COMP201
Computer Systems & Programming

Lecture #8 – The Stack and The Heap
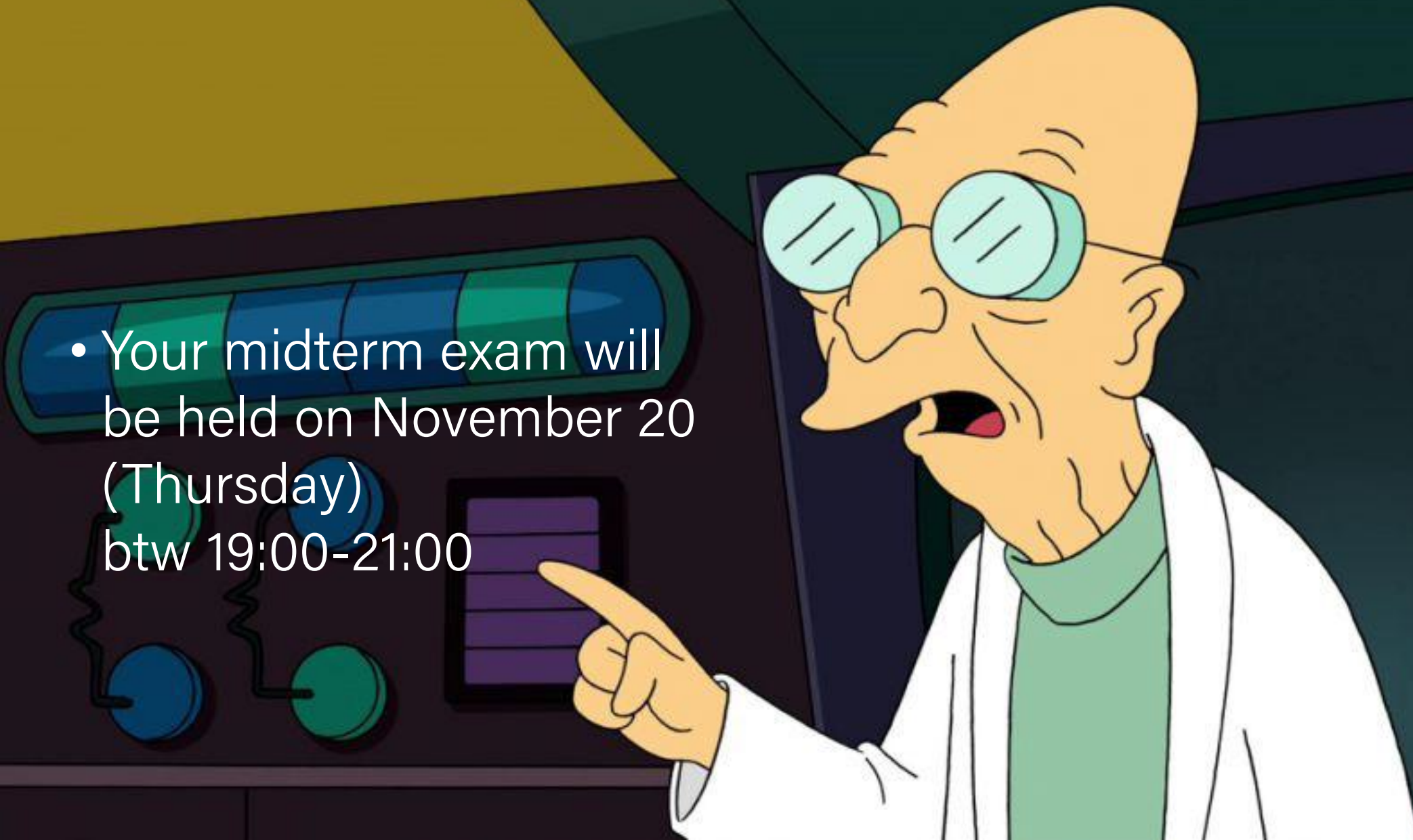
Aykut Erdem // Koç University // Fall 2025

Good news, everyone!

- Your midterm exam will be held on November 20 (Thursday) btw 19:00-21:00

Pointers Practice
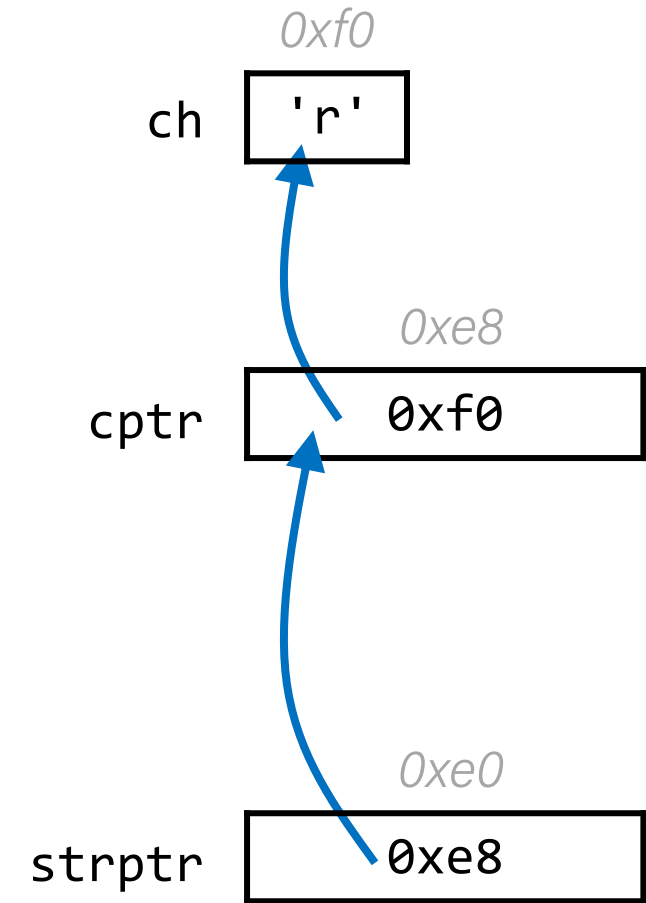
* Wars: Episode I (of 2)

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

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

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

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

0xf0

ch `'r'`

0xe8

cptr `0xf0`

0xe0

strptr `0xe8`

* Wars: Episode II (of 2)

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

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

char *cptr = &ch;

char **strptr = &cptr;
```
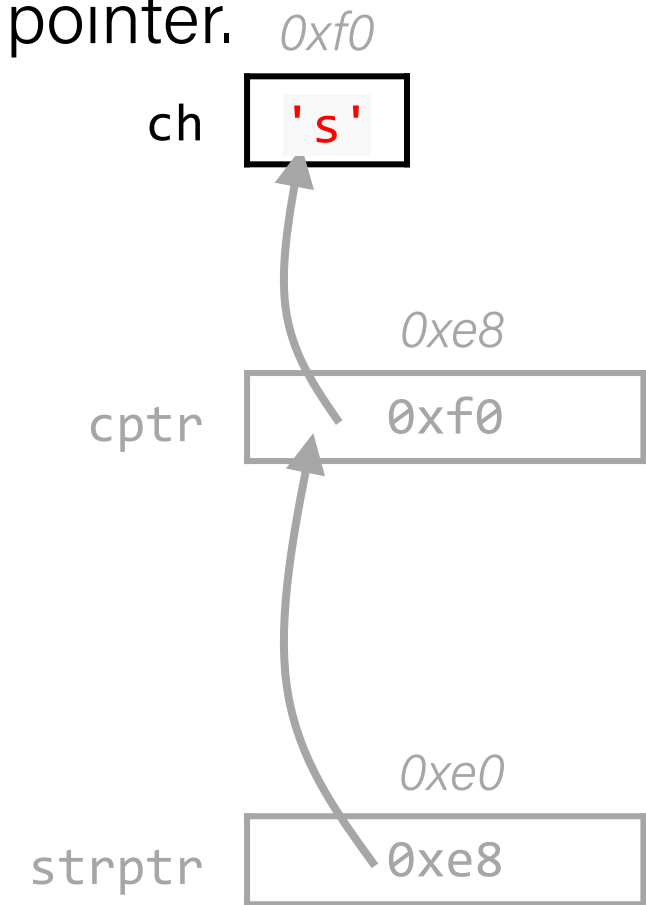
Increment value stored in ch

0xf0

ch   `'s'`

0xe8

cptr   `0xf0`

0xe0

strptr   `0xe8`

5

* Wars: Episode II (of 2)

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

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


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



char **strptr = &cptr;
```

Increment value stored in ch

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

0xf0

ch   `'t'`

0xe8

cptr   `0xf0`

0xe0

strptr   `0xe8`

* Wars: Episode II (of 2)

In <u>reading values from/storing values</u>, * **dereferences** a pointer.
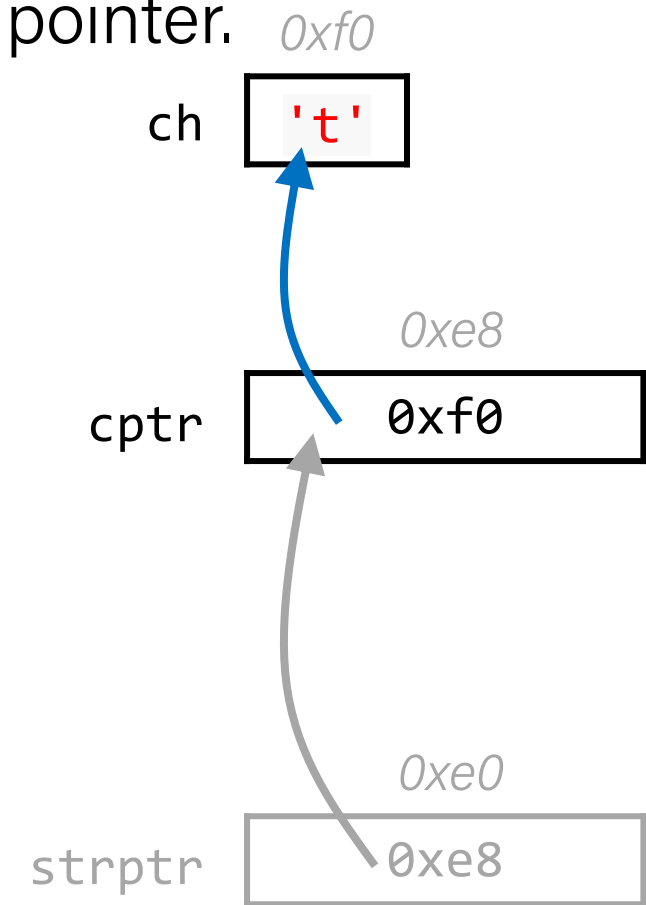
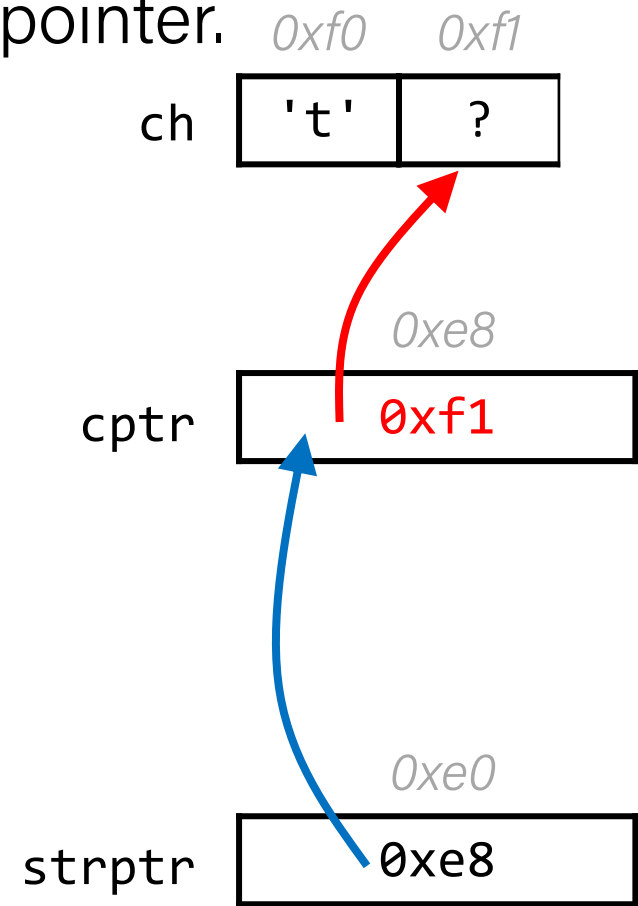*0xf0*   *0xf1*

```
char ch = 'r';
ch = ch + 1;
```
Increment value stored in ch

ch | 't' | ? |

```
char *cptr = &ch;
*cptr = *cptr + 1;
```
Increment value stored at
memory address in cptr
(increment char **pointed to**)

*0xe8*

cptr | 0xf1 |

```
char **strptr = &cptr;
*strptr = *strptr + 1;
```
Increment value stored at
memory address in cptr
(increment address **pointed to**)

0xe0

strptr | 0xe8 |

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.

0xffe800   *0xffe808*

a [10   ]   p [0xffe800   ]

0xffe804   *0xffe810*

b [20   ]   q [0xffe804   ]

🤔

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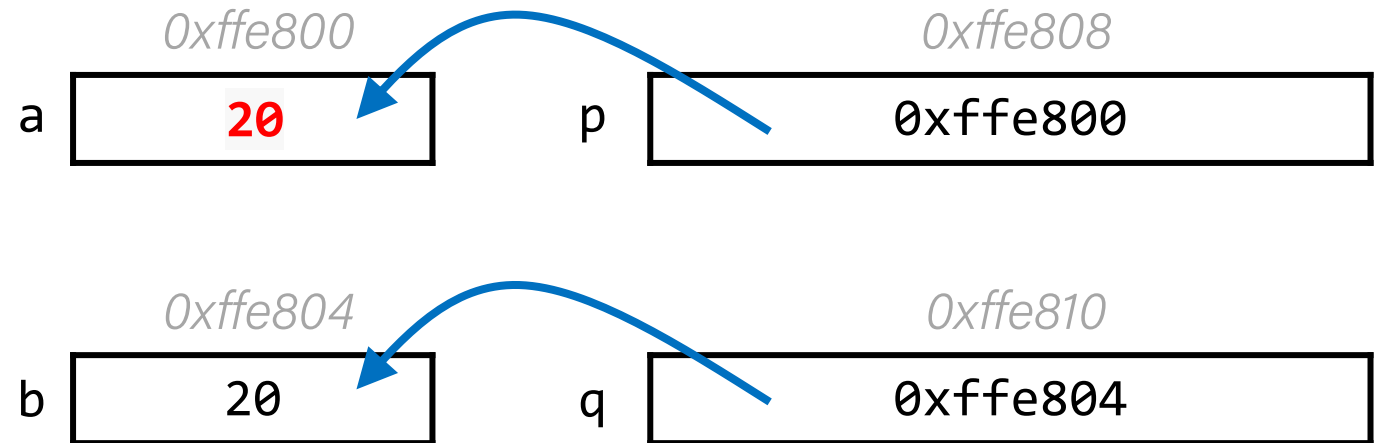
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

0xffe800   *0xffe808*

a | **20** | p | 0xffe800 |

0xffe804   *0xffe810*

b | 20 | q | 0xffe804 |

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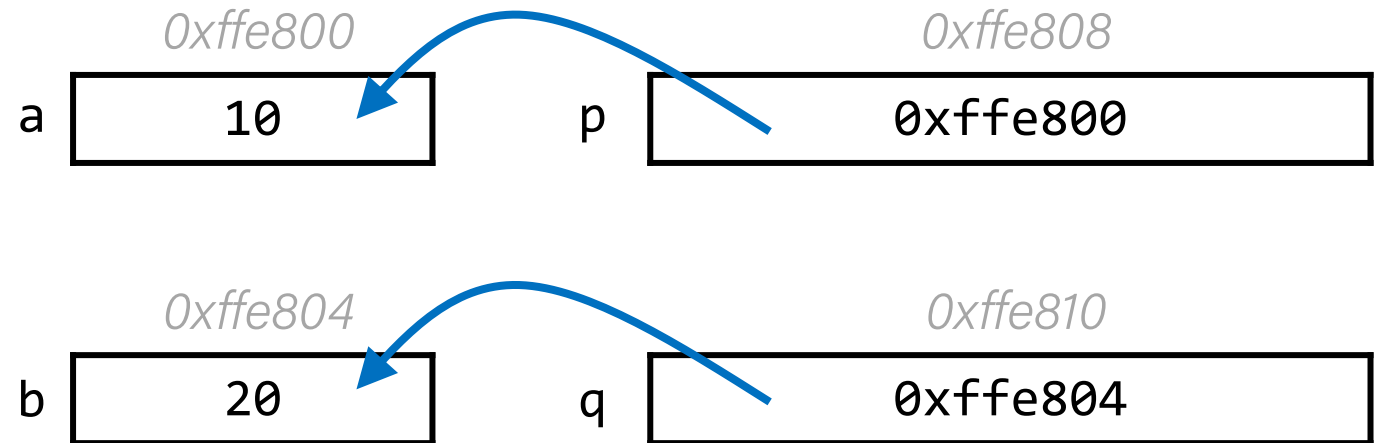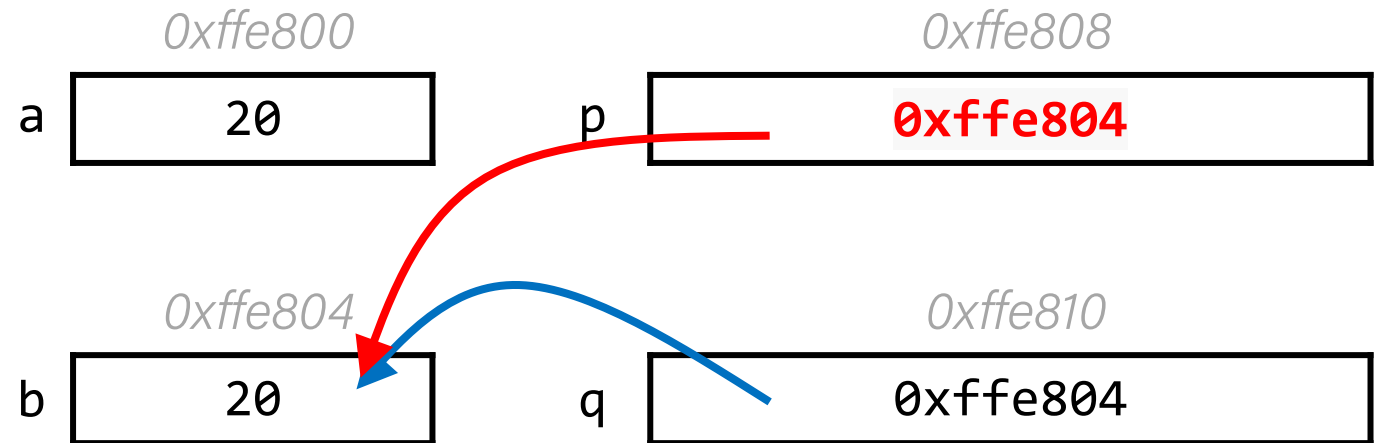
- Lines 2-5: Draw a diagram.
- Line 7: Update your diagram.
- Line 8: Update your diagram.

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
```

STACK

Address	Value
	...
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	...

str
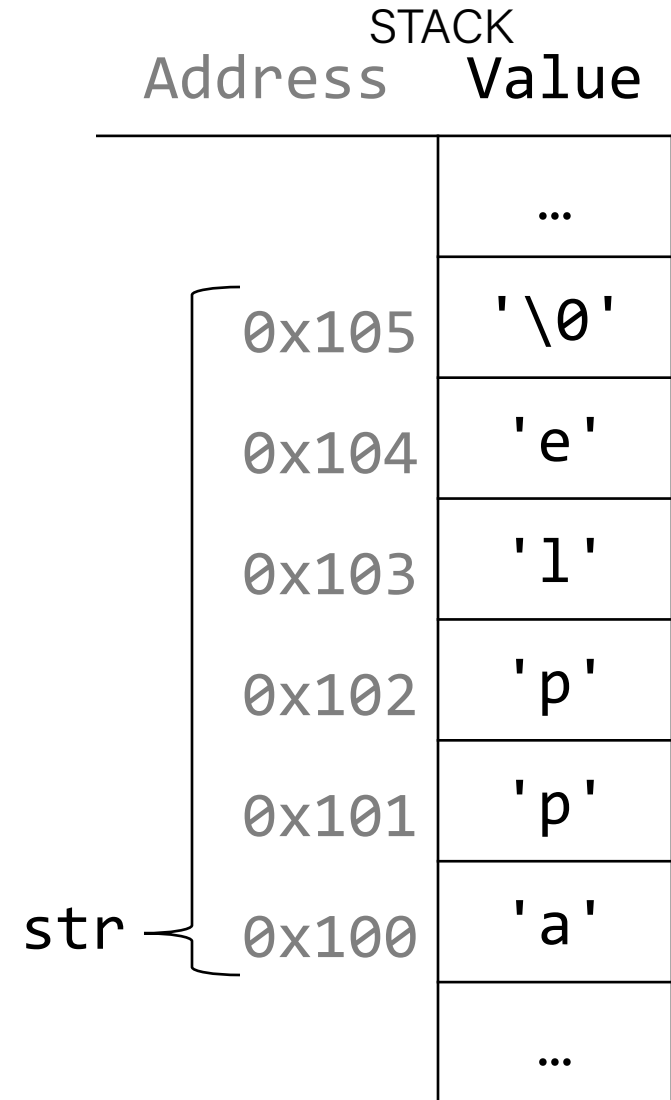
Arrays

An array variable refers to an entire block of memory. You cannot reassign an existing array to be equal to a new array.
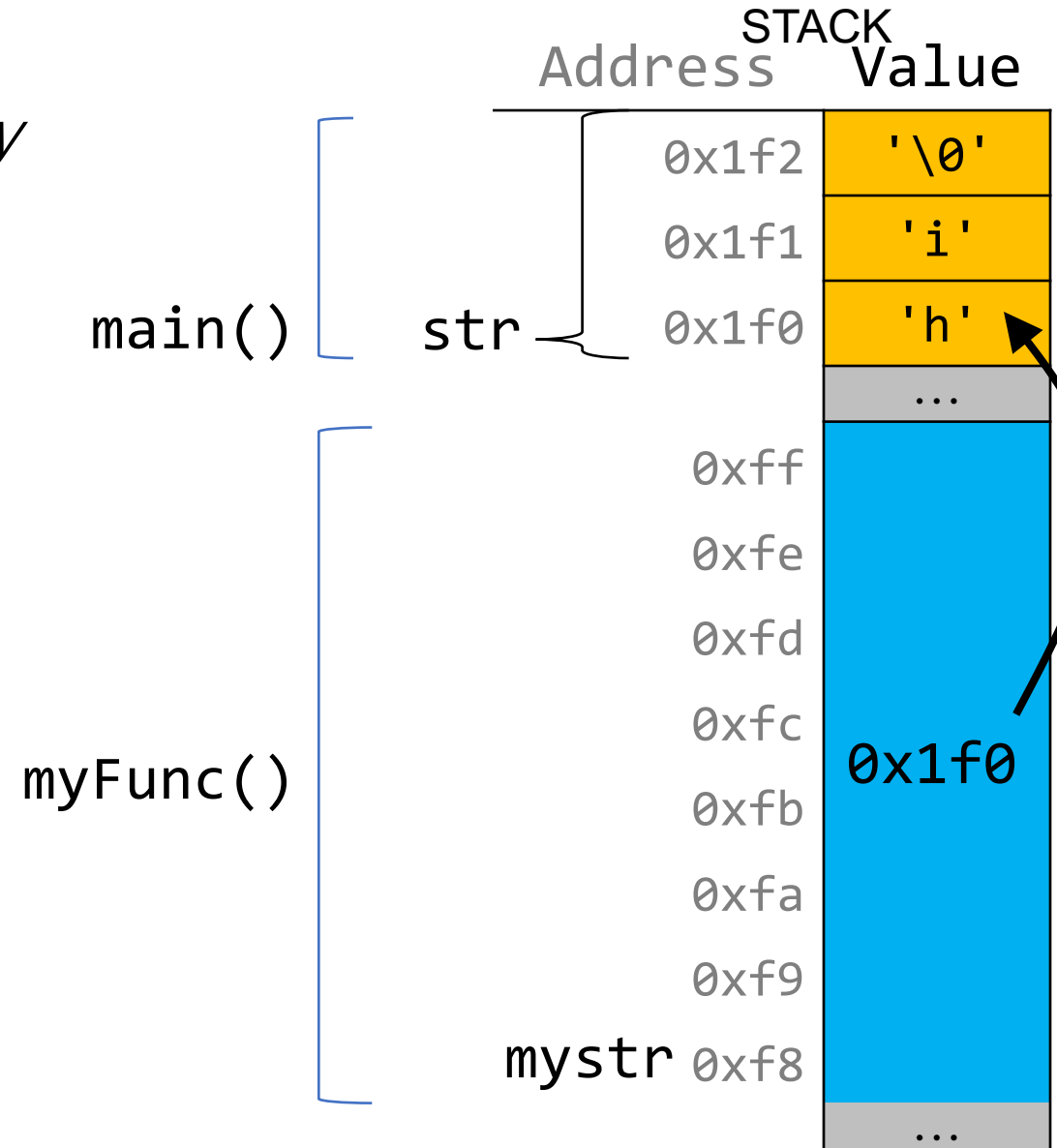
```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};
nums = nums2; // not allowed!
```

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);
    ...
}
```

STACK

Address	Value
0x1f2	'\0'
0x1f1	'i'
0x1f0	'h'
	...
0xff	
0xfe	
0xfd	
0xfc	
0xfb	0x1f0
0xfa	
0xf9	
0xf8	
	...

