Short-term Hands-on Supplementary Course on C Programming

SESSION 10: More Pointers

NIVEDHITHA D
KARTHIK D

Time: 6:30 - 8:00 PM
Date: June 29th, 2022
Location: Online

Agenda

1. Administrative Instructions
2. Pointers Recap
3. Pointer Arithmetic
4. const keyword and Pointers
5. Command Line Arguments
6. 2D Arrays and Pointers
7. Pointers and Functions
8. Tutorial: Functions with Pointers
9. Next Session

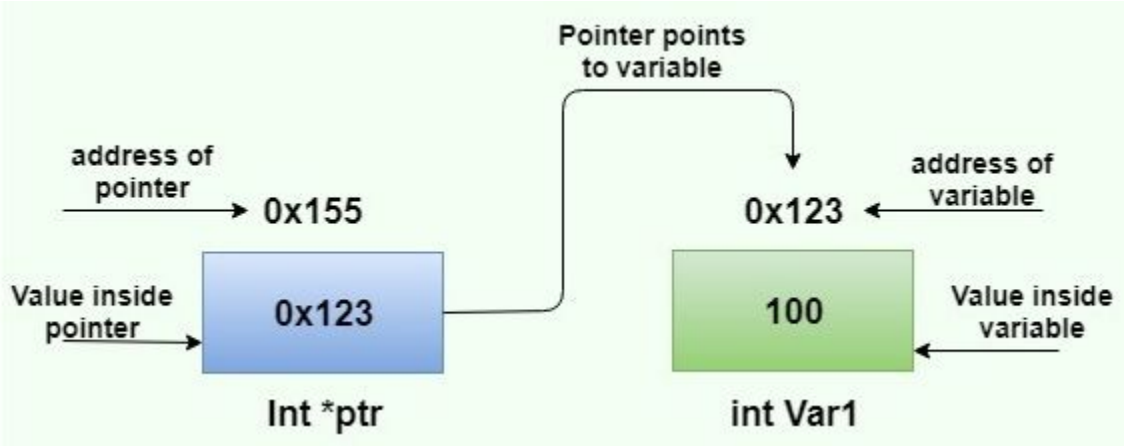
Administrative Instructions

- Please fill out the feedback form - will be shared in the chat
- Join us on Microsoft Teams,
  Team Code: **rzlaicv**

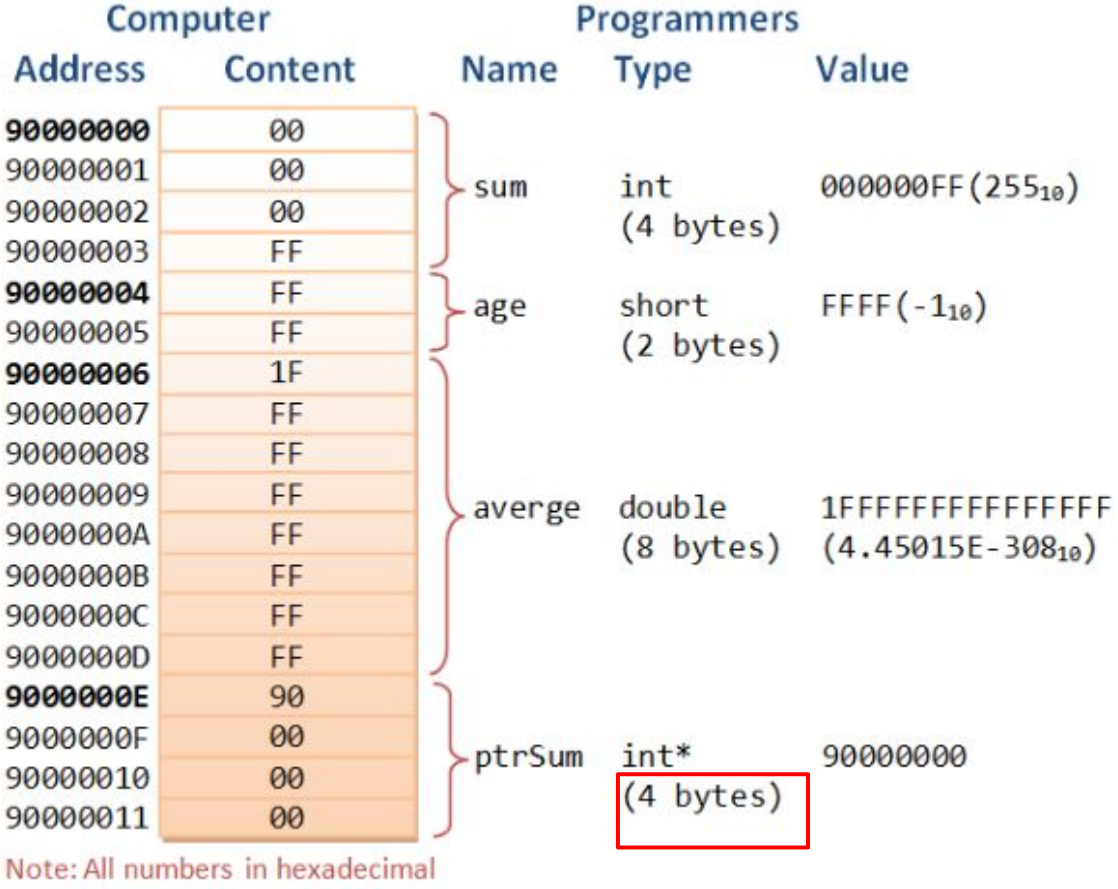
GITHUB REPOSITORY! ⭐🌟

What are Pointers?

