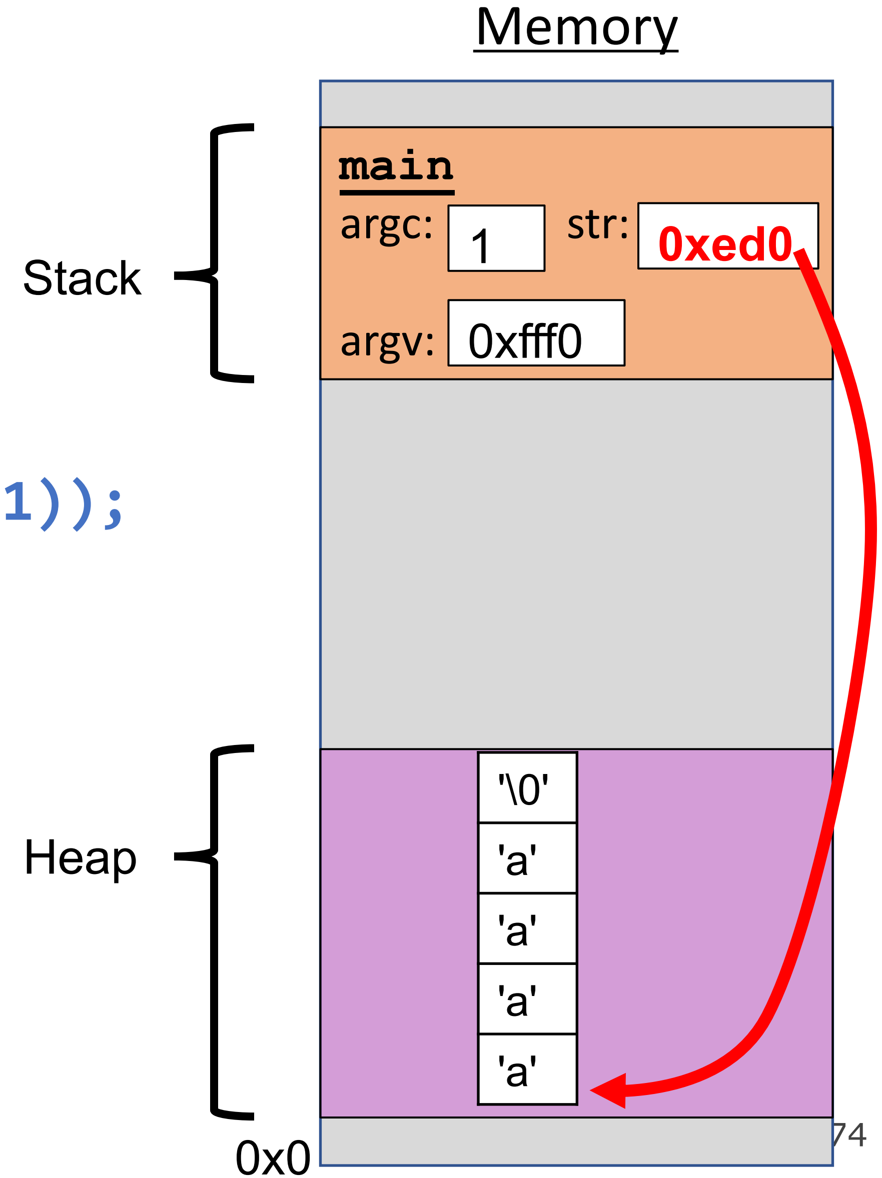
The Heap

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

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;
}
```



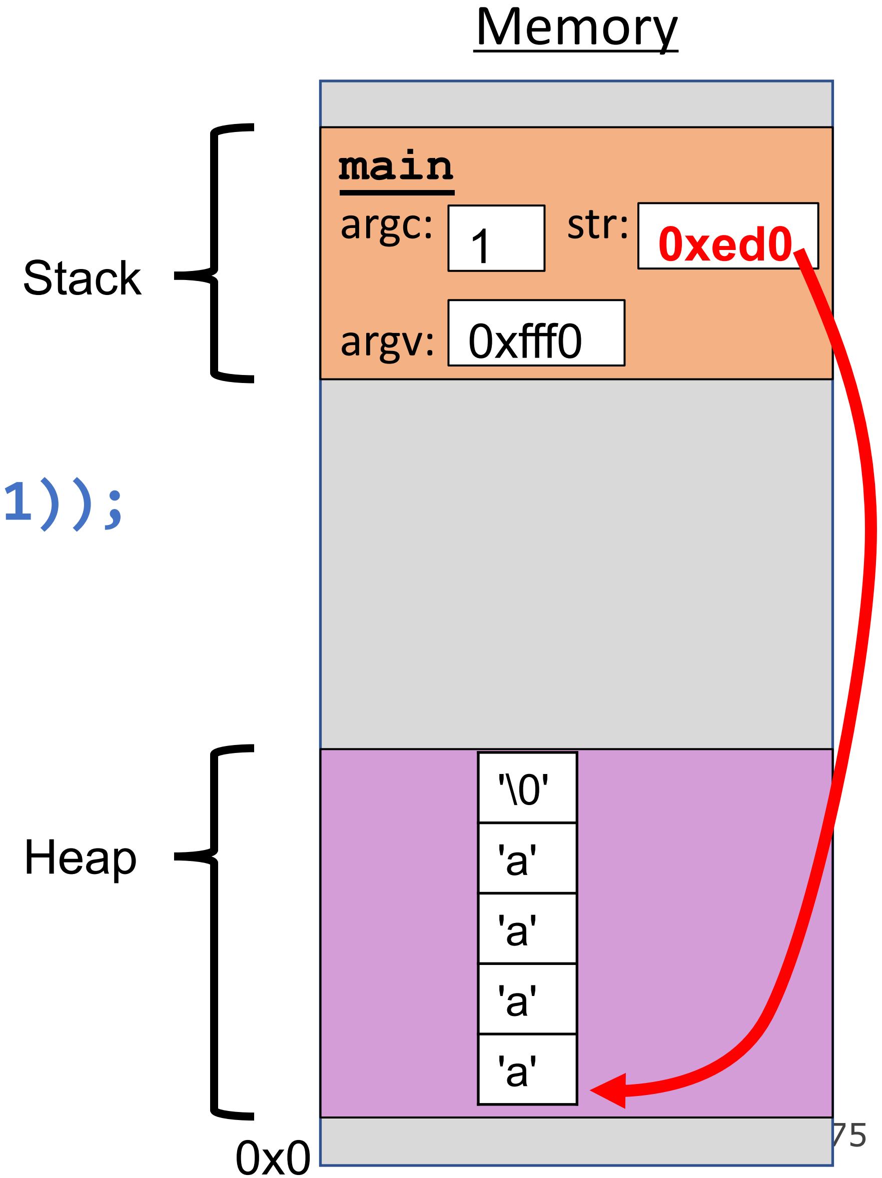
The Heap

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

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;