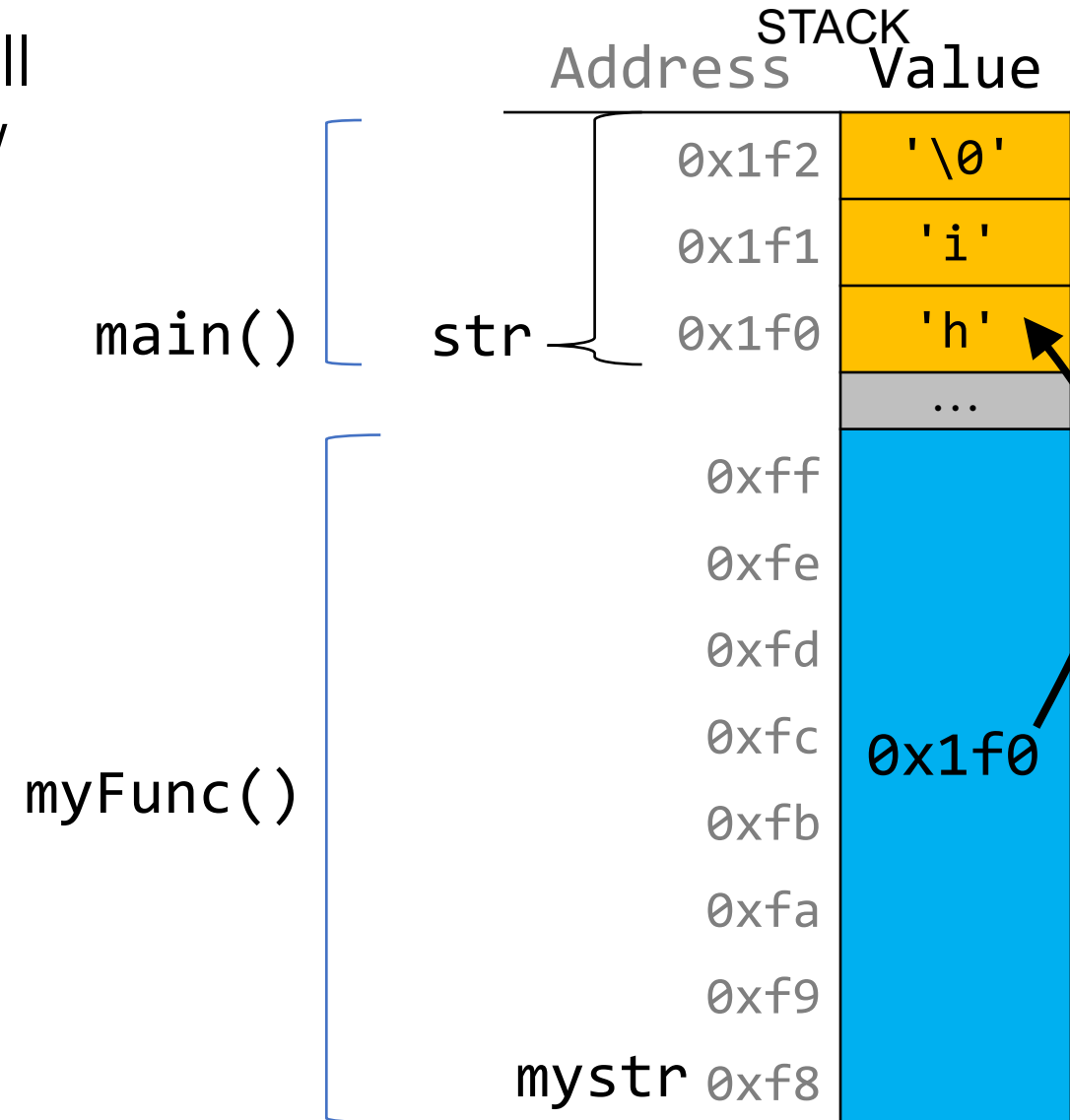main()   str

myFunc()

mystr

17

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);
    ...
}
```

STACK

Address	Value
0x1f2	'\0'
0x1f1	'i'
0x1f0	'h'
	...
0xff	
0xfe	
0xfd	
0xfc	0x1f0
0xfb	
0xfa	
0xf9	
0xf8	

main() str

myFunc()

mystr

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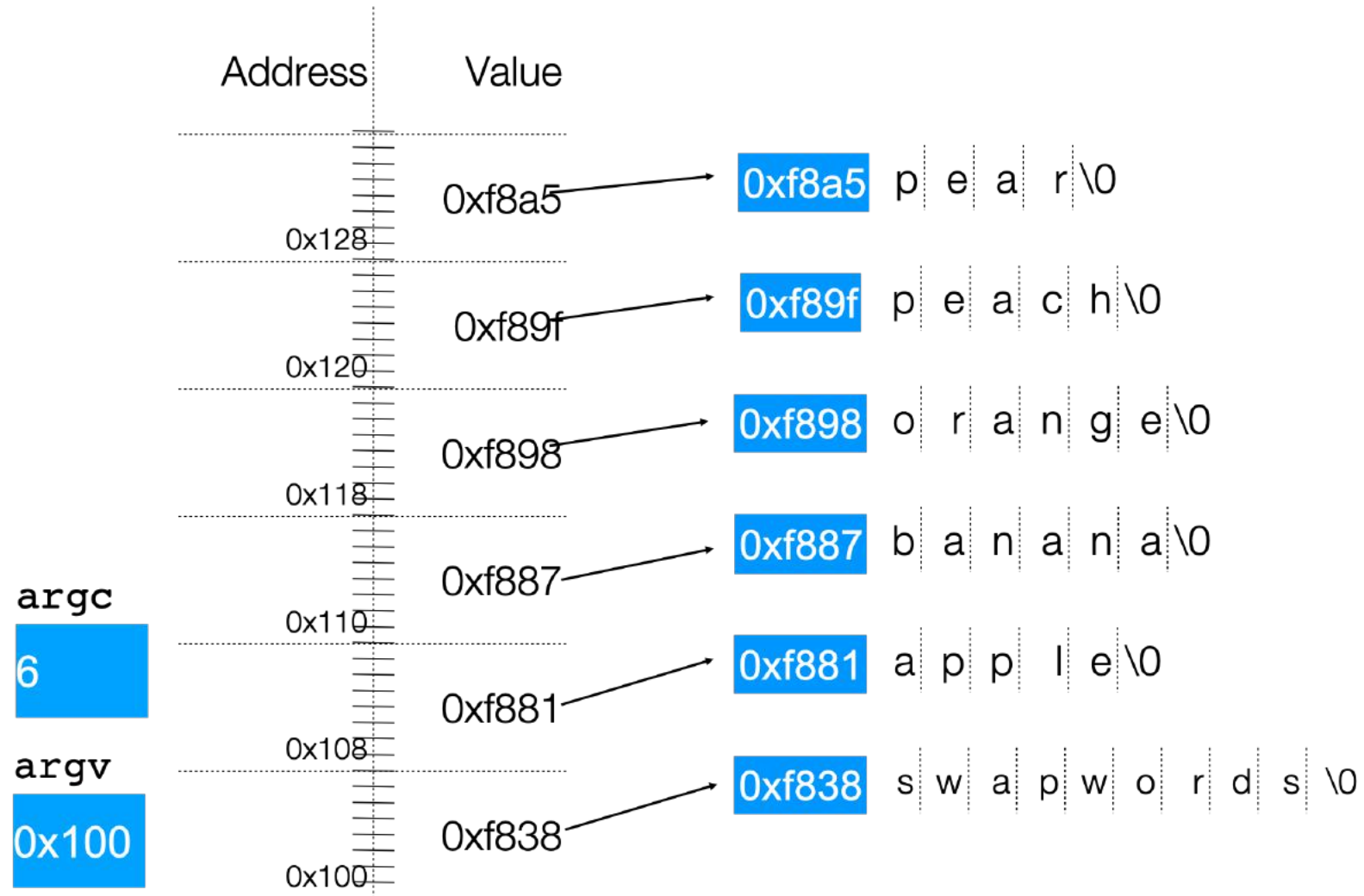
```
char *stringArray[5];   // space to store 5 char *s
```

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

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

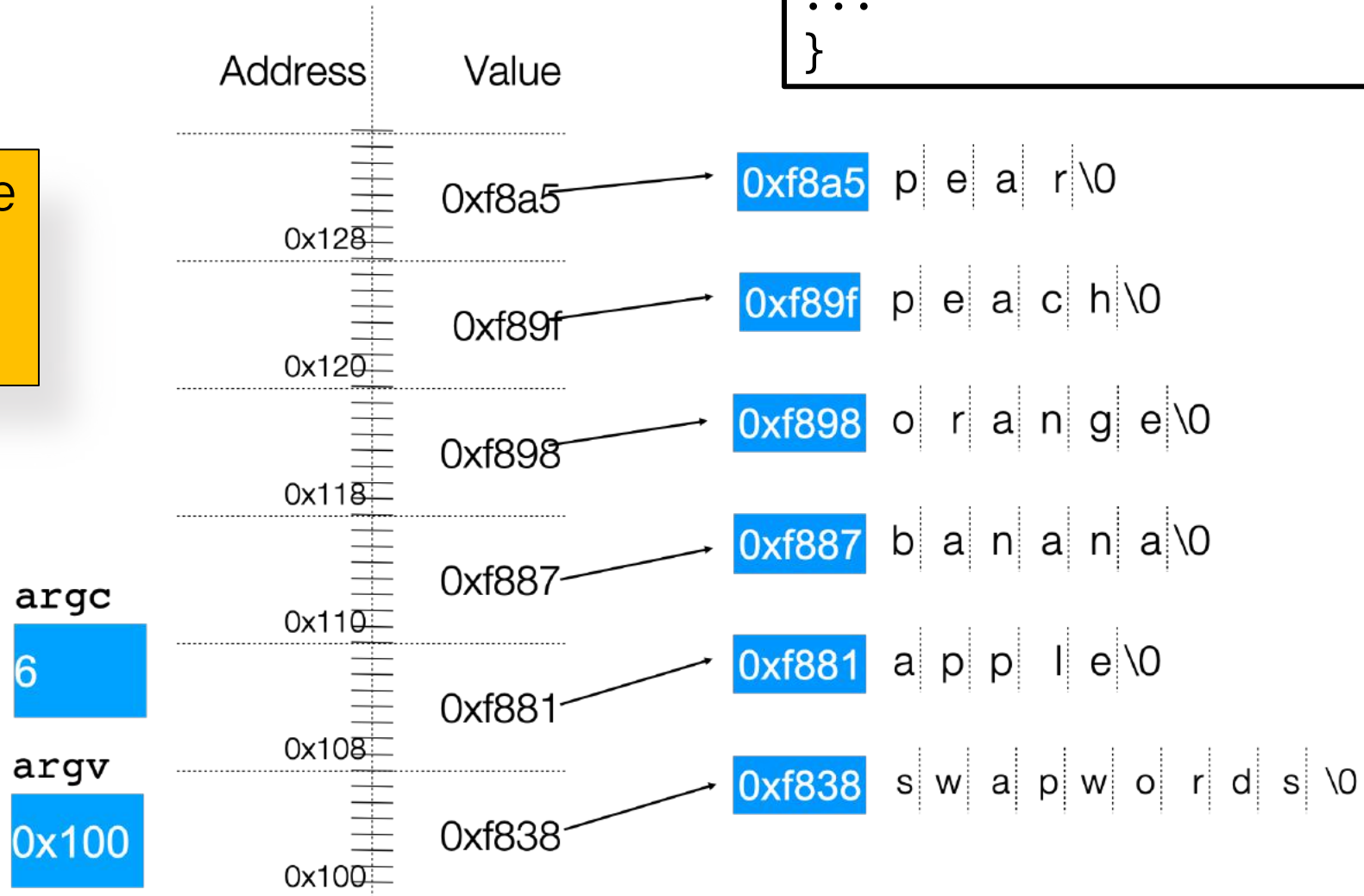# Arrays Of Pointers

./swapwords apple banana orange peach pear

Arrays Of Pointers

./swapwords apple banana orange peach pear

```
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'
	…

28

Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …            // e.g. 0xff0
int *nums1 = nums + 1;   // e.g. 0xff4
int *nums3 = nums + 3;   // e.g. 0xffc

printf("%d", *nums);     // 52
printf("%d", *nums1);    // 23
printf("%d", *nums3);    // 34
```

STACK

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

Pointer Arithmetic

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

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

// both of these add two places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff2.
char thirdLetter = str[2];       // 'p'
char thirdLetter = *(str + 2);    // 'p'
```

DATA SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

Pointer Arithmetic

Pointer arithmetic with two pointers does *not* give the byte difference. Instead, it gives the number of places they differ by.

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

STACK

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

Pointer Arithmetic

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

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

// How does it know to print out just the 4 bytes at xPtr?
printf("%d", *xPtr);     // 2
```

Pointer Arithmetic

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

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

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;

char str[6];
strcpy(str, "COMP201");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Pointer Arithmetic

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

- For this reason, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type.

Pointer arithmetic

Array indexing is "syntactic sugar" for pointer arithmetic:

$$\texttt{ptr + i} \qquad \Leftrightarrow \qquad \texttt{\&ptr[i]}$$

$$\texttt{*(ptr + i)} \qquad \Leftrightarrow \qquad \texttt{ptr[i]}$$

⚠️ 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

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

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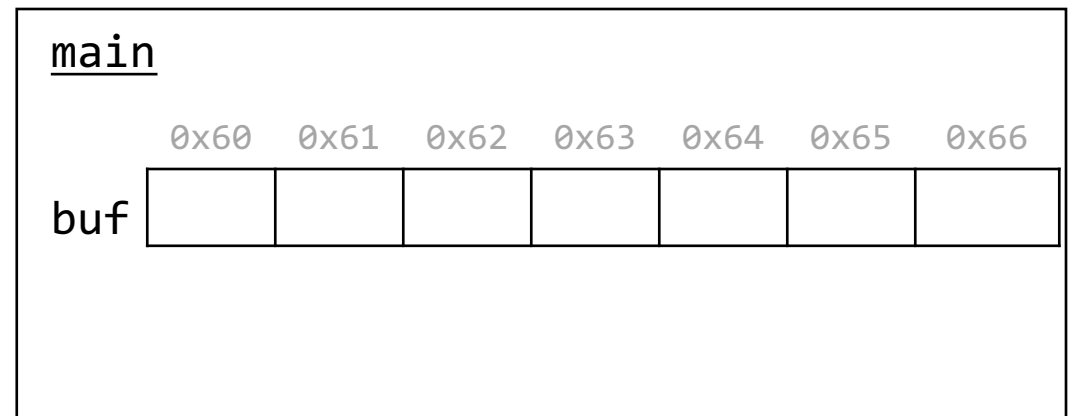- If no errors, what is printed?

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

🤔

Example: Pointer arithmetic

```
1  void func(char *str) {
2      str[0] = 'S';
3      str++;
4      *str = 'u';
5      str = str + 3;
6      str[-2] = 'm';
7  }

8  int main(int argc, const char *argv[]) {
9      char buf[] = "Monday";
10     printf("before func: %s\n", buf);
11     func(buf);
12     printf("after  func: %s\n", buf);
13     return 0;
14 }
```

func

str

main

| 0x60 | 0x61 | 0x62 | 0x63 | 0x64 | 0x65 | 0x66 |

buf

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

Code study: strncpy

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
1 char *strncpy(char *dest, const char *src, size_t n) {
2     size_t i;
3     for (i = 0; i < n && src[i] != '\0'; i++)
4         dest[i] = src[i];
5     for ( ; i < n; i++)
6         dest[i] = '\0';
7     return dest;
8 }
```

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:

```
1 char *strncpy(char *dest, const char *src, size_t n) {
2     size_t i;
3     for (i = 0; i < n && src[i] != '\0'; i++)
4         dest[i] = src[i];
5     for ( ; i < n; i++)
6         dest[i] = '\0';
7     return dest;
8 }
```

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

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

dest	
src	
n	5
i	

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

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 }
```
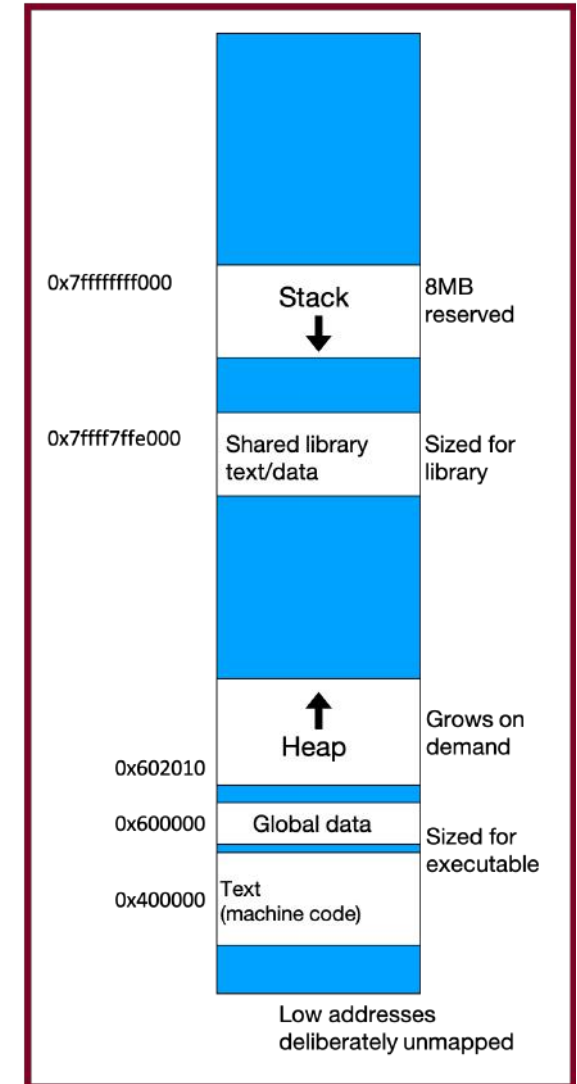
What is stored in each variable?

🤔

Bonus: Tricky addresses

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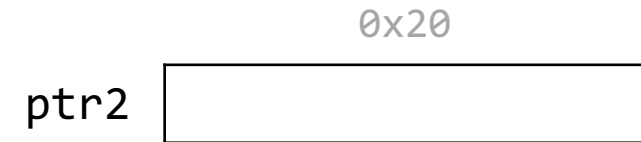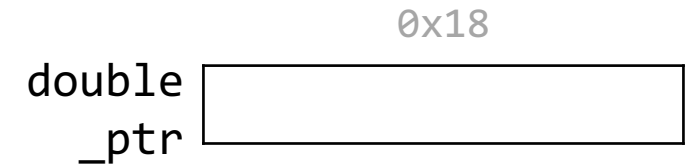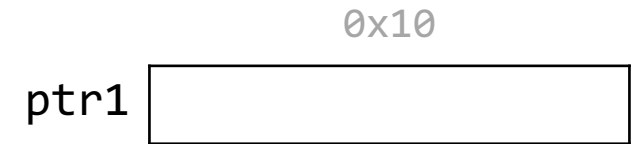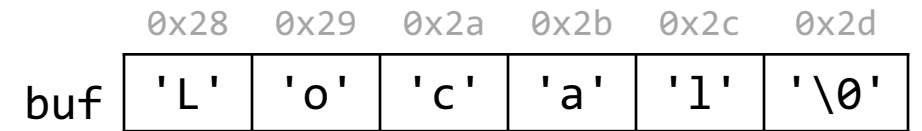
While Line 10 raises a compiler warning, functionally it will still work—because pointers are **addresses**.

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.

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;
}
```
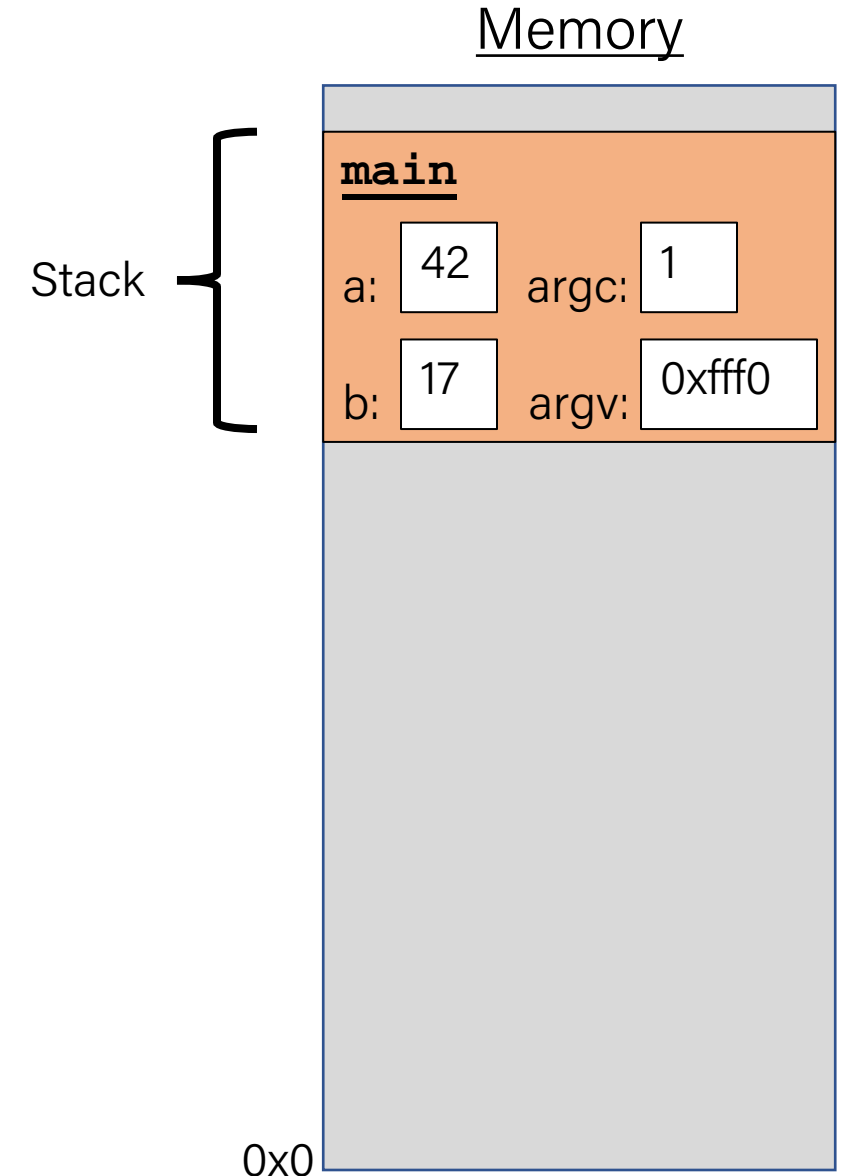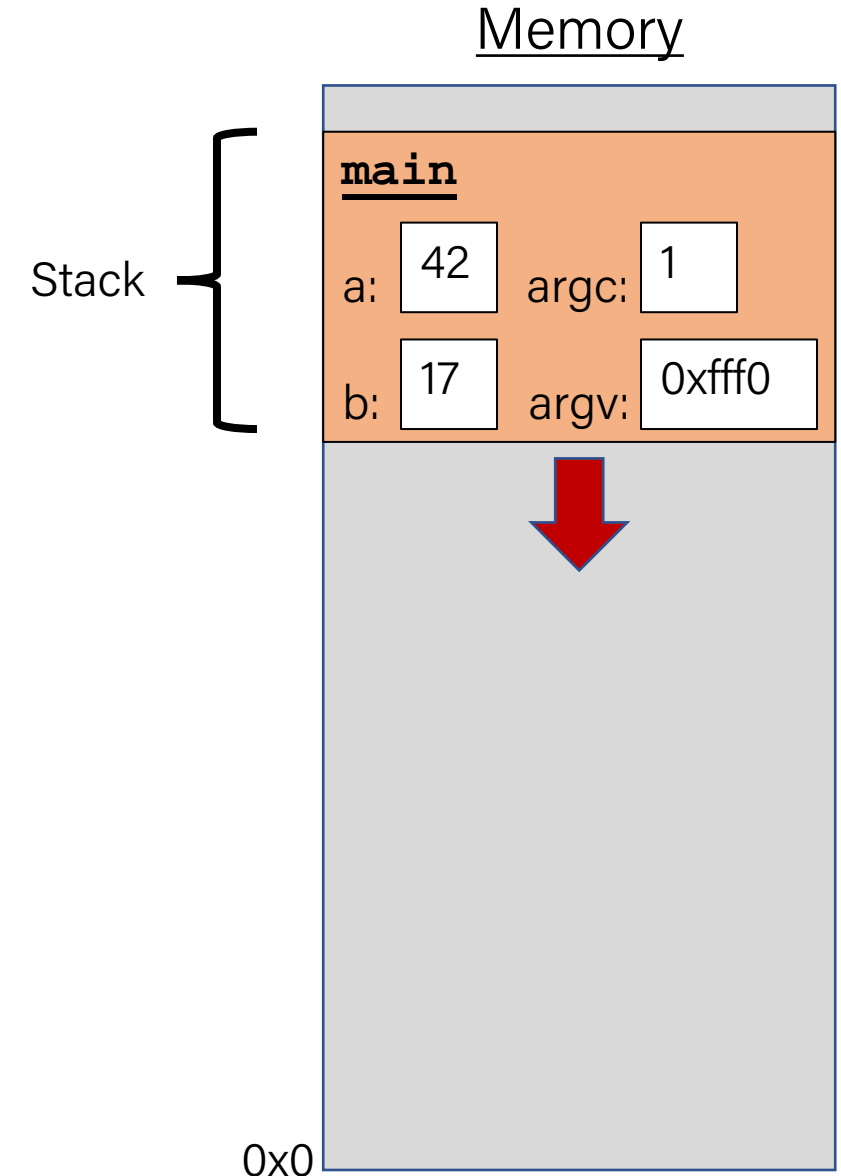
Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

