Day_9 C-Programming

(RECAP OF PREVIOUS DAY)
Structure, Structure within structure, Array of structure, Union, Difference between Structure and Union, Dynamic Memory Allocation: malloc, calloc, realloc and free.

Structure

A Structure is a collection of elements, which are of different data types.
It is a user-defined data type that allows grouping variables of different data types under a single name.

Syntax:

```
struct StructureName {
    dataType member1;
    dataType member2;
    ...
};
```

Example:

```
#include <stdio.h>

struct Student {
    int id;
    char name[50];
    float marks;
};

int main() {
    struct Student s1 = {1, "Alice", 95.5};

    printf("ID: %d\n", s1.id);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
```

```
}
```

Accessing Members:

- Use the dot operator (`.`) for direct access and use arrow operator(`->`) to access members through pointer
- Example: `s1.id` or `s1->id` **(We will see this after malloc function today)**

Differences Between Structures and Arrays:

Aspect	Structure	Array
Data Type	Can store multiple variables of **different data types**.	Stores multiple variables of the **same data type**.
Definition Syntax	Defined using the **struct** keyword.	Declared with the data type followed by the size. **int a[10]**
Memory Allocation	Memory is allocated for each member individually.	Memory is allocated as a contiguous block for all elements.
Accessing Elements	Accessed using the dot (.) operator and the member name.	Accessed using an index (e.g., array[index]).

Structure Within Structure

A structure can have another structure as a member.

Example:

```c
#include <stdio.h>

struct Address {
    char city[50];
    int pin;
};

struct Employee {
    int id;
    char name[50];
    struct Address addr; // Nested structure
};

int main() {
    struct Employee e1 = {1, "John", {"New York", 12345}};

    printf("ID: %d\n", e1.id);
    printf("Name: %s\n", e1.name);
    printf("City: %s\n", e1.addr.city);
    printf("PIN: %d\n", e1.addr.pin);

    return 0;
}
```

Array of Structures

An array of structures allows storing multiple records of the same structure type.

Example:

```c
#include <stdio.h>

struct Book {
    int id;
    char title[50];
    float price;
};

int main() {
    struct Book books[3] = {
```

```
        {1, "C Programming", 300.50},
        {2, "Data Structures", 450.75},
        {3, "Algorithms", 500.00}
    };

    for (int i = 0; i < 3; i++) {
        printf("Book ID: %d\n", books[i].id);
        printf("Title: %s\n", books[i].title);
        printf("Price: %.2f\n\n", books[i].price);
    }

    return 0;
}
```

Union

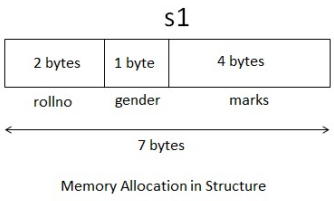
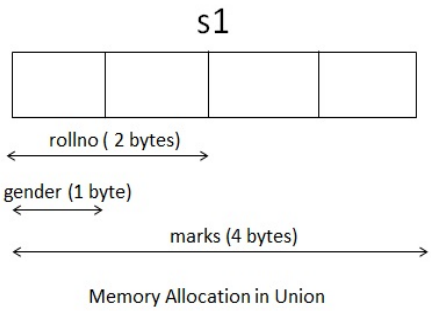
A union is similar to a structure, but it uses a single shared memory location for all its members. Only one member can store a value at a time.

Syntax:

```
union UnionName {
    dataType member1;
    dataType member2;
    ...
};
```

Difference between Structure and union

	Structure	**Union**
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.

Keyword	The keyword struct is used to define a structure	The keyword union is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
	 Memory Allocation in Structure	 Memory Allocation in Union
Memory Allocation	Each member within a structure is allocated a unique storage area of location.	Memory allocated is shared by individual members of the union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members share the same memory.
Accessing Members	Individual members can be accessed at a time.	Only one member can be accessed at a time.

Example:

```c
#include <stdio.h>

union Data {
    int i;
    float f
    char str[20];
};

int main() {
    union Data data;

    data.i = 10;
    printf("Integer: %d\n", data.i);

    data.f = 220.5;
    printf("Float: %.2f\n", data.f);

    strcpy(data.str, "C Programming");
    printf("String: %s\n", data.str);

    return 0;
}
```

Dynamic Memory Allocation

Static memory allocation	Dynamic memory allocation
It is the process of allocating memory at compile time.	It is the process of allocating memory during execution of program.
Fixed number of bytes will be allocated.	Memory is allocated as and when it is needed.
The memory is allocated in memory stack.	The memory is allocated from free memory pool (heap).

Dynamic memory allocation allows memory to be allocated at runtime. The standard library provides functions like **`malloc`**, **`calloc`**, **`realloc`**, **and** **`free`** in `<stdlib.h>`.

1. malloc()

Allocates a single block of memory without initializing it.

