

# Short-term Hands-on Supplementary Course on C Programming



## SESSION 14: Structures and Files

KARTHIK D  
NIVEDHITHA D

Mode: **Asynchronous**  
Location: Online



# Administrative Instructions

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

**GITHUB REPOSITORY!\*\***  

# Nearly the End !!!



**CAPSTONE PROJECT !!!!!**



Friends: How did you write this code so beautifully ?  
Me(Proudly):

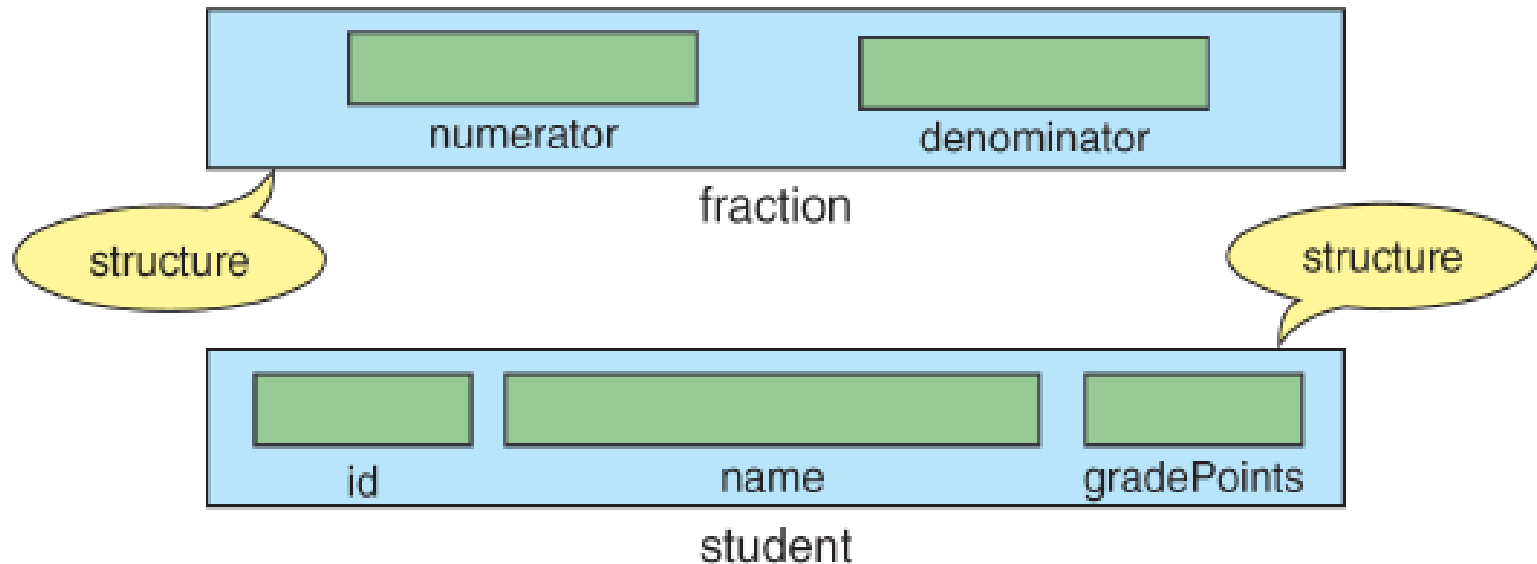


# Agenda

1. Pointers to Structures
2. Dynamic Initialization of Structures
3. Self-referential Structures
4. Linked Lists using Structures
5. Structures and Files
6. Tutorial: Store a Linked List in a File
7. Next Session: The Capstone Project

