Basic C Memory Model

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka, Daniel Gearhardt, Ash Fujiyama, Emily Shen

TAs:

Ahmed Abdellah	Ethan Weisberg	Maya Huizar
Angie Cao	Garrett O'Malley Kirsch	Meghana Vasireddy
August Fu	Hassan Rizwan	Perrie Quek
Caroline Begg	Iain Li	Sidharth Roy
Cathy Cao	Jerry Wang	Sydnie-Shea Cohen
Claire Lu	Juan Lopez	Vivi Li
Eric Sungwon Lee	Keith Mathe	Yousef AlRabiah

Poll Everywhere

pollev.com/cis2400

- How are you? Any Questions from last lecture?

Upcoming Due Dates TODO

- HW00 (Approx): Due Friday (Sept 6) @ 11:59 pm
  - Remember, we are super lenient with late days…
  - Don't stress out already pls
  - Take care of yourself.

- In general, there will be a ***lecture check in*** due before Lecture on Tuesdays!
  - Next one should be out sometime tomorrow morning

Lecture Outline

- Review
  - Bits, Bytes, Operators, and more.
- Revisiting: Char * & Char[]
- Global Memory
- The Stack
- The Heap
  - malloc() & free()
- Structs & C Data Structures

Base 2

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- The I'th bit represents 2^i
- We can use the prefix '0b' to denote base 2. (e.g. ***0b1101***)

Binary

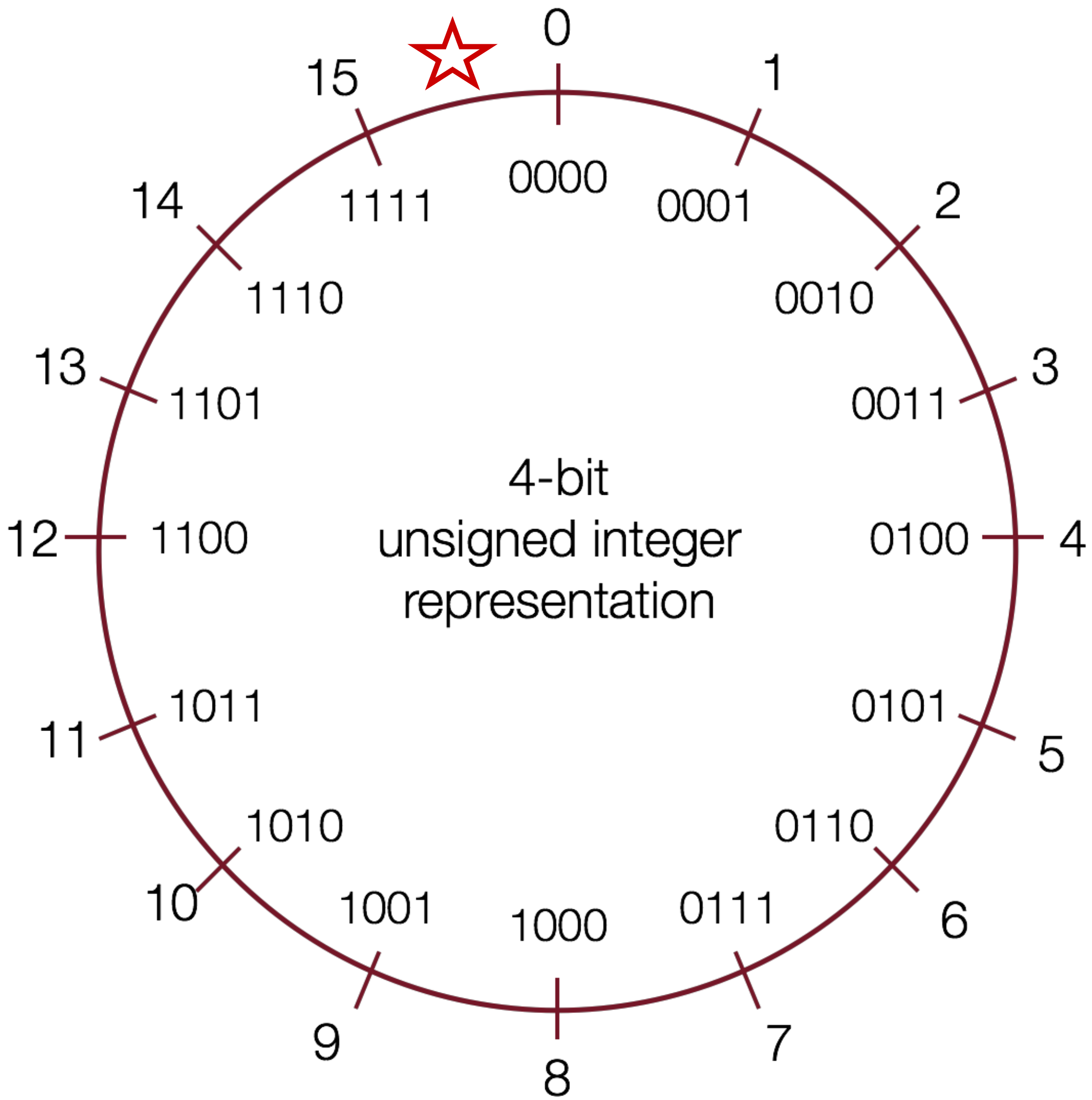
The Most Significant Bit (MSB)

The Least Significant Bit (LSB)

1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Note: This is One Byte (8 Bits).
The size of a char!

Unsigned Integers



Overflow

If you exceed the maximum value of your bit representation, you wrap around or ***overflow*** back to the smallest bit representation.

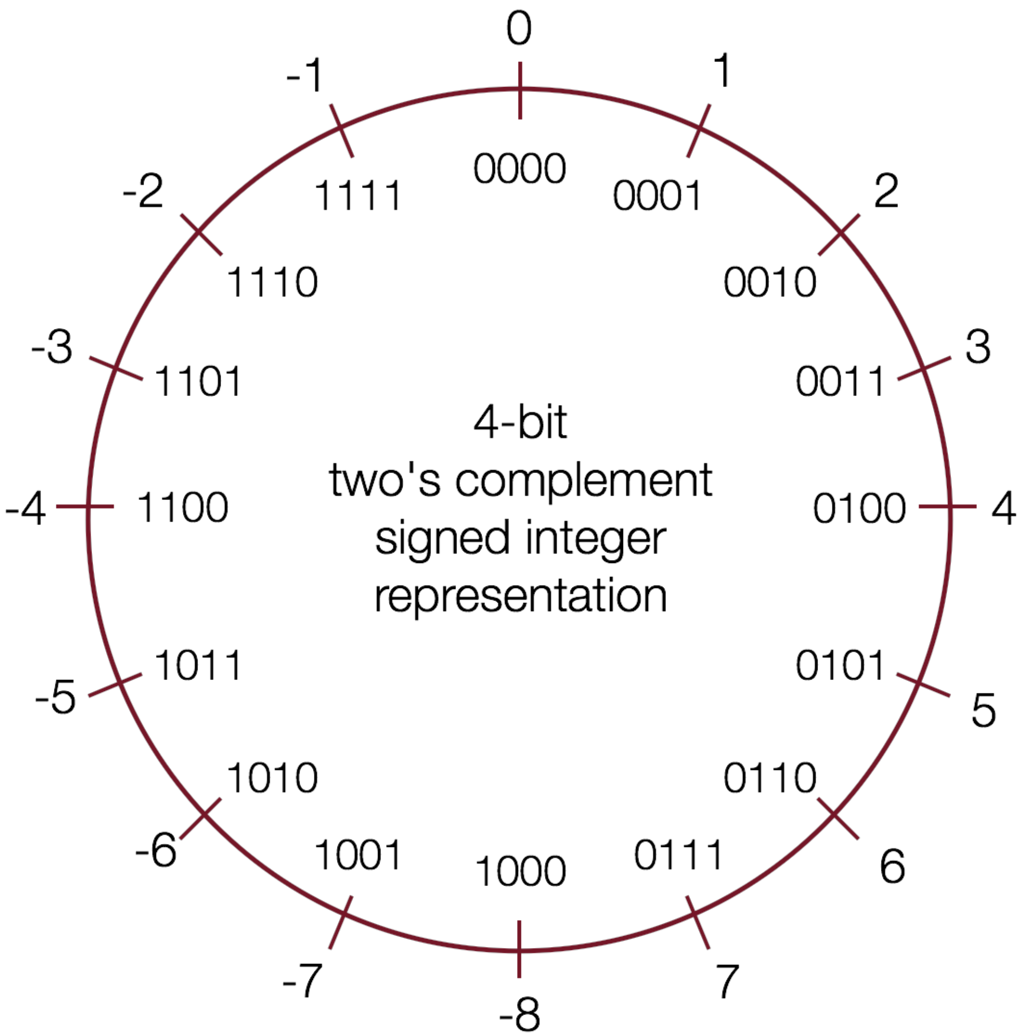
- 0b1111 + 0b1 = 0b0000

If you go below the minimum value of your bit representation, you wrap around or ***overflow*** back to the largest bit representation.

- 0b0000 - 0b1 = 0b1111

Here we're assuming we ***only*** have 4 bits to work with!

Two’s Compliment



Two's Compliment

- Here, we represent a positive number as **itself**, and its negative equivalent as the **two's complement of itself.**
- **The two's complement of a number is the binary digits inverted, plus 1.**
- **A nice consequence is all negative numbers have a 1 in the Most Significant Bit.**

Size Does Matter When Talking About Range

Type	Size (Bytes)	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

Bit Operator: & (and)

1 & 1 = 1

1 & 0 = 0

0 & 1 = 0

0 & 0 = 0

Only if both bits are one, will it stay one!

Bit Operator: | (or)

1 | 1 = 1

1 | 0 = 1

0 | 1 = 1

0 | 0 = 0

If either bits are one, will evaluate to one

Bit Operator: ^ (XOR)

1 ^ 1 = 0

ONLY IF a singular bit is one, will evaluate to one

1 ^ 0 = 1

0 ^ 1 = 1

0 ^ 0 = 0

More Bit Operators: << (left shift)

0b 0 0 1 0 1 << 1

0b 0 1 0 1 0

This operation shifts the bits ***n*** many times to the left.

More Bit Operators: >> (right shift)

0b 0 0 1 0 1 >> 1

0b 0 0 0 1 0

What happened to the LSB?

This operation shifts the bits ***n*** many times to the right.

Bits are “truncated” if they are right shifted by too much.

REMEMBER THIS

&& ***IS NOT &***

|| ***IS NOT |***

! IS NOT ~

Clarification on ~

- This "**flips**", "**negates**", or creates **the compliment** of a binary number.
- These terms are used sometimes interchangeably.

~ 0b 1 1 0 1 0 1

0b 0 0 1 0 1 0

Poll Everywhere

pollev.com/cis2400

- What will be the resulting value of num be in binary?

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

  unsigned char num = 0xff;

  num = num & 0xf0;
  num = num ^ 0x01;
  num = ~num;

  ...
}
```

A. 0b11110001

B. 0b10110001

C. 0b11110000

D. 0b00001110

E. What is binary?

Poll Everywhere

pollev.com/cis2400

- What will be the resulting value of num be in binary?

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  unsigned char num = 0xff;

  num = num & 0xf0;
  num = num ^ 0x01;
  num = ~num;

  ...
}
```

A. 0b11110001

B. 0b10110001

C. 0b11110000

D. 0b00001110

E. What is binary?

Goals for This Lecture:

- C Strings from the Perspective of Memory
- Char *s vs Char []'s
- To understand where data is stored over the ***lifetime*** of a C program

- Three types of data allocation:
  - Static (e.g. Globals)
  - Automatic (e.g. Local Variables & the stack)
  - Dynamic (e.g. stored on the Heap)
    - Covered by Travis next week!

Lecture Outline

- Review
  - Bits, Bytes, Operators, and more.
- Revisiting: Char * & Char[]
- Global Memory
- The Stack
- The Heap
  - malloc() & free()
- Structs & C Data Structures

Revisiting: Char * & Char[]

- C doesn't know what a "string" is.
- A string in C is simply an ***array of characters*** with a special ending value ***"\0"***

`"Miso"`

index	*0*	*1*	*2*	*3*	*4*
char	`'M'`	`'i'`	`'s'`	`'o'`	`'\0'`

This is what the string might look like directly in memory.

Revisiting: Char * & Char[]

```
char str[5];
str[0] = 'M'
str[1] = 'i'
str[2] = 's'
str[3] = 'o'
str[4] = '\0'
```

index	*0*	*1*	*2*	*3*	*4*
char	`'M'`	`'i'`	`'s'`	`'o'`	`'\0'`

And we're done right?

We need the Null Terminator ‼

- Here we have an array of 5 chars, where each char is a single byte
  - ***In total, 5 bytes!***

Revisiting: Char * & Char[]

`str[0] = 'M'`

- This *literally* inserts the ascii value of 'M' (0x4D) into `str[0]`

index	0	1	2	3	4
char	`'M'`	`'i'`	`'s'`	`'o'`	`'\0'`

Human readable; great for teaching!

ascii values

index	0	1	2	3	4
value	0x4D	0x69	0x73	0x6F	0x00

Under the hood it might actually look like this.....

Reminder: Two Hex digits are one byte. 0xff = 0b11111111

Revisiting: Char * & Char[]

- Each character takes up one byte of memory
- In C, things are byte addressable meaning that you can grab things "one byte at a time".
- Which should make sense if chars are one byte…

Lecture Outline

- Review
  - Bits, Bytes, Operators, and more.
- Revisiting: Char * & Char[]
- Strings as Arrays of Memory
- Global Memory
- The Stack
- The Heap
  - malloc() & free()
- Structs & C Data Structures

Strings as Arrays of Memory

- ❖ A more realistic view of arrays.

```
char str[5];
str[0] = "M"
str[1] = "i”
str[2] = "s"
str[3] = "o”
str[4] = "\0"
```

These are ***addresses.*** They show where the characters in this array are stored in memory.