Syntax: `void* malloc(size);`

Example:

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

int main() {
    int *ptr;
    ptr = (int *)malloc(5 * sizeof(int)); // Allocate memory for 5 integers, 5x2 =10 byte

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        ptr[i] = i + 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr); // Free the allocated memory
    return 0;
}
```

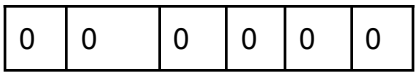
Example:
Structure Using arrow operator(->)

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

struct Student {
    int id;
    char name[50];
    float marks;
```

```
};

int main() {
    // Dynamically allocate memory for a structure
    struct Student *s1 = (struct Student *)malloc(sizeof(struct Student));

    // Assign values using the arrow operator
    s1->id = 1;
    snprintf(s1->name, sizeof(s1->name), "Alice"); // Use snprintf for strings
    s1->marks = 95.5;

    // Access and print values using the arrow operator
    printf("ID: %d\n", s1->id);
    printf("Name: %s\n", s1->name);
    printf("Marks: %.2f\n", s1->marks);

    // Free allocated memory
    free(s1);

    return 0;
}
```

2. calloc()

Allocates memory in multiple contiguous blocks(like in Array) and initializes all elements to zero.



Syntax: `void* calloc(n, size);`

Difference between malloc() and calloc()

Feature	malloc()	calloc()
Prototype	**void *malloc(size);**	**void *calloc(n, size);**

Number of Arguments	**Accepts 1 argument: the total memory size in bytes.**	**Accepts 2 arguments: the number of blocks (n) and size of each block.**
Memory Allocation	**Allocates one block of memory.**	**Allocates multiple blocks each of specified size.**
Memory Initialization	**Does not initialize memory (contains garbage values).**	**Initializes memory to 0 (all allocated bytes are set to zero).**
Usage	**Commonly used for a single block of memory.**	**Commonly used when allocating memory for arrays or multiple items.**

Example:

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

int main() {
    int *ptr;
    ptr = (int *)calloc(5, sizeof(int)); // Allocate memory for 5 integers, 5 blocks each of 2 Byte

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]); // All elements initialized to 0
    }

    free(ptr); // Free the allocated memory
    return 0;
}
```

3. realloc() – (to Reallocate Memory)

Changes the size of previously allocated memory.

Syntax: `void* realloc(void* ptr, size);`

Example:

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

int main() {
    int *ptr = (int *)malloc(3 * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 3; i++) {
        ptr[i] = i + 1;
    }

    // Reallocate memory for 5 integers
    ptr = (int *)realloc(ptr, 5 * sizeof(int));

    for (int i = 3; i < 5; i++) {
        ptr[i] = i + 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr); // Free the allocated memory
    return 0;
}
```

4. free() (to free previously allocated memory)

Releases dynamically allocated memory back to the system.

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

Important Notes:

1. Always use `free` to release memory allocated with `malloc`, `calloc`, or `realloc`.
2. Accessing freed memory leads to undefined behavior.

Programs to Practice (HW):

1. Define a structure to store employee details and display them.
2. Write a program to calculate the total and average marks of a student using a structure.
3. Create a program to store and display details of 5 books using an array of structures.
4. Implement a structure to store date (day, month, year) and write a program to calculate the difference between two dates.
5. Create a program to store details of a car (model, price, color) and display the most expensive car.
6. Define a structure for a `Student` with a nested structure for `Address` (street, city, pin).
7. Create a program to store and display details of players, including their personal details (nested structure).
8. Implement a structure for a company with nested structures for employee details and their department details.
9. Write a program to calculate the total salary of employees, where salary details are part of a nested structure.
10. Create a structure for a `Library` with a nested structure for `Book` and display all the books by a specific author.
11. Store and display details of 10 students using an array of structures.
12. Create a program to store and display details of products in a shop using an array of structures.
13. Write a program to calculate the average marks of students stored in an array of structures.
14. Implement a program to display the details of employees earning above a specific salary using an array of structures.
15. Store and sort details of movies (title, year, rating) using an array of structures.
16. Create a union to store different types of data (int, float, char) and demonstrate memory sharing.
17. Write a program to store and display data of a variable length (e.g., an integer, a float, or a string) using a union.
18. Compare and demonstrate memory usage in structures and unions.

19. Implement a union to represent a record that can store either a student's marks or attendance.
20. Write a program to use a union to store and display a value as both integer and floating-point number.
21. Allocate memory for an integer array using `malloc` and input values dynamically.
22. Write a program to create a dynamically allocated 2D array using `malloc`.
23. Use `calloc` to allocate memory for an array of integers and initialize all elements to 0.
24. Demonstrate the use of `realloc` to resize a dynamically allocated array.
25. Implement a program to create a linked list using `malloc` for dynamic memory allocation.
26. Create a program to dynamically allocate memory for a string and copy a user-provided string into it.
27. Write a program to allocate memory for `n` students using `calloc` and input their details.
28. Demonstrate freeing allocated memory using `free` after a program finishes execution.
29. Implement a program to find the sum of an array of numbers entered by the user, using dynamic memory allocation.
30. Dynamically allocate memory for a structure (e.g., Student) and input details.
31. Create a structure to store employee details and dynamically allocate memory for an array of employees.
32. Write a program to store book details using an array of structures, allocating memory dynamically.
33. Implement a nested structure for storing details of students and dynamically allocate memory for `n` students.
34. Use a union and dynamically allocate memory to store either an integer or a float, based on user input.
35. Write a program to store employee details and calculate the average salary, using both structures and `malloc`.
36. Demonstrate the use of `calloc` to initialize an array of structures with default values.
37. Create a program to input and display an array of employee details, resizing it using `realloc`.
38. Write a program to store and manipulate date details using dynamic memory allocation and structures.
39. Use a structure with dynamically allocated memory to store strings of variable lengths.
40. Create a dynamic array of unions to store mixed data types and display their contents.