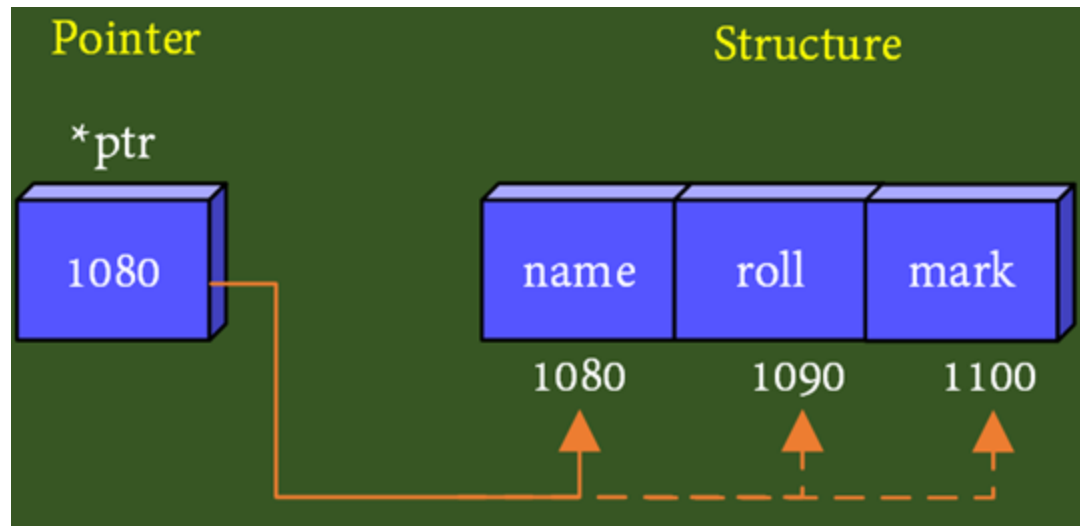
# Recap: Structures in C

A **structure** in C is a user-defined data type. It is used to bind the two or more similar or different data types or data structures together into a single type. Structure is created using struct keyword and a structure variable is created using struct keyword and the structure tag name.



# Pointers to Structures [[demo](#)]

- Declaration of structure pointer
- Initialization of struct pointer
- Accessing members through pointer
  - (\*) asterisk and (.) operators
  - **(->) arrow operator** [recommended]



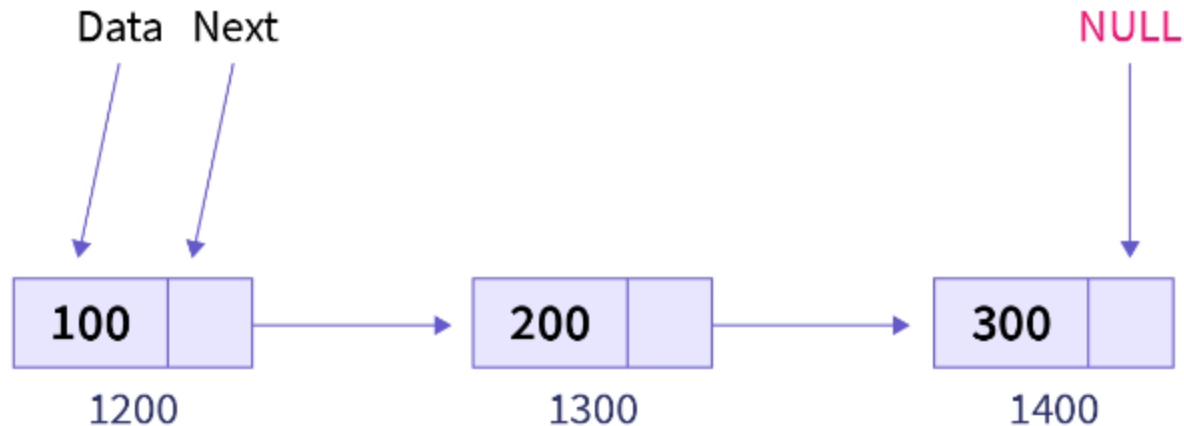
# Dynamic Init of Structures

Static Memory	Dynamic Memory		
Stack	Heap	<b>malloc()</b> method	<b>calloc()</b> method
Global Variable		It is generally used to allocate a <b>single</b> memory block of the given size (in bytes) during the run-time of a program.	It is generally used to allocate contiguous ( <b>multiple</b> ) blocks of memory of given size (in bytes) during the run-time of a program.
Instructions (Outside main() function)		It <b>doesn't</b> initialize the allocated memory block and contains some garbage value.	It <b>initializes</b> all the allocated memory blocks with <b>0</b> (zero) value.
		<b>malloc()</b> takes a <b>single argument</b> i.e. the size of memory block that is to be allocated.	<b>calloc()</b> takes <b>two arguments</b> i.e. the number of elements and the size of one element that are to be allocated.
		syntax: <b>(cast-data-type *)malloc(size-in-bytes)</b>	syntax: <b>(cast-data-type *)calloc(num, size-in-bytes)</b>
		Example : <code>float *ptr = (float *)malloc(sizeof(float));</code>	Example : <code>float *ptr = (float *)calloc(10, sizeof(float));</code>