46

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;
}
```
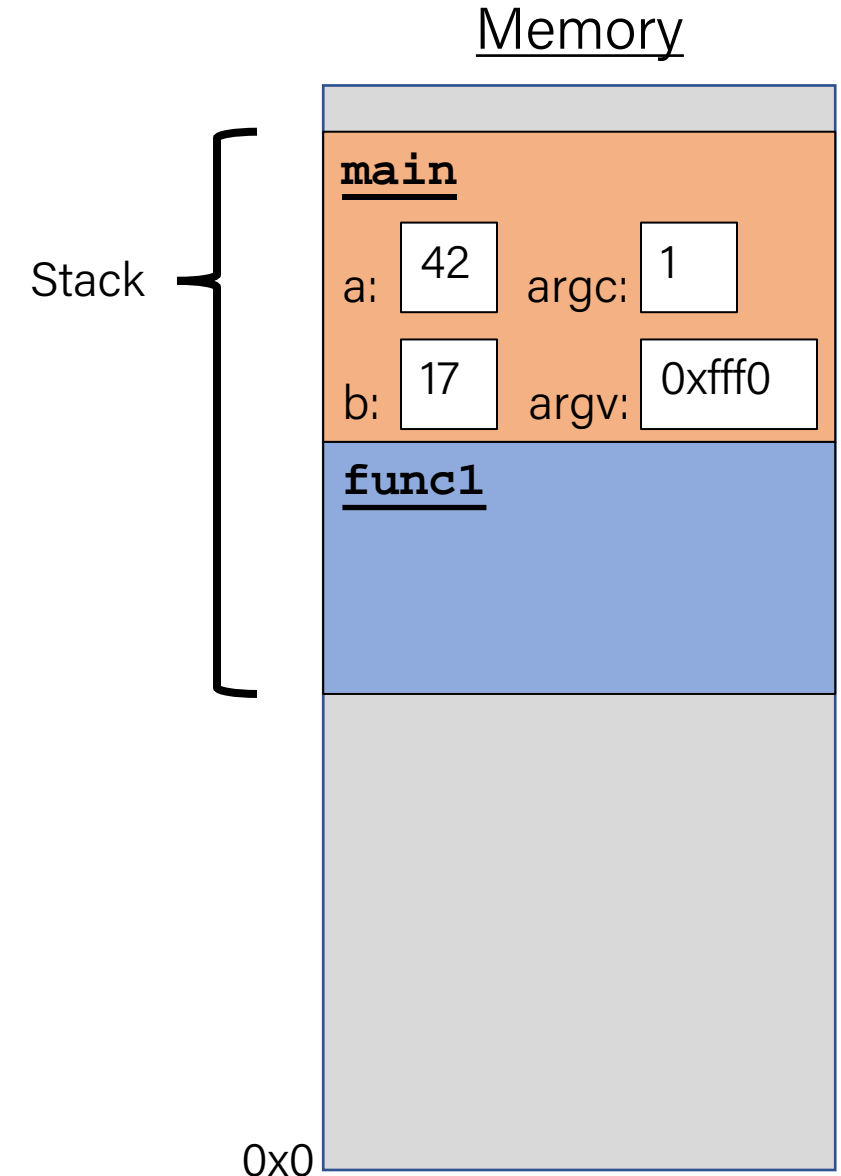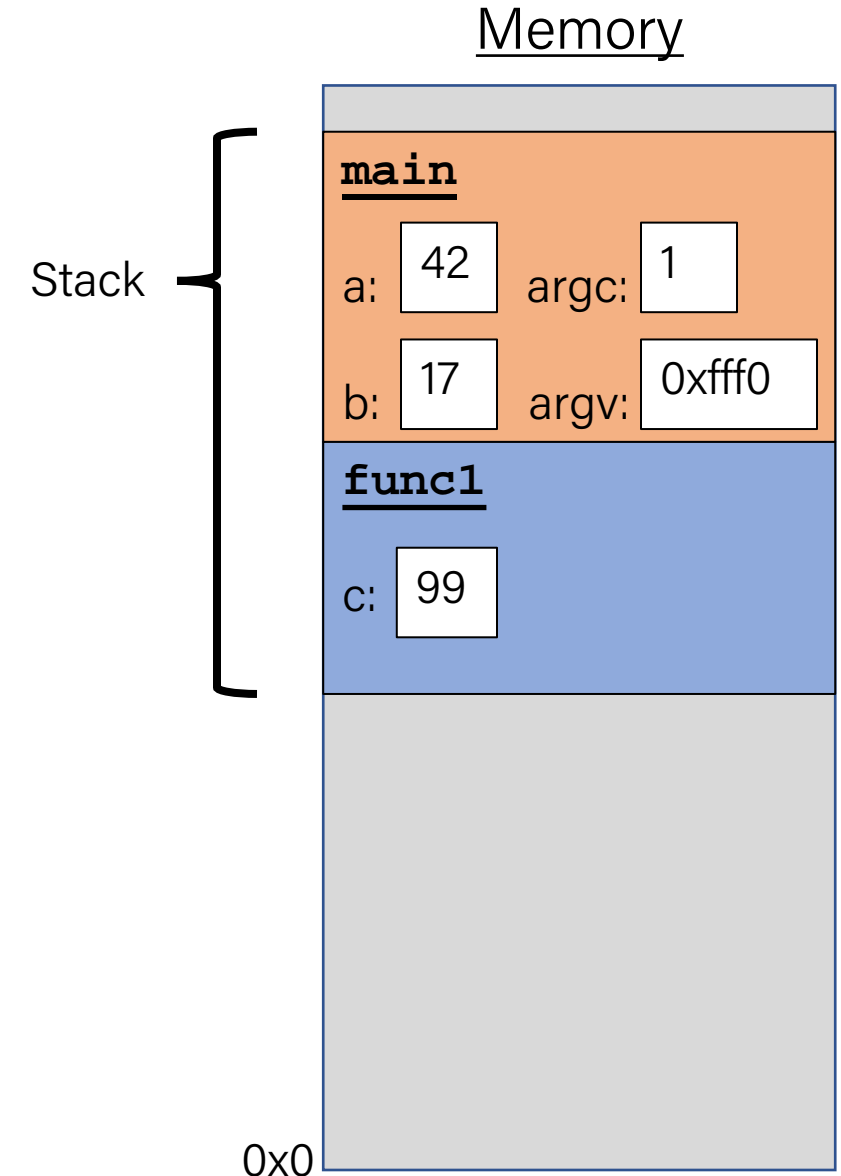

Memory

Stack

main

a: 42   argc: 1

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

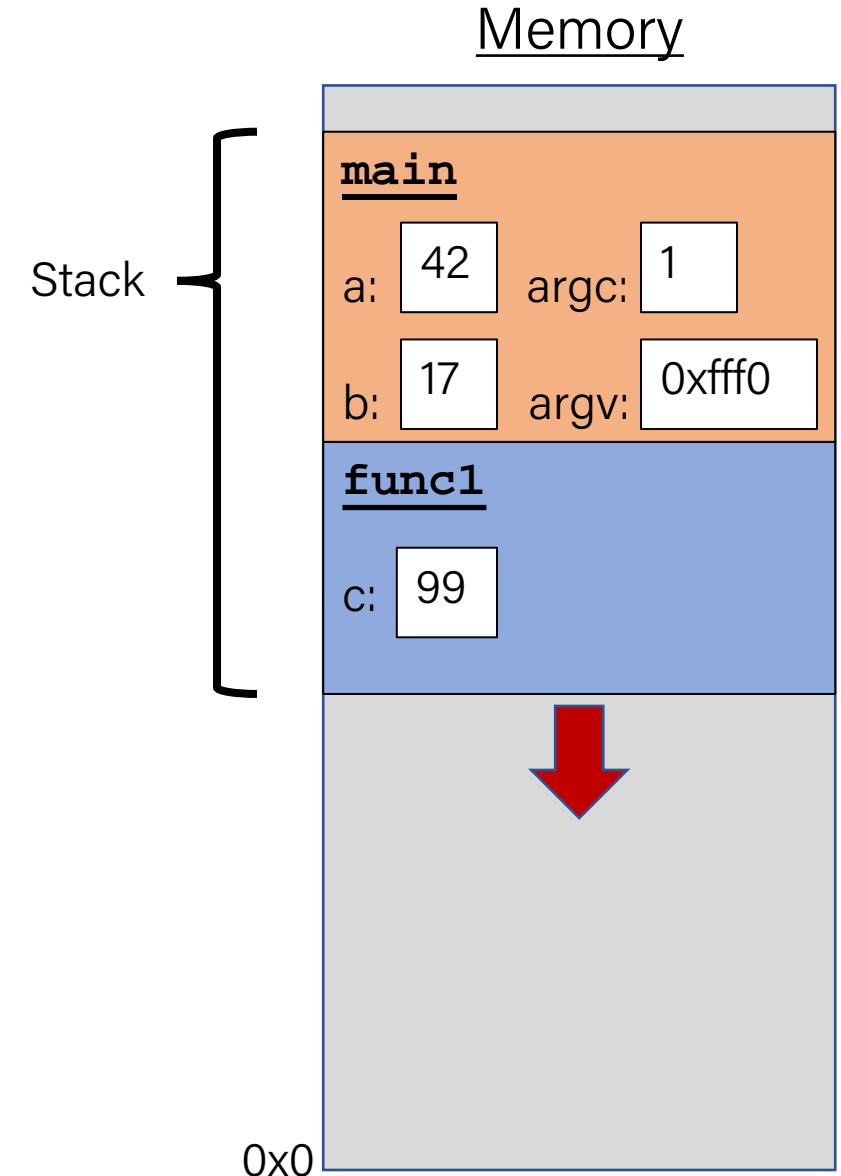# The Stack

```
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

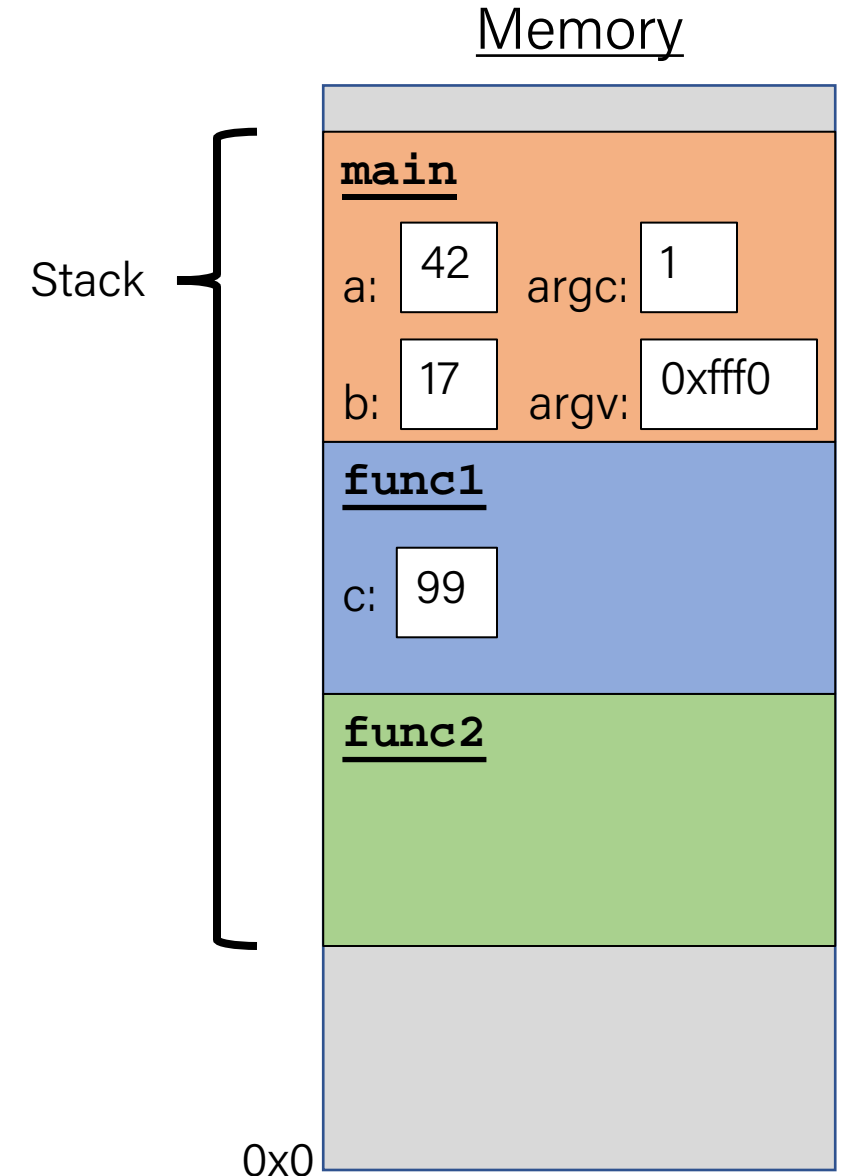# The Stack

```
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
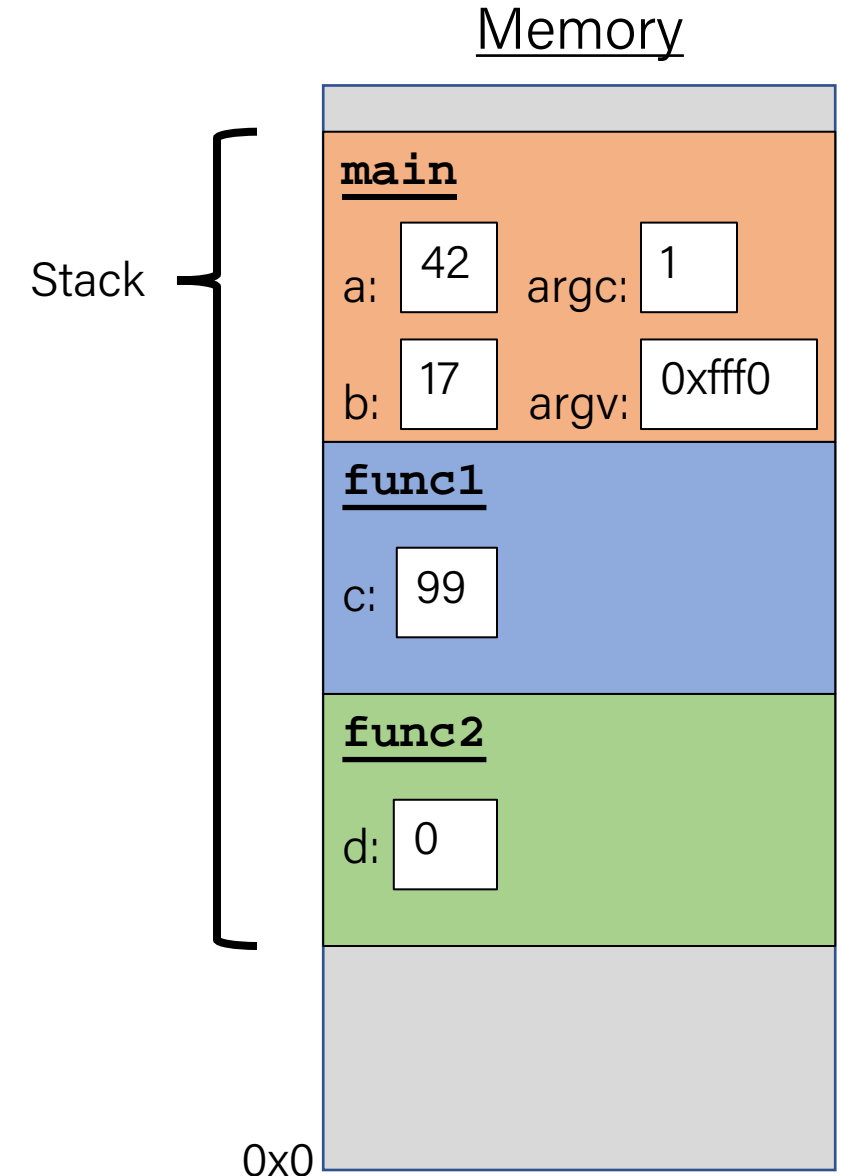
0x0

50

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

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;
}
```
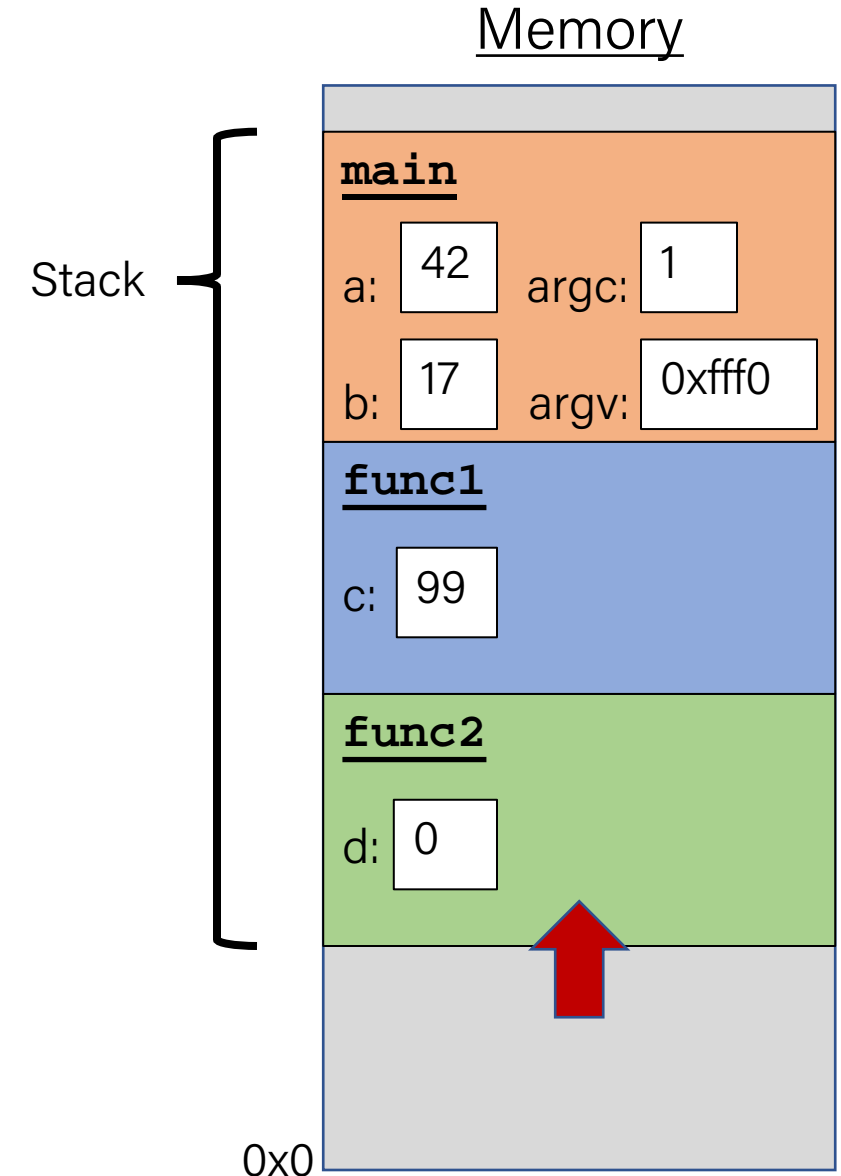
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

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;
}
```
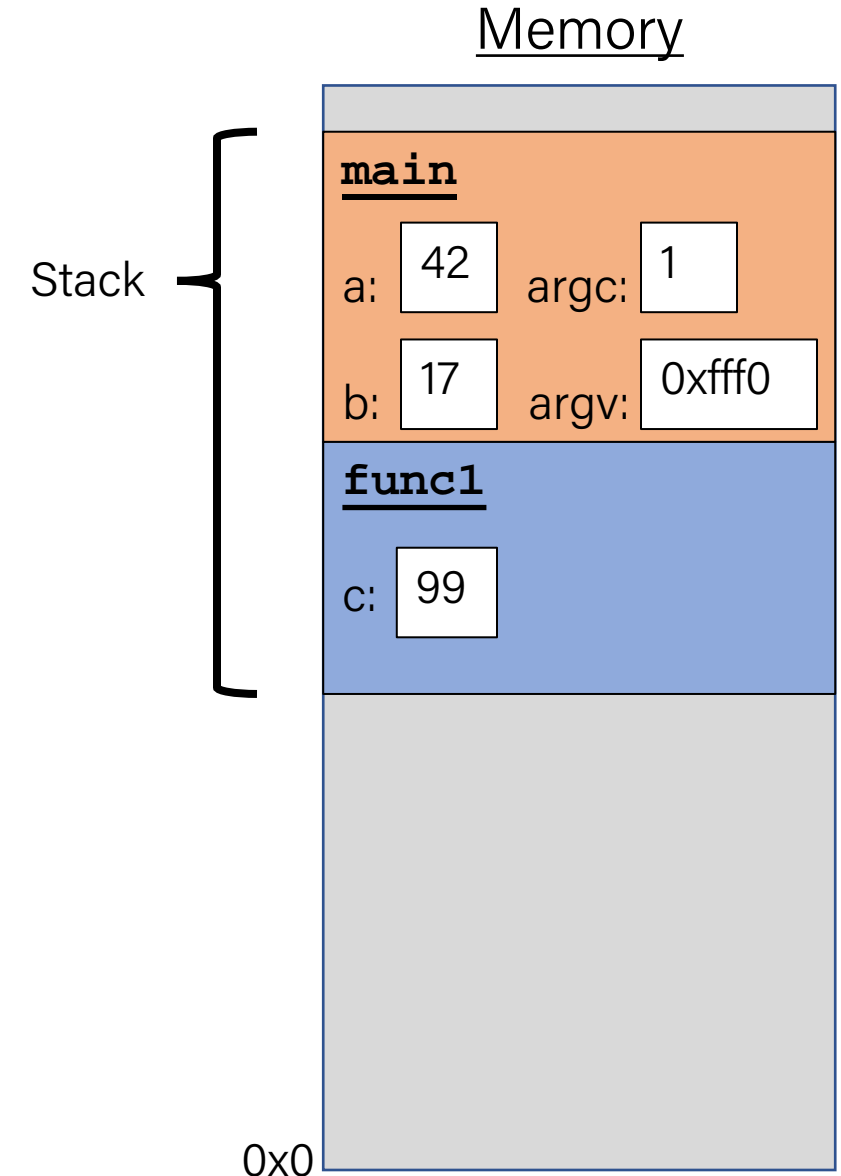
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

d: 0

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;
}
```
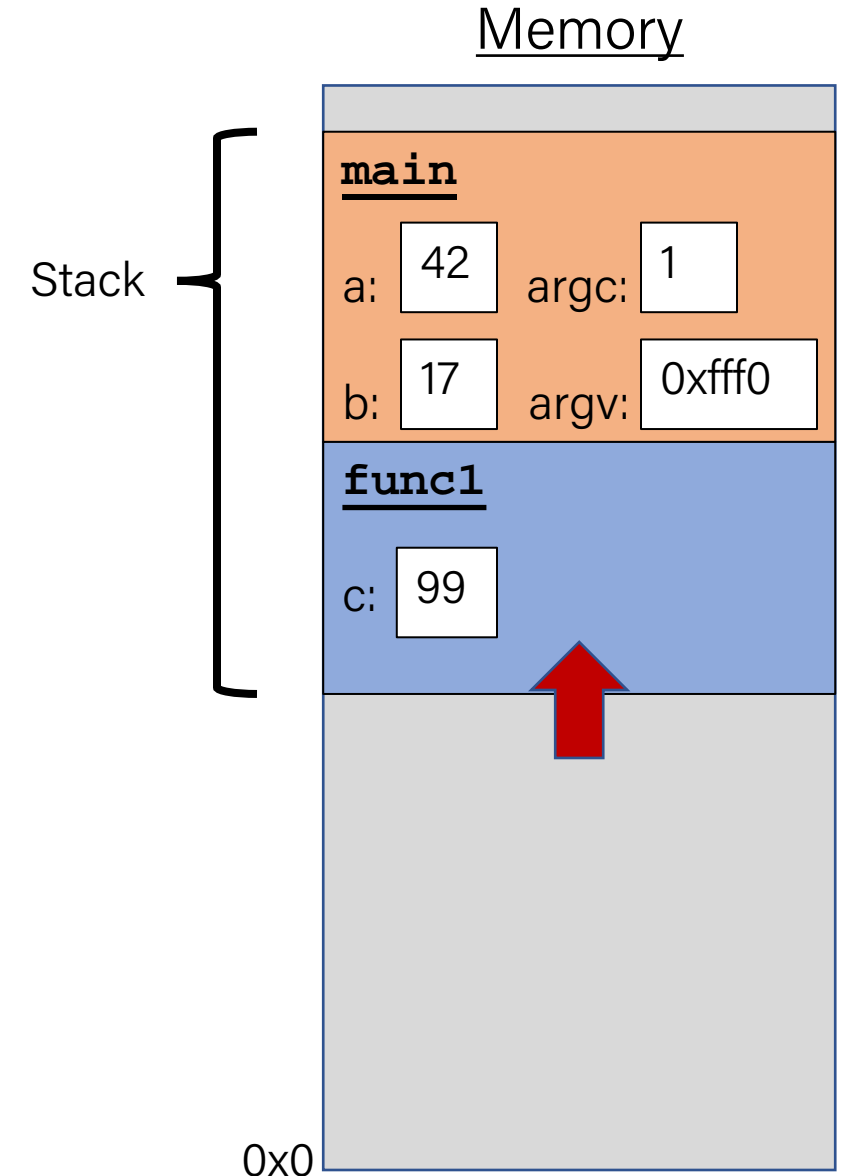
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