Address	0xffff00	0xffff01	0xffff02	0xffff03	0xffff04
char	'M'	'i'	's'	'o'	'\0'

Strings as Arrays of Memory

- Notice that these characters are literally next to each other.

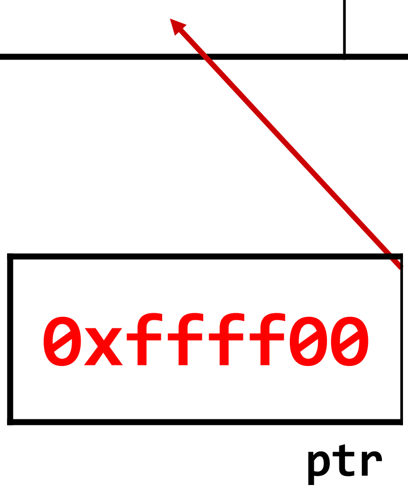
These ***addresses*** go from `0xffff0` to `0xffff04`.

Address	`0xffff00`	`0xffff01`	`0xffff02`	`0xffff03`	`0xffff04`
char	`'M'`	`'i'`	`'s'`	`'o'`	`'\0'`

Strings as Arrays of Memory

```
char str[5] = “Miso”;
char *ptr = str;
```

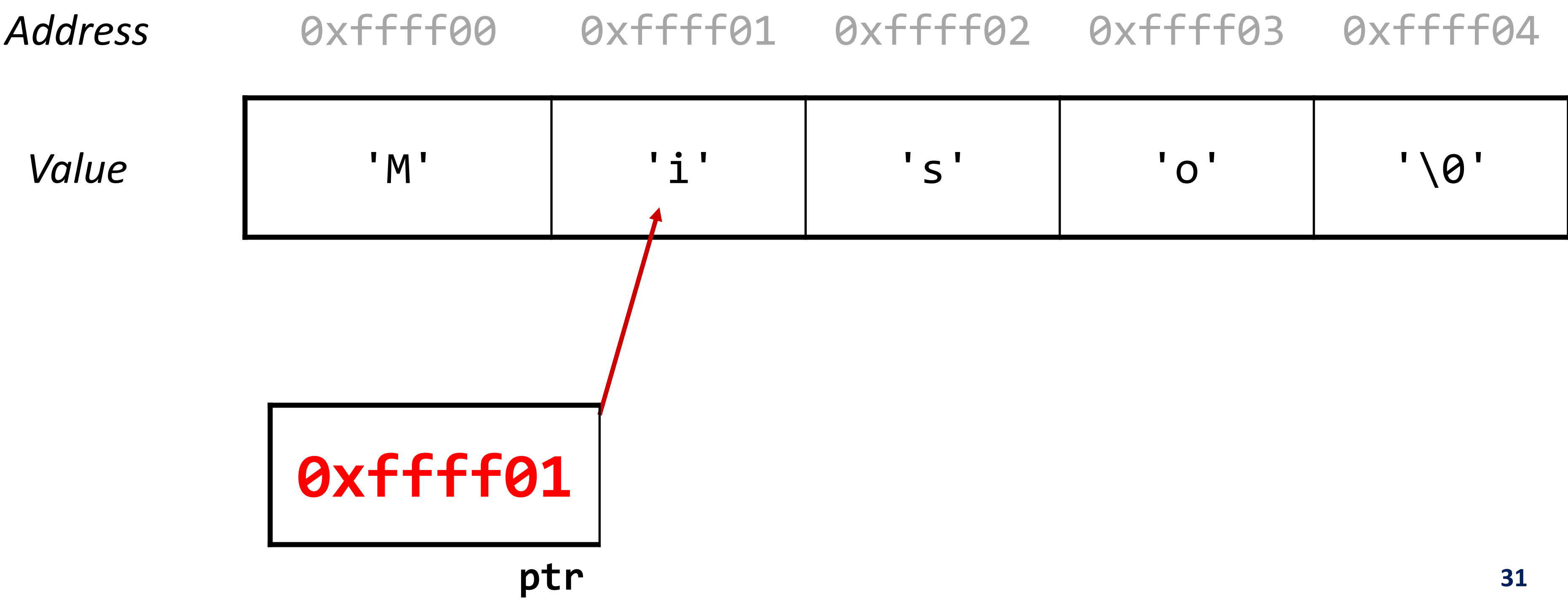
Address	0xffff00	0xffff01	0xffff02	0xffff03	0xffff04
Value	'M'	'i'	's'	'o'	'\0'



The value of the pointer ***is*** the address of the first character in the array

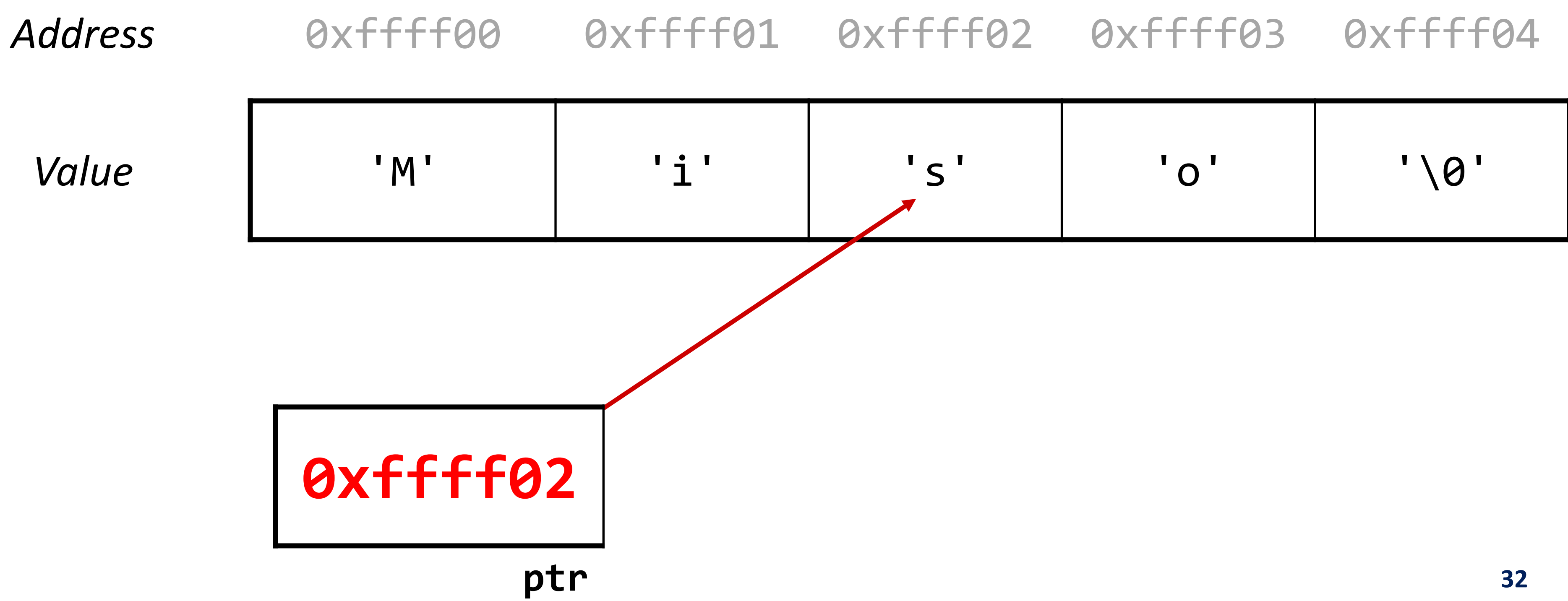
Strings as Arrays of Memory

```
char str[5] = “Miso”;
char *ptr = str;
ptr = &str[1]
```



Strings as Arrays of Memory

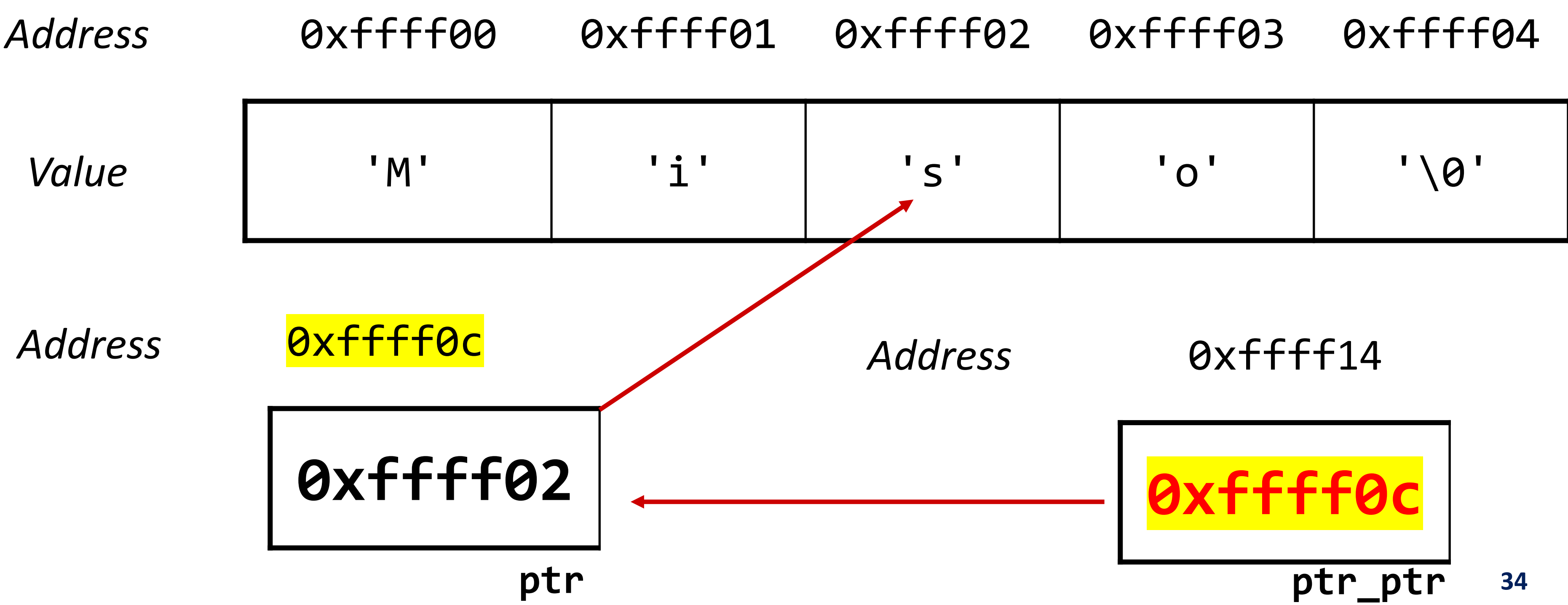
```
char str[5] = “Miso”;
char *ptr = str;
ptr = &str[2]
```



Strings as Arrays of Memory

```
char str[5] = “Miso”;
char *ptr = str;
ptr = &str[2]
char **ptr_ptr = &ptr;
```

Typically, as you declare variables, they get initialized right next to each other.



Strings as Arrays of Memory

Address	`0xffff00`	`0xffff01`	`0xffff02`	`0xffff03`	`0xffff04`
Value	`'M'`	`'i'`	`'s'`	`'o'`	`'\0'`

Address `0xffff0c`

`0xffff02`

`ptr`

Address `0xffff14`

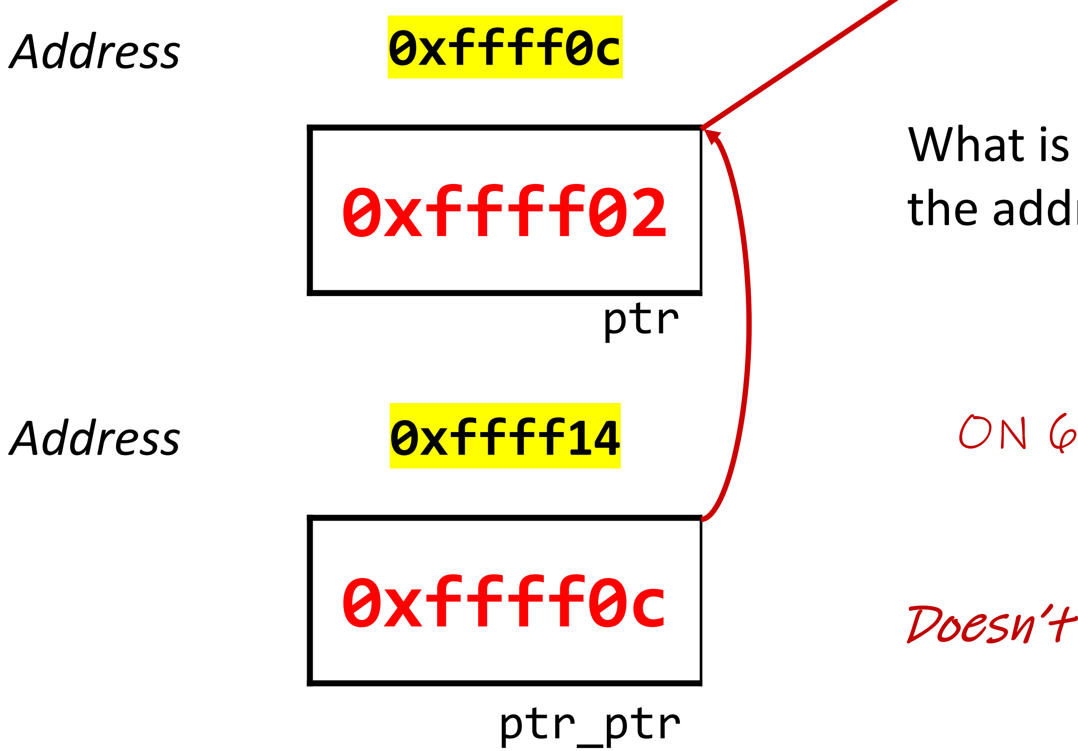
`0xffff0c`

`ptr_ptr`

- ***& Operator*** grabs the address of a variable
- *** Operator** grabs the value @ the address.
- A pointer is a variable that holds the address of another variable
- We **have to be explicit** about its' type

Strings as Arrays of Memory

Address	0xffff00	0xffff01	0xffff02	0xffff03	0xffff04
Value	'M'	'i'	's'	'o'	'\0'



What is the difference/distance between the address **0xffff0c** and **0xffff14**?