# Self-referential Structures

```
struct Node
{
    int data;
    struct Node* next;
};
```

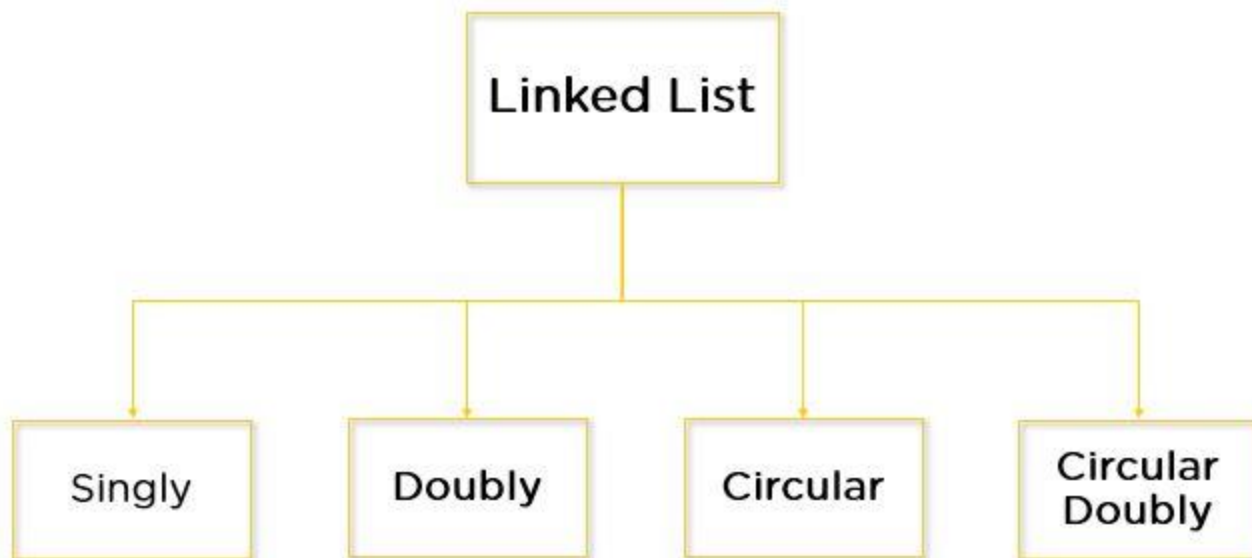
```
struct Node* node1 = (struct Node*)malloc(sizeof(struct Node));
```