A *pointer variable* (or *pointer* in short) is basically the same as the other variables, which can store a piece of data. Unlike normal variable which stores a value (such as an `int`, a `double`, a `char`), a *pointer stores a memory address*.



Declaring and using Pointers




```
type *ptr;
// or
type* ptr;
// or
type * ptr;
```

```c
#include <stdio.h>

int main(void) {
    int sum = 255;
    short age = -1;
    double average = 4.45015E-308;
    int* ptrSum = &sum;
}
```

Pointer Rules

1. A pointer stores a reference to its pointee. The pointee, in turn, stores something useful.
2. The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this one rule.
3. Allocating a pointer does not automatically assign it to refer to a pointee. Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.
4. Assignment between two pointers makes them refer to the same pointee which introduces sharing.

Pointer Arithmetic

When you do pointer arithmetic, you are adjusting the pointer by a certain *number of places* (e.g. characters).

```
char *str = "apple";    // e.g. 0xff0
char *str1 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

printf("%s", str);      // apple
printf("%s", str1);     // pple
printf("%s", str3);     // le
```

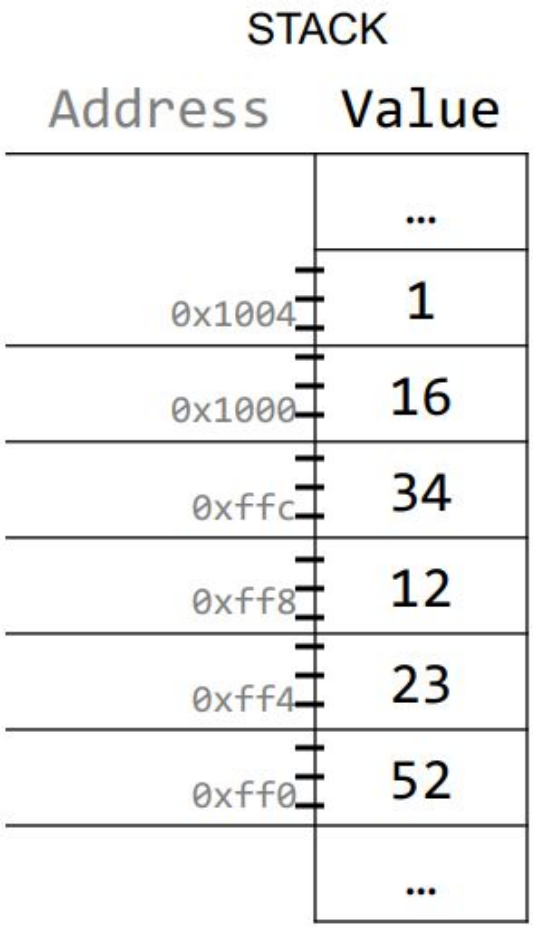
DATA SEGMENT

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

Pointer Arithmetic

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

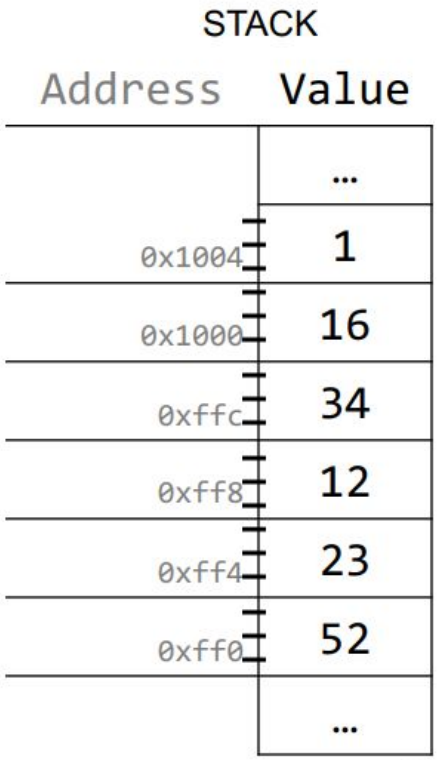
```
// nums points to an int array
int *nums = …              // e.g. 0xff0
int *nums1 = nums + 1;  // e.g. 0xff4
int *nums3 = nums + 3;  // e.g. 0xffc

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



Pointer Arithmetic

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 *nums3 = nums + 3;  // e.g. 0xffc
int *nums2 = nums3 - 1; // e.g. 0xff8

printf("%d", *nums);    // 52
printf("%d", *nums2);   // 12
printf("%d", *nums3);   // 34
```



Pointer Arithmetic

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

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

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

```
ptr + i      <=>    &ptr[i]
*(ptr + i)   <=>    ptr[i]
```