Its 0x8!

ON 64Bit Computers, Pointers are *ALWAYS 8 Bytes.*

*Doesn't matter if they're char * or char ** or char ***********

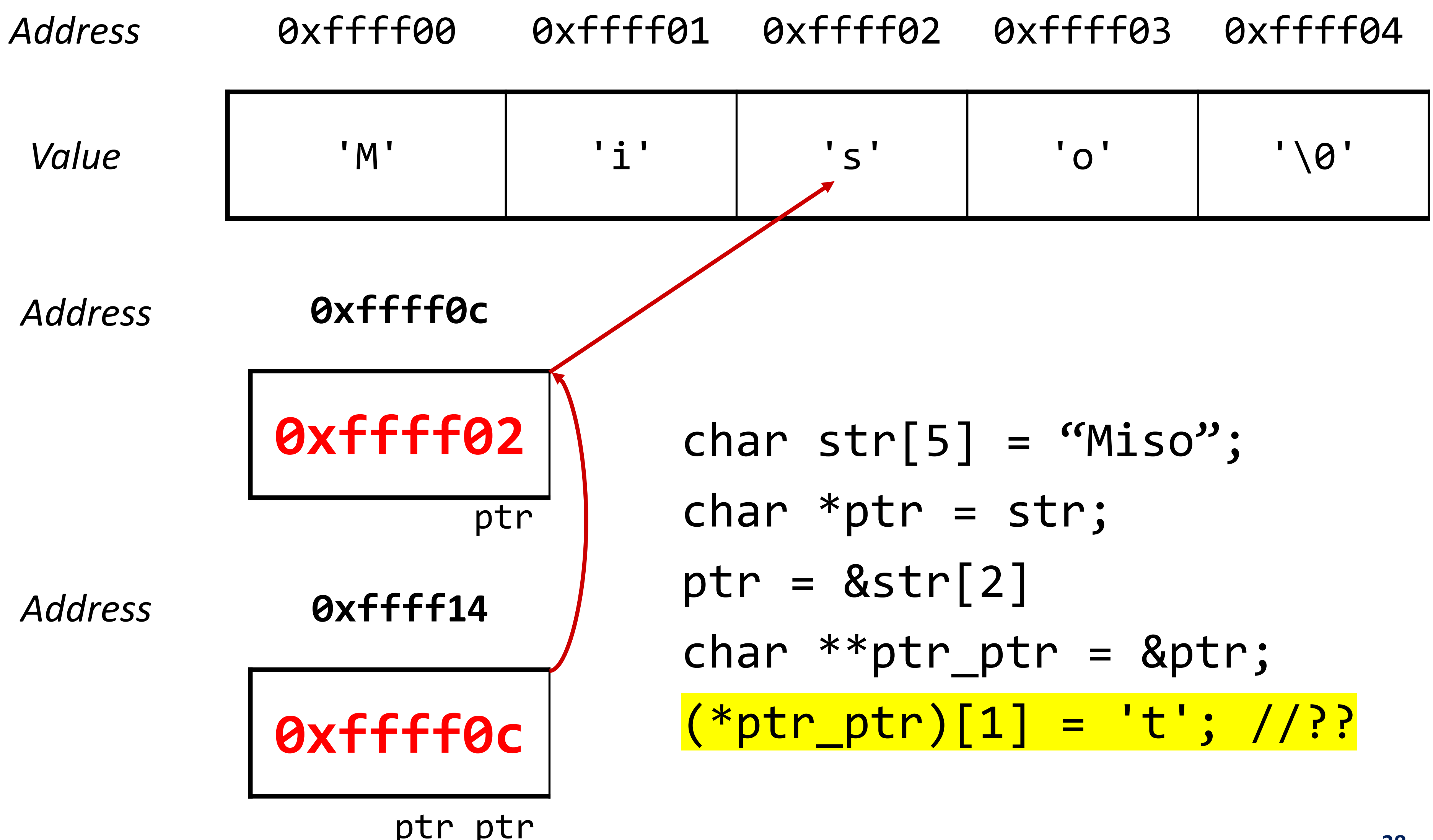
C Strings as Arguments

- As a parameter, it is always passed as a **char ***.
- C passes the ***location or address*** of the first character rather than a copy of the whole array.

```
int doSomethingForMe(char *str) {
        str[2] = 'l'; // modifies original string!
        printf("%s\n", str);  // prints milo
}

char ourString[5];
...     // e.g. this string is “Miso”
doSomethingForMe(myString);
printf("%s\n", str); // prints milo
```

We can still use a char * the same way as a char[].

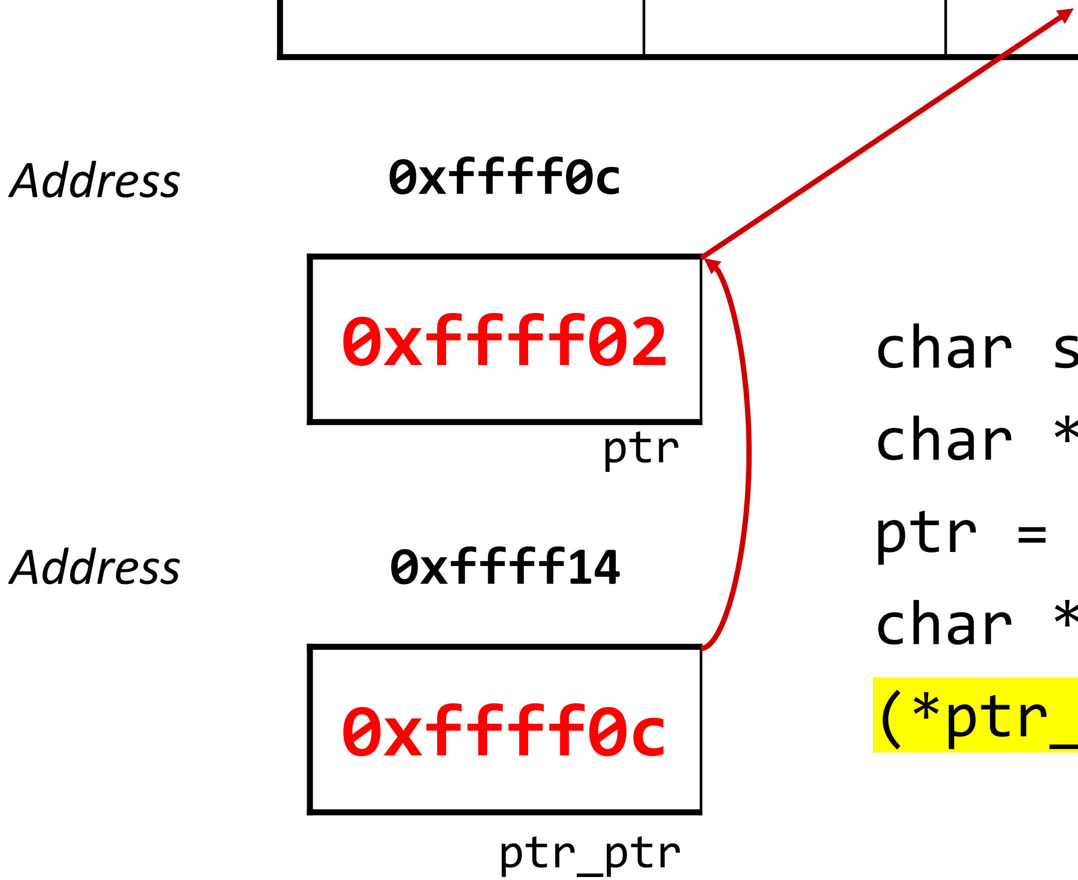
Strings as Arrays of Memory




```
char str[5] = “Miso”;
char *ptr = str;
ptr = &str[2]
char **ptr_ptr = &ptr;
(*ptr_ptr)[1] = 't'; //??
```

Strings as Arrays of Memory

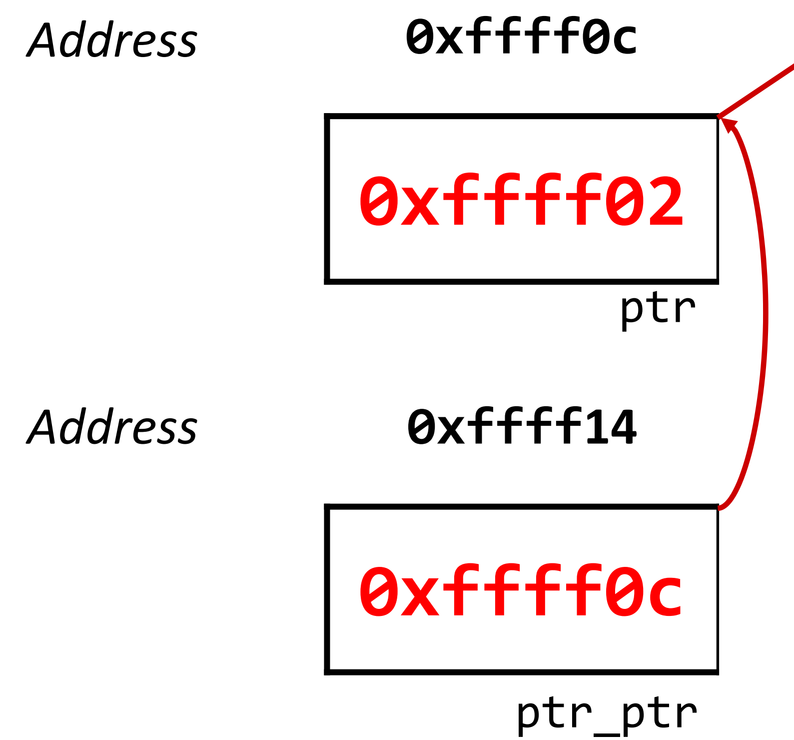
Address	0xffff00	0xffff01	0xffff02	0xffff03	0xffff04
Value	'M'	'i'	's'	't'	'\0'




```
char str[5] = “Miso”;
char *ptr = str;
ptr = &str[2]
char **ptr_ptr = &ptr;
(*ptr_ptr)[1] = 't’; //??
```

Strings as Arrays of Memory

Address	0xffff00	0xffff01	0xffff02	0xffff03	0xffff04
Value	'M'	'i'	's'	't'	'\0'



`(*ptr_ptr)[1] = 't’; //??`

`*ptr_ptr`: This gets the value at the address stored in ptr_ptr.

`(addr)[1]`: This accesses the value at an address offset by ***one unit*** from addr.

The unit size is determined by the type we are pointing to.

Char * vs Char []

- char * is an 8-byte pointer
  - it stores an address of a character
- char[] is an array of characters
  - it stores the actual characters of a string
- char[] is automatically passed as a char *
  - (pointer to its first character)

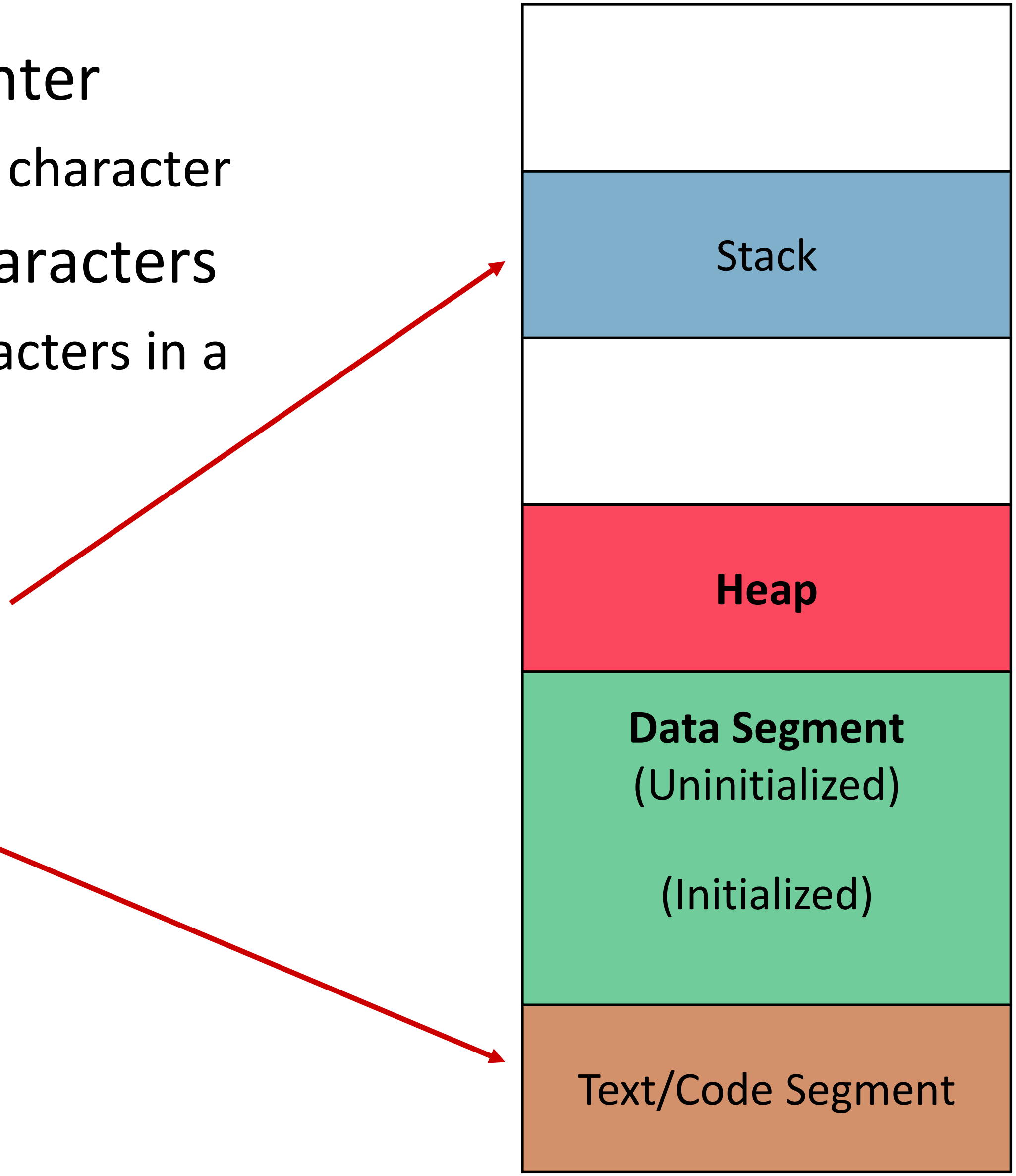
Lecture Outline

- Review
  - Bits, Bytes, Operators, and more.
- Revisiting: Char * & Char[]
- Strings as Arrays of Memory
- C Memory
  - Memory Diagram
  - Global Memory
  - The Stack