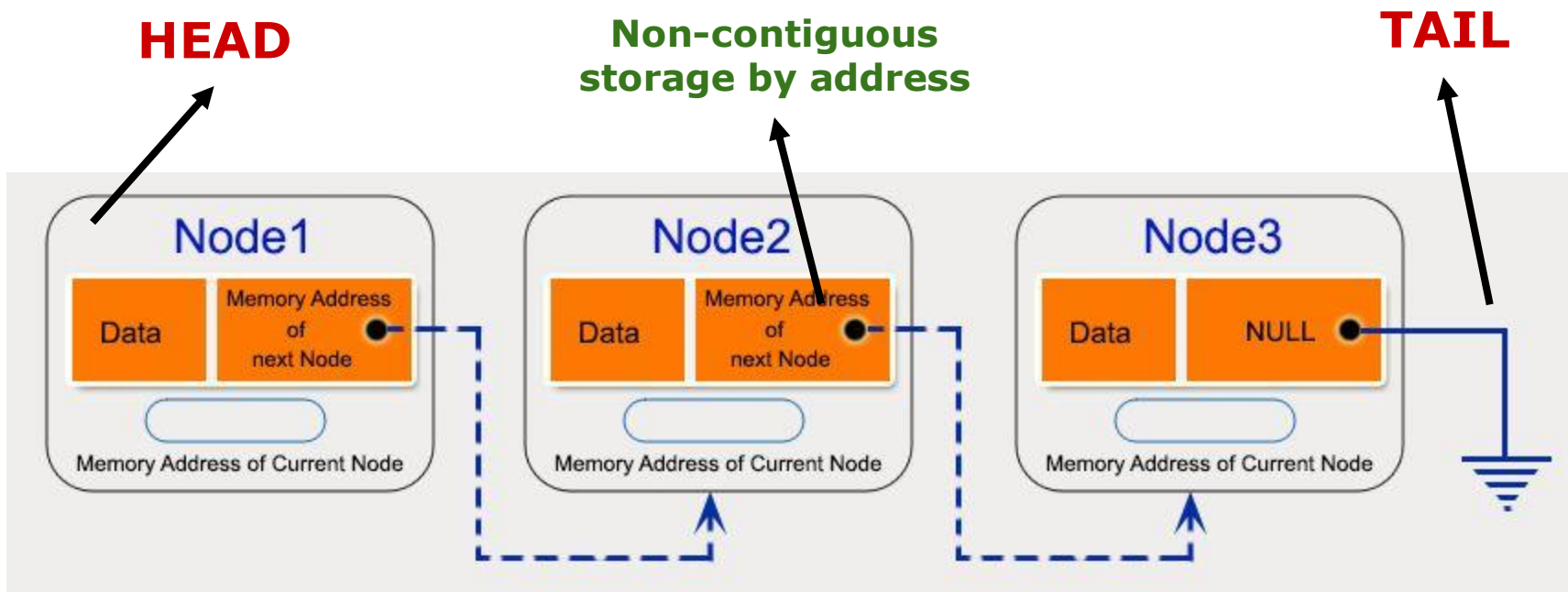
# Linked Lists

- A **linear data structure**.
- Unlike arrays, elements are **NOT stored contiguously**.
- Elements are usually regarded as **nodes**.
- Nodes can store **complex, custom data formats** — defined by the underlying self-referential structure.
- Linked lists can be classified based on **how their nodes are linked** together.



# Bonus: Singly Linked Lists

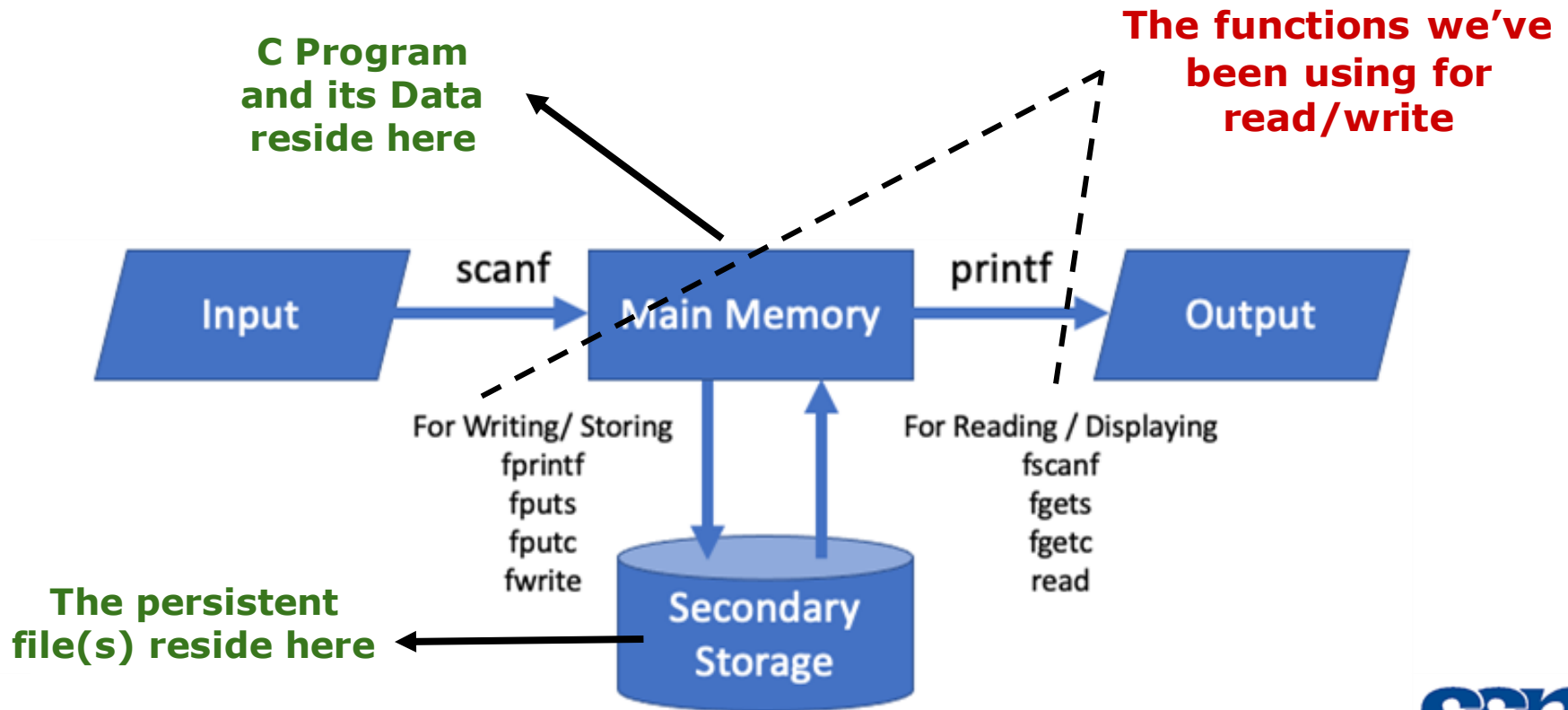
- **Unidirectional linked list** — each node only points to its next node.
- Can be **traversed in one direction only**, i.e., from head node to tail node.