func2

d: 0

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;
}
```
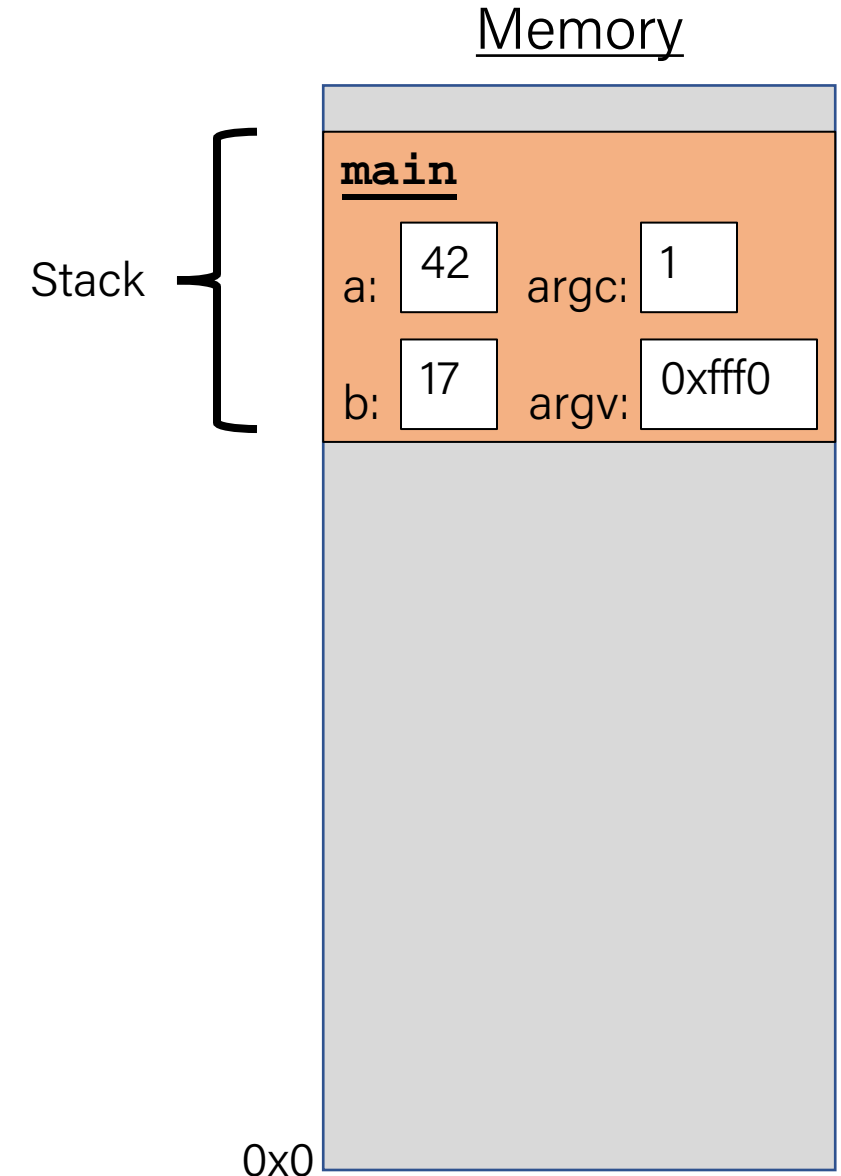
Memory

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;
}
```
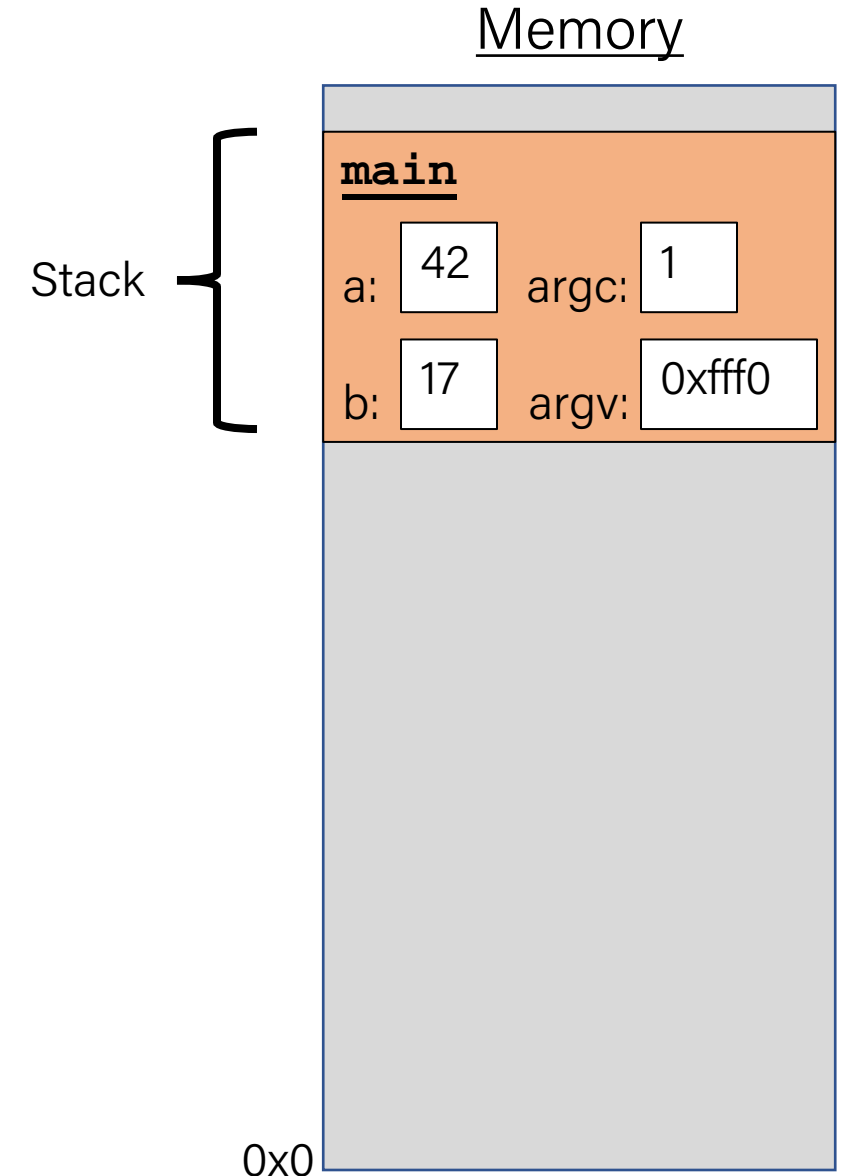
Memory

Stack

main

a: 42   argc: 1

b: 17   argv: 0xfff0

func1

c: 99

0x0

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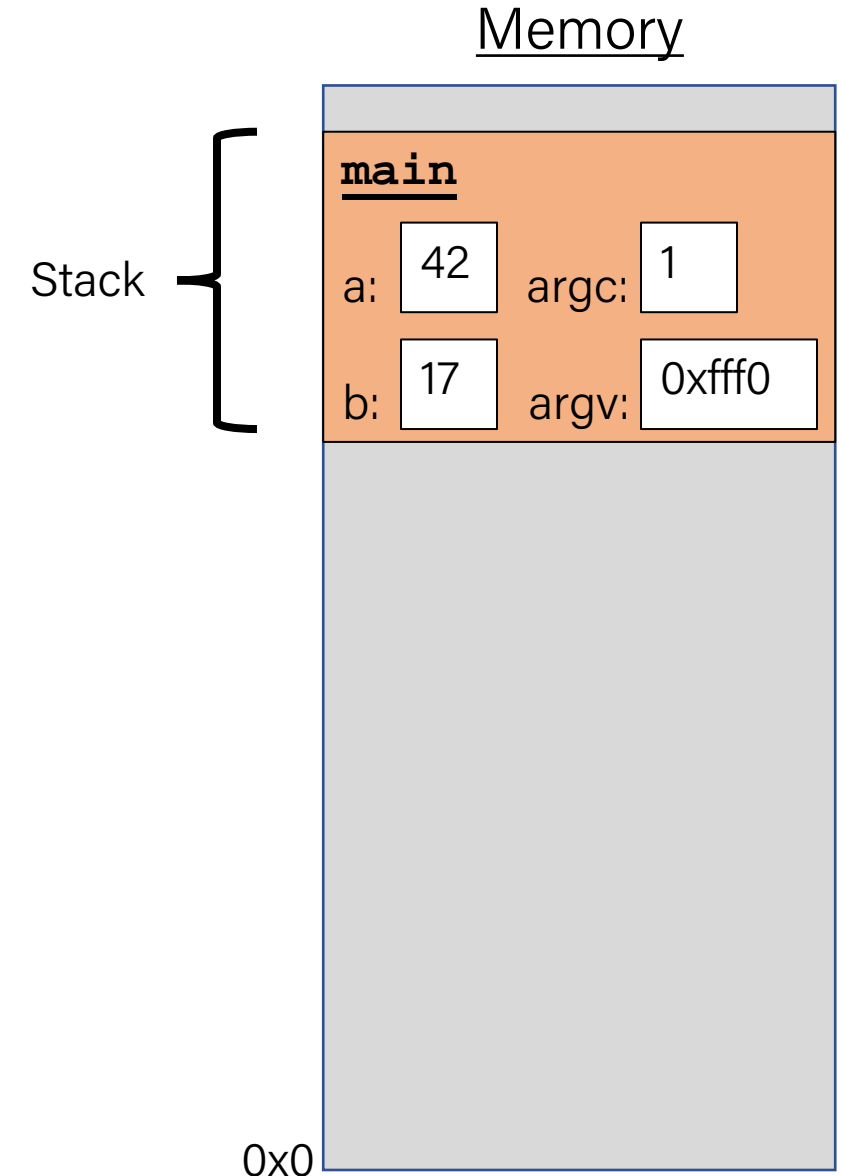
```
void func2() {
    int d = 0;
}

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

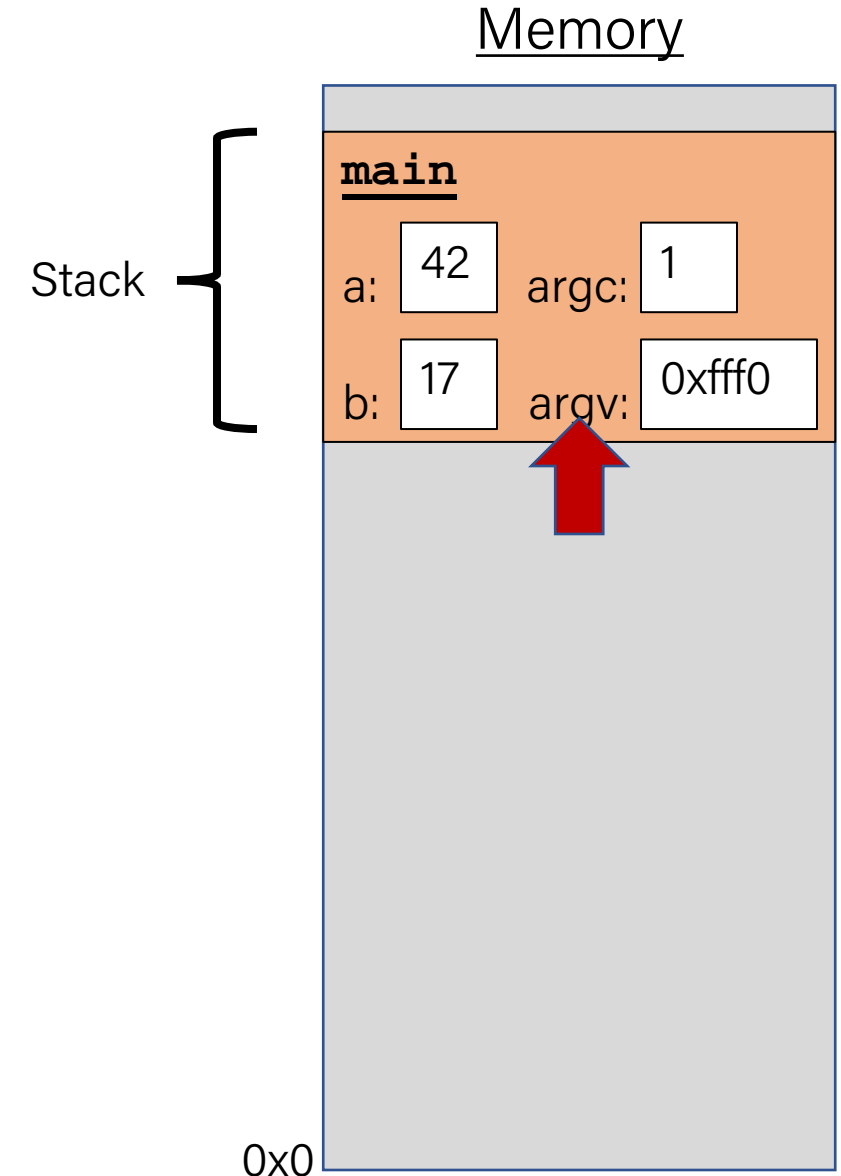# The Stack

```
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

```
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;
}
```
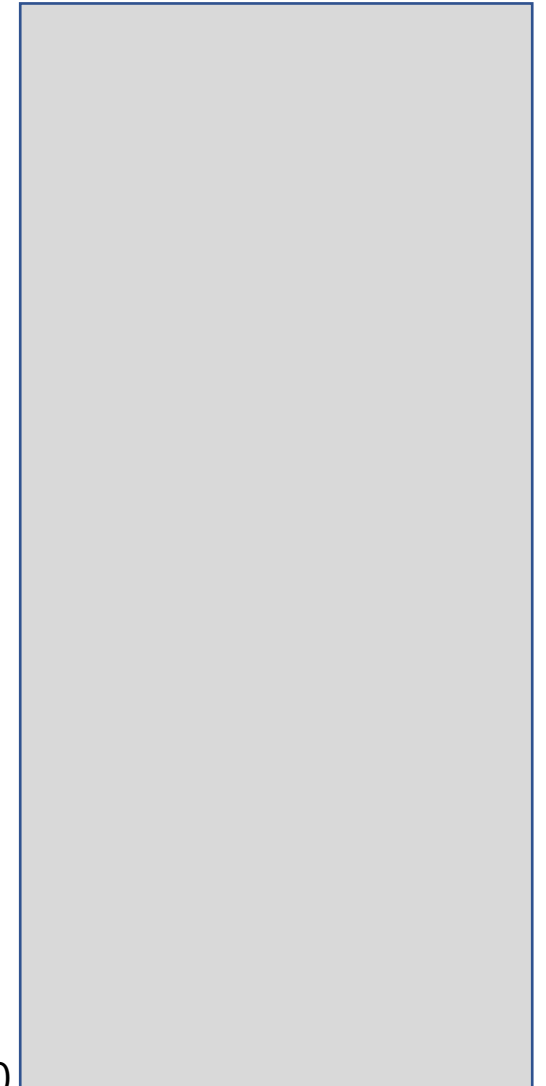
Memory

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;
}
```
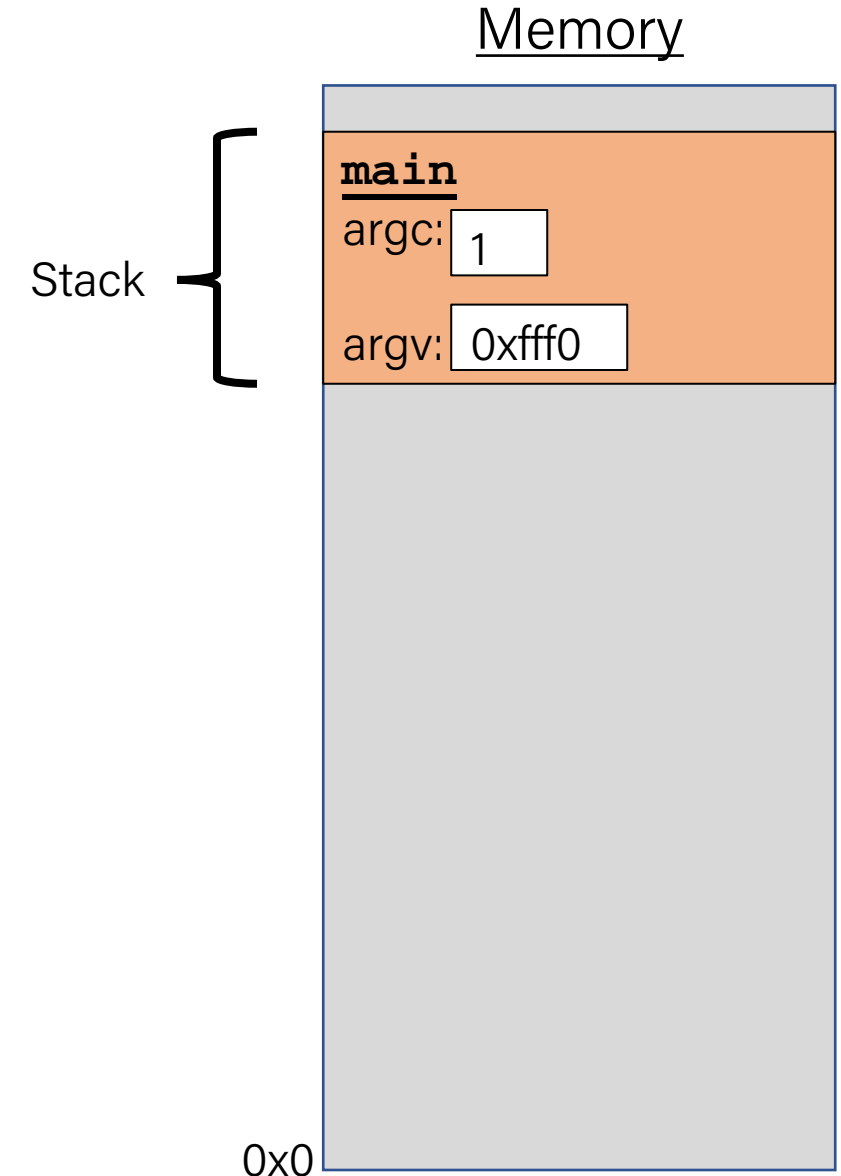
Memory

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

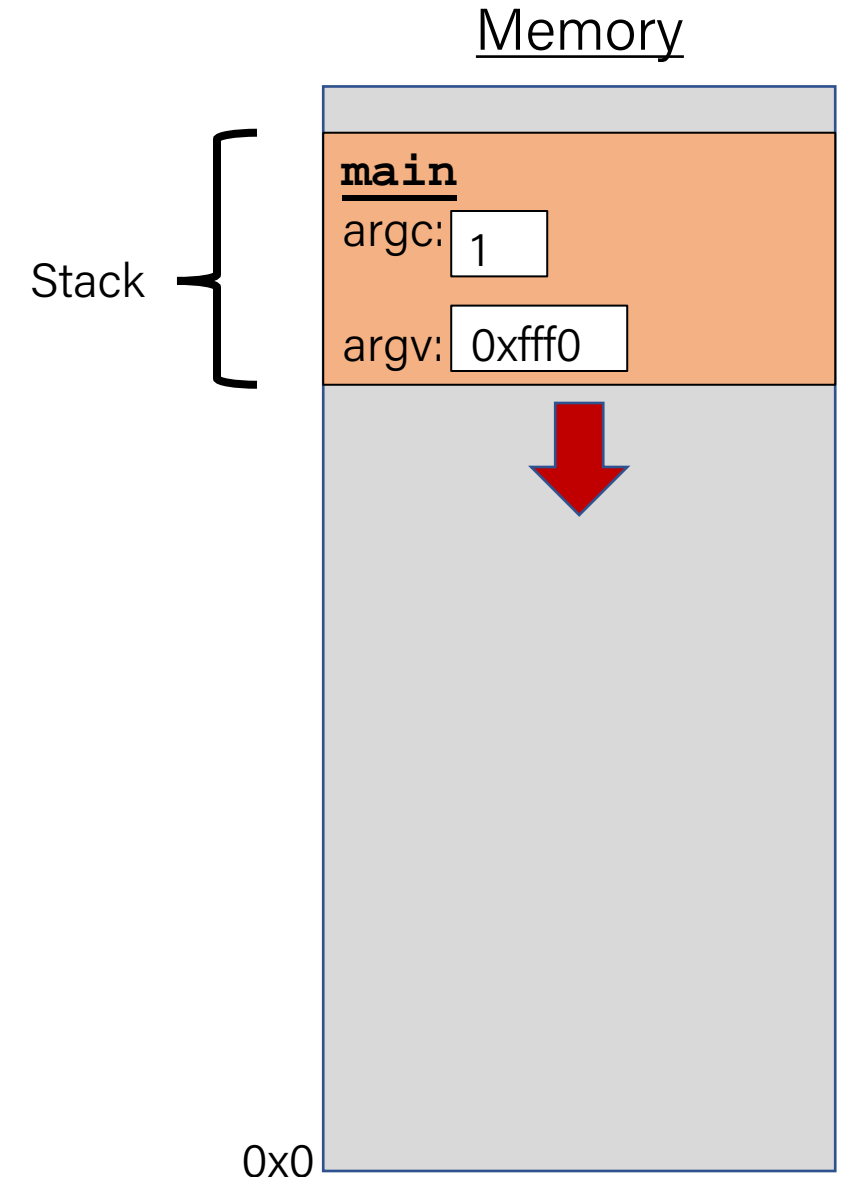**main**

argc: 1

argv: 0xfff0

0x0

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

0x0

64

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;
}
```
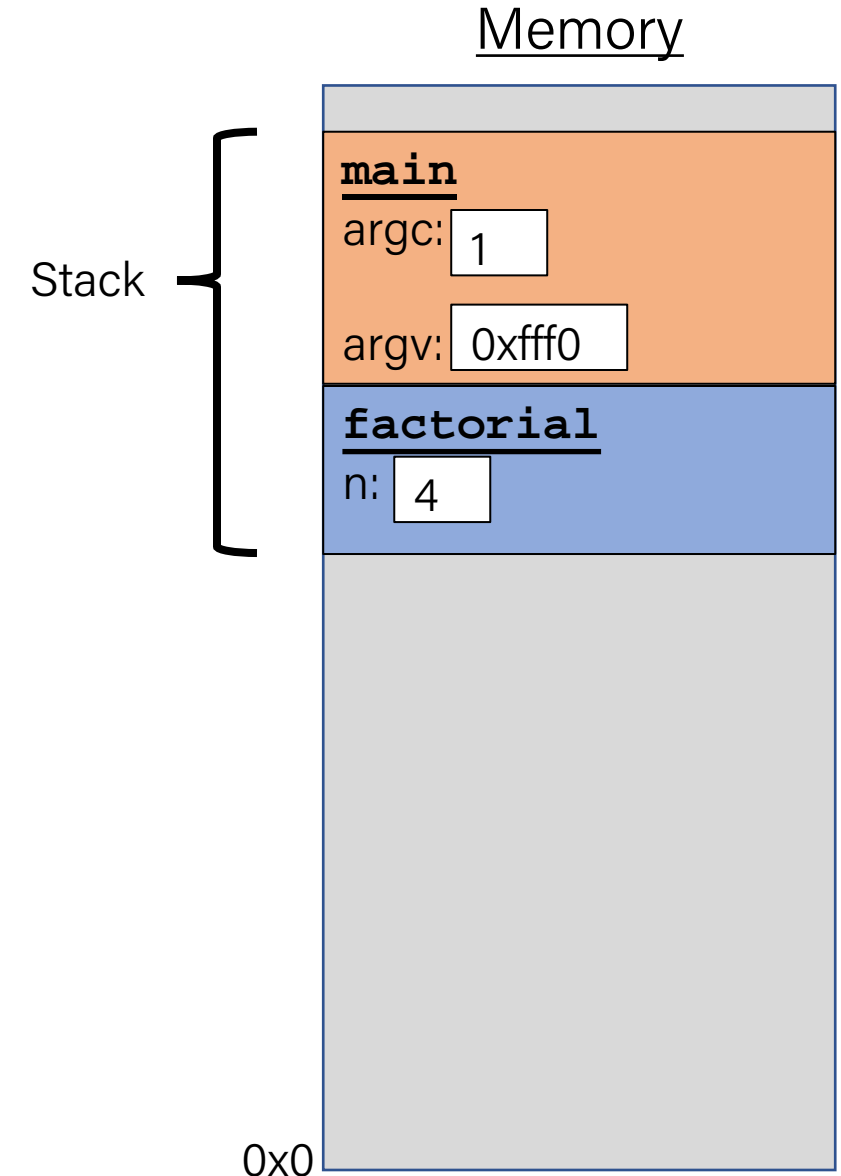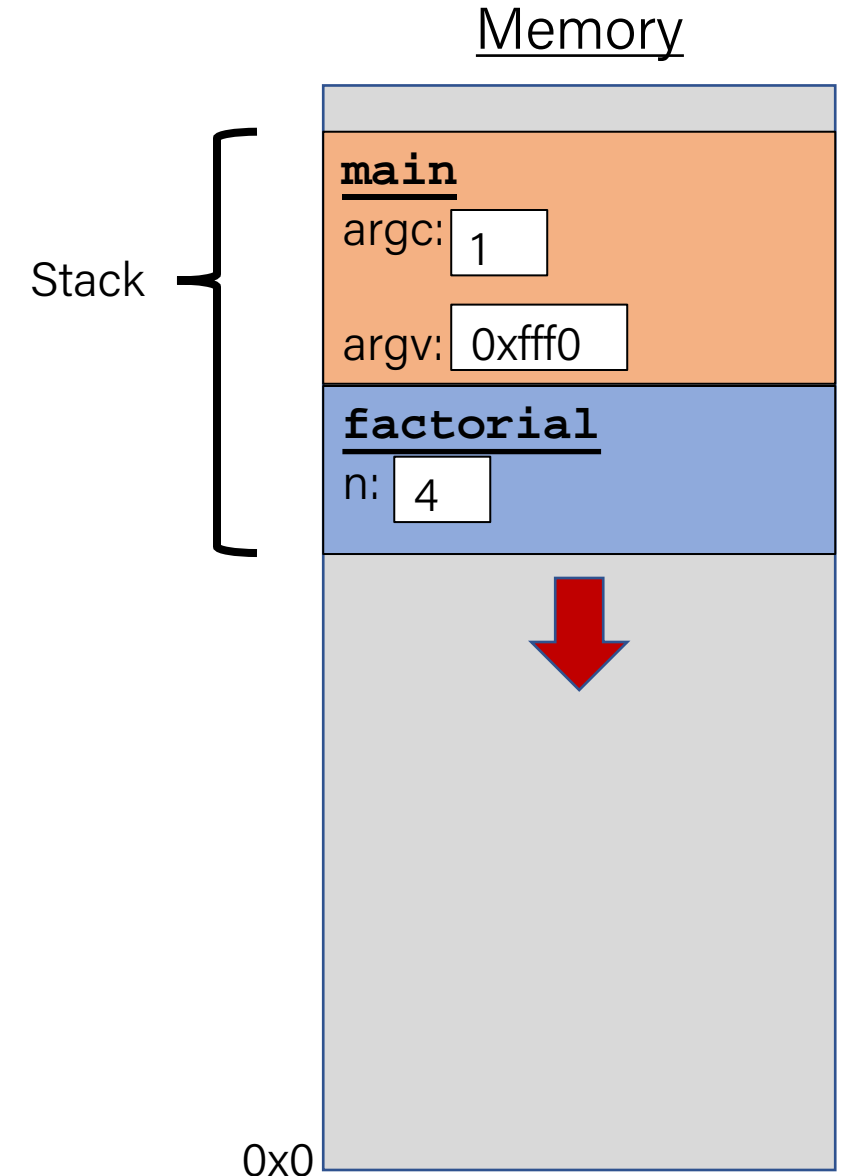
Memory