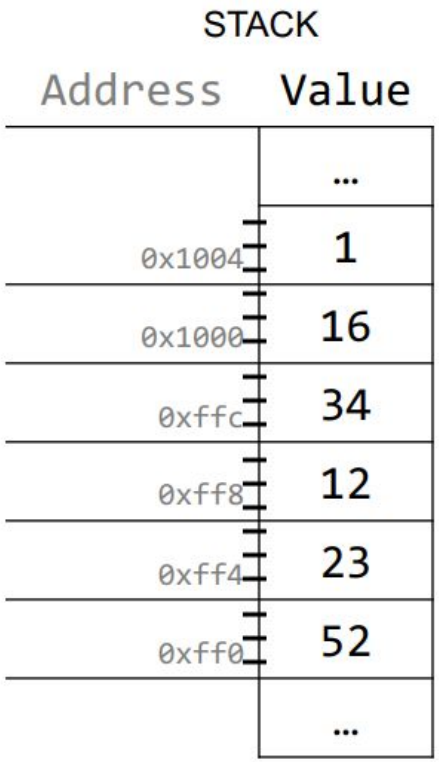
DATA SEGMENT

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

Pointer Arithmetic

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

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



Pointer Arithmetic

How does the code know how many bytes it should 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, "CS107");

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

Pointer Arithmetic

Essentially, these are the four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
1  *p++   // same as *(p++): increment pointer, and dereference
2  unincremented address
3  *++p   // same as *(++p): increment pointer, and dereference
4  incremented address
   ++*p   // same as ++(*p): dereference pointer, and increment the value it
   points to
   (*p)++ // dereference pointer, and post-increment the value it points to
```

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

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 and pointers

- *const* pointer