Memory Diagram of C Program

- char * is an 8-byte pointer
  - it stores an address of a character
- char[] is an array of characters
  - it stores the actual characters in a string

```
char str[5] = "Miso";

char *ptr = "Ube";
```

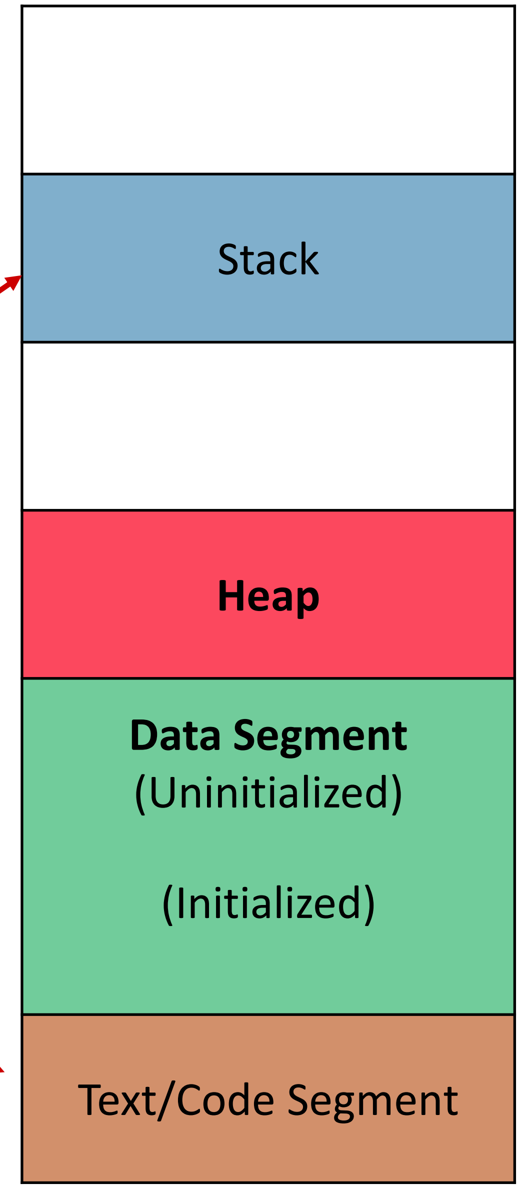


Memory Diagram of C Program

- Stack Memory
  - Readable and Modifiable
- Text/Code Segment
  - Readable Only

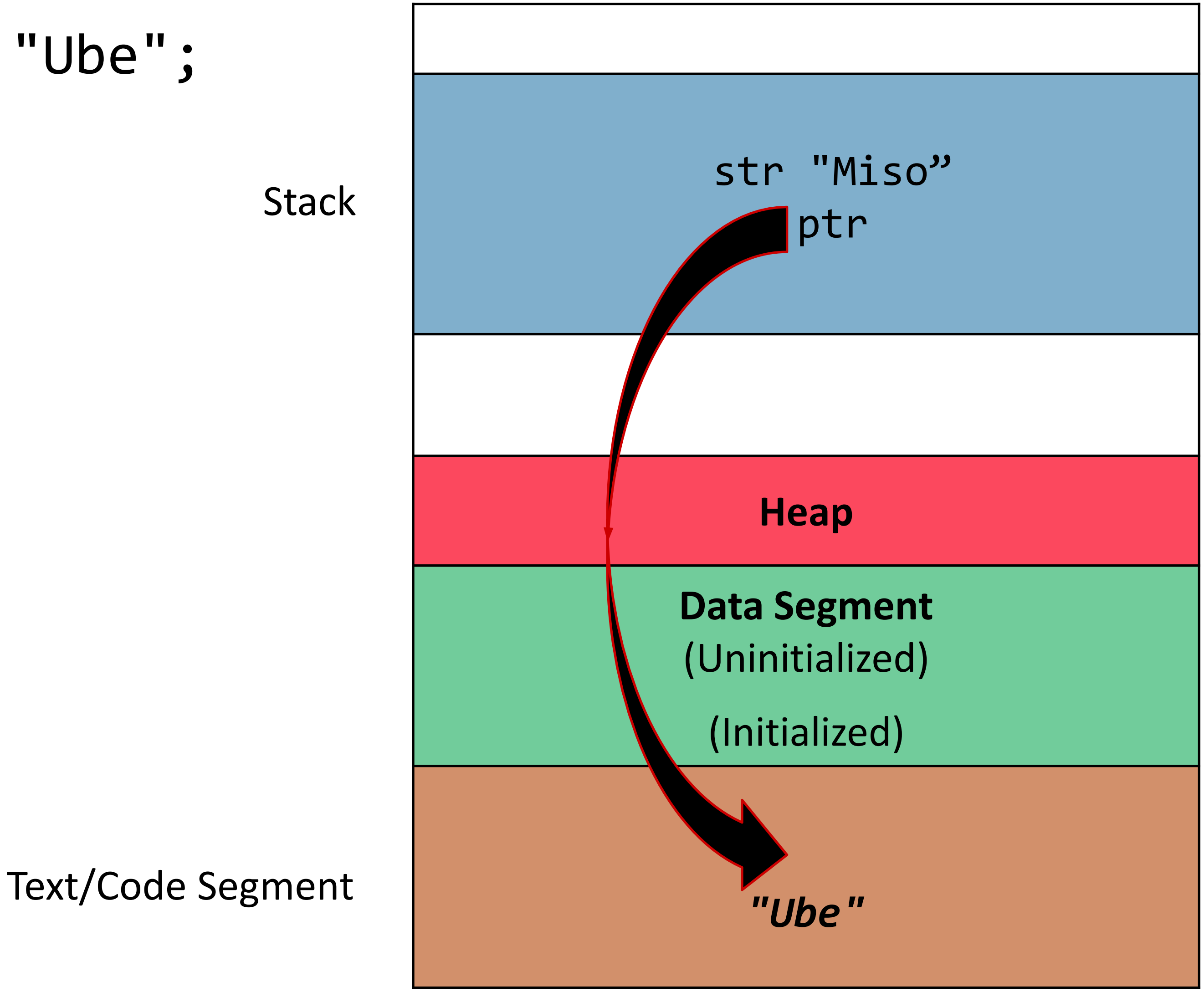
```
char str[5] = "Miso";

char *ptr = "Ube";
```

Note, here the arrows are showing you where the strings themselves live. NOT THE VARIABLES

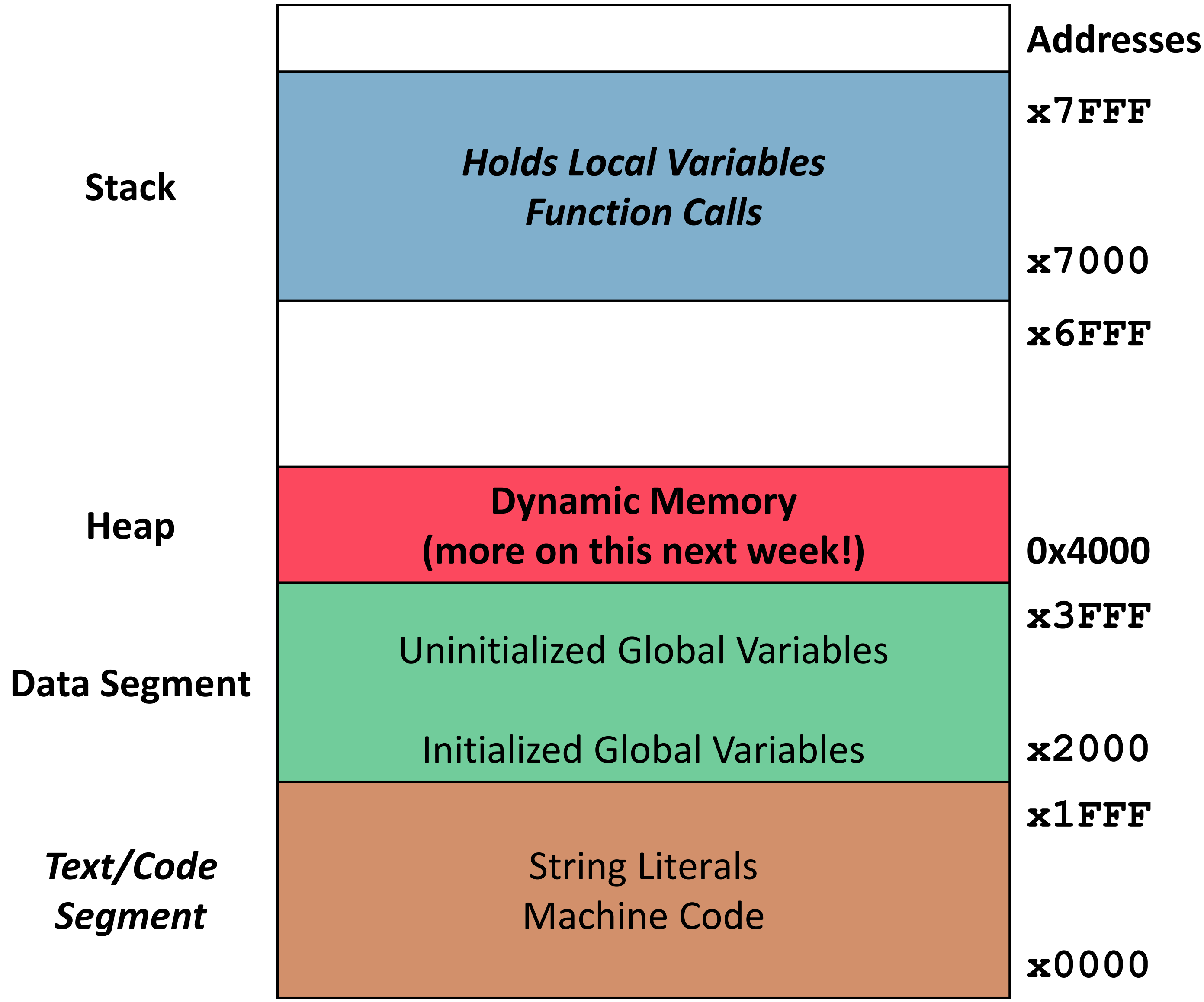


Memory Diagram of C Program

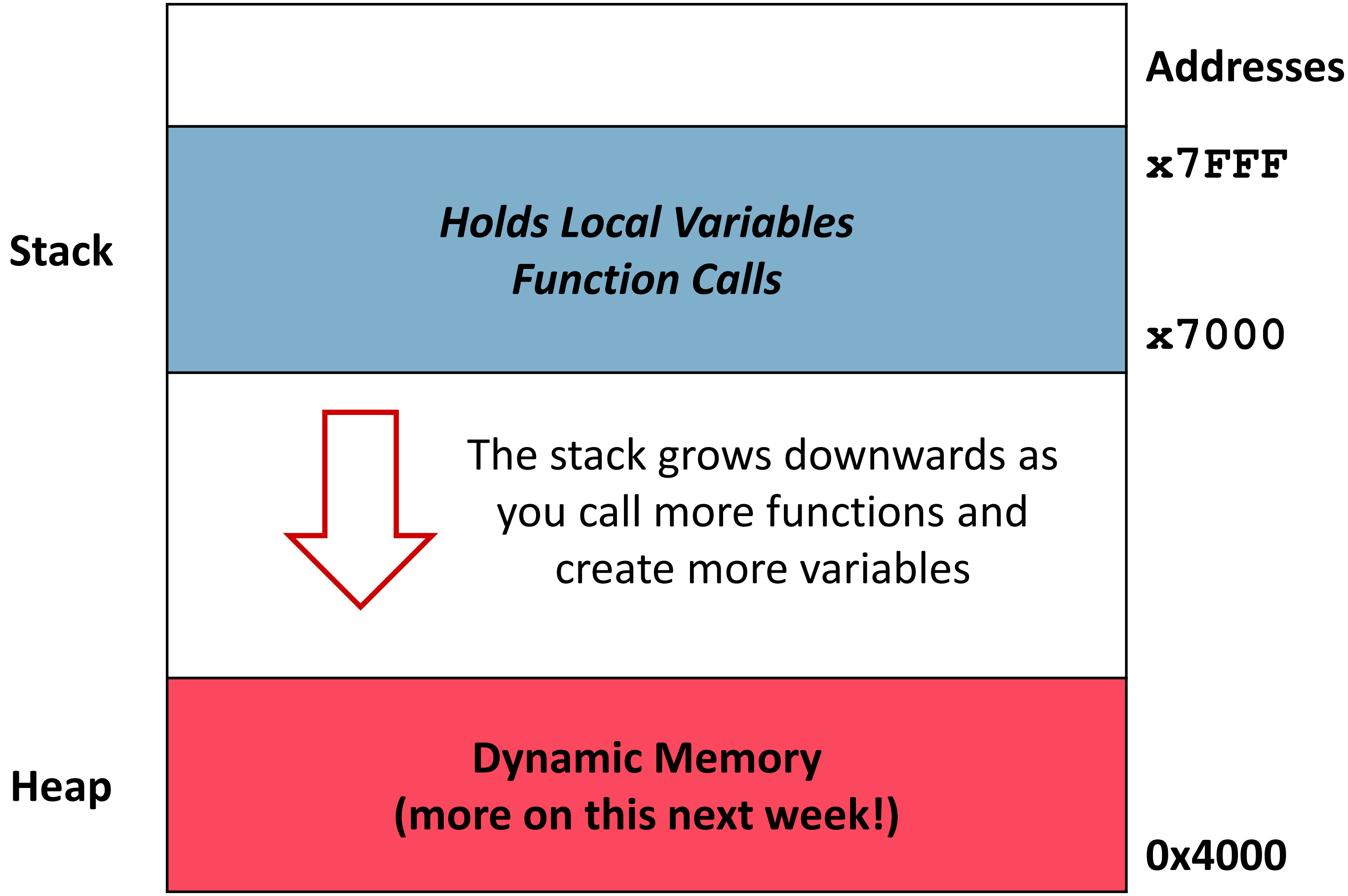
```
char str[5] = "Miso";
char *ptr = "Ube";
```