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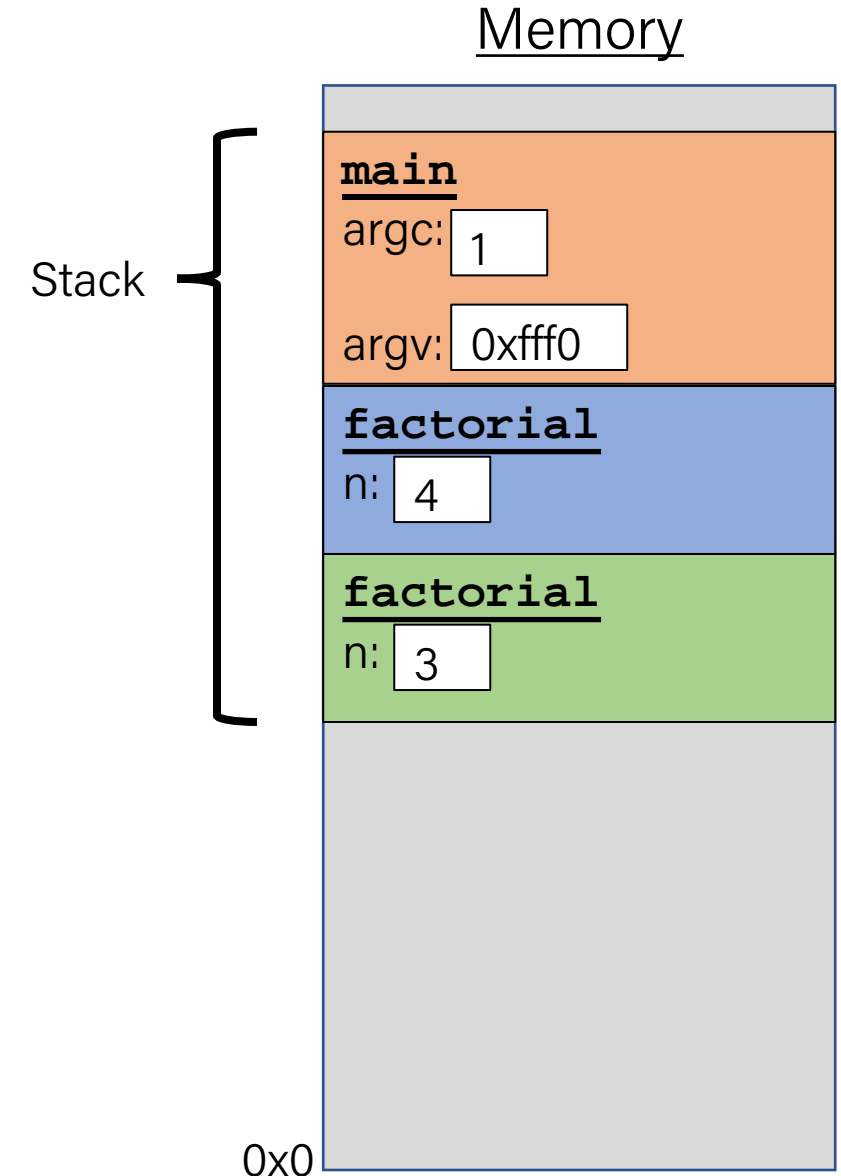
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

0x0

66

The Stack

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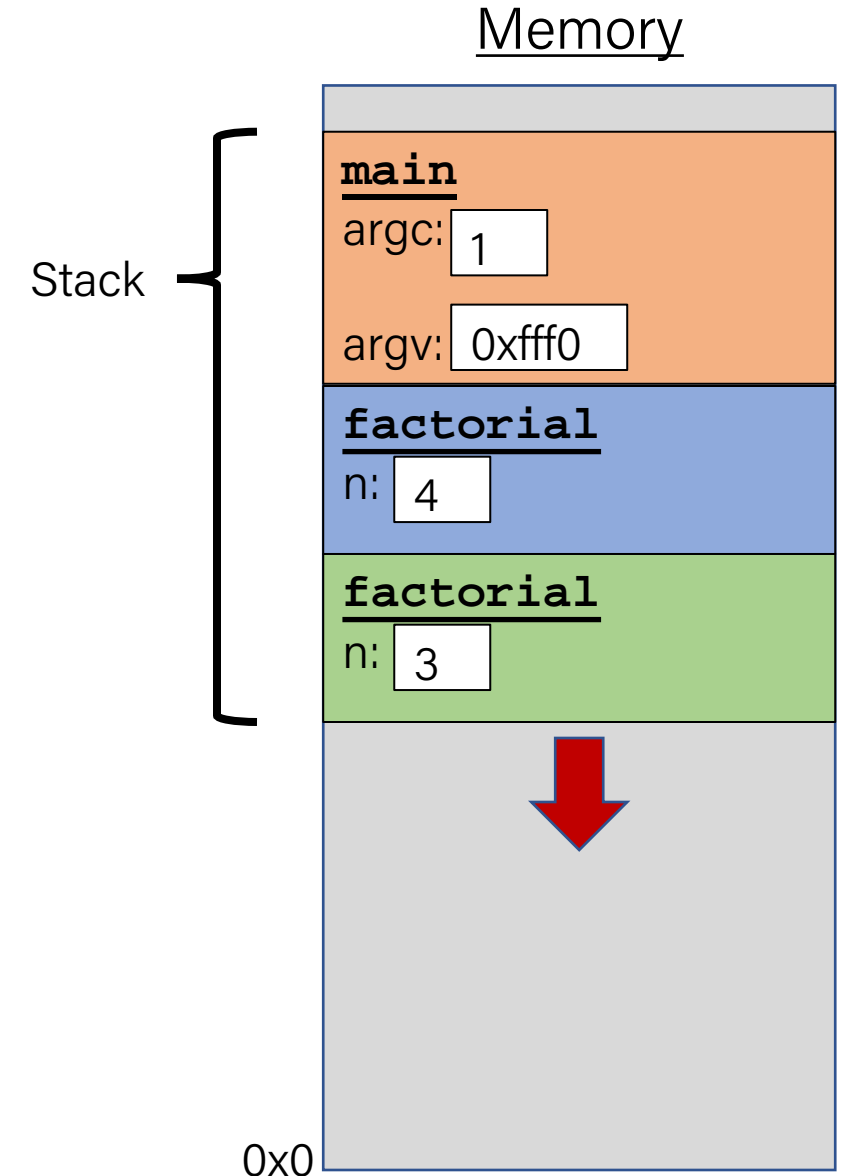
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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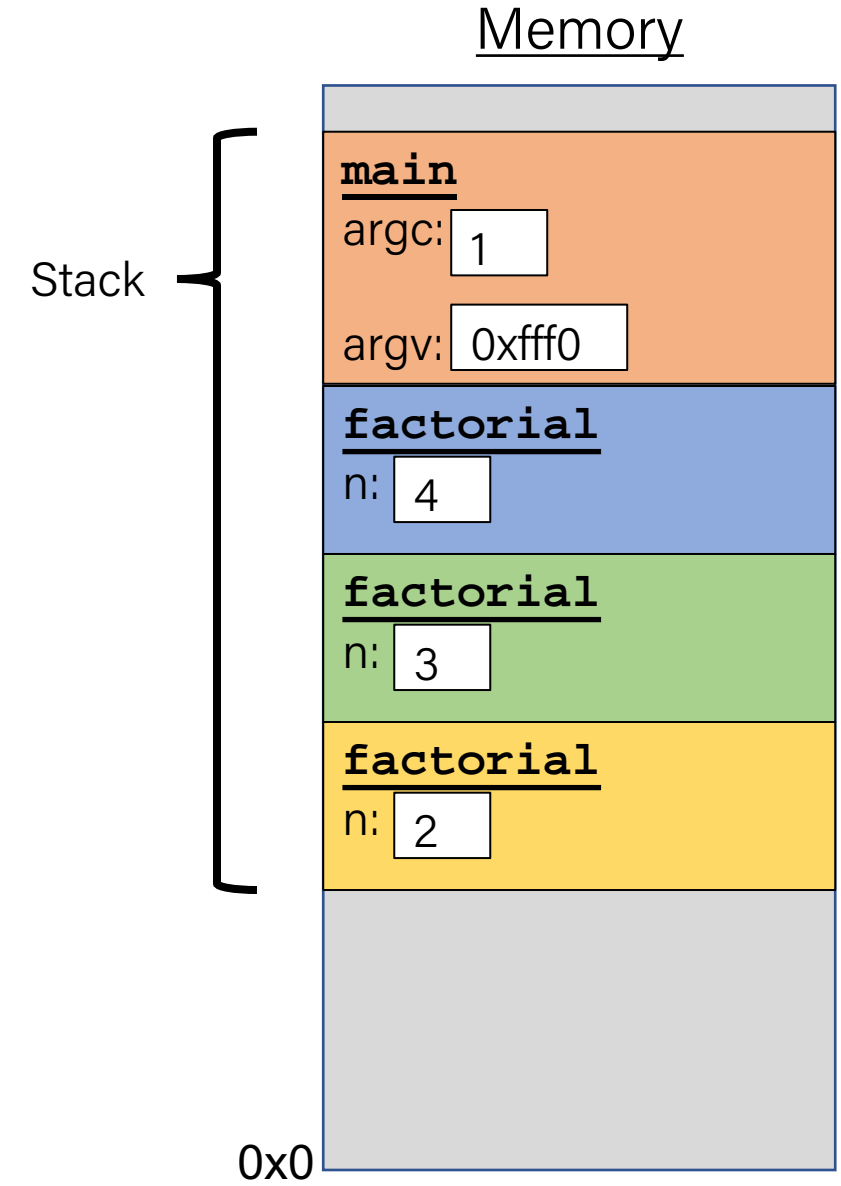
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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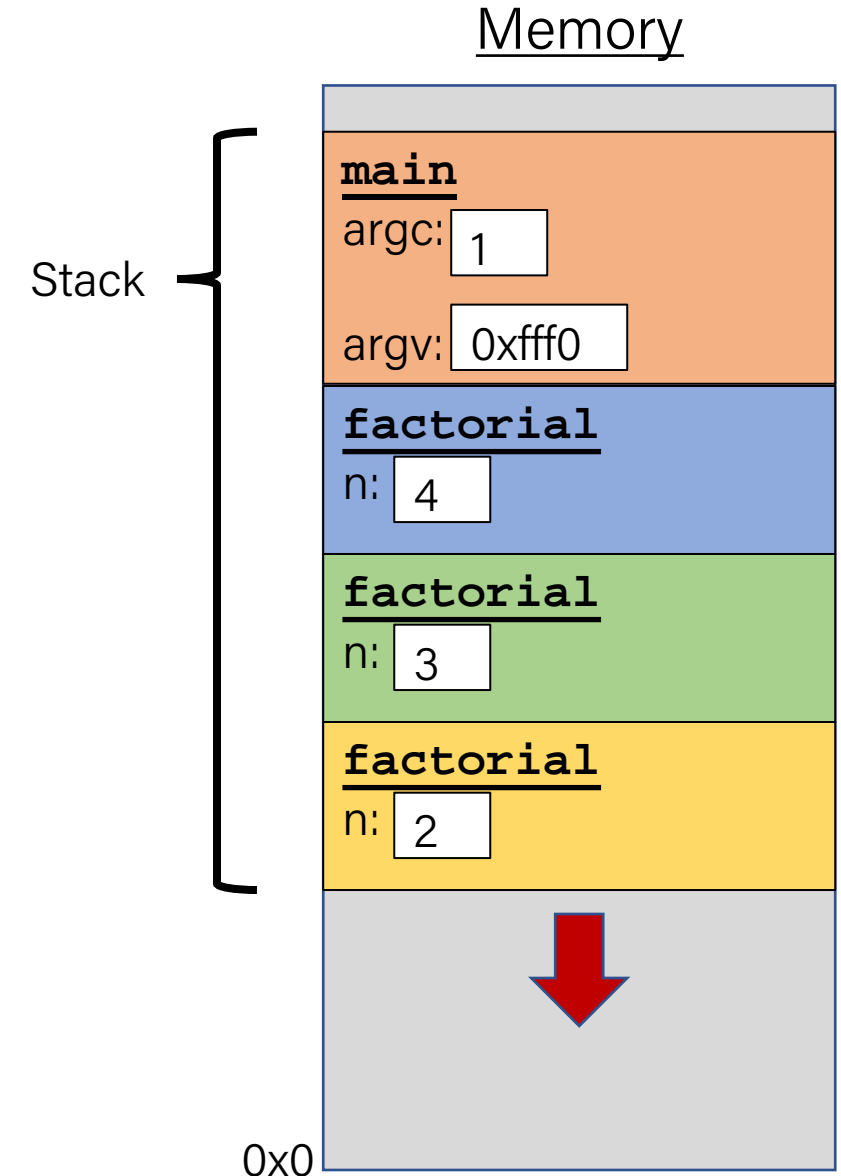
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

factorial
n: 2

0x0

The Stack

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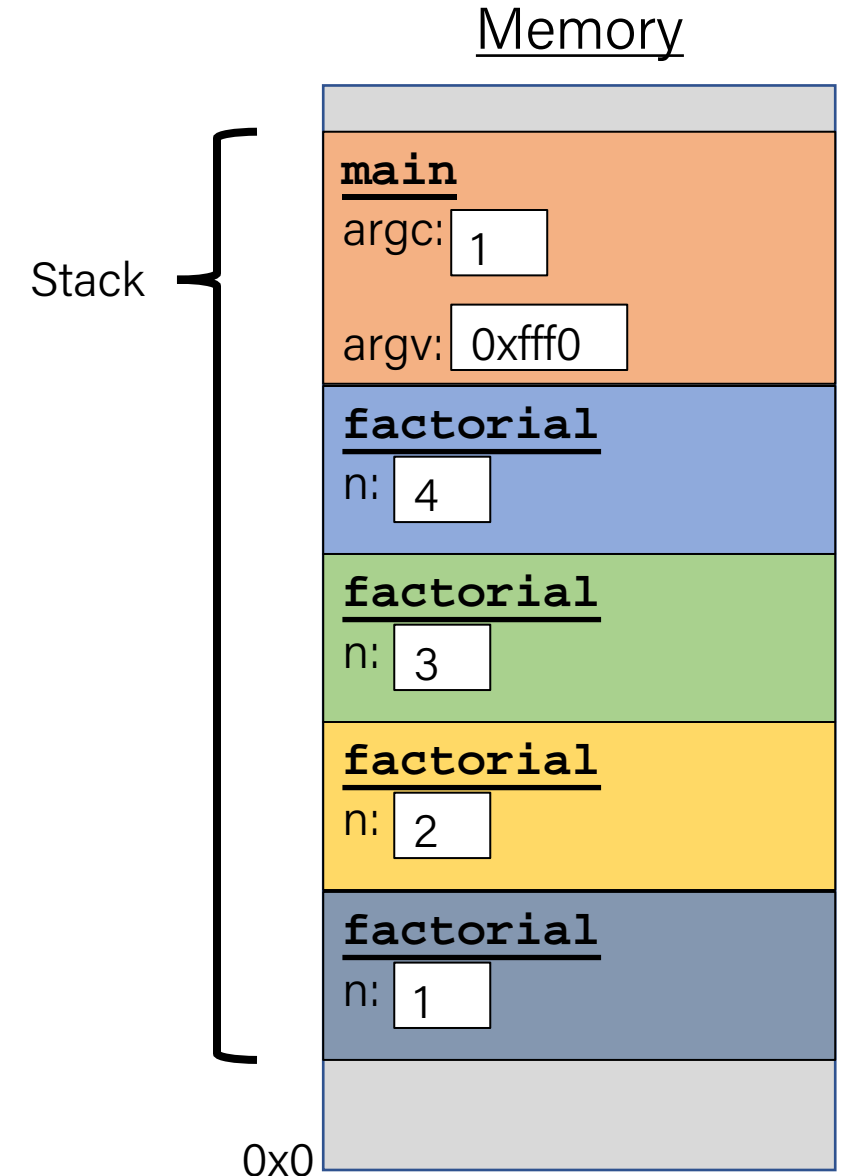
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

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

70

The Stack

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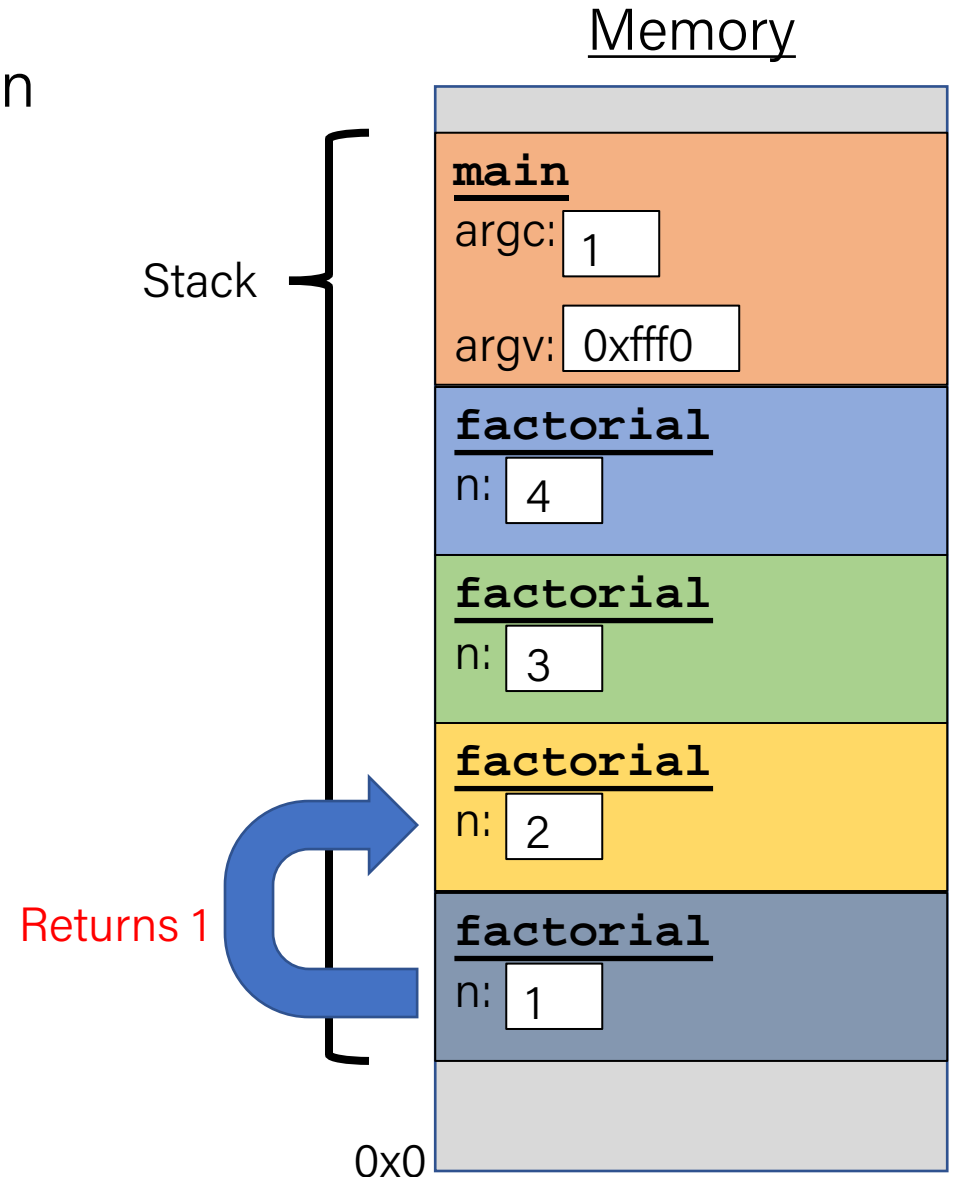
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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

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;
}
```
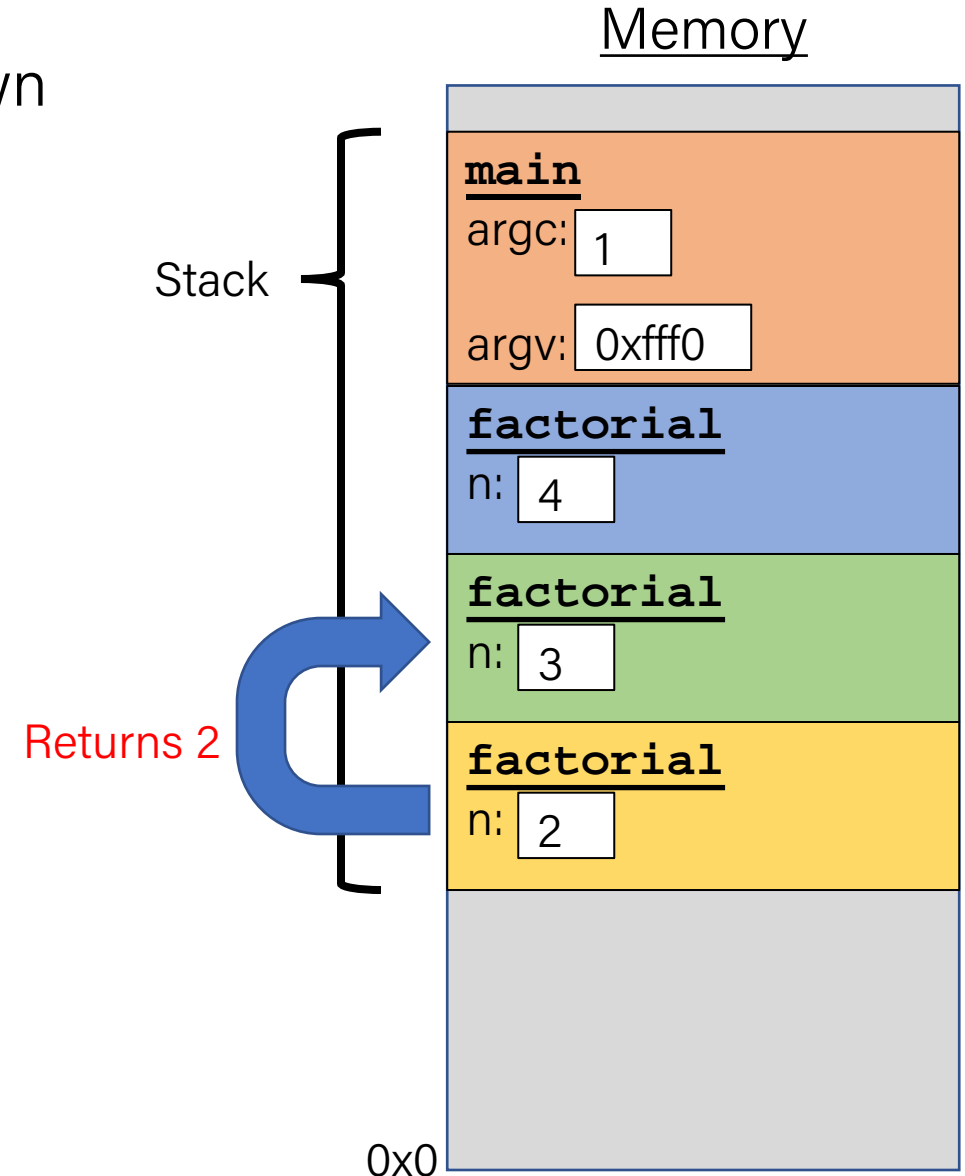
Memory

Stack

main
argc: 1
argv: 0xfff0

factorial
n: 4

factorial
n: 3

factorial
n: 2

Returns 1

factorial
n: 1

0x0

The Stack

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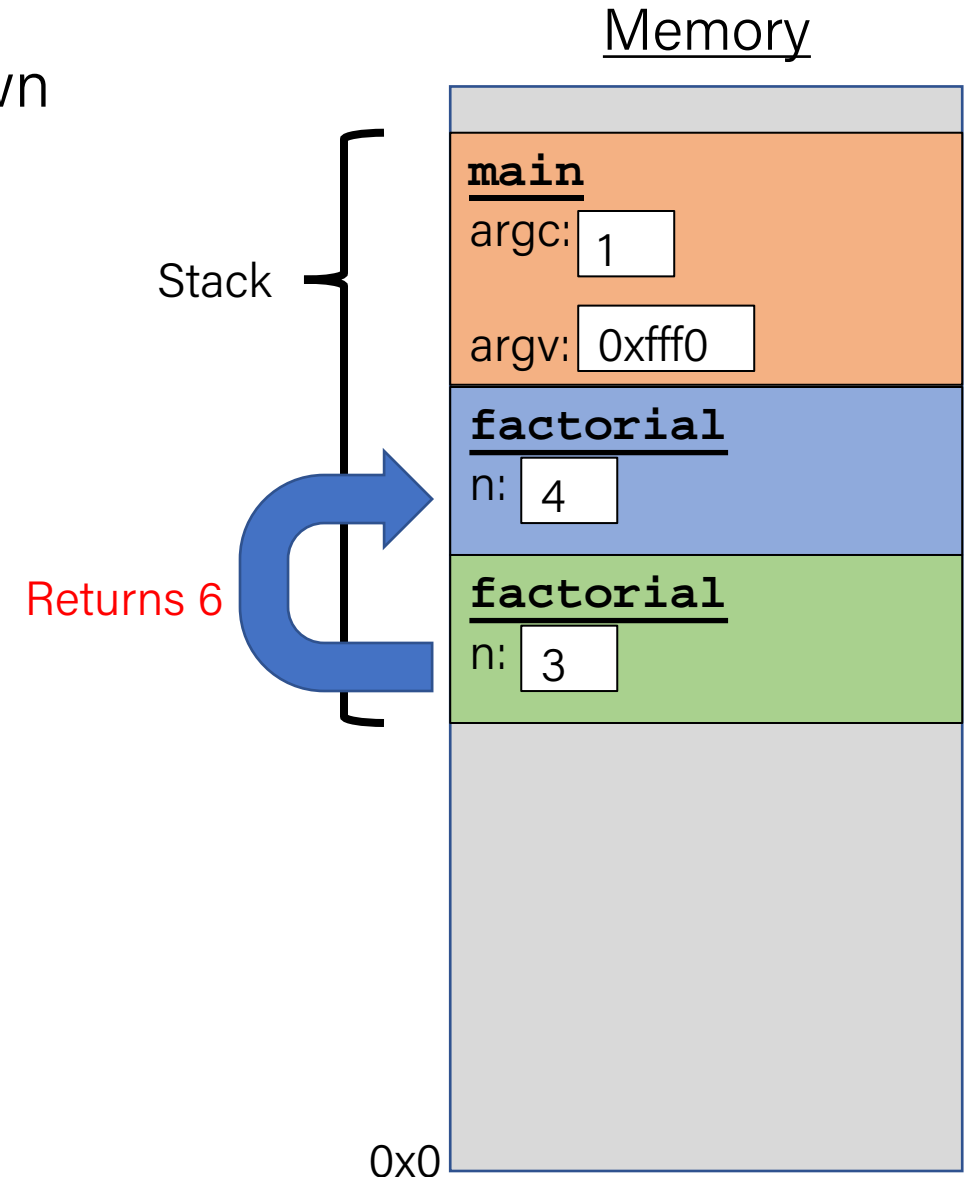
```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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