```
<type of pointer> *const <name of pointer>;
```

- pointer to a *const*

```
const <type of pointer>* <name of pointer>
```

- const pointer to a *const*

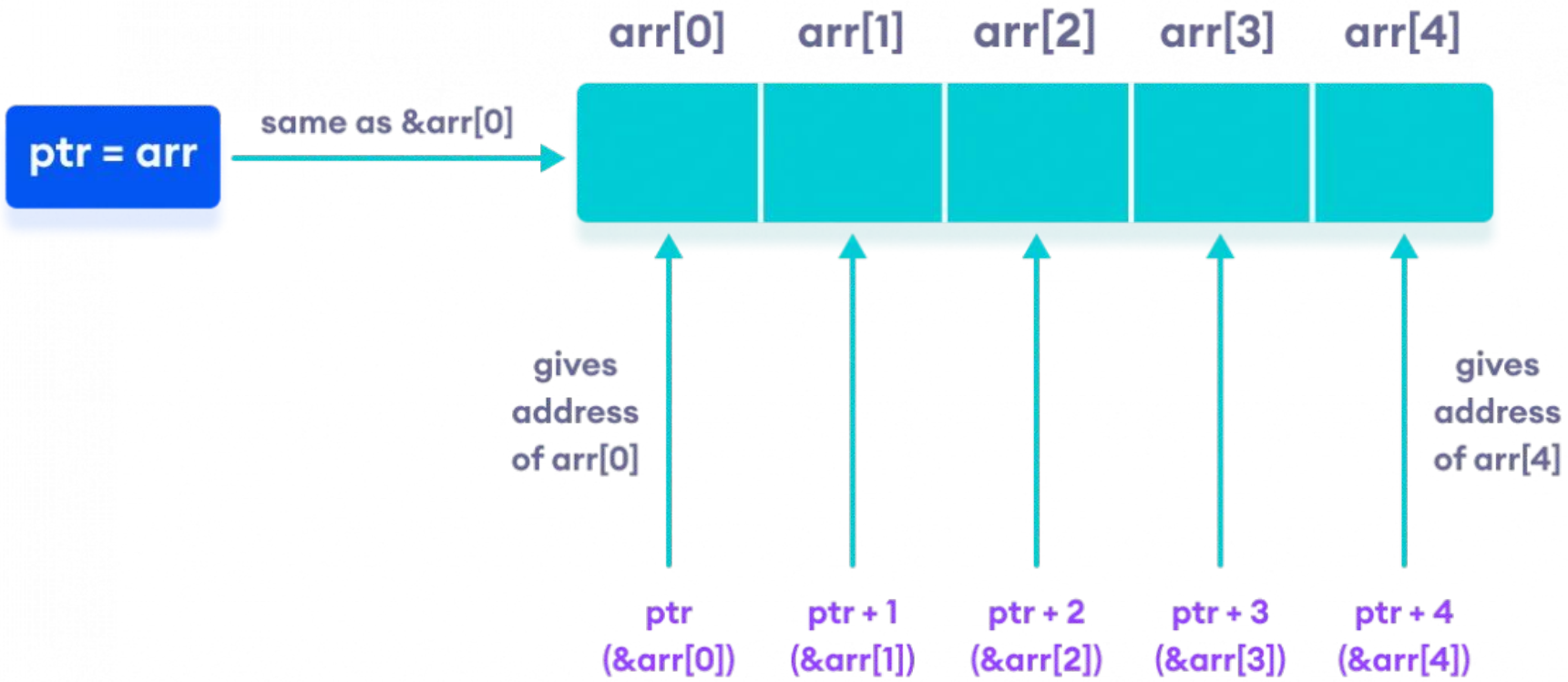
```
const <type of pointer>* const <name of the pointer>;
```

KARTHIK!!

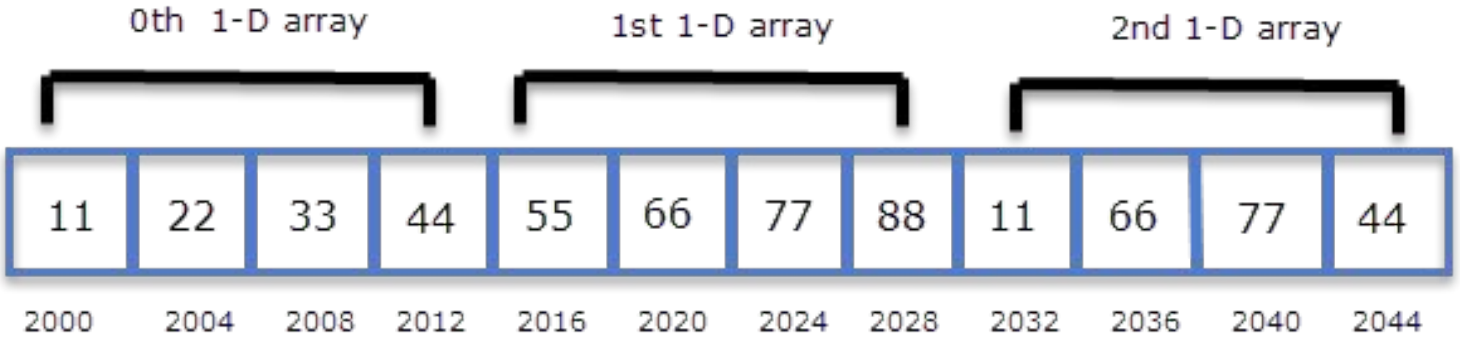
Remember Precedence when Using Pointer Operators

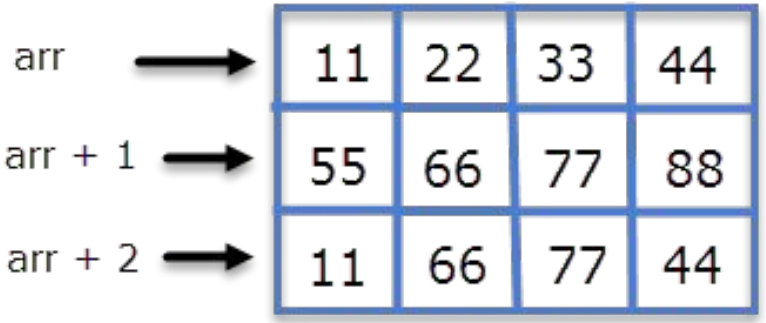
Precedence	Operator	Description	Associativity
1	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	`(`*type*`){`*list*`}`	Compound literal(C99)	