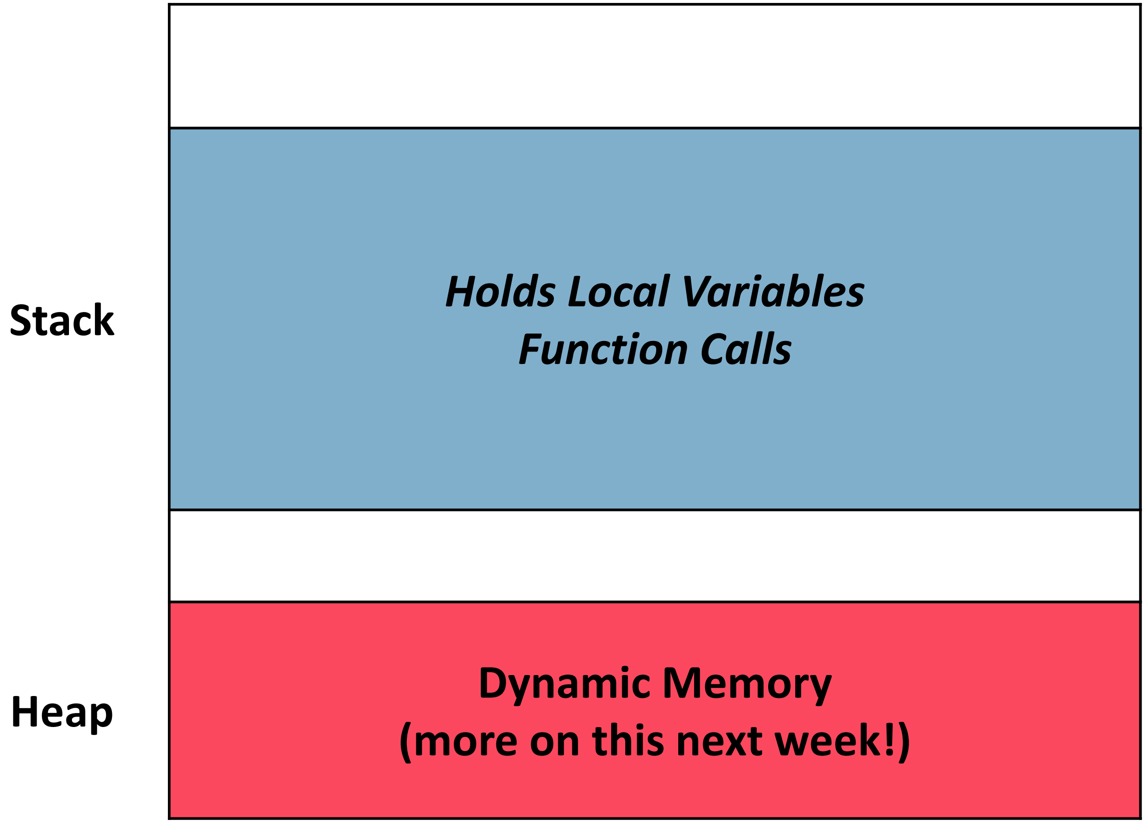
Memory Diagram of C Program



Memory Diagram of C Program



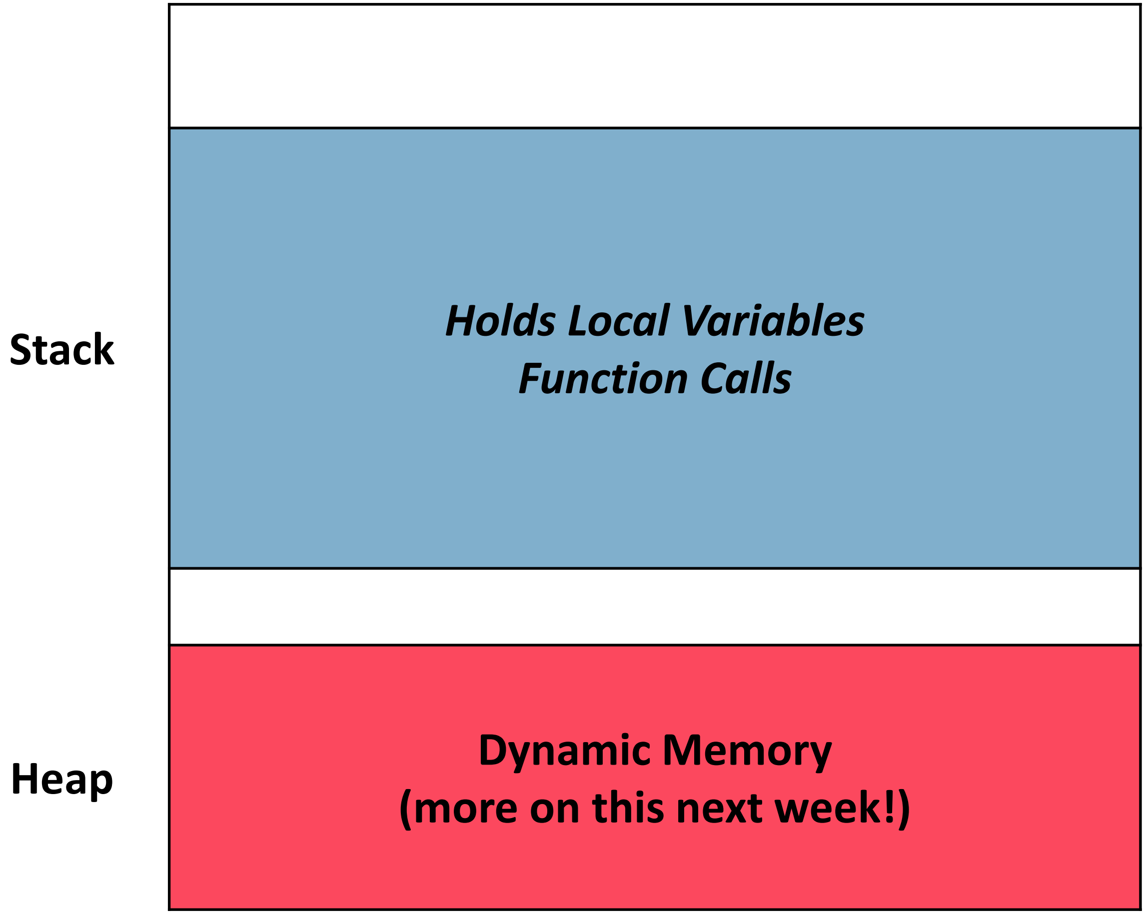
Memory Diagram of C Program



As you begin to use up all the stack space…

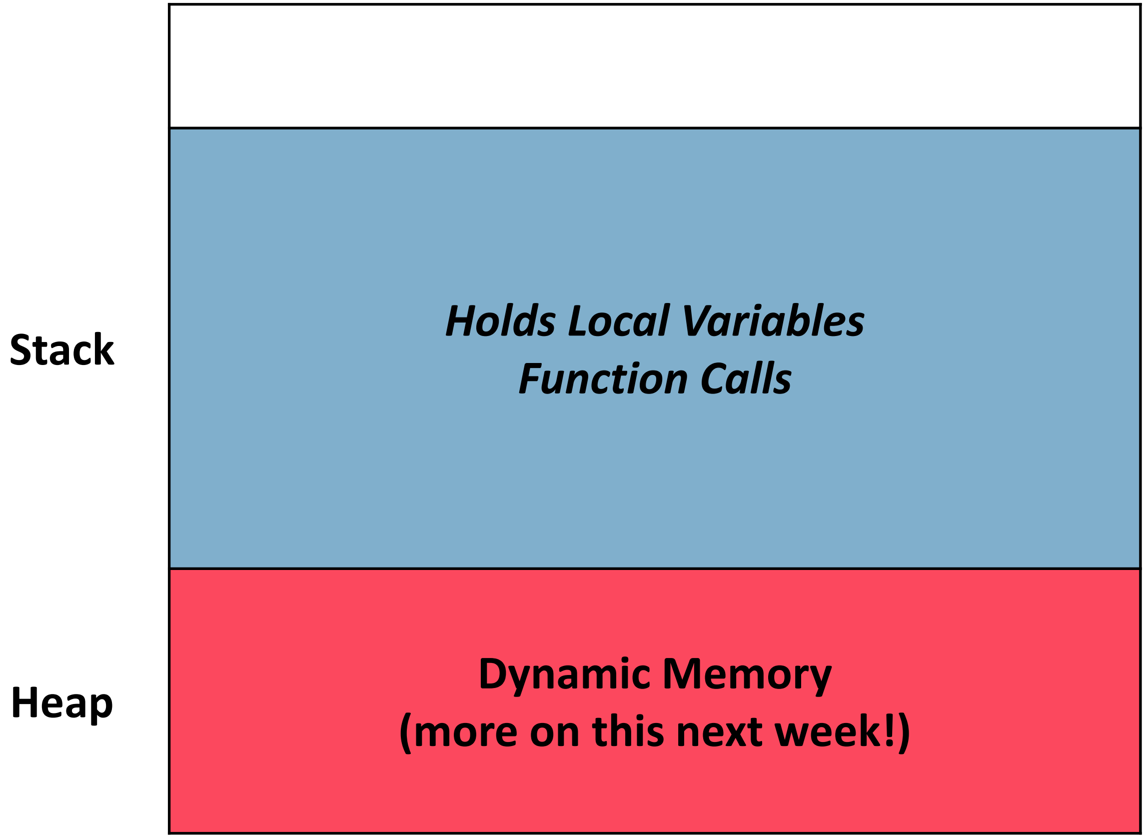
(e.g. with recursive calls…)

Memory Diagram of C Program



The stack can flow into the heap…

Memory Diagram of C Program



Or in other words…

stack overflow…

No more memory for stack to grow. :/

Lecture Outline

- Review
  - Bits, Bytes, Operators, and more.
- Revisiting: Char * & Char[]
- Strings as Arrays of Memory
- C Memory
  - Memory Diagram
  - Global Memory
  - The Stack

Global Variables in C

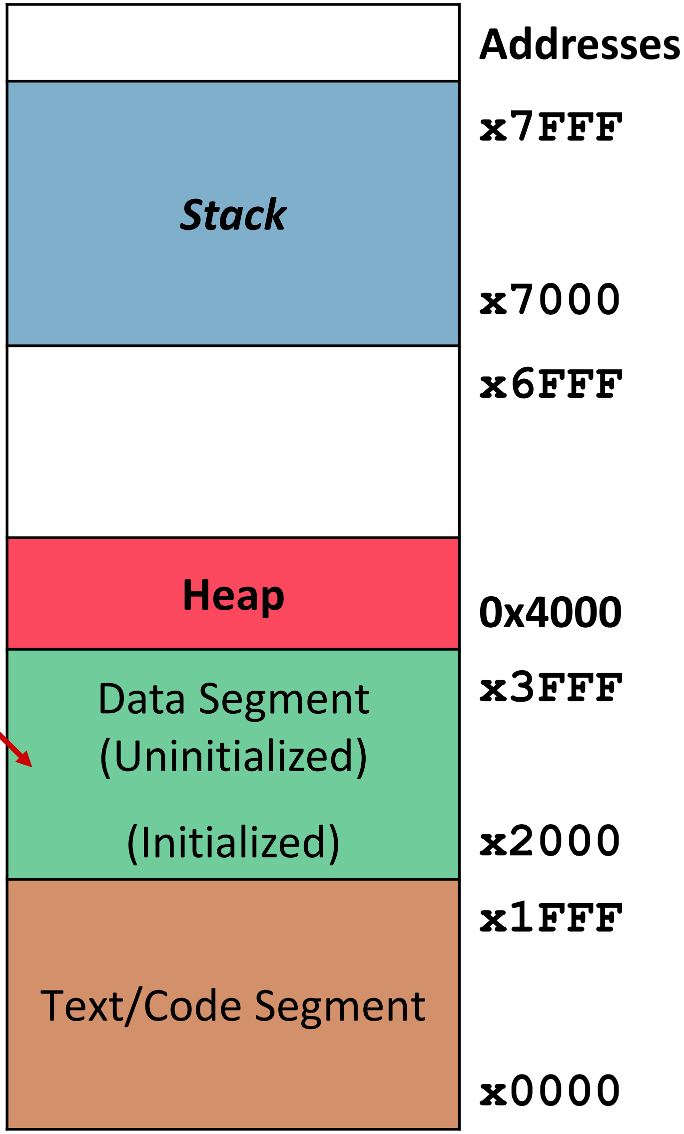
```
#include <stdio.h>
#include <stdlib.h>

int x = 0;

void incr_globals() {
  x++;
}

int main() {
  printf("x: %d\n", x);  // prints 0
  incr_globals();
  printf("x: %d\n", x);  // prints 1
  return EXIT_SUCCESS;
}
```

Declaring a variable outside of a function makes it "global"

- Global variables exist outside of any function, can be accessed from any function
- Exist throughout the entire lifespan of a program

Global Variables in Memory

- Global variables can be stored at a static (un-changing) address.
- Reading/writing to that variable just involves going to that static memory location.
- These variable are "allocated as soon as the program is loaded. Program exiting will "de-allocate" the variable.