Returns 2

factorial
n: 2

0x0

73

The Stack

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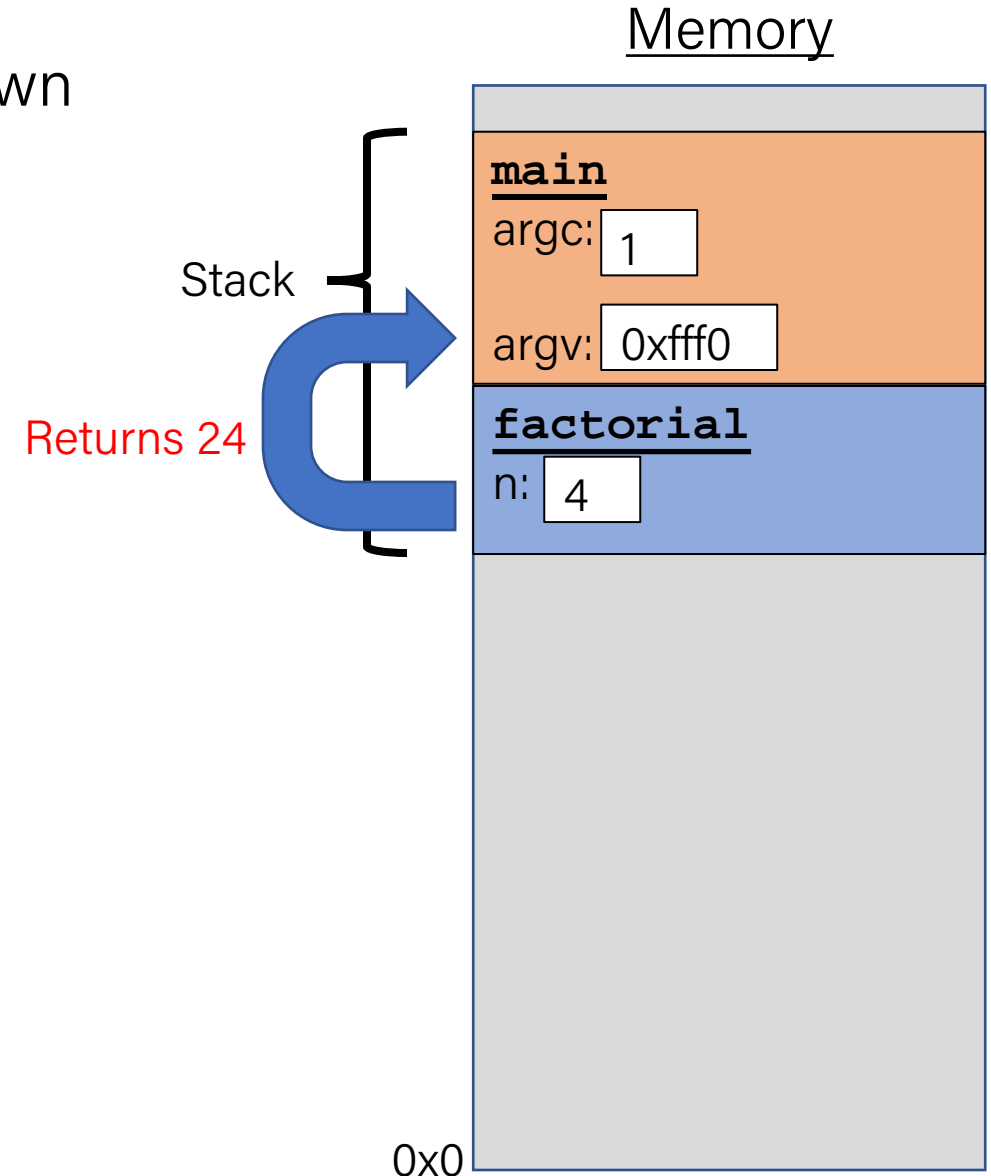
```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

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.

```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;
}
```
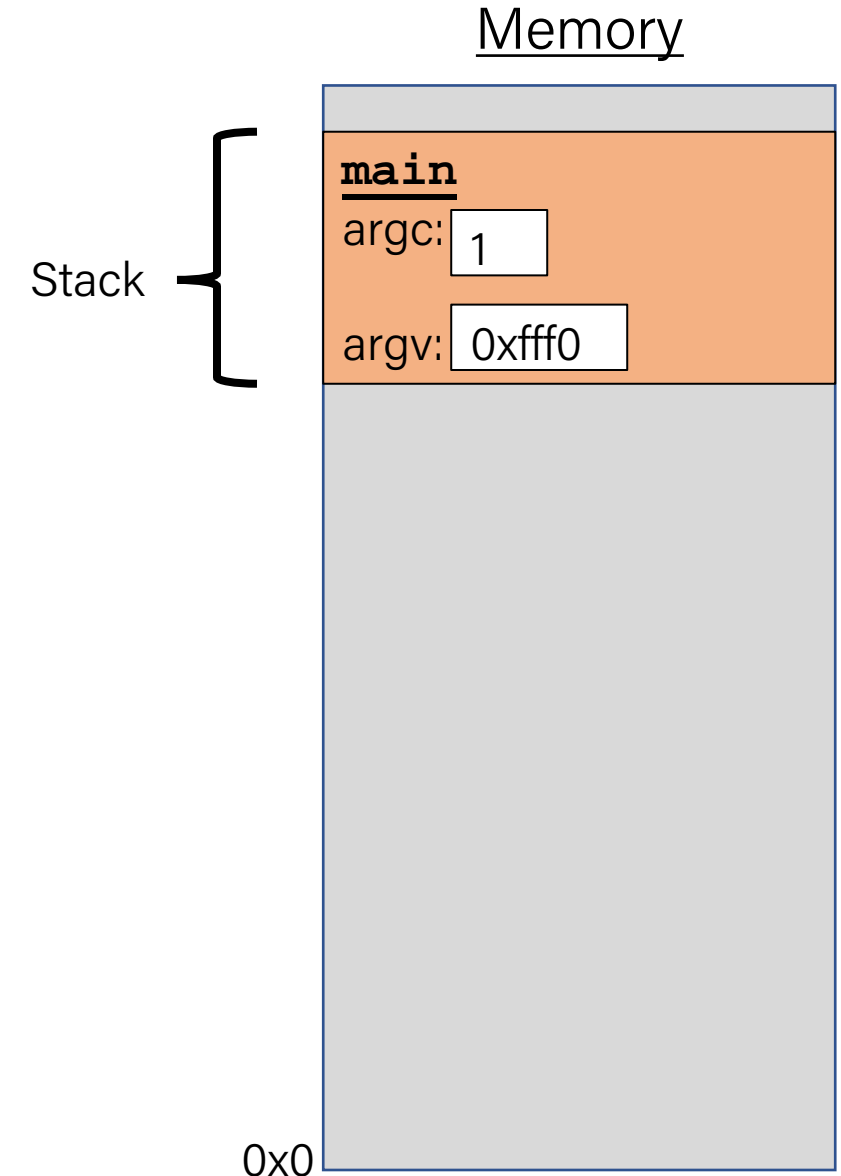
Memory

Stack

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;
}
```
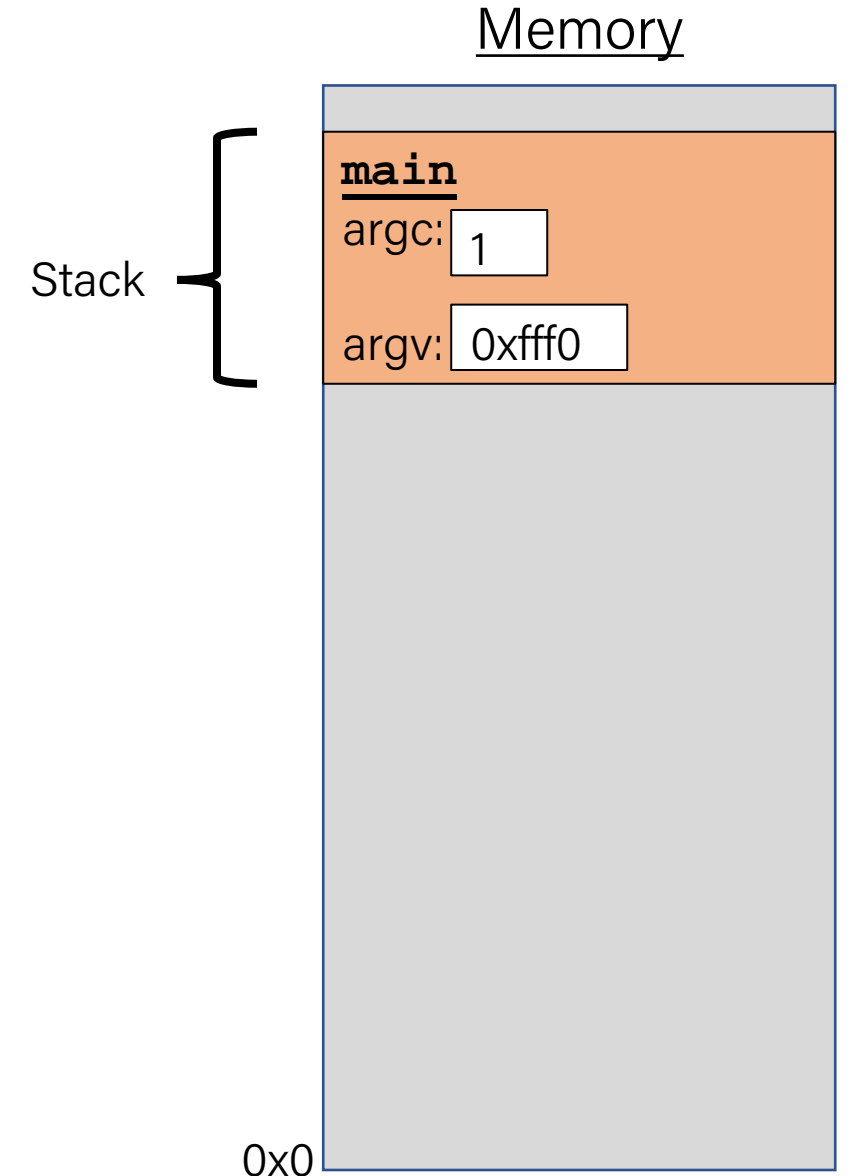
Memory

Stack

main

argc: 1

argv: 0xfff0

0x0

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;
}
```
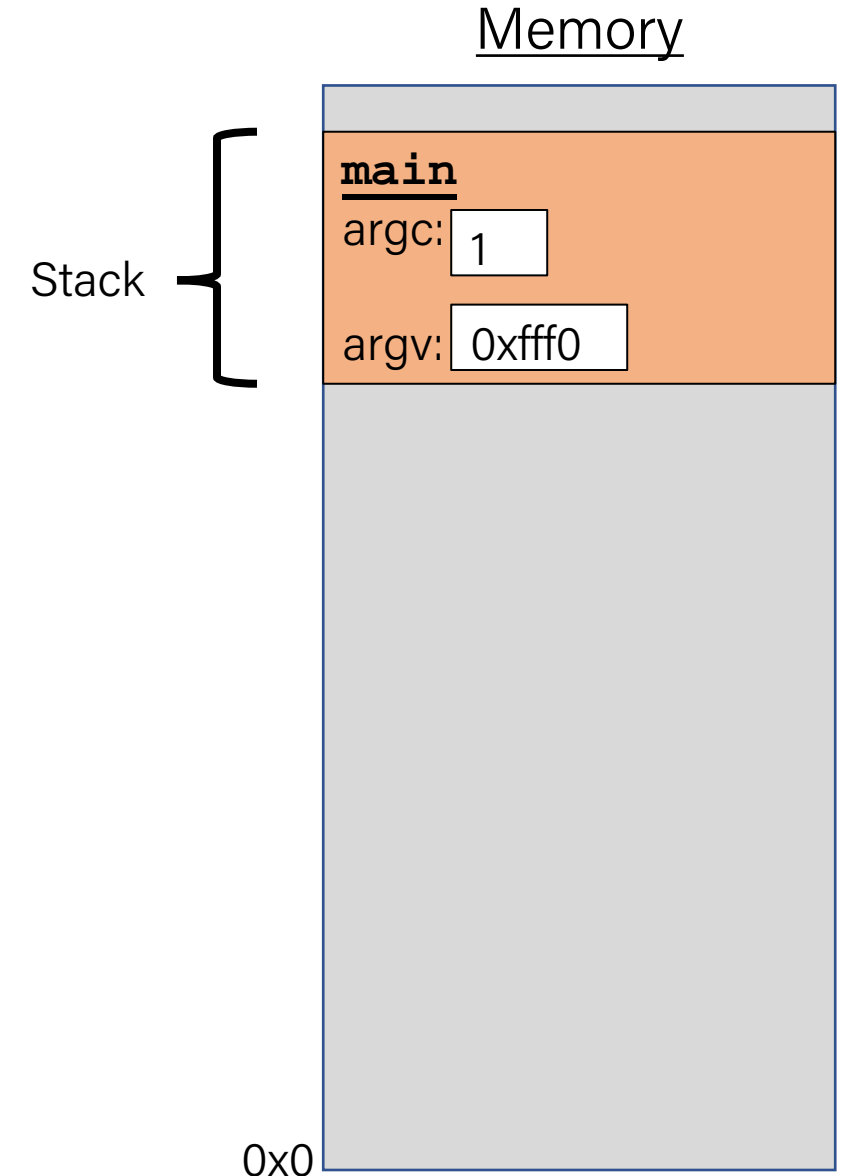
Memory

Stack

main

argc: 1

argv: 0xfff0

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?
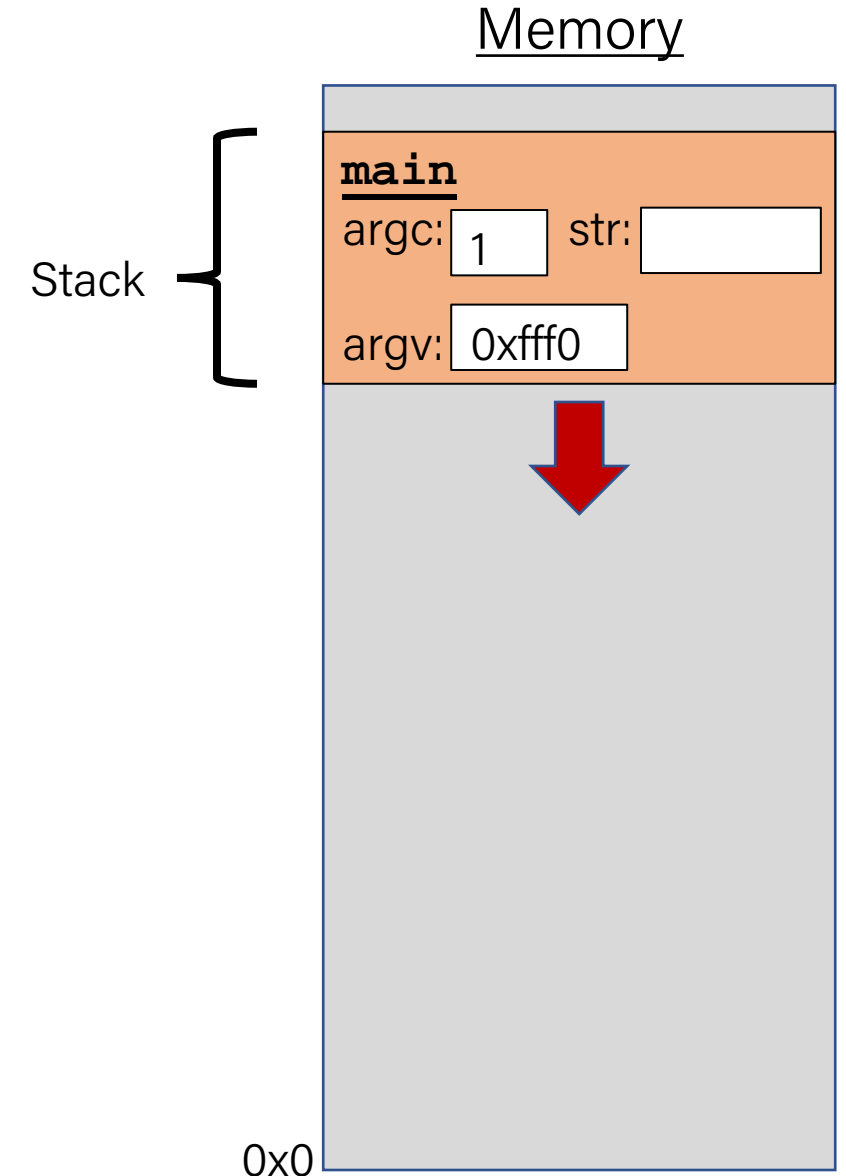
The Stack

```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

argv: 0xfff0
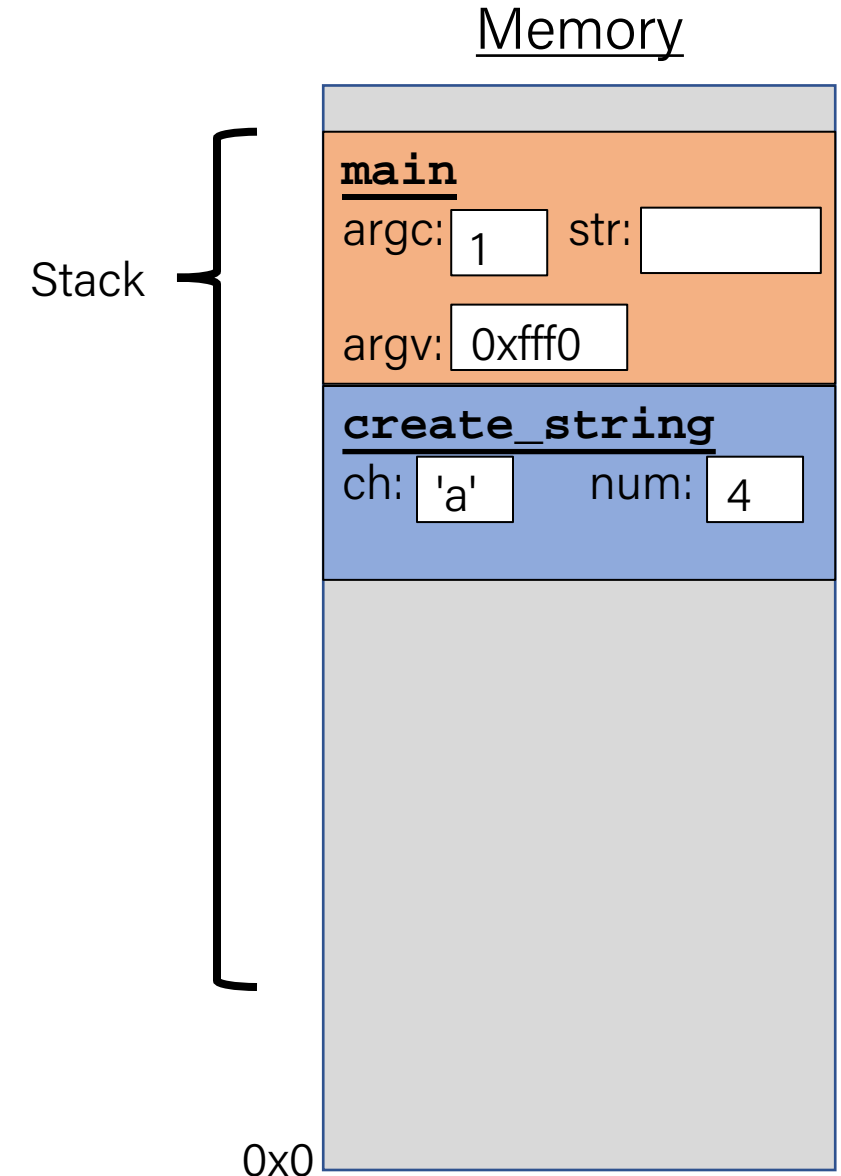
0x0

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

0x0

80

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;
}
```
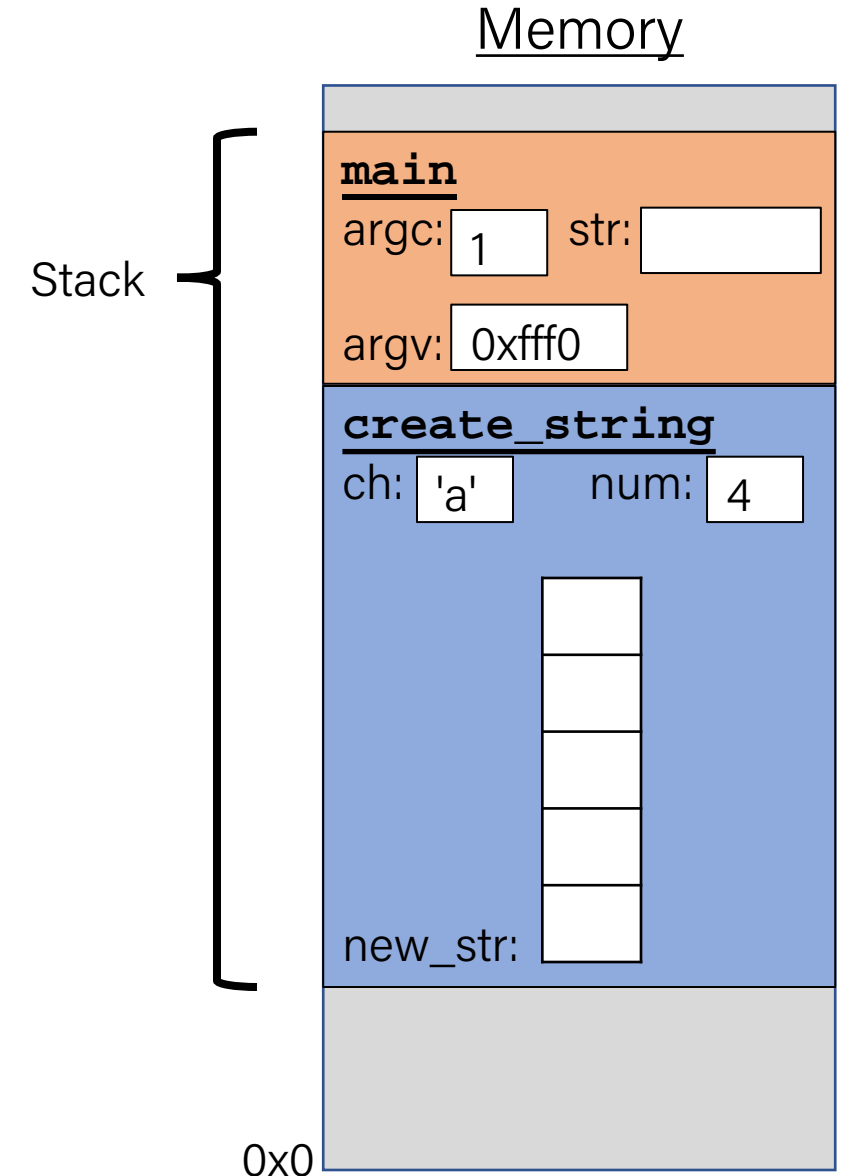
Memory

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;
}
```
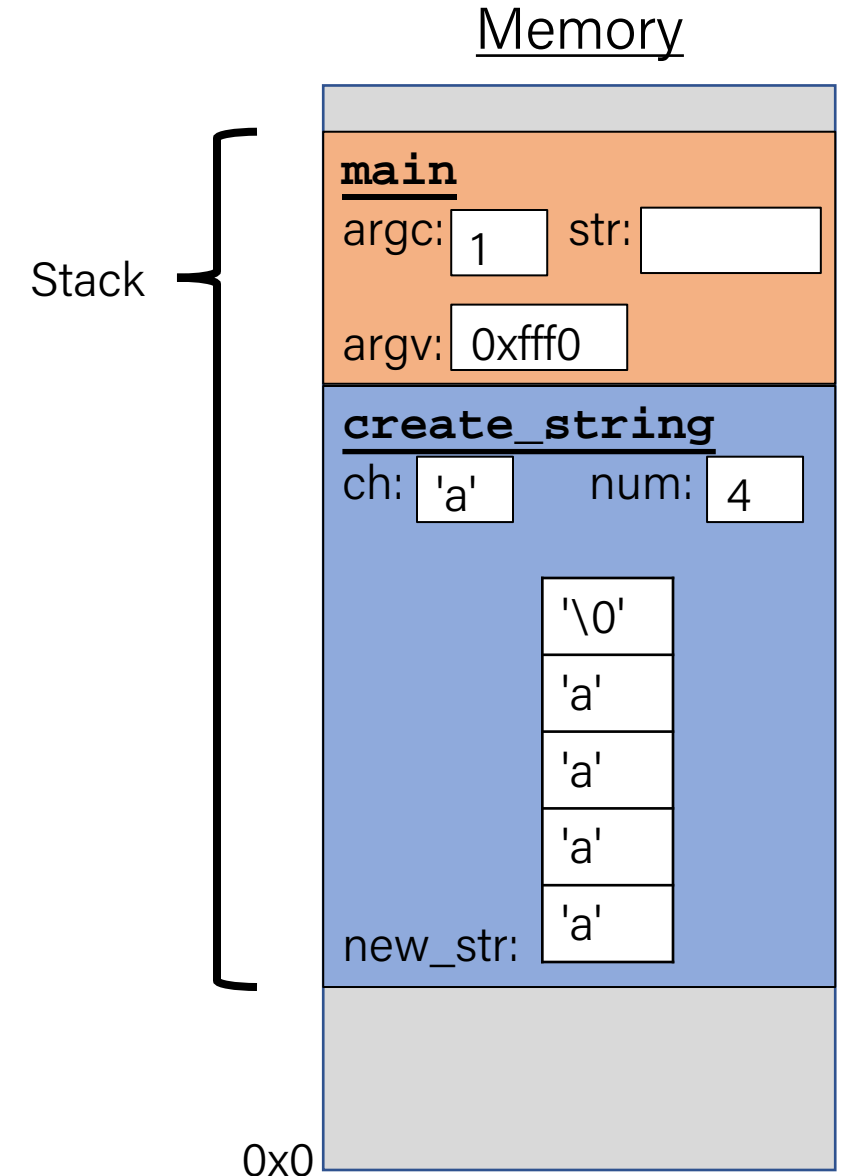
Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str:

0x0

The Stack

```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;
}
```
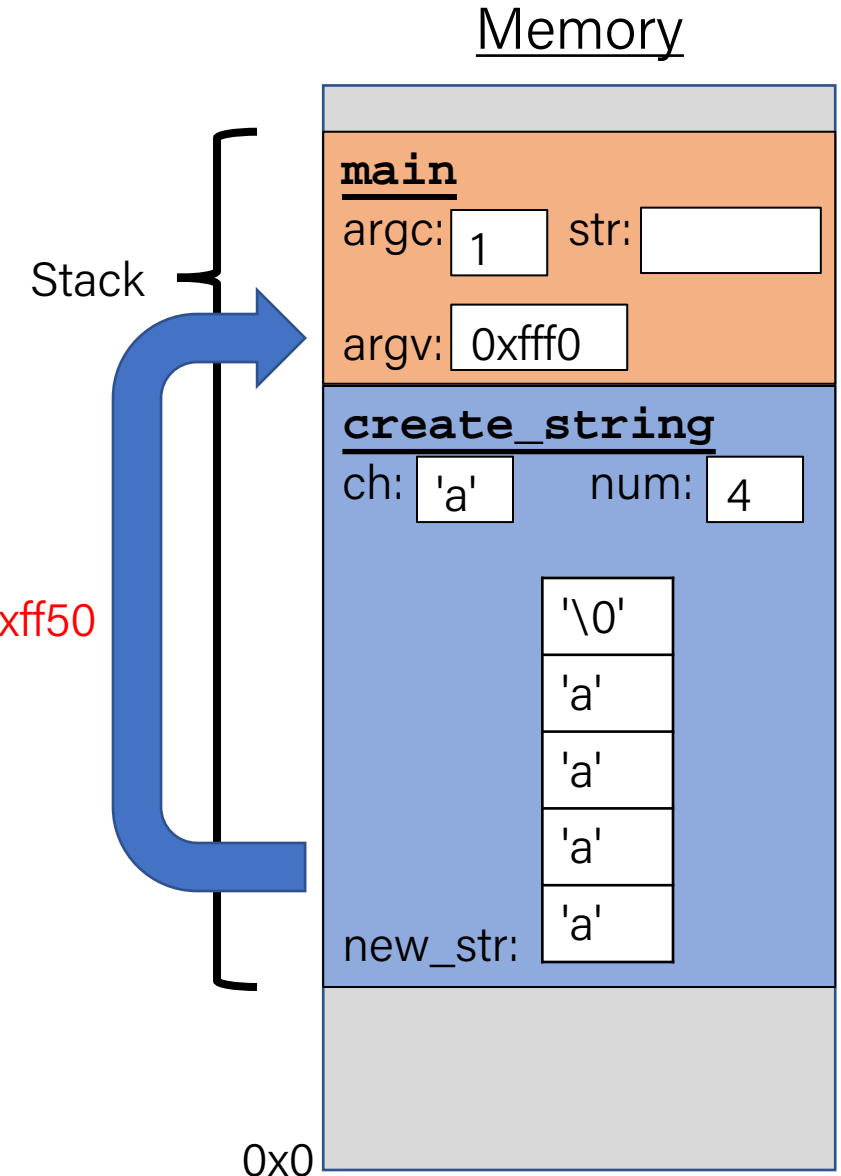
Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str:

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

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;
}
```
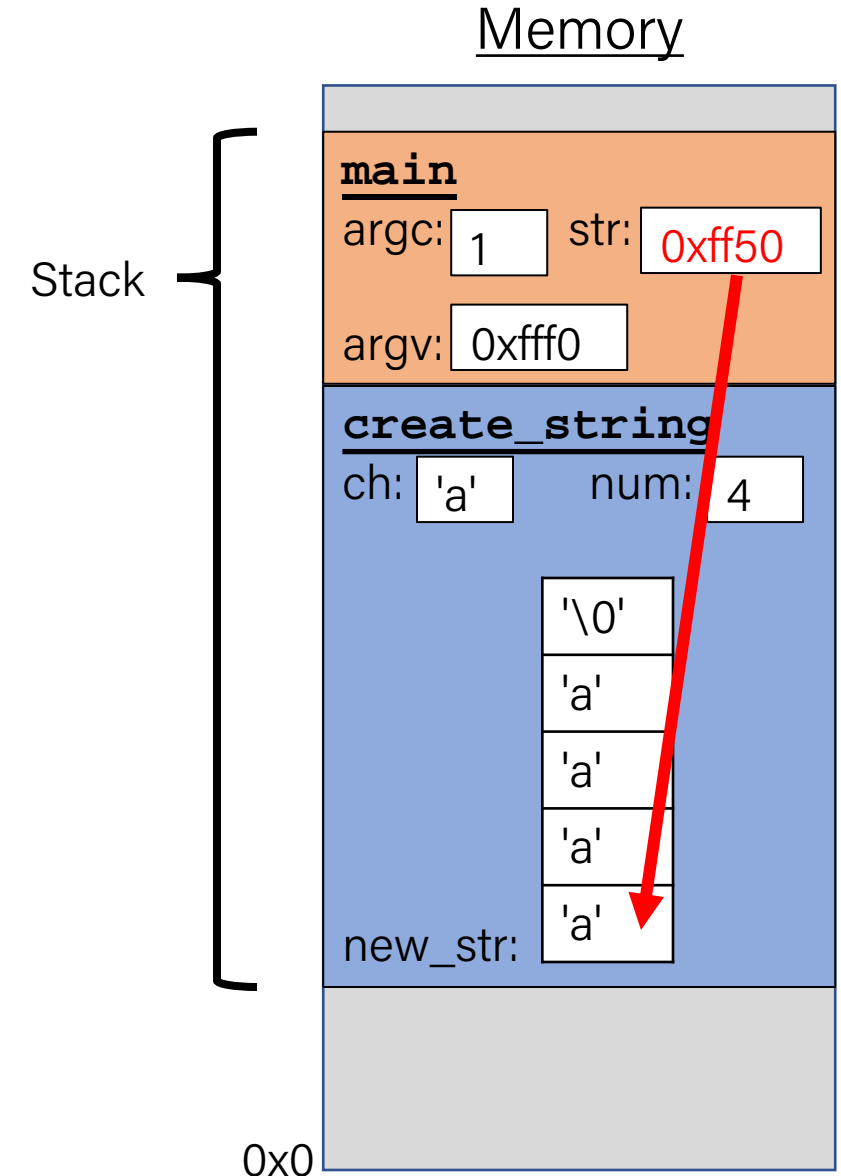
Returns e.g. 0xff50

Memory

Stack

main
argc: 1   str:
argv: 0xfff0

create_string
ch: 'a'   num: 4

'\0'
'a'
'a'
'a'
new_str: 'a'
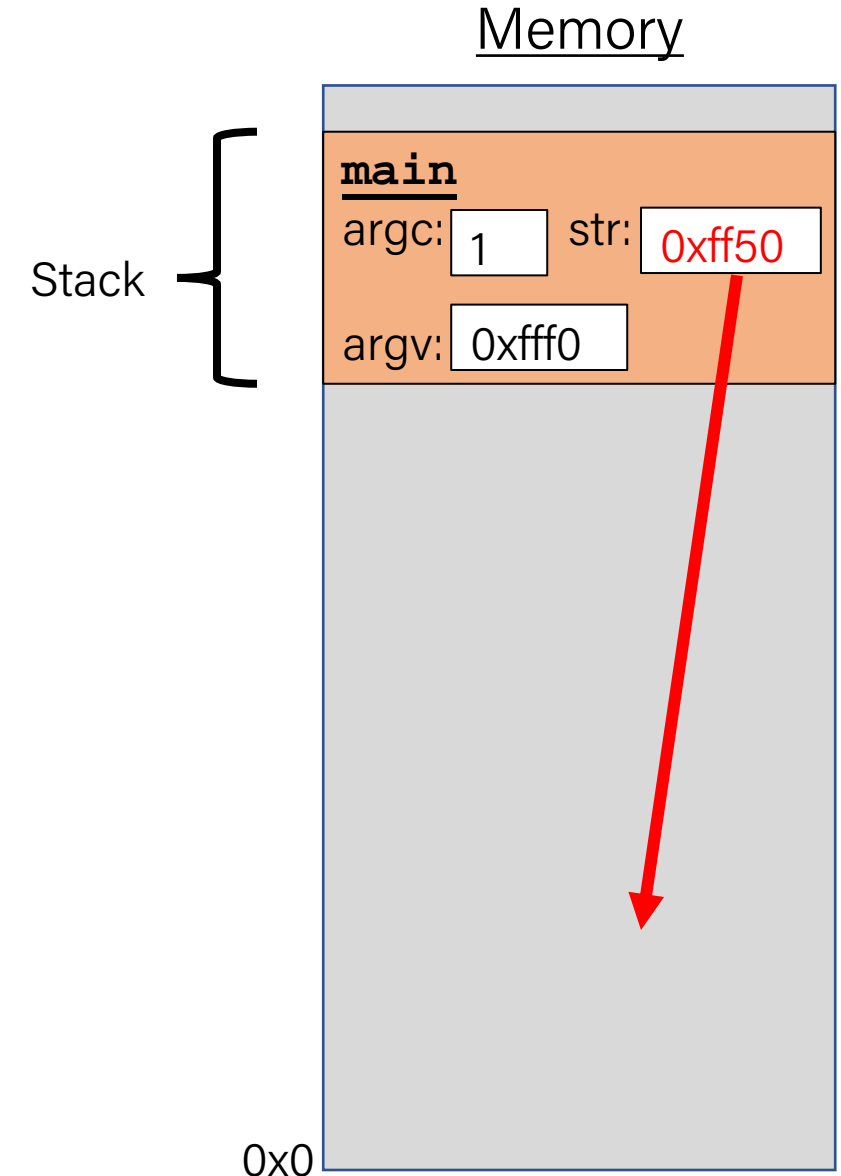
0x0

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

create_string

ch: 'a'   num: 4

'\0'

'a'

'a'

'a'

new_str: 'a'

0x0

85

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;
}
```
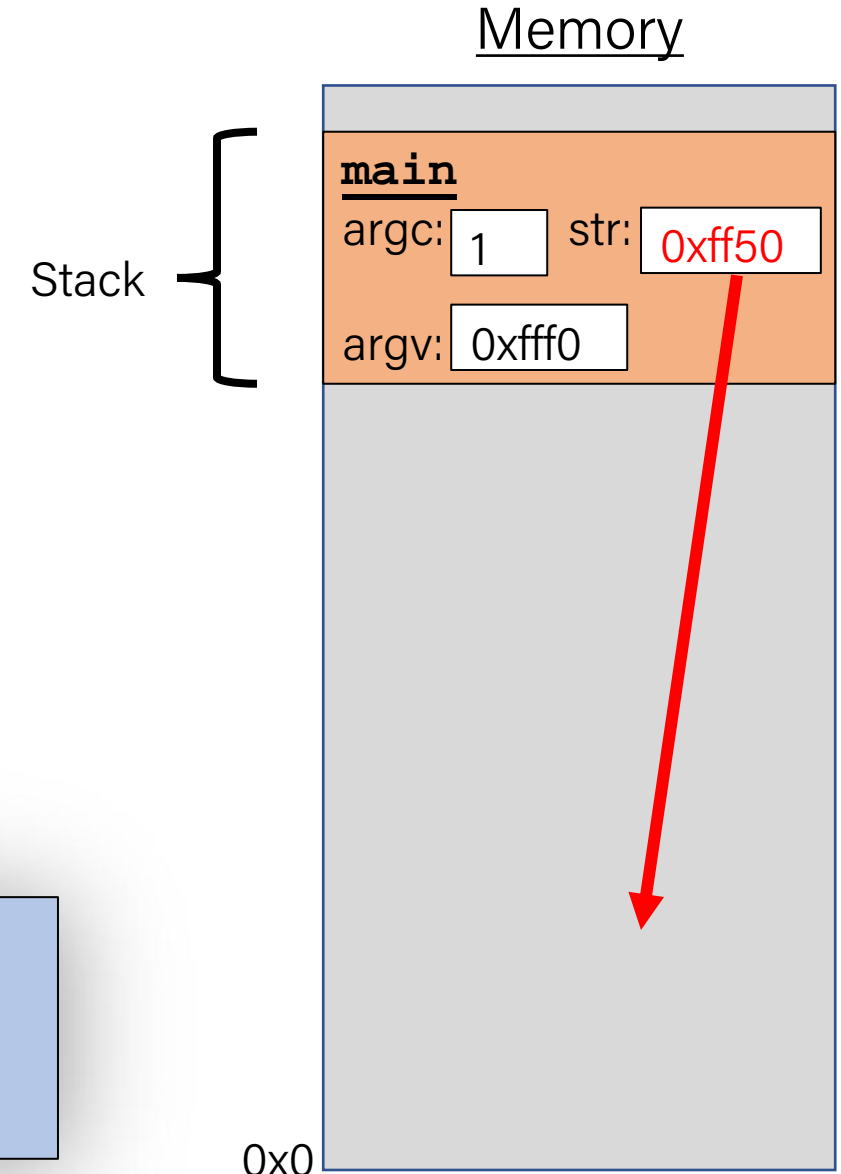
Memory

Stack

main

argc: 1   str: 0xff50

argv: 0xfff0

0x0

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;
}
```
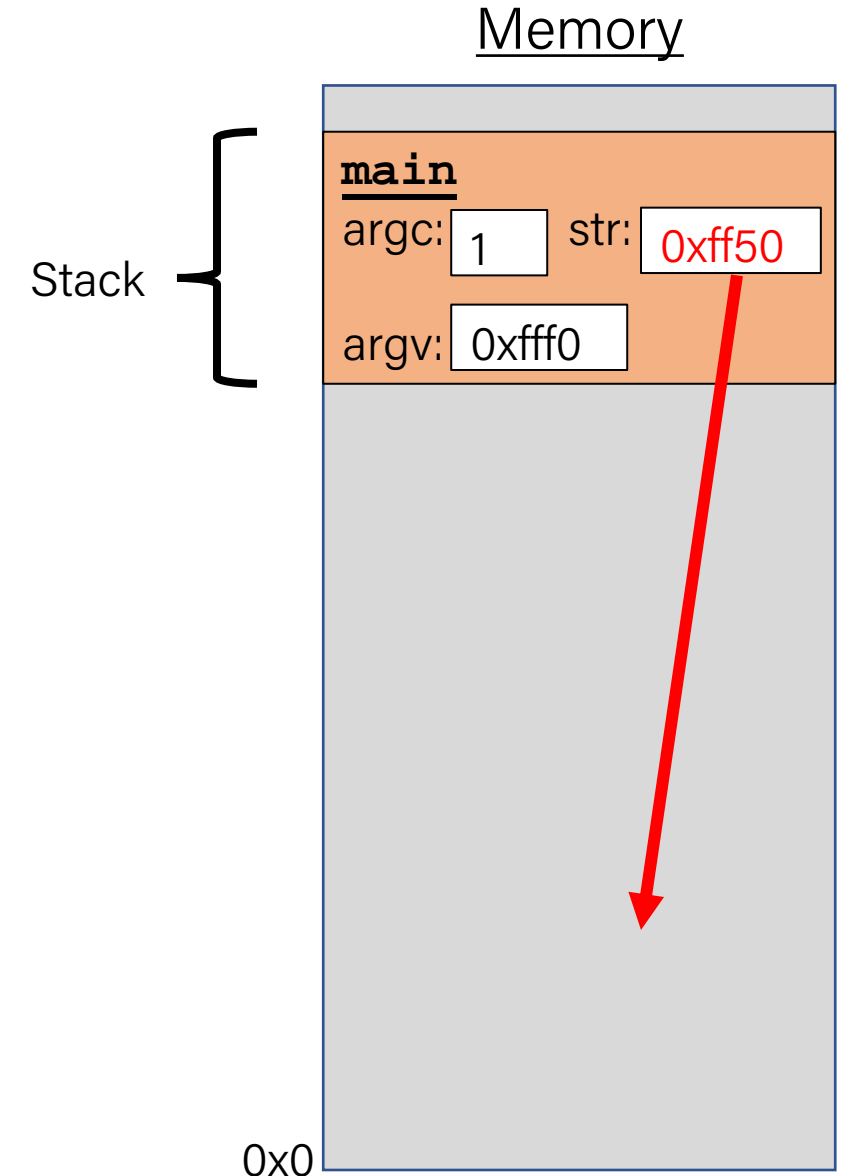
Memory

Stack

main

argc: 1   str: 0xff50

argv: 0xfff0

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

87

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

88

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

```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?
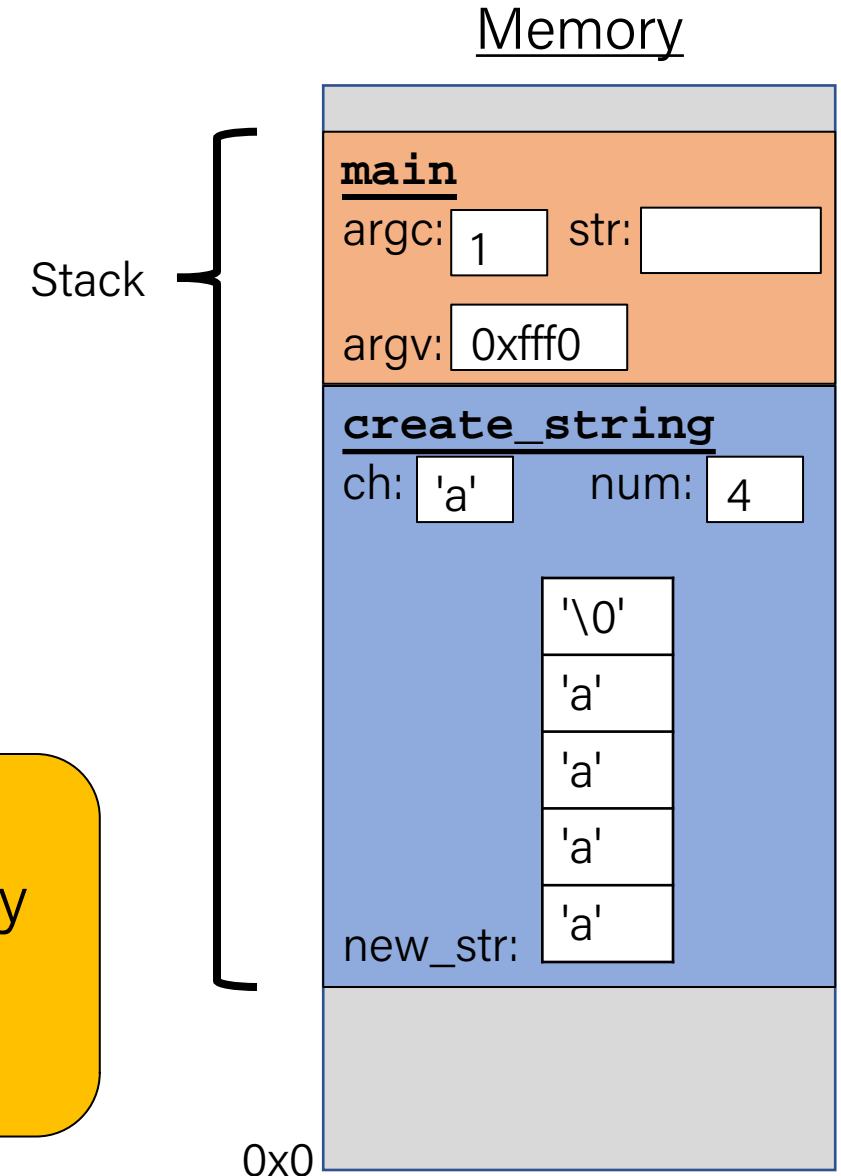
0x0
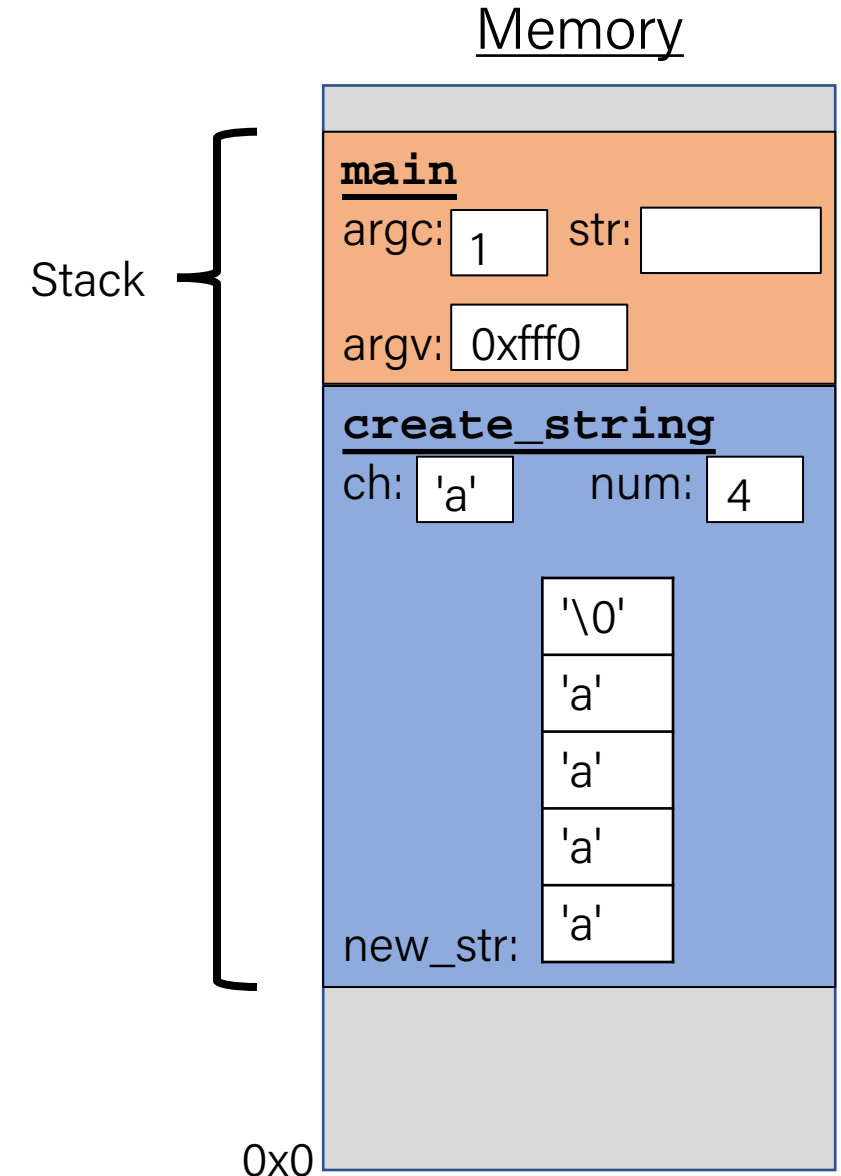
91