2	`++ --`	Prefix increment and decrement[note 1]	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of[note 2]	
	`_Alignof`	Alignment requirement(C11)	

Pointers & Arrays



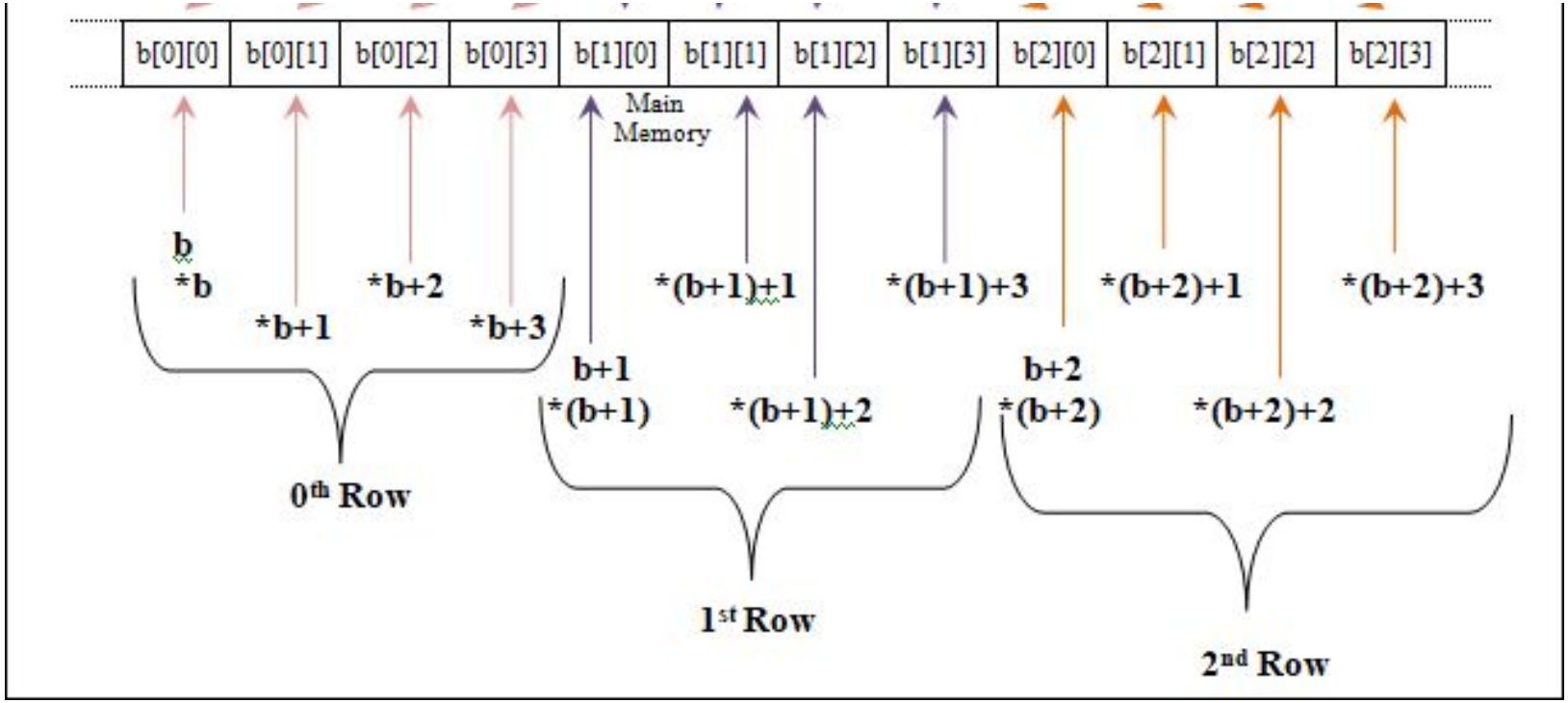
2D Arrays in Memory





arr points to 0th 1-D array
(arr + 1) points to 1st 1-D array
(arr + 2) points to 2nd 1-D array

In general,
(arr + i) points to ith 1-D array



One step further,

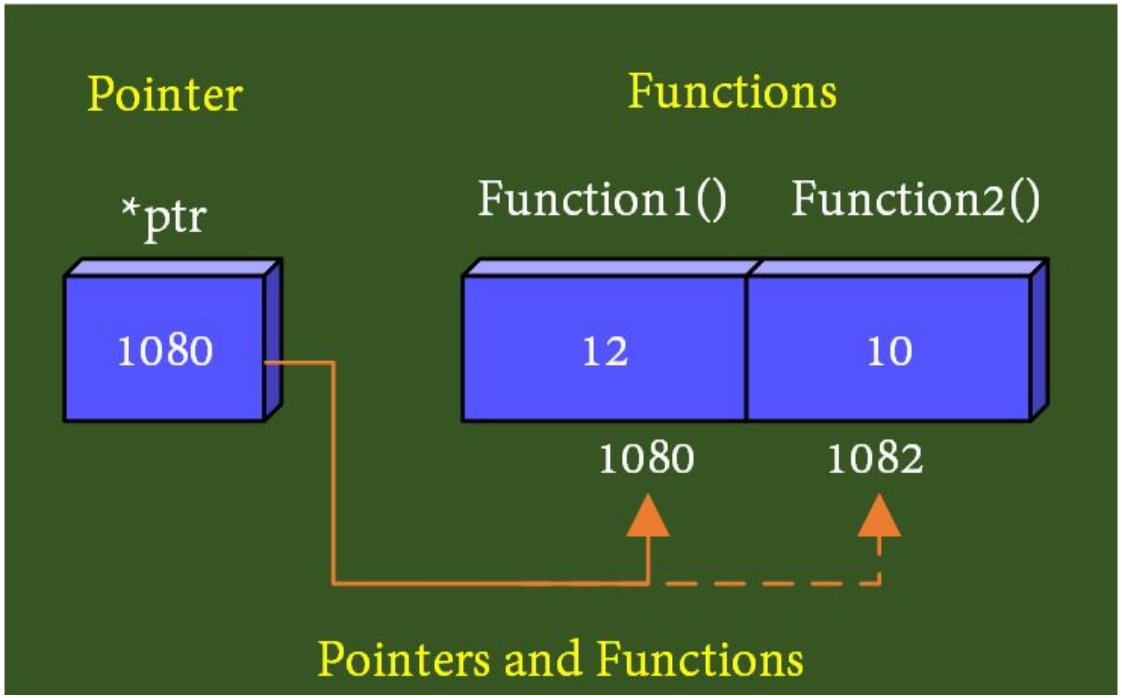
(b + i) points to 0th element of the 1-D array
(b + i) + 1 points to 1st element of the 1-D array