}
```



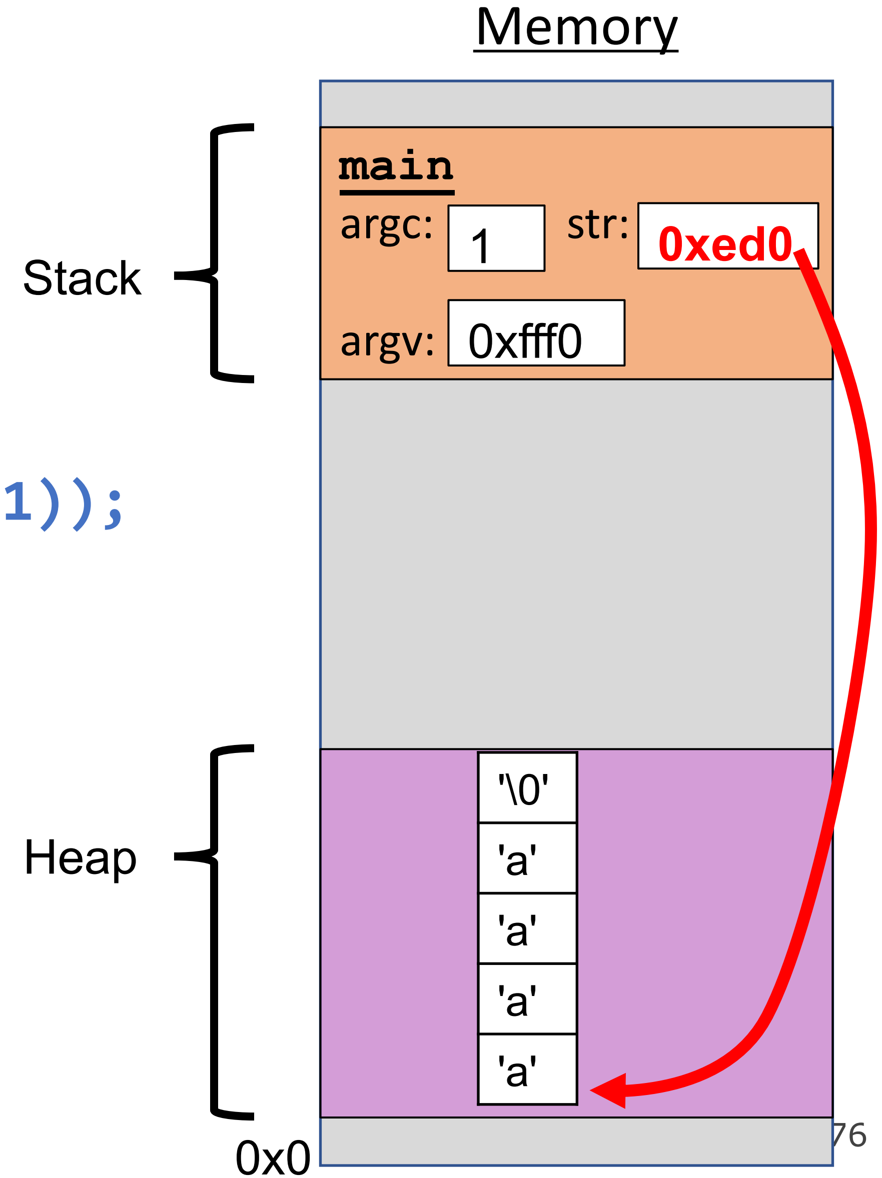
The Heap

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

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;
}
```



The Heap

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

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;
}
```