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;
}
```
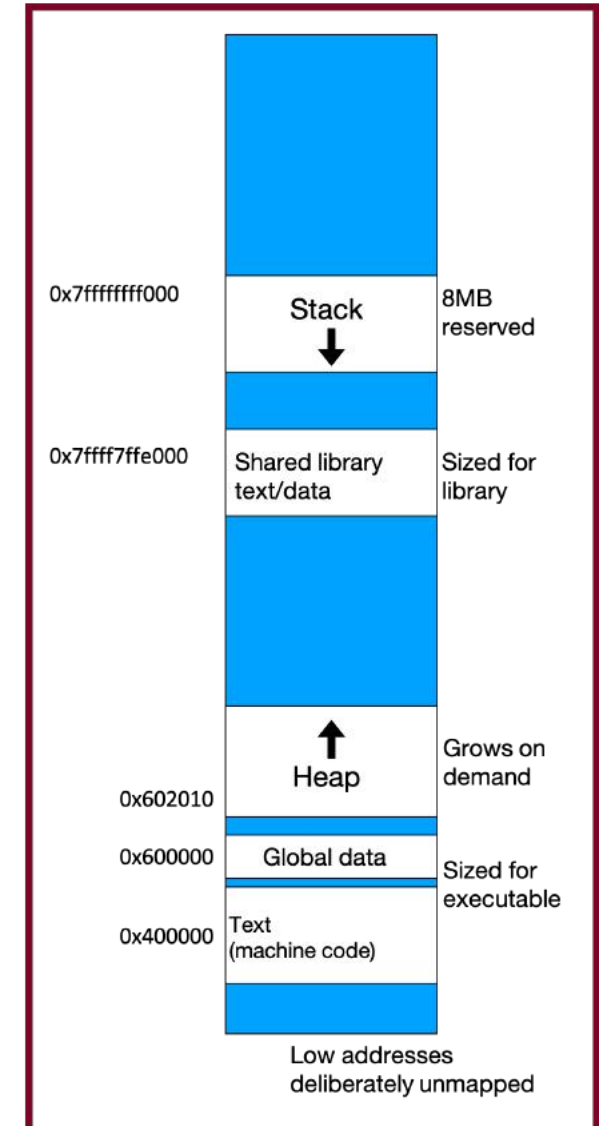
Memory

Stack

main

argc: 1   str:

argv: 0xfff0

create_string

ch: 'a'   num: 4

'\0'

'a'

'a'

'a'

new_str: 'a'

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

0x0

92

The Heap

- The **heap** is a part of memory that you can manage yourself.

- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.

- Unlike the stack, the heap grows **upwards** as more memory is allocated.

The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.
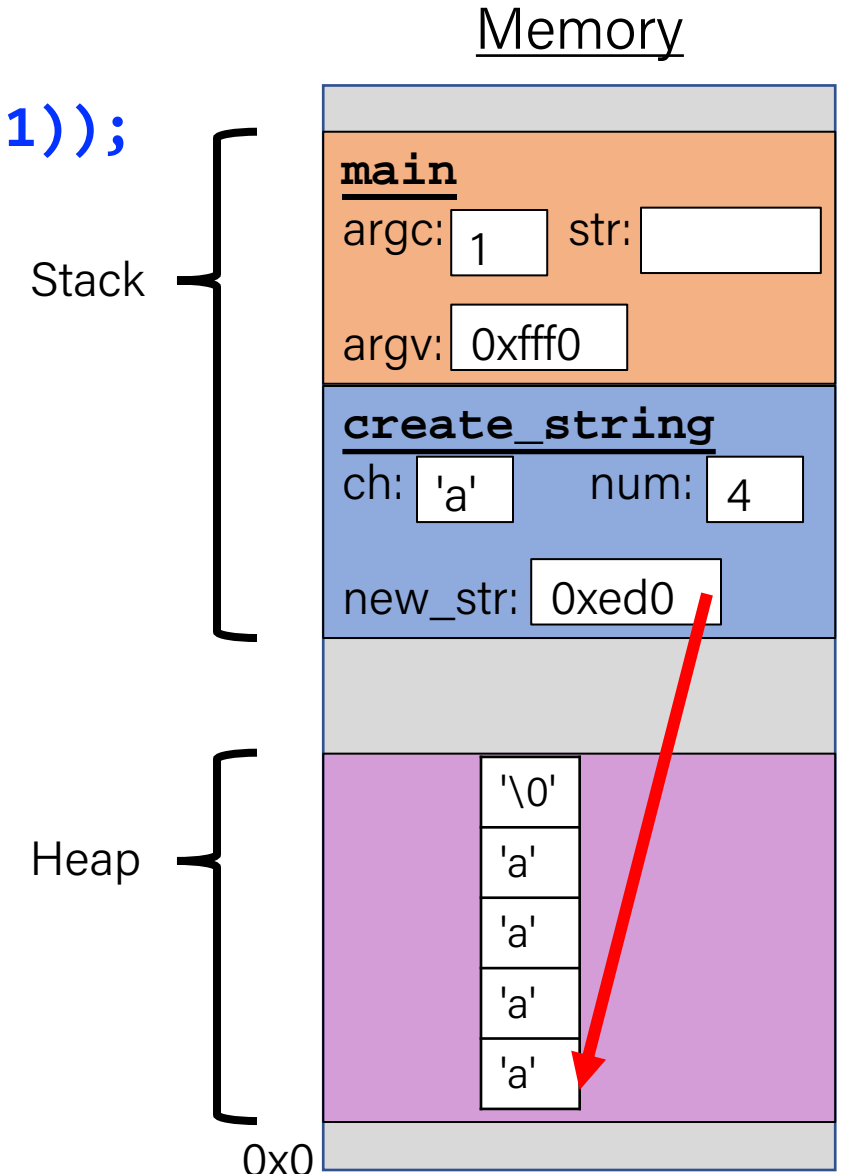
malloc

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

To allocate memory on the heap, use the **malloc** function ("memory allocate") and specify the number of bytes you'd like.

- This function returns a pointer to *the **starting address** of the new memory.* It doesn't know or care whether it will be used as an array, a single block of memory, etc.

- **void *** means a pointer to generic memory. You can set another pointer equal to it without any casting.

- The memory is *not* cleared out before being allocated to you!

- If `malloc` returns `NULL`, then there wasn't enough memory for this request.
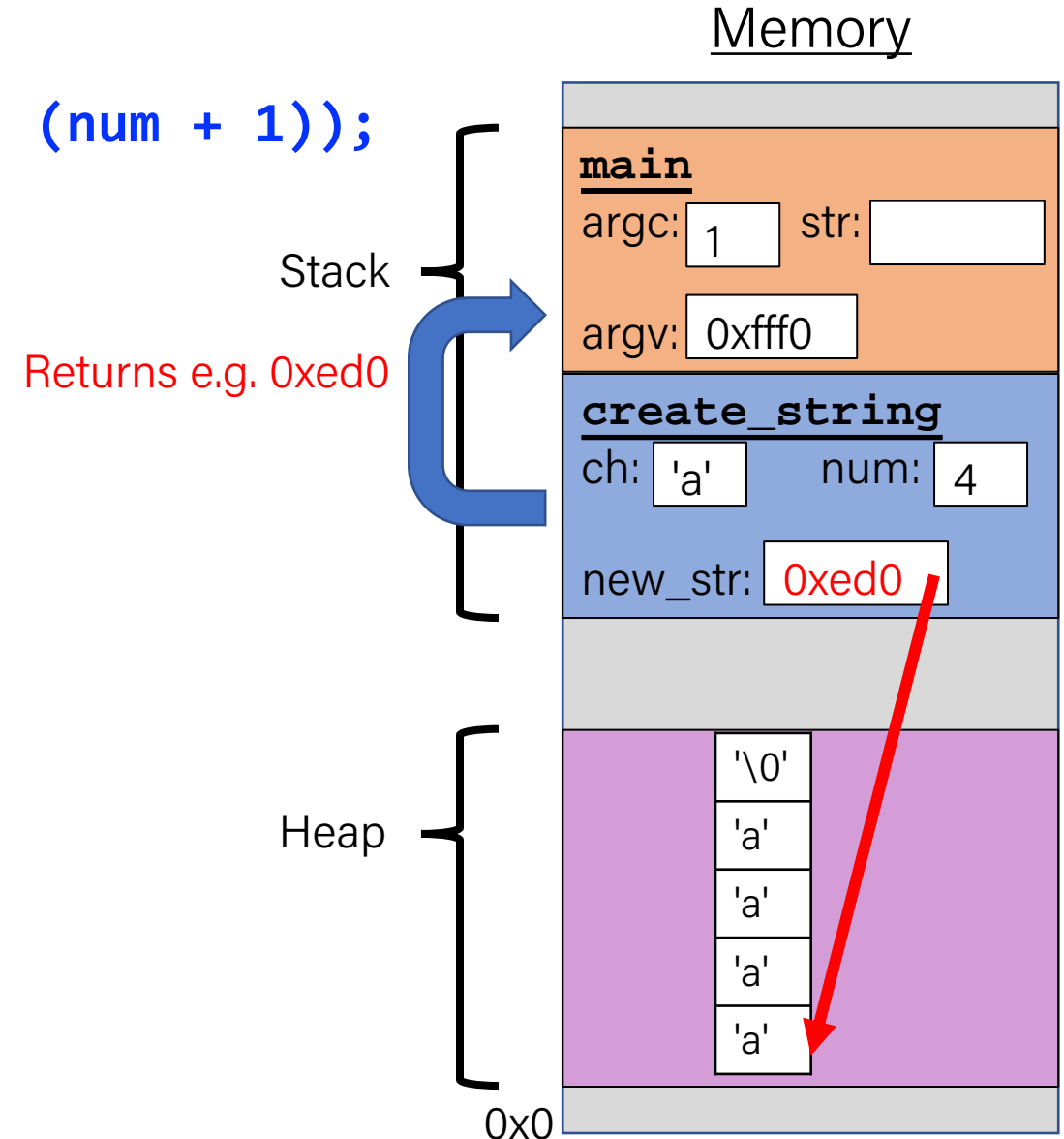
The Heap

```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:
argv: 0xfff0

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

Heap

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

0x0

95

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;
}
```
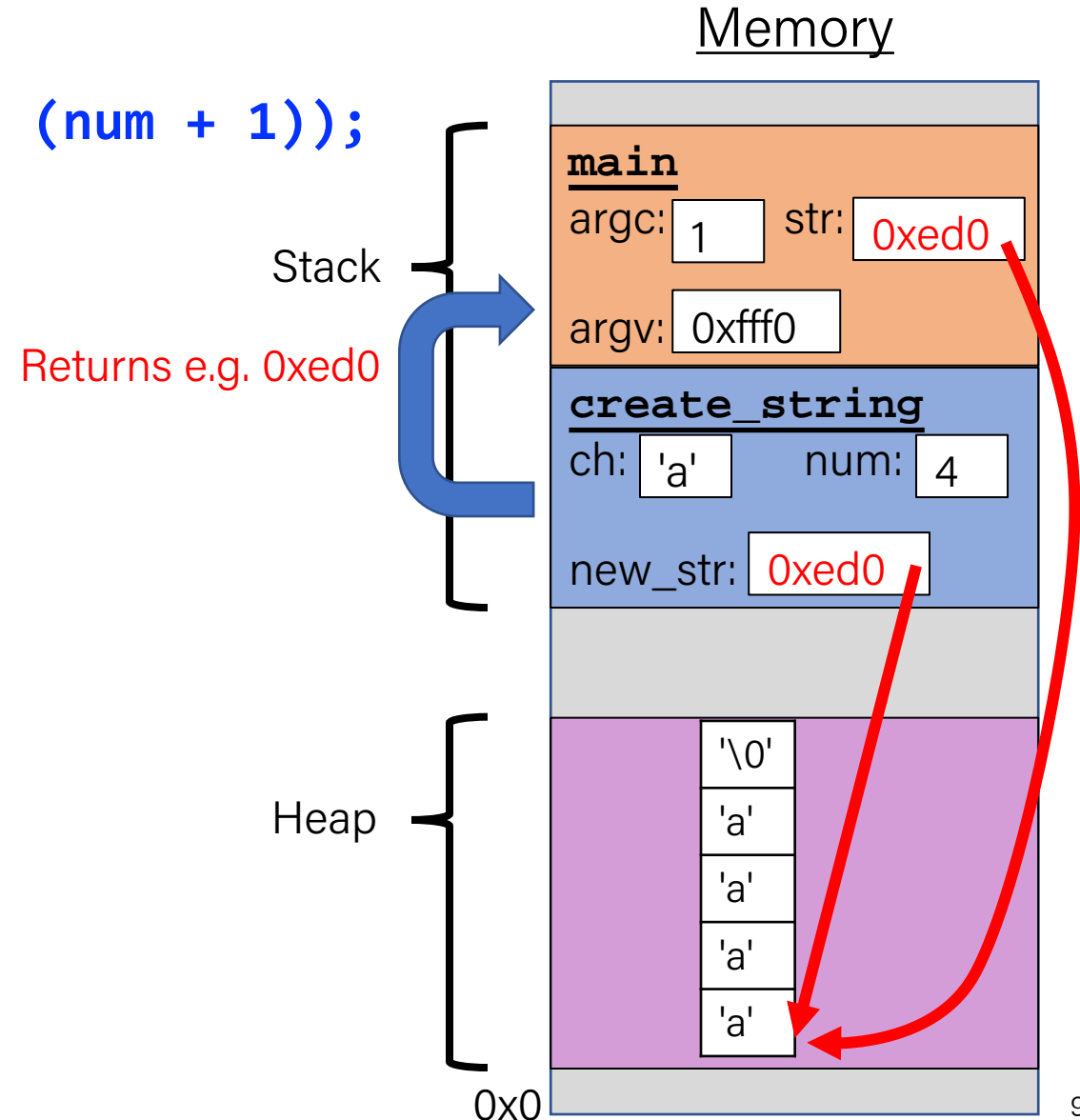
Returns e.g. 0xed0

Memory

Stack

Heap

main
argc: 1   str:
argv: 0xfff0

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

'\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;
}
```
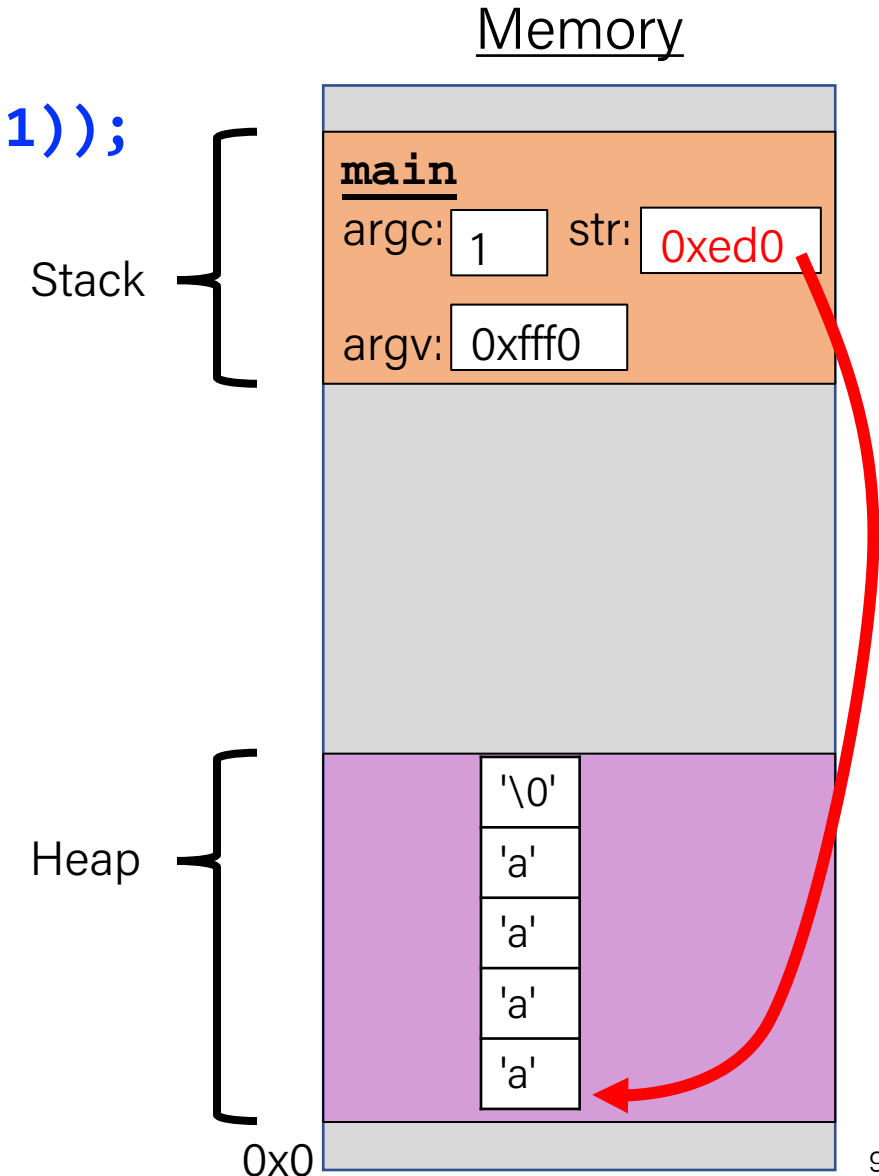
Memory

Stack

Returns e.g. 0xed0

main

argc: 1   str: 0xed0

argv: 0xfff0

create_string

ch: 'a'   num: 4

new_str: 0xed0

Heap

'\0'

'a'

'a'

'a'

'a'

0x0

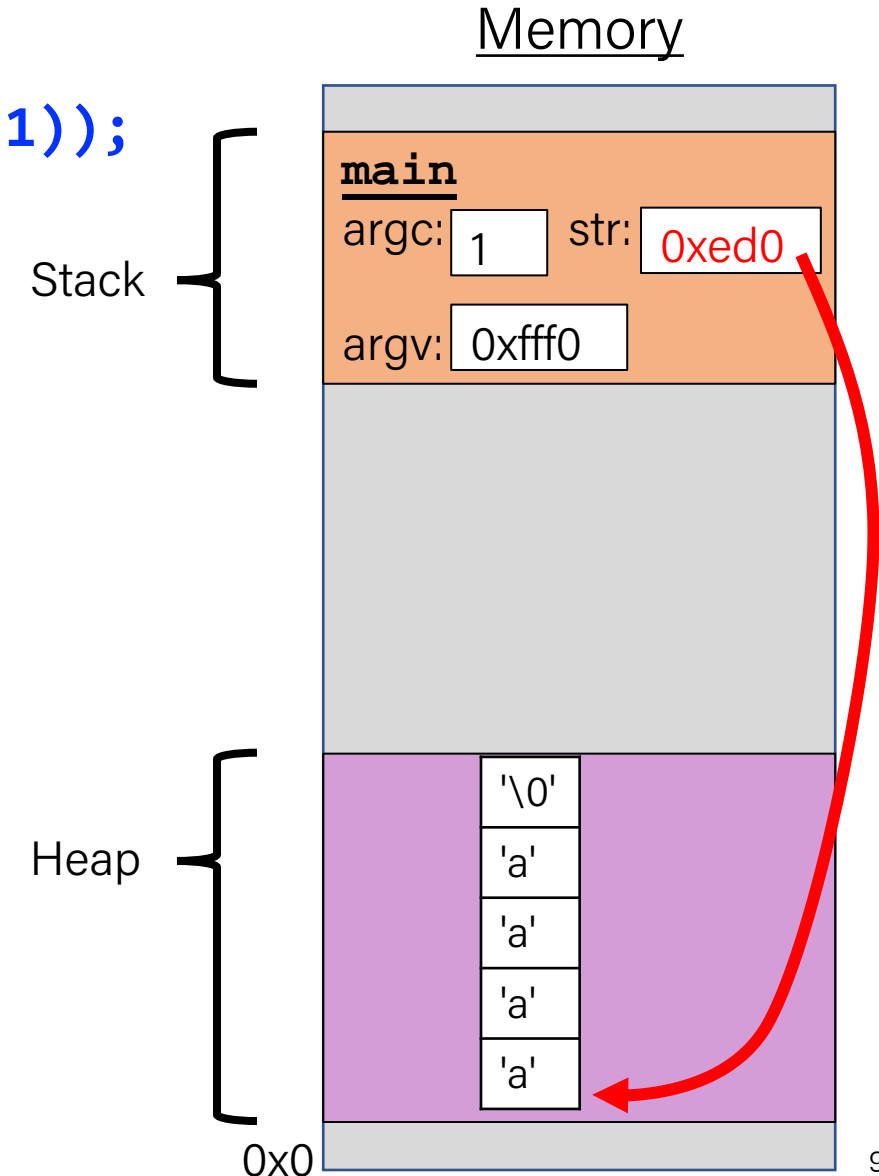97

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

main

argc: 1   str: 0xed0

argv: 0xfff0

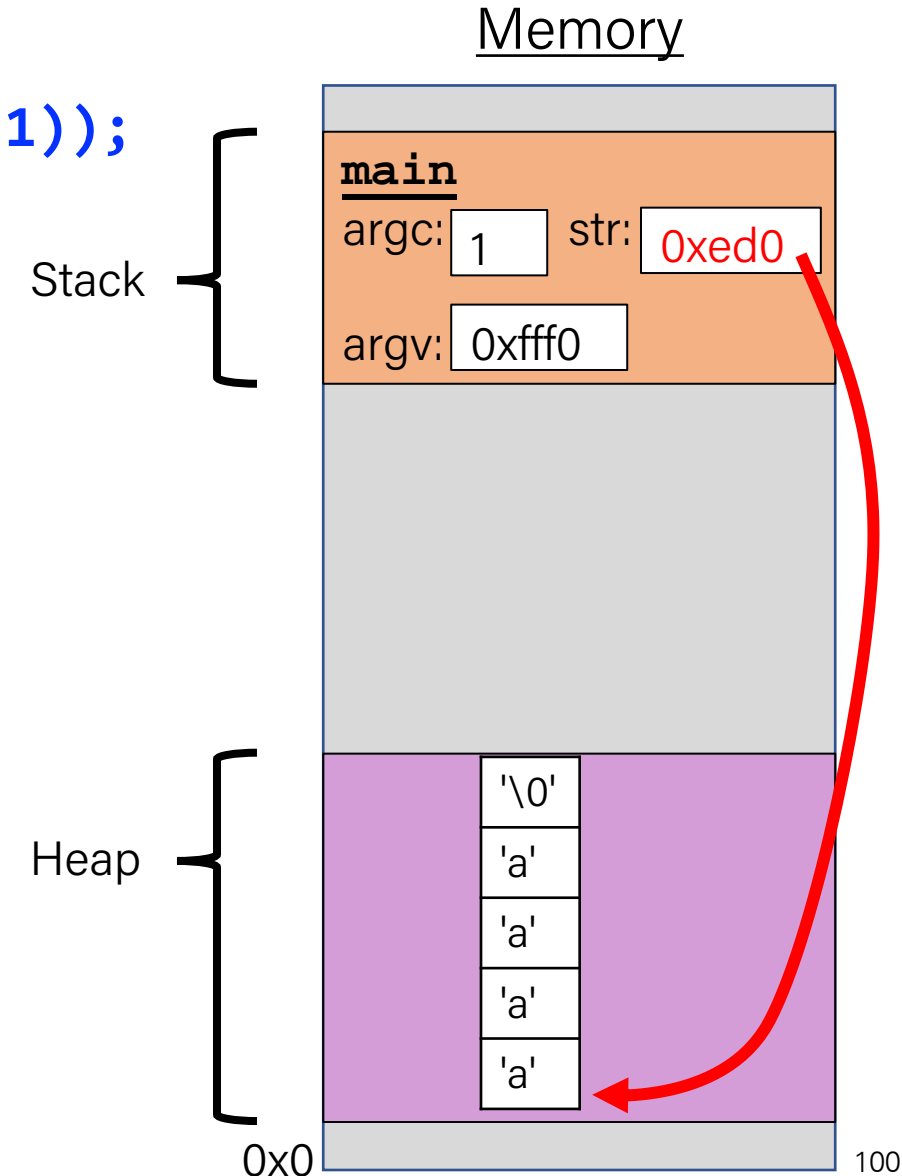Stack

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

99

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

100

Exercise: `malloc` multiples

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

```
1  int *array_of_multiples(int mult, int len) {
2      /* TODO: arr declaration here */
3
4      for (int i = 0; i < len; i++) {
5          arr[i] = mult * (i + 1);
6      }
7      return arr;
8  }
```

Line 2: How should we declare `arr`?

A. `int arr[len];`
B. `int arr[] = malloc(sizeof(int));`
C. `int *arr = malloc(sizeof(int) * len);`
D. `int *arr = malloc(sizeof(int) * (len + 1));`
E. Something else

🤔

Exercise: `malloc` multiples

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

```
1 int *array_of_multiples(int mult, int len) {
2     /* TODO: arr declaration here */
3
4     for (int i = 0; i < len; i++) {
5         arr[i] = mult * (i + 1);
6     }
7     return arr;
8 }
```

- Use a pointer to store the address returned by malloc.

- Malloc's argument is **the number of bytes** to allocate.

⚠ **This code is missing an assertion.**

Line 2: How should we declare `arr`?

A.   `int arr[len];`
B.   `int arr[] = malloc(sizeof(int));`
C.   `int *arr = malloc(sizeof(int) * len);`
D.   `int *arr = malloc(sizeof(int) * (len + 1));`
E.   Something else

Always `assert` with the heap

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

```
1  int *array_of_multiples(int mult, int len) {
2      int *arr = malloc(sizeof(int) * len);
3      assert(arr != NULL);
4      for (int i = 0; i < len; i++) {
5          arr[i] = mult * (i + 1);
6      }
7      return arr;
8  }
```

- If an allocation error occurs (e.g. out of heap memory!), malloc will return NULL. This is an important case to check **for robustness**.

- **assert** will crash the program if the provided condition is false. A memory allocation error is significant, and we should terminate the program.

Other heap allocations: `calloc`

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

calloc is like `malloc` that **zeros out** the memory for you—thanks, `calloc`!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- **calloc** is more expensive than `malloc` because it zeros out memory. Use only when necessary!

Other heap allocations: `strdup`

```
char *strdup(char *s);
```

strdup is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **malloc** and copy in the string yourself.

```
char *str = strdup("Hello, world!"); // on heap
str[0] = 'h';
```

Implementing strdup

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

```c
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?

```
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);
```

Recap

- Arrays in Memory

- Arrays of Pointers

- Pointer Arithmetic

- The Stack

- The Heap and Dynamic Memory

Next time: *realloc, Memory bugs*