Variables in Functions

- Variables declared outside of functions (global variables) exist over the lifetime of the program

- What about variables in functions?
  - Function parameters, local variables, return values etc.
  - Exist only for the lifetime of an instance of execution of a function
  - There may be multiple instances of a function at a time, needing multiple (but separate) sets of variables (e.g. recursion)
  - **Where do these exist in memory?**

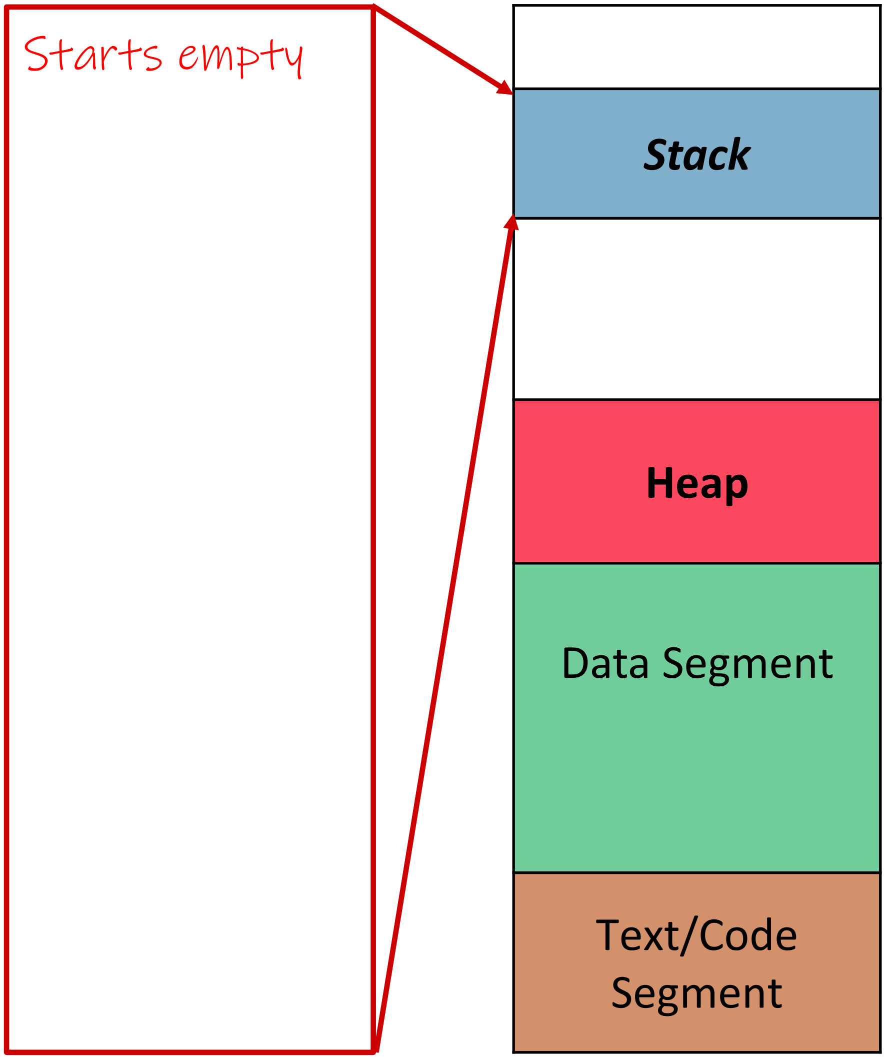
The Stack

- Local variables are stored in a portion of memory called the "Stack" sometimes called the "Call Stack".
  - Whenever a function is invoked, we "push" a "stack frame" for that function onto the top of the stack.
  - The stack frame contains important information about the execution of the function and has space for every local variable
  - When a function exits, its stack frame is "popped" and the local variables are "deallocated"

Stack Example 1:

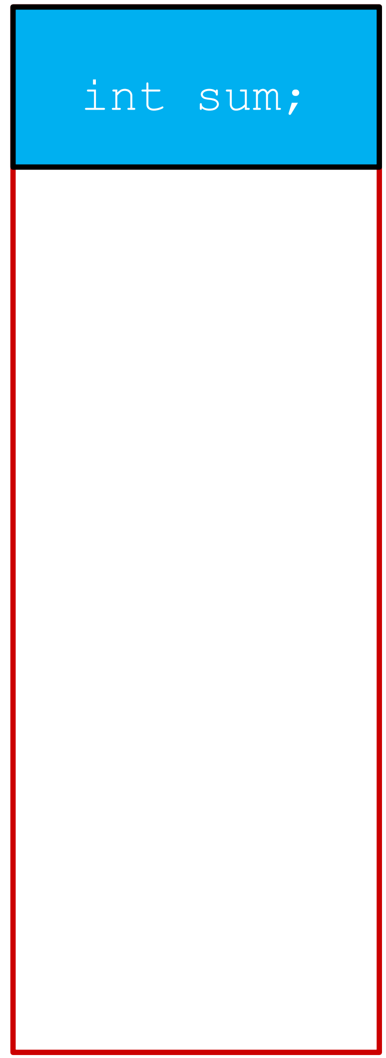
```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

Zooming in on the bottom of the stack



Stack Example 1:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);   // ← current line
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

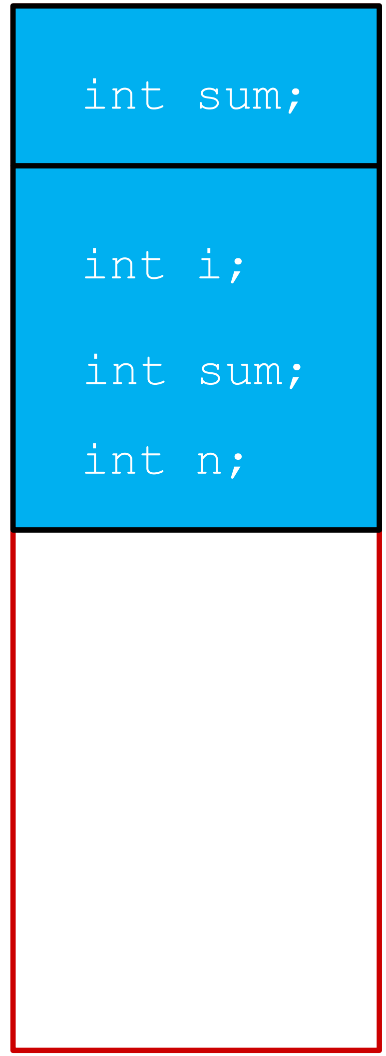


Stack frame for `main()`

Stack frame for main is created when CPU starts executing it

Stack Example 1:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      sum += i;
   }
   return sum;
}

int main() {
   int sum = sum(3);
   printf("sum: %d\n", sum);
   return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `sum()`

Stack Example 1:

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

`int sum;`

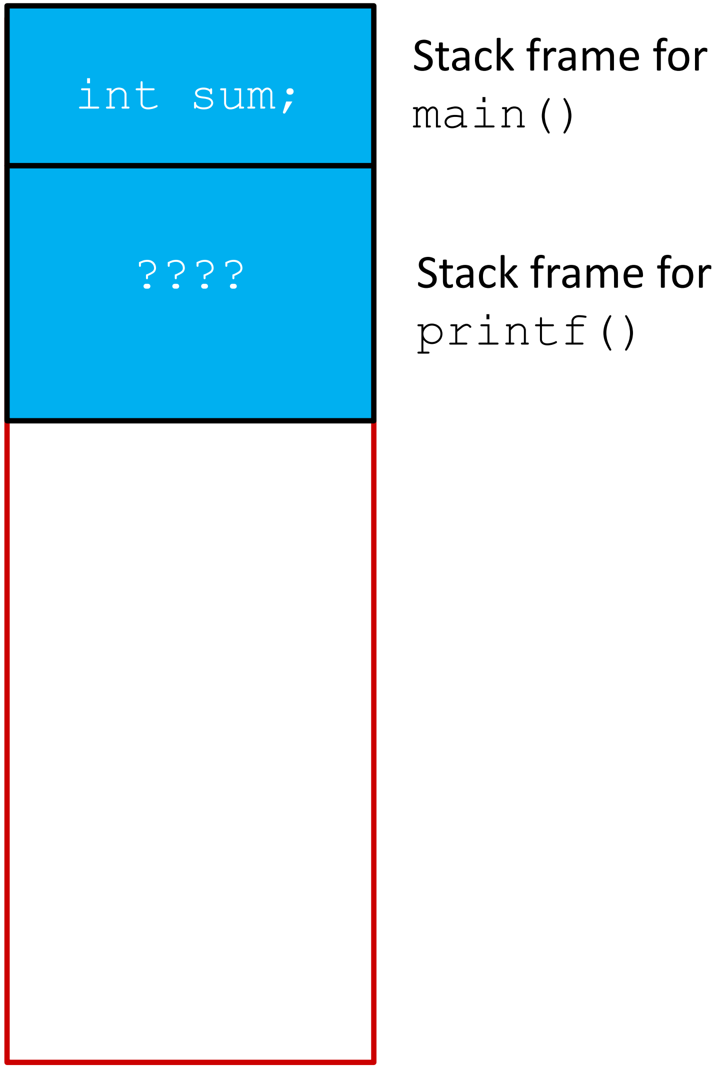
Stack frame for `main()`

`main()`'s stack frame is now top of the stack and we keep executing **`main()`**

`sum()`'s stack frame goes away after **`sum()`** returns.

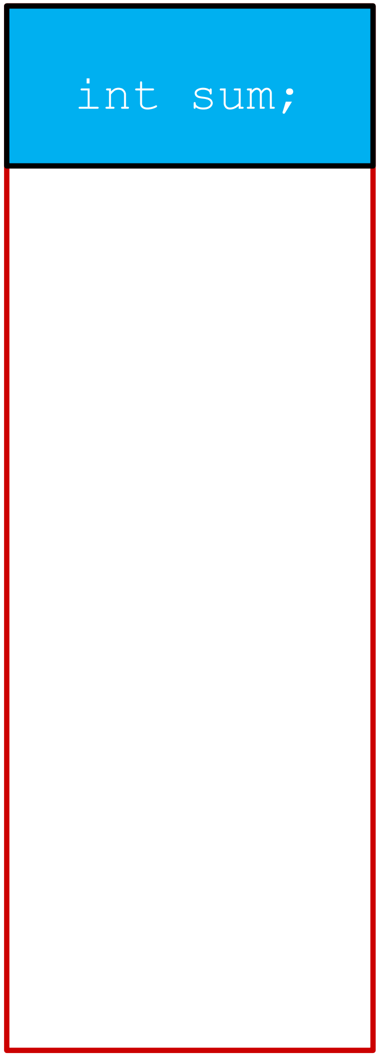
Stack Example 1:

```
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack Example 2:

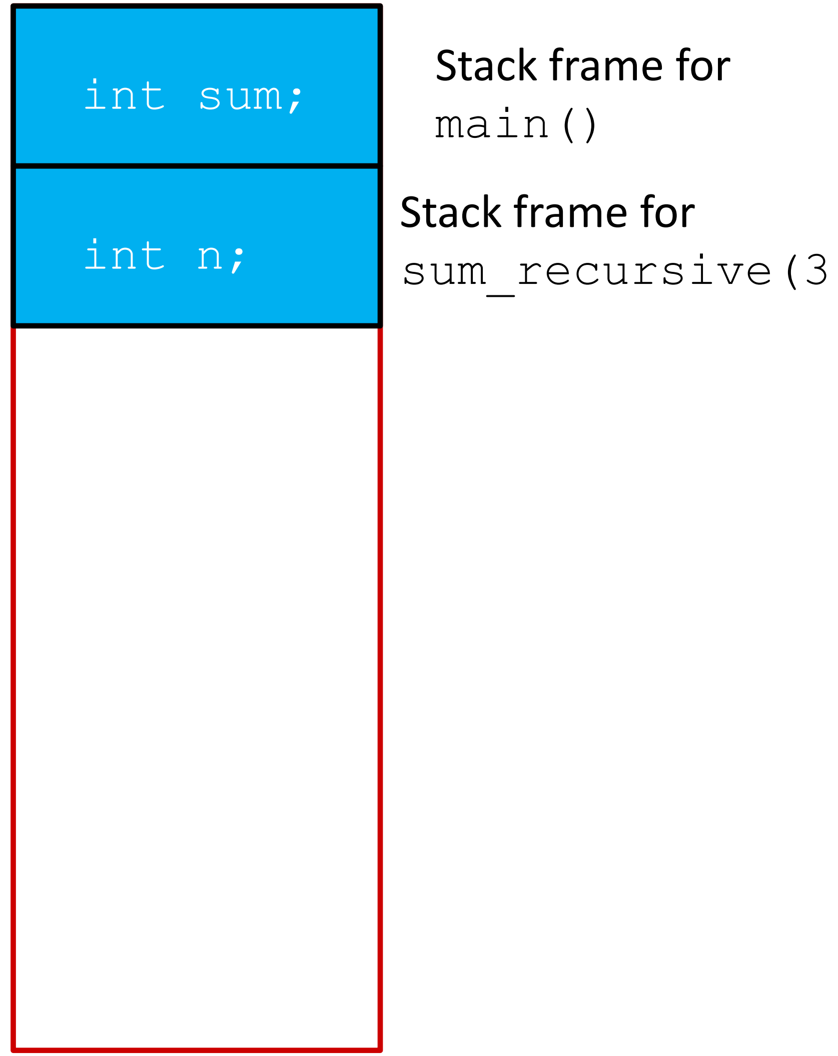
```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

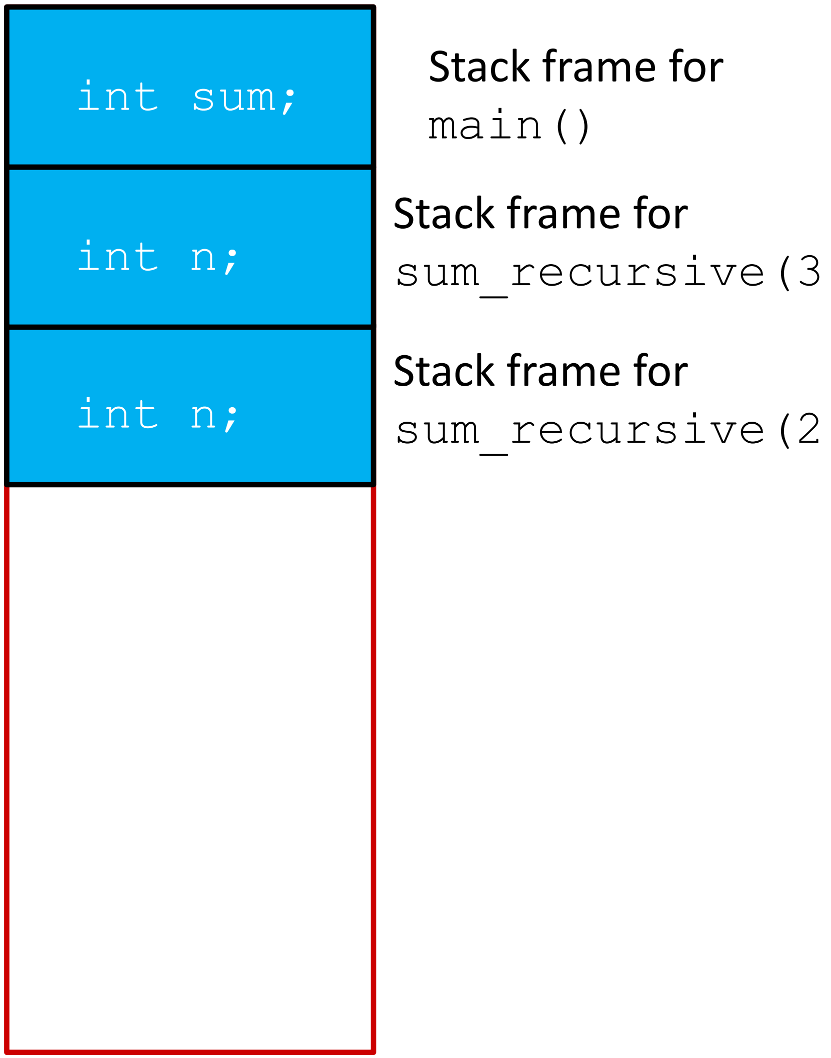
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



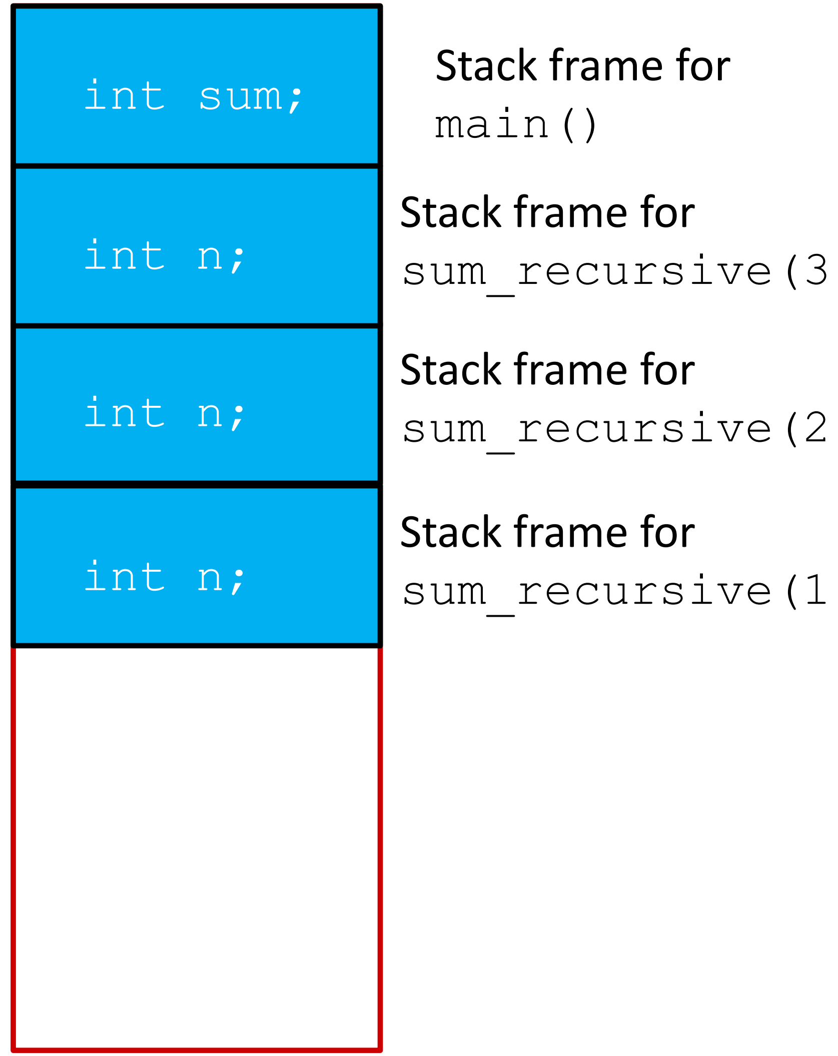
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



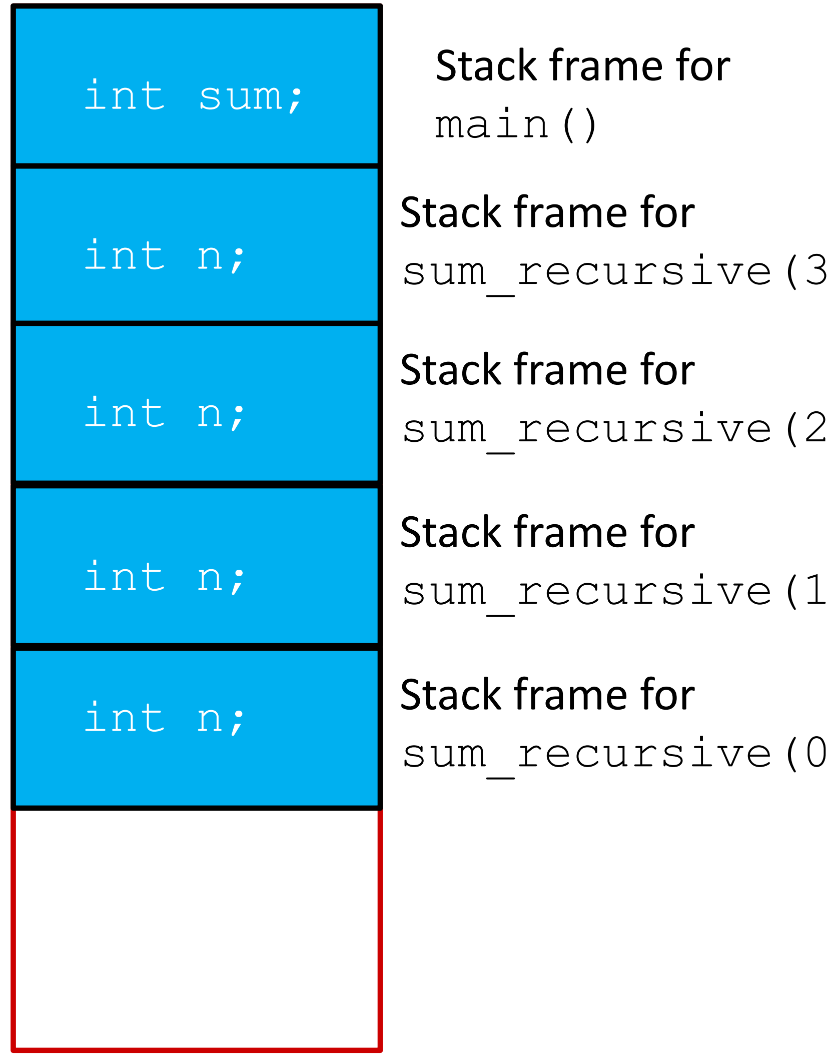
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



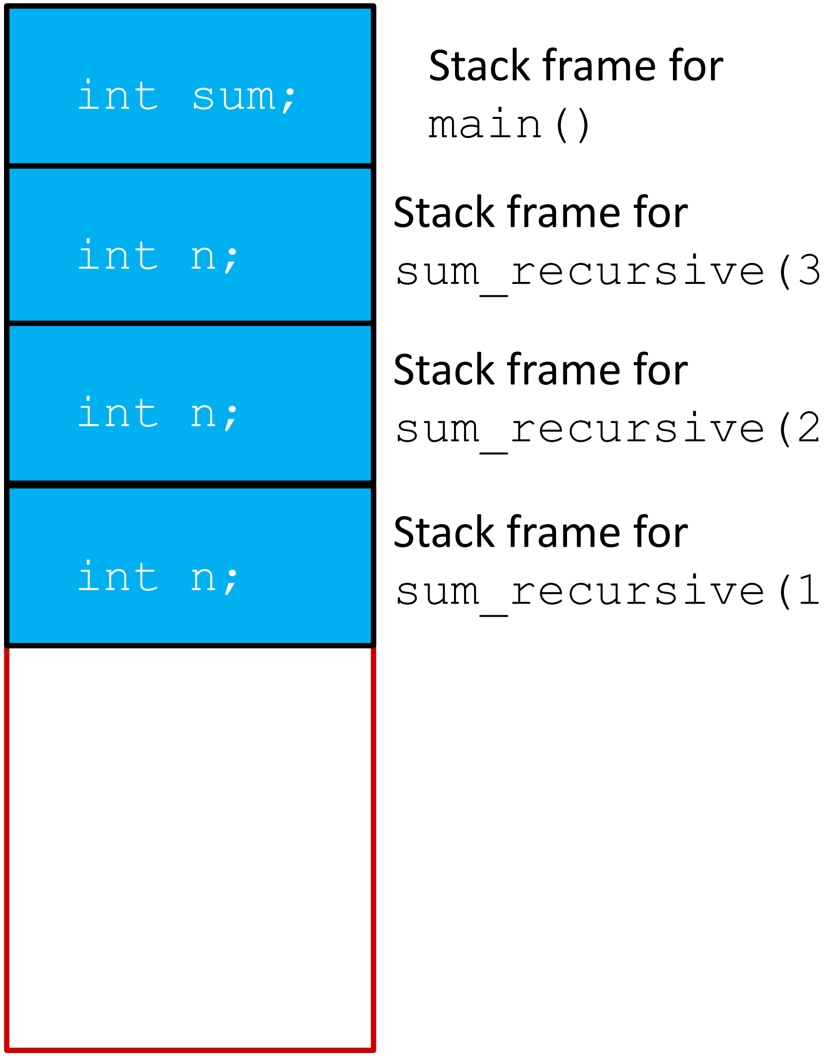
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



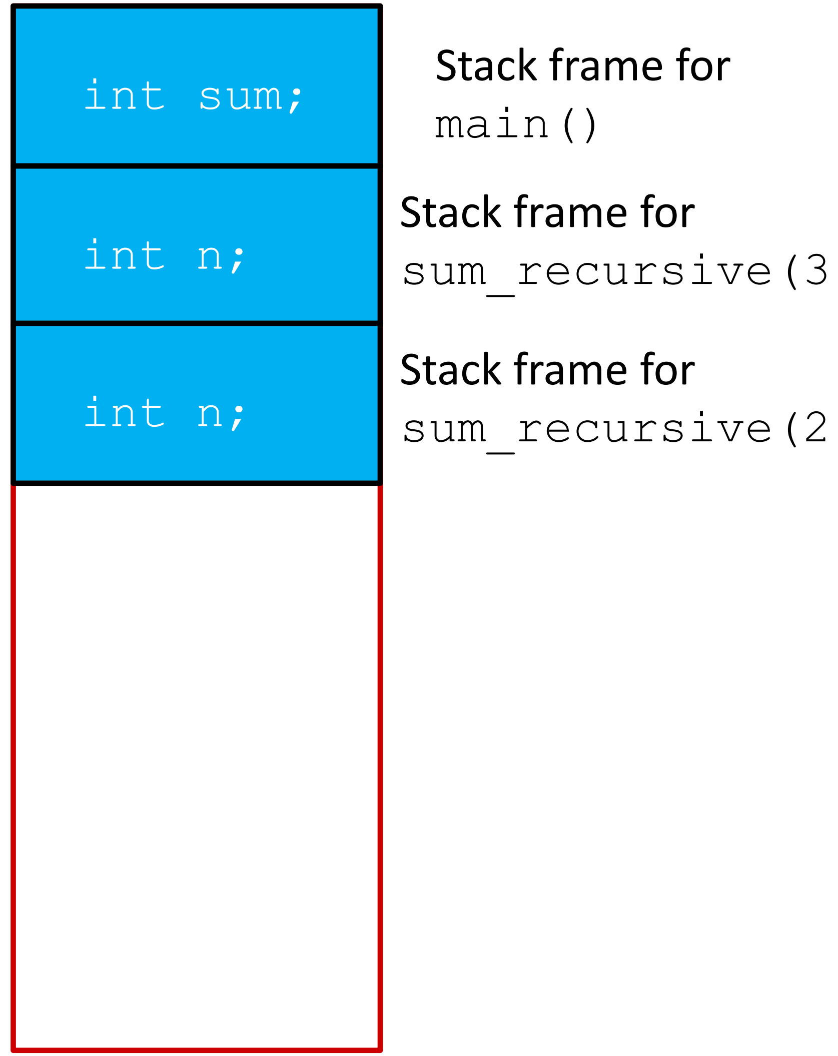
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



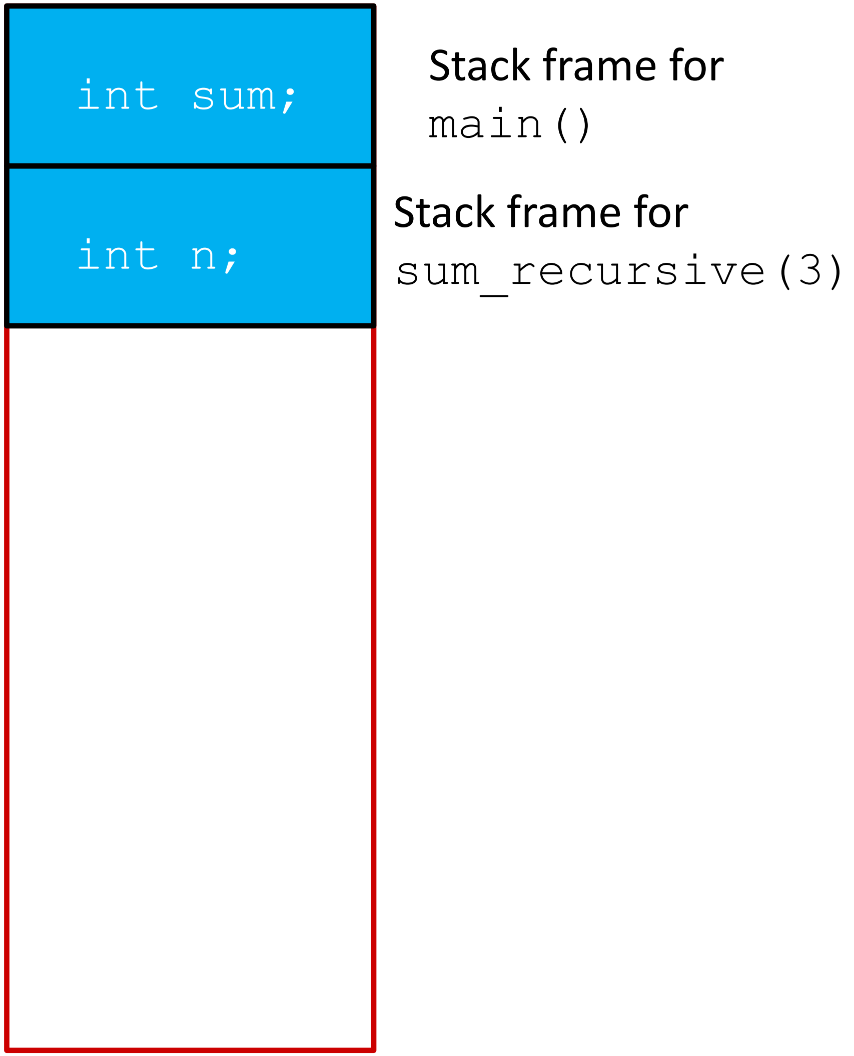
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



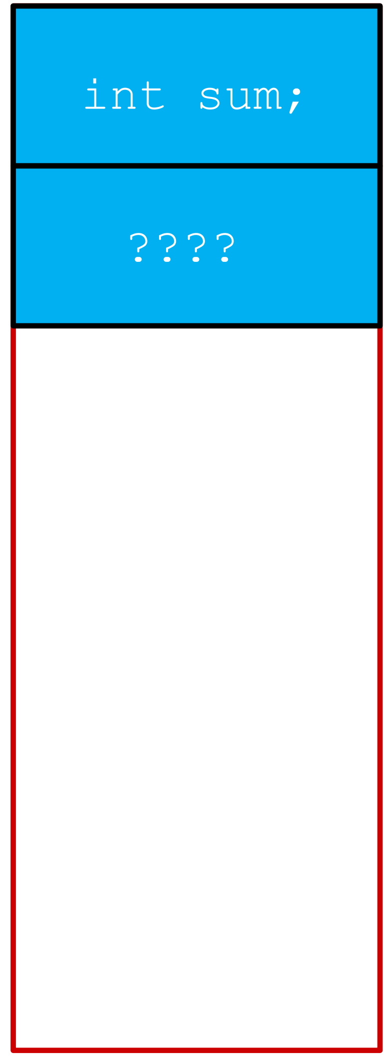
Stack Example 2:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack Example 2:

```
#include <stdio.h>
#include <stdlib.h>

int sum_recursive(int n) {
  if (n == 0) {
    return n;
  }
  return n + sum_recursive(n-1);
}

int main() {
  int sum = sum_recursive(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `printf()`

Memory Allocation So Far

- So far, we have seen two kinds of memory allocation:

```
int counter = 0;      // global var

int main() {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main() {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return 0;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

- `a`, `x`, `y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns

Poll Everywhere

pollev.com/cis2400

- The following program compiles without errors. Does it work as seemingly intended though?

A. Yes

B. No

C. I'm not sure

D. Skibidi

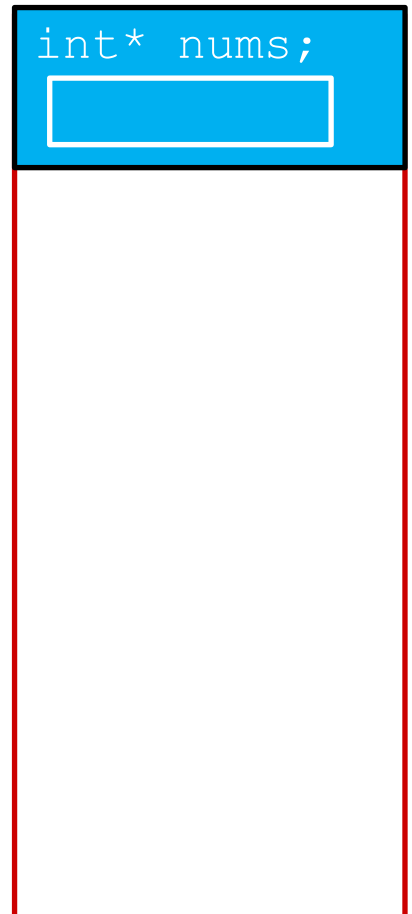
```
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/cis2400

- The following program compiles without errors. Does it work as seemingly intended though?

```c
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



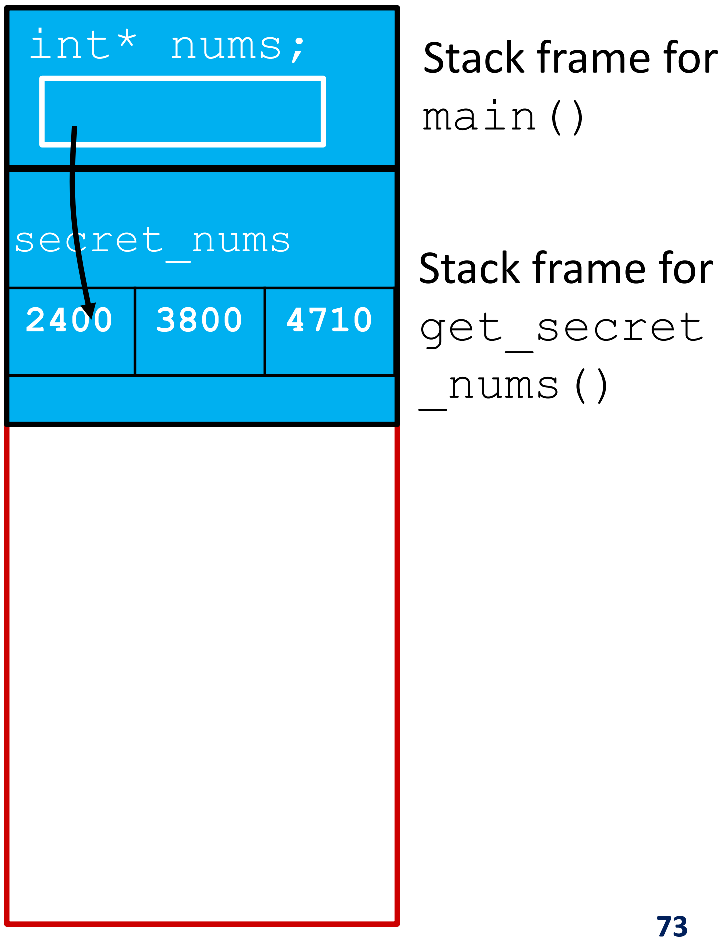
Stack frame for `main()`

Poll Everywhere

pollev.com/cis2400

- The following program compiles without errors. Does it work as seemingly intended though?

```c
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```

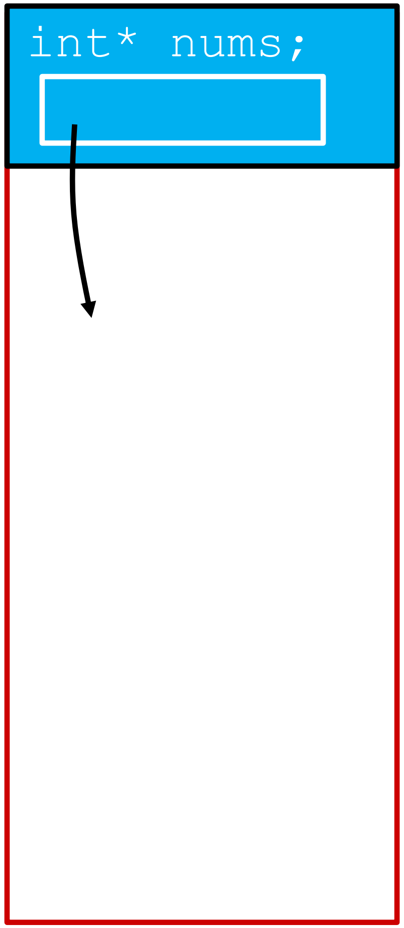


Poll Everywhere

pollev.com/cis2400

- The following program compiles without errors. Does it work as seemingly intended though?

```c
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

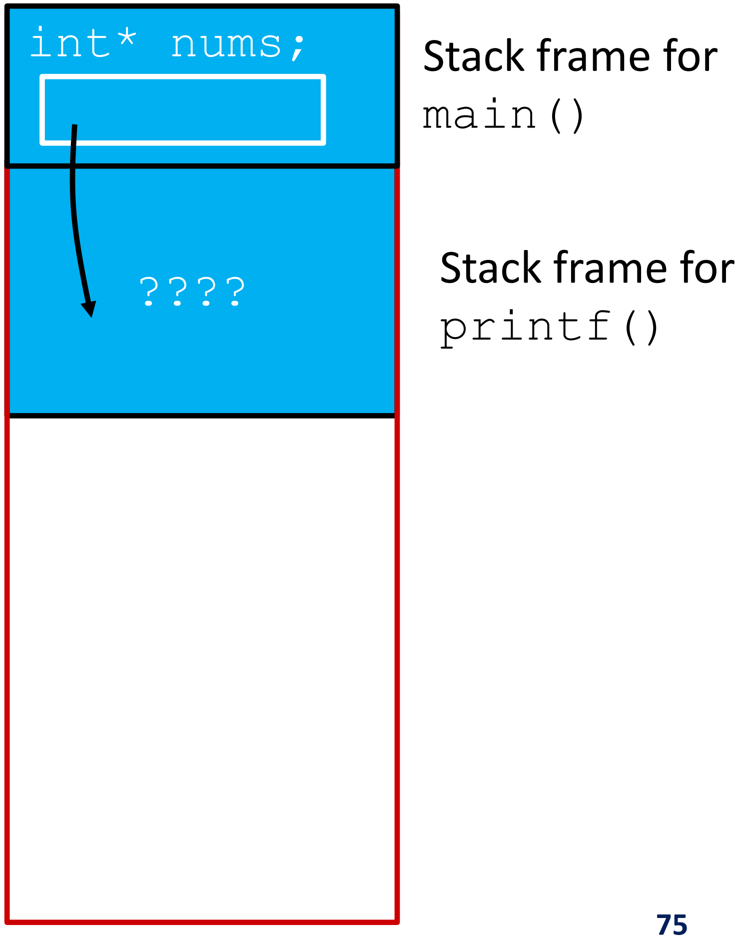
Stack frame for `get_secret_nums()` is gone…

Poll Everywhere

pollev.com/cis2400

- The following program compiles without errors. Does it work as seemingly intended though?

```
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



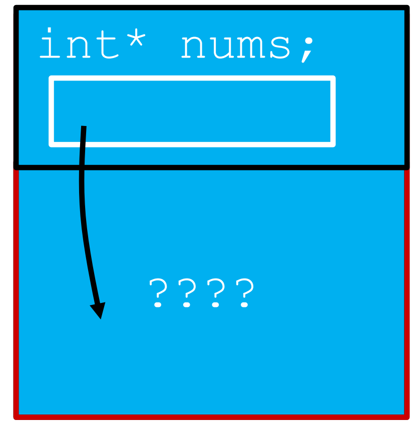
B. No

Poll Everywhere

pollev.com/cis2400

- The following program compiles without errors. Does it work as seemingly intended though?

```c
#include <stdio.h>
#include <stdlib.h>

int* get_secret_nums() {
  int secret_nums[] = {2400, 3800, 4710};
  return secret_nums;
}

int main() {
  int* nums = get_secret_nums();
  printf("%d\n", nums[0]);
  return EXIT_SUCCESS;
}
```



Stack frame for `main()`

Stack frame for `printf()`

Why?

When printf() is called we overwrite the local vars created by the get_secret_nums function call.

B. No

Lecture Take-aways

- char * is an 8-byte pointer
  - it stores an address of a character
- char[] is an array of characters
  - it stores the actual characters of a string
- A pointer is a variable that holds the address of another variable
- Memory is Split into 4 Spaces
  - Stack, Heap, Data Segment, Text/Code Segment
- Global variables can be stored at a static (un-changing) address. (Data Segment)
- Local variables are stored in a portion of memory called the "Stack"

Have a great weekend!