- We'll now look at an implementation of singly linked lists, using self-referential structures

# Recap: File Handling

- Files are used to **store data persistently** between executions of a program
- Basic steps in file manipulation — ***open a stream, read/write, close the stream***
- File access modes, position seeking, ...



# Structures and Files

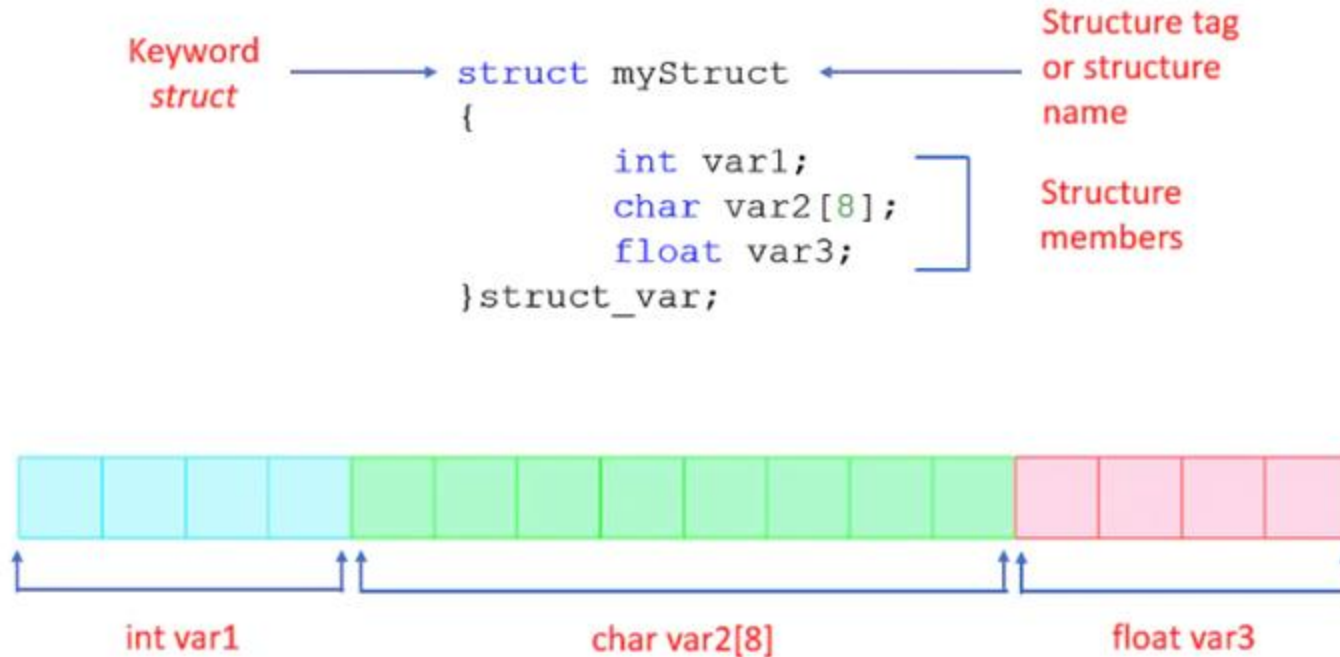
- Remember that there are 2 file data formats — **text and binary**
- Binary files are ideal to store structures in files
- A few implementation details:
  - Can't store pointer attributes directly — this requires additional measures, such as **data serialization**
  - Use the binary data read/write functions —

**size\_t** **fwrite**( **const void** \*buffer, **size\_t** size, **size\_t** count, **FILE** \*stream );

**size\_t** **fread**( **const void** \*buffer, **size\_t** size, **size\_t** count, **FILE** \*stream );

# Structures and Files

- Remember how structures are represented in memory?



- The writing program, **encodes memory segment** into **bit-sequence**
- When written to a file, this **entire memory segment is stored as bits** in the binary file
- The reading program, **decodes bit-sequence** into structure's **memory segment** — the *structure definition must be known!*

# TUTORIAL

## IMPLEMENTING SINGLY-LINKED LISTS

## STORING and ACCESSING LINKED LISTS from FILES

- We **serialize** the list into nodes, when **storing** to the file
- We **reconstruct** the list, when **reading** from the file
- Of course, there are ***other serialization strategies*** that you could come up with!

# Next Session



**CAPSTONE PROJECT !!!!!**



Friends: How did you write this code so beautifully ?  
Me(Proudly):