The Heap

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

```
int *array_of_multiples(int n, int mult) {
     ...
}
```

The Heap

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

```
int *array_of_multiples(int n, int mult) {
    int arr[n];    // on the stack! ☹
    for (int i = 0; i < n; i++) {
        arr[i] = mult * (i + 1);
    }
    return arr;
}
```

The Heap

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

```
int *array_of_multiples(int n, int mult) {
    int *arr = malloc(sizeof(int) * n);   // on the heap! ☺
    for (int i = 0; i < n; i++) {
        arr[i] = mult * (i + 1);
    }
    return arr;
}
```

strdup

strdup is a convenience function that returns a 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
```

Note that heap memory can be written to, so you can change the contents of a heap-allocated string.

```
str[0] = 'h';
```

calloc

calloc is a variant of malloc that zeros out allocated memory before returning a pointer to it.

```
void *calloc(size_t nmemb, size_t size)
```

- The parameters are slightly different than malloc. The first parameter is the number of elements you would like to allocate space for, and the second parameter is the size of each element.
- This was originally designed to allocate arrays, but works for any memory allocation. It just multiplies the two numbers together for the total size.
- calloc is more expensive than malloc because it zeros out memory. Use only when necessary!

The Heap

- `malloc` returns a pointer to a certain number of allocated bytes. It doesn't know or care whether it will be used as an array, a single block of memory, etc. It just allocates and returns bytes for you.
- If an allocation error occurs (e.g. out of heap memory!), malloc will return NULL. This is an important case to check, for robustness.

```
char *bytes = malloc(4);
assert(bytes != NULL);
```

- **assert** will crash the program if the provided condition is false. Here, if we receive a memory error (which is significant), we terminate the program.

Assert

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

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

Cleaning Up

- 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 *address on the heap for the memory you no longer need.*

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

- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

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!
```

Freeing Memory

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

```
char *bytes = malloc(4);     // e.g. 0xffff32
char *ptr = bytes;           // also 0xffff32!
…
free(bytes);                 // free memory at 0xffff32
…
free(ptr);          // free memory at 0xffff32 again! ☹
```

Freeing Memory

Make sure you free the same address you received from a previous allocation call. You cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
…
free(bytes + 1);            // cannot do this!
…
free(bytes);                // can only free bytes
```

Demo: Pig Latin

Plan For Today

- Miscellaneous Useful Topics
  - **const**
  - Structs
  - Ternary
- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- **Announcements**
- `Realloc`
- **Practice:** Pig Latin Part 2

Announcements

- Assignment 1 grades released later today
- Assignment 3 released tonight
  - Memory, pointers, and stack/heap
  - Emphasis on debugging

Debugging

1. **Observe** the bug.
2. Create a **simple**, **reproducible** input.
3. **Narrow** the search space.
4. **Analyze** using GDB and pictures.
5. **Devise and run experiments** until you identify the root cause.
6. **Modify** code to squash bug.

Starting with this assignment, we will only be able to help you with debugging in office hours if you fill out the signup form with information from these steps!

Plan For Today

- Miscellaneous Useful Topics
  - **const**
  - Structs
  - Ternary
- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- Announcements
- **`Realloc`**
- **Practice:** Pig Latin Part 2

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!

realloc

- Another advantage of heap memory is that you can request more for an existing allocation if you need it.
- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.

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

- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.
- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

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

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

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

realloc

- realloc only accepts pointers that were previously returned 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.

Demo: Pig Latin Part 2

Why We ❤️ The Stack

- **It is fast**. Your program already has that memory reserved for it!
- **It is convenient**. Memory is handled automatically, and is fast because old memory is left in place and marked as usable for future function calls.
- **It is safe**. You specify variable types, and the compiler can therefore do checks on the data. We'll see later this is not necessarily true on the heap.

Why We ❤️ The Heap

- **It is plentiful**. The stack has at most 8MB by default. The heap can provide more on demand!
- **Allocations are resizable.** Unlike on the stack, if you allocate something (e.g. an array), you can change the size of it later using realloc.
- **Scope.** The memory is not cleaned up when its function exits; instead, you control when the memory is freed.

Recap

- Miscellaneous Useful Topics
- The Stack
- The Heap and Dynamic Memory
- **Practice:** Pig Latin
- Announcements
- `Realloc`
- **Practice:** Pig Latin Part 2

Next time: C Generics