(b + i) + 2 points to 2nd element of the 1-D array

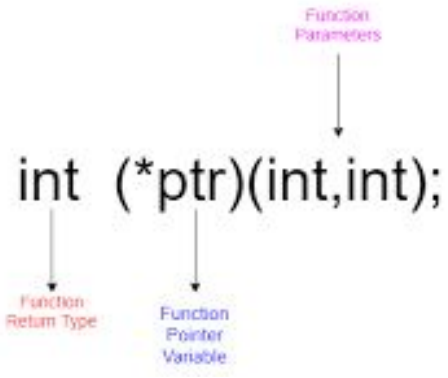
In general,
(b + i) + j points to **jth** element of **ith** 1-D array
→ *Similar to* **b[i][j]**

Function Pointers in C

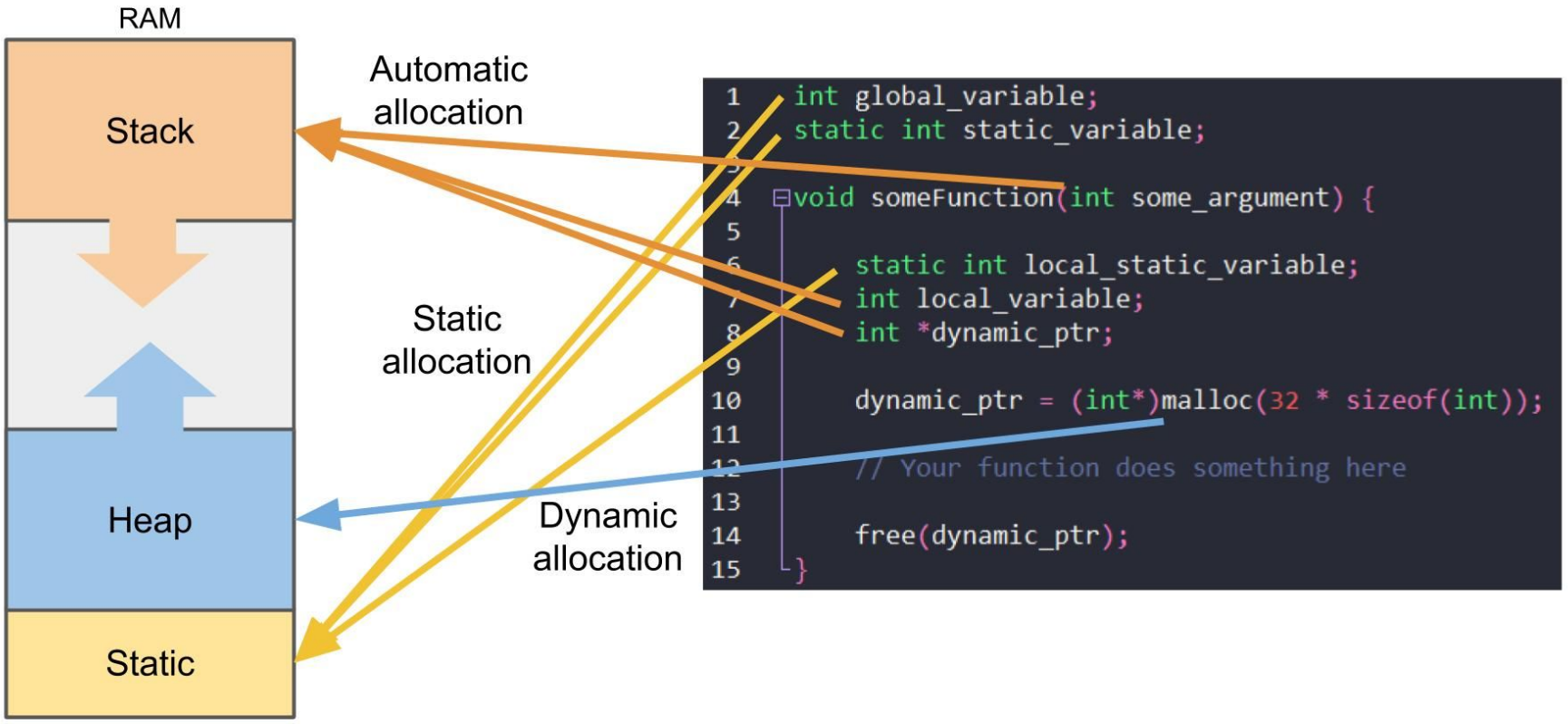
- A function pointer is a **variable that stores the address of a function** that can later be called through that function pointer

- But why?!
  - Callback Functions
  - Functions as Arguments





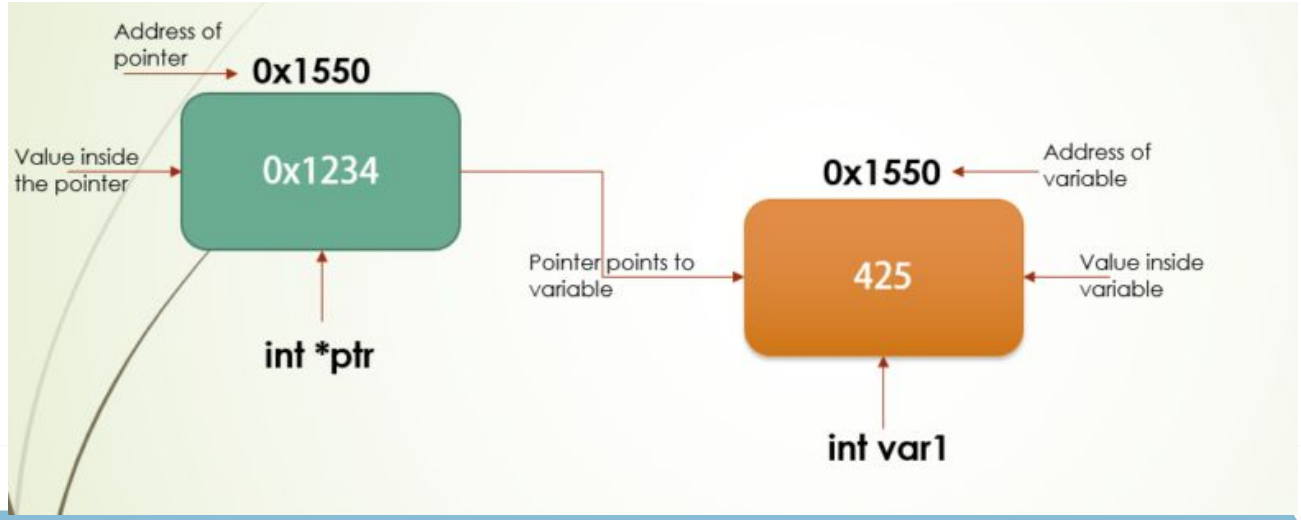
Revisit: Memory Allocation in C



Revisit: Static vs. Dynamic Memory Allocation

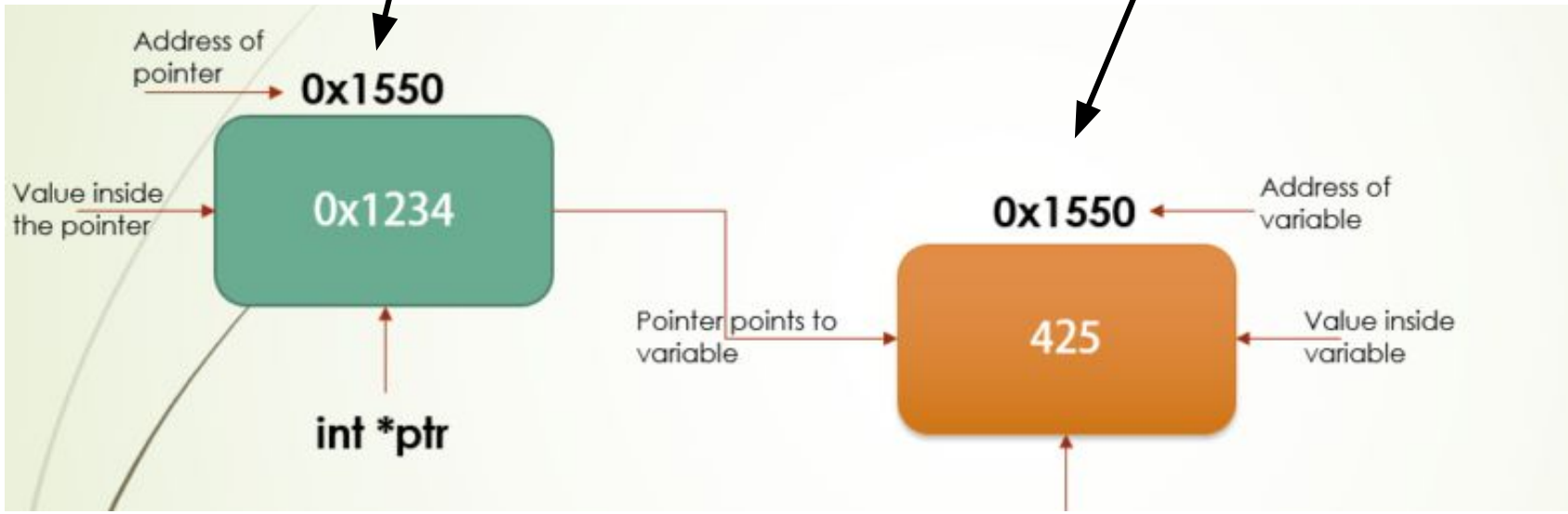
Dynamic Memory	Static Memory
Allocated at run time	Allocated at compile time
Memory can be altered during program execution	Memory cannot be altered during program execution
Example: Linked list	Example: Array

- The **heap** is often called **unnamed variable space**

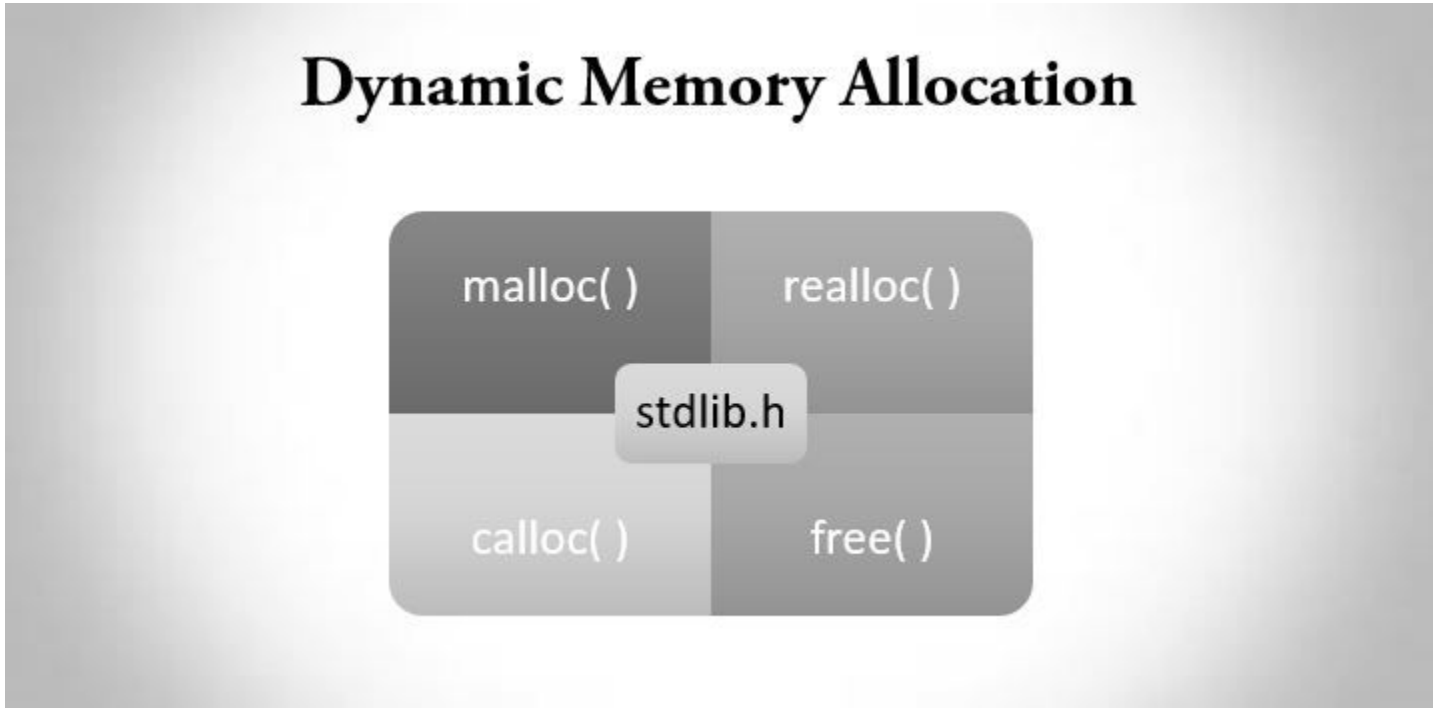


This is in
Stack/Global

This is in **Heap**
(if dynamically allocated)



Dynamic Memory Allocation in C



Syntax:

- void *malloc(size_t size);
- void *calloc(size_t num, size_t size);
- void *realloc(void *ptr, size_t new_size);
- void free(void* ptr);

TUTORIAL

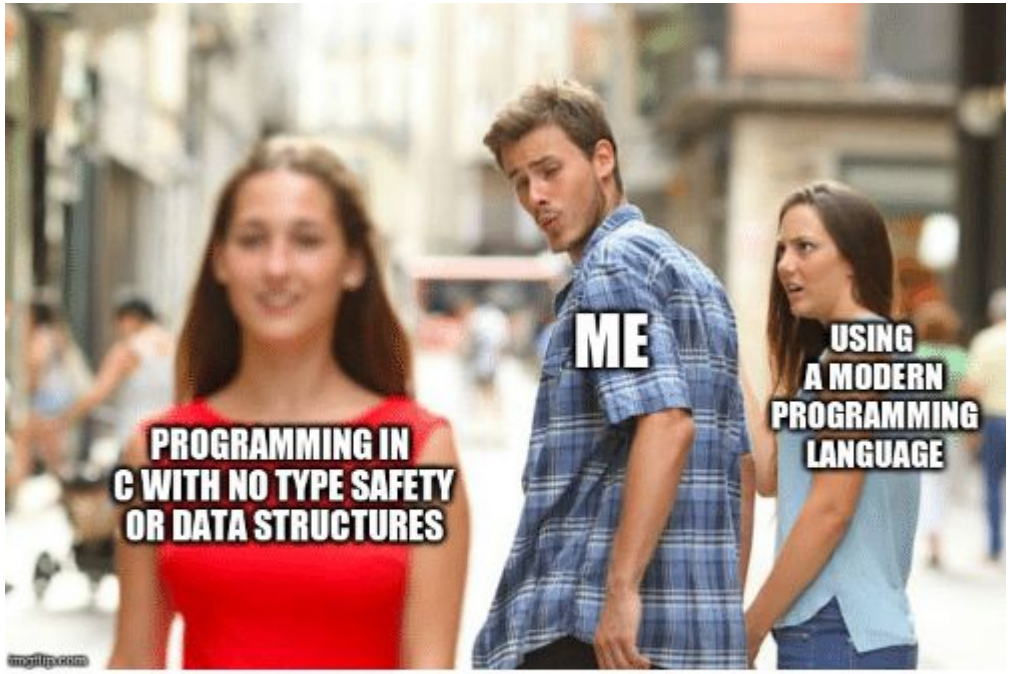
Pointers to Functions

```
int* map_with_one_int_arg(int (*function)(int), int *arr, int size){

    int *ret_arr = (int*)malloc(sizeof(int)*size);
    // int ret_arr[size];

    for(int i=0; i<size; i++){
        *(ret_arr+i) = function(*(arr+i));
    }

    return ret_arr;
}
```

2D Arrays using Pointers, and Dynamic Memory Allocation

Next Session

STRUCTURES!



I really enjoy C for some reason...

Any Questions